Logical agents

Two key components
• Representation of the world
  • Background information
  • Percepts
• Ability to reason:
  Derive new information based on inference
Knowledge base (KB)

- Sentence – A statement about the agent’s world
  - Based on sensors
    - Danger around corner.
    - Range to target is 50 m.
    - Day is cloudy.
  - Based on background knowledge (or possibly learning)
    - Solar cells lose efficiency on cloudy days.
    - Ascending inclines requires more power.

Knowledge base (KB) operations

Agent interacts with KB

- Tell – Agent informs KB of perceptions
  - Adds new timestamped sentences to the KB

- Ask – Agent asks KB which action should be taken.
  - Reasoning occurs here.

- Tell – Agent tells KB that the action performed.
KB operations

We will use the following functions to build sentences:

• Make-Percept-Sentence – Construct a sentence from a percept.

• Make-Action-Query – Construct sentence that asks which action should be done at current time.

• Make-Action-Sentence – Construct sentence indicating that the specified action was executed at time t.

KB-agent

class KBagent:
    KB # knowledge base
    t = 0

    def getaction(percept):
        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 # caller performs action

    t could be a real timestamp instead of a counter
Beware the wumpus

• Early computer game
  • Maze of dark caverns with
    • Pits – too dark, you might fall in
    • Wumpus – Sedentary beast. Does not move but it will eat you alive if you stumble into it...
  • Pot of gold
• Your goal
  • Armed with a bow and single arrow, climb down into the caverns, grab the gold and get out without falling into a pit or being gobbled up by the wumpus.
  • Try not to shoot the wumpus, he’s an apex predator and good for the environment.

Beware the wumpus

• Actions
  • Move forward
  • Turn left
  • Turn right
  • Shoot (can only do this once) – Arrow flies in the direction you fire until it hits a wall or kills the wumpus.
  • Grab – Grab the gold if it is in your current position.
  • Climb – Climb out of the case, only valid in starting cave.
Sensors

- Chemical – Wumpus emits a *stench* that can be perceived in the next cavern
- Tactile
  - *breeze* is perceived when next to a pit
  - *bump* when agent walks into a wall
- Visual – All that glitters is gold, agent perceives *glitter* when in the same room as the gold.
- Auditory – The wumpus is usually pretty quiet, but emits a blood curdling *scream* when it dies.

Beware the wumpus

- Environment
  - 4x4 grid of caverns*
  - Grids other than the start square
    - P(pit) = .2
    - Wumpus placed randomly in non-pit, non-start cave
  - Agent starts in x=1, y=1 facing such that they move positively along the y axis.
  - Some environments are unfair (e.g. gold in a pit or surrounded by pits ~ 21% of time)

*Written by Gregory Yob in the early 1970s, the cave topology was based on a dodecahedron.*
Welcome to wumpus world

Initial knowledge base

• Basic knowledge of environment (e.g. perceived a breeze next to a pit.
• Current location [1,1] is safe

• First percept:
  [No Stench, No Breeze, No Glitter, No Bump, No Scream]

• What can we learn?
[No Stench, No Breeze, No Glitter, No Bump, No Scream]

Agent moves to 2,1

2,1 Percepts
No Stench
Breeze
No Glitter
No Bump
No Scream

White caves – visited
Light gray caves – surrounding visited ones
Dark gray caves – The great unknown
Danger approaches...

A - agent position
OK – safe
B – breeze
P? – possible pit

2,1 Percepts
No Stench
Breeze
No Glitter
No Bump
No Scream

Knowledge base is extended based on percept breeze

Enumeration of instances
Three possible worlds
Playing it safe

• We don’t know if it is safe to move into 3,1 or 2,2
• Move back to 1,2

• What can we learn based on our rules?

Logical deduction

We used rules to infer something that percepts did not tell us.

Tricky, tricky…
A little formality

• Models
  • Representation of world
  • Assignment of values to variables
  • All possible worlds consists of all possible assignments to
    model variables.
  • The knowledge base is some subset of the possible
    worlds.
• Sentences
  • Require syntax
  • Have semantics with respect to models.
    In most logical models: True/False
    e.g. There is a pit at 3,1

Satisfaction and entailment

• Suppose \( \alpha \) is true in model \( M \), then we state:
  \( M \) satisfies \( \alpha \) or equivalent
  or: \( M \) is a model of \( \alpha \)
• \( M(\alpha) \) means all models that satisfy \( \alpha \).

• Reasoning – entailment
  \( \alpha \models \beta \) if and only if \( M(\alpha) \subseteq M(\beta) \)
  \( \alpha \) is a stronger assertion* than \( \beta \); there may be more
    worlds associated with \( \beta \)
  That is: \( \beta \) logically follows from \( \alpha \) if \( \alpha \not\models \beta \)

*stronger can be interpreted as tighter here
Examples of entailment

- House is cornflower blue $\models$ house is a shade of blue

- $x = 0 \models xy = 0$
  
  $x = 0 \cdot 1 = 0$
  
  $x = 0 \cdot 2 = 0$
  
  $x = 0 \cdot 3 = 0$
  
  $x = 0 \cdot 4 = 0$

Return to wumpus world

$\alpha_1 = \text{No pit at } [1, 2]$

$\alpha_2 = \text{No pit at } [2, 2]$

Does $KB \models \alpha_1$? $KB \models \alpha_2$?
Inference algorithms...

- Are sound if inference only derives entailed sentences
  
  If our algorithm entailed $\text{KB} \models \alpha_2$, we might fall into a pit!

  *She’s a witch* – not very sound...

- Are complete if they can derive any sentence that is entailed.
  
  Becomes an issue when the left-hand side is infinite.

Inference algorithms

If our algorithm is sound, then our entailments are correct, but...

what connection does this have with reality?

*Grounding* is the correspondence between model and reality.

If the KB is well grounded, our entailments *should* follow in the real world.
Sentence construction: Propositional Logic

Propositional logic is for the most part the logic you learned to program with:
Sentence → AtomicSentence | ComplexSentence
AtomicSentence → true | false | Literal
ComplexSentence → (Sentence) | [Sentence]
  | → Sentence  negation
  | Sentence ^ Sentence  conjunction
  | Sentence v Sentence  disjunction
  | Sentence ⇒ Sentence  implication
  | Sentence ⇔ Sentence  biconditional

operator priority of logical connectives is shown by order, e.g. (A v B ^ C) implies (A v (B ^ C)), literal is a propositional symbol e.g. A

Sentence semantics

• Sentences are reduced to true or false with respect to a specific model.
• In the Wumpus cave
  • We might denote the presence or absence of a pit by literals indexed by location: P_{x,y}
  • Example: P_{1,2}, P_{2,2}, and P_{3,1} that have true/false values in any given model.
• Models must specify values for each proposition.
• To resolve a sentence: apply logical connectives to truth values (see Fig. 7.8 for truth tables)
Knowledge base rules

In the wumpus cave,
• denote pit & wumpus presence/absence by $P_{i,j}$ and $W_{i,j}$
• there is a breeze if and only if there is a neighboring pit:
  \[ B_{2,2} \iff P_{2,1} \lor P_{3,3} \lor P_{1,1} \lor P_{3,3} \]
  \[ B_{1,1} \iff P_{2,2} \lor P_{2,1} \]
• there is a stench iff the Wumpus is in the next cavern
  \[ S_{2,2} \iff W_{2,1} \lor W_{2,3} \lor W_{1,2} \lor W_{3,2} \]
  \[ S_{1,1} \iff W_{1,2} \lor W_{2,1} \]

Stench and breeze sentences are true in all wumpus worlds.

Percepts

• There is no pit at 1,1
  $\neg P_{1,1}$
• There is no breeze at 1,1
  $\neg B_{1,1}$
• There is a breeze at 2,1
  $B_{2,1}$
Simple entailment algorithm

def TruthTable-Entails(KB, α):
    symbols = proposition-symbols in KB and α
    return TT-Check-All(KB, α, symbols, {})

def TT-Check-All(KB, α, symbols, model):
    if empty(symbols):
        if pl-true(kb, model):  # Does the KB entail the model?
            return pl-true(α, model)  # Is α entail the model?
        else return True  # return true when KB does not hold
    else:
        # recursively enumerate the models
        (s, others) = (first(symbols), rest(symbols))
        return TT-Check-All(KB, α, rest, model U {s=True}) and
               TT-Check-All(KB, α, rest, model U {s=False})

Concrete example:

• Knowledge base:  ¬P ∨ Q, Q ⇒ M
• Does ¬P ∨ Q, Q ⇒ M ⊨ P ∨ Q ∨ M?

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>M</th>
<th>¬P ∨ Q</th>
<th>Q ⇒ M</th>
<th>P ∨ Q ∨ M</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Concrete example:

- **Knowledge base:** \( \neg P \lor Q \)
- **Does** \( \neg P \lor Q \models \neg (P \land \neg Q) \lor R \)?

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>( \neg P \lor Q )</th>
<th>( \neg (P \land \neg Q) \lor R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

```python
def TruthTable-Entails(KB=\( \neg P \lor Q \), Q, M, \( \alpha = P \lor Q \lor M \)):
    symbols = proposition-symbols in KB and \( \alpha \) # P, Q, M
    return TT-Check-All(KB, \( \alpha \), {P, Q, M}, {})

def TT-Check-All(KB, \( \alpha \), symbols, model):
    if empty(symbols):
        if pl-true(kb, model): # Does the KB entail the model?
            return pl-true(\( \alpha \), model) # Is \( \alpha \) entail the model?
        else return True # return true when KB does not hold
        else:
            # recursively enumerate the models
            (s, others) = (first(symbols), rest(symbols))
            return TT-Check-All(KB, \( \alpha \), rest, model U {s=True}) and
                   TT-Check-All(KB, \( \alpha \), rest, model U {s=False})

    TT-Check-All(KB, \( \alpha \), {Q, M}, {P=T})
    TT-Check-All(KB, \( \alpha \), {Q, M}, {P=F})
```

Eventually for every model...

- if \( \alpha \) entail the KB for \( P=T, Q=T, M=T \), return \( \alpha \) entail the KB for \( P=T, Q=T, M=T \)
- else return True
Simple entailment algorithm

Summary of model checking

• Recursively generates all possible combinations of truth values for every symbol.
• Checks if the knowledge base is a subset of a specific symbol value assignment
  • If so, returns if sentence $\alpha$ holds as well.
  • Otherwise returns true as we don’t care about whether $\alpha$ holds outside the KB (implies).

Can we prove things without enumerating everything?

or, a refresher course in theorem proving...

Concepts

• Logical equivalence:
  $\alpha \equiv \beta$ if they are true in the same set of models

  Alternative definition:
  Equivalent if they entail one another:

  $\alpha \models \beta \iff \beta \models \alpha$
Concepts

• Validity – Sentences are valid (called tautologies) if they are true in all models.
  e.g. \((a \land b) \lor \neg a \lor \neg b\)

• Deduction theorem
  For arbitrary sentences \(\alpha, \beta, \alpha \vdash \beta\) iff \((\alpha \Rightarrow \beta)\) is valid.

  So if we can show \((\alpha \Rightarrow \beta)\), is a tautology,
  we know \(\alpha \vdash \beta\)

Concepts

• Satisfiability – There exists some model such that a sentence is true.

• Sometimes, it is easier to show that something is valid by showing that its contradiction is not satisfiable:
  \(\alpha\) is valid iff \(\neg \alpha\) is not satisfiable
  (If no model satisfies \(\neg \alpha\), then \(\alpha\) must be true)

• which leads to:
  \(\alpha \vdash \beta\) iff \((\alpha \land \neg \beta)\) is not satisfiable
  Remember, \(\alpha \vdash \beta\) iff \(\alpha \Rightarrow \beta \equiv \neg \alpha \lor \beta\)
Inference rules

• Notation for rewriting logic

\[
\frac{\text{Sentence}_1, \text{Sentence}_2}{\text{Sentence}_3}
\]

means sentences 1 and 2 imply 3

• Rules
  
  • Modus ponens
    
    \[
    \frac{\alpha \Rightarrow \beta, \alpha}{\beta}
    \]
    
    breeze ⇒ pit in neighboring cavern, breeze
    
    pit in neighboring cavern

  • And elimination
    
    \[
    \frac{\alpha \wedge \beta}{\alpha}
    \]
    
    glitter ∧ breeze ∧ stench
    
    glitter

Notation and inference rules

• Rules
  
  • Biconditional elimination
    
    \[
    \frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}
    \]
    
    \[
    \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}
    \]
    
    Note that we can also write these as equivalences
    
    e.g. \( \alpha \Leftrightarrow \beta \equiv (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha) \)
Inference rules

- Commutativity, associativity, and distributivity for $\land$ and $\lor$
  
  \begin{align*}
  (\alpha \lor \beta) & \equiv (\beta \lor \alpha) \\
  (\alpha \land \beta) & \equiv (\beta \land \alpha) \\
  ((\alpha \lor \beta) \lor \gamma) & \equiv (\alpha \lor (\beta \lor \gamma)) \\
  ((\alpha \land \beta) \land \gamma) & \equiv (\alpha \land (\beta \land \gamma)) \\
  (\alpha \lor (\beta \land \gamma)) & \equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma)) \\
  (\alpha \land (\beta \lor \gamma)) & \equiv ((\alpha \land \beta) \lor (\alpha \land \gamma))
  \end{align*}

- Double-negation elimination
  
  \[ \lnot(\lnot \alpha) \equiv \alpha \]

- Contraposition
  
  \[ (\alpha \Rightarrow \beta) \equiv (\lnot \beta \Rightarrow \lnot \alpha) \]

- Implication elimination
  
  \[ (\alpha \Rightarrow \beta) \equiv (\lnot \alpha \lor \beta) \]

- DeMorgan’s rule
  
  \begin{align*}
  \lnot(\alpha \land \beta) & \equiv (\lnot \alpha \lor \lnot \beta) \\
  \lnot(\alpha \lor \beta) & \equiv (\lnot \alpha \land \lnot \beta)
  \end{align*}
Proof by hand

- Knowledge base
  \[ \neg P_{1,1}, \quad B_{1,1} \iff P_{1,2} \vee P_{2,1}, \quad B_{2,1} \iff P_{1,3} \vee P_{2,2} \vee P_{3,1} \]

- Percepts
  \[ \neg B_{1,1}, B_{2,1} \]

- Does the KB entail the lack of a pit at 1,2? \( KB \models (\alpha = \neg P_{1,2}) \)
  \[ B_{1,1} \iff P_{1,2} \vee P_{2,1} \] from KB
  \[ (B_{1,1} \Rightarrow P_{1,2} \vee P_{2,1}) \wedge (P_{1,2} \vee P_{2,1} \Rightarrow B_{1,1}) \] biconditional elim.
  \[ P_{1,2} \vee P_{2,1} \Rightarrow B_{1,1} \] and elimination
  \[ \neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}) \] contrapositive
  \[ \neg(P_{1,2} \vee P_{2,1}) \] from percept \( \neg B_{1,1} \)
  \[ \neg P_{1,2} \wedge \neg P_{2,1} \] DeMorgan's Rule
  \[ \neg P_{1,2} \] and elimination

Using inference

- Typically more efficient as we do not have to enumerate every possible value.

- Can be formulated as a search:
  - Initial State – Initial knowledge base
  - Actions – Inference rules applied to sentences
  - Results – Action result is the sentence rewritten by the inference rule
  - Goal – State containing the sentence we are trying to prove
Suppose we learn something new

• Suppose $KB \models \alpha$

• What if we learn $\beta$?

$$KB \land \beta \models \alpha$$

Propositional logic is a *monotonic* system, new knowledge will not change what is entailed; in other words: *propositional logic will not change its mind.*

Automated theorem proving

• Searching on logic rules works, but is complicated.
• Can we do something simpler?

• The resolution rule exploits clause pairs with $P_i$ and $\neg P_i$:

$$\frac{l_i \lor l_2 \lor \ldots \lor l_{i-1} \lor l_k \lor m_1 \lor m_2 \lor \ldots \lor m_j \lor \ldots \lor m_u}{l_1 \lor l_2 \lor \ldots \lor l_{i-1} \lor l_{i+1} \lor \ldots \lor l_k \lor m_1 \lor m_2 \lor \ldots \lor m_j \lor \ldots \lor m_u}$$

where $l_i \equiv \neg m_j$

example:

$$\frac{P_{1,1} \lor P_{3,1}, \neg P_{1,1} \lor \neg P_{2,2}}{P_{3,1} \lor \neg P_{2,2}}$$
Automated theorem proving

The resolution rule can produce duplicates:

\[
\frac{A \lor B, A \lor \neg B}{A \lor A}
\]

so we remove these by \textit{factoring}

\[
\frac{A \lor B, A \lor \neg B}{A}
\]

We will assume that all results are factored.

Conjunctive normal form

All propositional logic sentences can be transformed to conjunctions of disjunctive clauses

Remove implications:

\[
\begin{align*}
\alpha \iff \beta &\rightarrow (\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha) \\
\alpha \Rightarrow \beta &\rightarrow \neg \alpha \lor \beta
\end{align*}
\]

Associate negations with literals only

\[
\begin{align*}
\neg (\neg \alpha) &\rightarrow \alpha \\
\neg (\alpha \lor \beta) &\rightarrow \neg \alpha \land \neg \beta \\
\neg (\alpha \land \beta) &\Rightarrow \neg \alpha \lor \neg \beta
\end{align*}
\]
Breeze/Pit Example

\[ B_{1,1} \iff (P_{1,2} \lor P_{2,1}) \]
\[
(B_{1,1} \Rightarrow (P_{1,2} \lor P_{2,1})) \land ((P_{1,2} \lor P_{2,1}) \Rightarrow B_{1,1})
\]
\[
(\neg B_{1,1} \lor (P_{1,2} \lor P_{2,1})) \land ((\neg P_{1,2} \lor P_{2,1}) \lor B_{1,1})
\]
\[
(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land ((\neg P_{1,2} \land \neg P_{2,1}) \lor B_{1,1})
\]
\[
(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land ((\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1}))
\]

Resolution algorithm

• Permits us to show KB \models \alpha using only the resolution rule.
• All sentences must be in conjunctive normal form
• Resolution algorithm
  • Lets us show that something is satisfiable, that a model exists where a proposition is true
  • We want to show KB \models \alpha, but the resolution algorithm only shows us satisfiability.
  • Fortunately, KB \models \alpha \iff KB \land \neg \alpha so if we cannot satisfy \alpha, we have a proof by contradiction.
Resolution algorithm

```python
def PL-Resolution(KB, α):
    clauses = set of conjunctive normal form clauses of $KB \land \neg \alpha$
    new = {}
    while True:
        for each distinct pair with complementary literals $C_i$, $C_j$ ∈ clauses:
            resolvents = PL-Resolve($C_i$, $C_j$)
            if $\emptyset$ ∈ resolvents, return True # could not satisfy, contradiction
            new = new U resolvents
        if new ⊆ clauses, return False   # everything satisfied
        clauses = clauses U new
```

There is no pit $\alpha = \neg P_{1,2}$

Applications of the resolution rule, Fig. 7.13
Is the resolution theorem complete?

RC(S) – The resolution closure of S, is the set of all clauses derivable by repeated application of the resolution clause to S and its derivatives.

The ground resolution theorem states:
clauses S are unsatisfiable $\rightarrow \varepsilon \in RC(S)$

Completeness of ground-resolution theorem

If the conditional:
S is unsatisfiable $\rightarrow \varepsilon \in RC(S)$
is true, then its contrapositive must also be true
$\varepsilon \notin RC(S) \rightarrow S$ is satisfiable

Assume that we have a set, RC(S), that does not contain the empty clause. Can we construct a model (values for proposition symbols) that is true?

Let’s try...
Ground theorem completeness

There are a finite number of proposition symbols in RC(S). For convenience, rename to: \( P_1, P_2, P_3, \ldots, P_k \)

For \( i = 1 \) to \( k \)

\[
P_i = \begin{cases} 
\text{false} & \text{when some clause in } RC(S) \text{ contains } \neg P_i \\
\text{true} & \text{otherwise} 
\end{cases}
\]

and literals \( P_i, P_2, \ldots, P_{i-1} \) all evaluate to false

Ground theorem completeness

\[ RC(S) = \{ \neg P_3 \lor P_1, \neg P_1 \lor P_2 \lor P_3, P_2 \lor P_1, \neg P_1, P_2 \} \]

\( \ldots \) other clauses due to resolution

- Example construction

(This is only to paint broad strokes, it is based on the clauses shown above which are incomplete.)

\( P_1 = \text{false as } \neg P_1 \)

\( P_2 = \text{true} \) No \( \neg P_2 \) that requires \( \text{false} \) for clause to be true over \( i = 1, 2 \)

\( P_3 = \text{false} \) \( \neg P_3 \lor P_1 \)
Ground theorem completeness

• Suppose RC(S) was not satisfiable
• At some step i, we could no longer satisfy the first i propositions.
• Up to this point, we were fine, so any $P_{1..i-1}$ in the clause with $P_i$ must have evaluated to false
  • If $P_i$, we would assign true
  • If $\neg P_i$, would assign false
• Either case would be fine and we would proceed as normal.

Hence there must be clauses of the form

$$false \lor false \lor \ldots \lor P_i$$

and

$$false \lor false \lor \ldots \lor \neg P_i$$

which are not satisfiable.

• But we know

$$\alpha \lor P_i$$

and

$$\beta \lor \neg P_i$$

resolve to $\alpha \lor \beta$

which must be in RC(S), hence the failure would have had to occur earlier than i. So we must be able to satisfy S.
Restricted conjunctive normal form clauses

Full power of resolution is not always needed

\[ \neg B_{1,1} \lor P_{2,2} \lor P_{2,3} \]

CNF Clauses

\[ \neg L_{1,1} \lor \neg \text{Breeze} \lor B_{1,1} \]

Definite clause
Exactly one positive literal

Horn clause
At most one literal positive
\[ \neg P_{1,2} \lor \neg P_{2,3} \]

 Restricted CNF clauses

- Definite and horn clauses are interesting as there are more efficient resolution techniques.
- Many problems can be posed as Horn or definite clause problems:
  \[ \neg L_{1,1} \lor \neg \text{Breeze} \lor B_{1,1} \equiv (L_{1,1} \land \text{Breeze}) \Rightarrow B_{1,1} \]
- Horn clauses are usually written as implications and have names for clause parts.

\[ \text{Breeze} \]
\[ \frac{\text{fact}}{} \]
\[ \frac{(L_{1,1} \land \text{Breeze}) \Rightarrow B_{1,1}}{\text{body} \quad \text{head}} \]
\[ \equiv \text{True} \Rightarrow \text{Breeze} \]
Restricted CNF Clauses

• A Horn clause with no positive literal is called a goal clause.
• Efficient inference with Horn clauses is done through chaining.
  • forward-chaining – Data-driven, use known precepts to drive resolution of graph constructed from Horn clauses.
  • backward-chaining – Query driven, search and-or graph for goal.

• The language Prolog is a Horn clause processing language.

Horn clauses as graphs
Forward chaining

Prove L from A & B, continue propagating
Algorithm on p. 258, but you need only understand this intuitively.

Backward chaining

Goal directed reasoning – start with goal
Usually faster...

Start with query, e.g. Q and perform AND-OR search
Searching based on propositional model checks

- Recall $\alpha \models \beta \iff (\alpha \Rightarrow \beta \equiv \alpha \land \neg \beta)$
  so we work to satisfy $\neg \beta$ with respect to some knowledge base $\alpha$.

- In effect, this is a constraint satisfaction (SAT) problem and many of the techniques used for CSPs can be applied here. These can be applied to all CNF clauses:
  - Backtracking based solutions
  - Minimum conflict search solutions

Backtrack search

- Complete algorithm given in text (not responsible as similar in spirit to CSP backtrackers).
- General strategies for backtrack search
  - Components – When possible, partition into subproblems.
  - Variable/value ordering
  - Intelligent backtracking (e.g. conflict directed backtracking)
  - random restarts
  - Indexing to quickly find clauses with variables is highly useful.
Minconflict Local Search

• Same ideas as for CSPs
• Assign all variables and then attempt modifications until
  • goal is satisfied or
  • maximum number of steps has been reached

• Drawbacks
  • Failure to find solution does not mean one does not exist.
  • If steps set to $\infty$ and a solution does not exist, algorithm will never return...

as a consequence, usually used when we know there’s a solution

under- and over-constrained

• Under-constrained logic problems have lots of solutions and we can solve them quickly.
• Over-constrained logic problems have no solution.
• The hard ones are in between the two...
  as the ratio of clauses to available symbols increases, there comes a point where it becomes more difficult.
Putting it all together
Logical Agents

• Begin with knowledge base.
For the Wumpus world
• big long list of pits and breezes:
  \[ B_{1,1} \iff P_{1,1} \lor P_{2,1} \]
  \[ S_{1,1} \iff W_{1,2} \lor W_{2,1} \]
  \[ B_{2,2} \iff P_{2,1} \lor P_{2,3} \lor P_{1,2} \lor P_{3,2} \]
  \[ S_{2,2} \iff W_{2,1} \lor W_{2,3} \lor W_{1,2} \lor W_{3,2} \]
  \( \ldots \)
• what we know about starting square and that we have an arrow
  \[ \neg W_{1,1} \land \neg P_{1,1} \land \text{HaveArrow} \]

Logical Agents

• Wumpus world continued
• what we know about the wumpus:
  • At least one
    \[ W_{1,1} \lor W_{1,2} \lor W_{1,3} \lor W_{1,4} \lor W_{2,3} \lor \ldots \lor W_{3,4} \]
  • No more than one – for every pair of locations at least one of them must not have a wumpus
    \[ \neg W_{1,1} \lor \neg W_{1,2} \]
    \[ \neg W_{1,1} \lor \neg W_{1,3} \]
    \[ \neg W_{1,1} \lor \neg W_{1,4} \]
    \[ \neg W_{1,1} \lor \neg W_{2,1} \]
    \[ \neg W_{1,1} \lor \neg W_{2,2} \]
    \( \ldots \)
    \[ \neg W_{4,4} \lor \neg W_{4,1} \]
Percepts

• How do we incorporate percepts?
  • We might observe Stench at time 6, but not at time 7.
  • Remember, KB cannot change its mind! PL is monotonic.
  • We can modify MAKE-PERCEPT-SENTENCE to create distinct literals with time names: $Stench^6 \neg Stench^7$

Percepts

• We now have two types of variables:
  • Timestamped variables called fluents, and
  • atemporal variables that do not vary with time.

Should HaveArrow be a fluent or atemporal variable?
Actions and transition models

- Suppose agent initially at 1,1 facing east. An effect axiom could be added to model moving forward:

\[ L_{1,1}^0 \land \text{FacingEast}^0 \land \text{Forward}^0 \Rightarrow (L_{2,1}^1 \land \neg L_{1,1}^1) \]

- Other effect axioms could be added for:
  - grab, shoot, climb, turnleft, turnright.

Forward young gold hunter

So we move forward...

\[ L_{1,1}^0 \land \text{FacingEast}^0 \land \text{Forward}^0 \Rightarrow (L_{2,1}^1 \land \neg L_{1,1}^1) \]

the effect axiom lets us determine that we are now in L_{2,1}:

\[ \text{Ask}(KB, L_{2,1}^1) = \text{True} \]

What about?

\[ \text{Ask}(KB, \text{HaveArrow}^1) \]
The Frame Problem

Nothing made HaveArrow\(^1\) true...
the effects axiom only handled what changed.

We could add frame axioms for each time \(t\):

\[
\begin{align*}
\text{Forward}^0 & \Rightarrow (\text{HaveArrow}^0 \Leftrightarrow \text{HaveArrow}^1) \\
\text{Forward}^1 & \Rightarrow (\text{HaveArrow}^1 \Leftrightarrow \text{HaveArrow}^2) \\
\text{Forward}^2 & \Rightarrow (\text{HaveArrow}^2 \Leftrightarrow \text{HaveArrow}^3) \\
\vdots \\
\text{Forward}^8 & \Rightarrow (\text{WumpusAlive}^8 \Leftrightarrow \text{WumpusAlive}^9) \\
\text{Forward}^9 & \Rightarrow (\text{WumpusAlive}^9 \Leftrightarrow \text{WumpusAlive}^{10}) \\
\text{Forward}^{10} & \Rightarrow (\text{WumpusAlive}^{10} \Leftrightarrow \text{WumpusAlive}^{11}) \\
\end{align*}
\]

more generally...

\[
\begin{align*}
\text{Forward}^t & \Rightarrow (\text{HaveArrow}^t \Leftrightarrow \text{HaveArrow}^{t+1}) \\
\text{Forward}^t & \Rightarrow (\text{WumpusAlive}^t \Leftrightarrow \text{WumpusAlive}^{t+1}) \\
\end{align*}
\]

The Frame Problem

• When there are \(m\) actions and \(n\) fluents, there are \(O(mn)\) frame axioms for each time step.

• A better way to do this is to write successor-state axioms that shift the focus to the fluent and actions that might affect its state:

\[
F^{t+1} \Leftrightarrow ActionCausesF^t \lor (F^t \land \lnot ActionCausesNotF^t)
\]

that is, \(F^{t+1}\) is true iff we caused it or it was true we did not take an action to falsify \(F\).
Successor-state axiom example

• Hunter is at L_{1,1} at time t+1:

\[ L_{1,1}' \Leftrightarrow (L_{1,1} \land (\neg \text{Forward}^t \lor \text{Bump}^t)) \lor (L_{1,2} \land (\text{South}^t \land \text{Forward}^t)) \lor (L_{1,1}' \land (\text{West}^t \land \text{Forward}^t)) \]

• Hunter has arrow at time t+1:

\[ \text{HaveArrow}^t \Leftrightarrow (\text{HaveArrow}^t \land \neg \text{Shoot}) \]

Convenience axioms

• Just as we abstract code with subroutines, we can define axioms to make our lives simpler:

\[ OK_{x,y}^t \Leftrightarrow \neg P_{x,y} \land \neg (W_{x,y} \land \text{WumpusAlive}^t) \]

a cave location is safe if there’s neither a pit or a live Wumpus at that location.
Logical Agents

• Pure logic agents simply try to prove a goal
  • Wumpus example sentence contains clauses for:
    • Init\(^t\) – assertions about initial state
    • Transition\(^1\), … Transition\(^t\), successor-state axioms for all possible actions times 1,2,…,t.
    • Goal: HaveGold\(^t\) \& ClimbedOut\(^1\)
  • If a model satisfies the clause, we can look at the action variables at each time and we have our plan.
  • Otherwise not possible in t steps.

• Caveat: Not for partially observable environments, solver will simply set variables of hidden information to satisfy the problem

Logical Agents

• Hybrid agents - Use combination of logic, search and rule-based actions.
Hybrid Wumpus agent (partial)

hybrid-wumpus-agent(perceptlist, t) {  
    Tell(KB, Make-Percept-Sentence(perceptlist, t))  
    if t > 0:  
        Tell(KB, successor state axioms for time t)  
    safe = \{(x, y) : Ask(KB, Ok\textsubscript{x,y}) = True\}  
    # list of known safe caves  
    # check goals by priority  
    if Ask(KB, Glitter\textsuperscript{t}) = True:  
        plan = [Grab] + PlanRoute(current, \{[1,1]\}, safe) + [Climb]  
    else:  
        unvisited = \{(x, y) : Ask(KB, \negLt\textsubscript{x,y}) = false \forall t \leq t\}  
        # places we haven’t been  
        plan = PlanRoute(current, unvisited \cap \neg safe, safe)  
    if not plan and Ask(KB,HaveArrow\textsuperscript{t}) == True:  
        # no glitter or way to a safe square  
        possiblewumpus = \{(x, y) : Ask(KB, \neg W_{x,y}) == false\}  
        plan = plan-shot(current, possiblewumpus, safe)  
    if not plan:  
        # Couldn’t move anywhere or shoot, take a risk  
        mightbesafe = \{(x, y) : Ask(KB, \neg Ok\textsubscript{x,y}) == False\}  
        plan = PlanRoute(current, unvisited \cap mightbesafe, safe)  
    if not plan:  
        # Can’t get anywhere safely. Call it a day  
        plan = PlanRoute(current, \{[1,1]\}, safe) + [Climb]  
    action = pop(plan)  
    Tell(KB, Make-Action-Sentence(action, t))  
    t = t + 1  
    return action

PlanRoute(current, goals, allowed)  
# Plan a path from current to any goal through the set  
# of allowed nodes  
problem = RouteProblem(current, goals, allowed)  
return A* graph search(problem)