Algorithms for Computational Logic

Introduction

Emmanuel Hebrard (adapted from João Marques Silva, Inês Lynce and Vasco Manquinho)

Outline

1 Introduction to Boolean Satisfaction
2 Boolean Reasoning
Propositional Logic

A *proposition* is an assertion that can be:

- assigned a truth value (true or false)
- written using *atomic propositions* (or *atoms*) and *logic connectors*

An atom is a proposition written using a unique symbol.

- Atomic propositions:
  - “Adam follows the lecture”, “Adam works at home”, “Adam cheats at the exam”, “Adam passes the exam”

- Propositions:
  - “if Adam does not listen the lecture and does not work at home then he will not pass the exam unless he cheats”
Propositional Logic

Formulae (syntax)

A non-atomic proposition (Formula) $\phi$ is either:
- an atom
- the negation $\neg \psi$ of another proposition $\psi$
- the concatenation of two or more propositions $\phi_1$ and $\phi_2$ by a logical connector \{\&, \lor, \rightarrow, \ldots\}

\[
((\text{"listen lecture"} \land \text{"work at home"}) \lor \text{"cheat"} \lor \neg \text{"pass exam"}) \land
(\neg \text{"cheat"} \land \text{"pass exam"}) \lor \neg \text{"get diploma"})
\]

Models

Models (interpretations)

A model $A$ is a mapping from atoms in $\mathcal{X}$ to \{true, false\}. We write $A \models x$ for “Atom $x$ is true in model $A$”

A proposition $\phi$ written using atoms in $\mathcal{X}$ can be interpreted (given a truth value) using a model $A$ on $\mathcal{X}$:
- if $\phi$ is the negation of a proposition $\psi$, then $A \models \phi$ if and only if $A \not\models \psi$
- if $\phi$ is a conjunction $\phi_1 \land \phi_2$, then $A \models \phi$ if and only if $A \models \phi_1$ and $A \models \phi_2$
- if $\phi$ is a disjunction $\phi_1 \lor \phi_2$, then $A \models \phi$ if and only if $A \models \phi_1$ or $A \models \phi_2$

Ex: “listen lecture” $\land$ “work at home” $\land \neg$“cheat” $\land \neg$“pass exam” $\land \neg$“get diploma”
Introduction to Boolean Satisfaction

The Satisfiability Problem

SAT

- **data**: A Boolean formula $\phi$
- **question**: Does there exist an interpretation that satisfies $\phi$?

- A formula is *satisfiable* iff there exists an interpretation that satisfies it
- A formula $\varphi$ is *unsatisfiable* iff there is no interpretation that satisfies it
  - Write it UNSAT ($\varphi$)
- A formula is *valid* / *tautology* iff all interpretations satisfy it
  - Equivalent to UNSAT ($\neg \varphi$)
- A formula $\psi$ is an *implicate* of $\varphi$ iff all interpretations satisfying $\varphi$ also satisfy $\psi$
  - Equivalent to UNSAT ($\varphi \land \neg \psi$)
- A formula $\psi$ is an *implicant* of $\varphi$ iff $\varphi$ is an *implicate* of $\psi$
Examples of Applications

- Linux package upgrade
  - The Eclipse foundation uses Daniel le Berre’s SAT solver SAT4j to solve this problem
  - Equinox/p2/CUDFResolver

- (Re-)Attribution of the TV radiospectrum by the Federal Communications Commission (FCC) in 2017
  - The radiofrequency allocation problem corresponds to Graph Coloring
    - Vertices are broadcasters, colors are frequencies
    - Easy to encode as SAT
  - Reverse auction: the FCC buys frequencies and starts with high quotes that decrease at each round
    - Stops when it is not possible to assign frequencies to broadcasters who opted out
  - Critical to prove unsatisfiability (the auction yielded $20 billion)

Cook-Levin Theorem

- SAT is in NP, the interpretation $\sigma$ that satisfies it is a polynomial certificate

Théorème de Cook-Levin

- SAT is NP-complete
  - At least as hard as any problem in NP
  - If SAT is in $P$ then $P = NP$
Fragments of SAT are particular case defined by the language

- Using only negation (\(\neg\)), disjunction (\(\lor\)) and conjunction (\(\land\)) is not restrictive

Disjunctive Normal Form

- Disjunctive normal form:
  - Disjunction of conjunctions (sum) of literals (products)
  - Ex: \((\neg a \land b \land c) \lor (\neg b \land \neg c) \land (a \land \neg b)\)

- Every product is an implicant, and corresponds to an interpretation

- Satisfiability of a DNF is easy
Conjunctive Normal Form

- Conjunctive normal form:
  - Conjunction of disjunctions of literals (clauses)
  - Ex: \((\neg a \vee b \vee c) \land (\neg b \vee \neg c) \land (a \vee \neg b)\)

- For any formula \(\varphi\), there is a CNF formula \(\varphi'\) such that
  - \(\text{SAT}(\varphi) \iff \text{SAT}(\varphi')\)
  - \(|\varphi'| \in O(|\varphi|^{c})\) for some constant \(c\)

- Every clause is an implicate

- Validity of a CNF is easy

Horn Clauses

- Horn clause:
  - Clause with at most one positive literal
  - Ex: \((\neg a \mathbin{\lor} \neg c \mathbin{\lor} b) \land (\neg b \mathbin{\lor} \neg c) \land (\neg b \mathbin{\lor} a)\)
  - Equivalent to implications
    - \((a \land c \Rightarrow b) \land (b \land c \Rightarrow \text{false}) \land (b \Rightarrow a)\)
The DIMACS format

- **Comments**
  - c This line is a comment.
  - p cnf 5 7
  - -1 3 -5 4 0
  - 2 -3 0
  - 1 5 0
  - -3 -4 0
  - -1 2 4 0
  - -2 0
  - 2 -3 -5 0

- **Basic definitions**

  - **Typename/classes**
    - **Variable**: used for indexing → e.g., int from 0 to $n - 1$
    - **Literal**: used for indexing → e.g., int from 0 to $2n - 1$
    - **TruthValue**: three possibility (true, false, undef) → {1, 0, -1}
    - **Clause**: iterable list of literals

  - **Functions on variables**
    - $pos(Variable:x) \rightarrow Literal x$
      (e.g., 2x + 1)
    - $neg(Variable:x) \rightarrow Literal \neg x$
      (e.g., 2x)

  - **Functions on literals**
    - $sign(Literal:l) \rightarrow \{false, true\}$
      (e.g., $l \% 2$)
    - $not(Literal:l) \rightarrow \neg l$
      (e.g., $l \wedge 1$)
    - $var(Literal:l) \rightarrow x$
      (e.g., $l / 2$)
Data structures

- **model** \([\text{Variable} : x] \mapsto \text{TruthValue}\)
  - stores the current truth value of \(x\)
- **clauses** \([\text{Literal} : l] \mapsto [\text{Clause}, \ldots]\)
  - list of clauses containing literal \(l\)
- **unit-literals**
  - stack of true literals (efficient push\((\text{Literal}:l)\) and back\((\text{Literal})\) and pop-back\((\text{Literal})\))

Functions

- **val** \((\text{Variable}:x) \mapsto \text{TruthValue}\)
  - truth value of variable \(x\)
- **falsified** \((\text{Literal}:l) \mapsto \text{Boolean}\)
  - literal is falsified in model
- **satisfied** \((\text{Literal}:l) \mapsto \text{Boolean}\)
  - literal is satisfied in model

IN/OUT

- Functions from-dimacs\((\text{int}:d) \mapsto \text{Literal}\) and to-dimacs\((\text{Literal}:l) \mapsto \text{int}\)
- Functions read-dimacs() and write-dimacs()
Unit Propagation

- A clause forbids exactly one tuple
  \[(\bar{x} \lor y \lor z \lor \bar{v} \lor \bar{w}) \iff \neg(x \land \bar{y} \land \bar{z} \land v \land w)\]

- What can we deduce by looking at just one clause?
  - Nothing unless it is a unit clause \((p)\): then we deduce that the literal \(p\) is true
    - \(x\) is true if \(p = x\)
    - \(x\) is false if \(p = \bar{x}\)
  - If the clause has two (independent) literals, any one can be false, providing that the other is true
  - Incomplete proof system (e.g. \((x \lor a) \land (\bar{x} \lor a) \land (\bar{y} \lor \bar{a}) \land (y \lor \bar{a})\))

However it propagates: if we have the unit literal \(p\), a clause containing \(\bar{p}\) can be reduced, and maybe become unit, triggering more unit propagation

\[(\bar{x} \lor y \lor z \lor \bar{v} \lor \bar{w}) \land (\bar{p} \lor x) \land (\bar{p} \lor \bar{y}) \land (q \lor \bar{z}) \land (q \lor v) \land (p) \land (q \lor w) \land (\bar{q} \lor \bar{x} \lor y)\]

- \((p)\) is a unit clause
- \((x)\) and \((\bar{y})\)
- \((\bar{q})\) is a unit clause
- \((\bar{z})\), \((v)\) and \((w)\) are unit clauses
- Unit propagation produces an empty clause
- Unit propagation solves *Horn*-SAT

- If a *Horn*-SAT formula has no unit clause, then every clause has at least one negative literal
  - The model with all variables false satisfies the formula

- Otherwise, unit propagate until reaching an inconsistency or a subformula without unit clauses

---

**Implementing Unit Propagation**

- A clause can either be:
  - Satisfied iff it contains at least one true literal
  - Falsified iff it contains only false literals
  - Unit iff it contains a single unknown literal, and $n - 1$ false literals
  - Unresolved iff it contains no true literal and at least two unknown literals
Unit Propagation with counters

Unit propagation algorithm (counters)

Organise clauses per literals (Clauses(l) is the set of clauses containing literal l)
keep an initially null counter \( \#f_i \) of false literals for each clause \( c_i \)
Put all unit clauses (true literals) in a list
while There is a non-processed true literal l do
mark l as processed
foreach \( c_i \in \text{Clauses}(l) \) do
  increment \( \#f_i \)
  if \( \#f_i = |c_i| \) then return FAIL
  if \( \#f_i = |c_i| - 1 \) then
    find the last literal and add it to the list of true literals
// at most once per literal: \( O(s) \)
// \( \Theta(|c_i|) \) at most once per clause: \( O(s) \)

- Let \( \varphi \) have \( n \) variables and \( m \) clauses, and let \( s \) be the total number of literals \( s = \sum_{i=1}^{m} |c_i| \)
- Worst case: every variable \( x \) is unit propagated (\( x \) if \( |\text{Clauses}(x)| \geq |\text{Clauses}(\overline{x})| \), and \( \overline{x} \) otherwise)
- Overall linear time \( \Theta(s) \) amortized down a branch

Watched Literals

- Invariant Watch only two non-false literals per clause
  - Watch(l) is the list of clauses that watches literal l
- Non-watched literals can become false, it cannot make the clause unit or falsified as long as two unknown literals remain
- When a watched literal become false, a replacement must be found
- When no replacement can be found, the clause is either unit or falsified
- Nothing to do when backtracking: the literals watched at level \( i \) cannot be false at level \( i - 1 \)

\[ \downarrow \downarrow \]
\[ ?\ ?\ ?\ ?\ ? \]
unresolved
\[ \downarrow \downarrow \]
\[ ?\ ?\ @3\ ?\ @1 \]
unresolved
\[ \downarrow \downarrow \]
\[ @5\ ?\ @3\ ?\ @1 \]
unresolved
\[ \downarrow \downarrow \]
\[ @5\ @7\ @3\ @7\ @1 \]
unit
\[ \downarrow \downarrow \]
\[ ?\ ?\ @3\ ?\ @1 \]
backtrack to level 4
Finding a new watched literals

- Scan the clause from first to last literal: possibly $\Theta(|c_i|)$ scans each costing $\Theta(|c_i|)$
  - Quadratic
- Store the initial position of the watch and scan forward
  - Linear but we must update the position of the watchers when backtracking
- Circular list: scan forward, but past the end and back to the current position
  - The clause is scanned at most twice: linear and no need to do anything when backtracking!

Average Complexity

- Let $n$ be the number of variables, $m$ be the number of clauses, $s = \sum_{i=1}^{m} |c_i|$ be the overall size of the formula, $k$ be the number of true literals after unit propagation
- Consider first the clauses that unit propagated
  - They contain only variables among the $k$ true literals
  - In order to propagate them, every literal must be explored (to increment the counter of find a new watched): it takes linear time in both cases call that $O(K)$
- Consider now the $m'$ clauses that did not unit propagate (and let $s'$ be their total size)
  - The counters algorithm increments the counters of every clause containing one of the $k$ true literals
    - The average number of clauses per literal is $\frac{s'}{n}$ so $\Theta\left(\frac{k s'}{n}\right)$ time in average
    - Overall: $\Theta(O(K) + \frac{k s'}{n})$ time
  - The watched algorithm increments finds a new watched literal for each of the clauses that watch it
    - A literal is watched by $\frac{m'}{n}$ of these clauses in average
    - The probability that a random literal is not false is $\frac{n-k}{n}$, so the expected number of literals to scan to find a valid one to watch is $\frac{n}{n-k}$
    - Overall: $\Theta(O(K) + \frac{k m'}{n-k})$ time
Structure

- **watches** \([\text{Literal} : l] \mapsto [\text{Clause}, ...]\)
  - list of clauses watching literal \(l\)
- **int**: to-propagate
  - the first non-unit-propagated literal in \(\text{unit-literals}\)

Functions

- **get-rank** \((\text{Clause}: c, \text{Literal}: l) \mapsto \{0, 1\}\)
  - \(0\) if \(l\) is the first watched in \(c\), \(1\) otherwise
- **get-index** \((\text{Clause}: c, \{0, 1\}: r) \mapsto \text{int}\)
  - index of the \((r + 1)\)-th watched in \(c\)
- **set-watcher** \((\text{Clause}: c, \text{Literal}: l, \{0, 1\}: r)\)
  - set \(l\) as \((r + 1)\)-th watcher of \(c\)
- **assign** \((\text{Literal}: l)\)
  - push \(l\) onto \(\text{unit-literals}\) and set model \([\var(l)]\)

---

Algorithm

**Unit propagation algorithm (watched literals)**

```
foreach \(c \in \text{clauses}\) do
  \(r \leftarrow \text{get-rank}(c, l); \ start \leftarrow i \leftarrow \text{get-index}(c, r)\)
  \(p \leftarrow c[\text{get-index}(c, 1-r)]\)
  if not satisfied\((p)\) then
    while true do
      \(i \leftarrow i + 1\)
      if \(i = |c|\) then \(i \leftarrow 0\)
      if \(i = \text{start}\) then break
      if \(c[i] \neq p\) then
        if not falsified\((c[i])\) then
          set-watcher\((c, c[i], r)\)
          break

  if \(i = \text{start}\) then
    if falsified\((p)\) then return false
    assign\((p)\)

return true
```
Resolution rule:

\[
\frac{(\alpha \lor x) \quad (\beta \lor \neg x)}{\alpha \lor \beta}
\]

Complete proof system for propositional logic: If the formula \( \varphi \) is not satisfiable, then there is sequence of resolution steps that produce the empty clause \( \bot \):

\[
\begin{align*}
(x \lor a) & \quad (\neg x \lor a) & \quad (\neg y \lor \neg a) & \quad (y \lor \neg a) \\
\quad (a) & \quad \quad (\neg a) & \quad \quad \quad \bot
\end{align*}
\]

Self-subsuming resolution (with \( \alpha' \subseteq \alpha \)):

\[
\frac{(\alpha \lor x) \quad (\alpha' \lor \neg x)}{\alpha}
\]

Resolution solves 2-SAT in polynomial time.

Resolution is a complete refutation system for SAT (and hence for 2-SAT).

Resolvent clauses have at most 2 literals.

- There are at most \( n^2 \) binary clauses.
Algorithm for 2-SAT

\[(\neg y \lor z) \land (\neg z \lor \neg x) \land (x \lor \neg z) \land (\neg z \lor \neg x) \land (y \lor y) \land (y \lor z) \land (z \lor z) \land (x \lor x)\]

### Algorithm

- **x ∨ y** is equivalent to \( \neg x \Rightarrow y \) and \( \neg y \Rightarrow x \)
- Add transitive edges
  - If there is an inconsistency, then the formula is not satisfiable
  - If not, it is satisfiable, because the choice \( x \Rightarrow \neg x \) closes a cycle only if there is a path \( \neg x \Rightarrow x \)

### Proofs

- **SAT** is in **NP**: if an instance is satisfiable, it is possible to prove it efficiently
  - Just show a model and check clause by clause that it is correct (it is a certificate)
- What about the question “is \( \varphi \) unsatisfiable?”, or “is \( \varphi \) a tautology?”
  - There might not exist short certificates for problems in **coNP**, but we can provide a long one
- Proof system: maps to every unsatisfiable formula \( \varphi \) a refutation \( R \)
  - There is a polynomial algorithm (in \(|R|\)) to check the refutation proof
    - *Pebbling formulas*
\[ \varphi = (a \lor b) \land (\lnot a \lor c \lor \lnot d) \land (a \lor c \lor \lnot d) \land (\lnot c \lor \lnot e) \land (\lnot c \lor e) \land (c \lor d) \]

\[ c_1 = (\lnot c \lor e) \in \varphi \]
\[ c_2 = (\lnot c \lor \lnot e) \in \varphi \]
\[ c_3 = (\lnot c) \text{ resolvant of } c_1 \text{ and } c_2 \]
\[ c_4 = (a \lor c \lor \lnot d) \in \varphi \]
\[ c_5 = (\lnot a \lor c \lor \lnot d) \in \varphi \]
\[ c_6 = (c \lor \lnot d) \text{ resolvant of } c_4 \text{ and } c_5 \]
\[ c_7 = (c \lor d) \in \varphi \]
\[ c_8 = (c) \text{ resolvant of } c_6 \text{ and } c_7 \]
\[ c_9 = () \text{ resolvant of } c_3 \text{ and } c_8 \]
Resolution is sound and complete

- **Soundness**: if there exists a resolution refutation then the formula is unsatisfiable
  - Resolution is a *sound* proof system simply because the resolution step is sound

- **Completeness**: if a formula is unsatisfiable then there exists a resolution refutation of that formula
  - Tree search is obviously a complete proof system
  - To every search tree we can associate a resolution proof
  - Therefore resolution is a *complete* proof system

Resolution: conciseness

- What does make a proof system good? (besides soundness and completeness)
  - A good proof system is one that allows shorter proofs
    - If refutations are polynomial size in general, then $\text{NP} = \text{coNP}$

- For any tree search refutation, there is a resolution refutation of the same size

- There exist formulas with short resolution refutation but *exponential* tree search refutations
Pigeon Hole Principle

If \( m > n \) there is no injective mapping of \( m \) objects onto \( n \)

\[
\text{PHP}^{m \to n} : \quad (x_{1,1} \lor x_{1,2} \lor \ldots \lor x_{1,n}) \land \\
\ldots \\
(x_{m,1} \lor x_{m,2} \lor \ldots \lor x_{m,n}) \land \\
\bigwedge_{1 \leq i < j \leq m} (x_{i,1} \lor x_{i,1}) \land \\
\ldots \\
\bigwedge_{1 \leq i < j \leq m} (x_{i,n} \lor x_{i,n})
\]

- Pigeon 1 needs a hole
- Pigeon \( m \) needs a hole
- Hole 1 can contain at most 1 pigeon
- Hole \( n \) can contain at most 1 pigeon

Resolution refutations of the pigeon hole principle are exponential

Using induction, for instance, one can make a linear size refutation