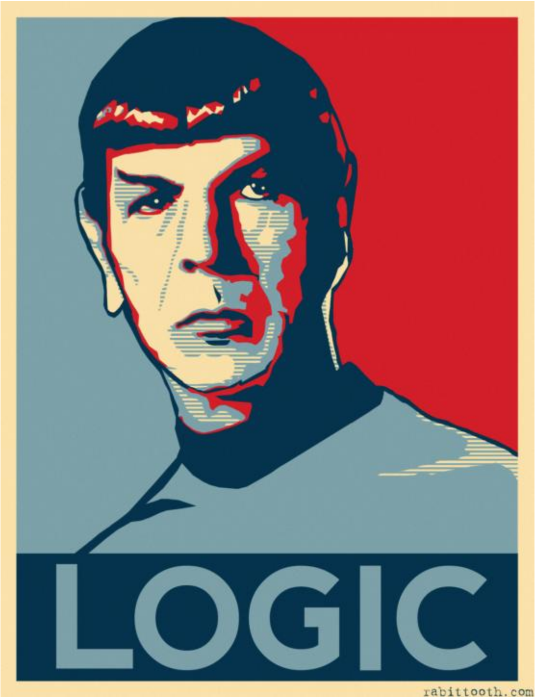
AI: Representation and Problem Solving

Logical Agent Algorithms



Instructor: Pat Virtue

Slide credits: CMU AI, http://ai.berkeley.edu

Plan

Last Time:

- Propositional logic
- Models and Knowledge Bases
- Satisifiability and Entailment

Today: Logical Agent Algorithms

- Entailment
  - Model checking: Truth table entailment
  - Theorem proving: (Forward chaining), resolution
- Satisfiability: DPLL
- Planning with logic

Plan

Last Time:

- Propositional logic
- Models and Knowledge Bases
- Satisifiability and Entailment

Propositional Logic Vocab

Literal

- Atomic sentence: True, False, Symbol, ¬Symbol

Clause

- Disjunction (OR) of literals: $A \vee B \vee \neg C$

Definite clause

- Disjunction (OR) of literals, *exactly one* is positive
- $\neg A \vee B \vee \neg C$

Horn clause

- Disjunction of literals, *at most one* is positive
- All definite clauses are Horn clauses

PL: Conjunctive Normal Form (CNF)

Every sentence can be expressed as a conjunction (AND) of clauses

Each clause is a disjunction (OR) of literals

Each literal is a symbol or a negated symbol

- Example: $(\neg A \lor \neg C \lor B) \land (\neg A \lor \neg B \lor C)$

PL: Conjunctive Normal Form (CNF)

Every sentence can be expressed as a conjunction of clauses

Each clause is a disjunction of literals

Each literal is a symbol or a negated symbol

Conversion to CNF by a sequence of standard transformations:

- At_1,1_0 ⇒ (Wall_0,1 ⇔ Blocked_W_0)
- At_1,1_0 ⇒ ((Wall_0,1 ⇒ Blocked_W_0) ∧ (Blocked_W_0 ⇒Wall_0,1))
- ¬At_1,1_0 v ((¬Wall_0,1 v Blocked_W_0) ∧ (¬Blocked_W_0 v Wall_0,1))
- (¬At_1,1_0 v ¬Wall_0,1 v Blocked_W_0) ∧ (¬At_1,1_0 v ¬Blocked_W_0 v Wall_0,1)

PL: Conjunctive Normal Form (CNF)

Every sentence can be expressed …

Each clause is a disjunction of literal…

Each literal is a symbol or a neg… symb…

Conversion to CNF by a se… quence … tandard transfor… s:

- $\text{At\_1,1\_0} \Rightarrow (\text{Wall\_0,1} \Leftrightarrow \text{Blocked\_W\_0})$
- $\text{At\_1,1\_0} \Rightarrow ((\text{Wall\_0,1} \Rightarrow \text{Blocked\_W\_0}) \wedge (\text{Blocked\_W\_0} \Rightarrow \text{Wall\_0,1}))$
- $\neg\text{At\_1,1\_0} \vee ((\neg\text{Wall\_0,1} \vee \text{Blocked\_W\_0}) \wedge (\neg\text{Blocked\_W\_0} \vee \text{Wall\_0,1}))$
- $(\neg\text{At\_1,1\_0} \vee \neg\text{Wall\_0,1} \vee \text{Blocked\_W\_0}) \wedge (\neg\text{At\_1,1\_0} \vee \neg\text{Blocked\_W\_0} \vee \text{Wall\_0,1})$

Replace biconditional by two implications

Replace $\alpha \Rightarrow \beta$ by $\neg\alpha \vee \beta$

Distribute v over $\wedge$

Logical Agent Vocab

Model

- Complete assignment of symbols to True/False

Sentence

- Logical statement
- Composition of logic symbols and operators

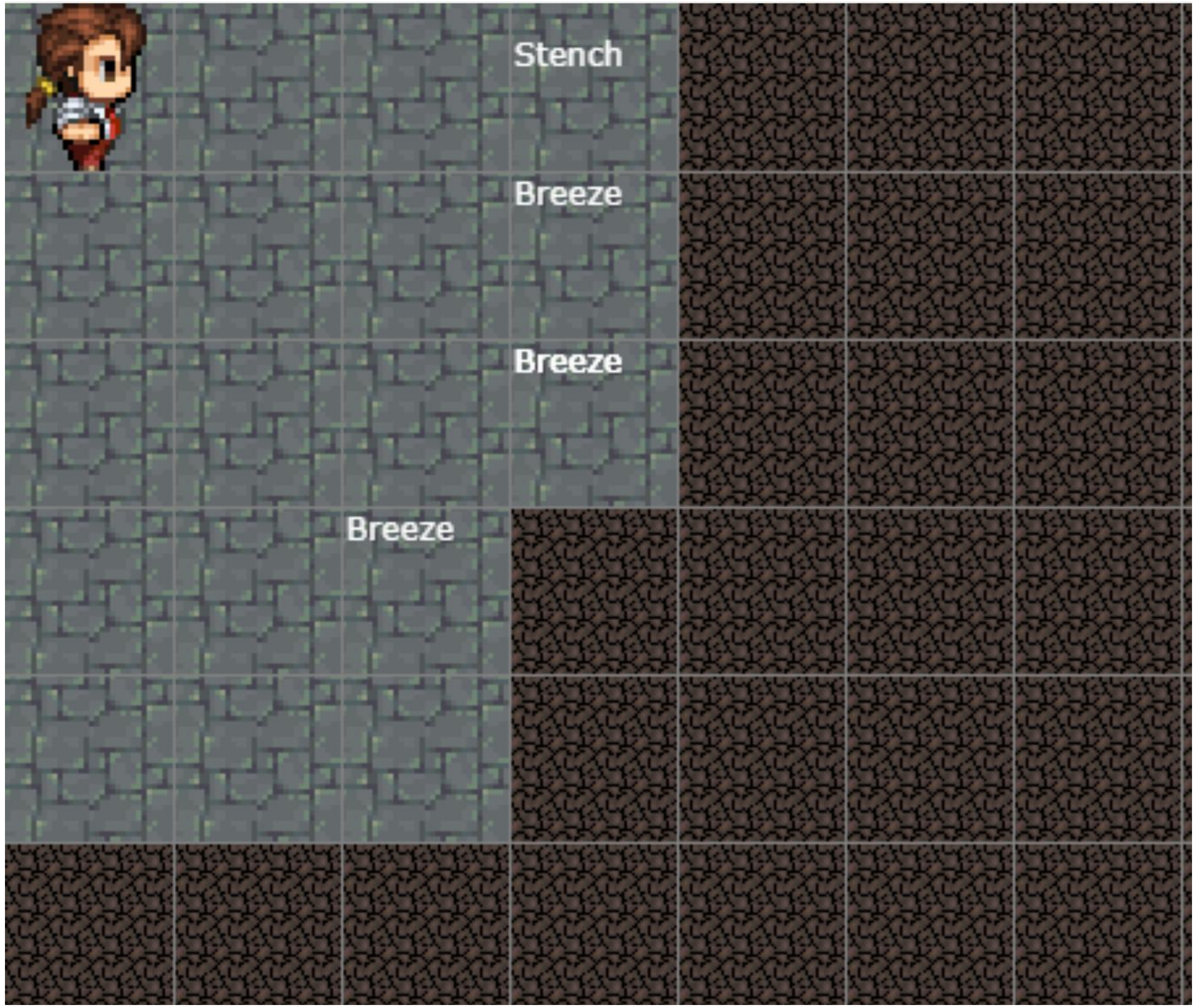
KB

- Collection of sentences representing facts and rules we know about the world

Query

- Sentence we want to know if it is *provably* True, *provably* False, or *unsure*.

Provably True, Provably False, or Unsure



http://thiagodnf.github.io/wumpus-world-simulator/

Logical Agent Vocab

Entailment

- Input: sentence1, sentence2
- Each model that satisfies sentence1 must also satisfy sentence2
- "If I know 1 holds, then I know 2 holds"
- (ASK), TT-ENTAILS, FC-ENTAILS, RESOLUTION-ENTAILS

Satisfy

- Input: model, sentence
- Is this sentence true in this model?
- Does this model satisfy this sentence
- "Does this particular state of the world work?’
- PL-TRUE

Logical Agent Vocab

Satisfiable

- Input: sentence
- Can find at least one model that satisfies this sentence
  - (We often want to know what that model is)
- "Is it possible to make this sentence true?"
- DPLL

Valid

- Input: sentence
- sentence is true in all possible models

Outline

Logical Agent Algorithms

- Vocab
- PL_TRUE
- Entailment
  - Model checking: Truth table entailment
  - Theorem proving:
  - (Forward chaining), resolution
- Satisfiability: DPLL
- Planning with logic

Propositional Logic

Check if sentence is true in given model

In other words, does the model *satisfy* the sentence?

function PL-TRUE?(α,model) returns true or false

But are models and propositional logic sentences α represented?

Propositional Logic

Check if sentence is true in given model

In other words, does the model *satisfy* the sentence?

function PL-TRUE?(α,model) returns true or false

if α is a symbol then return Lookup(α, model)

if Op(α) = $\neg$ then return **not**(PL-TRUE?(Arg1(α),model))

if Op(α) = $\wedge$ then return **and**(PL-TRUE?(Arg1(α),model),
PL-TRUE?(Arg2(α),model))

etc.

(Sometimes called “recursion over syntax”)

Outline

Logical Agent Algorithms

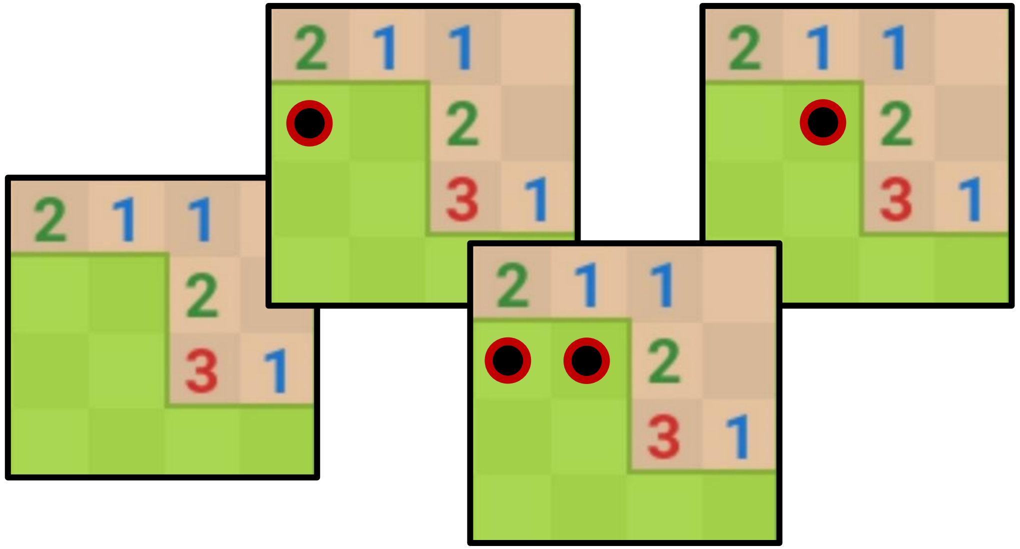
- Vocab
- PL_TRUE
- Entailment
  - Model checking: Truth table entailment
  - Theorem proving:
  - (Forward chaining), resolution
- Satisfiability: DPLL
- Planning with logic

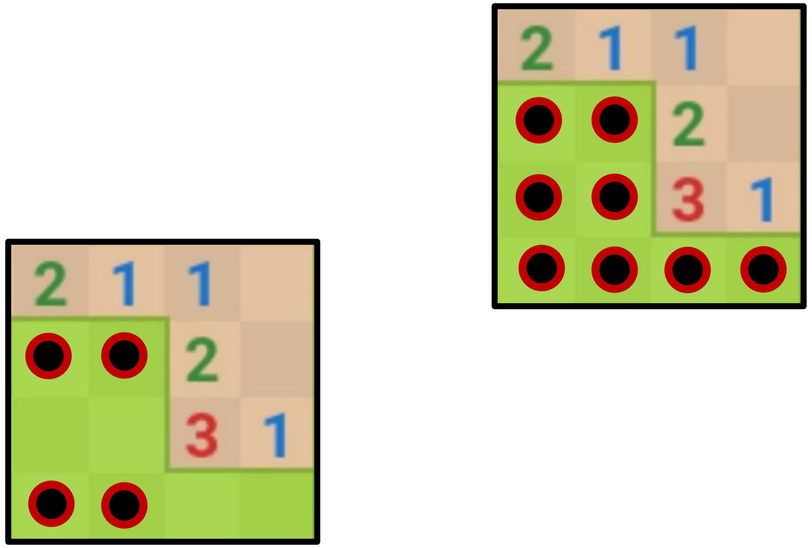
Inference: Proofs

A proof is a *demonstration* of entailment between α and β

Method 1: *model-checking*

- For every possible world, if α is true make sure that is β true too
- OK for propositional logic (finitely many worlds); not easy for first-order logic





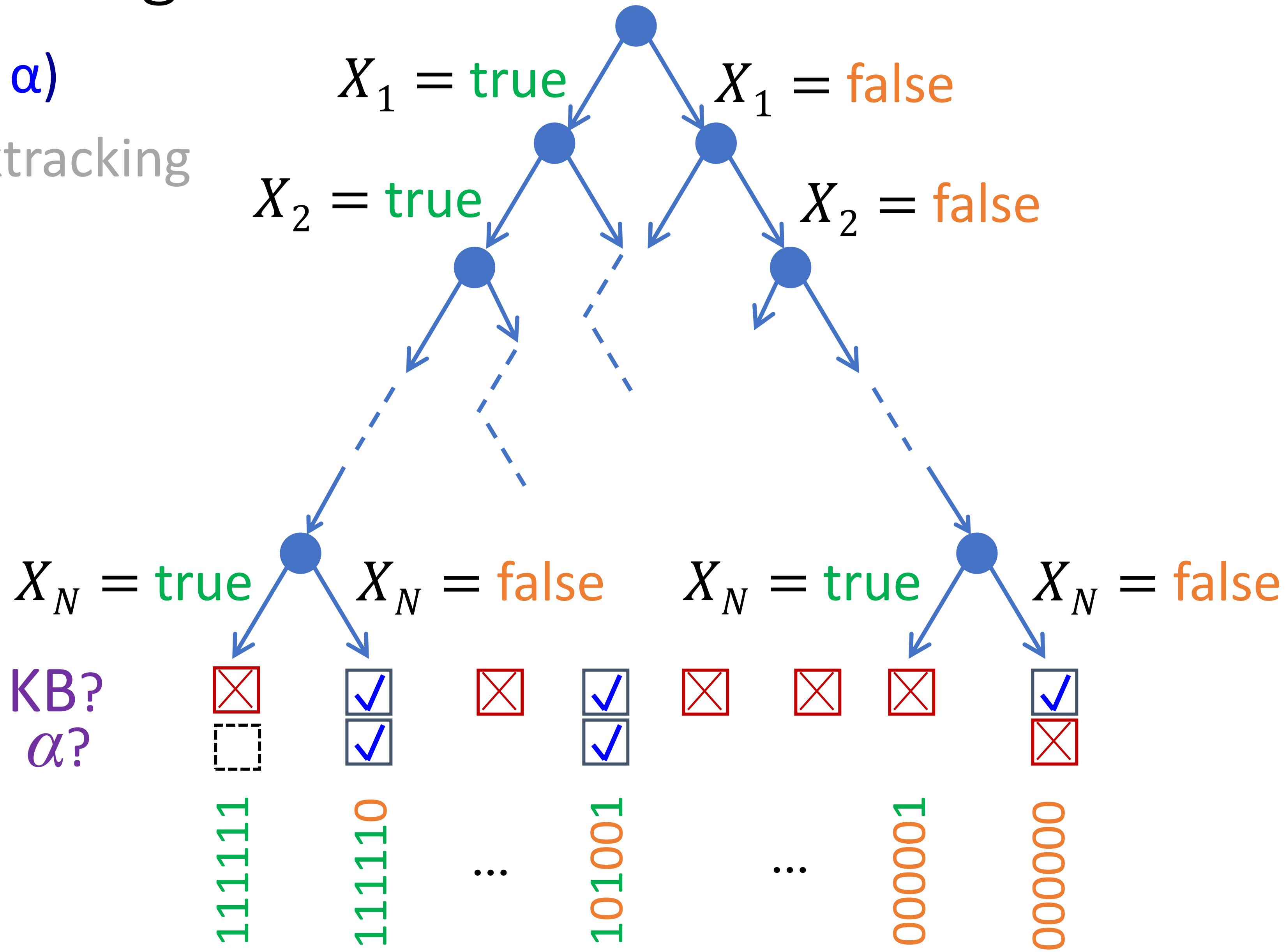
Simple Model Checking

function TT-ENTAILS?(KB, α)   Returns true or false

Simple Model Checking

function TT-ENTAILS?(KB, α)

Same recursion as backtracking



Simple Model Checking

```
function TT-ENTAILS?(KB, α)    Returns true or false
    return TT-CHECK-ALL(KB, α, symbols(KB) ∪ symbols(α), {})

function TT-CHECK-ALL(KB, α, symbols, model)    Returns true or false
    Recursively check to make sure all models
    that satisfy the KB also satisfy α
```

Simple Model Checking

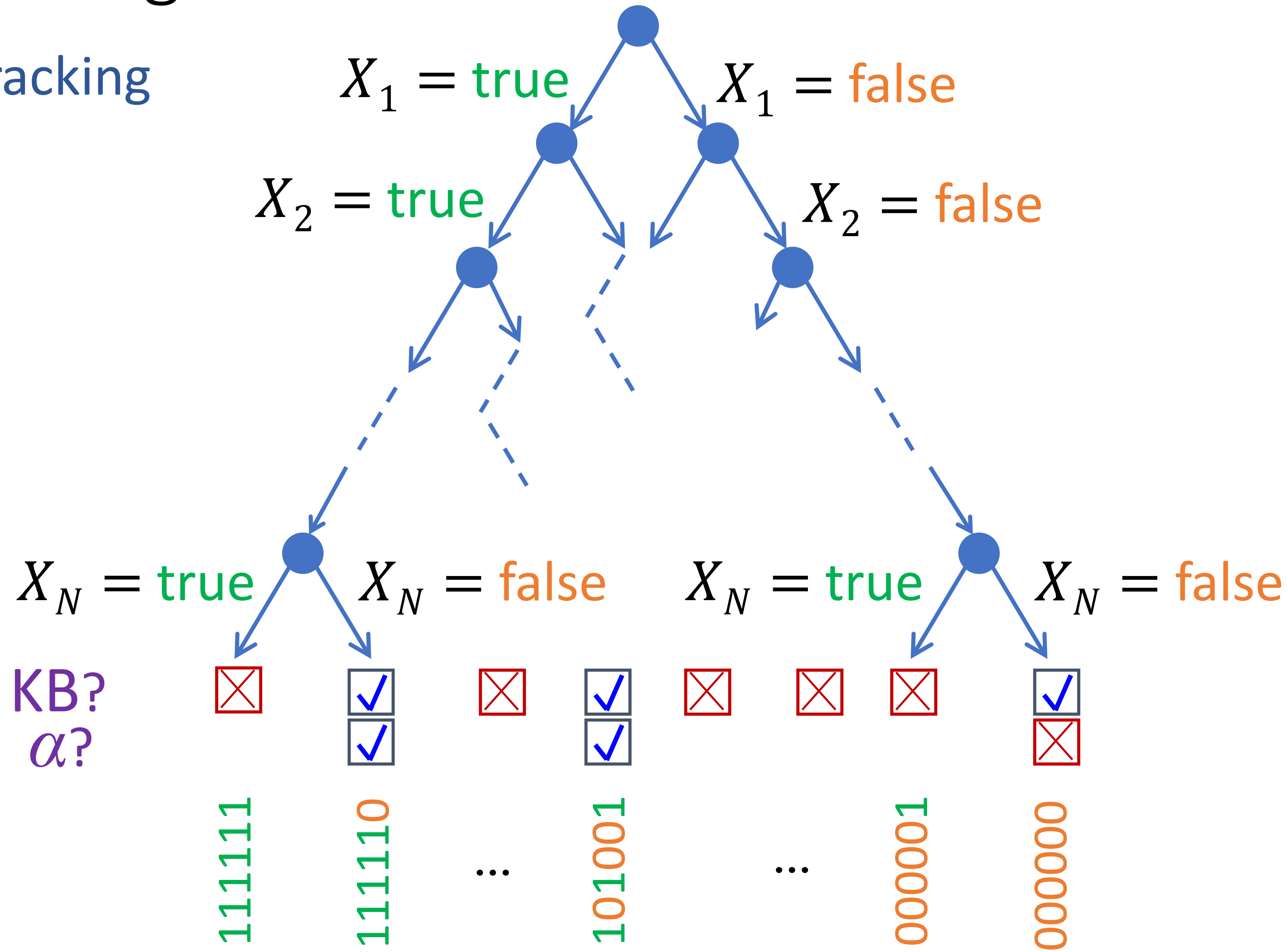
```
function TT-CHECK-ALL(KB, α, symbols, model)     Returns true or false
    if empty?(symbols) then
        if PL-TRUE?(KB, model) then
            return PL-TRUE?(α, model)
        else
            return true
    else
        X_i ← first(symbols)
        rest ← rest(symbols)
        return  and ( TT-CHECK-ALL(KB, α, rest, model ∪ {X_i = true})
                      TT-CHECK-ALL(KB, α, rest, model ∪ {X_i = false}) )
```

Simple Model Checking

Same recursion as backtracking

$O(2^N)$ time, linear space

Can we do better?



Inference: Proofs

A proof is a *demonstration* of entailment between α and β

Method 1: *model-checking*

- For every possible world, if α is true make sure that is β true too
- OK for propositional logic (finitely many worlds); not easy for first-order logic

Method 2: *theorem-proving*

- Search for a sequence of proof steps (applications of *inference rules*) leading from α to β
- E.g., from $P \wedge (P \Rightarrow Q)$, infer Q by *Modus Ponens*

Properties

- *Sound* algorithm: everything it claims to prove is in fact entailed
- *Complete* algorithm: every sentence that is entailed can be proved

Simple Theorem Proving: Forward Chaining

Forward chaining applies Modus Ponens to generate new facts:

- Given $X_1 \wedge X_2 \wedge \ldots X_n \Rightarrow Y$ and X_1, X_2, …, X_n
- Infer Y

Forward chaining keeps applying this rule, adding new facts, until nothing more can be added

Forward Chaining Algorithm

function PL-FC-ENTAILS?(KB, q)   Returns true or false

***KB* CLAUSES**

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

Properties

Forward Chaining is:

- Sound and complete for definite-clause KBs
- Complexity: linear time ☺

Resolution is another theorem-proving algorithm that is:

- Sound and complete for any PL KBs!
- Complexity: exponential time ☹

Vocab Reminder

Literal

- Atomic sentence:
  T, F, Symbol, ¬Symbol

Clause

- Disjunction of literals:
  $A \lor B \lor \neg C$

Definite clause

- Disjunction of literals, *exactly one* is positive
  $\neg A \lor B \lor \neg C$

Inference Rules

Notation Alert!

Modus Ponens

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

Unit Resolution

$$\frac{a \vee b, \quad \neg b \vee c}{a \vee c}$$

General Resolution

$$\frac{a_1 \vee \cdots \vee a_m \vee b, \quad \neg b \vee c_1 \vee \cdots \vee c_n}{a_1 \vee \cdots \vee a_m \vee c_1 \vee \cdots \vee c_n}$$

Resolution

Algorithm Overview

function PL-RESOLUTION?(KB, α) returns true or false

We want to prove that KB entails α

In other words, we want to prove that we cannot satisfy (KB and **not** α)

1. Start with a set of CNF clauses, including the KB as well as $\neg\alpha$
2. Keep resolving pairs of clauses until
   A. You resolve the empty clause
      Contradiction found!
      KB $\wedge \neg\alpha$ cannot be satisfied
      Return true, KB entails α
   B. No new clauses added
      Return false, KB does not entail α

Resolution

Example trying to prove $\neg P_{1,2}$

General Resolution

$$\frac{a_1 \vee \cdots \vee a_m \vee b, \quad \neg b \vee c_1 \vee \cdots \vee c_n}{a_1 \vee \cdots \vee a_m \vee c_1 \vee \cdots \vee c_n}$$

Knowledge Base

$\neg P_{2,1} \vee B_{1,1}$ | $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$ | $\neg P_{1,2} \vee B_{1,1}$ | $\neg B_{1,1}$

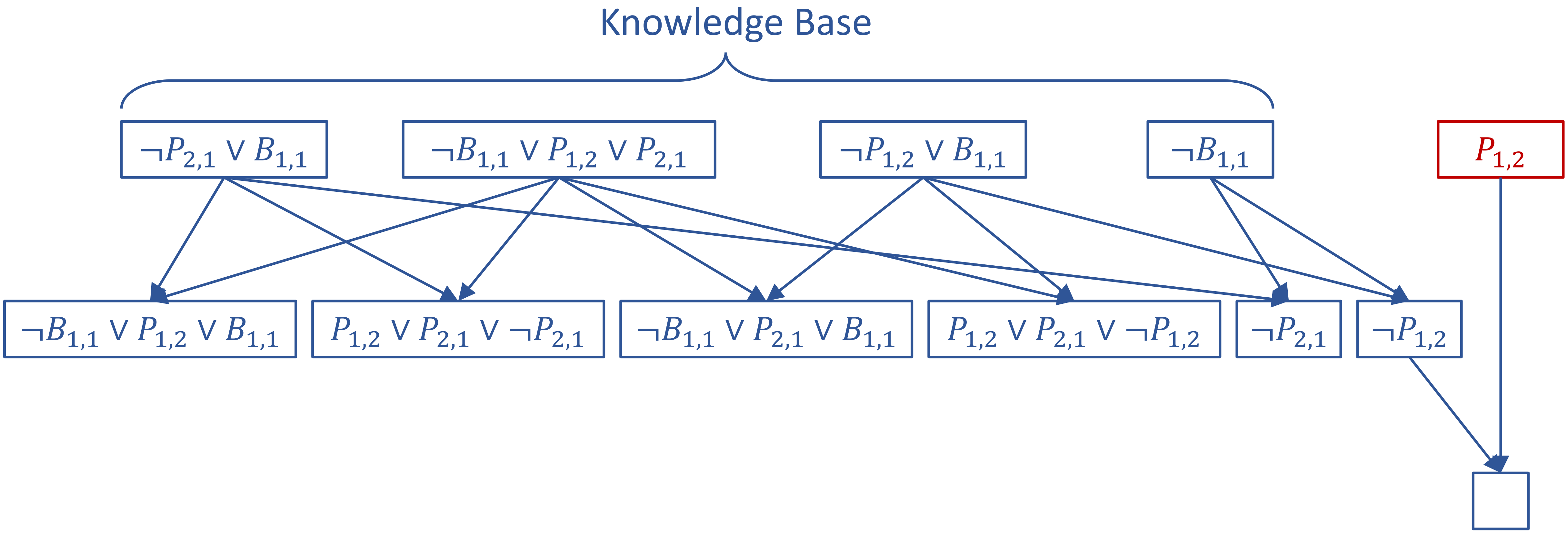
$\neg\neg P_{1,2}$

Resolution

Example trying to prove $\neg P_{1,2}$

General Resolution

$$\frac{a_1 \vee \cdots \vee a_m \vee b, \quad \neg b \vee c_1 \vee \cdots \vee c_n}{a_1 \vee \cdots \vee a_m \vee c_1 \vee \cdots \vee c_n}$$



Resolution

```
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 pair of clauses Ci, Cj in clauses do
      resolvents ← PL-RESOLVE(Ci, Cj)
      if resolvents contains the empty clause then
        return true
      new ← new ∪ resolvants
    if new ⊆ clauses then
      return false
    clauses ← clauses ∪ new
```

Properties

Forward Chaining is:

- Sound and complete for definite-clause KBs
- Complexity: linear time ☺

Resolution is another theorem-proving algorithm that is:

- Sound and complete for any PL KBs!
- Complexity: exponential time ☹

Vocab Reminder

Literal

- Atomic sentence:
  T, F, Symbol, ¬Symbol

Clause

- Disjunction of literals:
  $A \lor B \lor \neg C$

Definite clause

- Disjunction of literals, *exactly one* is positive
  $\neg A \lor B \lor \neg C$

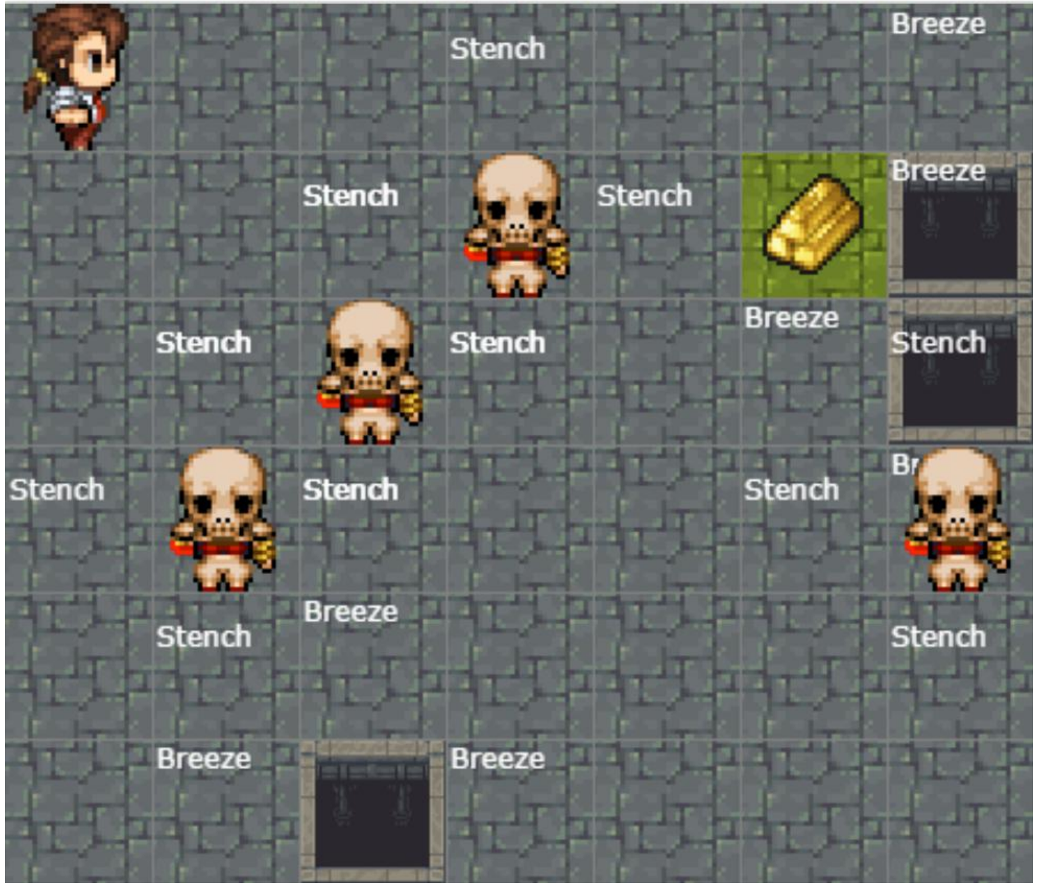
Outline

Logical Agent Algorithms

- Vocab
- PL_TRUE
- Entailment
  - Model checking: Truth table entailment
  - Theorem proving:
  - (Forward chaining), resolution
- Satisfiability: DPLL
- Planning with logic

Satisfiability and Entailment

A sentence is *satisfiable* if it is true in at least one world (e.g. CSPs!)



http://thiagodnf.github.io/wumpus-world-simulator/

Satisfiability and Entailment

A sentence is *satisfiable* if it is true in at least one world (cf CSPs!)

Suppose we have a hyper-efficient SAT solver; how can we use it to test entailment?

- Suppose $\alpha \models \beta$
- Then $\alpha \Rightarrow \beta$ is true in all worlds
- Hence $\neg(\alpha \Rightarrow \beta)$ is false in all worlds
- Hence $\alpha \wedge \neg\beta$ is false in all worlds, i.e., unsatisfiable

So, add the negated conclusion to what you know, test for (un)satisfiability; also known as `reductio ad absurdum`

Efficient SAT solvers operate on ***conjunctive normal form***

Efficient SAT solvers

DPLL (Davis-Putnam-Logemann-Loveland) is the core of modern solvers

Essentially a backtracking search over models with some extras:

- *Early termination*: stop if
  - all clauses are satisfied; e.g., $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by {A=true}
  - any clause is falsified; e.g., $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by {A=false, B=false}
- *Pure literals*: if all occurrences of a symbol in as-yet-unsatisfied clauses have the same sign, then give the symbol that value
  - E.g., A is pure and positive in $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$ so set it to true
- *Unit clauses*: if a clause is left with a single literal, set symbol to satisfy clause
  - E.g., if A=false, $(A \vee B) \wedge (A \vee \neg C)$ becomes $(\text{false} \vee B) \wedge (\text{false} \vee \neg C)$, i.e. $(B) \wedge (\neg C)$
  - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.

DPLL algorithm

```
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 or(DPLL(clauses, rest, model∪{P=true}),
             DPLL(clauses, rest, model∪{P=false}))
```

Outline

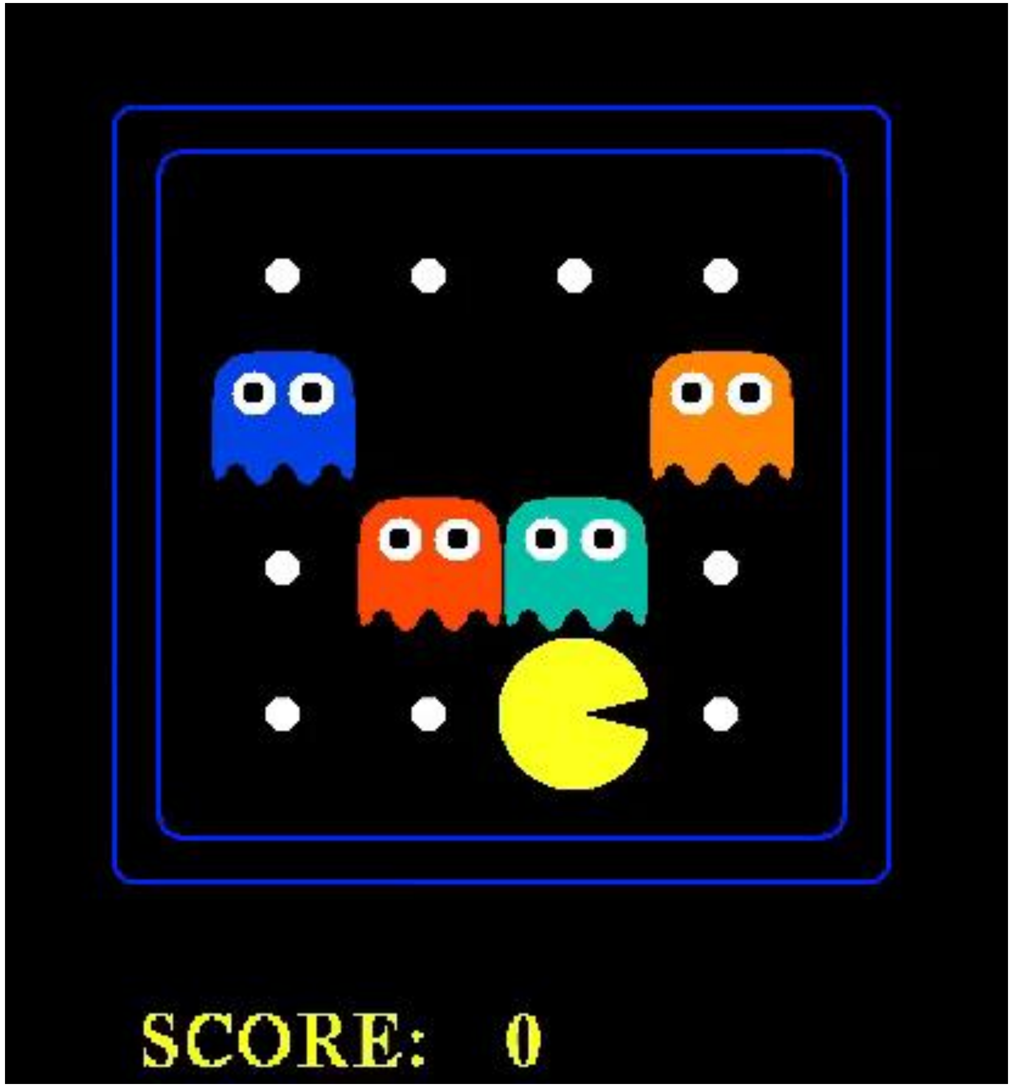
Logical Agent Algorithms

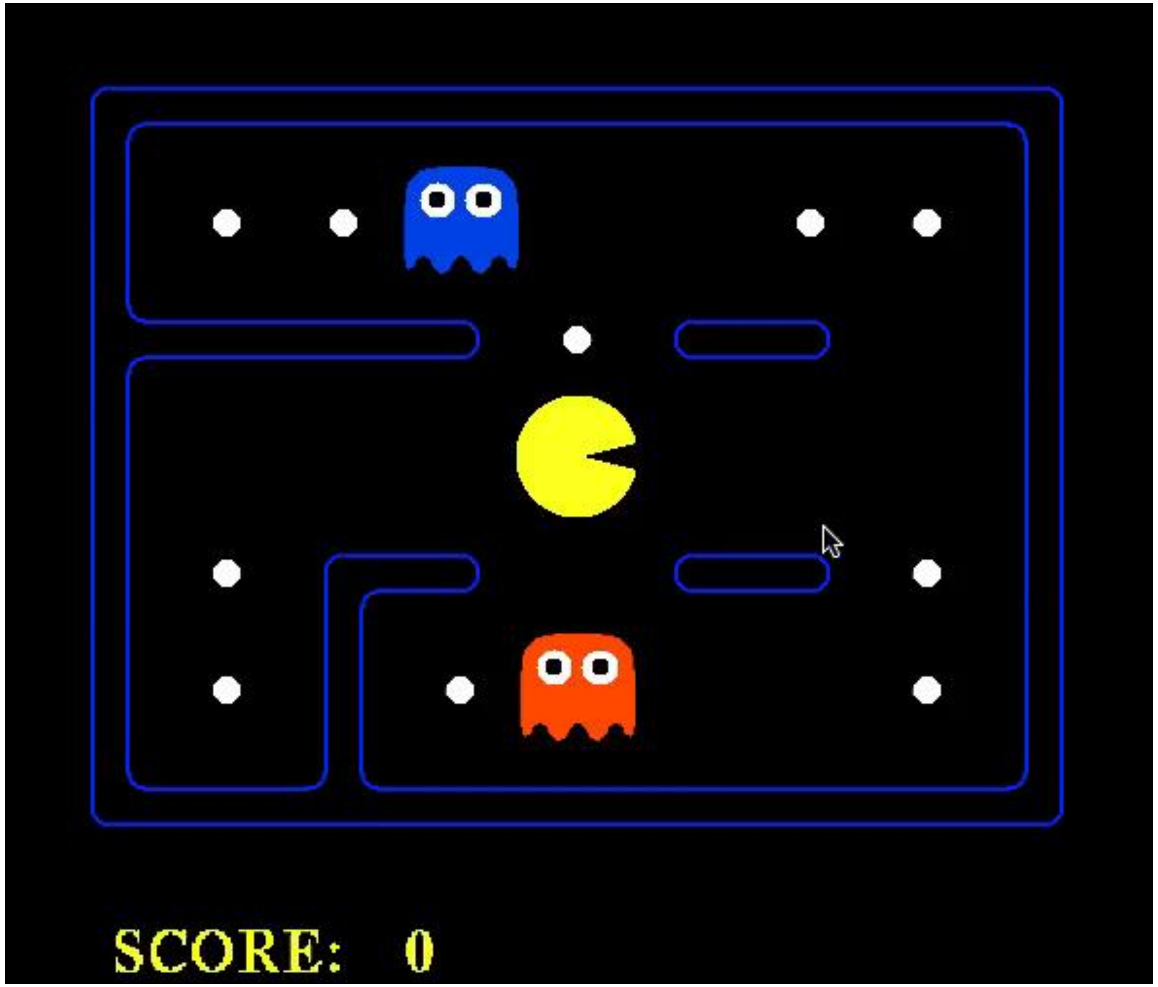
- Vocab
- PL_TRUE
- Entailment
  - Model checking: Truth table entailment
  - Theorem proving:
  - (Forward chaining), resolution
- Satisfiability: DPLL
- Planning with logic

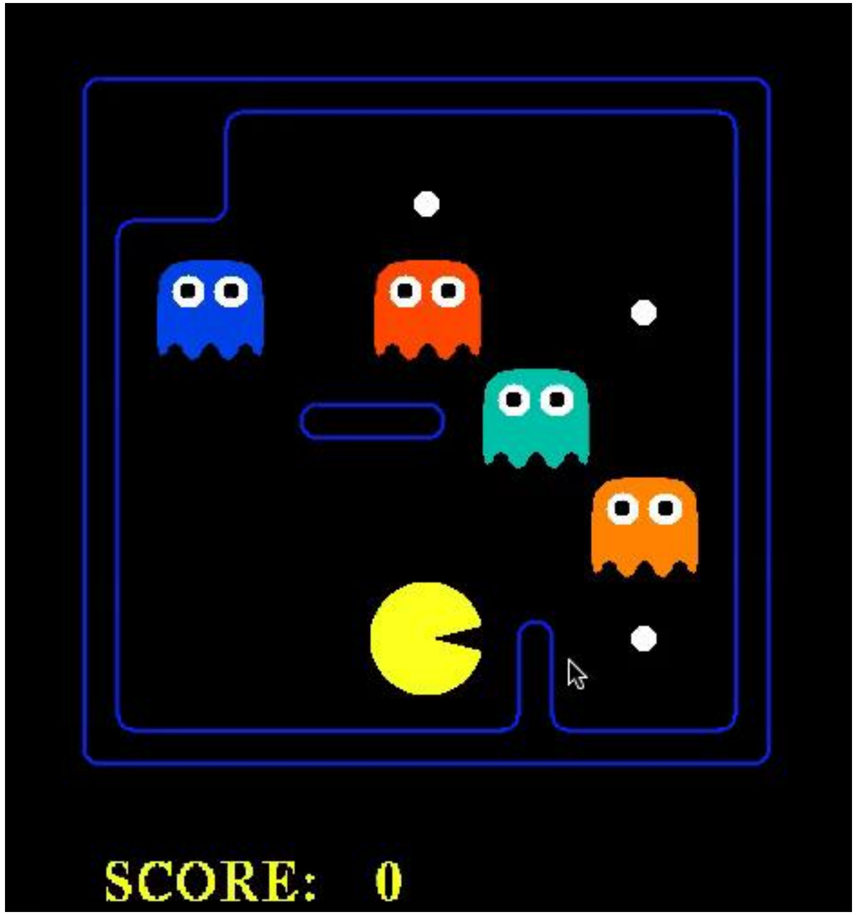
Planning as Satisfiability

Given a hyper-efficient SAT solver, can we use it to make plans?

Yes, for fully observable, deterministic case: planning problem is solvable iff there is some satisfying assignment for actions etc.


SCORE: 0


SCORE: 0


SCORE: 0

Planning as Satisfiability

Given a hyper-efficient SAT solver, can we use it to make plans?

Yes, for fully observable, deterministic case: planning problem is solvable iff there is some satisfying assignment for actions etc.

For T = 1 to infinity, set up the KB as follows and run SAT solver:

- Initial state, domain constraints
- Transition model sentences up to time T
- Goal is true at time T
- *Precondition axioms*: $At\_1,1\_0 \wedge N\_0 \Rightarrow \neg Wall\_1,2$ etc.
- *Action exclusion axioms*: $\neg(N\_0 \wedge W\_0) \wedge \neg(N\_0 \wedge S\_0) \wedge$ .. etc.

Initial State

The agent may know its initial location:

- At_1,1_0

Or, it may not:

- At_1,1_0 v At_1,2_0 v At_1,3_0 v … v At_3,3_0

We also need a *domain constraint* – cannot be in two places at once!

- ¬(At_1,1_0 ∧ At_1,2_0) ∧ ¬(At_1,1_0 ∧ At_1,3_0) ∧ …
- ¬(At_1,1_1 ∧ At_1,2_1) ∧ ¬(At_1,1_1 ∧ At_1,3_1) ∧ …
- …

Fluents and Effect Axioms

A *fluent* is a state variable that changes over time

How does each *state variable* or *fluent* at each time gets its value?

Fluents for PL Pacman are Pacman_x,y_t , e.g., Pacman _3,3_17

Fluents and Effect Axioms

A *fluent* is a state variable that changes over time

How does each *state variable* or *fluent* at each time gets its value?

Fluents for PL Pacman are Pacman_x,y_t , e.g., Pacman _3,3_17

Fluents and Successor-state Axioms

A *fluent* is a state variable that changes over time

How does each *state variable* or *fluent* at each time gets its value?

Fluents for PL Pacman are Pacman_*x*,*y*_*t* , e.g., Pacman _3,3_17

A state variable gets its value according to a *successor-state axiom*

- $X_t \Leftrightarrow [X_{t-1} \wedge \neg$(some action$_{t-1}$ made it false)] v
  [$\neg X_{t-1} \wedge$ (some action$_{t-1}$ made it true)]

Fluents and Successor-state Axioms

A *fluent* is a state variable that changes over time

How does each *state variable* or *fluent* at each time gets its value?

Fluents for PL Pacman are Pacman_x,y_t , e.g., Pacman _3,3_17

A state variable gets its value according to a *successor-state axiom*

- $X_t \Leftrightarrow [X_{t-1} \wedge \neg$(some action$_{t-1}$ made it false)$]$ v
  $[\neg X_{t-1} \wedge$ (some action$_{t-1}$ made it true)$]$

For Pacman location:

- Pacman _3,3_17 ⇔ [Pacman _3,3_16 ∧ ¬((¬Wall_3,4 ∧ N_16) v (¬Wall_4,3 ∧ E_16) v …)]
  v [¬ Pacman _3,3_16 ∧ ((Pacman _3,2_16 ∧ ¬Wall_3,3 ∧ N_16) v
  (Pacman _2,3_16 ∧ ¬Wall_3,3 ∧ N_16) v …)]