Boolean Satisfiability

Announcements

- Lab5 was due Monday
- Lab6 due sunday

- Groups for the project sent out

Overview

1. Boolean Satisfiability

2. DPLL Algorithm

Boolean Satisfiability

Boolean logic review

A boolean expression is built from:

1. Variables (can evaluate to TRUE/FALSE)
2. Operators
    a. AND $\wedge$
    b. OR V
    c. NEGATION ~ / ! / ¬

A formula is said to be *satisfiable* if it can be made TRUE by assigning logical values to its variables.

Boolean Satisfiability Problem

Asks whether there exists an *interpretation* that *satisfies* a given Boolean formula

- UNSAT: `p ∧ !p`
- `SAT:` `p ∧ !q`

`Interpretation: p -> TRUE, q -> FALSE`

Examples:

SAT or UNSAT? If SAT, give the interpretation

1. $(A \lor B \lor C) \land (\neg A \lor \neg B \lor C) \land (A \lor \neg B \lor \neg C) \land (\neg A \lor B \lor \neg C) \land (A \lor B \lor \neg C)$
2. $(A \lor B \lor \neg C) \land (\neg A \lor C \lor D) \land (B \lor \neg D \lor C) \land (\neg B \lor \neg C \lor A)$

What does this have to do with software analysis?

```
if (y > 3 * x + 7) {
    if (x + y - z % 2 == 0) {
        if (z >= x) {
            int res = z / (z - x);
        }
    }
}
```

When will this program crash?

`z - x == 0` **AND**
`z >= x` **AND**
`x + y -z % 2 == 0` **AND**
`y > 3 * x + 7`

We can encode path constraints as boolean satisfiability problems!

Solving Boolean Satisfiability

Can we write a program to answer SAT / UNSAT?

SAT is the first problem that was proven to be NP-Complete

We can try every possible combination, but it will be slow.

Conjunctive Normal Form (CNF)

Each *clause* is separated by an AND operator

`C1 ∧ ... ∧ Cn`

And each *clause* and is of the form:

`Li  V ... V Lm`

where each Li is called a literal and is either a boolean variable or a negation of the variable

Conjunctive Normal Form (CNF)

Example 6.A The following is a CNF formula with two clauses, each of which contains two literals:

$$(p \vee \neg r) \wedge (\neg p \vee q)$$

The following formula is *not* in CNF:

$$(p \wedge q) \vee (\neg r)$$

■

DPLL Algorithm

Given a boolean formula in CNF form, DPLL decides UNSAT/SAT and gives an interpretation if its SAT

Alternates between two phases:

1. Deduction
   a. Tries to simplify the formula using the laws of logic
2. Search
   a. Searches for an interpretation

DPLL - Deduction Phase

Boolean Constant Propagation

$$(\ell) \wedge C_2 \wedge \cdots C_n$$

Suppose the first clause consists of a single literal (We call this a *unit clause*).

Any interpretation of the formula must assign ℓ to TRUE.

Simplify the formula by substituting TRUE for all ℓs

Boolean Constant Propagation (BCP) Example

$$(p) \wedge (\neg p \vee r) \wedge (\neg r \vee q)$$

p must be TRUE

$$\begin{aligned} & (\text{true}) \wedge (\neg \text{true} \vee r) \wedge (\neg r \vee q) \\ \equiv\ & (r) \wedge (\neg r \vee q) \end{aligned}$$

r must be TRUE

$$\begin{aligned} & (\text{true}) \wedge (\neg \text{true} \vee q) \\ \equiv\ & (q) \end{aligned}$$

$$\{p \mapsto \text{true}, q \mapsto \text{true}, r \mapsto \text{true}\}$$

Boolean Constant Propagation (BCP) Example 2

$(x) \land (p \lor r) \land (\neg p \lor q) \land (\neg q \lor \neg r)$

Deduction + Search

Once the simplified formula no longer contains unit clauses, we have to go back to the brute force approach....

Iteratively chooses variables and tries to replace them with TRUE or FALSE calling DPLL recursively in the resulting formula

Deduction + Search Example

$(x) \land (p \lor r) \land (\neg p \lor q) \land (\neg q \lor \neg r)$

After BCP:

$(p \lor r) \land (\neg p \lor q) \land (\neg q \lor \neg r)$

First level of recursion: p -> TRUE

$(q) \land (\neg q \lor \neg r)$

After BCP: q -> TRUE

$\neg r$

BCP again! r-> FALSE

x -> TRUE

p -> TRUE

q -> TRUE

r -> FALSE

DPLL

Algorithm 1: DPLL

Data: A formula F in CNF form
Result: $I \models F$ or UNSAT

▷ Boolean constant propagation (BCP)
while *there is a unit clause* (ℓ) *in* F **do**
  Let F be $F[\ell \mapsto \text{true}]$

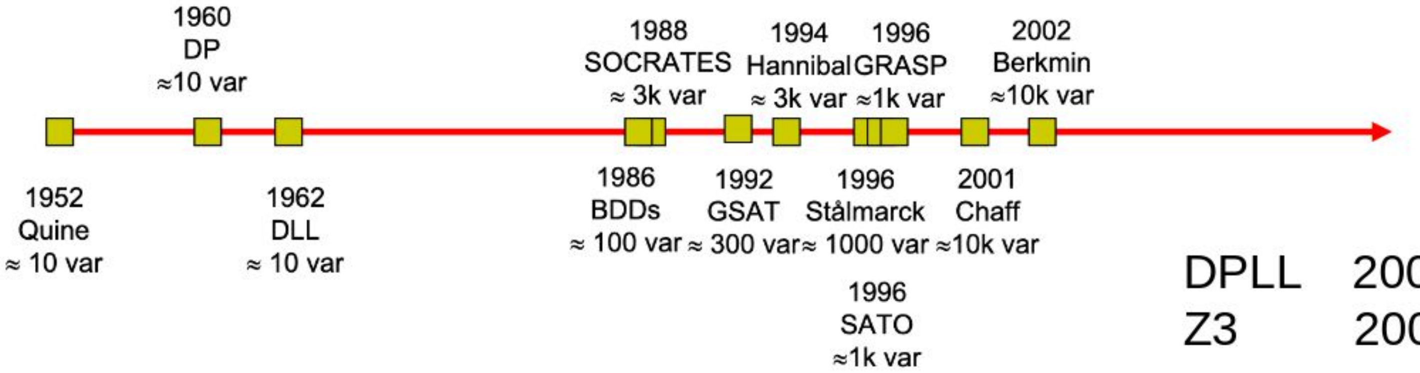
if F *is* true **then return** SAT

▷ Search
for *every variable* p *in* F **do**
  If DPLL($F[p \mapsto \text{true}]$) is SAT **then return** SAT
  If DPLL($F[p \mapsto \text{false}]$) is SAT **then return** SAT
return UNSAT
▷ The model I that is returned by DPLL when the input is SAT is maintained implicitly in the sequence of assignments to variables (of the form $[l \mapsto \cdot]$ and $[p \mapsto \cdot]$)

Activity

Trace DPLL on two boolean formulae

SAT Solvers Timeline



DPLL 2004
Z3 2008

We went from handling ~10 variables to 1M+ variables....

Annual SAT competition: https://satcompetition.github.io/

Satisfiability Modulo Theories (SMT)

Remember our example?

```
if (y > 3 * x + 7) {

    if (x + y - z % 2 == 0) {

        if (z >= x) {

            int res = z / (z - x);

        }

    }

}
```

When will this program crash?

`z - x == 0` **AND**
`z >= x` **AND**
`x + y -z % 2 == 0` **AND**
`y > 3 * x + 7`

We can encode path constraints as boolean satisfiability problems!

Satisfiability Modulo Theories (SMT)

A generalization of SAT

SAT only handles theory of Booleans

SMT extends it to handle integers, reals, arrays, strings, bit-vectors, etc.

Boolean Abstraction

We can turn a formula with *linear real arithmetic* into a boolean formula using **boolean abstraction**

$$F \triangleq (x \leqslant 0 \vee x \leqslant 10) \wedge (\neg x \leqslant 0)$$

$$F^B \triangleq (p \vee q) \wedge (\neg p)$$

Every unique linear inequality is replaced with a boolean variable

The boolean abstraction of F is denoted F^B

We use superscript T to map boolean formulae back to their theory formulae: $(F^B)^T = F$

Boolean Abstraction

If F^B is UNSAT, then F is UNSAT.

Example: $F \triangleq (x \leqslant 0 \wedge (\neg x \leqslant 0)$

$$F^B \triangleq (p) \wedge (\neg p)$$

Boolean Abstraction

If F^B is SAT, then F is not necessarily SAT!

$$F \triangleq x \leqslant 0 \wedge x \geqslant 10.$$

$F^B = \quad p \wedge q$

During abstraction, relations between the inequalities are lost. X cannot be <= 0 and >= 10, but p and q have no relation!

SMT Solvers

To solve SMT, we can slightly modify DPLL to $DPLL^T$

- Start by treating the formula as if its completely boolean, then incrementally add more and more *theory* information until we can conclusively say it is SAT or UNSAT
- Requires a boolean abstraction of F
- Requires access to a *theory solver*

DPLL^T

First checks if F^B is UNSAT. Since we know that F will also be UNSAT

Algorithm 2: DPLL^T

Data: A formula F in CNF form over theory T
Result: $I \models F$ or UNSAT

Let F^B be the abstraction of F
while true **do**
  If DPLL(F^B) is UNSAT **then return** UNSAT
  Let I be the model returned by DPLL(F^B)
  Assume I is represented as a formula
  if I^T *is satisfiable (using a theory solver)* **then**
    return SAT and the model returned by theory solver
  else
    Let F^B be $F^B \wedge \neg I$

Queries a theory solver on the *concrete* output of DPLL. (I is an interpretation)

If SAT, we're done!
IF UNSAT, doesn't necessarily mean it's UNSAT... negate and continue

DPLL^T Example

F: $x \geqslant 10 \wedge (x < 0 \vee y \geqslant 0)$

What is F^B? $p \wedge (q \vee r)$

1. **First iteration of DPLL on F^B**

Returns SAT: { p -> T, q -> T }
I = p ∧ q

2. **Query Theory Solver on I^T**

What is **I**^{T}? $\underbrace{x \geqslant 10}_{p} \wedge \underbrace{x < 0}_{q}$

Theory solver returns **UNSAT**

3. **Rerun DPLL on** F^B ∧ ! I

What is F^B ∧ ! I ? $p \wedge (q \vee r) \wedge \underbrace{(\neg p \vee \neg q)}_{\neg I_1}$

Returns SAT: { p -> T, q -> F, r -> T}

4. **Query Theory Solver on I^T**

What is **I**^{T}?

I^T = (x>= 10) ∧ (x>=0) ∧ (y >= 0)

Theory solver returns { x->10, y-> 0 }

Summary

- Boolean Satisfiability
  - Path conditions can be modelled as SAT formulae
  - Solved with DPLL
  - We can use this to find bugs!

- SMT
  - Satisfiability Modulo Theories, where **theory** refers to a logical theory that describes a particular domain
  - We will learn this for Linear Arithmetic
  - There are also theories for arrays, bit-vectors, and equality

- Next Class: we will learn the Simplex Algorithm for solving a linear equation
  - This is our theory solver!