Warm-up:

The regions below visually enclose the set of models that satisfy the respective sentence $\gamma$ or $\delta$. For which of the following diagrams does $\gamma$ entail $\delta$. Select all that apply.

A)  
B)  
C)  
D)  
E)
Announcements

Midterm 1 Exam
- Grading should be finished tomorrow night
- Then we’ll let you know as soon as Canvas reflects your current grade

Assignments:
- P2: Optimization
  - Due Thu 2/21, 10 pm
- HW5
  - Out later tonight
Announcements

Index card feedback

- Thanks!
- Piazza with some responses tonight

Alita Class Field Trip!

- Saturday, 2/23, afternoon
- Details will be posted on Piazza
AI: Representation and Problem Solving

Logical Agents

Instructors: Pat Virtue & Stephanie Rosenthal

Slide credits: CMU AI, http://ai.berkeley.edu
The regions below visually enclose the set of models that satisfy the respective sentence $\gamma$ or $\delta$. For which of the following diagrams does $\gamma$ entail $\delta$. Select all that apply.
What about intersection feasible regions?

The regions below visually enclose the set of points that satisfy the respective constraints \( \gamma \) or \( \delta \). For which of the following diagrams is a solution point for \( \gamma \) guaranteed to be feasible in \( \delta \). Select all that apply.

A)  
B)  
C)  
D)  
E)
Piazza Poll 1

The regions below visually enclose the set of models that satisfy the respective sentence $\gamma$ or $\delta$. For which of the following diagrams does $\gamma$ entail $\delta$. Select all that apply.

A) $\gamma \cap \delta$
B) $\gamma \cup \delta$
C) $\gamma \cup \delta$
D) $\gamma \subset \delta$
E) $\delta \subset \gamma$
Entailment

Does the knowledge base entail my query?

- Query 1: $\neg P[1,2]$
- Query 2: $\neg P[2,2]$
Logical Agent Vocab

**Model**
- Complete assignment of symbols to True/False

**Sentence**
- Logical statement
- Composition of logic symbols and operators

**KB**
- Collection of sentences representing facts and rules we know about the world

**Query**
- Sentence we want to know if it is *probably* True, *provably* False, or *unsure.*
Logical Agent Vocab

Entailment

- Input: sentence1, sentence2
- Each model that satisfies sentence1 must also satisfy sentence2
- "If I know 1 holds, then I know 2 holds"
- (ASK), TT-ENTAILS, FC-ENTAILS

Satisfy

- Input: model, sentence
- Is this sentence true in this model?
- Does this model satisfy this sentence
- "Does this particular state of the world work?"
- PL-TRUE
Logical Agent Vocab

**Satisfiable**
- Input: *sentence*
- Can find at least one model that satisfies this *sentence*
  - (We often want to know what that model is)
- "Is it possible to make this *sentence* true?"
- **DPLL**

**Valid**
- Input: *sentence*
- *sentence* is true in all possible models
Propositional Logical Vocab

Literal
- Atomic sentence: True, False, Symbol, \( \neg \)Symbol

Clause
- Disjunction of literals: \( A \lor B \lor \neg C \)

Definite clause
- Disjunction of literals, *exactly one* is positive
- \( \neg A \lor B \lor \neg C \)

Horn clause
- Disjunction of literals, *at most one* is positive
- All definite clauses are Horn clauses
Entailment

How do we implement a logical agent that proves entailment?

- Logic language
  - Propositional logic
  - First order logic

- Inference algorithms
  - Theorem proving
  - Model checking
Propositional Logic

Check if sentence is true in given model
In other words, does the model satisfy the sentence?

function PL-TRUE?(α,model) returns true or false
    if α is a symbol then return Lookup(α, model)
    if Op(α) = \( \neg \) then return not(PL-TRUE?(Arg1(α),model))
    if Op(α) = \( \wedge \) then return and(PL-TRUE?(Arg1(α),model),
                                      PL-TRUE?(Arg2(α),model))
    etc.

(Sometimes called “recursion over syntax”)
Simple Model Checking

function TT-ENTAILS?(KB, α) returns true or false
Simple Model Checking, contd.

Same recursion as backtracking
O($2^n$) time, linear space
We can do much better!

\[
P_1 = \text{true} \quad P_1 = \text{false} \\
P_2 = \text{true} \quad P_2 = \text{false} \\
P_n = \text{false} \quad P_n = \text{true}
\]
Piazza Poll 2

Which would you choose?

- DFS
- BFS
Simple Model Checking

function $\text{TT-ENTAILS?}(KB, \alpha)$ returns true or false
  return $\text{TT-CHECK-ALL}(KB, \alpha, \text{symbols}(KB) \cup \text{symbols}(\alpha),\{\})$

function $\text{TT-CHECK-ALL}(KB, \alpha, \text{symbols},\text{model})$ returns true or false
  if empty?(symbols) then
    if $\text{PL-TRUE?}(KB, \text{model})$ then return $\text{PL-TRUE?}(\alpha, \text{model})$
    else return true
  else
    $P \leftarrow \text{first}(\text{symbols})$
    rest $\leftarrow \text{rest}(\text{symbols})$
    return $\text{and} (\text{TT-CHECK-ALL}(KB, \alpha, \text{rest}, \text{model} \cup \{P = \text{true}\})$
      $\text{TT-CHECK-ALL}(KB, \alpha, \text{rest}, \text{model} \cup \{P = \text{false}\}))$
Inference: Proofs

A proof is a *demonstration* of entailment between $\alpha$ and $\beta$

Method 1: *model-checking*
- For every possible world, if $\alpha$ is true make sure that is $\beta$ true too
- OK for propositional logic (finitely many worlds); not easy for first-order logic

Method 2: *theorem-proving*
- Search for a sequence of proof steps (applications of *inference rules*) leading from $\alpha$ to $\beta$
- E.g., from $P \land (P \Rightarrow Q)$, infer $Q$ by *Modus Ponens*

Properties
- *Sound* algorithm: everything it claims to prove is in fact entailed
- *Complete* algorithm: every sentence that is entailed can be proved
Simple Theorem Proving: Forward Chaining

Forward chaining applies Modus Ponens to generate new facts:

- Given $X_1 \land X_2 \land ... \land X_n \Rightarrow Y$ and $X_1, X_2, ..., X_n$
- 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 $\text{True} \Rightarrow X$)
### Forward Chaining Algorithm

**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`

<table>
<thead>
<tr>
<th><strong>Clauses</strong></th>
<th><strong>Count</strong></th>
<th><strong>Inferred</strong></th>
<th><strong>Agenda</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>P ⇒ Q</code></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>L ∧ M ⇒ P</code></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>B ∧ L ⇒ M</code></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>A ∧ P ⇒ L</code></td>
<td>2</td>
<td>A false</td>
<td></td>
</tr>
<tr>
<td><code>A ∧ B ⇒ L</code></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>A</code></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>B</code></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Forward Chaining Example: Proving Q

**Clauses**

- P ⇒ Q
- L ∧ M ⇒ P
- B ∧ L ⇒ M
- A ∧ P ⇒ L
- A ∧ B ⇒ L
- A
- B

**Count**

- 1/0
- 2/1/0
- 2/1/0
- 2/1/0
- 2/1/0
- 0
- 0

**Inferred**

- A: false, true
- B: false, true
- L: false, true
- M: false, true
- P: false, true
- Q: false, true

**Agenda**

```
A B M L P Q
```
### Forward Chaining Algorithm

**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 inferred table as a model \( m \), assigning true/false to symbols
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 \( m \)
Satisfiability and Entailment

A sentence is *satisfiable* if it is true in at least one world (cf CSPs!)

Suppose we have a hyper-efficient SAT solver; how can we use it to test entailment?

- Suppose $\alpha \models \beta$
- Then $\alpha \implies \beta$ is true in all worlds
- Hence $\neg(\alpha \implies \beta)$ is false in all worlds
- Hence $\alpha \land \neg\beta$ 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:

- \( \text{At}_1,1_0 \Rightarrow (\text{Wall}_0,1 \Leftrightarrow \text{Blocked}_W_0) \)
- \( \text{At}_1,1_0 \Rightarrow ((\text{Wall}_0,1 \Rightarrow \text{Blocked}_W_0) \land (\text{Blocked}_W_0 \Rightarrow \text{Wall}_0,1)) \)
- \( \neg \text{At}_1,1_0 \lor ((\neg \text{Wall}_0,1 \lor \text{Blocked}_W_0) \land (\neg \text{Blocked}_W_0 \lor \text{Wall}_0,1)) \)
- \( (\neg \text{At}_1,1_0 \lor \neg \text{Wall}_0,1 \lor \text{Blocked}_W_0) \land (\neg \text{At}_1,1_0 \lor \neg \text{Blocked}_W_0 \lor \text{Wall}_0,1) \)

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

Essentially a backtracking 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=true\}
  - any clause is falsified; e.g., \((A \lor B) \land (A \lor \neg C)\) is satisfied by \{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 pure and positive in \((A \lor B) \land (A \lor \neg C) \land (C \lor \neg B)\) so set it to true

- **Unit clauses**: if a clause is left with a single literal, set symbol to satisfy clause
  - E.g., if \(A=false\), \((A \lor B) \land (A \lor \neg C)\) becomes \((false \lor B) \land (false \lor \neg C)\), i.e. \((B) \land (\neg C)\)
  - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.
DPLL algorithm

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, modelU{P=value})

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

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

  return or(DPLL(clauses, rest, modelU{P=true}),
             DPLL(clauses, rest, modelU{P=false}))
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 for actions etc.
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 for actions etc.

For $T = 1$ to infinity, set up the KB as follows and run SAT solver:
- Initial state, domain constraints
- Transition model sentences up to time $T$
- Goal is true at time $T$
- **Precondition axioms**: $\text{At}_{1,1} \land \text{N}_0 \Rightarrow \neg \text{Wall}_{1,2}$ etc.
- **Action exclusion axioms**: $\neg(\text{N}_0 \land \text{W}_0) \land \neg(\text{N}_0 \land \text{S}_0) \land ..$ etc.
Initial State

The agent may know its initial location:

- $\text{At}_1,1_0$

Or, it may not:

- $\text{At}_1,1_0 \lor \text{At}_1,2_0 \lor \text{At}_1,3_0 \lor \ldots \lor \text{At}_3,3_0$

We also need a domain constraint – cannot be in two places at once!

- $\neg (\text{At}_1,1_0 \land \text{At}_1,2_0) \land \neg (\text{At}_1,1_0 \land \text{At}_1,3_0) \land \ldots$
- $\neg (\text{At}_1,1_1 \land \text{At}_1,2_1) \land \neg (\text{At}_1,1_1 \land \text{At}_1,3_1) \land \ldots$
- $\ldots$
Transition Model

How does each *state variable* or *fluent* at each time gets its value?

State variables for PL Pacman are $At_{x,y_t}$, e.g., $At_{3,3_{17}}$

A state variable gets its value according to a *successor-state axiom*

- $X_t \iff [X_{t-1} \land \neg \text{(some action}_{t-1} \text{ made it false)}] \lor \neg X_{t-1} \land \text{(some action}_{t-1} \text{ made it true)}$

For Pacman location:

- $At_{3,3_{17}} \iff [At_{3,3_{16}} \land \neg \text{((\neg Wall}_{3,4} \land N_{16}) \lor \neg Wall_{4,3} \land E_{16}) \lor \ldots] \lor \neg At_{3,3_{16}} \land ((At_{3,2_{16}} \land \neg Wall_{3,3} \land N_{16}) \lor (At_{2,3_{16}} \land \neg Wall_{3,3} \land N_{16}) \lor \ldots)]$