Programming 2 - SS23

Project 3 - SAT Solver

Presentors: Lisa-Marie Rolli, Michael Schott, Lea Herrmann
German slides by: Niklas Metzger, Jan Baumeister, Philip Lukert

23. Mai 2023

University of Saarland

Overview

1. Input
2. Tseitin
3. DPLL
4. Technical notes

$\downarrow$ Input

boolean formula

$\downarrow$ Tseitin

CNF

$\downarrow$ DPLL

Satisfiable?

Input

What are Boolean Formulas?

`!a`	a must **not** apply
`a && b`	a **and** b must apply
`a || b`	a **or** b must apply
`a => b`	b must apply, **if** a applys
`a <=> b`	a and b must have the **same** truth value

example

```
(a || b) => !a
```

What are Boolean Formulas?

`!a`	a must **not** apply
`a && b`	a **and** b must apply
`a || b`	a **or** b must apply
`a => b`	b must apply, **if** a applys
`a <=> b`	a and b must have the **same** truth value

example

`(a || b) => !a`

a	b	(a || b) => !a
1	1	0
1	0	0
0	1	1
0	0	1

SAT = Satisfiability

Is there a variable assignment that satisfies the formula?

```
(a || b) => !a
```

SAT = Satisfiability

Is there a variable assignment that satisfies the formula?

`(a || b) => !a`

a	b	(a || b) => !a
1	1	0
1	0	0
0	1	1
0	0	1

Yes e.g. a=0, b=1.

The problem is NP-hard.

Conjunctive Normal Form CNF

$$\begin{aligned} formula &= clause \,\&\&\, clause \,\&\&\, \cdots \,\&\&\, clause \\ clause &= (literal \,||\, literal \,||\, \cdots \,||\, literal) \\ literal &= variable \mid !variable \\ variable &= [a - z\ A - Z\ 0 - 9]+ \end{aligned}$$

examples

`(a || b || !c) && (c || !d)`

`a && b && !c`

Reverse Polish Notation

$$
\begin{aligned}
\mathit{formula} &= \mathit{variable} \mid \mathit{formula}\ \mathit{unaryop} \mid \mathit{formula}\ \mathit{formula}\ \mathit{binaryop} \\
\mathit{unaryop} &= ! \\
\mathit{binaryop} &= \&\& \mid \ || \ \mid\ => \ \mid\ <=> \\
\mathit{variable} &= [a-z\ A-Z\ 0-9]+
\end{aligned}
$$

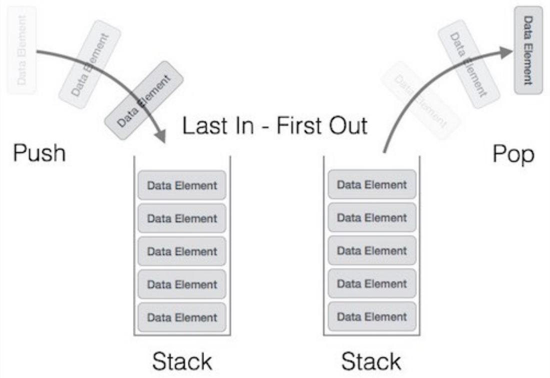
example

`a b ||` `a !` `=>`

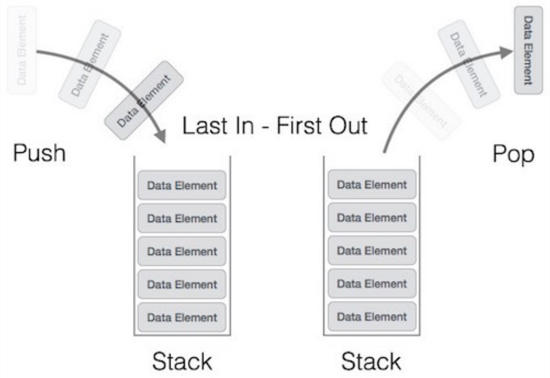
$\downarrow$

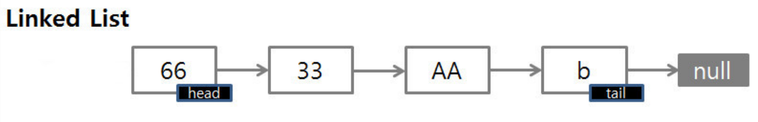
`(a || b)` `=>` `!a`

What is a Stack?



What is a Stack?





www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
www.dreamscoder.com/viewprogram.php?id=111

parseFormula()

`a b || a ! =>`

`<empty>`

parseFormula()

```
a b || a ! =>
```

```
a
```

parseFormula()

`a b || a ! =>`

b
a

parseFormula()

`a b || a ! =>`

`a || b`

parseFormula()

`a b || a ! =>`

`a`
`a || b`

parseFormula()

`a b || a ! =>`

`!a`
`a || b`

parseFormula()

`a b || a ! =>`

```
(a || b) =>
!a
```

Project - Structs

```c
typedef struct PropFormula {
    FormulaKind kind;

    union {
        VarIndex var;
        struct PropFormula* single_op;
        struct PropFormula* (operands[2]);
    } data;
} PropFormula;
```

```c
typedef enum FormulaKind {
    VAR,

    AND,
    OR,
    IMPLIES,
    EQUIV,

    NOT,
} FormulaKind;
```

```c
mkVarFormula(VarTable* vt, char* name)
```

```c
mkBinaryFormula(FormulaKind kind, PropFormula* left_op,
                                  PropFormula* right_op)
```

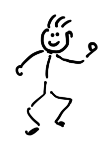
```c
mkUnaryFormula(FormulaKind kind, PropFormula* operand)
```

```c
FormulaKind toKind(const char* str)
```

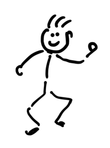
```c
parseFormula(FILE* input, VarTable* vt)
```

Tseytin

Introduction

 $\xrightarrow{\text{Read in}}$ **boolean formular**

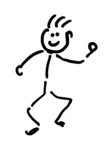
Introduction



$\xrightarrow{\text{Read in}}$ **boolean formular**

CNF $\xrightarrow{\text{DPLL}}$ **Satisfiable?**

Introduction

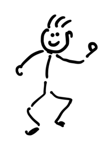


$\xrightarrow{\text{Read in}}$ **boolean formular**

De Morgan's Laws $\rightarrow$ exponential increase

CNF $\xrightarrow{\text{DPLL}}$ **Satisfiable?**

Introduction

 $\xrightarrow{\text{Read in}}$ **boolean formular**

De Morgan's Laws $\rightarrow$ exponential increase

Tseitin

CNF $\xrightarrow{\text{DPLL}}$ **Satisfiable?**

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

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

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

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

$\quad \neg b \Rightarrow c$

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

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

$\quad \neg b \Rightarrow c$

$\quad\quad \neg b$

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

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

$\quad \neg b \Rightarrow c$

$\quad\quad \neg b \qquad \rightarrow \qquad x_1 \Leftrightarrow \neg b$

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

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

$$\begin{array}{rcl} \neg b \Rightarrow c & \rightarrow & x_2 \Leftrightarrow (x_1 \Rightarrow c) \\ \neg b & \rightarrow & x_1 \Leftrightarrow \neg b \end{array}$$

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

$$
\begin{array}{ccc}
a \wedge (\neg b \Rightarrow c) & \rightarrow & x_3 \Leftrightarrow (a \wedge x_2) \\
\neg b \Rightarrow c & \rightarrow & x_2 \Leftrightarrow (x_1 \Rightarrow c) \\
\neg b & \rightarrow & x_1 \Leftrightarrow \neg b
\end{array}
$$

General Idea

- avoids exponential increase by adding additional variables
- each subformula is represented by a new variable

Example

$$
\begin{array}{ccc}
a \wedge (\neg b \Rightarrow c) & \rightarrow & x_3 \Leftrightarrow (a \wedge x_2) \\
\neg b \Rightarrow c & \rightarrow & x_2 \Leftrightarrow (x_1 \Rightarrow c) \\
\neg b & \rightarrow & x_1 \Leftrightarrow \neg b
\end{array}
$$

$$(x_1 \Leftrightarrow \neg b) \wedge (x_2 \Leftrightarrow (x_1 \Rightarrow c)) \wedge (x_3 \Leftrightarrow (a \wedge x_2)) \wedge x_3$$

Transformation of ¬

$$
\begin{aligned}
& x_1 \Leftrightarrow \neg a \\
\equiv & (x_1 \Rightarrow \neg a) \wedge (\neg a \Rightarrow x_1) \\
\equiv & (\neg x_1 \vee \neg a) \wedge (a \vee x_1)
\end{aligned}
$$

Transformation of $\wedge$

$$
\begin{aligned}
& x_1 \Leftrightarrow (a \wedge b) \\
\equiv & (x_1 \Rightarrow (a \wedge b)) \wedge ((a \wedge b) \Rightarrow x_1) \\
\equiv & (\neg x_1 \vee (a \wedge b)) \wedge (\neg(a \wedge b) \vee x_1) \\
\equiv & (\neg x_1 \vee (a \wedge b)) \wedge (\neg(a \wedge b) \vee x_1) \\
\equiv & (\neg x_1 \vee a) \wedge (\neg x_1 \vee b) \wedge (\neg a \vee \neg b \vee x_1)
\end{aligned}
$$

Transformation of $\vee$

$$
\begin{aligned}
& x_1 \Leftrightarrow (a \vee b) \\
\equiv & (\neg x_1 \vee a \vee b) \wedge (\neg a \vee x_1) \wedge (\neg b \vee x_1)
\end{aligned}
$$

Transformation of $\Rightarrow$

$$
\begin{aligned}
& x_1 \Leftrightarrow (a \Rightarrow b) \\
\equiv & (\neg x_1 \vee \neg a \vee b) \wedge (a \vee x_1) \wedge (\neg b \vee x_1)
\end{aligned}
$$

Transformation of $\Leftrightarrow$

$$
\begin{aligned}
& x_1 \Leftrightarrow (a \Leftrightarrow b) \\
\equiv & (\neg x_1 \vee \neg a \vee b) \wedge (\neg x_1 \vee \neg b \vee a) \wedge (x_1 \vee \neg a \vee \neg b) \wedge (x_1 \vee a \vee b)
\end{aligned}
$$

Example

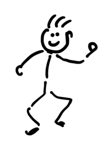
$$
\begin{aligned}
& a \wedge (\neg b \Rightarrow a) \\
\equiv & (x_1 \Leftrightarrow \neg b) \wedge (x_2 \Leftrightarrow (x_1 \Rightarrow c)) \wedge (x_3 \Leftrightarrow (a \wedge x_2)) \wedge x_3 \\
\equiv & (\neg x_1 \vee \neg b) \wedge (b \vee x_1) \\
\wedge & \ (\neg x_2 \vee \neg x_1 \vee c) \wedge (x_1 \vee x_2) \wedge (\neg c \vee x_2) \\
\wedge & \ (\neg x_3 \vee a) \wedge (\neg x_3 \vee x_2) \wedge (\neg a \vee \neg x_2 \vee x_3) \\
\wedge & \ x_3
\end{aligned}
$$

In the Project

addClauses()

DPLL

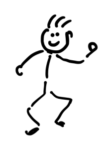
DPLL - What is it?

 $\xrightarrow{\text{Read input}}$ **boolean forumla**

$\Big\downarrow$ Tseytin

CNF $\xrightarrow{\text{DPLL}}$ **Satisfiable?**

DPLL - What is it?



Read input

boolean forumla

Tseytin

CNF

DPLL

Satisfiable?

Terminology

- Termination:
  The algorithm stops.
- Backtracking:
  The last iterations of the algorithm are undone until a variable is found, which can be assigned a new value.

Terminology

- Unit Clause
  A unit clause is a clause in which all literals except one are already set.

Example

Under this assignment both of these clauses are unit clauses:

$$(a \vee b \vee v) \wedge (d), where\ a = false, b = false$$

- Unit-Propagation:
  The next unit clause is satisfied and the algorithm is continued.

Algorithm

Algorithm: DPLL

```
while not terminated do
    if all clauses are fulfilled then
        abort(sat)
    end
    if one clause is false then
        if reset possible then
            reset
            end iteration
        else
            abort(unsat)
        end
    end
    if unit clause exists then
        fulfill any unit clause
        end iteration
    end
    select next free variable and set to true
end
```

Example DPLL

Example

Consider the following example:

$$(A \vee B \vee C) \wedge (\neg A \vee B) \wedge (\neg C)$$

1. C is a unit clause $\rightarrow$ set C to *false*
2. No unit clause left, choose A $\rightarrow$ set A to *true*
3. $(\neg A \vee B)$ is a unit clause $\rightarrow$ set B to *true*

Backtracking Implementation

- Use the stack from part 1 to track assignments
- Store both the variable and a `Reason`
- A reason can be `CHOOSEN` or `IMPLIED`
  - `CHOOSEN`: value was freely chosen $\rightarrow$ alternative value possible
  - `IMPLIED`: no alternative value possible
- When backtracking, the stack is popped until a `CHOOSEN` variable is found

$\rightarrow$ set to implied value

Backtracking Pseudocode

Algorithm: Backtracking

```
while ! isEmpty(Stack) do
    temp = peek(Stack)
    if temp->reason == CHOOSEN then
        updateVariable(temp)
        return
    else
        updateVariable(temp)
        pop(Stack)
    end
end
```

DPLL Example with Backtracking

Example

In the following example backtracking is required:

$$(B \vee \neg C) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (C \vee \neg D) \wedge (A)$$

Stack before and after backtracking:

$$\begin{array}{lcccr} \text{D IMPLIED} & & & & \text{D IMPLIED} \\ \text{C IMPLIED} & \textit{Backtracking} & & \textit{continue DPLL} & \text{C IMPLIED} \\ \text{B CHOSEN} & \longrightarrow & \text{B IMPLIED} & \rightarrow & \text{B IMPLIED} \\ \text{A IMPLIED} & & \text{A IMPLIED} & & \text{A IMPLIED} \end{array}$$

Implementation details

- Relevant files:
  - `dpll.c`
  - `dpll.h`
- Implement the function `iterate()` in `dpll.c`
- The struct `Assignment` provides a data structure for a variable and a reason

Attention!

Many functions are already implemented (e.g.`pushAssignement`). Understand and use them!

Technical notes

Questions?