CS 188: Artificial Intelligence

Inference in Propositional Logic

Instructors: Stuart Russell and Dawn Song

University of California, Berkeley
Inference (reminder)

- **Method 1: model-checking**
  - For every possible world, if $\alpha$ is true make sure that is $\beta$ true too
- **Method 2: theorem-proving**
  - Search for a sequence of proof steps (applications of *inference rules*) leading from $\alpha$ to $\beta$
- **Sound** algorithm: everything it claims to prove is in fact entailed
- **Complete** algorithm: every that is entailed can be proved
Simple theorem proving: Forward chaining

- Forward chaining applies Modus Ponens to generate new facts:
  - Given \( X_1 \land X_2 \land \ldots \land X_n \Rightarrow Y \) and \( X_1, X_2, \ldots, X_n \), infer \( Y \)
- Forward chaining keeps applying this rule, adding new facts, until nothing more can be added
- Requires KB to contain only definite clauses:
  - (Conjunction of symbols) \( \Rightarrow \) symbol; or
  - A single symbol (note that \( X \) is equivalent to \( True \Rightarrow X \))
- Runs in linear time using two simple tricks:
  - Each symbol \( X_i \) knows which rules it appears in
  - Each rule keeps count of how many of its premises are not yet satisfied
**Forward chaining algorithm: Details**

**function** PL-FC-ENTAILS?(KB, q) **returns** true or false

count ← a table, where count[c] is the number of symbols in c’s premise
inferred ← a table, where inferred[s] is initially false for all s
agenda ← a queue of symbols, initially symbols known to be true in KB

**while** agenda is not empty **do**

p ← Pop(agenda)
if p = q then **return** true
if inferred[p] = false then
  inferred[p] ← true
  **for each** clause c in KB where p is in c.premise **do**
    decrement count[c]
  **if** count[c] = 0 **then** add c.conclusion to agenda

**return** false
Properties of forward chaining

- Theorem: FC is sound and complete for definite-clause KBs
- Soundness: follows from soundness of Modus Ponens (easy to check)
- Completeness proof:
  1. FC reaches a fixed point where no new atomic sentences are derived
  2. Consider the final set of known-to-be-true symbols as a model $m$ (other ones false)
  3. Every clause in the original KB is true in $m$
     - Proof: Suppose a clause $a_1 \land \ldots \land a_k \Rightarrow b$ is false in $m$
     - Then $a_1 \land \ldots \land a_k$ is true in $m$ and $b$ is false in $m$
     - Therefore the algorithm has not reached a fixed point!
  4. Hence $m$ is a model of KB
  5. If KB $\models q$, $q$ is true in every model of KB, including $m$
Resolution (briefly)

- The resolution inference rule takes two implication sentences (of a particular form) and infers a new implication sentence:

  - Example: \( A \land B \land C \Rightarrow U \lor V \)
  \[
  D \land E \land U \Rightarrow X \lor Y \\
  \]
  \[
  A \land B \land C \land D \land E \Rightarrow V \lor X \lor Y \\
  \]

- Resolution is complete for propositional logic
- Exponential time in the worst case
Satisfiability and entailment

- A sentence is *satisfiable* if it is true in at least one world.
- Suppose we have a hyper-efficient SAT solver (**WARNING: NP-COMPLETE** 😈 😈 😈); how can we use it to test entailment?
  - $\alpha \models \beta$
  - iff $\alpha \implies \beta$ is true in all worlds
  - iff $\neg(\alpha \implies \beta)$ is false in all worlds
  - iff $\alpha \land \neg \beta$ is false in all worlds, i.e., unsatisfiable
- So, add the *negated* conclusion to what you know, test for (un)satisfiability; also known as *reductio ad absurdum*
- Efficient SAT solvers operate on *conjunctive normal form*
Conjunctive normal form (CNF)

- Every sentence can be expressed as a conjunction of clauses
- Each clause is a disjunction of literals
- Each literal is a symbol or a negated symbol

Conversion to CNF by a sequence of standard transformations:

- $\text{At}_{1,1,0} \Rightarrow (\text{Wall}_{0,1} \iff \text{Blocked}_{W,0})$
- $\text{At}_{1,1,0} \Rightarrow ((\text{Wall}_{0,1} \Rightarrow \text{Blocked}_{W,0}) \land (\text{Blocked}_{W,0} \Rightarrow \text{Wall}_{0,1}))$
- $\neg \text{At}_{1,1,0} \lor ((\neg \text{Wall}_{0,1} \lor \text{Blocked}_{W,0}) \land (\neg \text{Blocked}_{W,0} \lor \text{Wall}_{0,1}))$
- $(\neg \text{At}_{1,1,0} \lor \neg \text{Wall}_{0,1} \lor \text{Blocked}_{W,0}) \land (\neg \text{At}_{1,1,0} \lor \neg \text{Blocked}_{W,0} \lor \text{Wall}_{0,1})$

- Replace biconditional by two implications
- Replace $\alpha \Rightarrow \beta$ by $\neg \alpha \lor \beta$
- Distribute $\lor$ over $\land$
Efficient SAT solvers

- **DPLL** (Davis-Putnam-Logemann-Loveland) is the core of modern solvers
- Recursive depth-first search over models with some extras:
  - *Early termination*: stop if
    - all clauses are satisfied; e.g., \((A \lor B) \land (A \lor \neg C)\) is satisfied by \(\{A=\text{true}\}\)
    - any clause is falsified; e.g., \((A \lor B) \land (A \lor \neg C)\) is satisfied by \(\{A=\text{false}, B=\text{false}\}\)
  - *Pure literals*: if all occurrences of a symbol in as-yet-unsatisfied clauses have the same sign, then give the symbol that value
    - E.g., \(A\) is pure and positive in \((A \lor B) \land (A \lor \neg C) \land (C \lor \neg B)\) so set it to \text{true}
  - *Unit clauses*: if a clause is left with a single literal, set symbol to satisfy clause
    - E.g., if \(A=\text{false}\), \((A \lor B) \land (A \lor \neg C)\) becomes \((\text{false} \lor B) \land (\text{false} \lor \neg C)\), i.e. \((B) \land (\neg C)\)
    - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.
function DPLL(clauses, symbols, model) returns true or false

if every clause in clauses is true in model then return true
if some clause in clauses is false in model then return false

P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
if P is non-null then return DPLL(clauses, symbols–P, model∪\{P=value\})

P, value ← FIND-UNIT-CLAUSE(clauses, model)
if P is non-null then return DPLL(clauses, symbols–P, model∪\{P=value\})

P ← First(symbols); rest ← Rest(symbols)

return or(DPLL(clauses, rest, model∪\{P=true\}), DPLL(clauses, rest, model∪\{P=false\}))
Efficiency

- Naïve implementation of DPLL: solve ~100 variables
- Extras:
  - Smart variable and value ordering
  - Divide and conquer
  - Caching unsolvable subcases as extra clauses to avoid redoing them
  - Cool indexing and incremental recomputation tricks so that every step of the DPLL algorithm is efficient (typically O(1))
    - Index of clauses in which each variable appears +ve/-ve
    - Keep track number of satisfied clauses, update when variables assigned
    - Keep track of number of remaining literals in each clause
- Real implementation of DPLL: solve ~100000000 variables
Circuit verification: does this VLSI circuit compute the right answer?
Software verification: does this program compute the right answer?
Software synthesis: what program computes the right answer?
Protocol verification: can this security protocol be broken?
Protocol synthesis: what protocol is secure for this task?
Lots of combinatorial problems: what is the solution?
Planning: how can I eat all the dots???
A knowledge-based agent

```plaintext
function KB-AGENT(percept) returns an action

persistent: KB, a knowledge base
            t, an integer, initially 0

TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))

action ← ASK(KB, MAKE-ACTION-QUERY(t))

TELL(KB, MAKE-ACTION-SENTENCE(action, t))

t ← t + 1

return action
```
Reminder: Partially observable Pacman

- Pacman perceives wall/no-wall in each direction
- Variables:
  - Wall_0,0, Wall_0,1, ...
  - Blocked_W_0, Blocked_N_0, ..., Blocked_W_1, ...
  - W_0, N_0, ..., W_1, ...
  - At_0,0_0, At_0,1_0, ..., At_0,0_1, ...
Pacman’s knowledge base: Basic PacPhysics

- **Map**: where the walls are and aren’t
- **Initial state**: Pacman is definitely somewhere
- **Domain constraints**:
  - Pacman does exactly one action at each step
  - Pacman is in exactly one location at each step
- **Sensor model**: \(<\text{Percept}_t> \iff \langle\text{some condition on world}_t\rangle\>
- **Transition model**:
  - \(<\text{at } x,y_t> \iff [\text{at } x,y_{t-1} \text{ and stayed put}] \lor [\text{next to } x,y_{t-1} \text{ and moved to } x,y]\)
State estimation

- **State estimation** means keeping track of what’s true now.
- A logical agent can just ask itself!
  - E.g., ask whether $KB \land <\text{actions}> \land <\text{percepts}> |\models \text{At}_2,2_6$
- This is “lazy”: it analyzes one’s whole life history at each step!
- A more “eager” form of state estimation:
  - After each action and percept
    - For each state variable $X_t$
      - If $KB \land \text{action}_{t-1} \land \text{percept}_t |\models X_t$, add $X_t$ to $KB$
      - If $KB \land \text{action}_{t-1} \land \text{percept}_t |\models \neg X_t$, add $\neg X_t$ to $KB$
Example: Localization in a known map

- Initialize the KB with **PacPhysics** for $T$ time steps
- Run the Pacman agent for $T$ time steps:
  - After each action and percept
    - For each variable $At_{x,y}_t$
      - If $KB \land action_{t-1} \land percept_t \models At_{x,y}_t$, add $At_{x,y}_t$ to KB
      - If $KB \land action_{t-1} \land percept_t \models \neg At_{x,y}_t$, add $\neg At_{x,y}_t$ to KB
    - Choose an action
- Pacman’s *possible* locations are those that are not provably false
Localization demo

- Percept
- Action
- Percept
- Action
- Percept
Localization demo

- Percept
- Action: SOUTH
- Percept
- Action
- Percept
Localization demo

- Percept
- Action: SOUTH
- Percept
- Action: SOUTH
- Percept
- Action
- Percept
Localization demo

- Percept
- Action: SOUTH
- Percept
- Action: SOUTH
- Percept
Localization demo

- Percept
- Action
- Percept
- Action
- Percept
- Percept
Localization demo

- Percept
- Action: WEST
- Percept
- Action
- Percept
- Action
- Percept
Localization demo

- Percept
- Action: WEST
- Percept
- Action
- Percept
- Action
- Percept
Localization demo

- Percept
- Action \textit{WEST}
- Percept
- Action \textit{WEST}
- Percept
- Action
- Percept
Localization demo

- Percept
- Action: WEST
- Percept
- Action: WEST
- Percept
- Action
- Percept
Localization demo

- Percept
- Action \textit{WEST}
- Percept
- Action \textit{WEST}
- Percept
- Action \textit{WEST}
- Percept
Localization demo

- Percept
- Action $WEST$
- Percept
- Action $WEST$
- Percept
- Action $WEST$
- Percept
Localization with random movement
Example: Mapping from a known relative location

- Without loss of generality, call the initial location 0,0
- The percept tells Pacman which actions work, so he always knows where he is
  - “Dead reckoning”
- Initialize the KB with **PacPhysics** for \( T \) time steps, starting at 0,0
- Run the Pacman agent for \( T \) time steps
  - At each time step
    - Update the KB with previous action and new percept facts
    - For each wall variable \( \text{Wall}_{x,y} \)
      - If \( \text{Wall}_{x,y} \) is entailed, add to KB
      - If \( \neg \text{Wall}_{x,y} \) is entailed, add to KB
    - Choose an action
- The wall variables constitute the map
Mapping demo

- Percept
- Action: NORTH
- Percept
- Action: EAST
- Percept
- Action: SOUTH
- Percept
Example: Simultaneous localization and mapping

- Often, dead reckoning won’t work in the real world
  - E.g., sensors just count the number of adjacent walls (0,1,2,3 = 2 bits)
- Pacman doesn’t know which actions work, so he’s “lost”
  - So if he doesn’t know where he is, how does he build a map???
- Initialize the KB with PacPhysics for $T$ time steps, starting at 0,0
- Run the Pacman agent for $T$ time steps
  - At each time step
    - Update the KB with previous action and new percept facts
    - For each $x,y$, add either $\text{Wall}_{x,y}$ or $\neg \text{Wall}_{x,y}$ to KB, if entailed
    - For each $x,y$, add either $\text{At}_{x,y,t}$ or $\neg \text{At}_{x,y,t}$ to KB, if entailed
    - Choose an action
Planning as satisfiability

- Given a hyper-efficient SAT solver, can we use it to make plans?
- Yes, for fully observable, deterministic case:
  - planning problem is solvable iff there is some satisfying assignment
  - solution obtained from truth values of action variables
- For \( T = 1 \) to \( \infty \),
  - Initialize the KB with \texttt{PacPhysics} for \( T \) time steps
  - Assert goal is true at time \( T \)
- Read off action variables from SAT-solver solution
SCORE: 0
Logical inference computes entailment relations among sentences

Theorem provers apply inference rules to sentences
  - Forward chaining applies modus ponens with definite clauses; linear time
  - Resolution is complete for PL but exponential time in the worst case

SAT solvers based on DPLL provide incredibly efficient inference

Logical agents can do localization, mapping, SLAM, planning (and many other things) just using one generic inference algorithm on one knowledge base