Chapter 7: Logical Agents

CS-4365 Artificial Intelligence

Chris Irwin Davis, Ph.D.

Email: chrisirwindavis@utdallas.edu
Phone: (972) 883-3574
Office: ECSS 4.603

Outline

- Knowledge-Based Agents
- The Wumpus World
- Logic
- Propositional Logic
- Propositional Theorem Proving
- Effective Propositional Model Checking
- Agents Based on Propositional Logic

§7.1 Knowledge-Based Agents

Knowledge-Based Agents

- Two most important components of AI systems:
  - Knowledge Base (KB)
  - Reasoning
- A KB is a set of representations of facts
  - Representations are called **sentences**.
  - Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.
  - Unlike a DB, the KB representation is such that allows reasoning.

Knowledge-Based Agents

- There must be a way to add new sentences to the knowledge base and a way to query what is known.
  - The standard names for these operations are TELL and ASK, respectively.
- The key issues that need to be addressed are:
  - How to represent the knowledge
  - How to find inference rules that allow us to reason on the KB
- The agent perceives the outside world and puts more facts on the KB, and then makes decisions about future actions

Knowledge-based Agent

- Each time the agent program is called, it does three things.
  - First, it TELLs the knowledge base what it perceives.
  - Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.
  - Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.

Generic Knowledge-Based Agent

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  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
```

Figure 7.1 A generic knowledge-based agent Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

Knowledge-based Agent

- The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other.
  - MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time.
  - MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time.
  - Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK.
- Later sections will reveal these details.

Knowledge Levels

- One can distinguish several knowledge levels:

Level	Primitives
Epistemological level	Concept types, inheritance and structuring relations
Logical level	Propositions, predicates, logical operators
Implementation level	Atoms, pointers, data structures

Examples

- Epistemological level:
  - Golden Gate Bridge links San Francisco and Marin County
  - Independent of Implementation
- Logical level:
  - Links (GG Bridge, SF, Marin)
- Implementation level:
  - pick a data representation for above

Knowledge-based Agent

- A knowledge-based agent can be built simply by TELLing it what it needs to know.
- Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative** approach to system building.
- In contrast, the **procedural** approach encodes desired behaviors directly as program code.

Knowledge-based Agent

- In the 1970s and 1980s, advocates of the two approaches engaged in heated debates.
- We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.
- We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself.
- These mechanisms, which are discussed in Chapter 18, create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

§7.2 The Wumpus World

Wumpus World

- The Wumpus World is a cave consisting of rooms consisting of rooms and passageways
- Somewhere in the cave is a terrible wumpus
- The wumpus can be shot by an agent, but the agent has only *one arrow*
- Rooms may contain
  - Bottomless pit
  - Gold!

PEAS Description

- **Performance Measure**:
  - +1,000 for exiting cave with gold
  - –10,000 for falling into a pit or being eaten by the wumpus
  - –1 for each action taken
  - –10 for using the up arrow
- **Environment**:
  - 4 x 4 grid of rooms, agent begins in square [1,1]
  - Locations of wumpus and gold chosen randomly
  - Probability of pit is 0.2 (other than start square)

PEAS Description

- **Actuators**:
  - Agent can move *Forward*, *TurnLeft* by 90°, *TurnRight* by 90°
  - If an agent tries to move *Forward* into a wall, no change
  - Action *Grab* can be used if in the square with gold
  - Action *Shoot* can be used to fire an arrow in the direction the agent is facing. The arrow continues until it hits either the wumpus or a wall. The agent has only one arrow, only the first *Shoot* action has any effect.
  - Action *Climb* can be used to exit the cave, but only from [1,1]

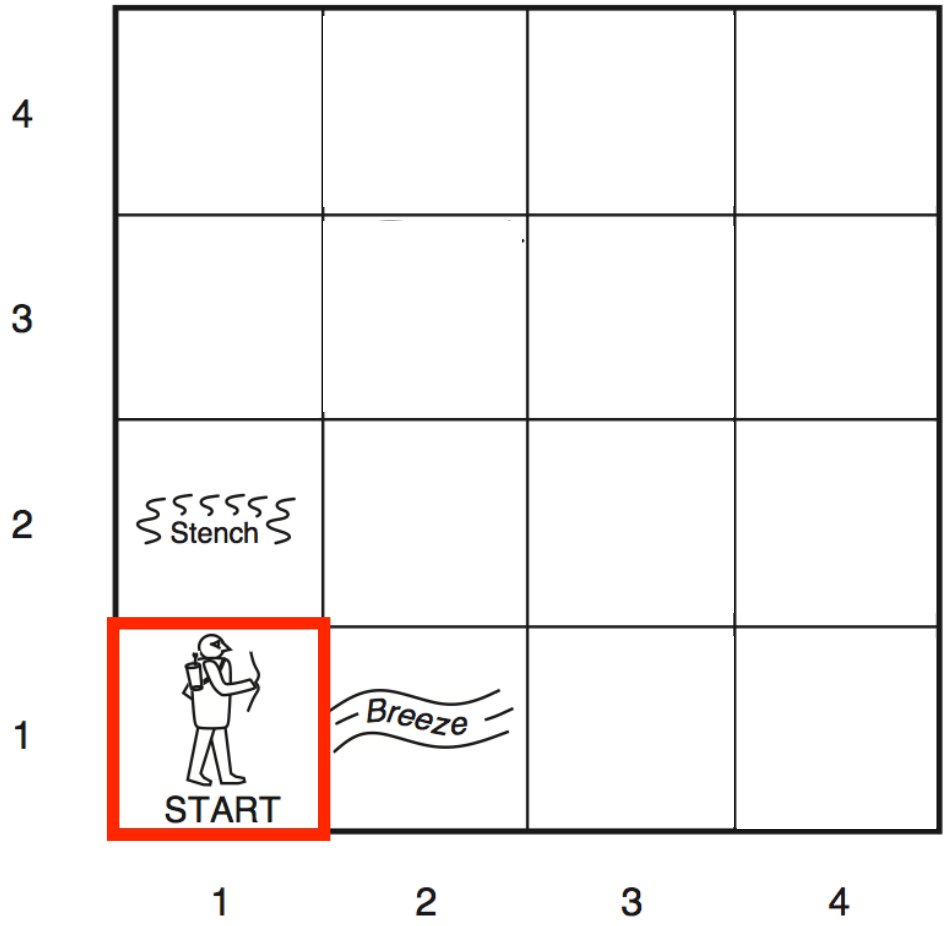
PEAS Description

- **Sensors**:
  - The agent has five sensors, each of which returns one bit of info
  - In the square containing the wumpus and adjacent squares agent perceives *Stench*
  - In squares directly adjacent to a pit the agent perceives *Breeze*
  - In the square with the gold the agent perceives *Glitter*
  - When an agent walks into a wall, it perceives *Bump*
  - When the wumpus is killed, it emits a *Scream* that can be perceived anywhere in the cave

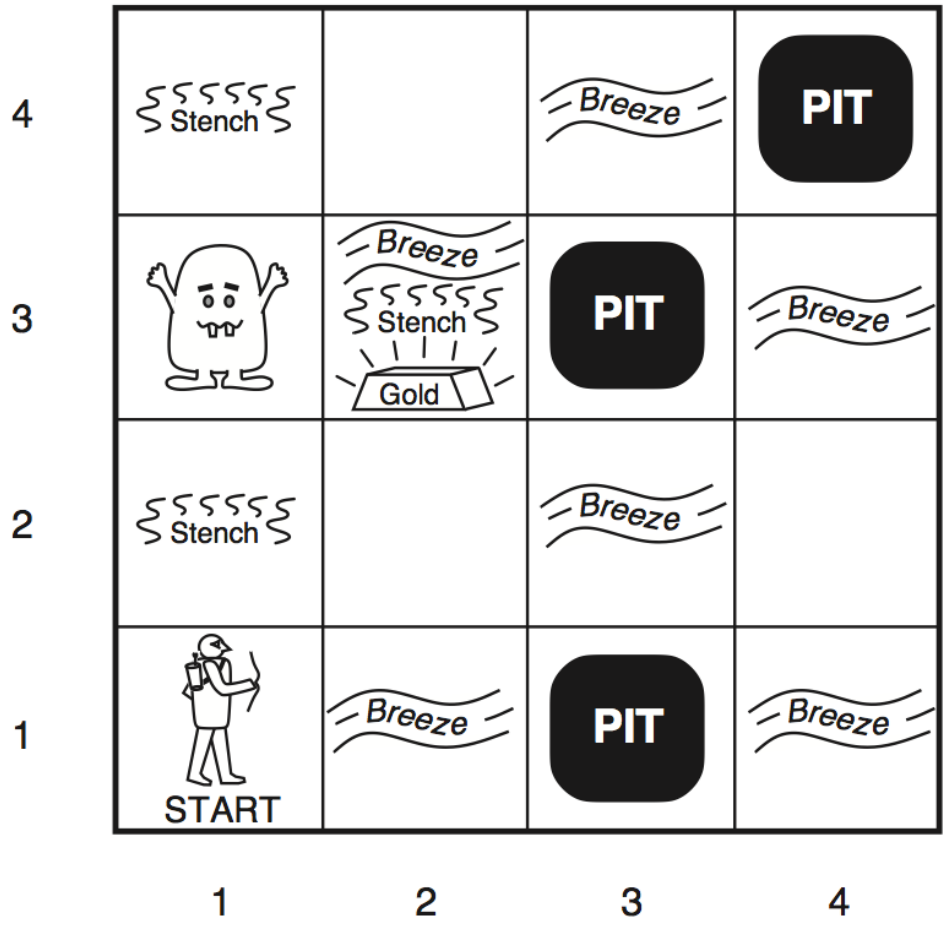
Wumpus World Example

- **Percepts:**
  - A *stench* in adjacent squares and in the square containing wumpus
  - A *breeze* in squares adjacent to a pit
  - A *glitter* in squares with gold
  - A *bump* when agent goes into a wall
  - A *scream* when wumpus is killed
- Vector for reading percepts
  - (Stench, Breeze, Glitter, Bump, Scream)
  - with values 1 or 0 as percepts indicate.

The Wumpus World



The Wumpus World



Wumpus World Characterization

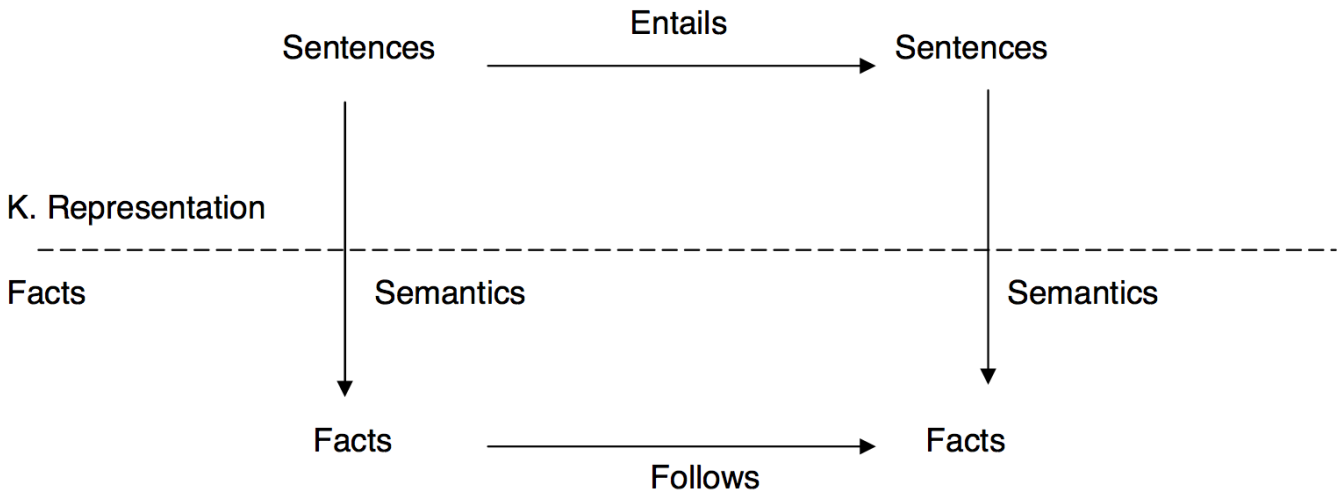
- **Fully Observable?**
  - No – only local perception
- **Deterministic?**
  - Yes – outcomes exactly specified
- **Episodic?**
  - No – sequential at the level of actions, and history is important
- **Static?**
  - Yes – Wumpus and Pits do not move
- **Discrete?**
  - Yes
- **Single-agent?**
  - Yes – Wumpus is essentially a natural feature

The Wumpus Example

- Actions
  - Go forward, turn right 90°, turn left 90°, grab, shoot, climb,
  - Agent dies, Wumpus dies.
- Goal
  - 1000 points for getting gold out of cave, 1 point penalty for each action taken, 10,000 point penalty for getting killed.
- The game repeats several times with different initial positions.

§7.3 Logic

- A *KR* language, like any language, has:
  - **Syntax**—specifies possible forms that sentences can take
  - **Semantics**—determines the facts in the world to which sentence refers.



Possible worlds or models

- In Section 7.1, we said that knowledge bases consist of **sentences**.
- These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed.
- The notion of syntax is clear enough in ordinary arithmetic:
  - "x + y = 4" is a well-formed sentence, whereas
  - "x4y+ =" is not
- A logic must also define the **semantics** or meaning of sentences. The semantics defines the truth of each sentence with respect to each possible world.

Possible worlds or models

- A sentence has to be considered with respect to a possible world or model.
  - Ex: $x + y = 4$
    - true when $x = 2$ and $y = 2$, but
    - false when $x = 1$ and $y = 1$
  - Ex: Clinton is the US President.
    - was true in the worlds during Jan. 1993 – Jan. 2001, but
    - not true today.
  - Ex: Dr. Davis is at UTD.
    - various interpretations are possible

Possible worlds or models

- Note – Possible worlds are related to contexts.
  If a sentence α is true in a model *m* we say that "α satisfies *m*", or "*m* is a model of α".
  - M(α) – defines the set of *all* models of α
- Now that we have a notion of truth, we are ready to talk about *Logical Reasoning*
  - Involves logical *entailment* between sentences, the idea that a sentence *follows logically* from another sentence

Entailment

- Logical reasoning is based on the concept of entailment – a sentence follows logically from another sentence.
  - $\alpha \models \beta$ (i.e. sentence α entails sentence β
  - Definition:
    - $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true,
  - Mathematically:
    - $\alpha \models \beta$ if and only if $M(\alpha) \subseteq M(\beta)$
- α is stronger (more specific) than β
- $M(\alpha)$ is equal to or fewer than $M(\beta)$
- Entailment is a relationship between sentences (syntax) that is based on semantics

- $KB \models \alpha$
  - "KB entails α".
- Sentences α may be inferred from KB using inference procedures.
- $KB \vdash_i \alpha$
  - "α is derived from KB using inference procedure *i*"
  - Entailment generates only true sentences given that the KB contains true sentences.

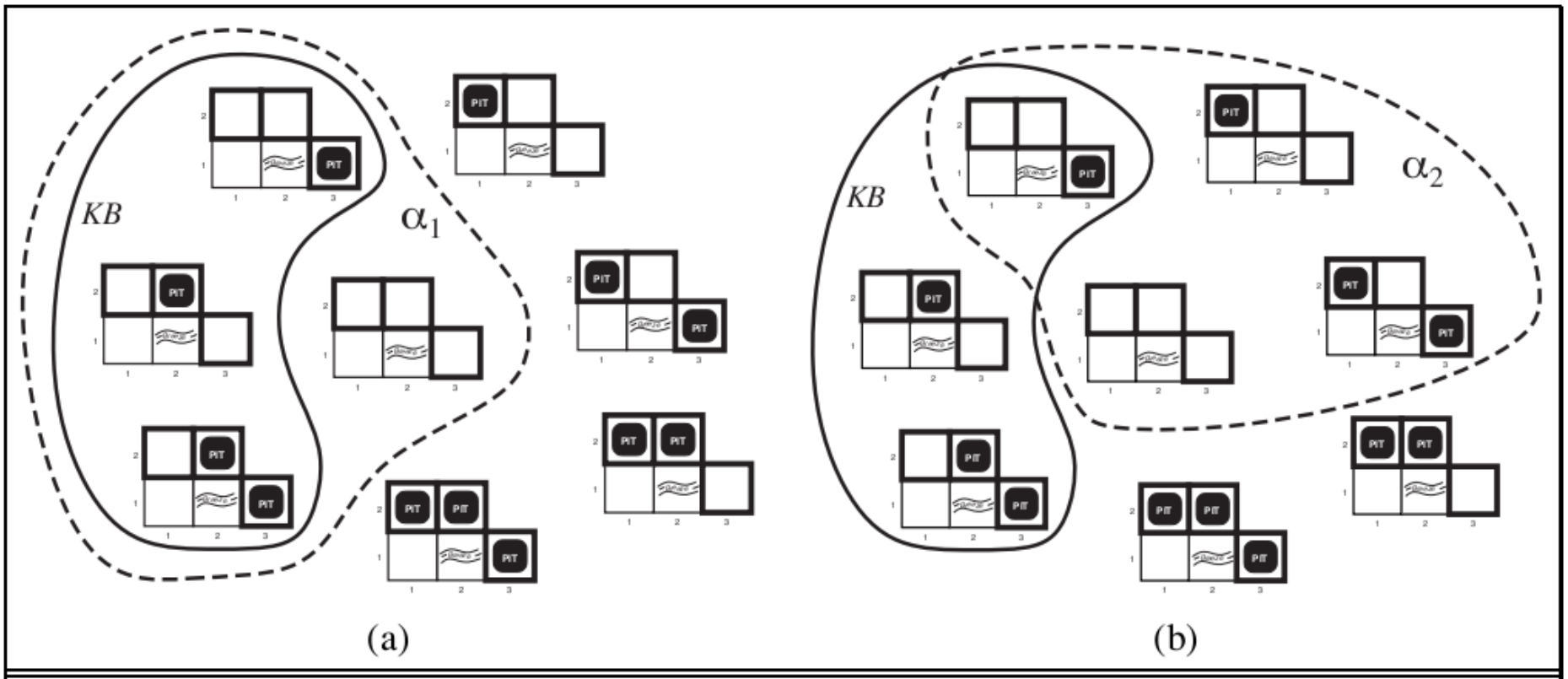


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

- An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**.
  - Soundness is a highly desirable property.
  - An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles.
  - It is easy to see that model checking, when it is applicable, is a sound procedure.

- The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed.
  - For real haystacks, which are *finite* in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack.
  - For many knowledge bases, however, the haystack of consequences is *infinite*, and completeness becomes an important issue.
  - Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

- Inferences may be:
  - **Sound** – when *generates* only entailed sentences that are true. (truth-preserving).
    A **proof** – is the *procedure* of getting sound inference (the steps toward the inference.
  - **Complete** – if it can find a proof for any sentence that is entailed from a KB.

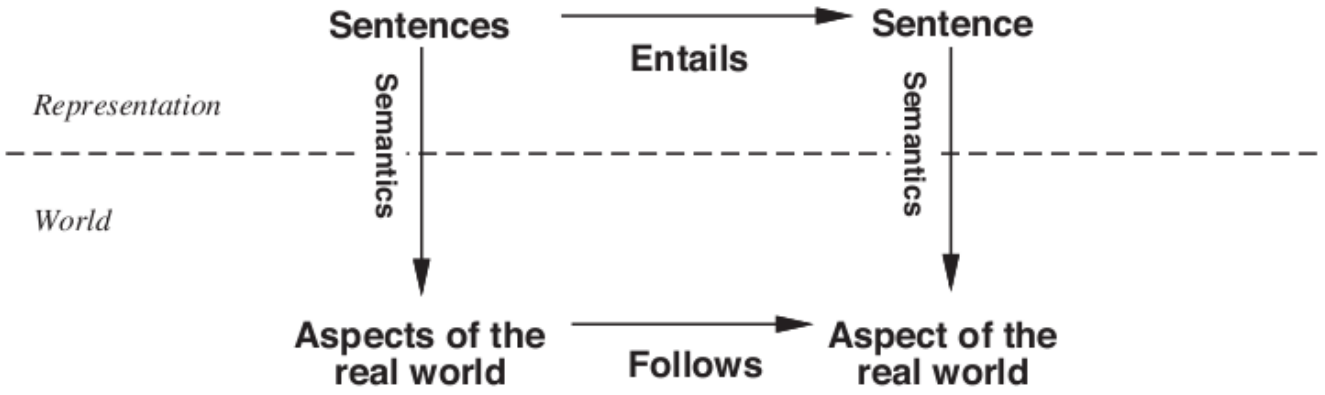


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

§7.4 Propositional Logic

Propositional Logic

- We look at: syntax, semantics, inference rules
  - **Syntax**
    - Symbols representing propositions:
      - P, Q, R,…
      - Ex: P: John is bold.
  - Logical constants: **True**, **False**
  - Wrapping parentheses ( ) that group symbols
  - Connectives
    - $\wedge$, $\vee$, $\Rightarrow$, $\rightarrow$, $\Leftrightarrow$, $\neg$
  - Precedence of connectives in propositional logic
    - $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$
    - $\neg P \vee Q \wedge R \Rightarrow S$ is equivalent to $((\neg P) \vee (Q \wedge R)) \Rightarrow S$

Syntax

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} &\rightarrow (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
&\mid \neg\ \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Semantics

- Atomic Sentences
  - *True* is true in every model and *False* is false in every model
  - The truth value of every other proposition symbol must be specified directly in the model

Semantics

- **Complex Sentences** – 5 rules, which hold for any subsentences P and Q in any model m
  - $\neg P$ is true iff P is false in m
  - $P \wedge Q$ is true iff both P and Q are true in m
  - $P \vee Q$ is true iff either P or Q is true in m
  - $P \Rightarrow Q$ is true unless P is true and Q is false in m
  - $P \Leftrightarrow Q$ is true iff P and Q are both true in m or are both false in m

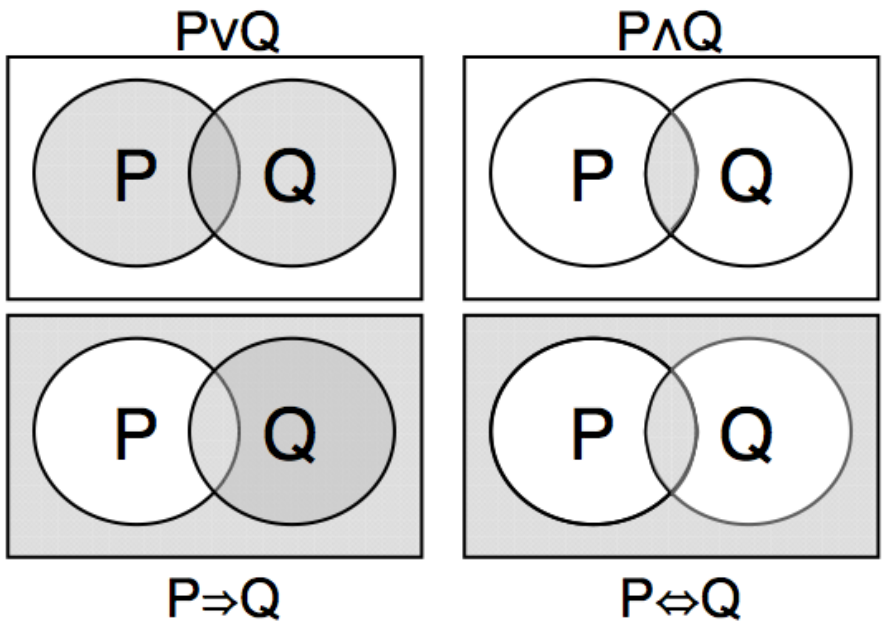
Propositional Logic

- Semantics results from the meaning of proposition symbols, constants and logical connectives.
- A sentence is:
  - **Valid** if is true for all interpretations
  - **Satisfiable** if is true for some interpretations
  - **Unsatisfiable** if is false for all interpretations

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

Figure 7.8 Truth tables for the five logical connectives.

Propositional Logic



Models of complex sentences in terms of the models of their components. In each diagram, the shaded parts correspond to the models of the complex sentences.

Propositional Logic: Some useful equivalent expressions

$P \wedge (Q \wedge R)$	$\Leftrightarrow$	$(P \wedge Q) \wedge R$	Associativity of conjunction
$P \vee (Q \vee R)$	$\Leftrightarrow$	$(P \vee Q) \vee R$	Associativity of disjunction
$P \wedge Q$	$\Leftrightarrow$	$Q \wedge P$	Commutativity of conjunction
$P \vee Q$	$\Leftrightarrow$	$Q \vee P$	Commutativity of disjunction
$P \wedge (Q \vee R)$	$\Leftrightarrow$	$(P \wedge Q) \vee (P \wedge R)$	Distributivity of $\wedge$ over $\vee$
$P \vee (Q \wedge R)$	$\Leftrightarrow$	$(P \vee Q) \wedge (P \vee R)$	Distributivity of $\vee$ over $\wedge$
$\neg(P \wedge Q)$	$\Leftrightarrow$	$\neg P \vee \neg Q$	de Morgan's Law
$\neg(P \vee Q)$	$\Leftrightarrow$	$\neg P \wedge \neg Q$	de Morgan's Law
$P \Rightarrow Q$	$\Leftrightarrow$	$\neg Q \Rightarrow \neg P$	Contraposition
$\neg \neg P$	$\Leftrightarrow$	P	Double Negation
$P \Rightarrow Q$	$\Leftrightarrow$	$\neg P \vee Q$	
$P \Leftrightarrow Q$	$\Leftrightarrow$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$	
$P \Leftrightarrow Q$	$\Leftrightarrow$	$(P \wedge Q) \vee (\neg P \wedge \neg Q)$	
$P \wedge \neg P$	$\Leftrightarrow$	False	
$P \vee \neg P$	$\Leftrightarrow$	True	

Truth Tables

Truth tables may be used for proving the validity of small sentences.

$$((P \vee H) \wedge \neg H) \Rightarrow P$$

P	H	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H \Rightarrow P$
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*

Truth table showing validity of a complex sentence.

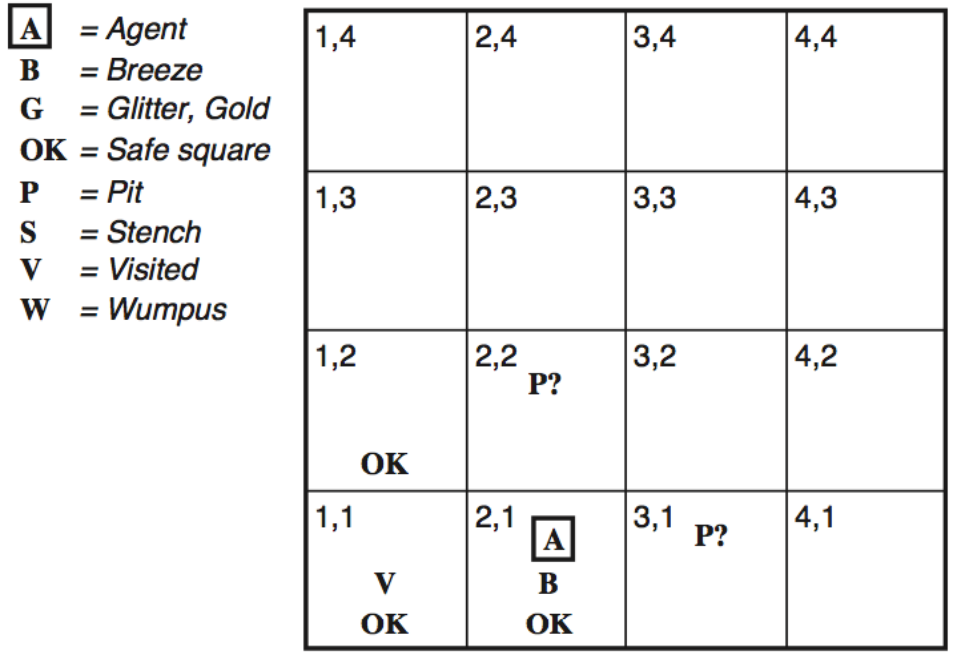
A proposition with n symbols requires 2^n rows of truth table, thus is impractical.

A Simple Knowledge Base

- There is no pit in [1,1]
  - $R_1 : \neg P_{1,1}$
- Notation:
  - We label each sentence R_i so that we can refer to them.
  - $P_{i,j}$ indicates the presence of a pit in [1,1]
- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:
  - $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
  - $R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

A Simple Knowledge Base

- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b)
  - $R_4 : \neg B_{1,1}$
  - $R_5 : B_{2,1}$



§7.5 Propositional Theorem Proving

Theorem Proving

- So far, we have shown how to determine entailment by *model checking* (i.e. enumerating models and showing that the sentence must hold in all models)
- Entailment can be accomplished via **theorem proving** – applying rules of inference directly to the sentences in the KB to construct a proof without consulting models
  - Logical Equivalence
  - Validity
  - Satisfiability

Logical Equivalence

- Two sentences α and β are logically equivalent if they are true in the same set of models
- $\alpha \equiv \beta$
- For example, $P \wedge Q \equiv Q \wedge P$
- Other equivalences on the next slide...

Standard Logical Equivalences

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{de Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{de Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

Validity

- A sentence is **valid** if it is true for *all* models
- For example, $P \vee \neg P$ is valid
- Valid sentences are also known as *tautologies*
- Why is this important?
- For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid
- Thus, we can prove $\alpha \models \beta$ if we can show $(\alpha \Rightarrow \beta)$ is true in every model

Satisfiability

- A sentence is **satisfiable** if it is true in, or satisfied by, *some* model
- For example, the KB given earlier $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$ is satisfiable because there are three models in which it is true

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Inference and Proofs

- Validity and Satisfiability are connected
- α is satisfiable iff $\neg\alpha$ is not valid
  - Which leads to
- $\alpha \models \beta$ iff the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable
- Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$
  - Proof by contradiction

Inference and Proofs

- $\alpha \models \beta$
- $\alpha \Rightarrow \beta$
- $\neg\alpha \vee \beta$ satisfiable
- $\neg(\neg\alpha \vee \beta)$ unsatisfiable
- $(\alpha \wedge \neg\beta)$ unsatisfiable (de Morgans)

Inference Rules for Propositional logic

- Seven inference rules for propositional logic.
  - Modus Ponens or Implication-Elimination
  - And-Elimination
  - And-Introduction
  - Or-Introduction
  - Double-Negation Elimination
  - Unit Resolution
  - Resolution
- The **unit resolution rule** is a special case of the **resolution rule**, which in turn is a special case of the **full resolution rule** for first-order logic.

Inference Rules for Propositional logic

Modus Ponens or **Implication-Elimination**: (From an implication and the premise of the implication you can infer the conclusion.)

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

Inference Rules for Propositional logic

Modus Ponens or **Implication-Elimination**: (From an implication and the premise of the implication you can infer the conclusion.)

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

And-Elimination: (From a conjunction, you can infer any of the conjuncts.)

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}$$

Inference Rules for Propositional logic

Modus Ponens or **Implication-Elimination**: (From an implication and the premise of the implication you can infer the conclusion.)

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

And-Elimination: (From a conjunction, you can infer any of the conjuncts.)

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}$$

And-Introduction: (From a list of sentences, you can infer their conjunction.)

$$\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}$$

Inference Rules for Propositional logic

Modus Ponens or **Implication-Elimination**: (From an implication and the premise of the implication you can infer the conclusion.)

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

And-Elimination: (From a conjunction, you can infer any of the conjuncts.)

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}$$

And-Introduction: (From a list of sentences, you can infer their conjunction.)

$$\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}$$

Or-Introduction: (From a sentence, you can infer its disjunction with anything else at all.)

$$\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}$$

Inference Rules for Propositional logic

Double-Negation Elimination: (From a doubly negated sentence, you can infer a positive sentence.)

$$\frac{\neg\neg\alpha}{\alpha}$$

Inference Rules for Propositional logic

Double-Negation Elimination: (From a doubly negated sentence, you can infer a positive sentence.)

$$\frac{\neg\neg\alpha}{\alpha}$$

Unit Resolution: (From a disjunction, if one of the disjuncts if false, then you can infer the other one is true.)

$$\frac{\alpha \vee \beta,\ \neg\beta}{\alpha}$$

Inference Rules for Propositional logic

Double-Negation Elimination: (From a doubly negated sentence, you can infer a positive sentence.)

$$\frac{\neg\neg\alpha}{\alpha}$$

Unit Resolution: (From a disjunction, if one of the disjuncts if false, then you can infer the other one is true.)

$$\frac{\alpha \vee \beta,\ \neg\beta}{\alpha}$$

Resolution: (This is the most difficult. Because β cannot be both true and false, one of the other disjuncts must be true in one of the premises. Or equivalently, implication is transitive.)

$$\frac{\alpha \vee \beta,\ \neg\beta \vee \gamma}{\alpha \vee \gamma}$$ or equivalently $$\frac{\neg\alpha \Rightarrow \beta,\ \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma}$$

Inference and Proofs

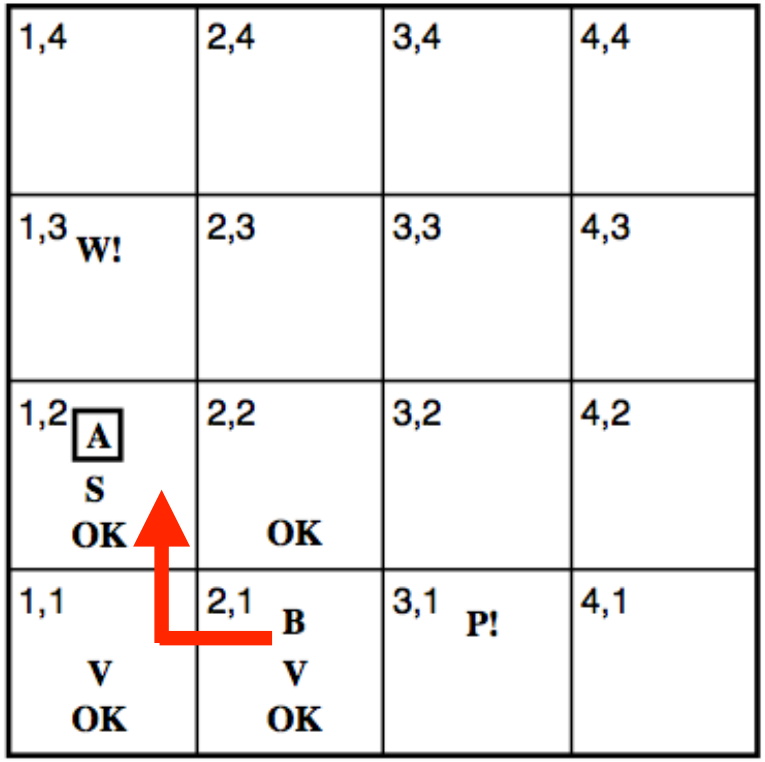
- Apply bi-conditional elimination to R_2
  - $R_6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
- Apply And-Elimination to R_6
  - $R_7 : ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
- Logical equivalence for contrapositives gives (using R_7)
  - $R_8 : (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}))$
- Now we can apply Modus Ponens (using R_8 and R_4)
  - $R_9 : \neg(P_{1,2} \vee P_{2,1})$
- Finally, we apply De Morgan's rule to R_9, concluding
  - $R_{10} : \neg P_{1,2} \wedge \neg P_{2,1}$

KB Inference Summary

- Starting with KB = $\{R_1, R_2, R_3, R_4, R_5\}$, we were able to infer R_{10} (i.e. neither $P_{1,2}$ nor $P_{2,1}$ contain a pit) via the intermediate inference of sentences/propositions R_6, R_7, R_8, and R_9.
- The knowledge base now contains statements/propositions KB = $\{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9, R_{10}\}$.
- We now continue to grow the KB through the KB-AGENT cycle of ASK, TELL, and Inference.

Proof By Resolution

- Consider the steps leading up to 7.4(a)
- The agent returns from [2,1], returning to [1,1], and then goes to [1,2]
  - Stench and no breeze



Proof By Resolution

- Add rules
  - $R_{11} : \neg B_{1,2}$
  - $R_{12} : B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$
- Using same process that derived R_{10}, we can derive the absence of pits in [2,2] and [1,3]. ($P_{1,1}$ is known to be pitless)
  - $R_{13} : \neg P_{2,2}$
  - $R_{14} : \neg P_{1,3}$

Note that we generated R_6 through R_9 in the course of inferring R10. We did not bother to show the intermediate inferences in arriving at either statements R_{13} or R_{14},

- We can apply bi-conditional elimination to R_3, followed by Modus Ponens with R_5 to obtain the fact that there is a pit in at least one of [1,1], [2,2], or [3,1], i.e. new statement R_{15}.
  - $R_{15} : P_{1,1} \vee P_{2,2} \vee P_{3,1}$

Proof By Resolution

- Now the application of the resolution rule
- The literal $\neg P_{2,2}$ in R_{13} resolves with the literal $P_{2,2}$ in R_{15} to give the resolvent:
  - $R_{16} : P_{1,1} \vee P_{3,1}$
  - i.e. There's a pit in one of [1,1], [2,2], or [3,1] — and it's not in [2,2], then it's in [1,1] or [3,1].
- Similarly, the literal $\neg P_{1,1}$ in R_1 resolves with the literal $P_{1,1}$ in R_{16} to give
  - $R_{16} : P_{3,1}$
  - i.e. If there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]

Proof By Resolution

- These last two inference steps are examples of the **unit resolution** inference rule

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

- Where ℓ is a literal and ℓ_i and m are **complementary literals** (i.e. one is the negation of the other)
- The unit resolution rule takes a clause (a disjunction of literals) and a literal and produces a new clause
  - Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**

§7.5.2 Resolution & CNF

Conjunctive Normal Form (CNF)

- The resolution rule applies only to disjunctions of literals, so it would seem to be relevant only to knowledge bases and queries consisting of such disjunctions.
- How, then, can it lead to a complete inference procedure for all of propositional logic?
- The answer is that every sentence of propositional logic is logically equivalent to a conjunction of disjunctions of literals.
- A sentence expressed as a conjunction of disjunctions of literals is said to be in conjunctive normal form or CNF.

$$(\ell_{1,1} \vee \ldots \vee \ell_{1,k}) \wedge \ldots \wedge (\ell_{n,1} \vee \ldots \vee \ell n, k)$$

Conjunctive Normal Form (CNF)

- Every sentence can be transformed into a 3-CNF sentence that has an equivalent set of models.
- Conversion of $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

Conjunctive Normal Form (CNF)

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Conjunctive Normal Form (CNF)

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) .$$

Conjunctive Normal Form (CNF)

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) .$$

3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences from Figure 7.11:

$$\neg(\neg\alpha) \equiv \alpha \quad \text{(double-negation elimination)}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{(de Morgan)}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{(de Morgan)}$$

In the example, we require just one application of the last rule:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) .$$

Conjunctive Normal Form (CNF)

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

   $$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) .$$

3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences from Figure 7.11:

   $\neg(\neg\alpha) \equiv \alpha$ (double-negation elimination)
   $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (de Morgan)
   $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (de Morgan)

   In the example, we require just one application of the last rule:

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) .$$

4. Now we have a sentence containing nested $\wedge$ and $\vee$ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing $\vee$ over $\wedge$ wherever possible.

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) .$$

A Simple Resolution Algorithm

- Inference procedures based on resolution work by using the principle of proof by contradiction discussed at the end of Section 7.4.
  - That is, to show that KB $\models \alpha$, we show that (KB $\wedge \neg\alpha$) is unsatisfiable.
    - In other words, to show that KB $\models \alpha$ (equivalently $\neg$KB $\vee \alpha$), we show that its negation, i.e. $\neg(\neg\text{KB} \vee \alpha)$ is unsatisfiable
    - $\neg(\neg\text{KB} \vee \alpha) \equiv (\text{KB} \wedge \neg\alpha)$ (de Morgan)
  - i.e. Proof by Contradiction.

A Simple Resolution Algorithm

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses $\leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new $\leftarrow \{\}$
  loop do
    for each C_i, C_j **in** *clauses* **do**
      resolvents $\leftarrow$ PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new $\leftarrow$ *new* $\cup$ *resolvents*
    if *new* $\subseteq$ *clauses* **then return** *false*
    clauses $\leftarrow$ *clauses* $\cup$ *new*

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

A Simple Resolution Algorithm: In English

- First, (KB $\wedge \neg\alpha$) is converted into CNF.
- Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present.
- The process continues until one of two things happens:
  - There are no new clauses that can be added, in which case α does not entail β, or
  - An application of the resolution rule derives the empty clause, in which case α entails β.

A Simple Resolution Algorithm

- The empty clause—a disjunction of no disjuncts—is equivalent to **False** because a disjunction is true only if at least one of its disjuncts is true.
  - Another way to see that an empty clause represents a contradiction is to observe that it arises only from resolving two complementary unit clauses such as $P \wedge \neg P$.
- We can apply the resolution procedure to a very simple inference in the Wumpus World. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is:
  - KB = $R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$
  - And we wish to prove α (which is, say, $\neg P_{1,2}$)

Converting to CNF

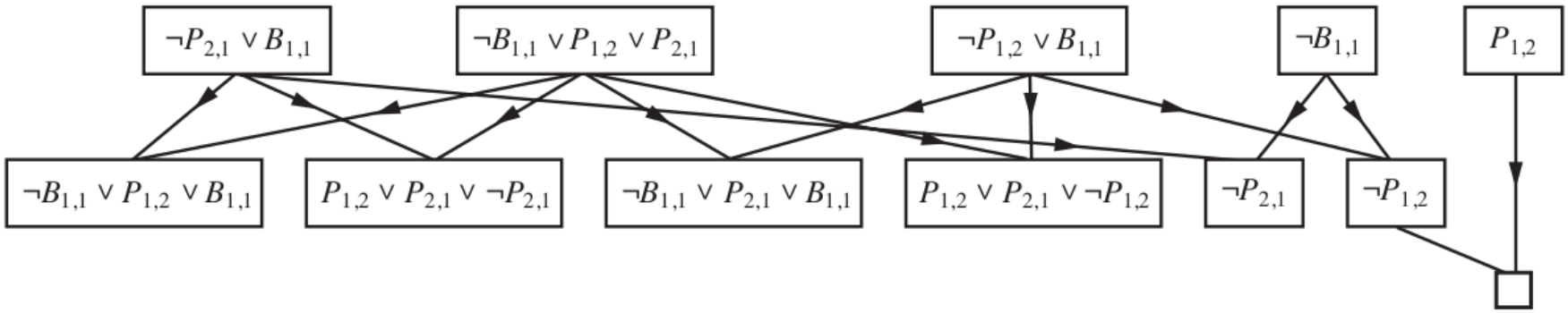
- $\text{KB} = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$
- $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
  - bi-conditional elimination
    - $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
  - implication elimination
    - $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
    - $\neg P_{1,2} \vee B_{1,1}$
    - $\neg P_{2,1} \vee B_{1,1}$
- $\neg B_{1,1}$

Converting to CNF: Summary

- KB = $R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$
  - $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
  - $\neg P_{1,2} \vee B_{1,1}$
  - $\neg P_{2,1} \vee B_{1,1}$
  - $\neg B_{1,1}$
- In other words…
  - KB = $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge (\neg B_{1,1})$

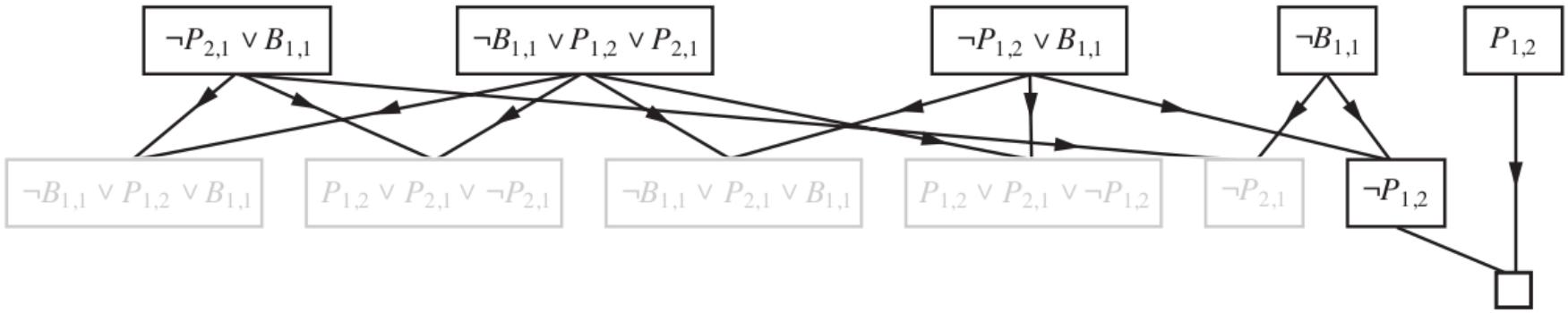
A Resolution Algorithm

- Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.
- Convert (KB $\wedge \neg\alpha$) into CNF
- The second row of the figure shows all the clauses obtained by resolving pairs in the first row.
- Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square



A Resolution Algorithm

- Inspection of the previous diagram (Figure 7.13) reveals that many resolution steps are pointless.
- For example, the clause $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ is equivalent to True $\vee P_{1,2}$ which is equivalent to True.
- Deducing that *True is true* is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.



Forward and Backward Chaining

- The completeness of resolution makes it a very important inference method.
- In many practical situations, however, the full power of resolution is not needed. Real-world knowledge bases often contain only clauses of a restricted kind called **Horn clauses**.
- A Horn clause is a disjunction of literals of which *at most one* is positive.
  - For example, the clause
  - $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$, where $L_{1,1}$ means that the agent's location is [1,1], is a Horn clause, whereas
  - $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is not.

Forward and Backward Chaining

- The restriction to just one positive literal may seem somewhat arbitrary and uninteresting, but it is actually very important for three reasons:
  - Every Horn clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.
  - Inference with Horn clauses can be done through the **forward chaining** and **backward chaining** algorithms
  - Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.

Horn Clauses

- Every Horn clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.
  - $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1}) \equiv (L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$.
  - Horn clauses like this one with *exactly one* positive literal are called **definite clauses**.
  - The positive literal is called the **head** and the negative literals form the **body** of the clause.
  - Definite clauses form the basis for logic programming

Horn Clauses

- A Horn clause with *no* positive literals can be written as an implication whose conclusion is the literal *False*.
- For example, the clause
  - $(\neg W_{1,1} \vee \neg W_{1,2})$—the wumpus cannot be in both [1,1] and [1,2]—is equivalent to
  - $(W_{1,1} \wedge W_{1,2}) \Rightarrow False$
- Such sentences are called integrity constraints in the database world, where they are used to signal errors in the data

Forward and Backward Chaining

UTD

- The **forward-chaining algorithm** PL-FC-ENTAILS?(KB,q) determines whether a single proposition symbol q—the query—is entailed by a knowledge base of Horn clauses.
- It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts.
- For example, if $L_{1,1}$ and *Breeze* are known and ($L_{1,1} \wedge$ *Breeze*) $\Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added.
- This process continues until the query q is added or until no further inferences can be made.

Forward and Backward Chaining

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

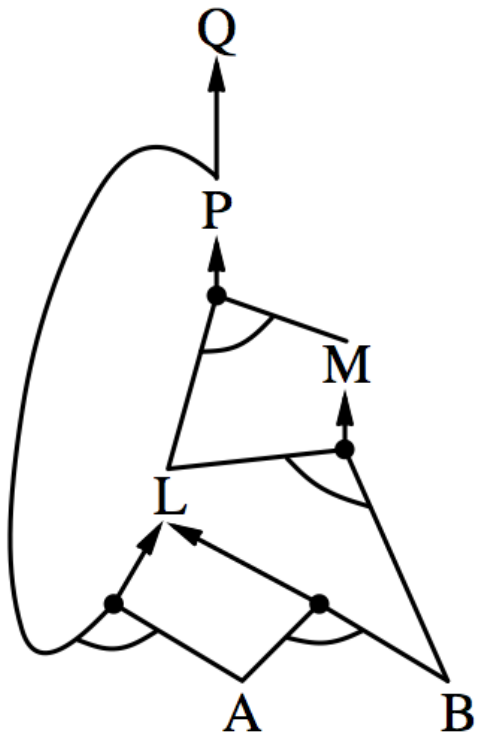
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

(a)



(b)

Figure 7.15 (a) A simple knowledge base of Horn clauses. (b) The corresponding AND–OR graph.

Forward and Backward Chaining

- The **backward-chaining algorithm**, as its name suggests, works backwards from the query.
- If the query q is known to be true, then no work is needed.
- Otherwise, the algorithm finds those implications in the knowledge base that conclude q.
- If all the premises of one of those implications can be proved true (by backward chaining), then q is true.
- When applied to the query Q in Figure 7.15, it works back down the graph until it reaches a set of known facts that forms the basis for a proof.

Forward and Backward Chaining

- Backward chaining is a form of goal-directed reasoning. It is useful for answering specific questions such as “What shall I do now?” and “Where are my keys?”
- Often, the cost of backward chaining is much less than linear in the size of the knowledge base, because the process touches only relevant facts.
- In general, an agent should share the work between forward and backward reasoning, limiting forward reasoning to the generation of facts that are likely to be relevant to queries that will be solved by backward chaining.

§7.6 Effective Propositional Model Checking

A Complete Backtracking Algorithm

- Davis-Putnam Algorithm (1960)
  - Davis, Logeman, Loveman (1962)
- Three improvements over TT-ENTAILS
  - Early termination
  - Pure symbol heuristic
  - Unit clause heuristic

A Complete Backtracking Algorithm

UTD

- **Early Termination**
- The algorithm detects whether the sentence must be true or false, even with a partially completed model.
- A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete.
  - For example, the sentence (A ∨ B) ∧ (A ∨ C) is true if A is true, regardless of the values of B and C. Similarly, a sentence is false if any clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

A Complete Backtracking Algorithm

UTD

- **Pure symbol heuristic**
- A *pure symbol* is a symbol that always appears with the same "sign" in all clauses.
  - For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure.
- It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals *true*, because doing so can never make a clause *false*.

A Complete Backtracking Algorithm

- **Unit clause heuristic**
  - A unit clause was defined earlier as a clause with just one literal.
- In the context of DPLL, it also means clauses in which all literals but one are already assigned *false* by the model.
  - For example, if the model contains B=*false*, then (B ∨ ¬C) becomes a unit clause because it is equivalent to (False ∨ ¬C), or just ¬C. Obviously, for this clause to be true, C must be set to false.
- One important consequence of the heuristic is that any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately

DPLL Algorithm

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(*clauses*, *symbols*, [ ])

---

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, EXTEND(P, *value*, *model*)
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* − P, EXTEND(P, *value*, *model*)
  P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, EXTEND(P, *true*, *model*)) **or**
      DPLL(*clauses*, *rest*, EXTEND(P, *false*, *model*))

Figure 7.16 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, it operates over partial models.

Local Search Algorithms

- Hill Climbing and Simulated Annealing
- These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function.
- Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job.

Local Search Algorithms

- One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT.
  - On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip.
  - It chooses randomly between two ways to pick which symbol to flip:
    - (1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state, or
    - (2) a "random walk" step that picks the symbol randomly.

Local Search Algorithms

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** a satisfying model or *failure*
  inputs: *clauses*, a set of clauses in propositional logic
    p, the probability of choosing to do a "random walk" move, typically around 0.5
    max_flips, number of flips allowed before giving up

  model ← a random assignment of *true*/*false* to the symbols in *clauses*
  for $i = 1$ **to** *max_flips* **do**
    if *model* satisfies *clauses* **then return** *model*
    clause ← a randomly selected clause from *clauses* that is false in *model*
    with probability *p* flip the value in *model* of a randomly selected symbol from *clause*
    else flip whichever symbol in *clause* maximizes the number of satisfied clauses
  return *failure*

Figure 7.17 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

Local Search Algorithms

- Local-search algorithms such as WALKSAT are most useful when we expect a solution to exist
- Local search cannot always detect unsatisfiability, which is required for deciding entailment.