Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Knowledge Representation: Propositional Logic

Today

- Starting today we will design agents that can form **representations** of a complex world, use a process of **inference** to derive new information about the world, and use that information to **deduce** what to do.
- They are called **knowledge-based agents** – combine and recombine information about the world with current observations to uncover hidden aspects of the world and use them for action selection.
- We need to know:
  - how to represent **knowledge?**
  - how to **reason** over that knowledge?

Knowledge-based agent

- A knowledge-based agent uses a **knowledge base** – a set of sentences expressed in a given language – that can be updated by operation TELL and can be queried about what is known using operation ASK.
- Answers to queries may involve **inference** – that is deriving new sentences from old (inserted using the TELL operations).

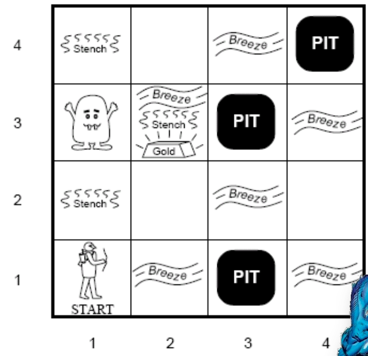
```
function KB-AGENT(percept) returns an action
   static: KB, a knowledge base
           t, a counter, initially 0, indicating time
   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   action ← ASK(KB, MAKE-ACTION-QUERY(t))
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

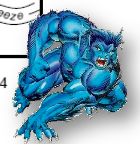
knowledge base contains information about observations as well as about own actions

inference will help the agent to select an action even if information about the world is incomplete

The Wumpus world – a running example

- A cave consisting of rooms connected by passageways, inhabited by the terrible **wumpus**, a beast that eats anyone who enters its room, containing rooms with bottomless **pits** that will trap anyone, and a room with a heap of **gold**.





- The agent will perceive a **Stench** in the directly (not diagonally) adjacent squares to the square containing the wumpus.
- In the squares directly adjacent to a pit, the agent will perceive a **Breeze**.
- In the square where the gold is, the agent will perceive a **Glitter**.
- When an agent walks into a wall, it will perceive a **Bump**.
- The wumpus can be shot by an agent, but the agent has only one arrow.
  - Killed wumpus emits a woeful **Scream** that can be perceived anywhere in the cave.

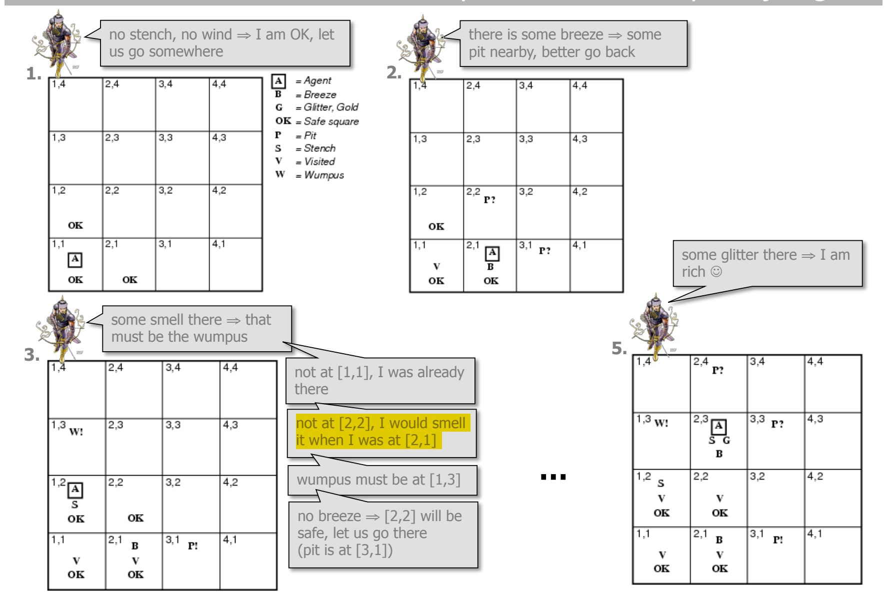
The Wumpus world – agent's view

- **Performance measure**
  - +1000 points for climbing out of the cave with the gold
  - -1000 for falling into a pit or being eaten by the wumpus
  - -1 for each action taken
  - -10 for using up the arrow
- **Environment**
  - 4 × 4 grid of rooms, the agent starts at [1,1] facing to the right
- **Senzors**
  - Stench, Breeze, Glitter, Bump, Scream
- **Actuators**
  - move Forward, TurnLeft, TurnRight
  - Grab, Shoot, Climb

The Wumpus world - environment

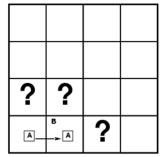
- **Fully observable?**
  - NO, the agent perceives just its direct neighbour (partially observable)
- **Deterministic?**
  - YES, the result of action is given
- **Episodic?**
  - NO, the order of actions is important (sequential)
- **Static?**
  - YES, the wumpus and pits do not move
- **Discrete?**
  - YES
- **One agent?**
  - YES, the wumpus does not act as an agent, it is merely a property of environment

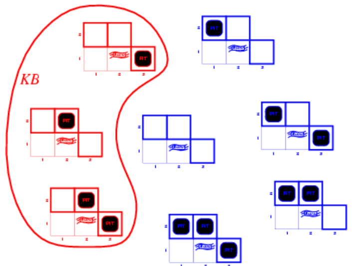
The Wumpus world – the quest for gold



The Wumpus world – possible models

- Assume a situation when there is no percept at [1,1], we went right to [2,1] and feel Breeze there.

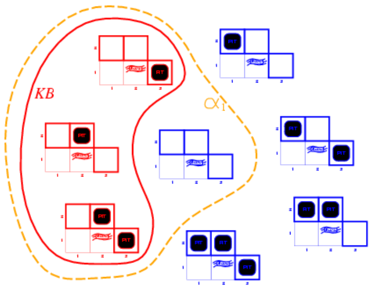




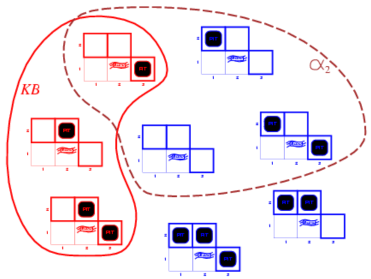
  - For pit detection we have 8 (=2^3) possible models (states of the neighbouring world).
  - Only three of these models correspond to our knowledge base, the other models conflict the observations:
    - no percept at [1,1]
    - Breeze at [2,1]

The Wumpus world – some consequences

- **Let us ask whether the room [1,2] is safe.**
- Is information α_1 = „[1,2] is safe" entailed by our representation?
- we compare models for KB and for α_1
- every model of KB is also a model for α_1 so α_1 is entailed by KB



- **And what about room [2,2]?**
- we compare models for KB and for α_2
- some models of KB are not models of α_2
- α_2 is not entailed by KB and we do not know for sure if room [2,2] is safe

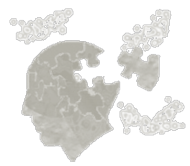


Inference in general

How to implement inference in general?

We will use **propositional logic**. Sentences are propositional expression and a knowledge base will be a conjunction of these expressions.

- **Propositional variables** describe the properties of the world
  - $P_{i,j}$ = true iff there is a pit at [i, j]
  - $B_{i,j}$ = true if the agent perceives Breeze at [i, j]
- **Propositional formulas** describe
  - known information about the world
    - $\neg P_{1,1}$ no pit at [1, 1] (we are there)
  - general knowledge about the world (for example, Breeze means a pit in some neighbourhood room)
    - $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
    - $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
    - …
  - observations
    - $\neg B_{1,1}$ no Breeze at [1, 1]
    - $B_{2,1}$ Breeze at [2, 1]
- We will be using **inference** for propositional logic.



Propositional logic at glance

- **Syntax** defines the allowable sentences.
  - a propositional variable (and constants true and false) is an (atomic) sentence
  - two sentences can be connected via logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ to get a (complex) sentence
- **Semantics** defines the rules for determining the truth of a sentence with respect to a particular model.
  - **model** is an assignment of truth values to all propositional variables
  - an atomic sentence P is true in any model containing P=true
  - semantics of complex sentences is given by the truth table

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*

Propositional logic – entailment and inference

- M is a **model** of sentence α, if α is true in M.
  - The set of models for α is denoted M(α).
- **entailment: KB $\models \alpha$** means that α is a logical consequence of KB
  - KB entails α iff M(KB) $\subseteq$ M(α)
- We are interested in **inference methods**, that can find/verify consequences of KB.
  - KB $\vdash_i \alpha$ means that algorithm i infers sentence α from KB
  - the algorithm is **sound** iff KB $\vdash_i \alpha$ implies KB $\models \alpha$
  - the algorithm is **complete** iff KB $\models \alpha$ implies KB $\vdash_i \alpha$

Inference methods

- There are basically two classes of inference algorithms.
  - **model checking**
    - based on enumeration of a truth table
    - Davis-Putnam-Logemann-Loveland (DPLL)
    - local search (minimization of conflicts)
  - **inference rules**
    - theorem proving by applying inference rules
    - a resolution algorithm

Enumeration

```
function TT-ENTAILS?(KB, α) returns true or false
   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
   else do
      P ← FIRST(symbols); rest ← REST(symbols)
      return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model)) and
             TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model))
```

The Wumpus world
α_1 = „[1,2] is safe" = „$\neg P_{1,2}$" is entailed by KB, as $P_{1,2}$ is always false for models of KB and hence there is no pit at [1,2]

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
false	false	false	false	false	false	false	false	true
false	false	false	false	false	false	true	false	true
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	true	false	false	false	false	false	false	true
false	true	false	false	false	false	true	true	true
false	true	false	false	false	true	false	true	true
false	true	false	false	false	true	true	true	true
false	true	false	false	true	false	false	false	true
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	true	true	true	true	true	true	false	false

- We simple explore all the models using the **generate and test** method.
- For each model of KB, it must be also a model for α.

A bit of logic

- Sentence (formula) is **satisfiable** if it is true in, or satisfied by, *some* model.
  Example: A ∨ B, C
- Sentence (formula) is **unsatisfiable** if it is not true in *any* model.
  Example: A ∧ ¬A
- Entailment can then be implemented as checking satisfiability as follows:
  KB ├ α if and only if (KB ∧ ¬α) is unsatisfiable.
  - proof by **refutation**
  - proof by **contradiction**
- Verifying if α is entailed by KB can be implemented as the satisfiability problem for the formula (KB ∧ ¬α).
  Usually the formulas are in a **conjunctive normal form** (CNF)
  - **literal** is an atomic variable or its negation
  - **clause** is a disjunction of literals
  - **formula** in CNF is a conjunction of clauses
  Example: (A ∨ ¬B) ∧ (B ∨ ¬C ∨ ¬D)

  Each propositional sentence (formula) can be represents in CNF.

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

DPLL

Davis, Putnam, Logemann, Loveland

– a sound and complete algorithm for verifying satisfiablity of formulas in a CNF (finds its model)

```
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, [P = value|model])
   P, value ← FIND-UNIT-CLAUSE(clauses, model)
   if P is non-null then return DPLL(clauses, symbols–P, [P = value|model])
   P ← FIRST(symbols); rest ← REST(symbols)
   return DPLL(clauses, rest, [P = true|model]) or
          DPLL(clauses, rest, [P = false|model])
```

Early termination for partial models
- clause is true if any of its literals is true
- formula is not true if any of its clauses is not true

Pure symbol heuristics
- a pure symbol always appears with the same „sign" in all clauses
- the corresponding literal set to true

Unit clause heuristics
- a unit clause is a clause with just one literal
- the corresponding literal set to true

branching for backtracking

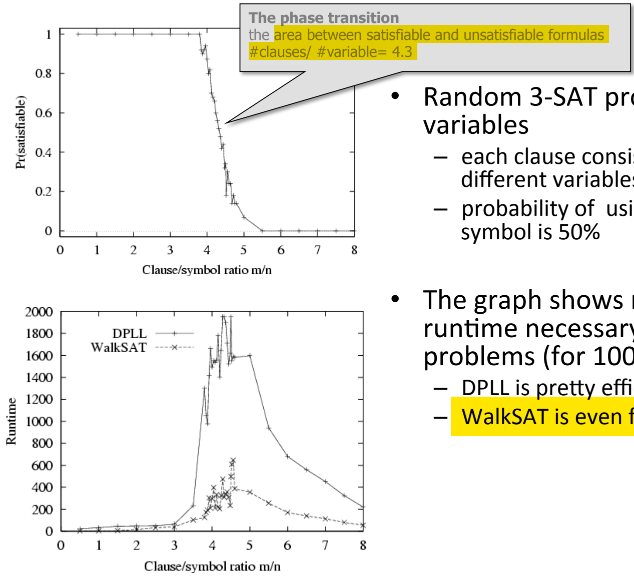
WalkSAT

- **Hill climbing merged with random walk**
  - minimizing the number of conflict (false) clauses
  - one local step corresponds to swapping a value of selected variable
  - **sound, but incomplete algorithm**

```
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
           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
```

WalkSAT vs. DPLL



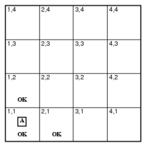
- Random 3-SAT problem with 50 variables
  - each clause consists of three different variables
  - probability of using a negated symbol is 50%
- The graph shows medians of runtime necessary to solve the problems (for 100 problems)
  - DPLL is pretty efficient
  - WalkSAT is even faster

Resolution principle

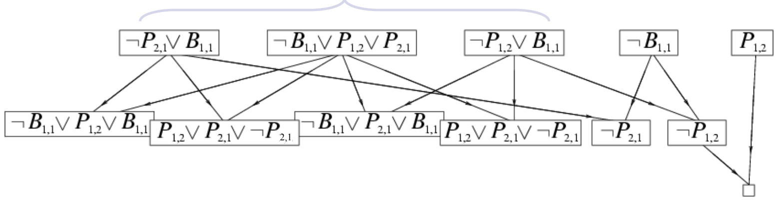
- The resolution algorithm proves unsatisfiability of the formula (KB ∧ ¬α) converted to a CNF. It uses **a resolution rule** that resolves two clauses with complementary literals (P and ¬P) to produce a new clause:

$$\frac{l_1 \vee \dots \vee l_k \qquad m_1 \vee \dots \vee m_n}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

  where l_i and m_j are the complementary literals
- The algorithms stops when
  - no other clause can be derived (then $\neg$ KB $\vdash \alpha$)
  - an empty clause was obtained (then KB $\vdash \alpha$)
- **Sound and complete algorithm**



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



Resolution algorithm

- For each pair of clauses with some complementary literals produce all possible resolvents. They are added to KB for next resolution.
  - an empty clause corresponds to false (an empty disjunction)
    - the formula is unsatisfiable
  - we reached a fixed points (no new clauses added)
    - formula is satisfiable and we can find its model

      take the symbols P_i one be one
      1. if there is a clause with $\neg P_i$ such that the other literals are false with the current instantiation of $P_1,\dots,P_{i-1}$, then P_i = false
      2. otherwise P_i = true

```
function PL-RESOLUTION(KB, α) returns true or false
   clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
   new ← { }
   loop do
      for each Ci, Cj in clauses do
          resolvents ← PL-RESOLVE(Ci, Cj)
          if resolvents contains the empty clause then return true
          new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

Horn clauses

- Many knowledge bases contain clauses of a special form – so called **Horn clauses**.
  - Horn clause is a disjunction of literals of which at most one is positive

  Example: $C \wedge (\neg B \vee A) \wedge (\neg C \vee \neg D \vee B)$
  - Such clauses are typically used in knowledge bases with sentences in the form of an implication (for example Prolog without variables)

  Example: $C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$
- We will solve the problem if a given propositional symbol – **query** – can be derived from the knowledge base consisting of Horn clauses only.
  - we can use a special variant of the resolution algorithm running in linear time with respect to the size of KB
  - **forward chaining** (from facts to conclusions)
  - **backward chaining** (from a query to facts)

Forward chaining

- From the known facts we derive using the Horn clauses all possible consequences until there are some new facts or we prove the query.
- This is a **data-driven method**.

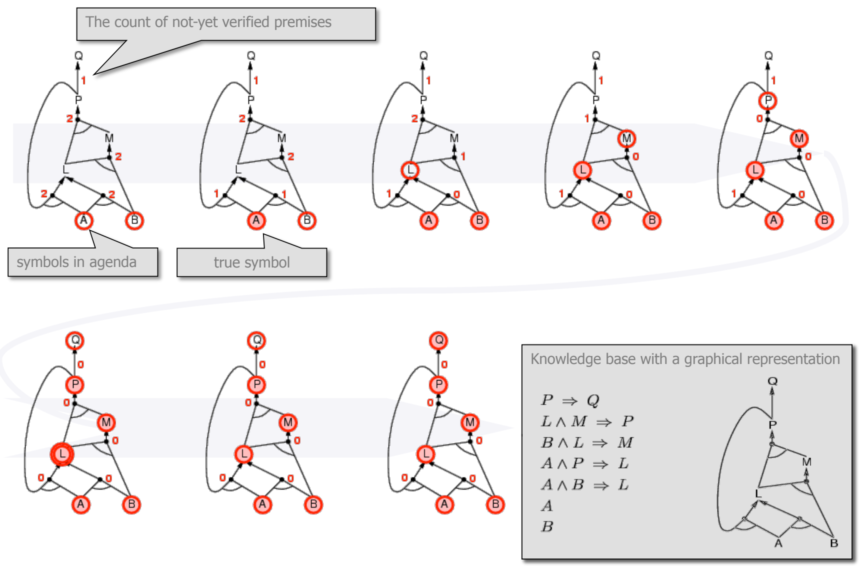
```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known to be true

  while agenda is not empty do
     p ← POP(agenda)
     unless inferred[p] do
        inferred[p] ← true
        for each Horn clause c in whose premise p appears do
           decrement count[c]
           if count[c] = 0 then do
              if HEAD[c] = q then return true
              PUSH(HEAD[c], agenda)
  return false
```

For each clause we keep thee number of not yet verified premises that is decreased when we infer a new fact. The clause with zero unverified premises gives a new fact (from the head of the clause).

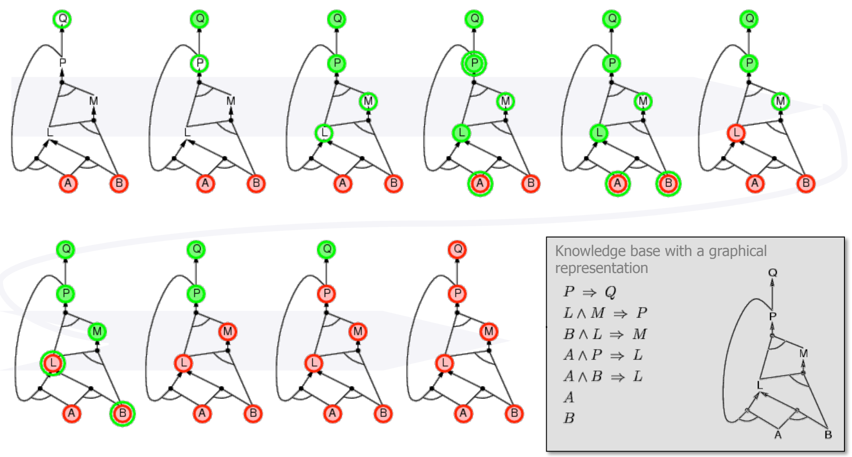
- **sound and complete** algorithm for Horn clauses
- **linear time complexity** in the size of knowledge base

Forward chaining in example



Backward chaining

- The query is decomposed (via the Horn clause) to sub-queries until the facts from KB are obtained.
- **Goal-driven reasoning**.



The Wumpus world – knowledge base

- For simplicity we will represent only the „physics“ of the wumpus world.
  - we know that
    - $\neg P_{1,1}$
    - $\neg W_{1,1}$
  - we also know why and where breeze appears
    - $B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y})$
  - and why a smell is generated
    - $S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y})$
  - and finally one “hidden” information that there is a single Wumpus in the world
    - $W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,4}$
    - $\neg W_{1,1} \vee \neg W_{1,2}$
    - $\neg W_{1,1} \vee \neg W_{1,3}$
    - …

- We should also include information about the **agent**.
  - where the agent is
    - $L_{1,1}$
    - FacingRight[1]
  - and what happens when agent performs actions
    - $L^{t}_{x,y} \wedge \text{FacingRight}^{t} \wedge \text{Forward}^{t} \Rightarrow L^{t+1}_{x+1,y}$
    - we need an upper bound for the number of steps and still it will lead to a huge number of formulas

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz

The Wumpus world – an agent

function PL-WUMPUS-AGENT(*percept*) **returns** an *action*
  inputs: *percept*, a list, [*stench*, *breeze*, *glitter*]
  static: *KB*, initially containing the “physics” of the wumpus world
    x, *y*, *orientation*, the agent's position (init. [1,1]) and orient. (init. *right*)
    visited, an array indicating which squares have been visited, initially *false*
    action, the agent's most recent action, initially null
    plan, an action sequence, initially empty

  update *x*, *y*, *orientation*, *visited* based on *action*
  if *stench* **then** TELL($KB, S_{x,y}$) **else** TELL($KB, \neg S_{x,y}$)
  if *breeze* **then** TELL($KB, B_{x,y}$) **else** TELL($KB, \neg B_{x,y}$)
  if *glitter* **then** *action* ← *grab*
  else if *plan* is nonempty **then** *action* ← POP(*plan*)
  else if for some fringe square [*i*,*j*], ASK($KB, (\neg P_{i,j} \wedge \neg W_{i,j})$) is *true* **or**
    for some fringe square [*i*,*j*], ASK($KB, (P_{i,j} \vee W_{i,j})$) is *false* **then do**
    plan ← A*-GRAPH-SEARCH(ROUTE-PB([*x*,*y*], *orientation*, [*i*,*j*], *visited*))
    action ← POP(*plan*)
  else *action* ← a randomly chosen move
  return *action*

Include information about observations

Try to find a safe room located in thefringe.

Or at least a room that is no provably unsafe.

Find a sequence of actions moving the agent to the selected room via known rooms.