PROPOSITIONAL LOGIC (3)

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

The story till now...

- Semantic entailment: $\varphi \models \psi$
  Are all models of formula φ also models of ψ?
  - If $\varphi \models \perp$, the formula φ is unsatisfiable
  - We are interested in procedures for determining this relationship
- **Approach 1:** search for a proof that uses the rules of natural deduction
  - Natural deduction provides "natural" proofs, i.e. short arguments such as humans would give; however, such proofs can be hard to find by a computer

The story till now…

- **Approach 2:** employ the rules of resolution
  - Note that $\varphi \models \psi$ iff $\varphi \wedge \neg\psi \models \bot$
  - We first *normalize* formulas φ and $\neg\psi$ in conjunctive normal form (giving φ' and ψ')
  - Then we repeatedly apply the *resolution rule* on $\varphi' \wedge \psi'$ till we either cannot derive new clauses or we derive $\bot$
    - If we derive $\bot$ by means of resolution, it can be shown that the formula is unsatisfiable
    - Otherwise, it is satisfiable

The story till now…

- Example of resolution

$$\varphi = (a \vee b \vee c) \wedge (\neg a \vee a') \wedge (\neg b \vee b') \wedge (\neg c \vee c')$$

$$\begin{aligned} \varphi \quad & \vdash_R \quad \varphi \wedge (a' \vee b \vee c) \wedge (a \vee b' \vee c) \wedge (a \vee b \vee c') = \varphi' \\ & \vdash_R \quad \varphi' \wedge (a' \vee b' \vee c) \wedge (a' \vee b \vee c') \wedge (a \vee b' \vee c') = \varphi'' \\ & \vdash_R \quad \varphi'' \wedge (a' \vee b' \vee c') \end{aligned}$$

- In the general case, the repeated application of resolution can yield an exponential number of clauses…
  - We would prefer not to store and generate all of these

The story till now…

- Resolution can be applied efficiently on *definite* clauses, by means of the forward chaining algorithm

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*** ← Resolution
    end if
until no fact could be added
return all facts in ***C***

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

The story continues

- Can we use the ideas of forward chaining and resolution in a more efficient algorithm?

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 $\bot$

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

  - 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: Fixing 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

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

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

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

DPLL: Fixing 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')

Branching

Optimizations of DPLL

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

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

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

Optimizations of DPLL

- **Clause learning:** 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

Example

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

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

SAT solvers are usually implementations of the DPLL algorithm. They are used for:

- 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

Progress in SAT solvers