Propositional Logic - Part 2

Outline

In this lecture we will present algorithms for two problems:

- **Satisfiability (SAT)**

  Given a propositional logic sentence ϕ, is it satisfiable?

- **Entailment**

  Given a propositional knowledge base KB and a propositional logic sentence α, do we have KB $\models \alpha$?

The algorithm TT-Entails

We start with the algorithm TT-Entails for the entailment problem.

TT-Entails performs a recursive enumeration of the finite number of assignments of truth values *true* and *false* to the symbols of the given sentence (i.e., it is a truth-table enumeration algorithm).

TT-Entails is **sound** because it implements directly the definition of entailment, and **complete** because it works for any KB and α and always terminates (since there only finitely many interpretations to examine).

The algorithm TT-ENTAILS (cont'd)

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

  $symbols \leftarrow$ a list of the proposition symbols in KB and α
  return TT-CHECK-ALL($KB, \alpha, symbols, \{\}$)

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

The algorithm TT-ENTAILS (cont'd)

Important note: the term **model**, as used in TT-ENTAILS (and Section 7.4.4 of the book), is essentially the term **interpretation** of my previous lecture. I have used the term model in a different way: **a satisfying interpretation!**

TT-ENTAILS is a **model checking** algorithm because it enumerate all possible models of *KB* and checks whether they are also models of α.

In TT-ENTAILS, the function PL-TRUE?(*KB*, *model*) returns *true* if the sentence *KB* is satisfied by the interpretation *model*.

The variable *model* represents a partial interpretation — an assignment of truth values *true* or *false* to some of the symbols.

The keyword **and** in the algorithm is an infix Boolean function symbol of the pseudocode programming language, not an operator in propositional logic.

Computational complexity of TT-Entails

TT-Entails has worst-case time complexity $O(2^n)$ where n is the number of propositional symbols in KB and α. The worst-case space complexity is $O(n)$ because the enumeration is depth-first.

As we have already said, the entailment problem for propositional logic is **co-NP-complete**. This means that **every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input**.

Effective propositional model checking

We will now describe two families of efficient algorithms for SAT based on **model checking**:

- The **Davis-Putnam** algorithm which is based on backtracking search.
- The WalkSat algorithm which is based on local hill-climbing search.

Effective propositional model checking (cont'd)

You should not be surprised by the fact that the algorithms that we will discuss belong to the same families of algorithms that we saw for constraint satisfaction problems (CSPs) given that SAT can be expressed as a CSP as we discussed earlier.

The two algorithms that we will discuss are also important in their own right because **many combinatorial problems in Computer Science can be reduced to checking the satisfiability of a propositional sentence.**

Any improvement in satisfiability algorithms is expected to have huge consequences for our ability to handle complexity in general.

A complete backtracking algorithm for SAT

The algorithm we will present is sometimes called the **Davis-Putnam** algorithm since it was invented in 1960 by Martin Davis and Hilary Putnam. The version we will present was described by Davis, Logemann and Loveland in 1962, so we will use the acronym **DPLL** for it.

DPLL works with a sentence after transforming it in **conjunctive normal form** (CNF) i.e., a set of clauses. It is essentially a depth-first enumeration of possible models for the sentence.

However, DPLL goes beyond TT-ENTAILS in three ways: **early termination, pure symbol heuristic** and **unit clause heuristic.**

Improvements over TT-Entails

- **Early termination:** The algorithm detects whether the sentence is true or false, even with a partially completed model.
  - A clause is true if **any** literal is true; hence, the sentence as a whole could be judged true even before the model is complete.
  - 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.

Improvements over TT-Entails (cont'd)

- **Pure symbol heuristic:** A **pure symbol** is a symbol that always appears with the same "sign" (i.e., positive or negative) in all clauses. For example, in the three clauses

$$(A \vee \neg B), \quad (\neg B \vee \neg C), \quad (C \vee A)$$

the symbols A and B are pure and C is impure.

Note 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.

Improvements over TT-ENTAILS (cont'd)

- **Pure symbol heuristic (cont'd):**
  Note also that in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far.

  In the previous example, if the model contains $B = \mathit{false}$, then the clause $(\neg B \vee \neg C)$ is already *true*, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

  In the algorithm DPLL given below, the call to function FIND-PURE-SYMBOL returns a a symbol (or null) and the truth value to assign to that symbol in order to make the corresponding literals *true*.

Improvements over TT-Entails (cont'd)

- **Unit clause heuristic:** A **unit clause** is a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned to *false* by the model.

  For example, if the model contains $B = \mathit{true}$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to *false*. The unit clause heuristic assigns all such symbols before branching on the remainder.

Improvements over TT-ENTAILS (cont'd)

- **Unit clause heuristic (cont'd):**
  One important consequence of the heuristic is that any attempt prove (by refutation) a literal that is already in the knowledge base will succeed immediately.

  Notice also that assigning one unit clause can create another unit clause — for example, when C is set to *false*, $C \vee A$ becomes unit clause, causing *true* to be assigned to A.

  This cascade of forced assignments is called **unit propagation**.

  In the algorithm DPLL given below, the call to function FIND-UNIT-CLAUSE returns a a symbol (or null) and the truth value to assign to that symbol in order to make the corresponding unit clause *true*.

The algorithm DPLL

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

  $clauses \leftarrow$ the set of clauses in the CNF representation of s
  $symbols \leftarrow$ 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*
  $P, value \leftarrow$ FIND-PURE-SYMBOL($symbols, clauses, model$)
  if P is non-null **then return** DPLL($clauses, symbols - P, model \cup \{P{=}value\}$)
  $P, value \leftarrow$ FIND-UNIT-CLAUSE($clauses, model$)
  if P is non-null **then return** DPLL($clauses, symbols - P, model \cup \{P{=}value\}$)
  $P \leftarrow$ FIRST($symbols$); $rest \leftarrow$ REST($symbols$)
  return DPLL($clauses, rest, model \cup \{P{=}true\}$) **or**
        DPLL($clauses, rest, model \cup \{P{=}false\}$))

Improvements to DPLL

The following techniques can be used to speed up DPLL, and **SAT solvers** in general, significantly (some of them we have seen earlier e.g., for CSPs):

- **Component analysis:** As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient component detection algorithm, a solver can gain considerable speed by working on each component separately.
- **Variable and value ordering:** The above implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** suggests choosing the variable that appear most frequently in the remaining clauses.

Improvements to DPLL (cont'd)

- **Intelligent backtracking:** All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually, a limited-size conflict set is kept, and rarely used conflicts are dropped.
- **Random restarts:** Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value ordering) are made. Conflict clauses that are learned in the first run are retained after the restart and can help prune the search space.
  Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.

Improvements to DPLL (cont'd)

- **Clever indexing:** The speedup methods used in DPLL itself, as well as the tricks used in modern SAT solvers, require fast indexing of such things as "the set of clauses in which variable X_i appears as a positive literal".

  This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so **the indexing structures must be updated dynamically** as the computation proceeds.

With these enhancements, **modern SAT solvers can handle problems with tens of millions of variables.** They have revolutionized areas such as **hardware verification** and **security protocol verification,** which previously required laborious, hand-guided proofs.

Local search algorithms

Provided that we have a good evaluation function, we can apply local search algorithms such as HILL-CLIMBING, SIMULATED-ANNEALING and MIN-CONFLICTS to SAT.

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.** This is exactly what MIN-CONFLICTS does.

All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time.

The space usually contains **many local minima**, to escape from which various forms of randomness are required. In recent years, there have been many algorithms that try to **find a good balance between greediness and randomness.**

The algorithm WalkSAT

One of the simplest and most effective algorithms to emerge from all this work is called WalkSAT.

On every iteration, WalkSAT selects randomly an unsatisfied clause and picks a symbol in the clause to flip. It chooses with probability p between two ways to pick which symbol to flip:

- A "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state, and
- a "random walk" step that picks the symbol randomly.

The algorithm WalkSAT (cont'd)

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$(0, 1) \leq p$ **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 algorithm WalkSAT (cont'd)

When algorithm WalkSAT returns a model, the input sentence is indeed satisfiable, but when it returns *failure*, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time.

If we set $max\_flips = \infty$ and $p > 0$, WalkSAT will eventually return a model (if one exists), because the random walk steps will eventually hit a solution. **However, if the sentence is unsatisfiable, the algorithm will never terminate.**

For this reason, WalkSAT is most useful when we expect a solution to exist.

On the other hand, WalkSAT cannot always detect **unsatisfiability**, which is required for deciding entailment. Therefore, an agent cannot **reliably** use WalkSAT to prove that a square is safe in the wumpus world.

The landscape of random SAT problems

Some SAT problems are harder than others. Easy problems can be solved by any of the algorithms we presented earlier, but because we know that SAT is NP-complete, **at least some problem instances must require exponential time.**

In our lecture on CSPs, we saw some surprising facts for the n-queens problem: although it was tricky to solve for backtracking search algorithms, it turned out to be very easy for local search algorithms such as MIN-CONFLICTS.

This is because **solutions of the n-queens problem are densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby**. Thus, n-queens is easy, because it is **underconstrained**.

Underconstrained SAT problems

When we look at SAT problems in CNF, an **underconstrained problem** is one with relatively **few clauses constraining the variables**.

For example, here is a randomly generated 3-CNF sentence with five symbols and five clauses:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \neg E) \wedge$$

$$(E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$$

16 of the 32 possible assignments are models of this sentence, so, on average, it would take just 2 random guesses to find a model.

Definition: A sentence is in ***k*-CNF form** if it is a conjunction of disjunctions with at most k literals.

Overconstrained SAT problems

On the other hand, an **overconstrained** SAT problem has many clauses relative to the number of variables and is likely to have no solutions.

Overconstrained problems are often **easy to solve**, because the constraints quickly lead either to a solution or to a dead end.

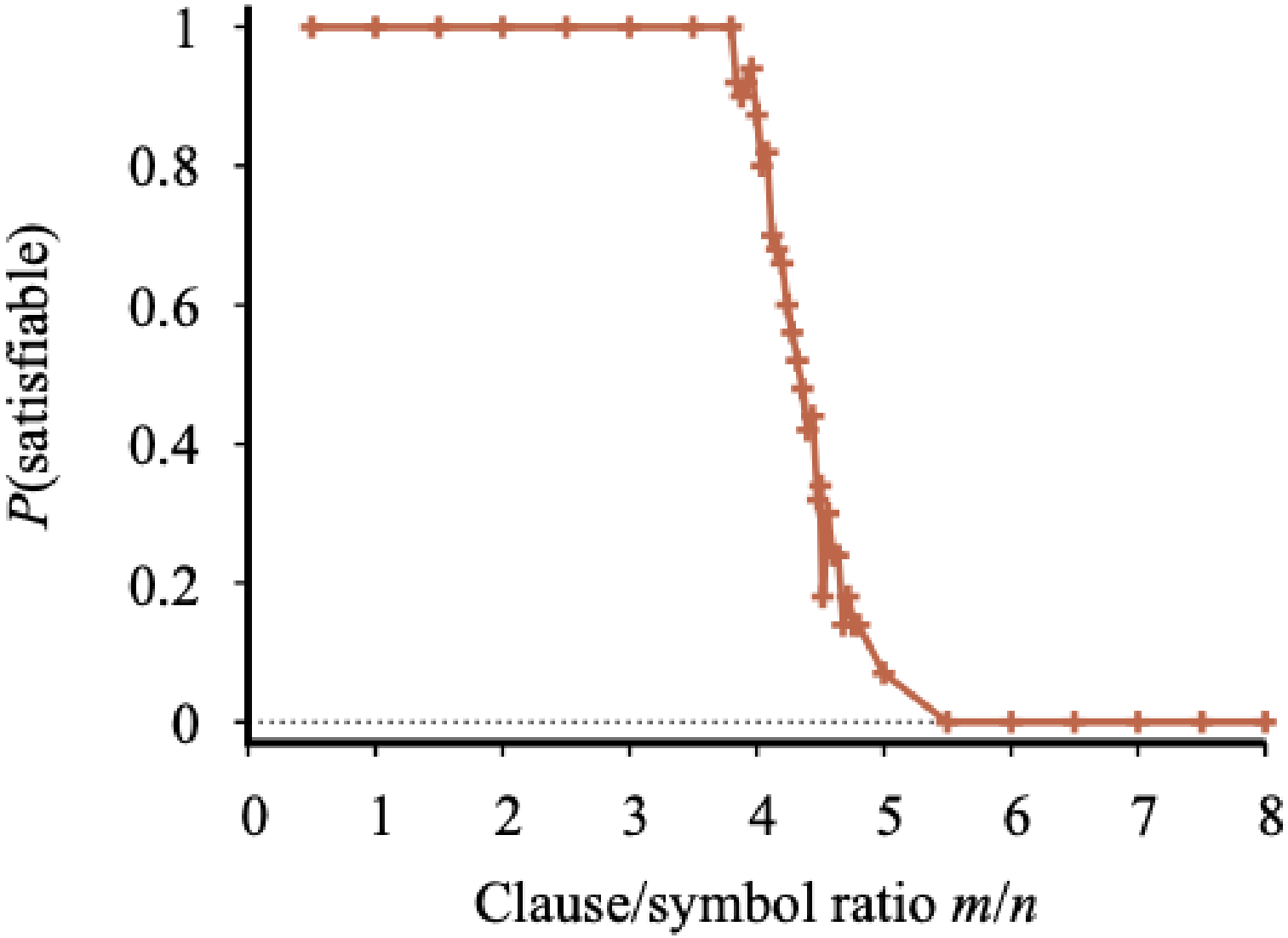
Random SAT problems

How can we generate **random SAT problems**?

Notation: $CNF_k(m, n)$ denotes a k-CNF sentence with m clauses and n symbols, where the clauses are chosen uniformly, independently, and without replacement from among all clauses with k different literals that are positive or negative at random.

Given a source of random sentences, we can measure the probability of satisfiability. The figure on the next slide shows the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the **clause/symbol ratio** m/n.

Random SAT problems (cont'd)



Random SAT problems (cont'd)

As we expect:

- For small m/n, the probability of satisfiability is close to 1.
- For large m/n, the probability of satisfiability is close to 0.

The probability drops fairly sharply around $m/n = 4.3$. Empirically, we can find that the "cliff" stays in roughly the same place (for $k = 3$) and gets sharper and sharper as n increases.

The satisfiability threshold conjecture

The **satisfiability threshold conjecture** says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(rn, n)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above.

The conjecture remains unproven, even for special cases like $k = 3$.

Whether it is a theorem or not, this kind of thresholding effect is certainly common, for satisfiability problems as well as other types of NP-hard problems.

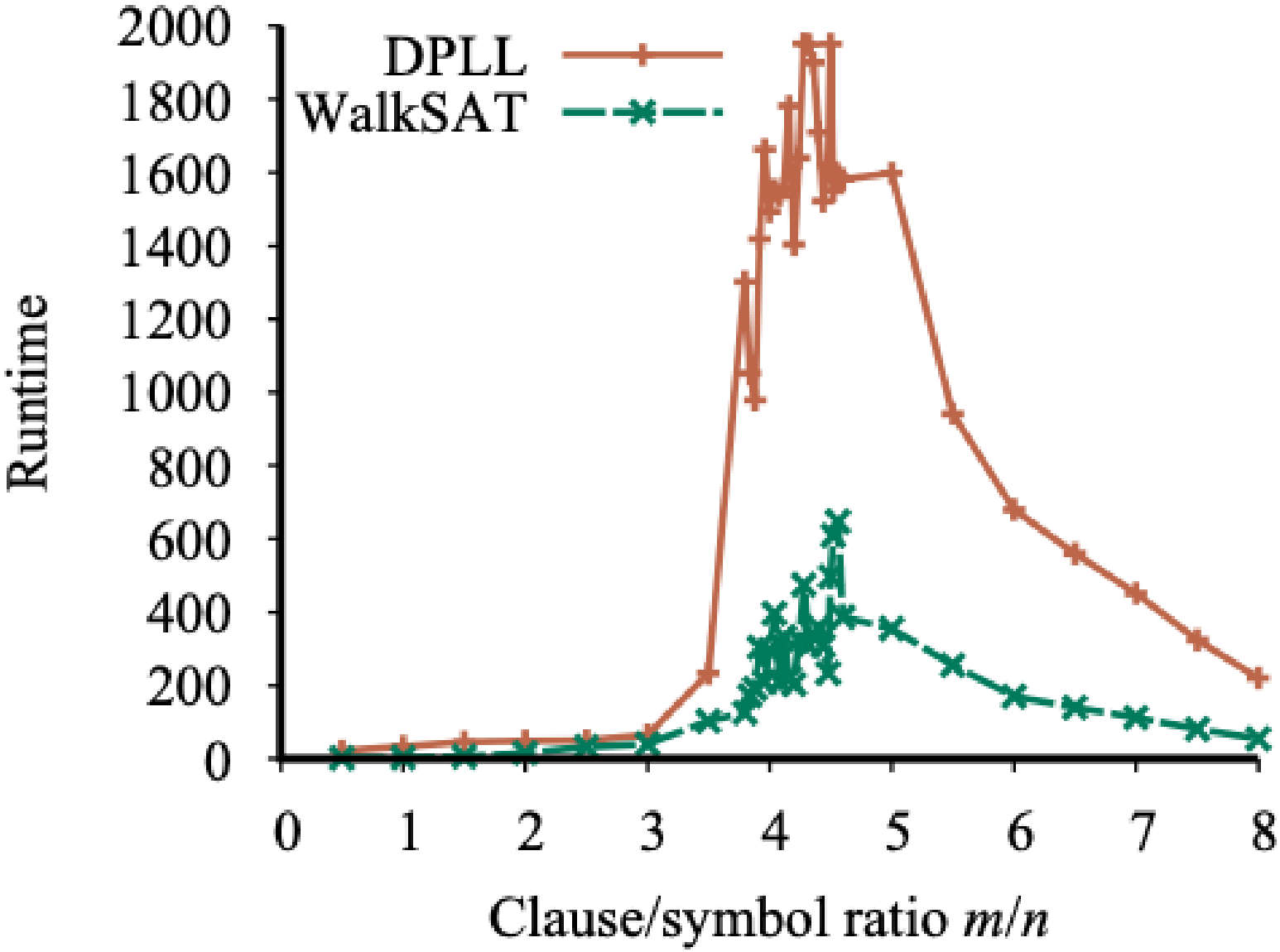
Where are the hard SAT problems?

The SAT problems that are hard to be solved are also often at the threshold value.

The figure on the next slide shows that 50-symbol problems at the threshold value of 4.3 are about 20 times more difficult to solve than those at ratio of 3.3. The graph shows median run time (measured in number of iterations) for both DPLL and WalkSAT on random 3-CNF sentences.

The underconstrained problems are easiest to solve because it is so easy to guess a solution. The overconstrained problems are not as easy as the underconstrained, but still are much easier than the ones right at the threshold.

Where are the hard SAT problems? (cont'd)



Agents based on propositional logic

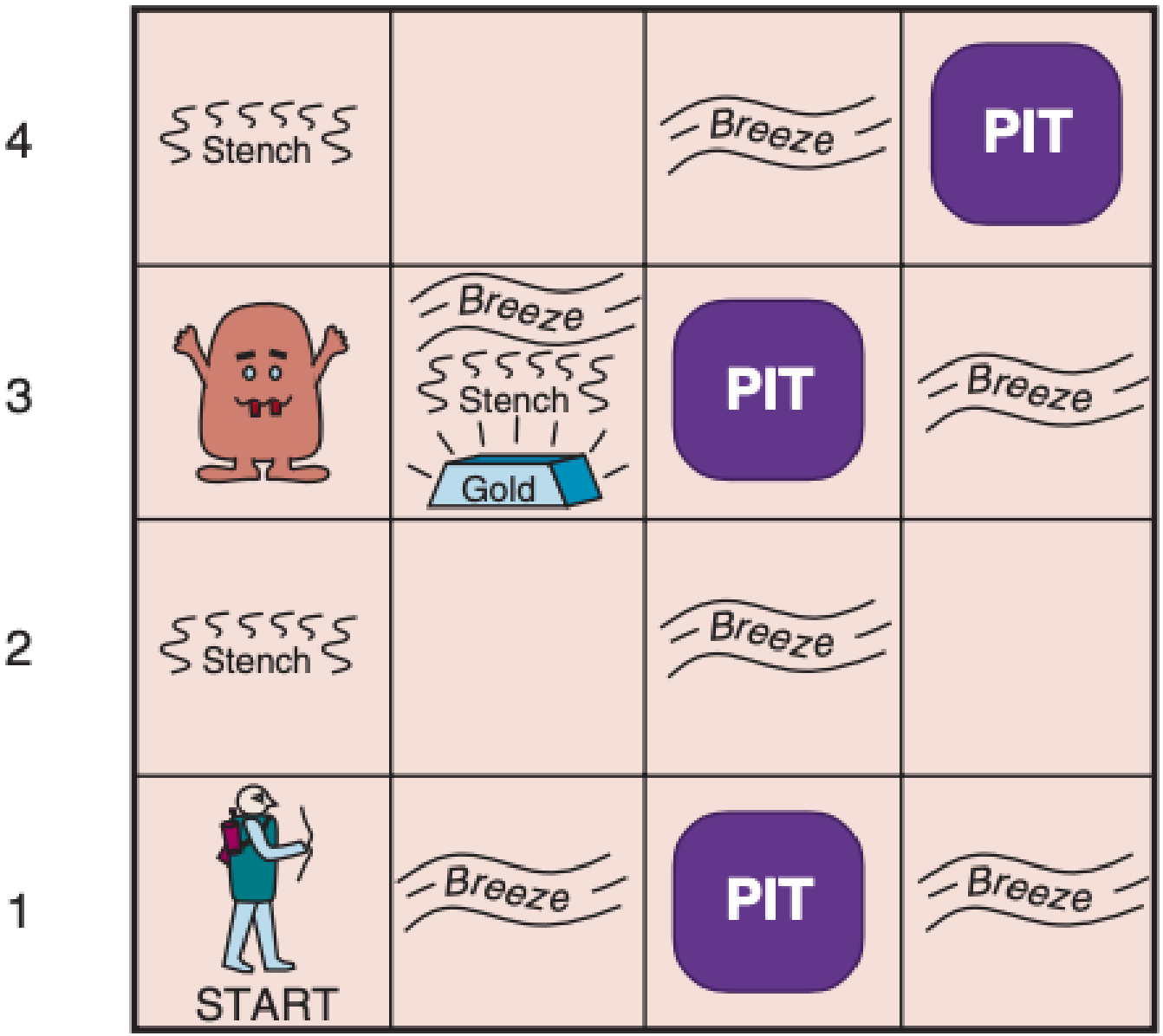
Logical agents operate by **deducing what to do from a knowledge base of sentences about the world.**

The knowledge base consists of:

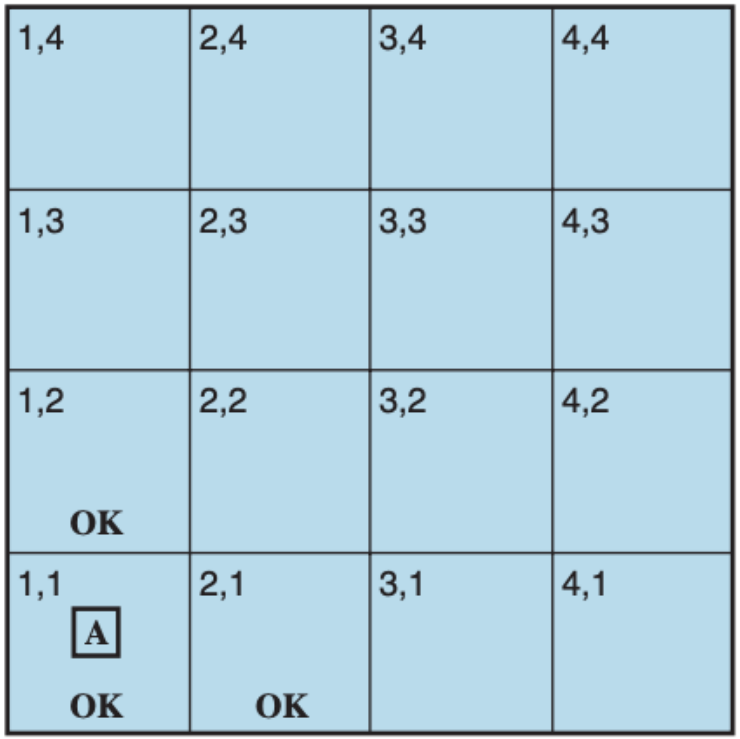
- **Axioms** i.e., general sentences about how the world works (the "physics" of the world).
- **Percept sentences** obtained from the agent's experience in particular world.

As an example, let us focus on the problem of **deducing the current state of the wumpus world**: where the agent is, is that square safe and so on.

The wumpus world

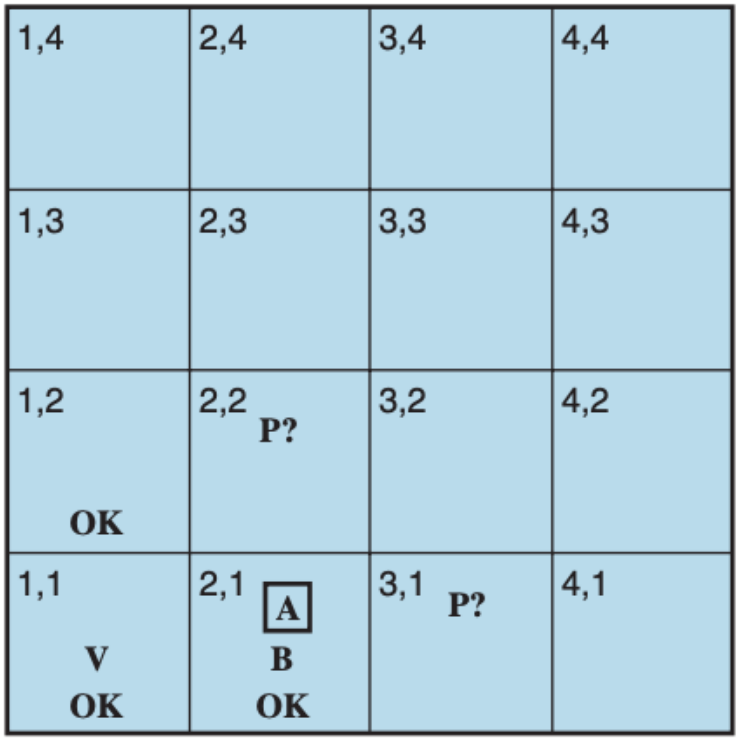


The wumpus world (cont'd)



(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*



(b)

Representing the current state of the wumpus world

The agent knows that the starting square contains no pit: $\neg P_{1,1}$

The agent knows that the starting square contains no wumpus: $\neg W_{1,1}$

For each square, the agent knows that the square is breezy if and and only if a neighboring square has a pit. Similarly, a square is smelly if and only if a neighboring square has a wumpus. Thus, we include a large collection of formulas of the following form:

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

$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$

$$\ldots$$

Representing the current state of the wumpus world (cont'd)

The agent also knows that there is exactly one wumpus.

This is expressed in two parts. First, we have to say there is **at least one wumpus**:

$$W_{1,1} \vee W_{1,2} \vee \cdots \vee W_{4,3} \vee W_{4,4}$$

Then, we have to say that there is **at most one wumpus**. How can express this?

Representing the current state of the wumpus world (cont'd)

For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\neg W_{1,1} \vee \neg W_{1,2}$$

$$\neg W_{1,1} \vee \neg W_{1,3}$$

$$\ldots$$

$$\neg W_{4,3} \vee \neg W_{4,4}$$

Agent percepts

We are using $S_{1,1}$ to mean that there is a stench in square $[1, 1]$.

Can we use a single propositional symbol *Stench* to mean that the agent perceives a stench?

Unfortunately, we can't: if there was no stench at a previous time step, then $\neg Stench$ would already be asserted, and the new assertion would **result in a contradiction!**

The problem is solved when we realize that **a percept asserts something only about the current time**. Thus, we can have propositional symbols

$$\ldots, \neg Stench^3, Stench^4, \ldots$$

and there is no contradiction.

The same goes for breeze, bump, glitter, and scream percepts.

Fluents

The idea of associating propositional symbols with time steps extends to any aspect of the world that changes with time.

For example, the initial knowledge base includes $L^0_{1,1}$ i.e., the agent is in square $[1, 1]$ at time 0. Similarly, $FacingEast^0$, $HaveArrow^0$, and $WumpusAlive^0$.

We use the noun **fluent** (from the Latin **fluens**, flowing) **to refer to an aspect of the world that changes.** Propositional symbols, like *Stench*, that take a time superscript will be called **fluents.**

Symbols associated with permanent aspects of the world do need a superscript and are sometimes called **atemporal variables**.

Fluents (cont'd)

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced as follows. For any time step t and any square $[x, y]$, we assert

$$L^t_{x,y} \Leftrightarrow (Breeze^t \Leftrightarrow B_{x,y})$$

$$L^t_{x,y} \Leftrightarrow (Stench^t \Leftrightarrow S_{x,y}).$$

Now, of course, we need axioms that allow the agent to keep track of fluents such as $L^t_{x,y}$.

Transition model

These fluents change as the result of actions taken by the agent, so we need to write down the **transition model** of the wumpus world as a set of logical sentences.

First, **we need proposition symbols for the occurrences of actions.**

As with percepts, these symbols are indexed by time; thus, $Forward^0$ means that the agent executes the action *Forward* at time 0.

By convention, **the percept of a give time step happens first, followed by the action for that time step, followed by a transition to the next time step.**

Effect axioms

To describe how the world changes, we will write **effect axioms** that specify the outcome of an action at the next time step.

For example, if the agent is at location $[1, 1]$ facing east at time 0 and goes *Forward*, the result is that the agent is in square $[2, 1]$ and no longer is in $[1, 1]$:

$$L^0_{1,1} \wedge \mathit{FacingEast}^0 \wedge \mathit{Forward}^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1})$$

We would need one such sentence for each possible timestep, for each of the 16 squares and each of the four orientations. We would also need similar sentences for the other actions:

$$\mathit{Grab}, \mathit{Shoot}, \mathit{Climb}, \mathit{TurnLeft}, \mathit{TurnRight}$$

Effect axioms (cont'd)

Let us suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base.

Given the above effect axiom, combined with the initial assertions about the state at time 0, the agent can now deduce that it is in $[2, 1]$.

In other words, $\text{ASK}(KB, L^1_{2,1}) = \mathit{true}$

The frame problem

Unfortunately, if we $\text{ASK}(KB, HaveArrow^1)$, the answer is *false*, that is **the agent cannot prove that it still has the arrow; nor can it prove it doesn't have it.**

The information has been lost because the effect axiom fails to state what remains **unchanged** as the result of an action. The need to do so gives rise to the **frame problem**.

Frame axioms

One possible solution to the frame problem would be to add **frame axioms** explicitly asserting all the propositions that remain the same. For example, for each time t, we would have

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$

$$Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$

$$\ldots$$

where we explicitly mention every proposition that stays unchanged from time t to time $t+1$ under the action *Forward*.

This proposal is inefficient. In a world with m different actions and n fluents, the set of frame axioms will be of size $O(mn)$ for every time step.

The frame problem (cont'd)

The above specific manifestation of the frame problem has been called the **representational frame problem**.

The representational frame problem is significant because the real world has very many fluents. Fortunately for us humans, each action typically changes no more than some small number k of those fluents i.e., **the world exhibits locality**.

Solving the representational frame problem requires defining the transition model with a set of axioms of size $O(mk)$ rather than of size $O(mn)$.

There is also an **inferential frame problem:** the problem of projecting forward the result of a t-step plan of action in time $O(kt)$ rather than $O(nt)$.

Solving the frame problem

The solution to the frame problem that we will present involves **changing our focus from writing axioms about actions to writing axioms about fluents.**

Thus, for each fluent F, we will have an axiom that defines the truth value of F^{t+1} in terms of fluents (including F itself) at time t and the actions that may have occured at time t.

Now, the truth value of F^{t+1} can be set in one of two ways:

- Either the action at time t causes F to be true at $t+1$, or
- F was already true at time t and the action at time t does not cause it to be false at time $t+1$.

Successor-state axioms

An axiom of this form is called a **successor-state axiom** and has the following form:

$$F^{t+1} \Leftrightarrow \mathit{ActionCausesF}^t \vee (F^t \wedge \neg \mathit{ActionCausesNotF}^t)$$

One of the simplest successor state axioms is the one for *HaveArrow*. Because there is no action for reloading, the $\mathit{ActionCausesF}^t$ part goes away and we are left with

$$\mathit{HaveArrow}^{t+1} \Leftrightarrow (\mathit{HaveArrow}^t \wedge \neg \mathit{Shoot}^t).$$

Successor-state axioms (cont'd)

For the agent's location, the successor-state axioms are more elaborate.

For example, $L_{1,1}^{t+1}$ is true if either:

- The agent moved *Forward* from $[1, 2]$ when facing south, or from $[2, 1]$ when facing west; or
- $L_{1,1}^{t}$ was already true and the action did not cause movement (either because the action was not *Forward* or because the action bumped into a wall).

Written out in propositional logic, this becomes:

$$L_{1,1}^{t+1} \Leftrightarrow (L_{1,1}^{t} \wedge (\neg Forward^{t} \vee Bump^{t+1}))$$

$$\vee (L_{1,2}^{t} \wedge (\neg FacingSouth^{t} \vee Forward^{t}))$$

$$\vee (L_{2,1}^{t} \wedge (\neg FacingWest^{t} \vee Forward^{t}))$$

Using Ask

Given a complete set of successor-state axioms and axioms specifying the initial world state and the "physics" of the world, the agent will be able to Ask and answer any answerable question about the current state of the world.

Using Ask (cont'd)

For example, let us assume the following sequence of percepts and actions in the wumpus world:

$$\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 \; ; \; Forward^0$$

$$\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 \; ; \; TurnRight^1$$

$$\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 \; ; \; TurnRight^2$$

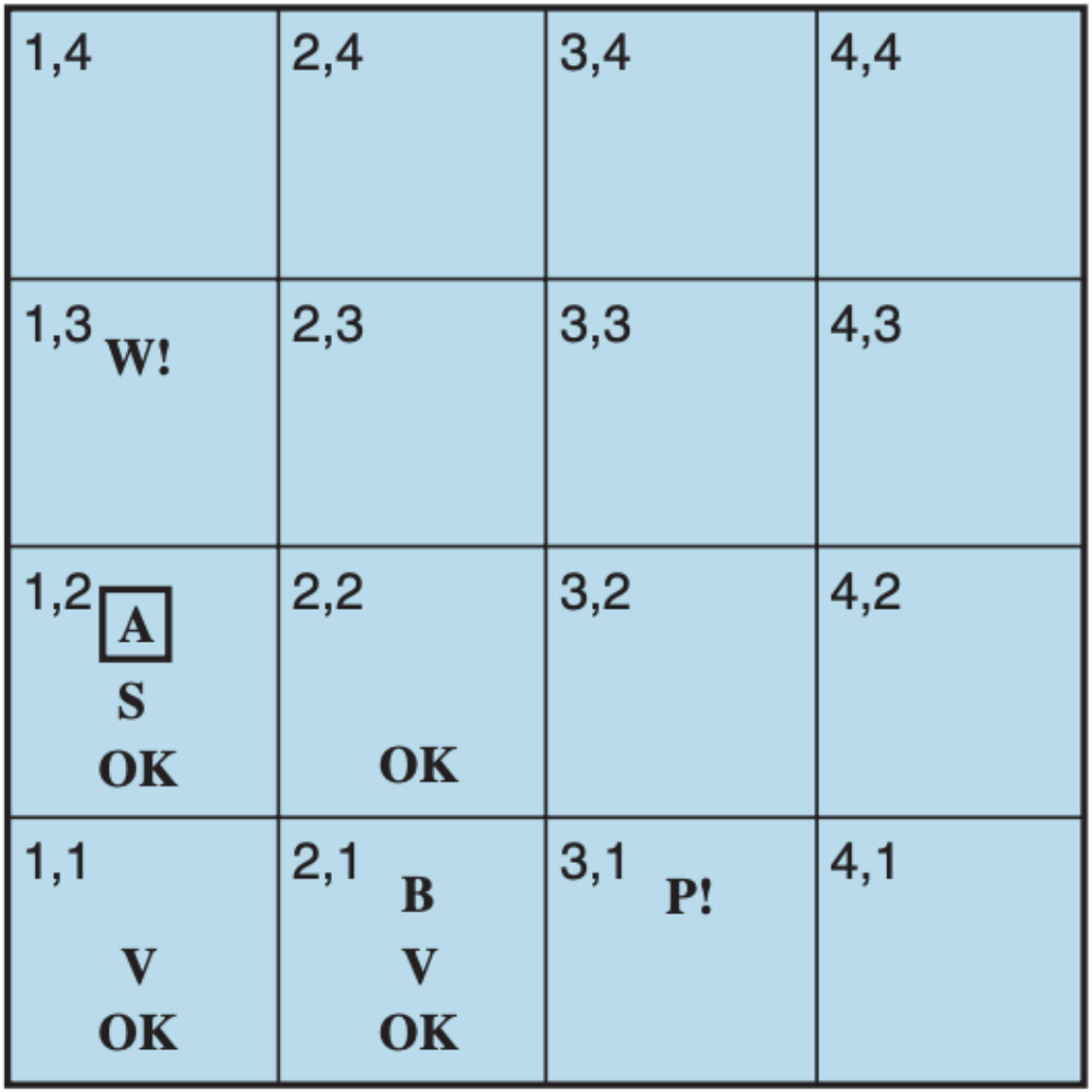
$$\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 \; ; \; Forward^3$$

$$\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 \; ; \; TurnRight^4$$

$$\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 \; ; \; Forward^5$$

$$Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6$$

The state of the wumpus world



A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

Using Ask (cont'd)

At this point, we have $\text{Ask}(KB, L^6_{1,2}) = \mathit{true}$, so the agent knows where it is.

Moreover, $\text{Ask}(KB, W_{1,3}) = \mathit{true}$ and $\text{Ask}(KB, P_{3,1}) = \mathit{true}$, so the agent has found the wumpus and one of the pits.

The most important question for the agent is whether a square is OK to move into — that is, whether the square is free of a pit or live wumpus.

It is convenient to add axioms for this, having the form

$$OK^t_{x,y} \Leftrightarrow \neg P_{x,y} \wedge \neg(W_{x,y} \wedge \mathit{WumpusAlive}^t).$$

Finally, $\text{Ask}(KB, OK^6_{2,2}) = \mathit{true}$, so the square $[2, 2]$ is OK to move into.

Using Ask (cont'd)

Given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK, and can do so in just a few milliseconds for small-to-medium wumpus world.

The qualification problem

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: **we need to confirm that all the necessary preconditions of an action hold, for it to have its intended effect.**

We said that the *Forward* action moves ahead unless there is a wall in the way, but there are many other unusual exceptions that could cause the action to fail: the agent might trip and fall, be stricken with a heart attack, be carried away by giant bats, etc.

Specifying all these exceptions is called the **qualification problem**.

There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out. **Probability theory** allows us to summarize all the exceptions without explicitly naming them.

Planning

Planning is an important subarea of Artificial Intelligence with many interesting research results and implemented systems.

Classical planning is the task of finding a sequence of actions (i.e., a **plan**) to accomplish a goal in discrete, deterministic, static, fully observable environment.

A planning problem can be cast as a **search problem**. Then, it can be solved using any of the **search algorithms** we presented in the **problem solving** part of the course.

In this lecture, we will show how to solve planning problems using propositional inference.

Planning by propositional inference

The basic idea for **solving planning problems using propositional inference** is very simple:

- Construct a propositional logic sentence ϕ that is the conjunction of:
  - $Init^0$, a collection of assertions about the initial state;
  - $Transition^1, \ldots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time t;
  - the assertion that the goal is achieved at time t, for example, $HaveGold^t \wedge ClimbedOut^t$.
- Present the sentence ϕ to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the problem is unsolvable.
- Assuming a model is found, extract from the model those variables that represent actions and are assigned *true*. Together they represent a plan to achieve the goal.

The algorithm SATPLAN

The following algorithm is a propositional planning procedure which implements the basic idea just given.

function SATPLAN(*init*, *transition*, *goal*, $T_{\max}$) **returns** solution or *failure*
  inputs: *init*, *transition*, *goal*, constitute a description of the problem
    $T_{\max}$, an upper limit for plan length

  for $t = 0$ **to** $T_{\max}$ **do**
    cnf $\leftarrow$ TRANSLATE-TO-SAT(*init*, *transition*, *goal*, *t*)
    model $\leftarrow$ SAT-SOLVER(*cnf*)
    if *model* is not null **then**
      return EXTRACT-SOLUTION(*model*)
  return *failure*

The algorithm SATPLAN (cont'd)

Because the agent does not know how many steps it will take to reach the goal, SATPLAN tries each possible number of steps t, up to some maximum conceivable plan length T_{max}.

In this way, **it is guaranteed to find the shortest plan if one exists.**

Because of the way SATPLAN searches for a solution, **this approach cannot be used in a partially observable environment**; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

Constructing the knowledge base for SATPLAN

The key step in using SATPLAN is the **construction of the knowledge base (the sentence ϕ that is transformed into CNF).**

It is important to revisit this issue since there is a significant difference between the way we constructed the knowledge base when we wanted to decide entailment and the way it needs to be constructed now that we have to decide satisfiability.

Consider, for example, the agent's location, initially $[1, 1]$ and suppose the agent's unambitious goal is to be in $[2, 1]$ at time 1.

The initial knowledge base contains $L^0_{1,1}$ and the goal is $L^1_{2,1}$. Using ASK, we can prove $L^1_{2,1}$ if $Forward^0$ is asserted, and, reassuringly, we cannot prove $L^1_{2,1}$ if $Shoot^0$.

Finding plans with SATPLAN

SATPLAN will also find this plan [$Forward^0$].

But unfortunately, SATPLAN also finds the plan [$Shoot^0$]. How could this be? To find out, we inspect the model that SATPLAN constructs: it includes the assignment $L^0_{2,1}$ i.e., the agent can be in $[2, 1]$ at time 1 by being there at time 0 and shooting.

The problem here is that **we did not assert in the knowledge base that the agent cannot be in two squares at the same time.**

For entailment, $L^0_{2,1}$ is unknown and cannot be used in a proof. For satisfiability, on the other hand, $L^0_{2,1}$ is unknown and can, therefore, be set to whatever value helps to make the goal *true*.

How can we fix the knowledge base so this does not happen?

Finding plans with SATPLAN (cont'd)

We can fix the knowledge base by asserting that, at each time step, the agent is in exactly one location, **using a collection of sentences similar to those used to assert the existence of exactly one wumpus.**

Alternatively, we can assert $\neg L^0_{x,y}$ for all locations other than $[1,1]$; the successor-state axiom for locations takes care of subsequent time steps.

The same fixes work to make sure that the agent has exactly one orientation at each time step.

Finding plans with SATPLAN (cont'd)

SATPLAN has more surprises in store, however.

The first one is that it finds models with **impossible actions**, such as shooting with no arrow.

To understand why, we need to look more carefully at what successor-state axioms say about **actions whose preconditions are not satisfied.**

The axioms do predict correctly that nothing will happen when such an action is executed, but they do not say that the action cannot be executed.

Precondition axioms

To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.

For example, we need to say, for each time t, that

$$Shoot^t \Rightarrow HaveArrow^t.$$

This ensures that if a plan selects the *Shoot* action at any time, it must be the case that the agent has an arrow at that time.

Action exclusion axioms

SATPLAN's second surprise is the creation of **plans with multiple simultaneous actions.**

For example, it may come up with a model in which both $Forward^0$ and $Shoot^0$ are true, which is not allowed.

To eliminate the problem, we introduce **action exclusion axioms**: for every pair of actions A_i^t and A_j^t, we add the axiom

$$\neg A_i^t \vee \neg A_j^t.$$

Action exclusion axioms (cont'd)

It might be beneficial to selectively introduce action exclusion axioms e.g., allow $Forward^t$ and $Shoot^t$ but disallow $Grab^t$ and $Shoot^t$.

In this case, we introduce action exclusion axioms only for pairs of actions that **interfere with each other**.

Because SATPLAN finds the shortest legal plan, we can be sure it will take advantage of this fact and produce plans with multiple simultaneous actions that do not interfere with each other.

Summary

SATPLAN finds models for a sentence containing the initial state, the goal, the successor state axioms, the precondition axioms, and the action exclusion axioms.

It can be shown that this collection of axioms is sufficient, in the sense there are no longer any "spurious" solutions.

Any model satisfying the propositional sentence can be used to extract a **valid plan** for the original problem.

Modern SAT solving technology makes this approach quite practical.

Conclusions

We presented a **declarative approach to logical agent construction**: the agent works by a combination of asserting propositional logic sentences in the knowledge base and performing logical inference.

This approach has some weaknesses hidden in phrases such as "for each time t" and "for each square $[x, y]$". For any practical agent, these phrases have to be implemented by code that generates instances of the general sentence schema automatically for insertion in the knowledge base.

For a wumpus world of reasonable size – one comparable to a small computer game – we might need a 100×100 board and 1000 time steps, leading to knowledge bases with **tens or hundreds of millions of sentences.**

Conclusions (cont'd)

Not only does this becomes impractical, but it also illustrates a deeper problem: we know something about the wumpus world — namely that the "physics" works the same way across squares and all time steps — that **we cannot express directly in the language of propositional logic.**

To solve this problem, we need a more expressive language, one in which phrases like "for each time t" and "for each square $[x, y]$" can be written in a natural way.

First-order logic is such a language. In first-order logic, a wumpus world of any size and duration can be described in about ten logic sentences rather than ten million or ten trillion.

Readings

These slides are based on Sections 7.4.4, 7.6 and 7.7 of Chapter 7 of the AIMA book.

Note: I took the liberty to use material from the above sections verbatim without using quotes.