PROPOSITIONAL LOGIC (2)

based on

Huth & Ruan
Logic in Computer Science:
Modelling and Reasoning about Systems
Cambridge University Press, 2004

Russell & Norvig
Artificial Intelligence:
A Modern Approach
Prentice Hall, 2010

Clauses

- Clauses are formulas consisting only of $\vee$ and $\neg$

$$p \vee q \vee \neg r$$
$$\neg p \vee \neg q$$

(brackets within a clause are not allowed!)

they can also be written using →, $\vee$ (after →) and $\wedge$ (before →)

$$r \rightarrow p \vee q$$
$$p \wedge q \rightarrow \bot$$
$$\top \rightarrow p \vee q$$
$$\top \rightarrow \bot$$

Clause without positive literal

Clause without negative literal

Empty clause is considered *false*

an atom or its negation is called a *literal*

Conjunctive & Disjunctive Normal Form

- A formula is in **conjunctive normal form** if it consists of a conjunction of clauses

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

$$(r \rightarrow p \vee q) \wedge (q \rightarrow p) \wedge (\top \rightarrow p \vee r)$$

  - "conjunction of disjunctions"
- A formula is in **disjunctive normal form** if it consists of a disjunction of conjunctions

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

Conjunctive & Disjunctive Normal Form

- The transformation from CNF to DNF is exponential

$$
(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_3 \vee q_3) =
\begin{array}{l}
(p_1 \wedge p_2 \wedge p_3) \vee \\
(p_1 \wedge p_2 \wedge q_3) \vee \\
(p_1 \wedge q_2 \wedge p_3) \vee \\
(p_1 \wedge q_2 \wedge q_3) \vee \\
(q_1 \wedge p_2 \wedge p_3) \vee \\
(q_1 \wedge p_2 \wedge q_3) \vee \\
(q_1 \wedge q_2 \wedge p_3) \vee \\
(q_1 \wedge q_2 \wedge q_3)
\end{array}
$$

Conjunctive Normal Form

- Any formula can be written in CNF

$$
\begin{aligned}
(p \vee q \rightarrow r) \vee (q \rightarrow p) &= \neg(p \vee q) \vee r \vee \neg q \vee p \\
&= (\neg p \wedge \neg q) \vee r \vee \neg q \vee p \\
&= (\neg p \vee r \vee \neg q \vee p) \\
&\quad \wedge(\neg q \vee r \vee \neg q \vee p) \\
&= (\neg q \vee r \vee p)
\end{aligned}
$$

(consequently, any formula can also be written in DNF, but the DNF formula may be exponentially larger)

Checking Satisfiability of Formulas in DNF

- Checking DNF satisfiability is easy: process one conjunction at a time; if at least one conjunction is not a contradiction, the formula is satisfiable

  → DNF satisfiability can be decided in polynomial time

$$(p_1 \wedge p_3 \wedge \neg p_3)\vee$$
$$(p_1 \wedge \neg p_2 \wedge \neg p_3)\vee$$
$$(p_1 \wedge \neg p_2 \wedge p_3)\vee$$
$$(\neg p_1 \wedge p_3 \wedge \neg p_3)\vee$$

Conversion to DNF is not feasible in most cases (exponential blowup)

Checking Satisfiability of Formulas in CNF

- No polynomial algorithm is known for checking the satisfiability of arbitrary CNF formulas

Example: we could use such an algorithm to solve graph coloring with *k* colors

- for each node *i*, create a formula

$$\phi_i = p_{i1} \vee p_{i2} \vee \cdots \vee p_{ik}$$

indicating that each node *i* must have a color

- for each node *i* and different pair of colors c_1 and c_2, create a formula

$$\phi_{ic_1c_2} = \neg(p_{ic_1} \wedge p_{ic_2}) = \neg p_{ic_1} \vee \neg p_{ic_2}$$

indicating a node may not have more than 1 color

- for each edge, create *k* formulas

$$\phi_{ijc} = \neg(p_{ic} \wedge p_{jc}) = \neg p_{ic} \vee \neg p_{jc}$$

indicating that a pair connected nodes *i* and *j* may not both have color *c* at the same time

Resolution Rule

Essential in most satisfiability solvers for CNF formulas is the **resolution rule** for clauses:

Given two clauses $l_1 \vee \cdots \vee l_k$ and $m_1 \vee \cdots \vee m_n$, where $l_1, \ldots, l_k, m_1, \ldots, m_n$ represent literals and it holds that $l_i = \neg m_j$, then it holds that

$$l_1 \vee \cdots \vee l_k, m_1 \vee \cdots \vee \cdots m_n \vdash_R$$
$$l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots m_n$$

Example: $p \vee q \vee \neg r, r \vee s \vdash_R p \vee q \vee s$
$r \rightarrow p \vee q, r \vee s \vdash_R p \vee q \vee s$

Proof for Resolution

on an example

1.	$p \vee q$	premise
2.	$q \to r$	premise
3.	p	assumption
4.	$p \vee r$	$\vee$i 3
5.	q	assumption
6.	r	$\to$e 2,5
7.	$p \vee r$	$\vee$i 6
8.	$p \vee r$	$\vee$e 1,3-4, 5-7

$\neg q \vee r$

Completeness of Resolution

- If it holds that $C_1, \ldots, C_n \models \bot$ for clauses $C_1, \ldots, C_n$ (i.e. the clauses are a contradiction), then we can derive $\bot$ from $C_1, \ldots, C_n$ by repeated application of the resolution rule

$$\begin{aligned} p, p \rightarrow q \vee r, q \rightarrow \bot, r \rightarrow \bot \quad & \vdash_R \quad q \vee r, q \rightarrow \bot, r \rightarrow \bot \\ & \vdash_R \quad r, r \rightarrow \bot \\ & \vdash_R \quad \bot \end{aligned}$$

How to find the resolution steps in general?
For some types of clauses it is easier…

Definite clauses & Horn clauses

- A **<u>definite clause</u>** is a clause with exactly one positive literal

$$p, q, p \wedge q \rightarrow t$$

- A **<u>horn clause</u>** is a clause with at most one positive literal

$$p, q, p \wedge q \rightarrow t, p \wedge q \rightarrow \perp$$

A clause with one positive literal is called a **fact**

Forward chaining for Definite clauses

- The **forward chaining algorithm** calculates facts that can be entailed from a set of definite clauses

C = initial set of definite clauses
repeat
    if there is a clause $p_1,\ldots,p_n \rightarrow q$ in ***C*** where $p_1,\ldots,p_n$ are
        facts in ***C*** **then**
        add fact q to ***C***
    end if
until no fact could be added
return all facts in ***C***

Resolution

This algorithm is complete for facts: any fact that is entailed, will be derived.

Forward chaining for Horn clauses

- We now also allow to add $\perp$ and other clauses without positive literals to ***C***
- We stop immediately $\perp$ when is found, and return that the set of formulas is contradictory.

$$\mathbf{C}_1 = \{p, p \to q, p \wedge q \to r, r \to \perp\}$$
$$\mathbf{C}_2 = \{p, q, p \to q, p \wedge q \to r, r \to \perp\}$$
$$\mathbf{C}_3 = \{p, q, r, p \to q, p \wedge q \to r, r \to \perp\}$$
$$\mathbf{C}_4 = \{p, q, r, \perp, p \to q, p \wedge q \to r, r \to \perp\}$$

Note:

1) a set of definite clauses is always satisfiable.

2) we can decide in linear time whether a set of Horn clauses is satisfiable.

Deciding entailment for Horn clauses

- Suppose we would like to know whether

$$C_1, \ldots, C_n \models p_1, \ldots, p_n \to q$$

where $C_1, \ldots, C_n$ are Horn clauses; then it suffices to determine whether

$$C_1, \ldots, C_n, p_1, \ldots, p_n \vdash_R q$$

(we can show this by means of → introduction)

- As entailment of facts can be decided in linear time, Horn clause entailment can be determined in linear time as well

Deciding satisfiability of CNF formulas: DPLL

- The DPLL algorithm for deciding satisfiability was proposed by Davis, Putman, Logeman and Loveland (1960, 1962)
- General ideas:
  - we perform **depth-first** over the space of all possible valuations
  - based on a partial valuation, we **simplify** the formula to remove redundant literals
  - based on the formula, we **fix** the valuation of as many atoms as possible

DPLL: Simplification

- If the valuation of atom *p* is "true"
  - every clause in which literal *p* occurs, is removed
  - from every clause in which *p* is negated, $\neg p$ is removed

$$\{p = true\}, (p \vee q) \wedge (q \vee \neg r) \Rightarrow \{p = true\}, (q \vee \neg r)$$
$$\{p = true\}, (\neg p \vee q) \wedge (q \vee \neg r) \Rightarrow \{p = true\}, (q \wedge (q \vee \neg r))$$

similar to resolution

- Similarly, if the valuation of atom *p* is "false"
  - every clause in which literal $\neg p$ occurs, is removed
  - from every clause in which *p* occurs, literal *p* is removed

DPLL: Simplification

- Special case 1 of simplification is when an empty clause is obtained, i.e. the clause $\perp$

$$\begin{aligned} \{p = true\}, \neg p \wedge (q \vee r) &\Rightarrow \{p = true\}, \perp \wedge (q \vee r) \\ &\Rightarrow \{p = true\}, \perp \end{aligned}$$

  - in this case the current valuation can never be extended to a valuation that satisfies the formula
- Special case 2 of simplification is when the empty CNF formula is obtained, i.e. the formula $\top$

$$\{\text{p=false}\}, \neg p \Rightarrow \{p = false\}, \top$$

  - in this case we have found a satisfying valuation

DPLL: Pure symbols

- If an atom always has the same sign in a formula (i.e., the literals *p* and $\neg p$ do not occur at the same time), the atom is called *pure*. We fix the valuation of a pure atom to the value indicated by this sign

$$\emptyset, (p \lor q) \land (p \lor \neg r) \Rightarrow \{p = true\}, (p \lor q) \land (p \lor \neg r)$$

$$\emptyset, (\neg p \lor q) \land (\neg p \lor \neg r) \Rightarrow \{p = false\}, (\neg p \lor q) \land (\neg p \lor \neg r)$$

  - Note: we can apply simplification afterwards and remove redundant clauses

DPLL: Unit clauses

- If a clause consists of only one literal (positive or negative), this clause is called a *unit clause*. We fix the valuation of an atom occurring in a unit clause to the value indicated by the sign of the literal.

$$\emptyset, p \wedge (q \vee r) \Rightarrow \{p = true\}, p \wedge (q \vee r)$$

- Also here, we apply simplification afterwards; after simplification, we may have new unit clauses, which we can use again; this process is called *unit propagation*

$$\emptyset, p \wedge (\neg p \vee r)$$
$$\Rightarrow \{p = true\}, p \wedge (\neg p \vee r)$$
$$\Rightarrow \{p = true\}, r \qquad \Rightarrow \{p = true, r = true\}, r$$

DPLL Algorithm

DPLL (valuations V, formula φ)

φ' = simplification of φ based on V

if φ' is an empty formula **then return** true

if φ' contains the empty clause **then return** false

if φ' contains a pure atom p with sign v **then**

return DPLL($V \cup \{p=v\}$, φ')

if φ' contains a unit clause for atom p with sign v **then**

return DPLL($V \cup \{p=v\}$, φ')

let p be an arbitrary atom occurring in φ'

if DPLL($V \cup \{p=true\}$, φ') **then return** true

else return DPLL($V \cup \{p=false\}$, φ')

Optimizations of DPLL

- **Component analysis:** if the clauses can be partitioned such that variables are not shared between clauses in different partitions, we solve the partitions independently

$$\underbrace{(p \vee q) \wedge (\neg p)}_{\text{component 1}} \wedge \underbrace{(r \vee s) \wedge r}_{\text{component 2}}$$

- **Value and variable ordering:** when choosing the next atom to fix, try to be clever (i.e. pick one that occurs in many clauses)

Optimizations of DPLL

- **<u>Clause learning:</u>** if a contradiction is found, try to find out which assignments caused this contradiction, and add a clause (entailed by the original CNF formula) to avoid this combination of assignments in the future

<u>Example</u>

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

Note: no unit propagation or pure literals present, branching necessary.

Optimizations of DPLL

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

No propagation possible, branch with *p*=true

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

No propagation possible, branch with *q*=true

$(r \vee t) \wedge (\neg r \vee t) \wedge (r \vee \neg t) \wedge (\neg r \vee \neg t)$

No propagation possible, branch with *r*=true

$t \wedge \neg t$

Conflict found in *t* → apply resolution on *t* for the original versions of conflicting clauses $(\neg r \vee t) \wedge (\neg r \vee \neg t)$ → clause $\neg r$ is entailed by the original formula, add $\neg r$ as learned clause to original formula → apply propagation on this formula new → *p=true, q=true, r=false* → search stops

Optimizations of DPLL

- **Random restarts:** if the search is unsuccessful too long, stop the search, and start from scratch with learned clauses (and possibly a different variable/value ordering)
- **Clever indexing:** use heavily optimized data structures for storing clauses, atoms, and lists of clauses in which atoms occur
- **Portfolios:** run several different solvers for a short time; use data gathered from these runs to select the final solver to execute

Applications of SAT solvers

- Model checking
- Planning
- Scheduling
- Experiment design
- Protocol design (networks)
- Multi-agent systems
- E-commerce
- Software package management
- Learning automata
- …

http://www.youtube.com/watch?v=0gt503wK7AI

First order logic

- Essentially, first order logic adds variables in logic formulas

Assume we have three cats (Anna, Bella, Cat), and cats have tails.

In **propositional logic**, we could write:
iscatAnna, iscatBella, iscatCat, iscatAnna → hastailAnna, iscatBella → hastailBella, iscatCat → hastailCat.

In **first order logic**, we would write:
iscat(anna), iscat(bella), iscat(anna), $\forall X$ iscat(X)→hastail(X)