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***

Conjunctive normal form (CNF)

- Every sentence can be expr… …lauses
- Each clause is a ***disjunction*** o…
- Each literal is a symbol or a …ymb…
- Conversion to CNF by a se…que…e of standard …ormations:
  - At_1,1_0 ⇒ (Wall_0,1 ⇔ Bl…cked_W_0)
  - At_1,1_0 ⇒ ((Wall_0,1 ⇒ Blocked_W_0) ⋀ (…ocked_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)

Replace biconditional by two implications

Replace α ⇒ β by ¬α v β

Distribute v over ⋀

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 ∨ B) ∧ (A ∨ ¬C) is satisfied by {A=true}
    - any clause is falsified; e.g., (A ∨ B) ∧ (A ∨ ¬C) is falsified 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 ∨ B) ∧ (A ∨ ¬C) ∧ (C ∨ ¬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 ∨ B) ∧ (A ∨ ¬C) becomes (false ∨ B) ∧ (false ∨ ¬C), i.e. (B) ∧ (¬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}))
```

Efficiency

- Naïve implementation of DPLL: solve ~100 variables
- Extras:
  - Smart variable and value ordering
    - E.g., choose values first for variables that appear in lots of clauses
  - Divide and conquer
    - If any set of clauses shares no variables with the other clauses, solve it separately
  - Caching unsolvable subcases as extra clauses to avoid redoing them
  - Cool indexing and incremental recomputation tricks so that every step of the DPLL algorithm is efficient (typically O(1))
    - Index of clauses in which each variable appears +ve/-ve
    - Keep track number of satisfied clauses, update when variables assigned
    - Keep track of number of remaining literals in each clause
- Real implementation of DPLL: solve ~100000000 variables

SAT solvers in practice

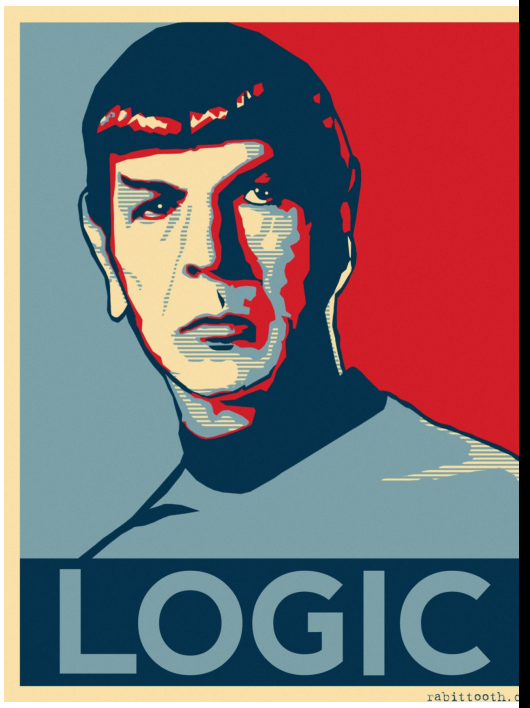
- Circuit verification: does this VLSI circuit compute the right answer?
- Software verification: does this program compute the right answer?
- Software synthesis: what program computes the right answer?
- Protocol verification: can this security protocol be broken?
- Protocol synthesis: what protocol is secure for this task?
- Lots of combinatorial problems: what is the solution?
- Planning: ***how can I eat all the dots???***

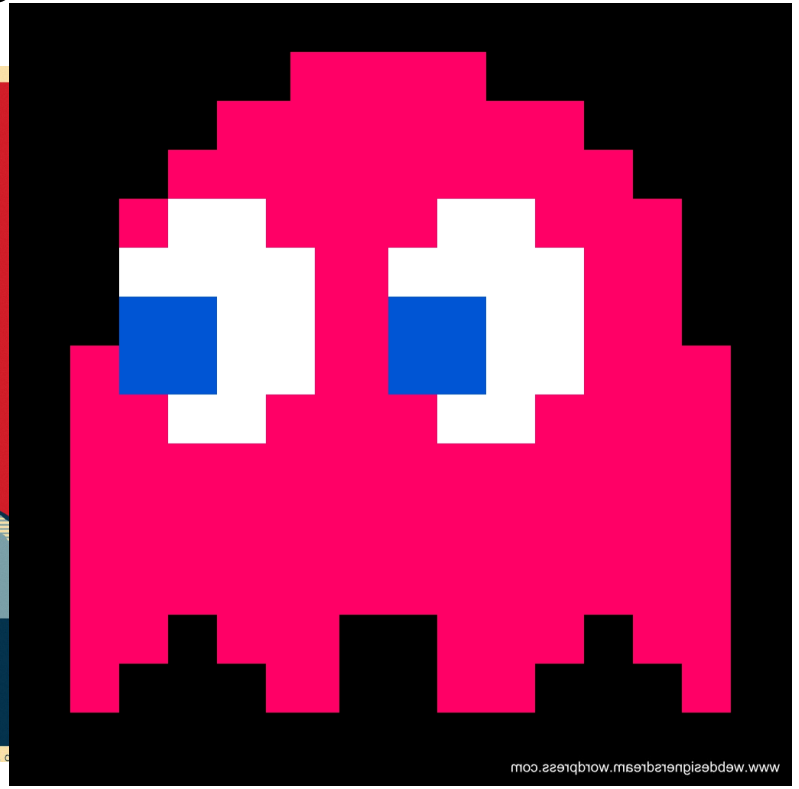
Summary

- Inference in propositional logic:
  - Inference algorithms determine whether $\alpha \models \beta$
    - Theorem provers apply inference rules to construct proofs
    - Model checkers enumerate models to establish entailment directly
  - Forward chaining is sound, complete, and linear-time for definite clauses
  - DPLL enumerates possible models via recursive depth-first search
  - Even though propositional logic KBs are often very large, modern SAT solvers (usually based on DPLL) are usually very efficient in practice

CS 188: Artificial Intelligence

Logical Agents





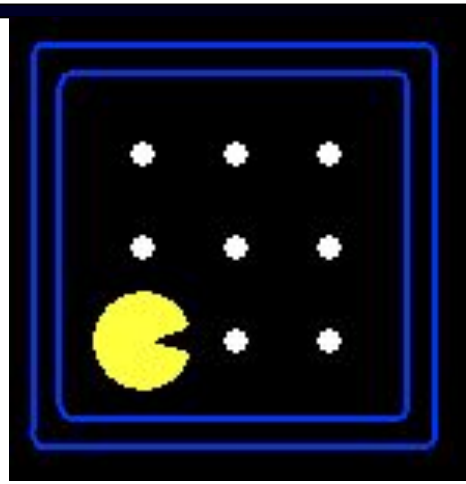
Instructor: Stuart Russell

University of California, Berkeley

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

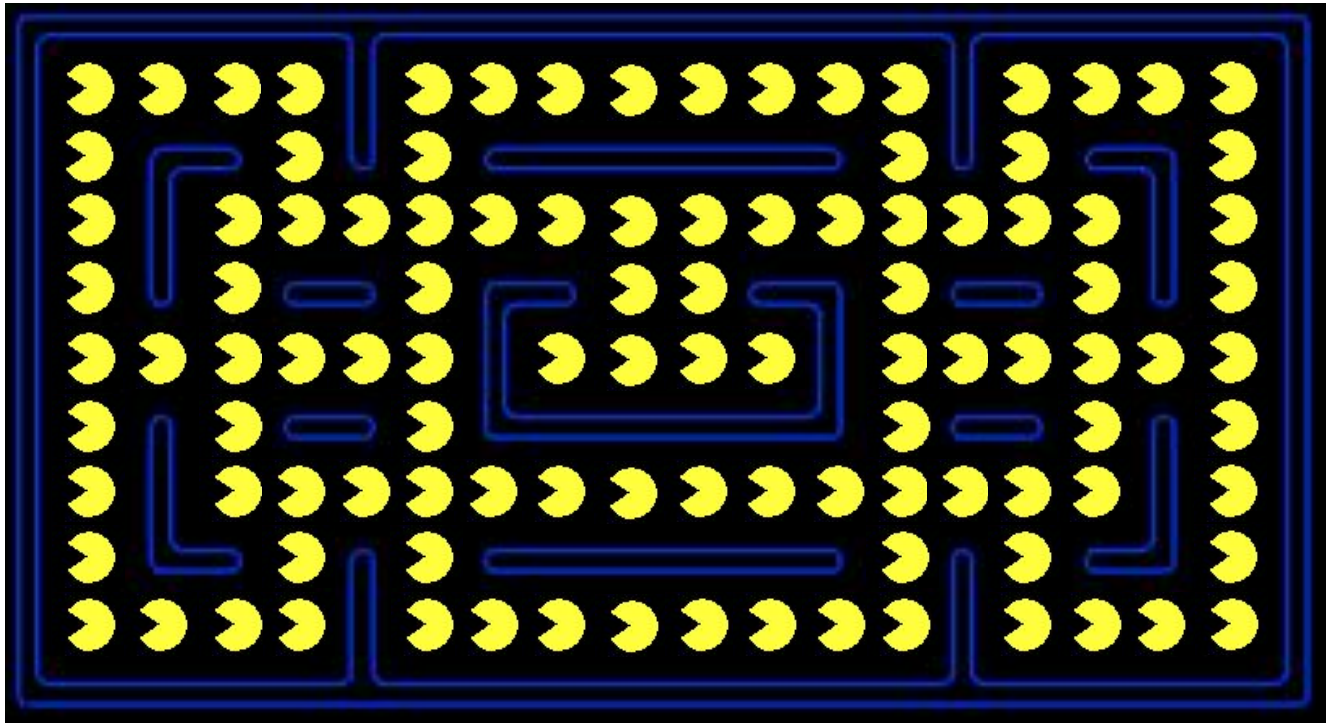
- ***State estimation*** means keeping track of what's true now
- A logical agent can just ask itself!
  - E.g., ask whether KB ⋀ <actions> ⋀ <percepts> |= At_2,2_6
- This is "lazy": it analyzes one's whole life history at each step!
- A more "eager" form of state estimation:
  - After each action and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

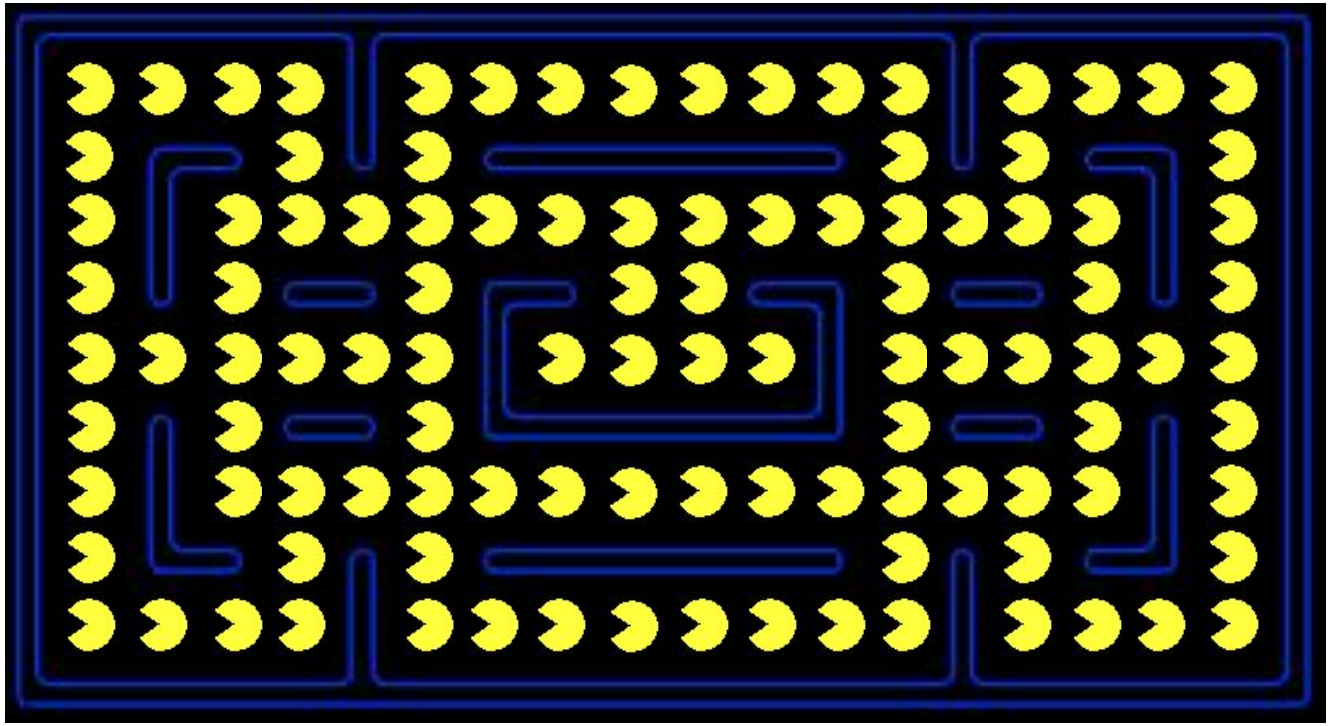
Localization demo

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



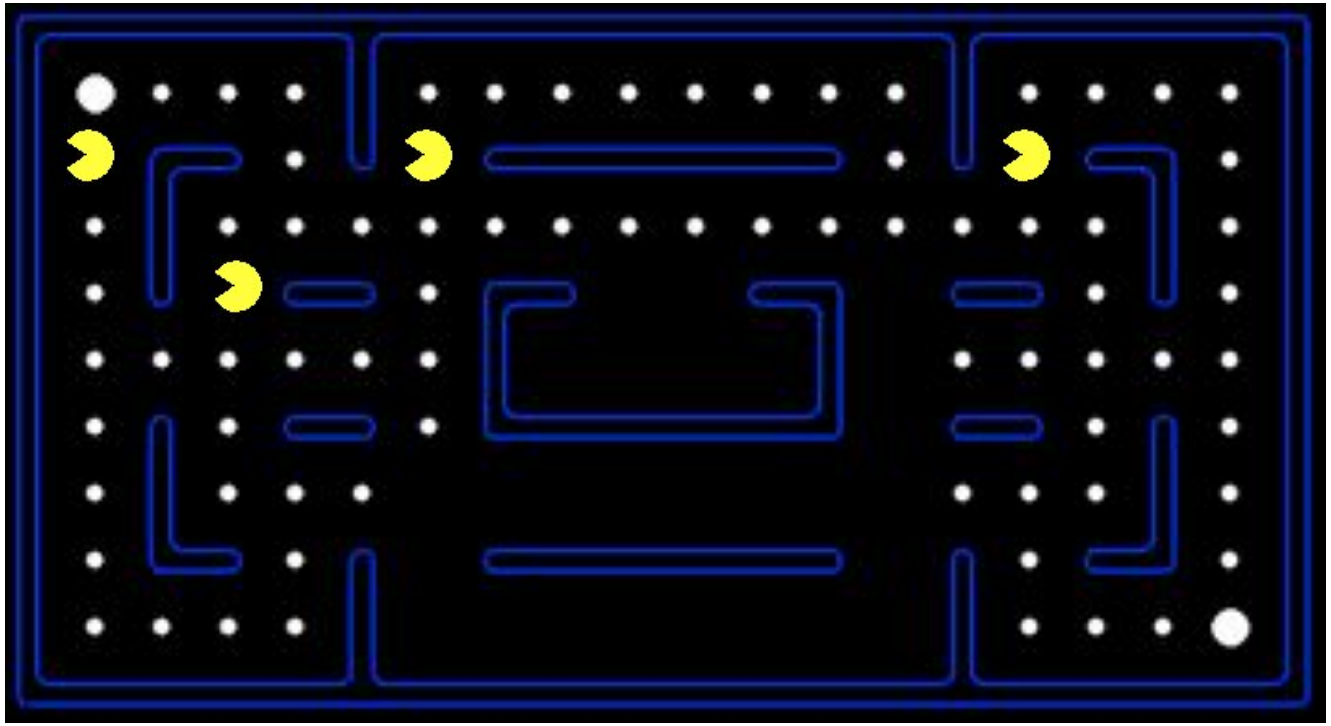
Localization demo

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



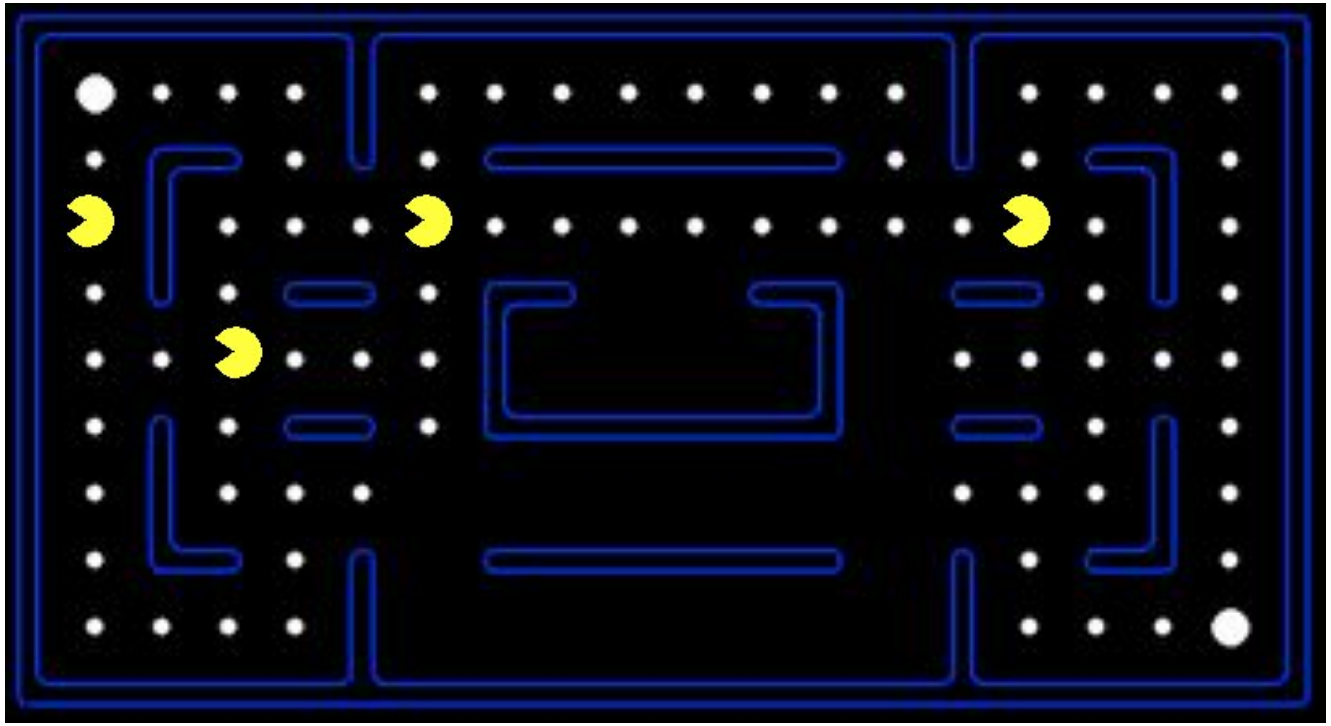
Localization demo

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



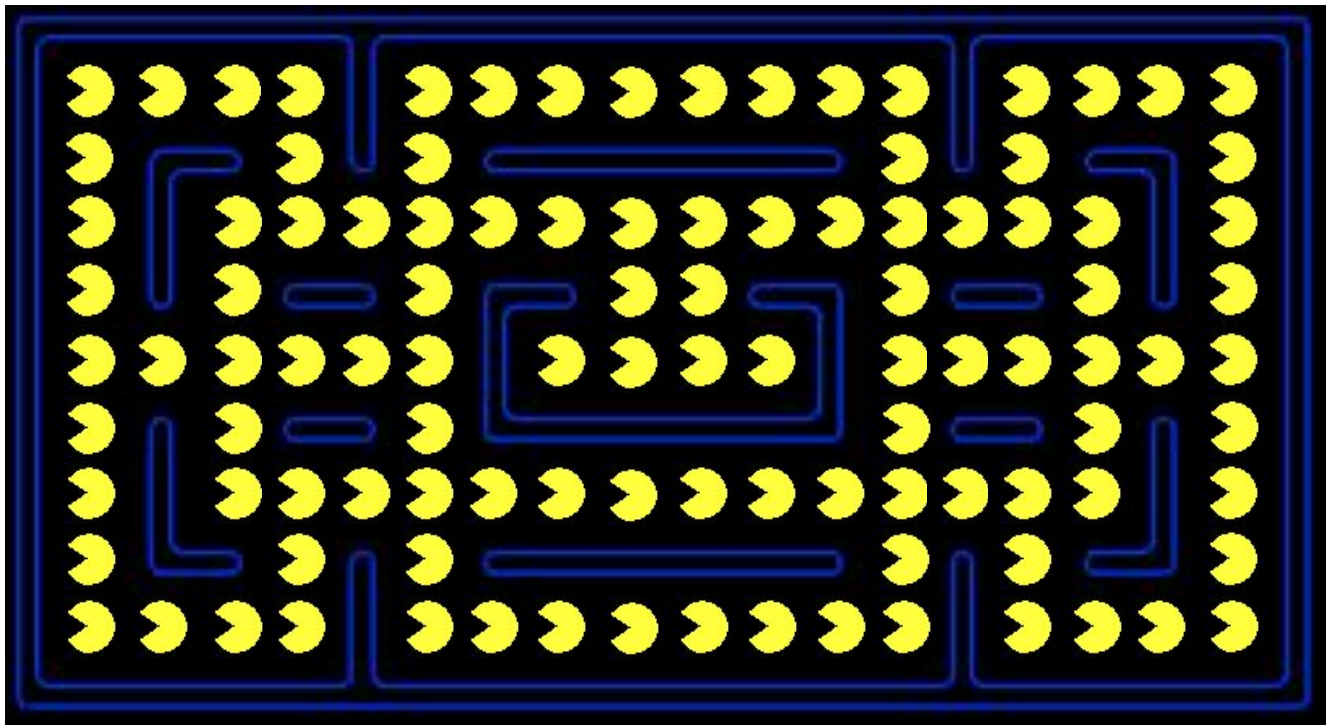
Localization demo

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



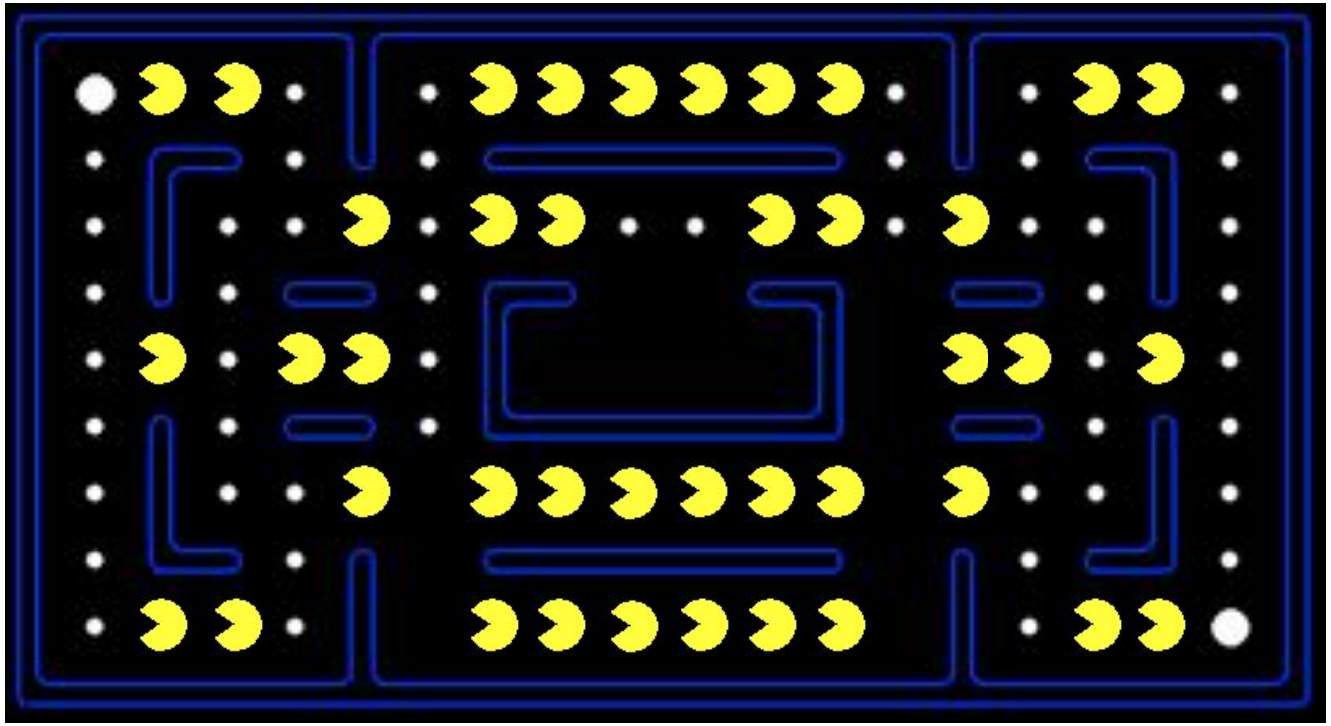
Localization demo

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



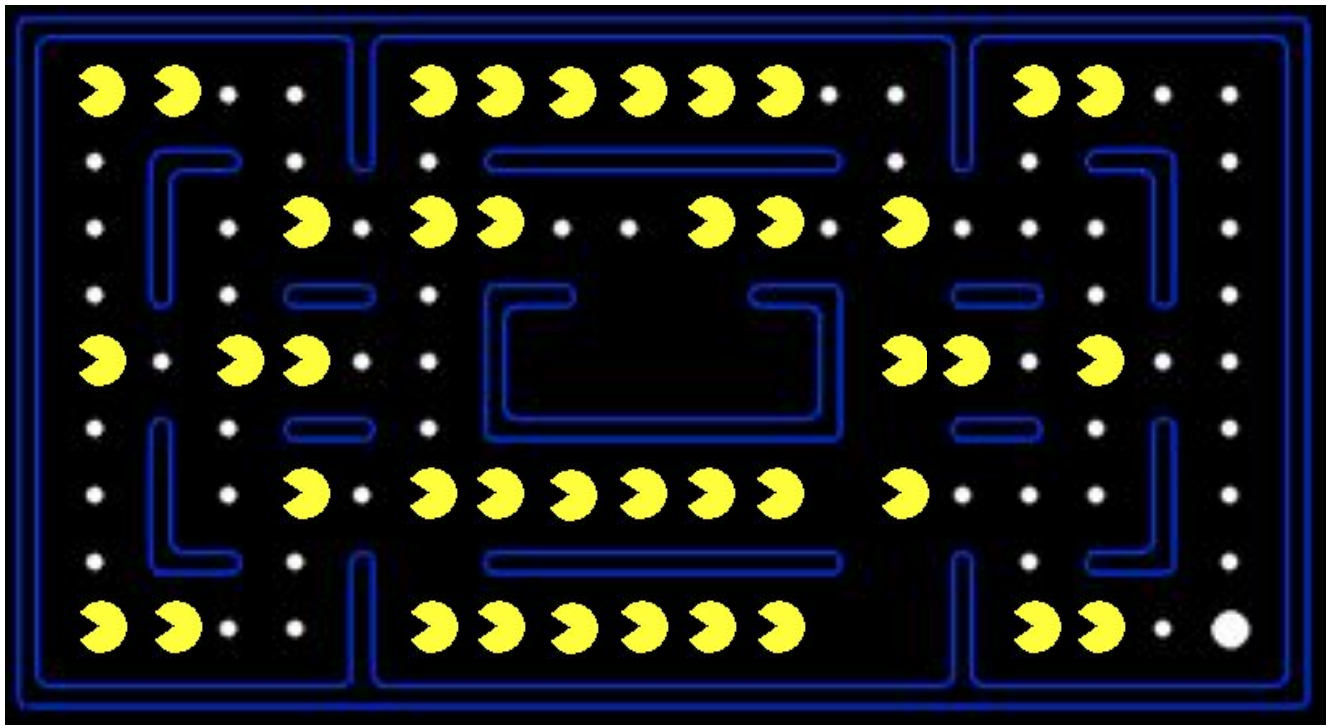
Localization demo

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



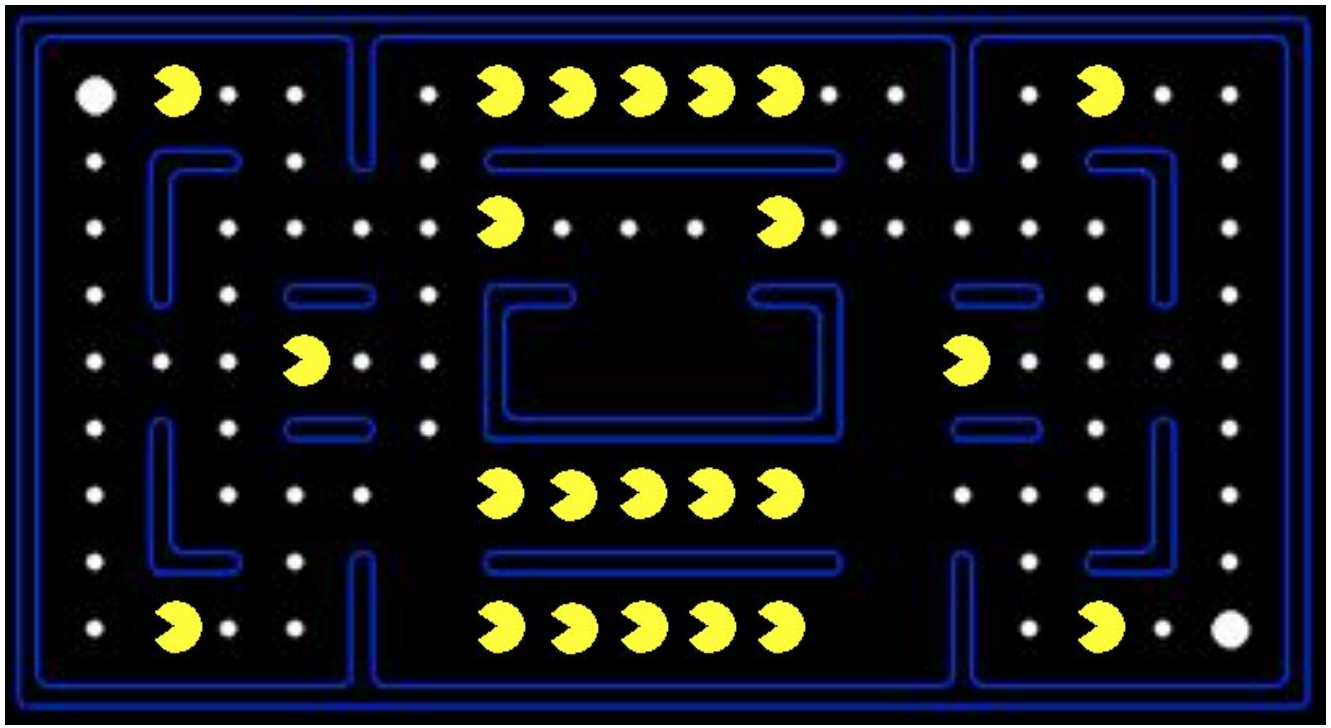
Localization demo

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



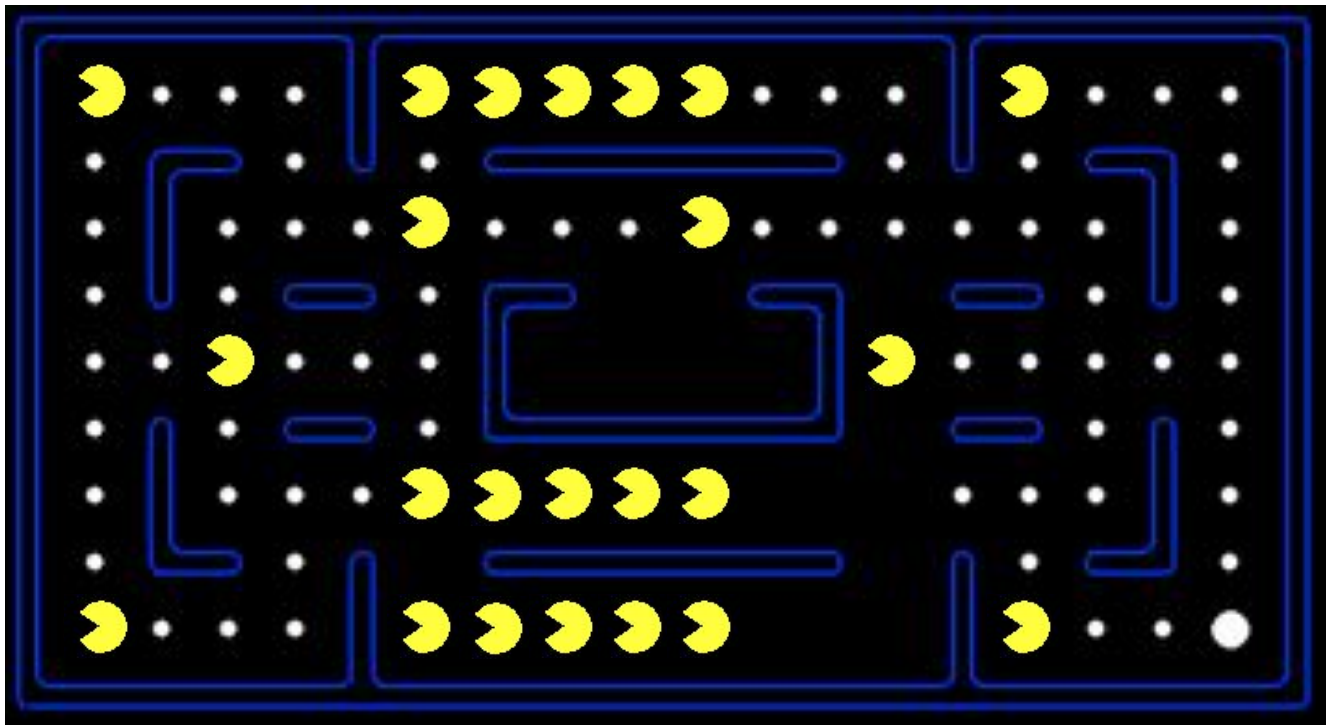
Localization demo

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



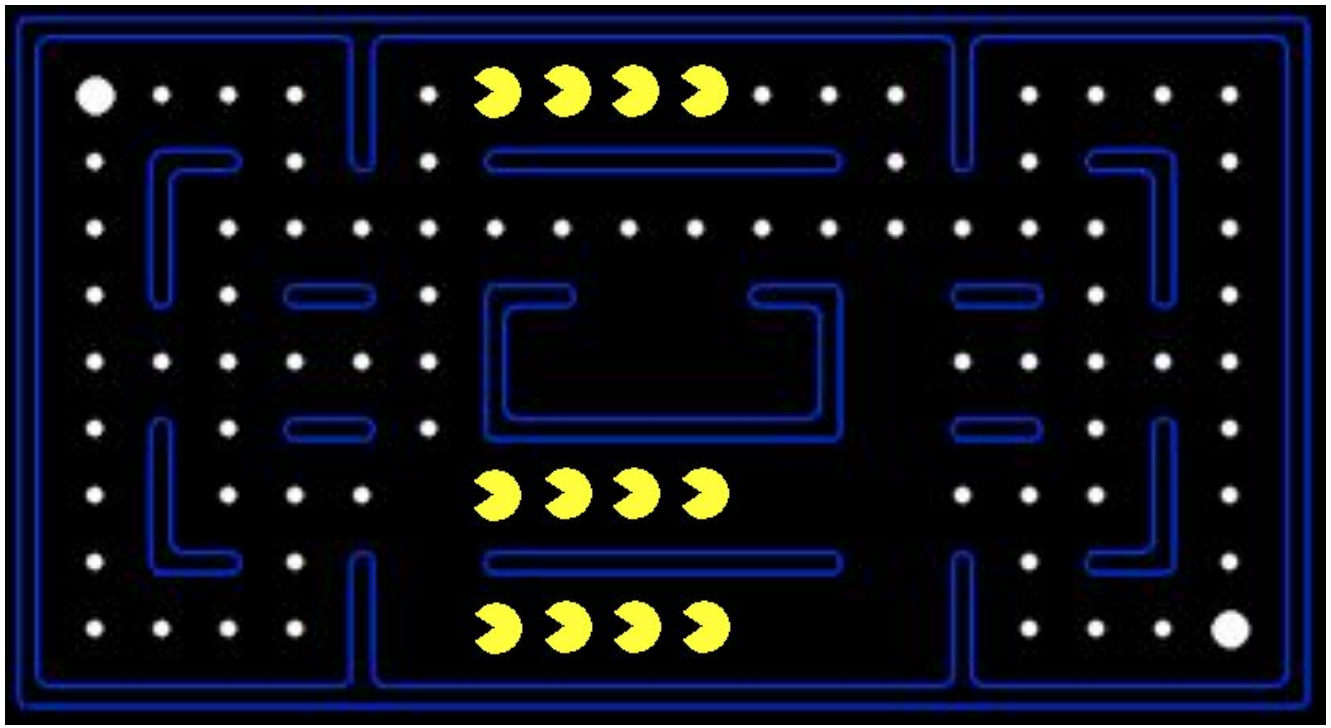
Localization demo

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



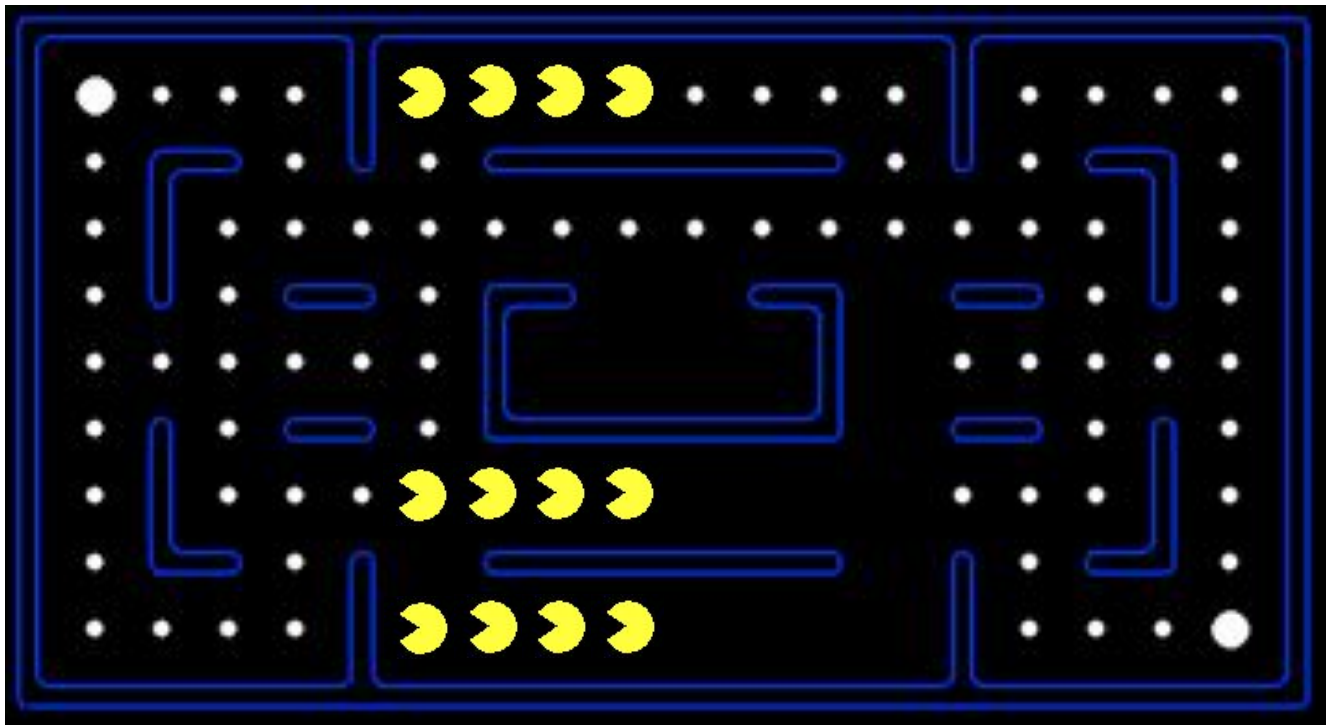
Localization demo

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



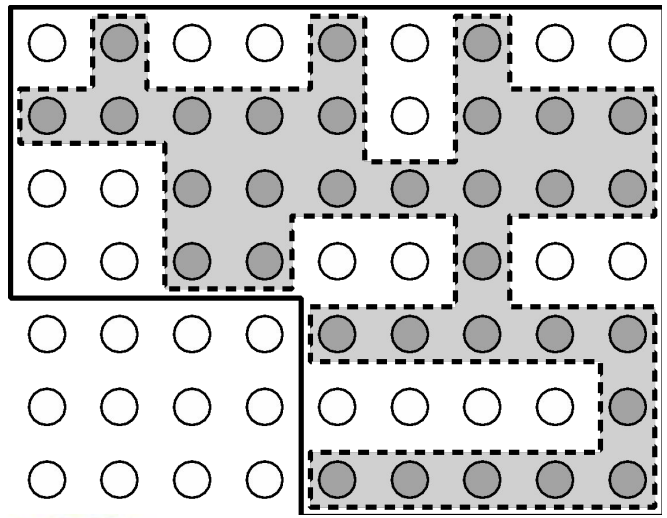
Localization demo

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



State estimation contd.

- Is the eager method enough for accurate state estimation?
  - No! There can be cases where neither X_t nor $\neg X_t$ is entailed, and neither Y_t nor $\neg Y_t$ is entailed, but some constraint, e.g., $X_t \vee Y_t$, ***is*** entailed
- Exact state estimation is intractable in general



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

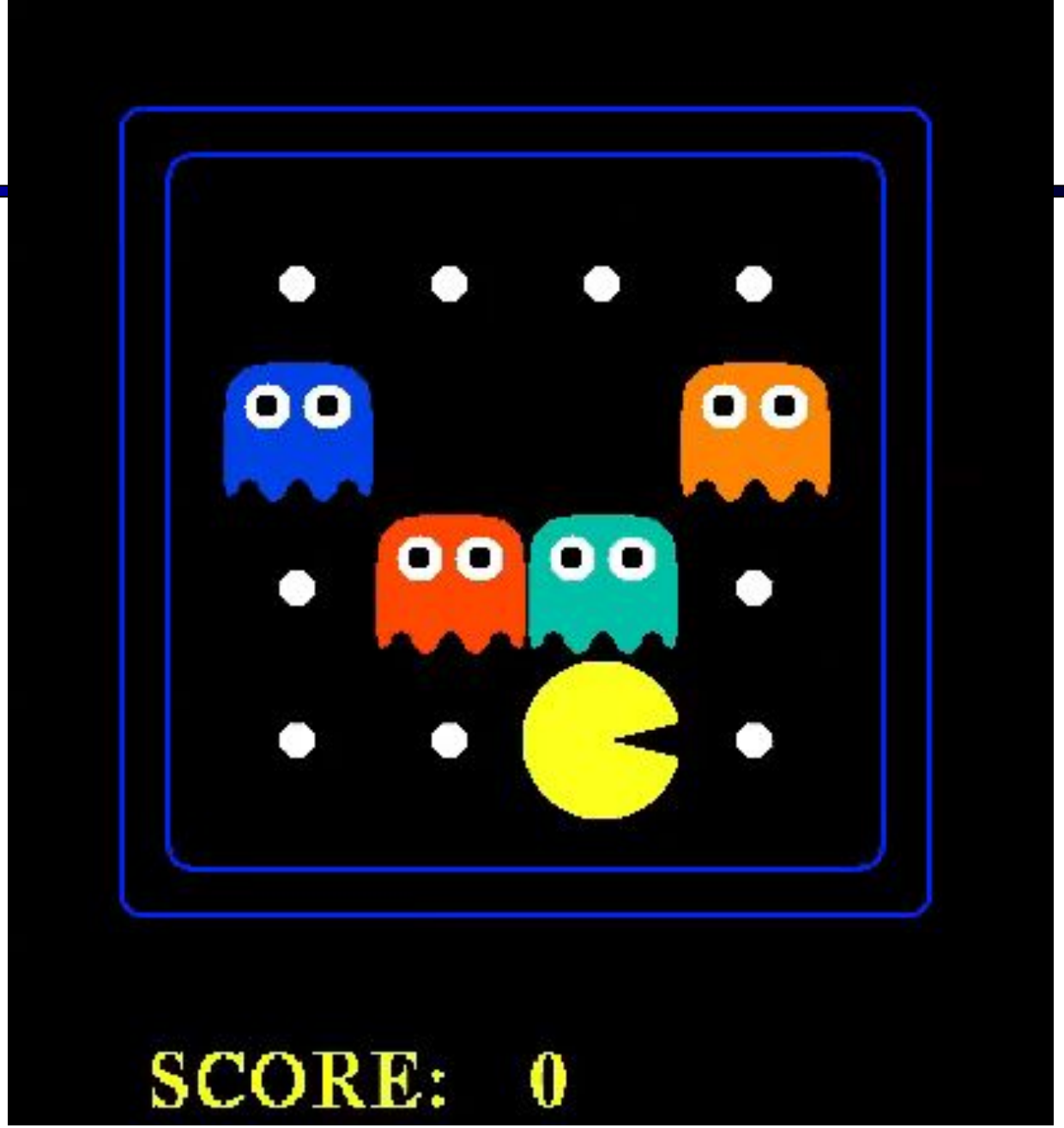
- Percept 
- Action *NORTH*
- Percept 
- Action *EAST*
- Percept 
- Action *SOUTH*
- Percept 

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 ∞, 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
- Read off action variables from solution


SCORE: 0


SCORE: 0


SCORE: 0

Summary

- One possible agent architecture: knowledge + inference
- Logics provide a formal way to encode knowledge
  - A logic is defined by: syntax, set of possible worlds, truth condition
- Logical inference computes entailment relations among sentences
- 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