Overview

I Artificial Intelligence

II Problem Solving

III Knowledge, Reasoning, Planning

7. Logical Agents

8. First-Order Logic

9. Inference in First-Order Logic

10. Knowledge Representation

11. Automated Planning

IV Uncertain Knowledge and Reasoning

V Machine Learning

VI Communicating, Perceiving, and Acting

VII Conclusions

Logical Agents

- Knowledge-based Agents
- The Wumpus World
- Logic
- Propositional Logic: A Very Simple Logic
- Propositional Theorem Proving
- Effective Propositional Model Checking
- Agents based on Propositional Logic

Advantages of Knowledge-based Agents

- Smart dealing with partially observable environments:
  - With percepts and knowledge the state of the world can be infered more precisely
    - Example: Medical diagnosis
- New tasks as explicitly described goals
- Fast knowledge acquisition by being told or by learning
- Adaptation to changes in the environment by updating relevant knowledge

Basic Structure of Knowledge-based Agent

In each time step:

- Store percepts in knowledge base (Tell)
- Infer an action from knowledge base (Ask)
- Perform action
- Store action in knowledge base (Tell)

- Logic is well suited to represent knowledge
  - Propositional logic
  - First-order logic

Agents and Knowledge

- **Problem solving agents** have little knowledge
    - Only what action are possible in a state and the resulting new state
    - A heuristic function assessing a state
- **Constraint problem solving agents** have slightly more knowledge
    - Factored representation of a state consisting of variables with values
    - Constraints on variables and values
- **Neural net agents** (future subject) have implicite knowledge
    - Encoding of a state in „neurons“ (arrays of numbers)
    - Weights connecting different neurons (arrays of numbers)

- **Logical Agents**
    - Representation of knowledge in a general form
    - Useful for many purposes and also for communication with humans

Knowledge-Based Agents

- **Knowledge base:** Contains representations of facts about the world in form of sentences
- **Sentence:** An individual representation expressed in a knowledge representation language
  - Sentences can be derived from other sententences or not; the latter are called **axioms**
- **Knowledge representation language**: There are general and special purpose representation languages; we deal with propositional and first-order logic
- **Tell and Ask**: Adding new sentences (Tell) and asking, if sentences are true (told or infered)
- **Inference**: Deriving new from known sentences; core ability of knowledge-based agents
- **Background knowledge**: What the agent knows initially.

Description Levels

An agent can be described on different abstraction levels:

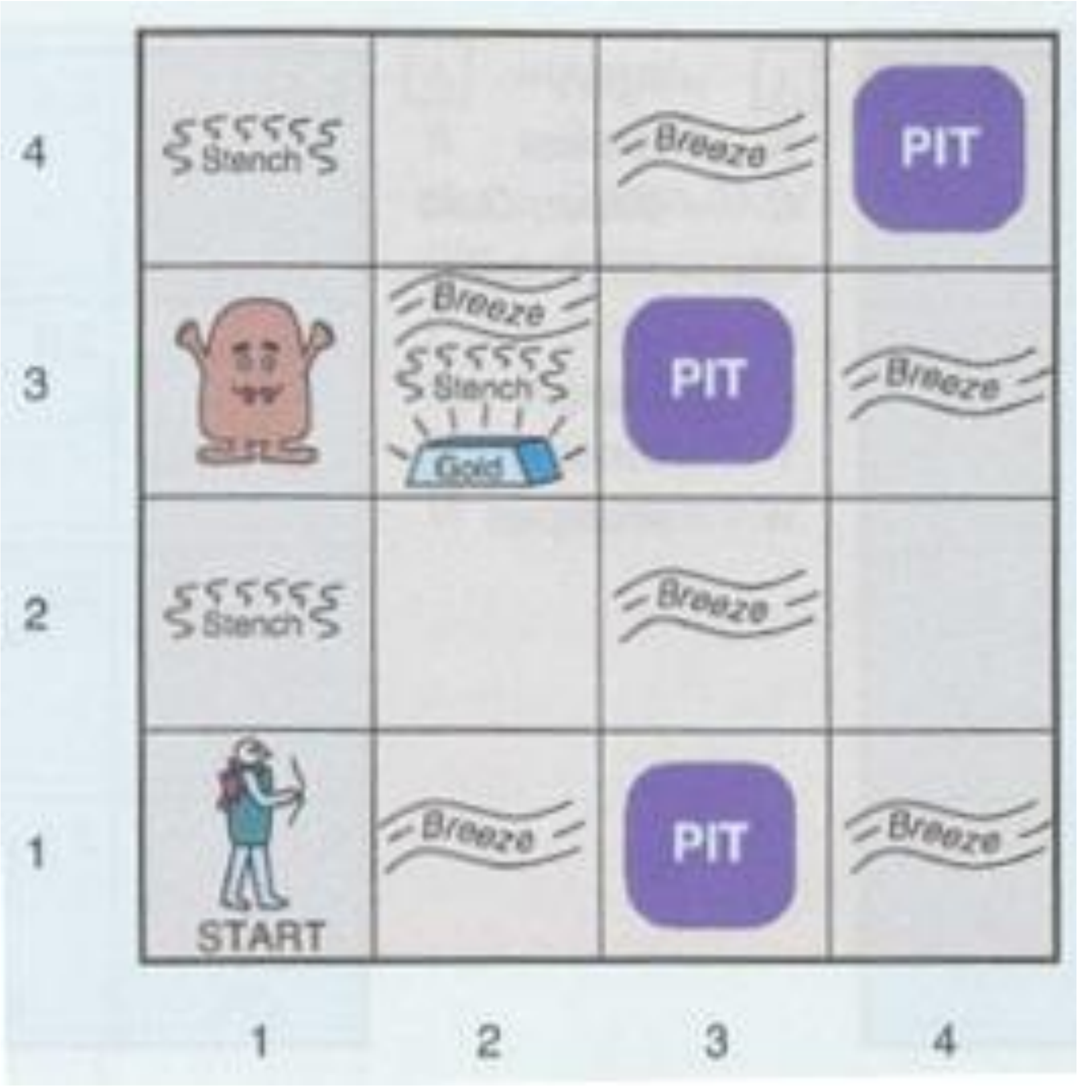
- **Knowledge level:** We need to specify, what the agent knows and what its goals are to determine its behaviour.
  - An automated car driving to a goal knows possible routes and chooses the best one based on traffic estimations.
- **Implementation level:** The implementation level contains all details (e.g. in a programming language), but we do not need to know them e.g. for communication with the agent.

- Building an agent on the knowledge level is called a **declarative approach** as opposed to a **procedural approach** encoding the behaviour directly into program code.

Wumpus World

The Wumpus world is a cave consisting fo rooms connected by pathways. The rooms may contain a dragon-like wumpus, equally dangerous pits, but also gold. Luckily, it is possible to sense a wumpus by a stench and pits by a breeze from neighbor rooms. The gold can be sensed by a glitter only from inside the room.

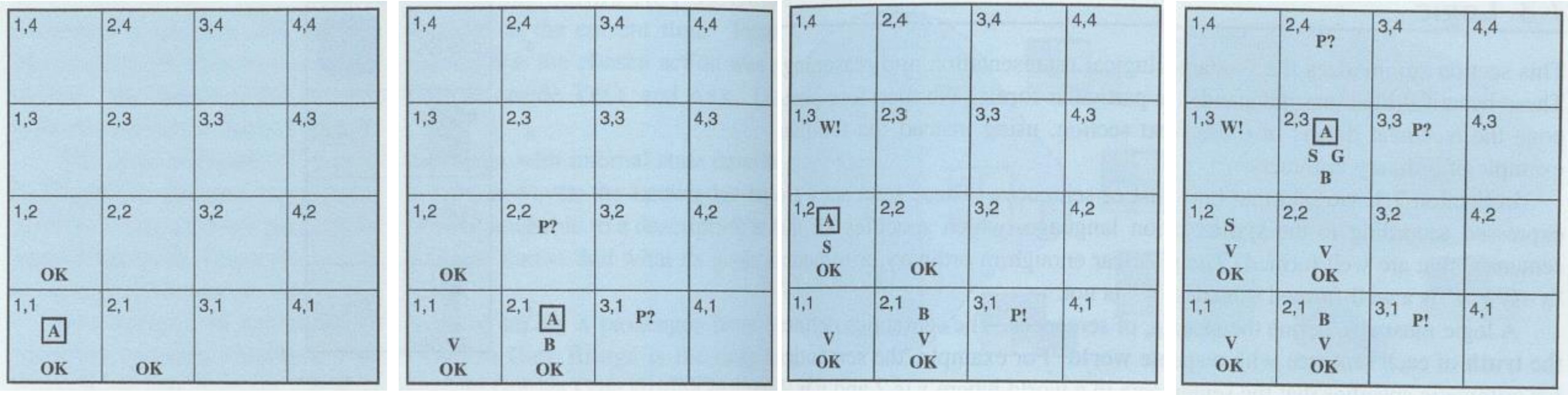
The agent can move from one room to a neighbor room and has an arrow beeing able to kill the wumpus, which the agent can sense by a scream. If moving in a room with the (living) wumpus or with a pit, the agent is dead.



PEAS Description of Wumpus World

- **Performance Measure:** 1000 points for the gold, -1000 points for being dead, -1 for each step and -10 for shooting the arrow.
- **Environment:** 4 x 4 matrix of rooms. The agent starts in field [1,1]. Locations of gold, wumpus and pits are randomly placed, with 20% of all rooms containing pits.
- **Actuators:** TurnLeft, TurnRight, Forward (having no effect in the direction of a wall), Die (in rooms with pits and the living wumpus), Grab (the gold), Shoot (allowed only once).
- **Sensors:** 5 boolean sensors for each room: Stench (for Wumpus), Breeze (for pits), Glitter (for gold), Bump (for hitting a wall), Scream (hearing the dying Wumpus)

Example Screens

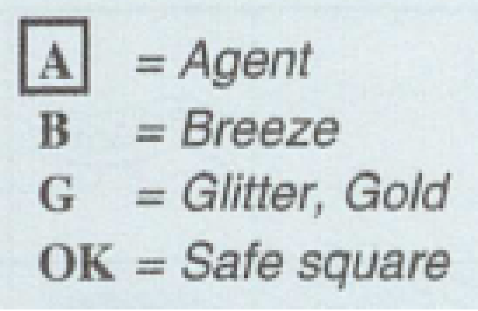


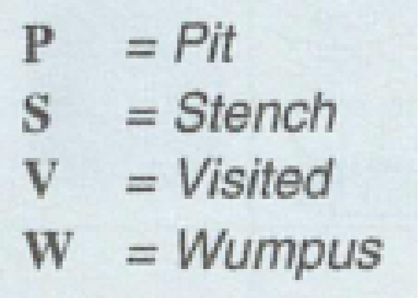
1 (left): Initial situation with percept [none, none, none, none, none]

2: After moving to [2,1]: [none, Breeze, none, none, none]

3: After moving back to [1,1] and to [1,2]: [Stench, none, none, none, none]

4: After moving to [2,2] and [2,3]: [Stench, Breeze, Glitter, none, none]





Basic Concepts of Logic

- **Syntax** of represetation language:
- **Semantic** of representation language:
- **Entailment** (logical implication):
- **Inference algorithm i:**
  - **sound:**
  - **complete:**
- **Model checking:**
- **Proof:**
- **Grounding:**

Basic Concepts of Logic

- **Syntax** of represetation language: Specifies well formed sentences, e.g. x+y=4 is well formed, but xy4 is not well formed.
- **Semantic** of representation language: Defines truth of sentences with respect to each pos-sible world (model), e.g. x+y=4 is true in a model with x=2 and y=2, but false, if x=1 and y=1.
- **Entailment** (logical implication): α |= β means, that in each model where α = true, β is also true, e.g. (x+y=4) |= (4=y+x).
- **Inference algorithm i:** WB $|-_{i}$ α means, that the inference algorithm i derives the sentence α from the knowledge base WB
    - **sound:** WB $|-_{i}$ α nur dann, wenn WB |= α
    - **complete:** i can derive all sentences, which are entailed
- **Model checking:** An inference algorithm enumerating all models for checking the conclusion (sound and complete)
- **Proof:** Sequence of inference steps for deriving a sentence
- **Grounding:** Connection between logic and real world: With sensors and with rules/learning

Methapor for Entailment and Inference

„Needle in haystick“

- Haystick: All sentences logically following from a knowledge base
- Needle: A particular sentence α

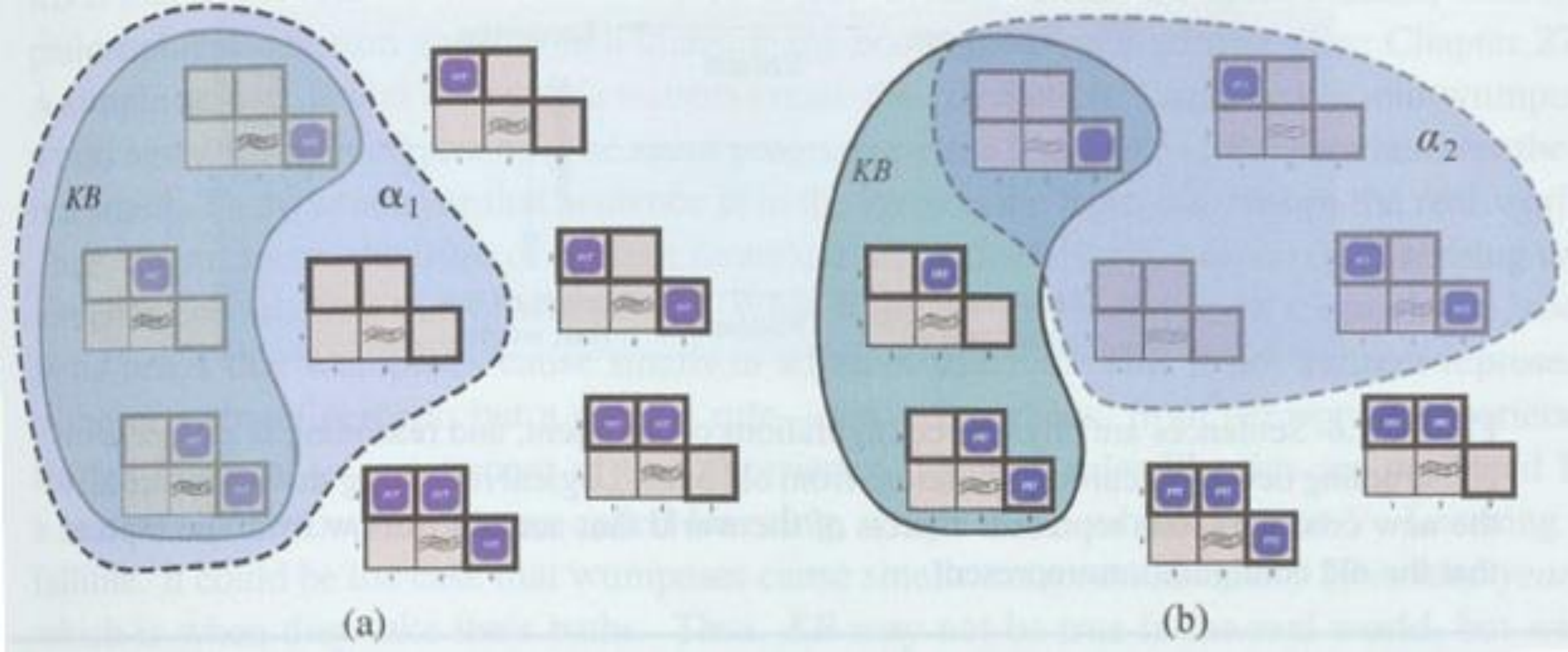
- **Entailment**: The needle α is in the haystick
- **Inference algorithm**: Procedure to find the needle
- **Sound inference algorithm**: Returns only needles being really in the haystick
- **Complete inference algorithm**: Returns all needles in the haystick
    - With finite haysticks, model checking is complete

Model Checking in Wumpus World

- Possible models for the presence of pits in squares [1,2], [2,2], [3,1].
- KB with observations of nothing in [1,1] and a breeze in [2,1] is shown by a solid line

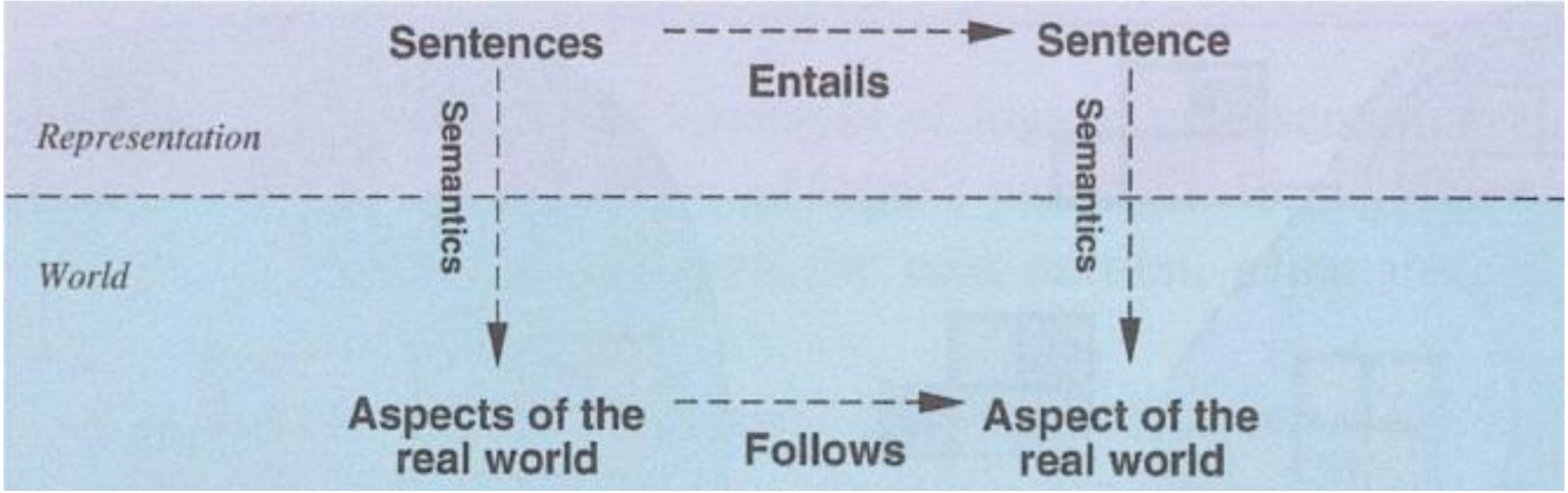
(a) Dotted line shows models of α_1 (no pit in [1,2]): in each model of KB: α_1 is valid → α_1 = true

(b) Dotted line shows models of α_2 (no pit in [2,2]): α_2 is in models of KB partly true, partly false → α_2 = unknown



Syntax and Semantic

If a knowledge base is true in the real world, than all derived sentences (with a sound inference algorithm) are also true in the real world.



Logic

- A logic consists of a formal system with syntax and semantic, from which inferences can be derived.

- Two important logics:
  - **Propositional logic** with facts and connectors
  - **First-order logic** with objects, predicates, connectors and quantors.

Propositional Logic

- Syntax
- Semantic
- Proof
  - By model checking
  - By inference

Syntax of Propositional Logic

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\, \mathit{Sentence} \,) \\
 & \mid & \neg\, \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Rightarrow\; \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Leftrightarrow\; \mathit{Sentence} \\
\\
\textsc{Operator Precedence} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Semantic of Propositional Logic

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*

- The table describes the rules in compact form, e.g.
  - If P is false and Q is false, then P $\wedge$ Q is false
  - etc.

Proof with Truth Table Enumeration

- For n boolean facts a truth table has 2^n entries.
- By enumeration, it is easy to check, whether a sentence is valid for all entries or models (tautology), for some entries (satisfiable), or for no entries (unsatisfiable).
- Example for a stepwise proof that the sentences $((P \vee H) \wedge \neg H) \Rightarrow P)$ is a tautology.

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*

- For checking, whether a sentence is true, given some facts in a knowledge base, we can construct the full truth table and check only those lines with the relevant facts.

Truth Table (TT) Enumeration Algorithm

- Each symbol in KB and α can have the values true and false. Enumerate all possible value combinations of the symbols and check, whether in all combinations, where KB is true, α is also true.
    - „PL-True?“ returns true, if a sentence holds within a model.
    - The variable „model“ represents a partial model, i.e. an assignment to some of the symbols

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

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, *symbols*, { })

function TT-CHECK-ALL(KB, α, *symbols*, *model*) **returns** *true* or *false*
  if EMPTY?(*symbols*) **then**
    if PL-TRUE?(KB, *model*) **then return** PL-TRUE?(α, *model*)
    else return *true*   // when KB is false, always return true
  else
    P ← FIRST(*symbols*)
    rest ← REST(*symbols*)
    return (TT-CHECK-ALL(KB, α, *rest*, *model* $\cup \{P = true\}$)
        and
        TT-CHECK-ALL(KB, α, *rest*, *model* $\cup \{P = false\}$))

Example for TT-Entails

- KB ist true if R_1 through R_5 is true (underlined) – just three rows of the 128 entries
  - R_1: $\neg P_{1,1}$; R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$; R_3: $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$; R_4: $\neg B_{1,1}$; R_5: $B_{2,1}$
- In these three rows, $P_{1,2}$ is false, so there is not pit in [1,2]

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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*

Propositional Theorem Proving

- Model checking is a sound and complete inference procedure, but may be slow: $O(2^n)$
- An alternative inference procedure is theorem proving with
  - logical equivalences
  - inference rules

Logical Equivalences

Easy to proof with truth table (as tautologies): α, β and γ denote arbitrary logical sentences

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

Inference Rules

- **Modus Ponens:** If $\alpha \Rightarrow \beta$ and α are given, sentence β can be infered

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

- **And-Elimination:** From a conjunction, any of the conjuncts can be infered

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

- **Unit-Resolution:** If a part of a disjunction is false, the rest must be true

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

- All logical equivalences (s. last slide) can also be used as inference rules.

Example for Using Inference Rules

- Same goal as in model checking example: Proof, that there is not pit in [1,2], given the KB R_1 through R_5:

R_1: $\neg P_{1,1}$

R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

R_3: $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

R_4: $\neg B_{1,1}$

R_5: $B_{2,1}$

1. Apply biconditional elimination to R_2 to obtain

$$R_6: \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

2. Apply And-Elimination to R_6 to obtain

$$R_7: \quad ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

3. Logical equivalence for contrapositives gives

$$R_8: \quad (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})).$$

4. Apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

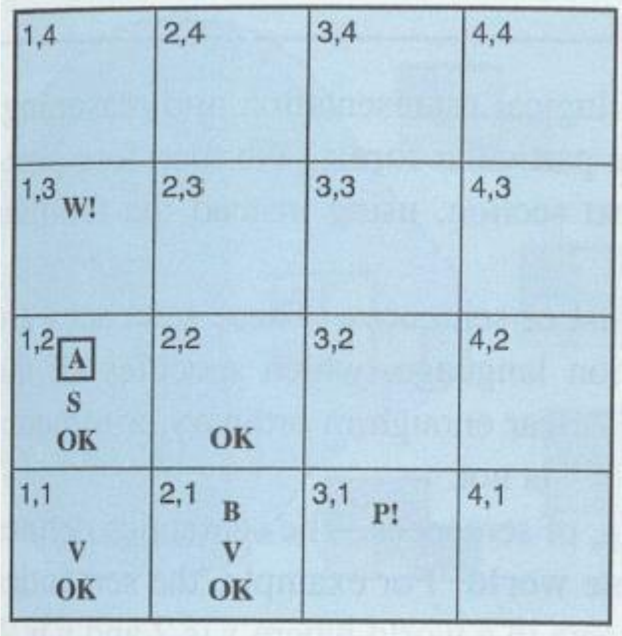
$$R_9: \quad \neg(P_{1,2} \vee P_{2,1}).$$

5. Apply De Morgan's rule, giving the conclusion

$$R_{10}: \quad \neg P_{1,2} \wedge \neg P_{2,1}.$$

That is, neither [1,2] nor [2,1] contains a pit.

Proof with Unit Resolution Rule



- Agent perceives in [1,1] nothing, in [2,1] a stench, but not a breeze, and in [2,1] a breeze, i.e. $\neg B_{1,2}$ and $B_{2,1}$
- Proof, that in [3,1] there is a pit, i.e. $P_{3,1}$
  - Facts in the knowledge base:
    - Since $B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$
    - $\neg B_{1,2} \Leftrightarrow \neg(P_{1,1} \vee P_{2,2} \vee P_{1,3})$ YIELD $\neg P_{1,1} \wedge \neg P_{2,2} \wedge \neg P_{1,3}$
      - $\neg P_{1,1}$
      - $\neg P_{2,2}$
    - $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$ YIELD $(P_{1,1} \vee P_{2,2} \vee P_{3,1})$
  - Derviations with unit resolution rule
    - $P_{1,1} \vee P_{2,2} \vee P_{3,1}$ AND $\neg P_{1,1}$ YIELD $P_{2,2} \vee P_{3,1}$
    - $P_{2,2} \vee P_{3,1}$ AND $\neg P_{2,2}$ YIELD $P_{3,1}$

Resolution

- Inference rules covered so far are sound, but we haven‘t discussed their completeness
- There is one single inference rule, which is sound and complete: **resolution**!
  - Generalization of unit resolution requiring sentences to be transformed in clause form
  - Unit resolution
    $$\frac{\ell_1 \vee \ell_2, \qquad \neg \ell_2}{\ell_1}$$
  - Resolution rule with two clauses:
    $$\frac{\ell_1 \vee \ell_2, \qquad \neg \ell_2 \vee \ell_3}{\ell_1 \vee \ell_3}$$
- General resultion rule (l_i and m_j are complementary, i.e. identical with and without negation)

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

Resolution (2)

- Resolution cannot enumerate all true sentences, but can decide, whether a sentence is false
  - proof by contradiction: in order to show (WB $|= \alpha$), show (WB $\wedge \neg\alpha$) is unsatisfiable.
    - Apply resolution algorithm

- Resolution only apply to clauses (i.e. disjunctions of literals)
  - Convert sentences in conjunction of clauses, so called **conjunctive normal form (CNF)**

Conjunctive Normal Form (CNF)

- Conjunction of disjunction of literals , e.g. $(l_{1,1} \vee \ldots \vee l_{1,k}) \wedge \ldots \wedge (l_{n,1} \vee \ldots \vee l_{n,m})$

$$
\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Fact} & \rightarrow & \mathit{Symbol} \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & \mathit{Fact} \mid (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{array}
$$

- A CNF clause such as $\neg A \vee \neg B \vee C$ can be written in definite clause form $A \wedge B \Rightarrow C$

Procedure for Converting in CNF

1. Replace all $\Leftrightarrow$ by replacing $(\alpha \Leftrightarrow \beta)$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
2. Replace all $\Rightarrow$ by replacing $(\alpha \Rightarrow \beta)$ with $(\neg\, \alpha \vee \beta)$
3. Replace all $\neg$ not in front of symbols, with 3 rules:

   $\neg(\neg\, \alpha) \equiv \alpha;$

   $\neg(\alpha \vee \beta) \equiv (\neg\, \alpha \wedge \neg\, \beta);$ (De Morgan)

   $\neg(\alpha \wedge \beta) \equiv (\neg\, \alpha \vee \neg\, \beta);$ (De Morgan)
4. Move $\vee$ with distributive law inwards

Example for Converting in CNF

Sentence to be converted: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Replace $\Leftrightarrow$ $\quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
2. Replace $\Rightarrow$ $\quad (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
3. Replace outer $\neg$ $\quad (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$
4. Move $\vee$ $\quad (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

Resolution Algorithm

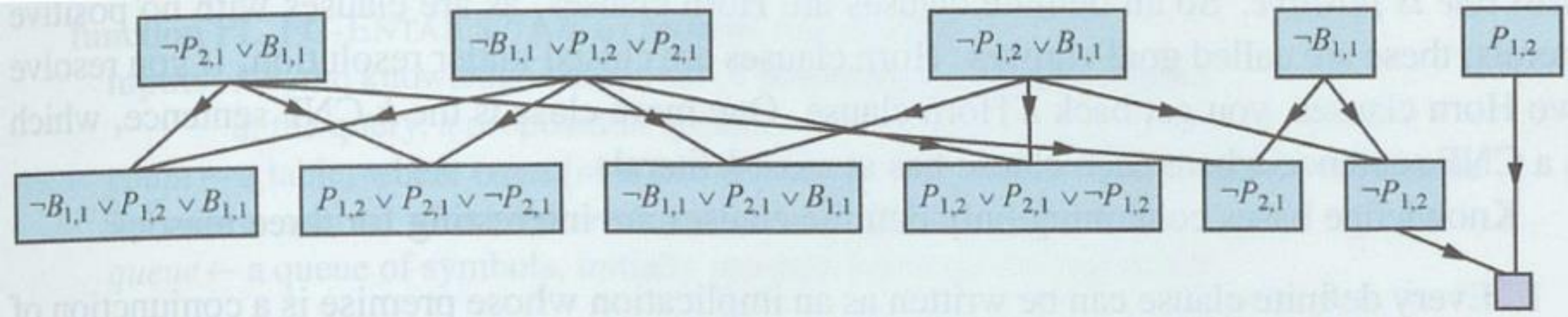
function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of $KB \land \neg\alpha$
  new ← { }
  while *true* **do**
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

Example for Resolution Algorithm

Goal: Show in start configuration of Wumpus world, that there is no pit in [1,2], because no breeze is in [1,1].

- The rule $B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$ is transformed in CNF together with the fact $\neg B_{1,1}$
- The goal $\neg P_{1,2}$ is negated and added to the clauses $P_{1,2}$
- The resolution algorithm is applied deriving many new clauses including the empty one
- Some clauses denoting „True“, e.g. $\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$, can be discarded immediately



Horn Clauses (Rules)

- Simplification of resolution: Use Horn clauses instead of (general) clauses
- Horn clause is a clause with at most one non-negated literal ($\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n \vee l$)
  - More intuitive notation as a rule with premise and conclusion: $l_1 \wedge l_2 \wedge \ldots \wedge l_n \Rightarrow l$
  - or as a fact (horn clause without non-negated literal): l
- Inference with horn clauses has linear complexity (to size of knowedge base)
  - Instead of exponential complexity in resolution algorithm

Two Algorithms for Inference with Horn Clauses

- Forward chaining:
  - Check for all rules, if its premise is satisfied
  - If so, add conclusion (fact) to knowledge base
  - Until goal is infered or no new facts can be infered
- Backward chaining:
  - Start with the goal (question)
  - Check all rules with the goal in the conclusion
  - If a fact in a premise of a rule is unknown, iterate with that fact as subgoal
  - Terminate if goal is infered

Forward Chaining Algorithm PL-FC-Entails?

- Not yet processed symbols are noted in a variable „queue“ and processed only once
- Rules have a counter for unsatisfied symbols in its premise, which is continuously decremented; if the counter = 0, the rules „fires“, i.e. its conclusion is added to queue

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
    p ← POP(queue)
    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 queue
  return false
```

Example for Forward Chaining

- Knowledge base as set of ruels and as And-Or-Graph

R1: $P \Rightarrow Q$

R2: $L \land M \Rightarrow P$

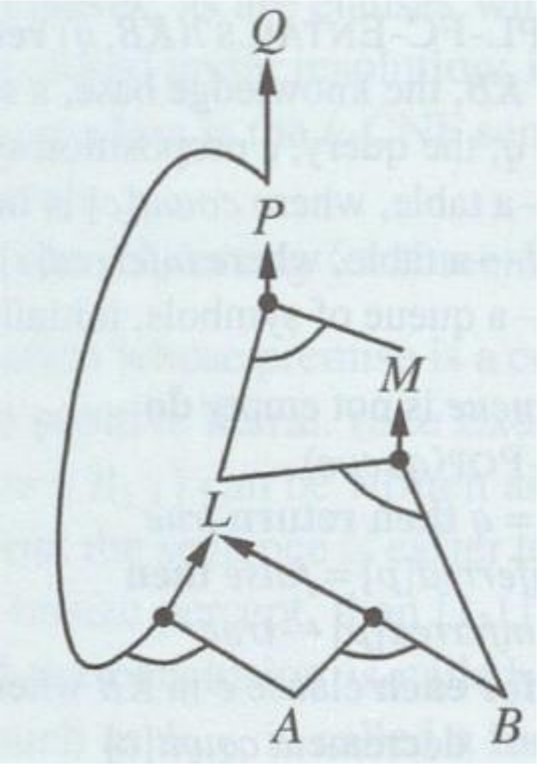
R3: $B \land L \Rightarrow M$

R4: $A \land P \Rightarrow L$

R5: $A \land B \Rightarrow L$

A

B



1. PL-PC-ENTAILS (KB, Q)
2. queue ← (A, B)
3. p ← A
4. queue ← (B)
5. inferred (A) = true
6. count (R4) = 2 – 1 = 1
7. count (R5) = 2 – 1 = 1
8. p ← B
9. queue ← ( )
10. inferred (B) = true
11. count (R3) = 2 – 1 = 1
12. count (R5) = 1 – 1 = 0
13. queue ← (L)
14. p ← L
15. queue ← ( )
16. inferred (L) = true
17. count (R2) = 2 – 1 = 1
18. count (R3) = 1 – 1 = 0
19. queue ← (M)
20. …

Backward Chaining Algorithm with Example

- Essentially And-Or-Graph-Search with goal to answer a query (ignoring the conditional plan)

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
  return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
  if problem.IS-GOAL(state) then return the empty plan
  if IS-CYCLE(path) then return failure
  for each action in problem.ACTIONS(state) do
     plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
     if plan ≠ failure then return [action] + plan]
  return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
  for each s_i in states do
     plan_i ← OR-SEARCH(problem, s_i, path)
     if plan_i = failure then return failure
  return true
```

Rule List:	R3: B ∧ L ⇒ M	**Fact List:**
R1: P ⇒ Q	R4: A ∧ P ⇒ L	A
R2: L ∧ M ⇒ P	R5: A ∧ B ⇒ L	B

1. Query (Goal) : Q
2. Rule list (Q) = (R1) {OR-Search}
3. Goal (R1) = (P) {AND-Search}
4. Rule list (P) = (R2) {OR-Search}
5. Goal (R2) = (L, M) {AND-Search}
6. Rule list (L) = (R4, R5) {OR-Search}
7. Goal (R4) = (A, P) {AND-Search}
8. A = True
9. P = {is on path; failure}
10. Goal (R5) = (A, B) {AND-Search}
11. A = True, B = True
12. L = True,
13. Rule list (M) = (R3) {OR-Search}
14. Goal (R3) = (B, L) {AND-Search}
15. B = True; L = True
16. M = True; P = True; Q = True

General Effective Inference with Arbitrary Clauses

- Goal: Checking satisfiability for a set of clauses: The **SAT-problem** for propositional logic
- Two algorithms:
  - **DPLL:** Davis-Putnam-Logemann-Loveland Algorithm
    - Complete Backtracking Algorithm with intelligent termination criteria
  - **WALKSAT:**
    - Hill-Climbing Search; not complete, but more efficient

DPLL

- Input: Sentences in CNF
- Like Backtracking and TT-Entails?, it is essentially a recursive, depth-first enumeration of possible models with three improvement over TT-Entails:
  - **Early termination:**
    - A (disjunctive) clause is true, if some literal is true
    - A (conjunctive) sentence is true, if all clauses are true
      - e.g. $(A \vee C) \wedge (A \vee B)$ = true, if A = true
    - A sentence is false, if a single clause is false
  - **Pure symbol heuristic:**
    - A symbol is „pure“, if it appears with the same „sign“ in all clauses (e.g. always negated). Pure symbols are preset: false for negated symbols, otherwise true.
  - **Unit clause heuristic:**
    - A unit clause contains just one literal. All such literals are tried first to reduce the search space (e.g. if setting such a literal to true and this yields a failure, it must be false).

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

General Tricks of SAT Solvers

Used also in DPLL algorithm (last slide) and in many other algorithms too:

- **Component analysis** (e.g. Tasmania in CSP)**:** If there are disjoint subsets (sharing no unassigned variables) either at the beginning or after assignment of some variables, then work on these subsets seperately
- **Variable and value ordering** (similar to MRV-, degree- and least-constraining value heuristic in CSP): E.g.: Transfer of degree heuristic to choose the variable that appears most frequently over all remaining clauses
- **Intelligent backtracking** (similar to CSP): Instead of chronological backtracking, backjumping to the relevant point of conflict and conflict clause learning (record conflicts to avoid repeating them as fas as memory is available)
- **Random restarts** (similar to hill climbing): If a run appears to be making no progress, try a complete restart with different random choices
- **Clever indexing**: Items used often (e.g. set of clauses with variable X_i as positive literal) should be indexed (with dynamic update)

Local Search Algorithm for Satisfiabilty

- **Basic idea (Hill Climbing, Simulated Annealing):**
  - Set randomly complete assignment for all variables (a model) and change value of variables until all clauses are satisfied or a threshhold is reached
- **Criteria for selection of variable** to be changed:
  1. It should be part of at least one unsatisfied clause
  2. It should reduce the maximal number of not satisfied clauses (like min-conflicts heuristic)
  3. It should contain some randomness (like simulated annealing, to avoid local minima)
- Implementation of criteria in **WalkSAT algorithm**
  - Choose unsatisfied clause (criteria 1)
  - Choose with a certain probability p a symbol (criteria 3) or choose that symbol, that maximizes the number of unsatisfied clauses (criteria 2)

WalkSAT algorithm

(many versions exist)

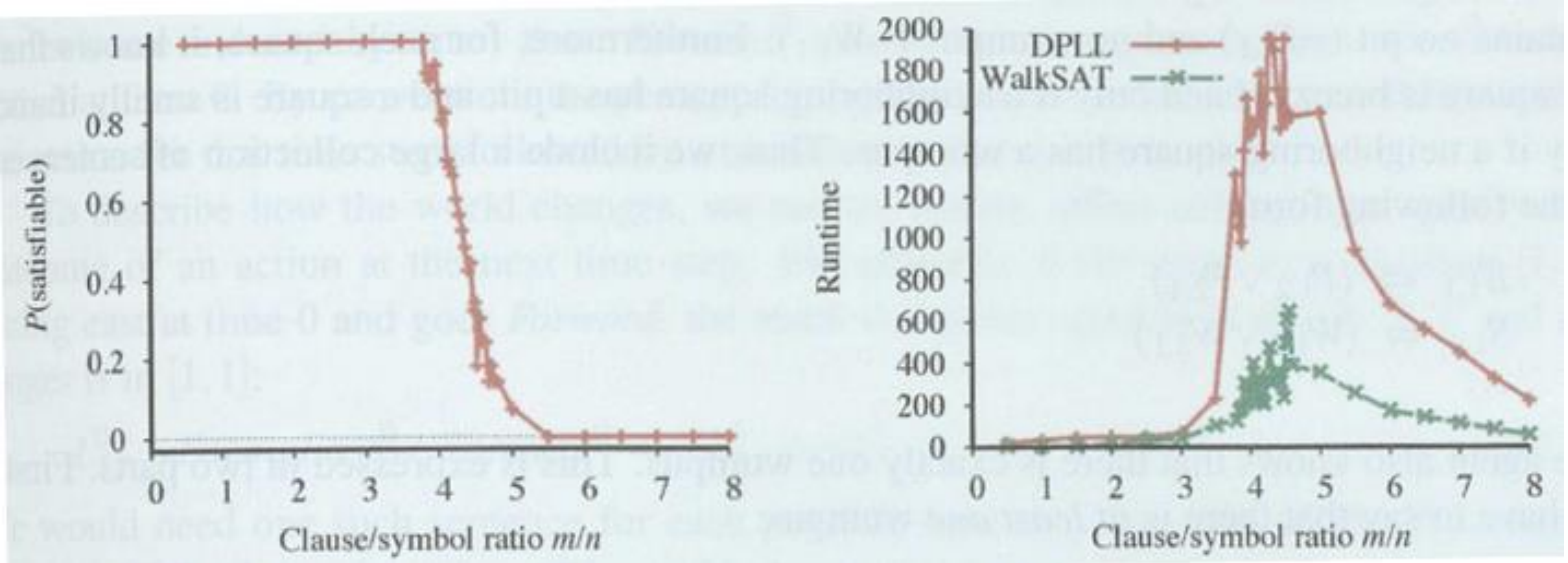
```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of value flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for each 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
     if RANDOM(0, 1) ≤ p then
        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
```

The Landscape of Random SAT Problems

- Some SAT problems are harder than others
  - Easy problems are underconstrained or overconstrained:
    - Underconstrained: Many solutions exist: e.g. the n-queens problem
      - e.g.: $(\neg D \lor B \lor C) \land (B \lor \neg A \lor \neg C) \land (\neg C \lor \neg B \lor E) \land (E \lor \neg D \lor B) \land (B \lor E \lor \neg C)$
    - Overconstrained: Many clauses relative to the number of variables,
      - Likely that no solution exist
  - Since SAT problems are NP-Complete, there are also difficult problems
  - Can be expressed as clause to symbol ratio
    - The example above contains 5 clauses with 5 symbols, i.e. clause/symbol ratio = 1
    - Randomly generat SAT-problems, which are difficult, have a clause/symbol ratio = 4,3

Performance of DPLL and WalkSAT on random 3-CNF

- Probability, that a random 3-CNF sentences problem with 50 symbols is satisfiable as a function of the clause/symbol ratio (varying from 1 to 8; left)
- Graph of the median run-time (in number of iterations) for DPLL and WalkSAT (right)
  - WalkSAT solved all problems correctly



Agents Based on Propositional Logic

- General idea of a hybrid agent:
  - Infer status of each square with logical inference based on general rules and percepts
  - Planning: If the agent perceives a glitter, he takes the gold, otherwise he moves to a safe square, otherwise he decides whether he can make a square safe by shooting his arrow, otherwise he moves to an unsafe square (but not a sure death)
  - Acting: The shortest path from his current position to the selected square is found by a A* search with safe squares only.
- Problems in propositional logic
  - Each square and each time step have to be handled separately causing tremendous overhead
    - Much more elegant in first-order logic
  - The transition from one situation to the next is best handled outside the logic world

Famous Problems

- Frame problem:
  - An action axiom not only has to decide what has changed, but also what remains unchanged
- Computational expense increases linear with time
  - History of percepts gets longer and longer over time

Frame Problem

- If every proposition is mentioned in every action axiom, this is very inefficient
  - Solution:
    - Write axioms not over actions, but over fluents (propositions, that can change)
    - For each fluent, define what actions can change it; otherwise it stays the same
      - e.g. $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^{t} \wedge \neg Shoot^{t})$
      - **„Successor-state axioms“**

Belief State

- The history of percepts should be replaced by a belief state
  - Some representation of the set of all possible current states of the world
  - Example: $\text{WumpusAlive}^1 \wedge L_{2,1}{}^1 \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2})$
  - Size of exact belief state: With n fluents there are 2^n possible physical states (i.e. assignments of truth values to those symbols)
  - The belief state is the powerset (set of all subsets) of the set of physical states, i.e. 2^{2^n}
    - Too large to be represented exactly
- Approximate state estimation
  - With logical expressions (see example above)
    - Popular representation: Just with conjunctions of literals, i.e. 1-CNF formulas
    - May lose some information, e.g. the disjunction in the example