CS 188: Artificial Intelligence

Inference in Propositional Logic

Quiz 1:
Is the following sentence true in the following model?

Model: {A: True, B: False, C: True, D: True, E: True}
Sentence: \( ((\neg A \lor B) \implies \neg (D \lor E)) \land (C \iff C) \)

Quiz 2:
Do the following sentences entail H?

\begin{align*}
A, C, D, F, G, F \land G &\Rightarrow D, B \land A \land G \Rightarrow E, \\
A \land E &\Rightarrow H, C \land D \land E \Rightarrow A, \\
F \land G &\Rightarrow E, B \Rightarrow E, L \Rightarrow B, B \Rightarrow H
\end{align*}

Slides mostly from Stuart Russell

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 \( \text{True} \Rightarrow X \))

- 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

Reminder about definite clauses: $X_1 \land X_2 \land \ldots \land X_n \Rightarrow Y$

```plaintext
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 symbols 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
```
### Forward chaining algorithm: Example

**Sentences: A, B, D, A \land B \Rightarrow C, C \land D \Rightarrow E**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>A \land B \Rightarrow C</th>
<th>C \land D \Rightarrow E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inferred?</td>
<td>Agenda?</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Query: F**
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 $m$
Backward chaining

- Puzzle to try at home: develop a “Backward chaining algorithm”
- Then look it up to see if you’ve reproduced the standard algorithm
- Idea:
  - Keep track of what you’re trying to prove
  - Look for sentences that might get you there
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 possible worlds
  - iff $\neg(\alpha \implies \beta)$ is false in all possible worlds
  - iff $\alpha \land \neg\beta$ is false in all possible 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

\[(A \lor \neg B \lor \neg C) \land (\neg A) \land (\neg D) \land (\neg E) \land (\neg F)\]

Convert anything to CNF with standard transformations!

- \[\neg \text{At}_1,1_0 \lor (\neg \text{Wall}_0,1 \lor \text{Blocked}_W_0) \lor (\neg \text{Blocked}_W_0 \lor \text{Wall}_0,1)\]

Replace biconditional by two implications

Replace \( \alpha \Rightarrow \beta \) by \( \neg \alpha \lor \beta \)

Distribute \( \lor \) over \( \land \)
Distributivity

- \((A \lor B) \land C = (A \land C) \lor (B \land C)\)
- (I’m in SF or I’m in Berkeley) and I’m alive
- (I’m in SF and I’m alive) or (I’m in Berkeley and I’m alive)
- \((A \land B) \lor C = (A \lor C) \land (B \lor C)\)
- \(\neg (A \land B) = (\neg A \lor \neg B)\)
- It’s not the case that (I’m alive and I’m kicking)
- (I’m not alive) or (I’m not kicking)
- \(\neg (A \lor B) = (\neg A \land \neg B)\)
Reduction to CNF

Goal: \((A \lor \neg B \lor \neg C) \land (\neg A) \land (\neg D \lor B \lor C \lor E \lor F) \land (\neg E \lor \neg F) \land (B \lor E)\)

1. Get rid of \(\iff\)
   - Replace \(\alpha \iff \beta\) with \((\alpha \implies \beta) \land (\beta \implies \alpha)\)
   - \(\alpha\) and \(\beta\) could be long expressions, not just symbols

2. Get rid of \(\implies\)
   - Replace \(\alpha \implies \beta\) with \(\beta \lor \neg \alpha\)

3. Distribute \(\neg\)s to lower levels
   - Replace \(\neg(\alpha \land \neg \beta)\) with \(\neg \alpha \lor \beta\)

4. If any \(\lor\)s are at a higher level than \(\land\)s, distribute down
   - Replace \((\alpha \land \beta \land \gamma) \lor (\varepsilon \land \delta)\) with...
   - \((\alpha \lor \varepsilon) \land (\beta \lor \varepsilon) \land (\gamma \lor \varepsilon) \land (\alpha \lor \delta) \land (\beta \lor \delta) \land (\gamma \lor \delta)\)
**Depth first search solver**

Reminder of conjunctive normal form: 

$$(A \lor B) \land (A \lor \neg C \lor D) \land (C \lor \neg B) \land (B)$$

**function** `DFSS(clauses, symbols, partmodel={})` **returns** true or false

**if** every clause in `clauses` is true in `partmodel` **then return** true

**if** some clause in `clauses` is false in `partmodel` **then return** false

---

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

**return** or( `DFSS(clauses, rest, partmodelU{P=true})`, `DFSS(clauses, rest, partmodelU{P=false})` )
Efficient SAT solvers

- **DPLL (Davis-Putnam-Logemann-Loveland)** is the core of modern solvers
- Recursive depth-first search over partial 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 \(\{A=true\}\)
    - any clause is falsified; e.g., \((A \lor B) \land (A \lor \neg C)\) is falsified if \(\{A=false, B=false\}\)
  - **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 positive in every clause of \((A \lor B) \land (A \lor \neg C) \land (C \lor \neg B)\) so set it to true
  - **Unit clauses**: if clause has one unresolved literal, set symbol to satisfy clause
    - E.g., if A=false, \((A \lor B) \land (\neg B \lor \neg C)\) becomes \((false \lor B) \land (\neg B \lor \neg C)\), so set B=true
    - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.
function DPLL(clauses, symbols, partmodel={}) returns true or false

if every clause in clauses is true in partmodel then return true

if some clause in clauses is false in partmodel then return false

clauses ← every clause in clauses that is not true (or false)

P, value ← FIND-PURE-SYMBOL(symbols, clauses, partmodel)

if P is non-null then return DPLL(clauses, symbols−P, partmodelU{P=value})

P, value ← FIND-UNIT-CLAUSE(clauses, partmodel)

if P is non-null then return DPLL(clauses, symbols−P, partmodelU{P=value})

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

return or(DPLL(clauses, rest, partmodelU{P=true}), DPLL(clauses, rest, partmodelU{P=false}))
DPLL Example

- Start with \((A \lor B) \land (\neg A \lor \neg C) \land (C \lor \neg B) \land (A \lor B \lor C)\)
- No pure symbols or unit clauses
  - \((T \lor B) \land (\neg T \lor \neg C) \land (C \lor \neg B) \land (T \lor B \lor C)\)
  - Remove satisfied clauses
    - \((\neg T \lor \neg C) \land (C \lor \neg B)\)
  - \(B\) is a pure symbol – set it to false
    - \((\neg T \lor \neg C) \land (C \lor \neg F)\)
  - Remove satisfied clauses
    - \((\neg T \lor \neg C)\)
  - \(C\) is a pure symbol (also a unit clause btw because the clause has one unresolved literal)
    - \((\neg T \lor \neg F)\)
  - All clauses satisfied!

Question: what partial model are we dealing with at this point? (What is a partial model?)
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 in positive / negative form
    - 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 ~100,000,000 variables
SAT solvers in practice

- 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???
### Resolution (briefly)

- **Every CNF clause can be written as**
  - Conjunction of symbols $\Rightarrow$ disjunction of symbols
  - $A \lor B \lor \neg C \lor \neg D = C \land D \Rightarrow A \lor B$

- **The resolution inference rule takes two such clauses and infers a new one by resolving complementary symbols:**

- **Example:**
  
  \[
  A \land B \land C \Rightarrow U \lor V \\
  D \land E \land U \Rightarrow X \lor Y
  \]
  
  \[
  \underbrace{A \land B \land C \land D \land E} \Rightarrow V \lor X \lor Y
  \]

- **Sentence unsatisfiable iff repeated resolution produces $() \Rightarrow ()$**

- **Resolution is complete for propositional logic, but exp-time**
A knowledge-based agent

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
Planning as satisfiability

- Given a hyper-efficient SAT solver, can we use it to make plans?
- Yes, for **fully observable, deterministic** case
- For $T = 1$ to $\infty$,
  - Initialize the KB with **PacPhysics** for $T$ time steps
  - Assert goal is true at time $T$
- Planning problem is solvable iff there is some satisfying assignment
- Solution obtained from truth values of action variables
- Read off action variables from SAT-solver solution
Basic PacPhysics for Planning

- **Map**: where the walls are and aren’t, where the food is and isn’t
- **Initial state**: Pacman start location (exactly one place), ghosts
- **Actions**: Pacman does exactly one action at each step
- **Transition model**:
  - `<at x,y_t> ⇔ [at x,y_t-1 and stayed put] v [next to x,y_t-1 and moved to x,y]`
  - `<food x,y_t> ⇔ [food x,y_t-1 and not eaten]`
  - `<ghost_B x,y_t> ⇔ [.....]`
- **Assertion of goal attainment**: Pacman achieves the goal by time T (not really “physics”)
Reminder: Partially observable Pacman

- Basic question: where am I?
- 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, ...

- Sensor model:
  - Blocked_W_0 \iff ((At_1,1_0 \land Wall_0,1) \lor (At_1,2_0 \land Wall_0,2) \lor (At_1,3_0 \land Wall_0,3) \lor ...)

- Map: where are the walls
- Initial state: Pacman definitely somewhere
- Domain constraints: e.g. only one action per timestep
- Transition model: how state variables change (or don’t)
State estimation

- **State estimation** means keeping track of what’s true now

- A logical agent can just ask itself!
  - E.g., ask whether \( \text{KB} \land <\text{actions}> \land <\text{percepts}> \models \text{At}_2,2_6 \)

- 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 \( \text{KB} \land \text{action}_{t-1} \land \text{percept}_t \models X_t \), add \( X_t \) to KB
      - If \( \text{KB} \land \text{action}_{t-1} \land \text{percept}_t \models \neg X_t \), 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 $KB \land \text{action}_{t-1} \land \text{percept}_t \models At_{x,y_t}$, add $At_{x,y_t}$ to KB
      - If $KB \land \text{action}_{t-1} \land \text{percept}_t \models \neg At_{x,y_t}$, 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
- Percept
Localization demo

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

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

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

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

- Percept
- Action \textit{WEST}
- Percept
- Action \textit{WEST}
- Percept
- Action \textit{WEST}
- Percept
Localization with random movement
Is the eager method enough for accurate state estimation?

No! There can be cases where neither $X_t$ nor $\neg X_t$ is entailed, and neither $Y_t$ nor $\neg Y_t$ is entailed, but some constraint, e.g., $X_t \lor Y_t$, is entailed.

E.g., the study at time $t$ was flawed or meat causes cancer.

Exact state estimation is intractable in general.

Requires keeping track of properties of combinations of state variables.
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 \texttt{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 Wall$_{x,y}$ or $\neg$Wall$_{x,y}$ to KB, if entailed
    - For each $x,y$, add either At$_{x,y\_t}$ or $\neg$At$_{x,y\_t}$ to KB, if entailed
    - Choose an action
Summary

- 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