Boolean Satisfiability

Introduction

Satisfiability (SAT): can we satisfy a boolean formula

- Is NP hard
- Can be reduced to many other hard problems

Outline

- SAT Problem
- Conjunctive Normal Form
  - N-ary Operators
  - Definitions
  - Conversions
- Davis-Putnam-Logemann-Loveland

SAT Problem

Input: A boolean formula:

- Variables $\mathrm{P} = \mathrm{p}_1 \ldots \mathrm{p}_\mathrm{n}$
- Formula ϕ: $\mathbb{B}^\mathrm{n} \mapsto \mathbb{B}$

Output: Is $\phi(\mathrm{P})$ satisfiable?

- $\exists \mathrm{P}, (\phi(\mathrm{P}) = \top)$
- What is P?

Example:

- Input:
  - $\mathrm{P} \equiv \{\mathrm{a}, \mathrm{b}\}$
  - $\phi \equiv \mathrm{a} \implies (\bot \vee \mathrm{b})$
- Output:
  - SAT, $\mathrm{a} = \bot$

Conjunctive Normal Form

N-ary Boolean

AND: $\alpha \wedge \beta = (\texttt{AND}\ \alpha\ \beta)$

OR: $\alpha \vee \beta = (\texttt{OR}\ \alpha\ \beta)$

Identity Element

Identity Element: a special element of a set for which a binary operation on that set leaves any element unchanged

$f(\alpha, \chi) = \alpha$

Arithmetic Example:

$a * \chi = a$, if $\chi = 1$

Boolean Example:

$a \wedge \chi = a$, if $\chi = \top$

Cancellation and the Identity Element

If we have variables that are the identity element (i.e. variable + identity = variable) we can just remove the variable that are the identity elements

Normal Forms

Normal Form: a standard or conventional way of writing a mathematical object

In rewrite systems: an object that cannot be further rewritten

*Often useful to define algorithms in terms of some normal form

Definitions

Literal: a single variable or its negation

- positive literal: p
- negative literal: $\neg p$

Conjunction: An n-ary AND. True when **all** of its arguments are true. Examples:

- $p_i \wedge p_j$
- $p_i \wedge (p_j \vee p_k)$

Disjunctions: An n-ary OR. True when **any** of its arguments are true. Examples:

- $p_i \vee p_j$
- $p_i \vee (p_j \wedge p_k)$

Conjunctive Normal Form (CNF)

A conjunction of disjunctions of literals

(**S-expression representation**):

(and (or $l_{0,0}$, $l_{0,1}$ ...)

(or $l_{1,0}$, $l_{1,1}$ ...)

...

(or $l_{n,0}$, $l_{n,1}$ ...))

where each $l_{i,j}$ is a literal, that is one of p or (NOT p)

(**Infix representation**):

$(p_i \vee p_j) \wedge (\neg p_i \vee p_k)$

Conversion to CNF

1. Eliminate $\iff$

   $(\alpha \iff \beta) \rightsquigarrow ((\alpha \implies \beta) \wedge (\beta \implies \alpha))$

2. Eliminate $\implies$

   $(\alpha \implies \beta) \rightsquigarrow (\neg\alpha \vee \beta)$

3. Eliminate $\oplus$

   $(\alpha \oplus \beta) \rightsquigarrow ((\alpha \vee \beta) \wedge \neg(\alpha \wedge \beta))$

4. Move in $\neg$

   Double negation: $(\neg(\neg\alpha)) \rightsquigarrow (\alpha)$

   De Morgan's Law: $(\neg(\alpha \wedge \beta)) \rightsquigarrow ((\neg\alpha \vee \neg\beta))$

   De Morgan's Law: $(\neg(\alpha \vee \beta)) \rightsquigarrow ((\neg\alpha \wedge \neg\beta))$

   At this point, this is called Negation Normal Form (NNF)

5. Distribute $\vee$ over $\wedge$: $(\alpha\vee\ (\beta \wedge \gamma)) \rightsquigarrow ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$

   $(\alpha\vee\ (\beta \wedge \gamma)) \rightsquigarrow ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

David-Putnam-Logemann-Loveland (DPLL)

DPLL Outline: For ϕ in conjunctive normal form:

Base Case:

1. If ϕ has all true clauses ($\forall$), return true
2. If ϕ has any false clauses ($\exists$), return false

Recursive Case:

1. Propagate values from unit (single-variable) clauses.
2. Choose a branching variable v
3. Branch (recurse) for v = 1 or v = 0

Unit Clauses

A disjunction in the conjunctive normal form that contains only a single, positive, or negative literal.

- Example: a, ¬ba

Unit propagation example:

Given (a ∧ (a ∨ b) ∧ (¬b ∨ c))

if a = 1, we can convert the formula into (¬b ∨ c)

Unit Propagation Algorithm

```
Procedure unit-propagate(φ)
if φ has some unit clause with variable v then
    φ' ← bind v in φ to make unit clause true;
    return unit-propagate(φ);
else
    return φ;
```

Steps:

1. Find a unit clause
2. Propagate assignment
3. Recurse

DPLL Algorithm

```
Procedure DPLL(φ)
let φ ← unit-propagate(φ) in
    if φ = true then // SAT
        return true;
    else if φ = false then // UNSAT
        return false;
    else // Recursive case
        let v ← choose-variable(φ) in
            if DPLL(φ' ∧ v) then // Try v = ⊤
                return true;
            else // Try v = ⊥
                return DPLL(φ' ∧ ¬v);
```

Steps:

1. Propagate unit clauses
2. If formula is true or false, return
3. Pick some variable v:
   3.1. Try Recursing with true v
   3.2. If unsat, try recursing with false

DPLL Example

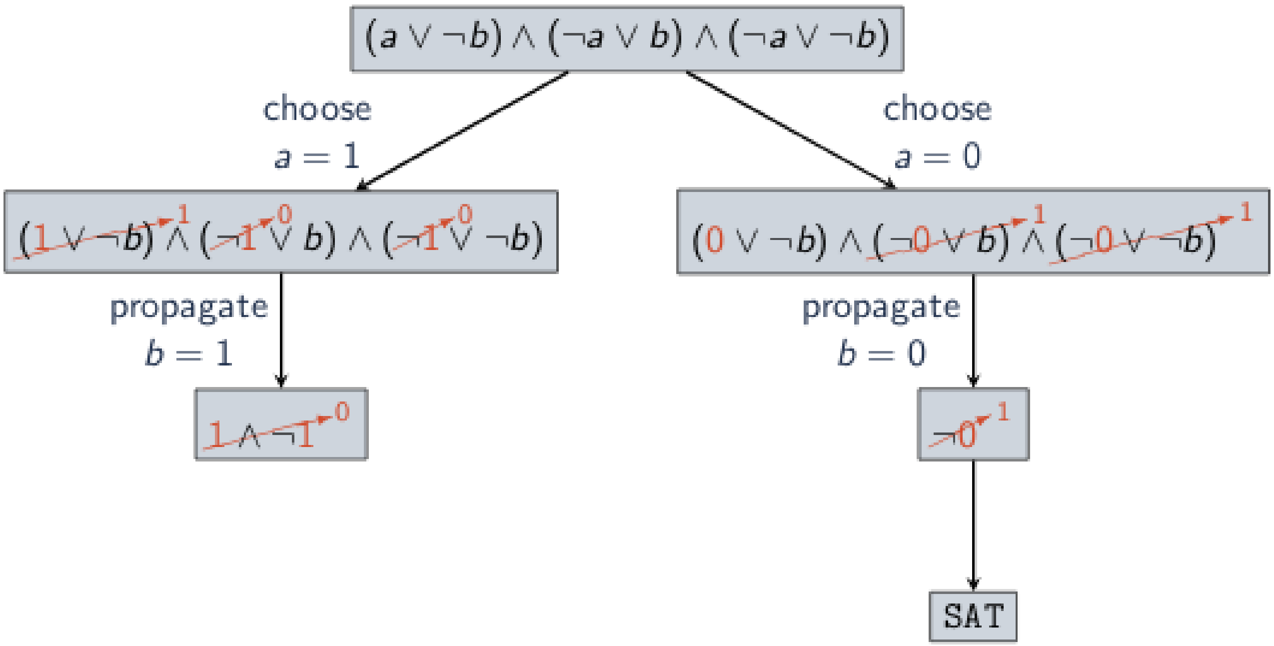

Figure 1: Example of DPLL algorithm in action