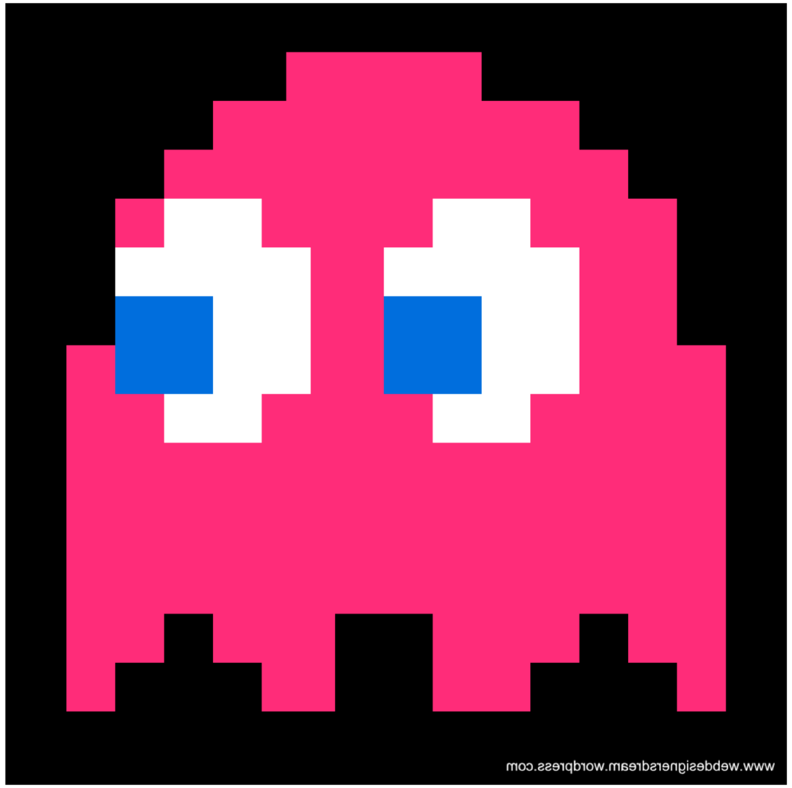
CS 188: Artificial Intelligence

Logic



Instructors: Angela Liu and Yanlai Yang

University of California, Berkeley

[These slides adapted from Stuart Russell and Dawn Song]

Outline

Propositional Logic

- Basic concepts of knowledge, logic, reasoning
- Propositional logic: syntax and semantics, Pacworld example
- Inference by theorem proving
- Inference by model checking
- A Pac agent using propositional logic

Agents that know things

- Agents acquire knowledge through perception, learning, language
  - Knowledge of the effects of actions (“transition model”)
  - Knowledge of how the world affects sensors (“sensor model”)
  - Knowledge of the current state of the world
- Can keep track of a partially observable world
- Can formulate plans to achieve goals

Knowledge, contd.

- Knowledge base = set of sentences in a formal language
- Declarative approach to building an agent (or other system):
  - ***Tell*** it what it needs to know (or have it ***Learn*** the knowledge)
  - Then it can ***Ask*** itself what to do—answers should follow from the KB
- Agents can be viewed at the ***knowledge level***
  i.e., what they ***know***, regardless of how implemented
- A single inference algorithm can answer any answerable question

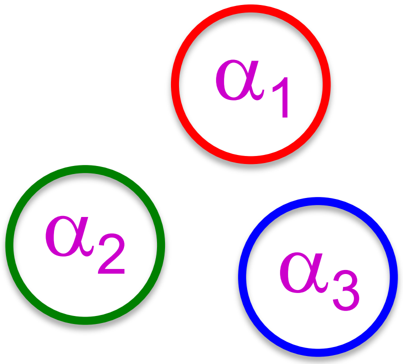
Knowledge base	Domain-specific facts
Inference engine	Generic code

Some reasoning tasks

- ***Localization*** with a map and local sensing:
  - Given an initial KB, plus a sequence of percepts and actions, where am I?
- ***Mapping*** with a location sensor:
  - Given an initial KB, plus a sequence of percepts and actions, what is the map?
- ***Simultaneous localization and mapping***:
  - Given …, where am I and what is the map?
- ***Planning***:
  - Given …, what action sequence is guaranteed to reach the goal?
- ***ALL OF THESE USE THE SAME KB AND THE SAME ALGORITHM!!***

Logic

- ***Syntax***: What sentences are allowed?
- ***Semantics***:
  - ***Possible worlds***
  - Which sentences are ***true*** in which worlds? (i.e., ***definition*** of truth)



Syntaxland

Semanticsland

A simple kind of logic

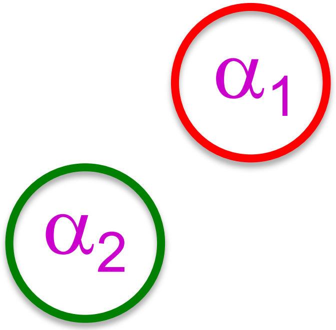
- Propositional logic
  - Syntax: $P \vee (\neg Q \wedge R)$; $X_1 \Leftrightarrow (\text{Raining} \Rightarrow \neg\text{Sunny})$
  - Possible world: {P=true,Q=true,R=false,S=true} or 1101
  - Semantics: $\alpha \wedge \beta$ is true in a world iff is α true and β is true (etc.)

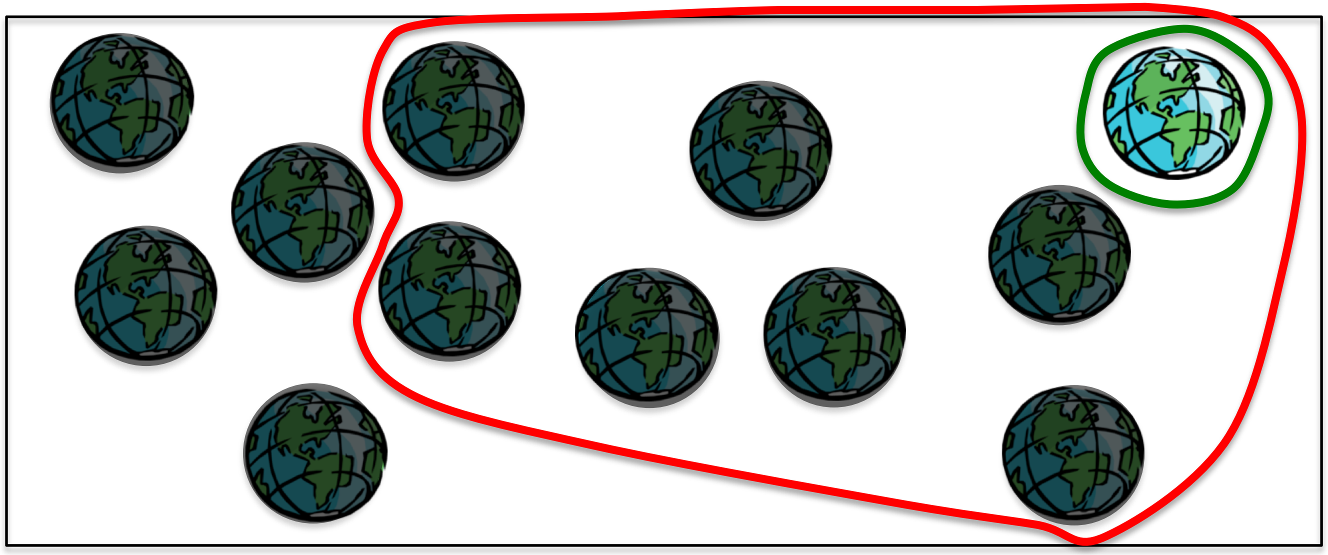
Propositional logic syntax

- Given: a set of proposition symbols $\{X_1, X_2, \ldots, X_n\}$
  - (we often add True and False for convenience)
- X_i is a sentence
- If α is a sentence then $\neg\alpha$ is a sentence
- If α and β are sentences then $\alpha \wedge \beta$ is a sentence
- If α and β are sentences then $\alpha \vee \beta$ is a sentence
- If α and β are sentences then $\alpha \Rightarrow \beta$ is a sentence
- If α and β are sentences then $\alpha \Leftrightarrow \beta$ is a sentence
- And p.s. there are no other sentences!

Inference: entailment

- ***Entailment***: $\alpha \models \beta$ ("α entails β" or "β follows from α") iff in every world where α is true, β is also true
  - I.e., the α-worlds are a subset of the β-worlds [***models***(α) $\subseteq$ ***models***(β)]
- In the example, $\alpha_2 \models \alpha_1$
- (Say α_2 is $\neg Q \wedge R \wedge S \wedge W$
  α_1 is $\neg Q$)





Inference: proofs

- 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)
- Method 2: ***theorem-proving***
  - Search for a sequence of proof steps (applications of ***inference rules***) leading from α to β
  - E.g., from $P \land (P \Rightarrow Q)$, infer Q by ***Modus Ponens***

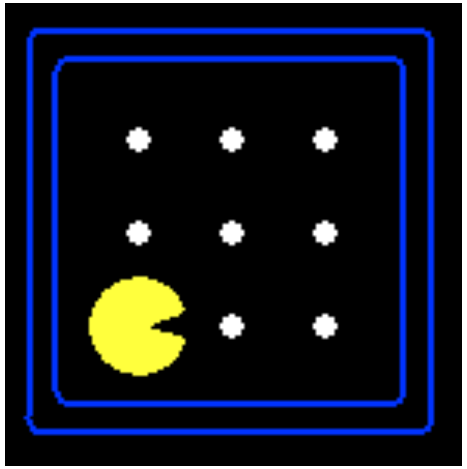
Propositional logic semantics in code

```
function PL-TRUE?(α,model) returns true or false
    if α is a symbol then return Lookup(α, model)
    if Op(α) = ¬ then return not(PL-TRUE?(Arg1(α),model))
    if Op(α) = ∧ then return  and(PL-TRUE?(Arg1(α),model),
                                  PL-TRUE?(Arg2(α),model))
    etc.
```

(Sometimes called “recursion over syntax”)

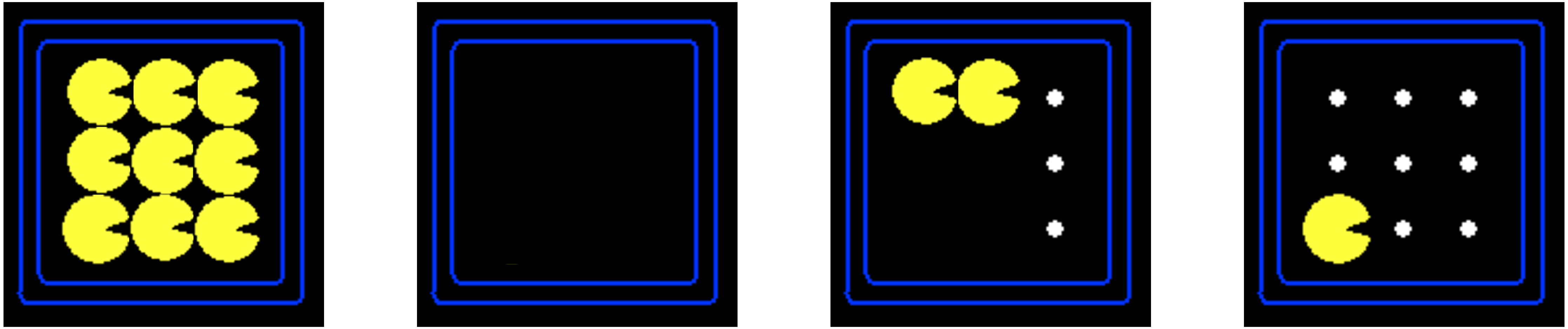
Example: Partially observable Pacman

- Pacman knows the map but perceives just wall/gap to NSEW
- Formulation: ***what variables do we need?***
  - Wall locations
    - Wall_0,0   there is a wall at [0,0]
    - Wall_0,1   there is a wall at [0,1], etc. (*N* symbols for *N* locations)
  - Percepts
    - ~~Blocked_W (blocked by wall to my West) etc.~~
    - Blocked_W_0 (blocked by wall to my West ***at time 0***) etc. (4*T* symbols for *T* time steps)
  - Actions
    - W_0 (Pacman moves West at time 0), E_0 etc. (4*T* symbols)
  - Pacman's location
    - At_0,0_0 (Pacman is at [0,0] at time 0), At_0,1_0 etc. (*NT* symbols)



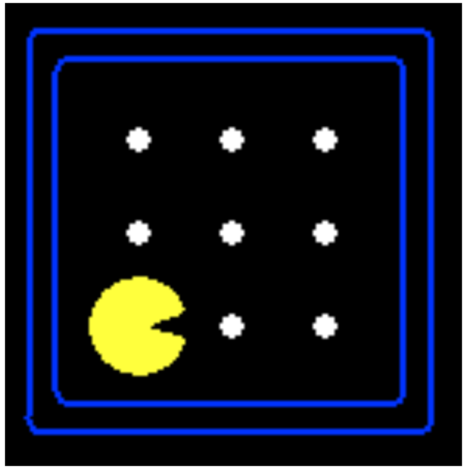
How many possible worlds?

- N locations, T time steps => $N + 4T + 4T + NT = O(NT)$ variables
- $O(2^{NT})$ possible worlds!
- N=200, T=400 => ~10^{24000} worlds
- Each world is a complete "history"
  - But most of them are pretty weird!



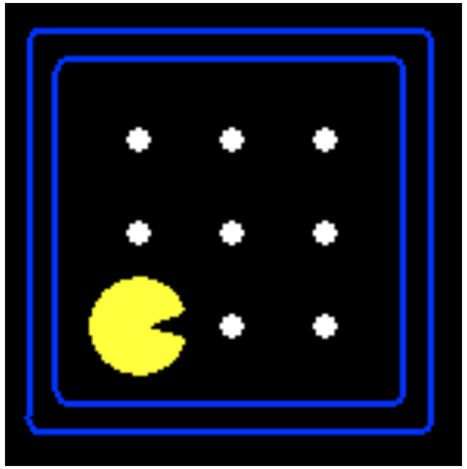
Pacman's knowledge base: Map

- Pacman knows where the walls are:
  - Wall_0,0 ∧ Wall_0,1 ∧ Wall_0,2 ∧ Wall_0,3 ∧ Wall_0,4 ∧ Wall_1,4 ∧ …
- Pacman knows where the walls aren't!
  - ¬Wall_1,1 ∧ ¬Wall_1,2 ∧ ¬Wall_1,3 ∧ ¬Wall_2,1 ∧ ¬Wall_2,2 ∧ …



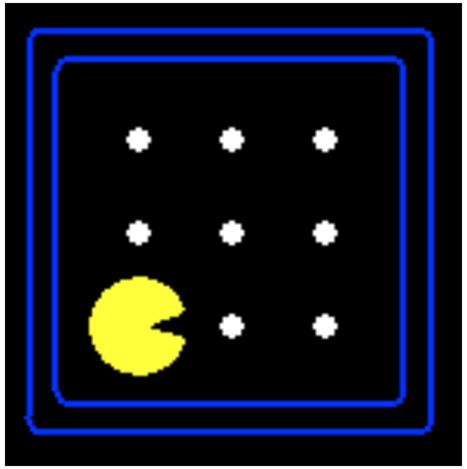
Pacman's knowledge base: Initial state

- Pacman doesn't know where he is
- But he knows he's somewhere!
  - At_1,1_0 ∨ At_1,2_0 ∨ At_1,3_0 ∨ At_2,1_0 ∨ …



Pacman's knowledge base: Sensor model

- State facts about how Pacman's percepts arise…
  - <Percept variable at t> ⇔ <some condition on world at t>
- Pacman perceives a wall to the West at time *t* ***if and only if*** he is in *x,y* and there is a wall at *x-1,y*
  - Blocked_W_0 ⇔ ((At_1,1_0 ∧ Wall_0,1) v
    (At_1,2_0 ∧ Wall_0,2) v
    (At_1,3_0 ∧ Wall_0,3) v ….)
  - 4T sentences, each of size *O*(*N*)
  - Note: these are valid for any map



Pacman's knowledge base: Transition model

- How does each ***state variable*** at each time gets its value?
  - Here we care about location variables, e.g., At_3,3_17
- A state variable X gets its value according to a ***successor-state axiom***
  - X_t ⇔ [X_t-1 ∧ ¬(some action_t-1 made it false)] v
    [¬X_t-1 ∧ (some action_t-1 made it true)]
- For Pacman location:
  - At_3,3_17 ⇔ [At_3,3_16 ∧ ¬((¬Wall_3,4 ∧ N_16) v (¬Wall_4,3 ∧ E_16) v …)]
    v [¬At_3,3_16 ∧ ((At_3,2_16 ∧ ¬Wall_3,3 ∧ N_16) v
    (At_2,3_16 ∧ ¬Wall_3,3 ∧ N_16) v …)]

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 $X_1, X_2, \ldots, X_n$, ***infer*** Y
- Forward chaining keeps applying this rule, adding new facts, until nothing more can be added
- Requires KB to contain only ***definite clauses***:
  - (Conjunction of symbols) $\Rightarrow$ symbol; or
  - A single symbol (note that X is equivalent to $True \Rightarrow X$)
- Runs in ***linear*** time using two simple tricks:
  - Each symbol X_i knows which rules it appears in
  - Each rule keeps count of how many of its premises are not yet satisfied

Forward chaining algorithm: Details

```
function PL-FC-ENTAILS?(KB, q) returns true or false
   count ← a table, where count[c] is the number of symbols in c's premise
   inferred ← a table, where inferred[s] is initially false for all s
   agenda ← a queue of symbols, initially symbols known to be true in KB
   while agenda is not empty do
         p ← Pop(agenda)
         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 agenda
   return false
```

Satisfiability and entailment

- A sentence is ***satisfiable*** if it is true in at least one world
- Suppose we have a hyper-efficient SAT solver (WARNING: NP-COMPLETE 👿👿👿); how can we use it to test entailment?
  - $\alpha \models \beta$
  - iff $\alpha \Rightarrow \beta$ is true in all worlds
  - iff $\neg(\alpha \Rightarrow \beta)$ is false in all worlds
  - iff $\alpha \wedge \neg\beta$ 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
- Recursive depth-first search over models with some extras:
  - ***Early termination***: stop if
    - all clauses are satisfied; e.g., $(A \lor B) \land (A \lor \neg C)$ is satisfied by {A=true}
    - any clause is falsified; e.g., $(A \lor B) \land (A \lor \neg C)$ 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 $(A \lor B) \land (A \lor \neg C) \land (C \lor \neg B)$ 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, $(A \lor B) \land (A \lor \neg C)$ becomes $(\text{false} \lor B) \land (\text{false} \lor \neg C)$, i.e. $(B) \land (\neg C)$
    - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.

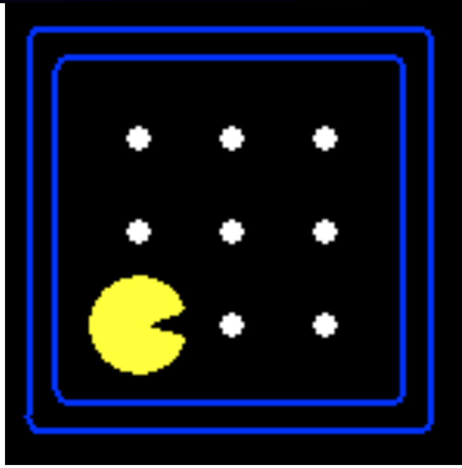
DPLL algorithm

```
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, modelU{P=value})
  P,value ←FIND-UNIT-CLAUSE(clauses,model)
  if P is non-null then return DPLL(clauses, symbols–P, modelU{P=value})
  P ← First(symbols); rest ← Rest(symbols)
  return or(DPLL(clauses,rest,modelU{P=true}),
            DPLL(clauses,rest,modelU{P=false}))
```

A knowledge-based agent

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

Reminder: Partially observable Pacman



- Pacman perceives wall/no-wall in each direction
- Variables:
  - Wall_0,0, Wall_0,1, …
  - Blocked_W_0, Blocked_N_0, …, Blocked_W_1, …
  - W_0 , N_0, …, W_1, …
  - At_0,0_0 , At_0,1_0, …, At_0,0_1 , …

Pacman’s knowledge base: Basic PacPhysics

- **Map**: where the walls are and aren’t
- **Initial state**: Pacman is definitely somewhere
- **Domain constraints**:
  - Pacman does exactly one action at each step
  - Pacman is in exactly one location at each step
- **Sensor model**: <Percept_t> ⇔ <some condition on world_t>
- **Transition model**:
  - <at x,y_t> ⇔ [at x,y_t-1 and stayed put] v [next to x,y_t-1 and moved to x,y]

State estimation

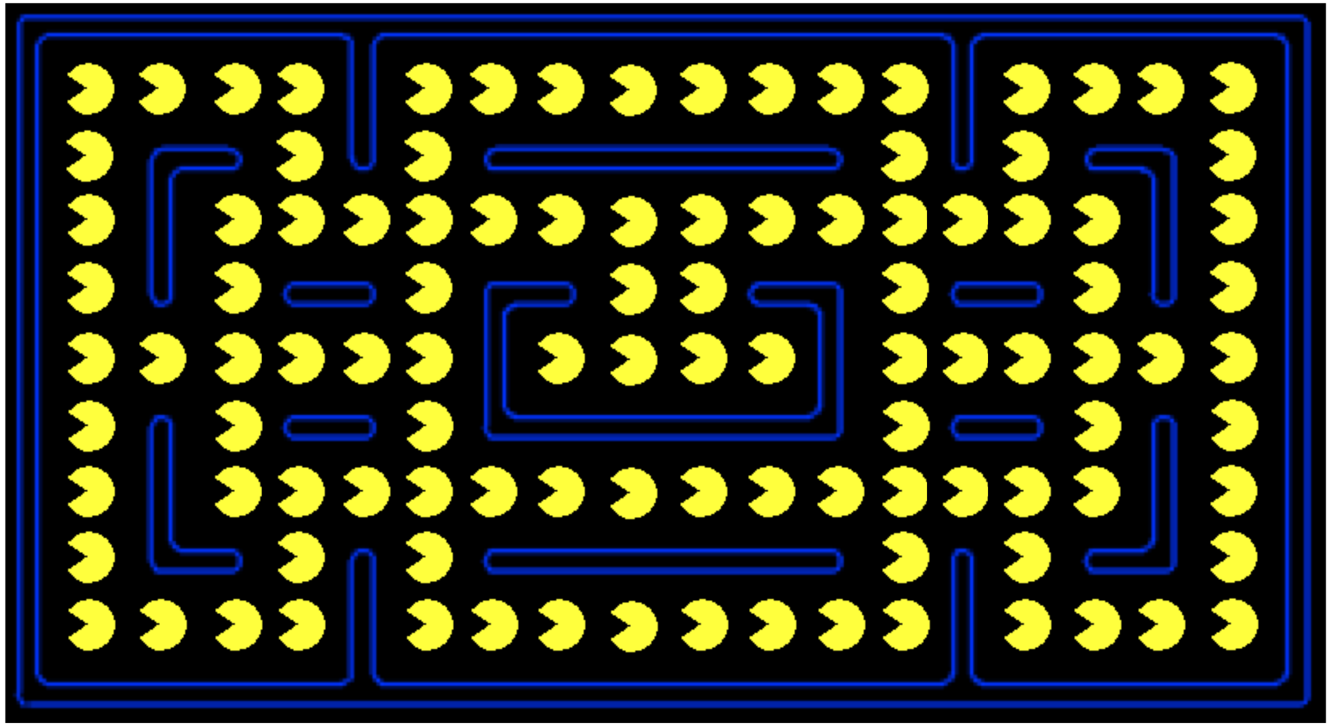
- ***State estimation*** means keeping track of what's true now
- A logical agent can just ask itself!
  - E.g., ask whether KB ∧ <actions> ∧ <percepts> |= At_2,2_6
- This is "lazy": it analyzes one's whole life history at each step!
- A more "eager" form of state estimation:
  - After each action and percept
    - For each state variable X_t
      - If KB ∧ action_t-1 ∧ percept_t |= X_t, add X_t to KB
      - If KB ∧ action_t-1 ∧ percept_t |= ¬X_t, add ¬X_t to KB

Example: Localization in a known map

- Initialize the KB with **PacPhysics** for *T* time steps
- Run the Pacman agent for *T* time steps:
  - After each action and percept
    - For each variable At_x,y_t
      - If KB ∧ action_t-1 ∧ percept_t |= At_x,y_t, add At_x,y_t to KB
      - If KB ∧ action_t-1 ∧ percept_t |= ¬ At_x,y_t, add ¬ At_x,y_t to KB
    - Choose an action
- Pacman's ***possible*** locations are those that are not provably false

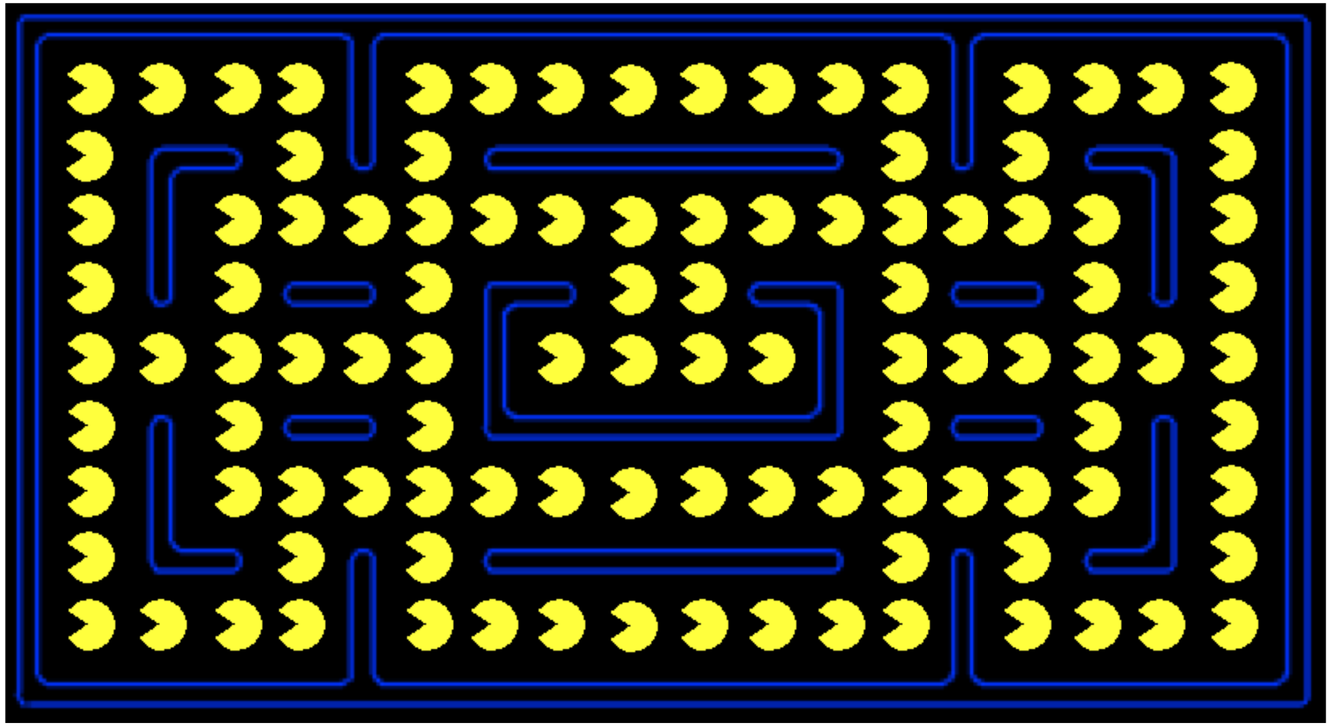
Localization demo

- Percept 
- Action
- Percept
- Action
- Percept
- Action
- Percept



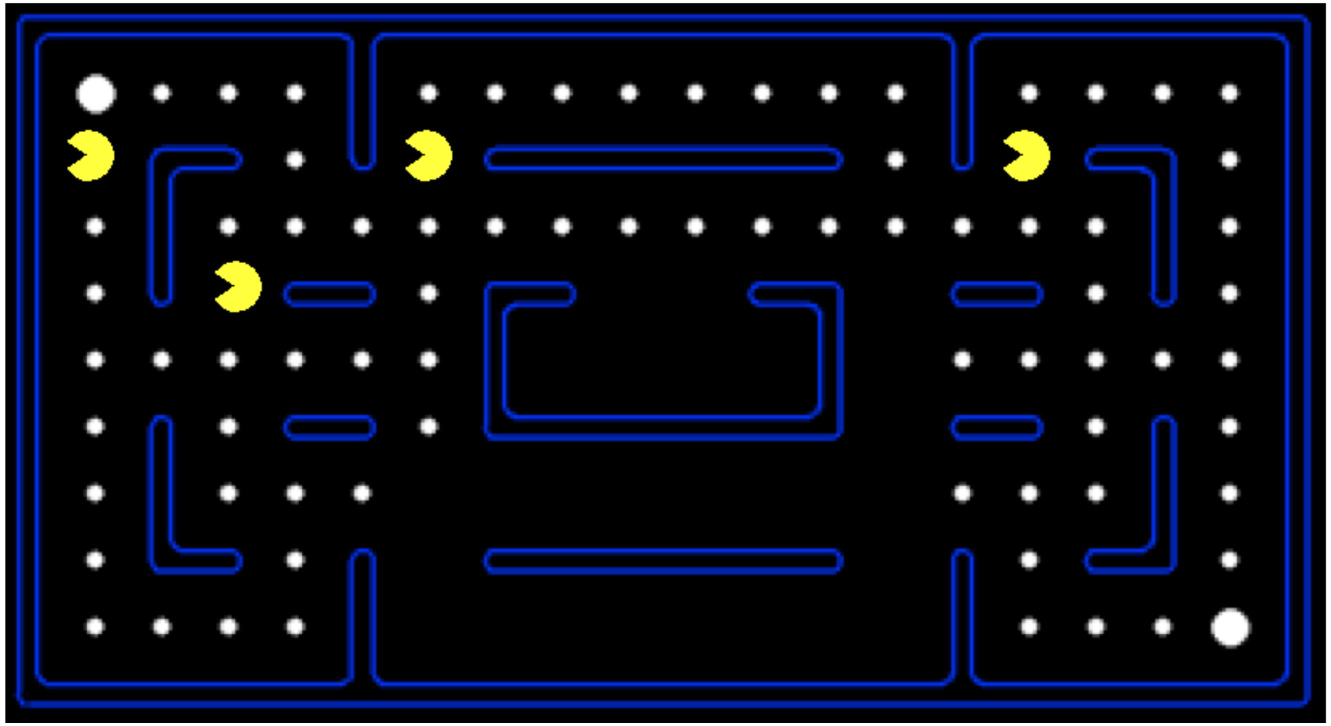
Localization demo

- Percept 
- Action *SOUTH*
- Percept
- Action
- Percept
- Action
- Percept



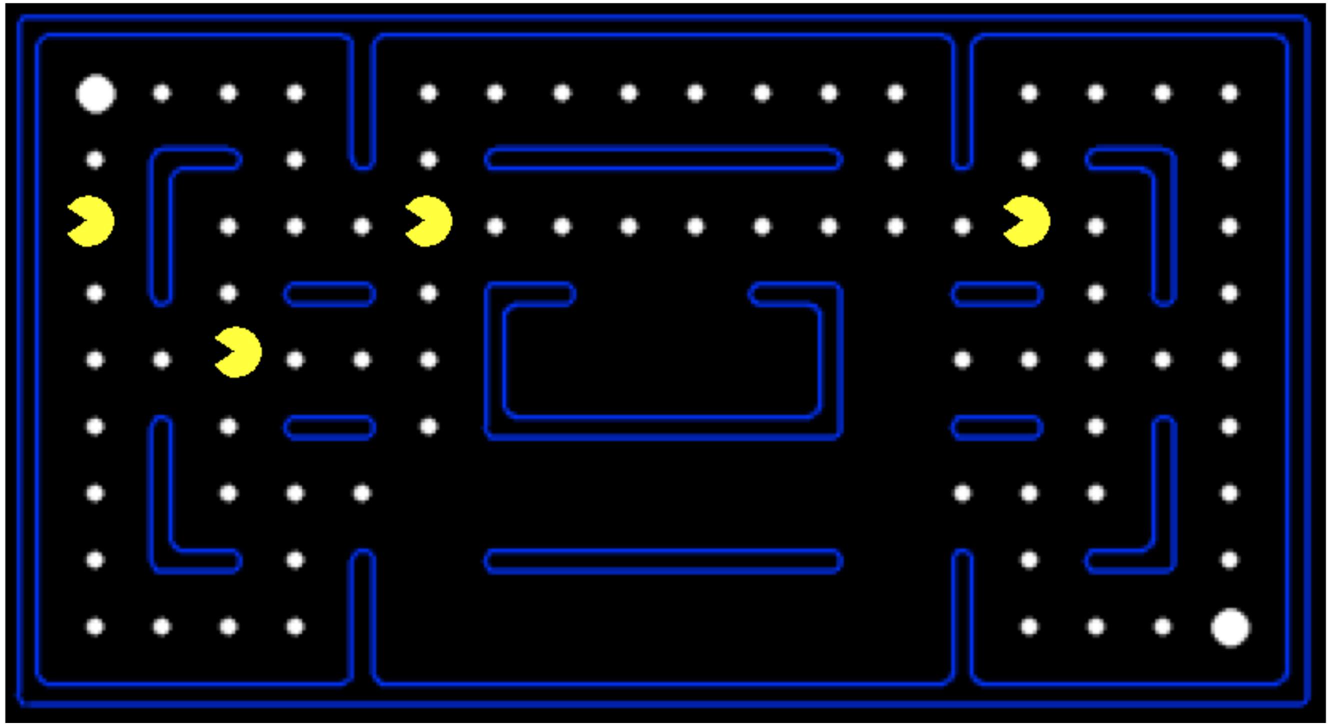
Localization demo

- Percept 
- Action *SOUTH*
- Percept 
- Action *SOUTH*
- Percept
- Action
- Percept

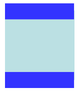


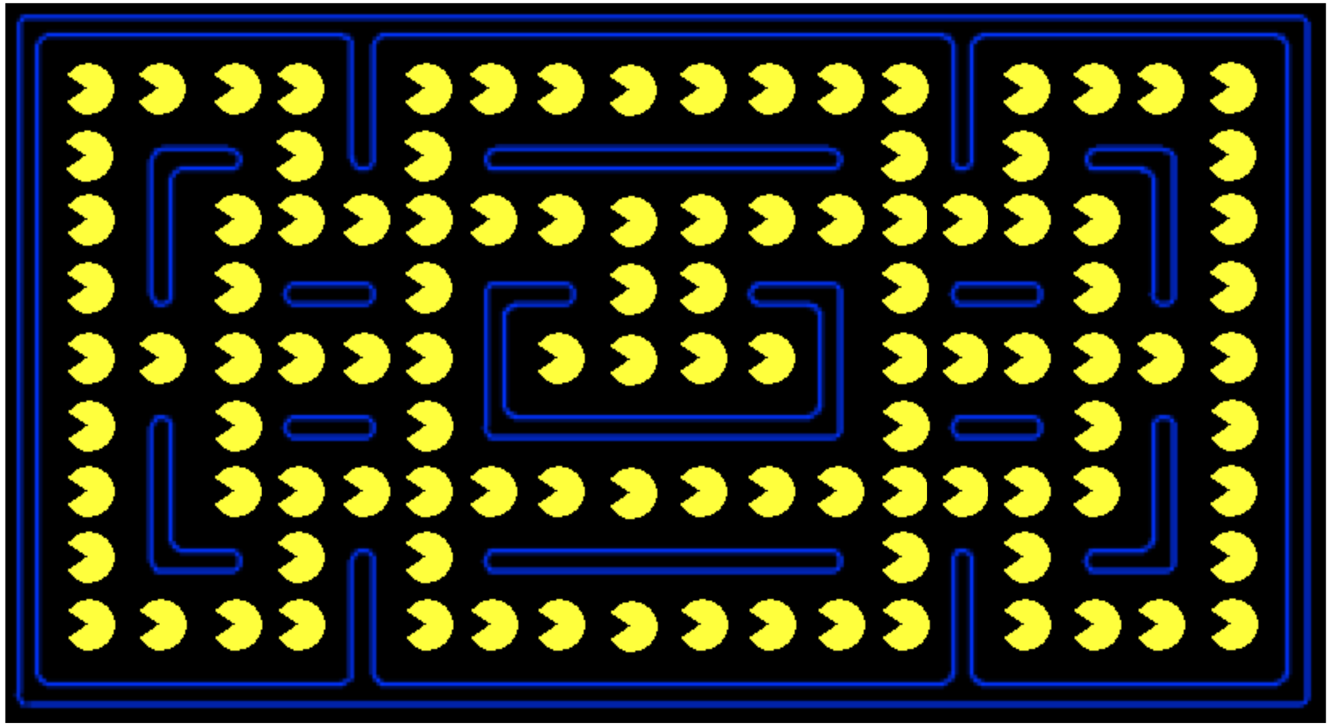
Localization demo

- Percept 
- Action *SOUTH*
- Percept 
- Action *SOUTH*
- Percept 
- Action
- Percept

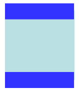


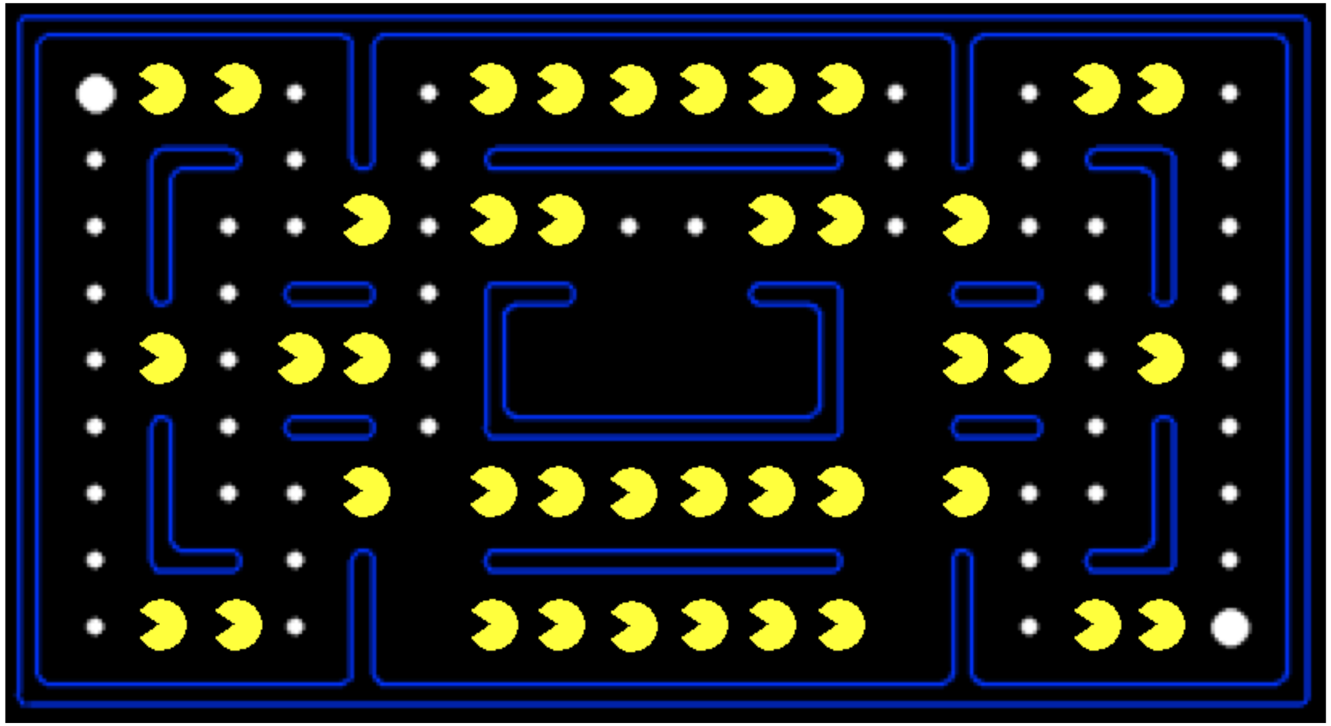
Localization demo

- Percept 
- Action
- Percept
- Action
- Percept
- Action
- Percept

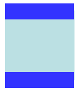
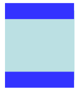


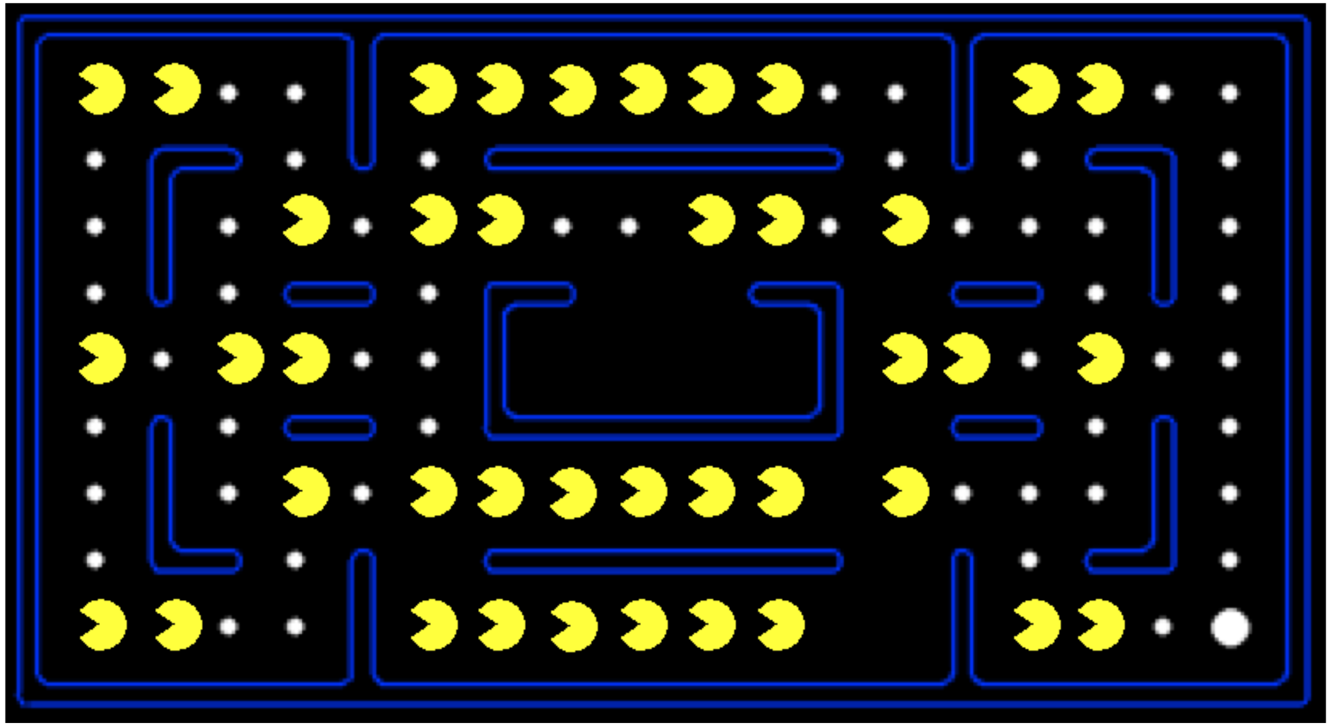
Localization demo

- Percept 
- Action *WEST*
- Percept
- Action
- Percept
- Action
- Percept

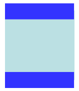
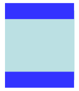


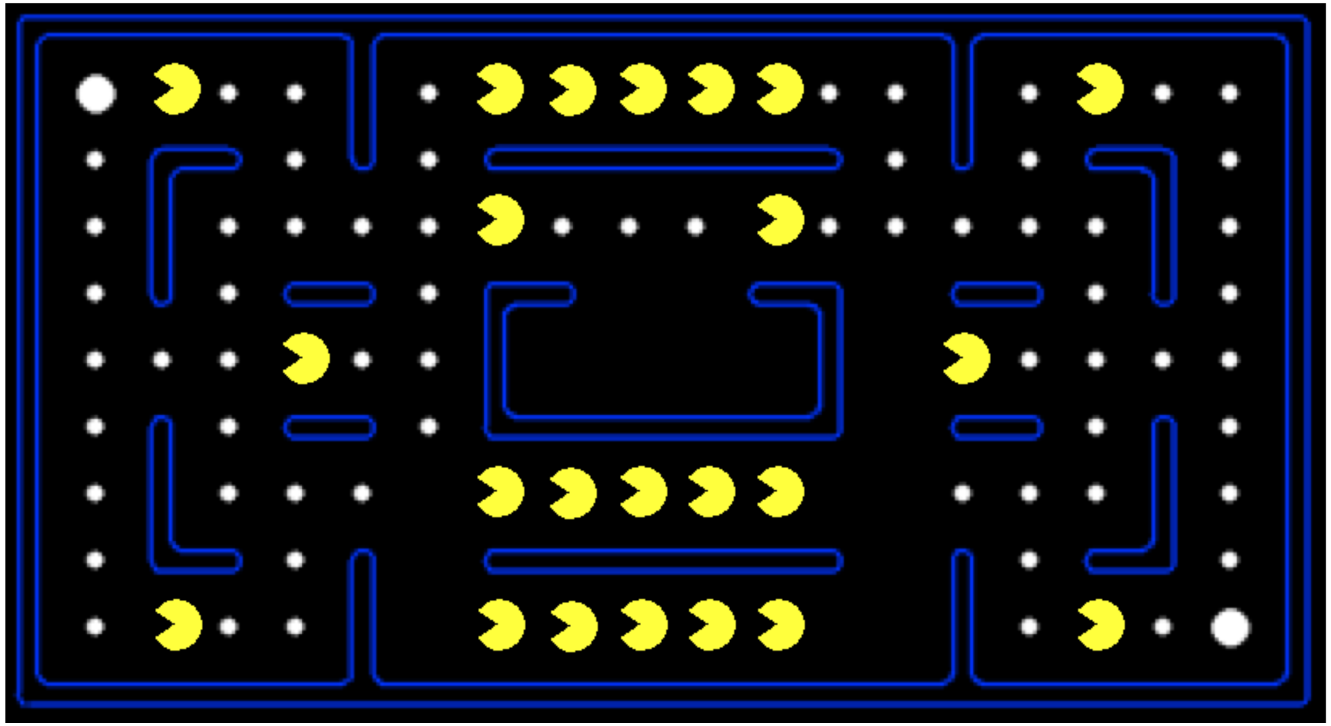
Localization demo

- Percept 
- Action *WEST*
- Percept 
- Action
- Percept
- Action
- Percept

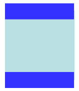
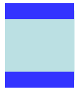
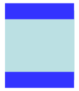


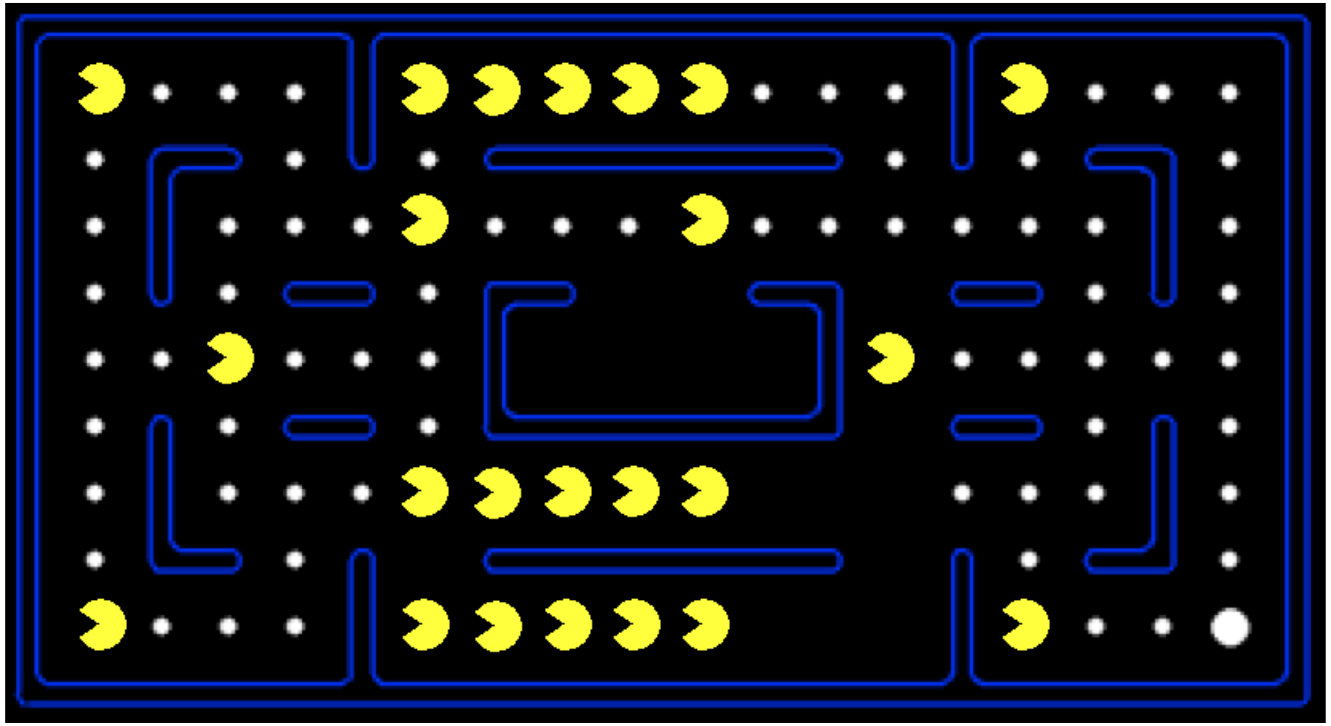
Localization demo

- Percept 
- Action *WEST*
- Percept 
- Action *WEST*
- Percept
- Action
- Percept

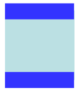
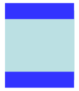
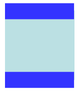


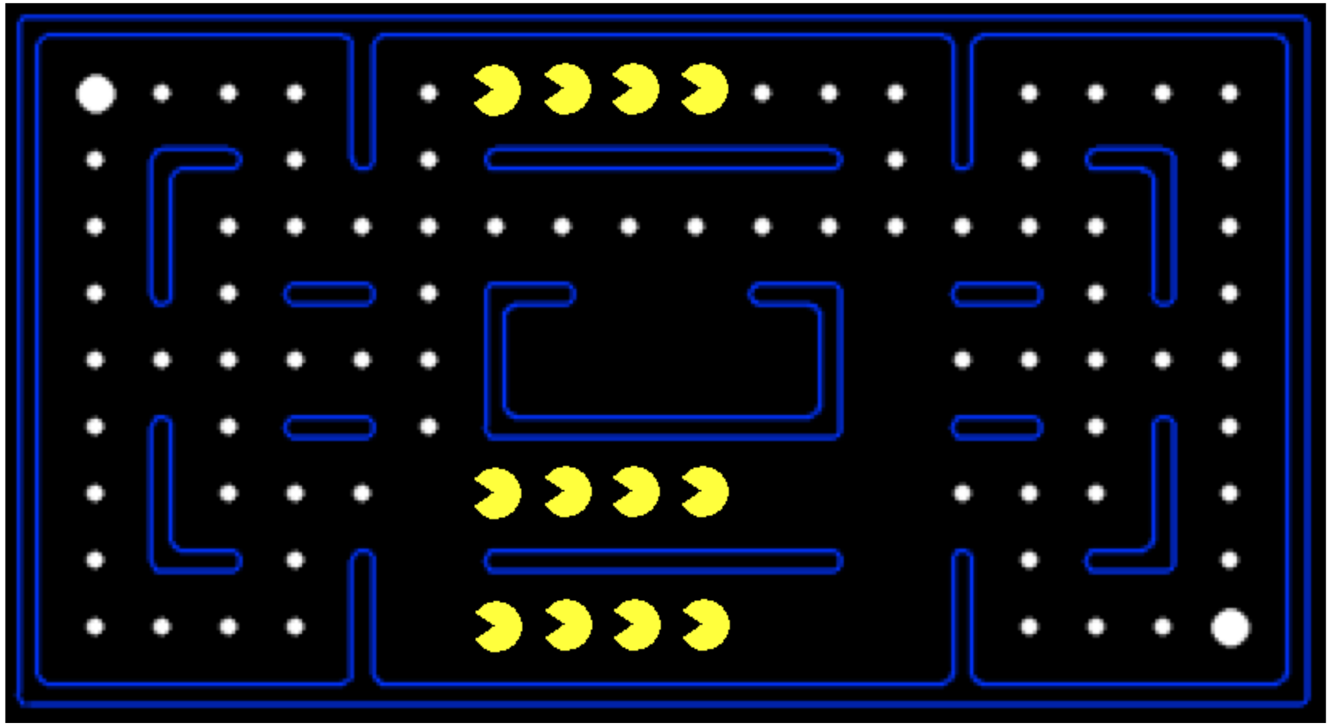
Localization demo

- Percept 
- Action *WEST*
- Percept 
- Action *WEST*
- Percept 
- Action
- Percept

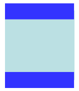
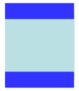
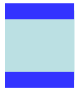


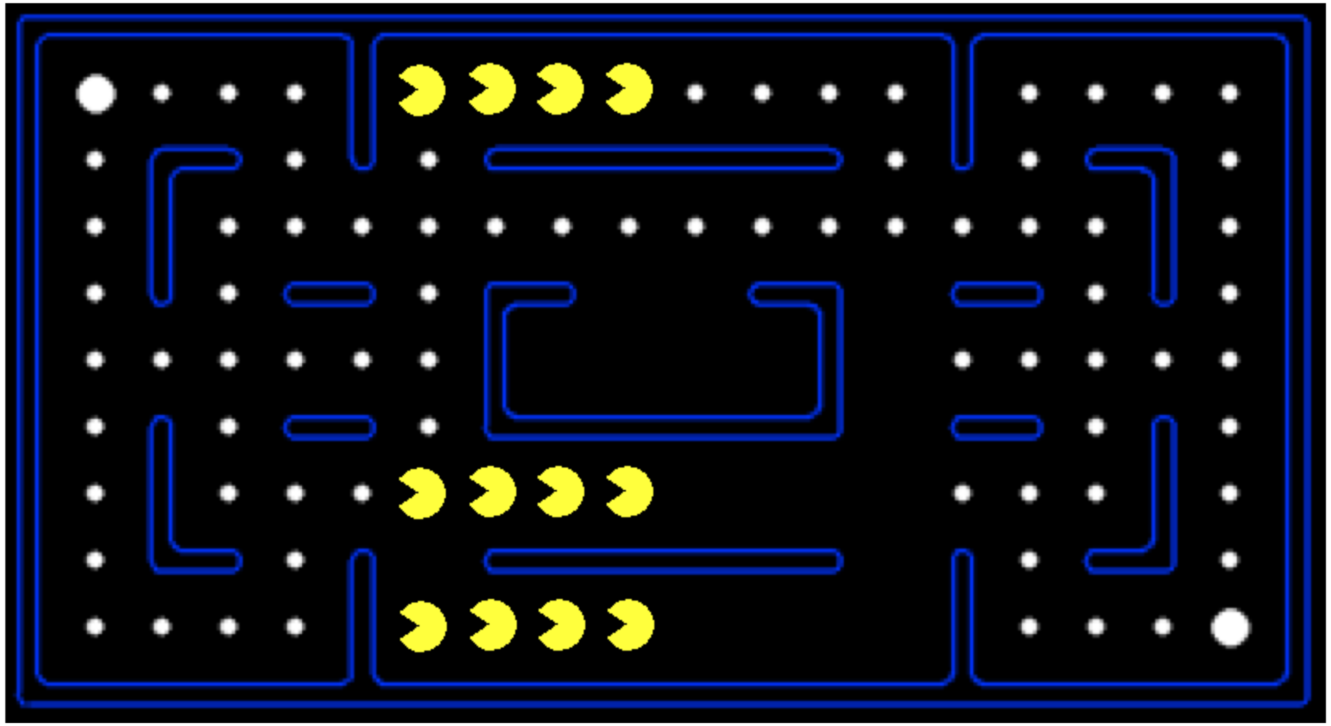
Localization demo

- Percept 
- Action *WEST*
- Percept 
- Action *WEST*
- Percept 
- Action *WEST*
- Percept

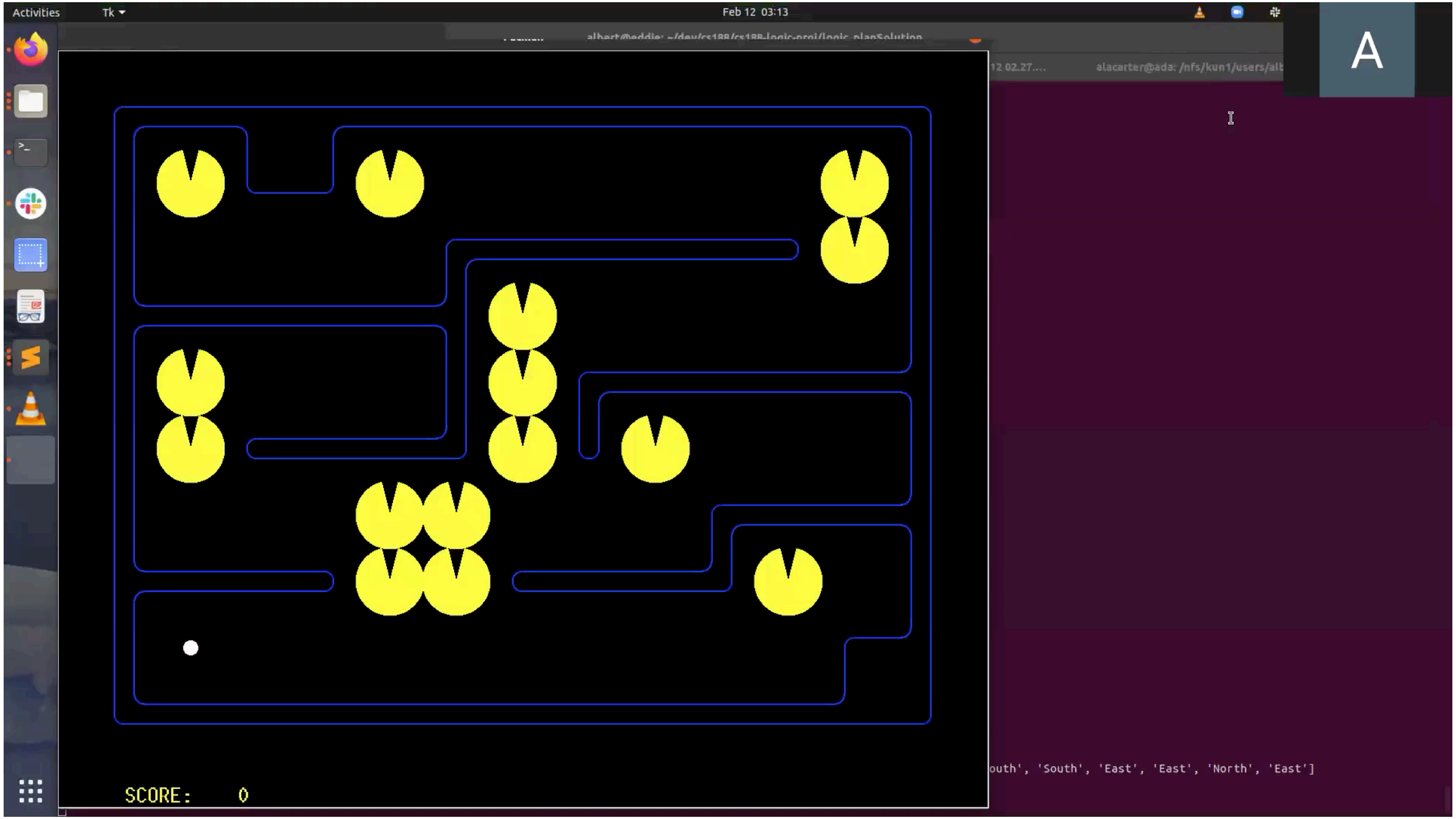


Localization demo

- Percept 
- Action *WEST*
- Percept 
- Action *WEST*
- Percept 
- Action *WEST*
- Percept



Localization with random movement



Example: Mapping from a known relative location

- Without loss of generality, call the initial location 0,0
- The percept tells Pacman which actions work, so he always knows where he is
  - "Dead reckoning"
- Initialize the KB with **PacPhysics** for *T* time steps, starting at 0,0
- Run the Pacman agent for *T* time steps
  - At each time step
    - Update the KB with previous action and new percept facts
    - For each wall variable Wall_x,y
      - If Wall_x,y is entailed, add to KB
      - If ¬Wall_x,y is entailed, add to KB
    - Choose an action
- The wall variables constitute the map

Mapping demo

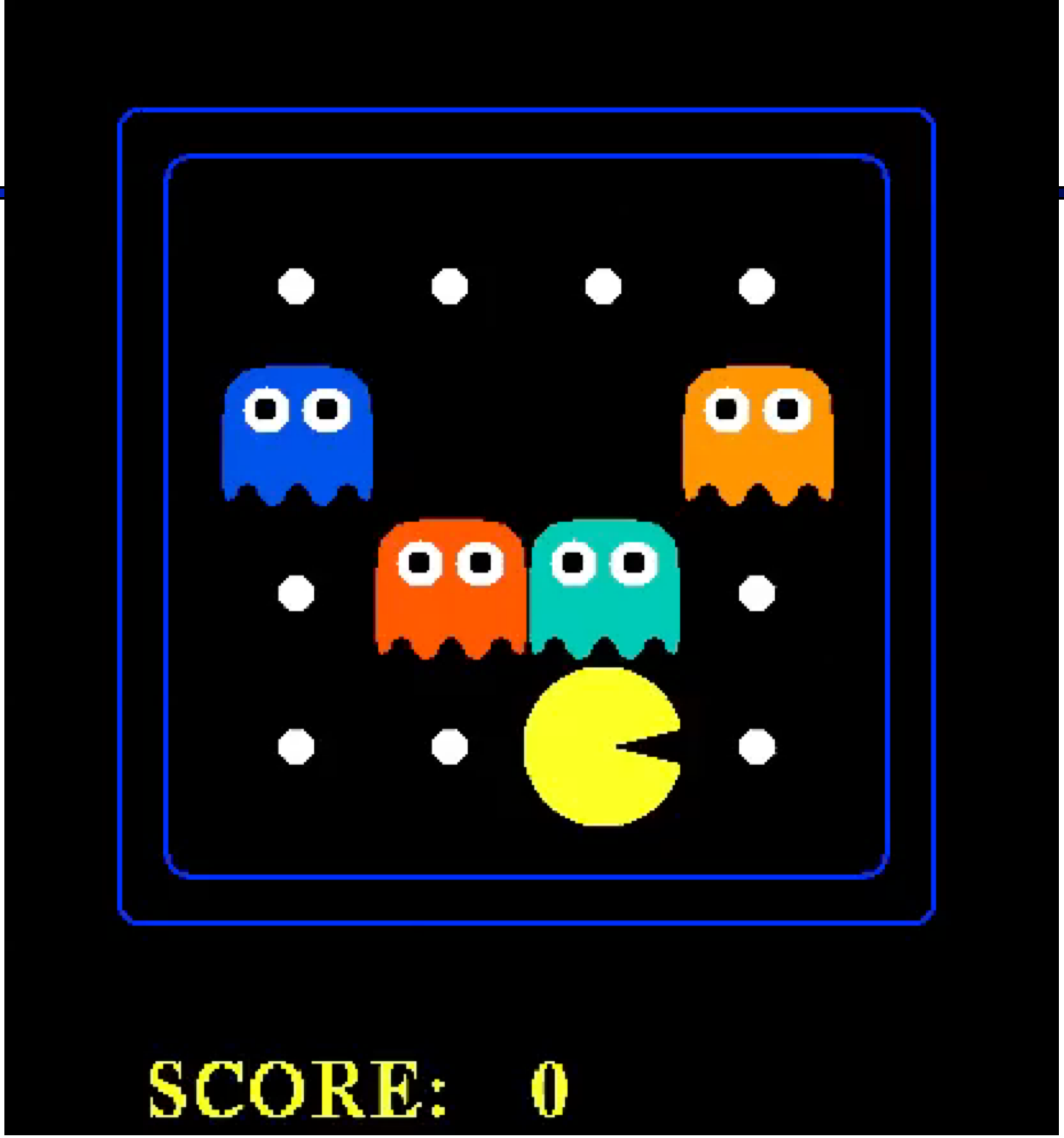
- Percept 
- Action *NORTH*
- Percept 
- Action *EAST*
- Percept 
- Action *SOUTH*
- Percept 

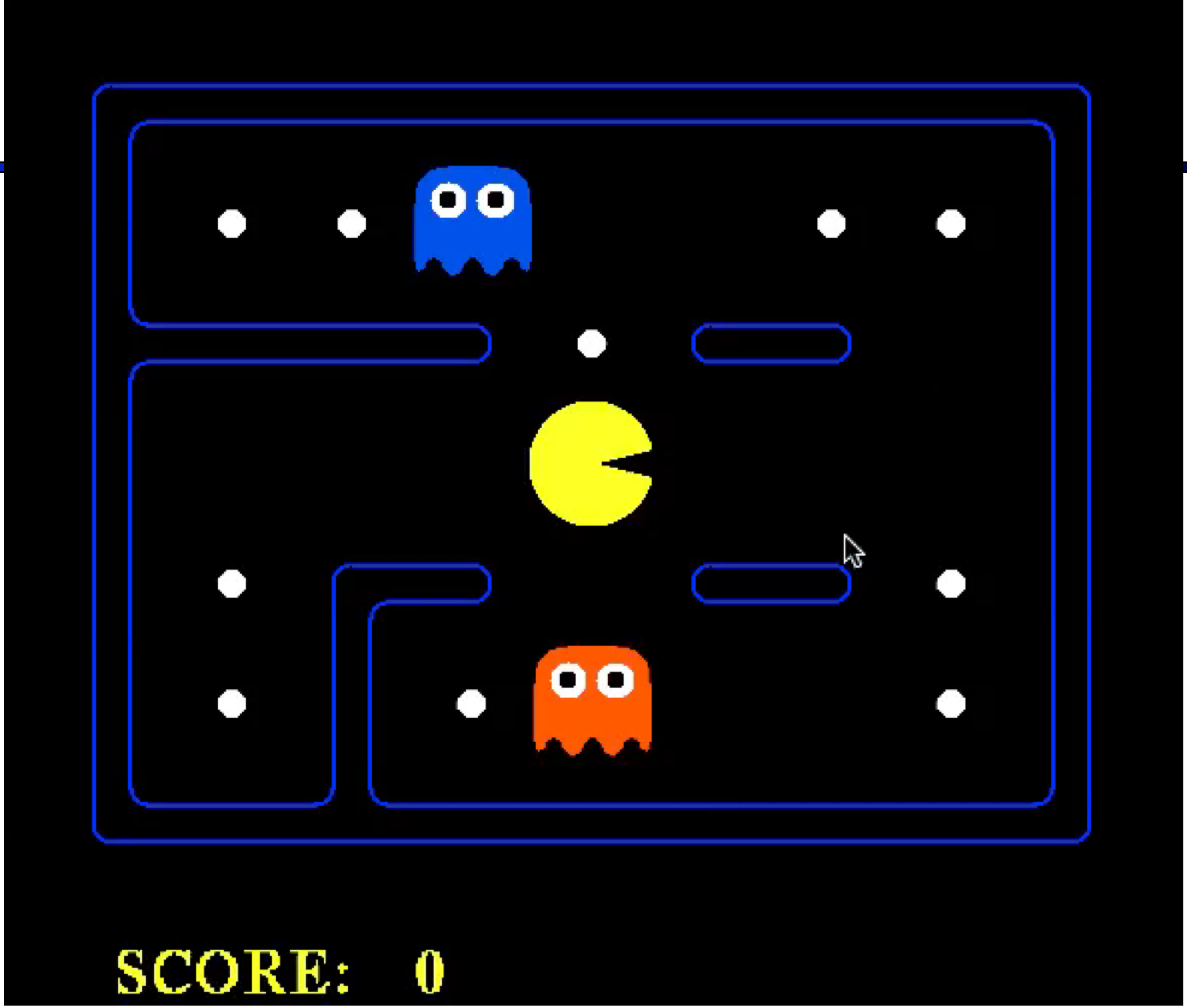
Example: Simultaneous localization and mapping

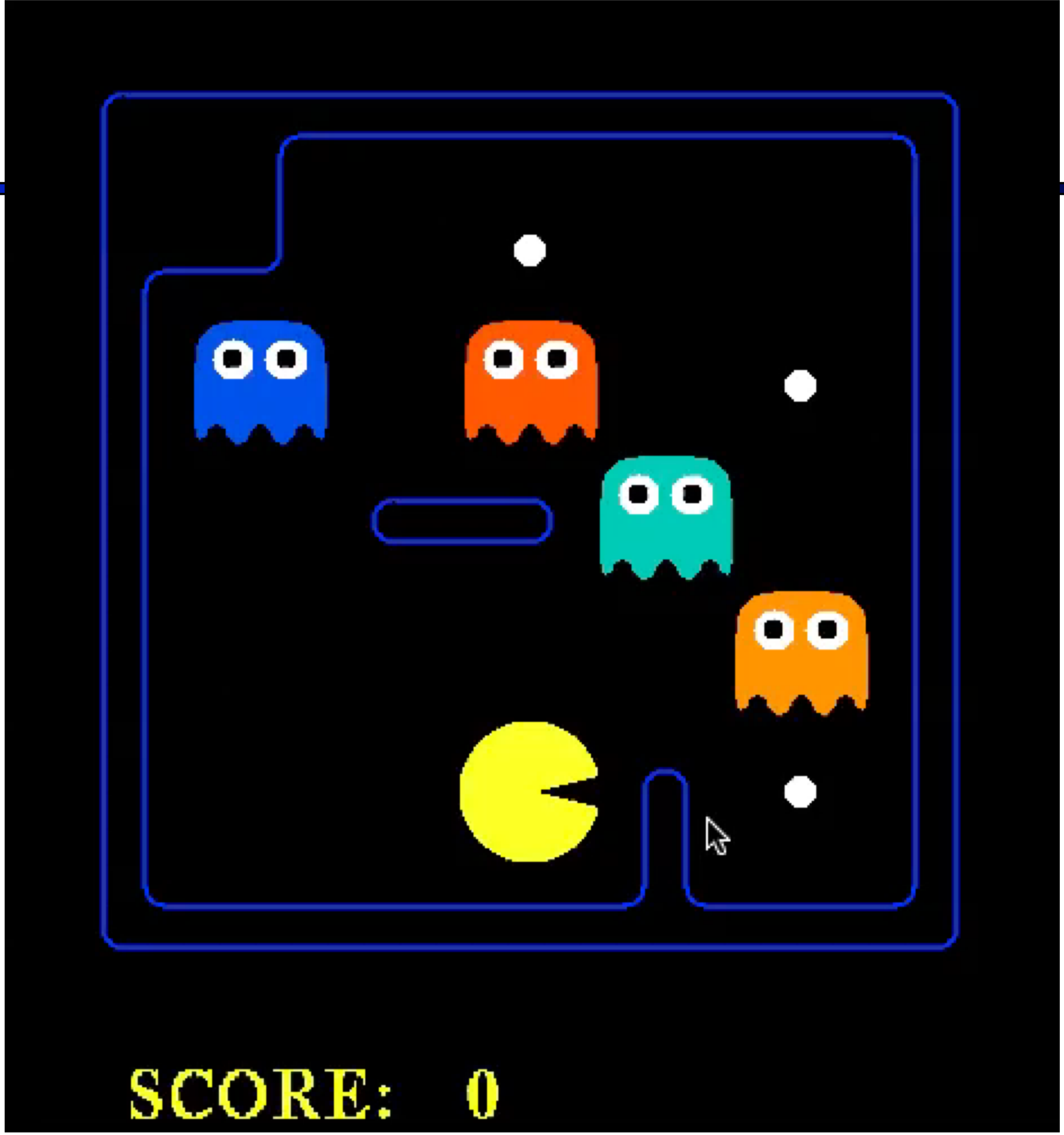
- Often, dead reckoning won't work in the real world
  - E.g., sensors just count the ***number*** of adjacent walls (0,1,2,3 = 2 bits)
- Pacman doesn't know which actions work, so he's "lost"
  - So if he doesn't know where he is, how does he build a map???
- Initialize the KB with **PacPhysics** for *T* time steps, starting at 0,0
- Run the Pacman agent for *T* time steps
  - At each time step
    - Update the KB with previous action and new percept facts
    - For each x,y, add either Wall_x,y or ¬Wall_x,y to KB, if entailed
    - For each x,y, add either At_x,y_t or ¬At_x,y_t to KB, if entailed
    - Choose an action

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
  - solution obtained from truth values of action variables
- For $T = 1$ to ∞,
  - Initialize the KB with **PacPhysics** for T time steps
  - Assert goal is true at time T
- Read off action variables from SAT-solver solution


SCORE: 0


SCORE: 0


SCORE: 0

Summary

- Logical inference computes entailment relations among sentences
- Theorem provers apply inference rules to sentences
  - Forward chaining applies modus ponens with definite clauses; linear time
  - Resolution is complete for PL but exponential time in the worst case
- SAT solvers based on DPLL provide incredibly efficient inference
- Logical agents can do localization, mapping, SLAM, planning (and many other things) just using one generic inference algorithm on one knowledge base