Announcements

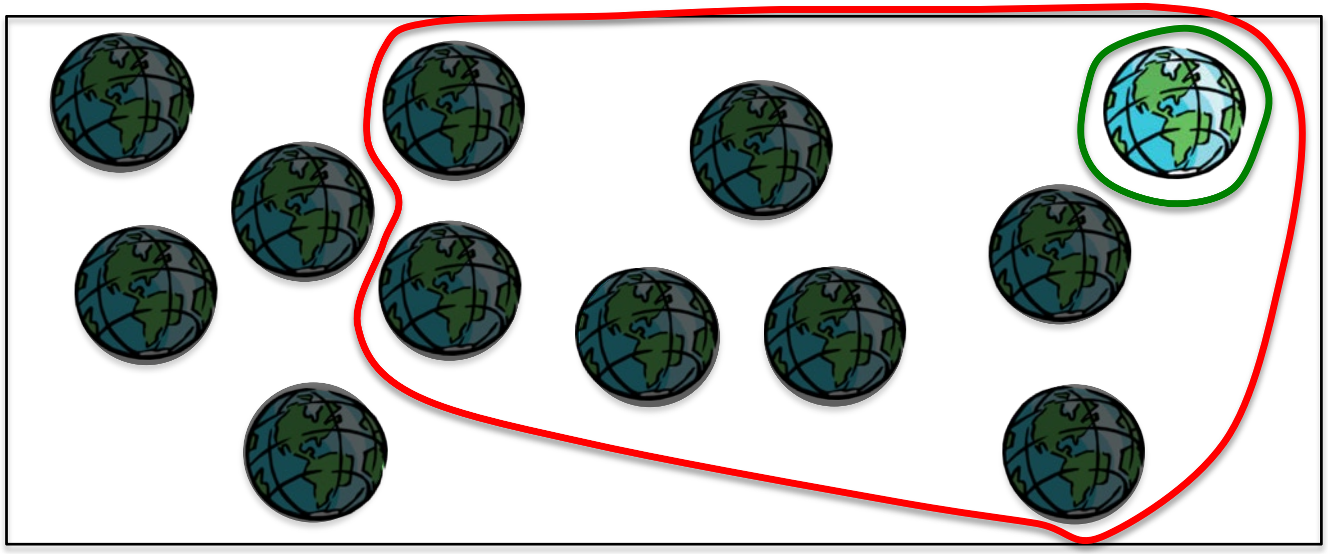
- HW
- Project

Inference: entailment

- ***Entailment***: $\alpha \models \beta$ ("α entails β" or "β follows from α") iff <u>in every world where α is true, β is also true</u>
  - I.e., the α-worlds are a ***subset*** of the β-worlds [***models***$(\alpha) \subseteq$ ***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 $\neg Q$)

α_1

α_2



Inference: proofs

- A proof is a ***demonstration*** of entailment between α and β
- ***Sound*** algorithm: everything it claims to prove is in fact entailed
- ***Complete*** algorithm: every that is entailed can be proved

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); 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 and $(P \Rightarrow Q)$, infer Q by ***Modus Ponens***

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!

Propositional logic semantics

- Let m be a model assigning **true** or **false** to $\{X_1, X_2, \ldots, X_n\}$
- If α is a symbol then its truth value is given in m
- $\neg\alpha$ is true in m iff α is false in m
- $\alpha \wedge \beta$ is true in m iff α is true in m <u>and</u> β is true in m
- $\alpha \vee \beta$ is true in m iff α is true in m <u>or</u> β is true in m
- $\alpha \Rightarrow \beta$ is true in m iff α is false in m <u>or</u> β is true in m
- $\alpha \Leftrightarrow \beta$ is true in m iff $\alpha \Rightarrow \beta$ is true in m <u>and</u> $\beta \Rightarrow \alpha$ is true in m

Example

- Let *m* be {A=true, B=true, C=false, D=false}
- Let α be the sentence $(A \wedge B) \vee (C \wedge \neg D)$
- α is true in *m* iff $(A \wedge B)$ is true in *m* <u>or</u> $(C \wedge \neg D)$ is true in *m*
  - $(A \wedge B)$ is true in *m* iff A is true in *m* <u>and</u> B is true in *m*
  - $(A \wedge B)$ is true in *m*
- α is true in *m*

Example

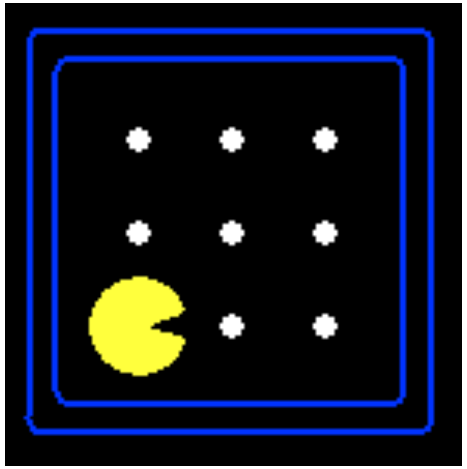
- Let α be the sentence $(A \land B) \lor (C \land \neg D)$
- In how many models is α true?

The plan

- Tell the logical agent what we know about PacPhysics
- Ask it what actions have to be true for the goal to be achieved

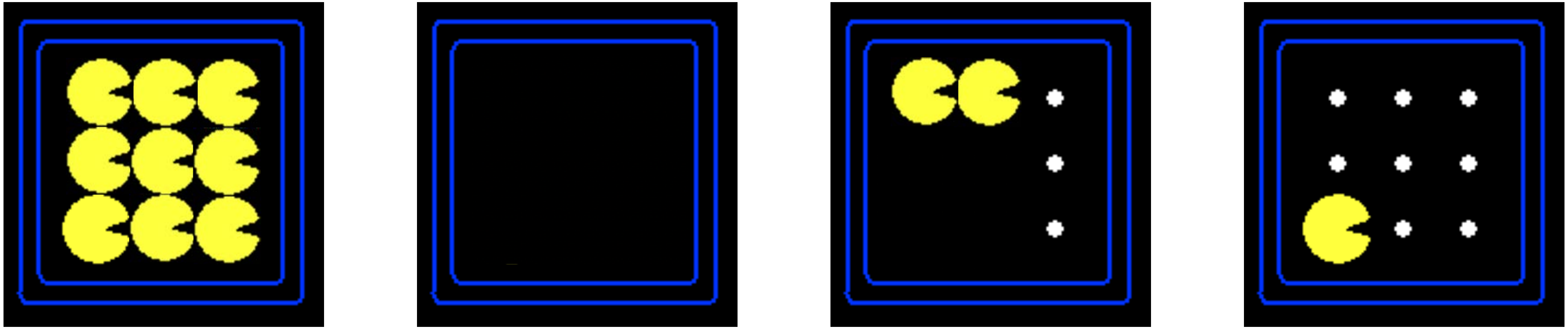
Partially observable Pacman

- 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)



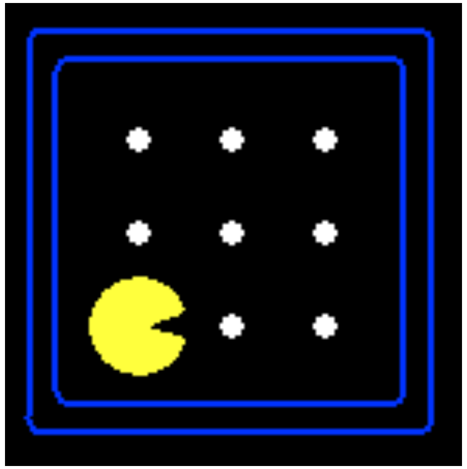
How many possible worlds?

- 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!



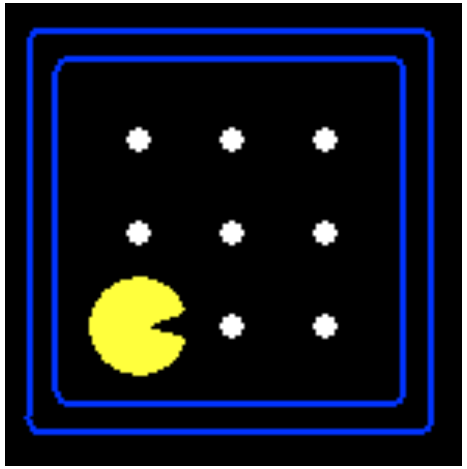
Pacman's knowledge base: Map

- 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 ∧ …



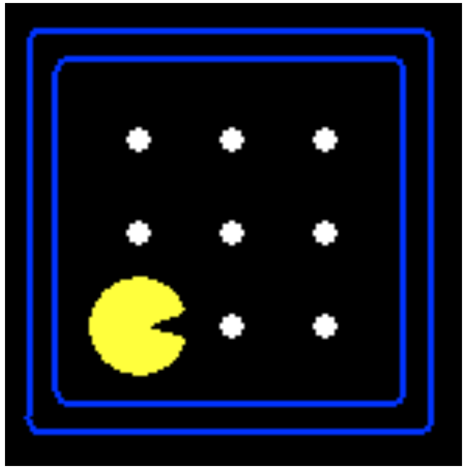
Pacman's knowledge base: Initial state

- 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 ∨ …
- And he knows he's not in more than one place!
  - ¬ (At_1,1_0 ∧ At_1,2_0) ∧ ¬ (At_1,1_0 ∧ At_1,3_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***
  - $X\_t \Leftrightarrow [X\_t\text{-}1 \wedge \neg(\text{some action\_t-1 made it false})] \vee$
    $[\neg X\_t\text{-}1 \wedge (\text{some action\_t-1 made it true})]$
- For Pacman location:
  - $At\_3,3\_17 \Leftrightarrow [At\_3,3\_16 \wedge \neg((\neg Wall\_3,4 \wedge N\_16) \vee (\neg Wall\_4,3 \wedge E\_16) \vee \ldots)]$
    $\vee\ [\neg At\_3,3\_16 \wedge ((At\_3,2\_16 \wedge \neg Wall\_3,3 \wedge N\_16) \vee$
    $(At\_2,3\_16 \wedge \neg Wall\_3,3 \wedge N\_16) \vee \ldots)]$

How many sentences?

- Vast majority of KB occupied by O(NT) transition model sentences
  - Each about 10 lines of text
  - N=200, T=400 => ~800,000 lines of text, or 20,000 pages
- This is because propositional logic has limited expressive power
- Are we really going to write 20,000 pages of logic sentences???
- No, but your code will generate all those sentences!
- (In first-order logic, we need O(1) transition model sentences)
- (State-space search uses atomic states: how do we keep the transition model representation small???)

Some reasoning tasks

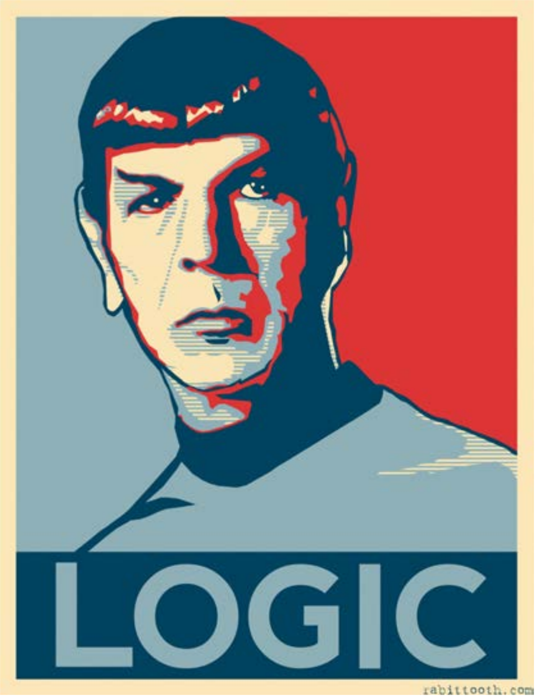
- ***Localization*** with a map and local sensing:
  - Given an initial KB, plus a sequence of percepts and actions, where am I?
- ***Mapping*** with a location sensor:
  - Given an initial KB, plus a sequence of percepts and actions, what is the map?
- ***Simultaneous localization and mapping***:
  - 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!!***

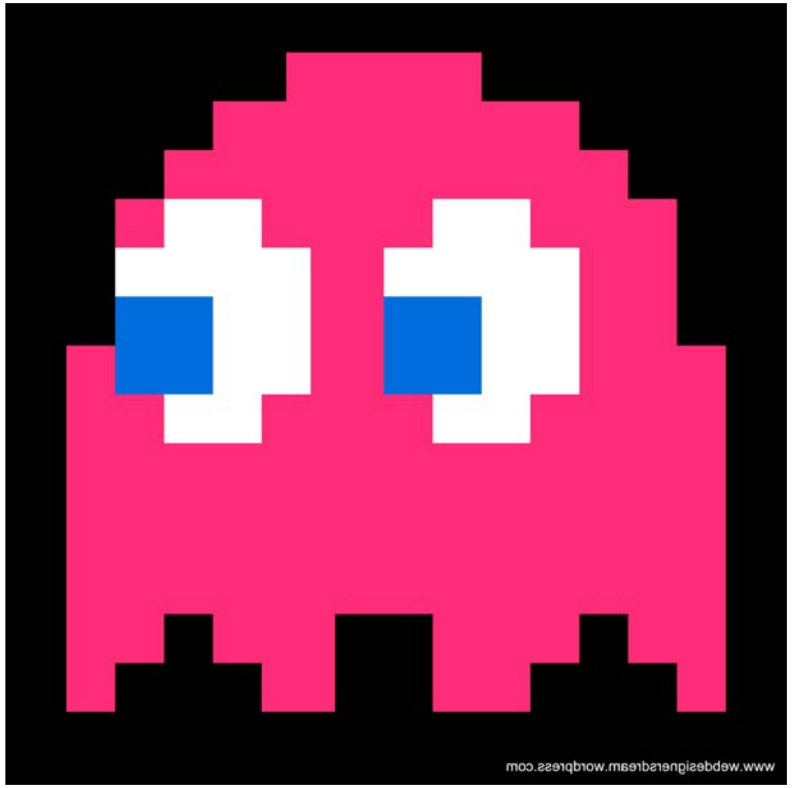
Summary

- 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
- A simple KB for Pacman covers the initial state, sensor model, and transition model
- Logical inference computes entailment relations among sentences, enabling a wide range of tasks to be solved

272SM: Introduction to Artificial Intelligence

Inference in Propositional Logic





Instructors: Tatjana Petrov

University of Trieste, Italy

Inference (reminder)

- Method 1: ***model-checking***
  - For every possible world, if α is true make sure that is β true too
- Method 2: ***theorem-proving***
  - Search for a sequence of proof steps (applications of ***inference rules***) leading from α to β
- ***Sound*** algorithm: everything it claims to prove is in fact entailed
- ***Complete*** algorithm: every that is entailed can be proved

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 sentenc *junction* of ***clauses***
- Each clause is a
- Each liter symbol
- Conversion t y a sequence of standard transformations:
  - $A \Rightarrow (W \Leftarrow$
  - $A \Rightarrow ((W \Rightarrow B) \wedge (B \Rightarrow W))$
  - $\neg A \vee ((\neg W \vee B) \wedge (\neg B \vee W))$
  - $(\neg A \vee \neg W \vee B) \wedge (\neg A \vee \neg B \vee W)$

Replace biconditional by two implications

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

Distribute $\vee$ over $\wedge$

Efficient SAT solvers

- DPLL (Davis-Putnam-Logemann-Loveland) is the core of modern solvers
- Recursive depth-first enumeration of 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 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 $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$ 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 $(false \vee B) \wedge (false \vee \neg C)$, 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, 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}))
```

DPLL: example

$$(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$$

- *model*: $\{\}$
- *symbols*: $[L, M, N, P, Q, R, S]$
- *clauses*: $(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$

… Early termination? Pure symbols? Unit Clause?

Efficiency

- Naïve implementation of DPLL: solve ~100 variables
- Extras:
  - Smart variable and value ordering
  - Divide and conquer
  - 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

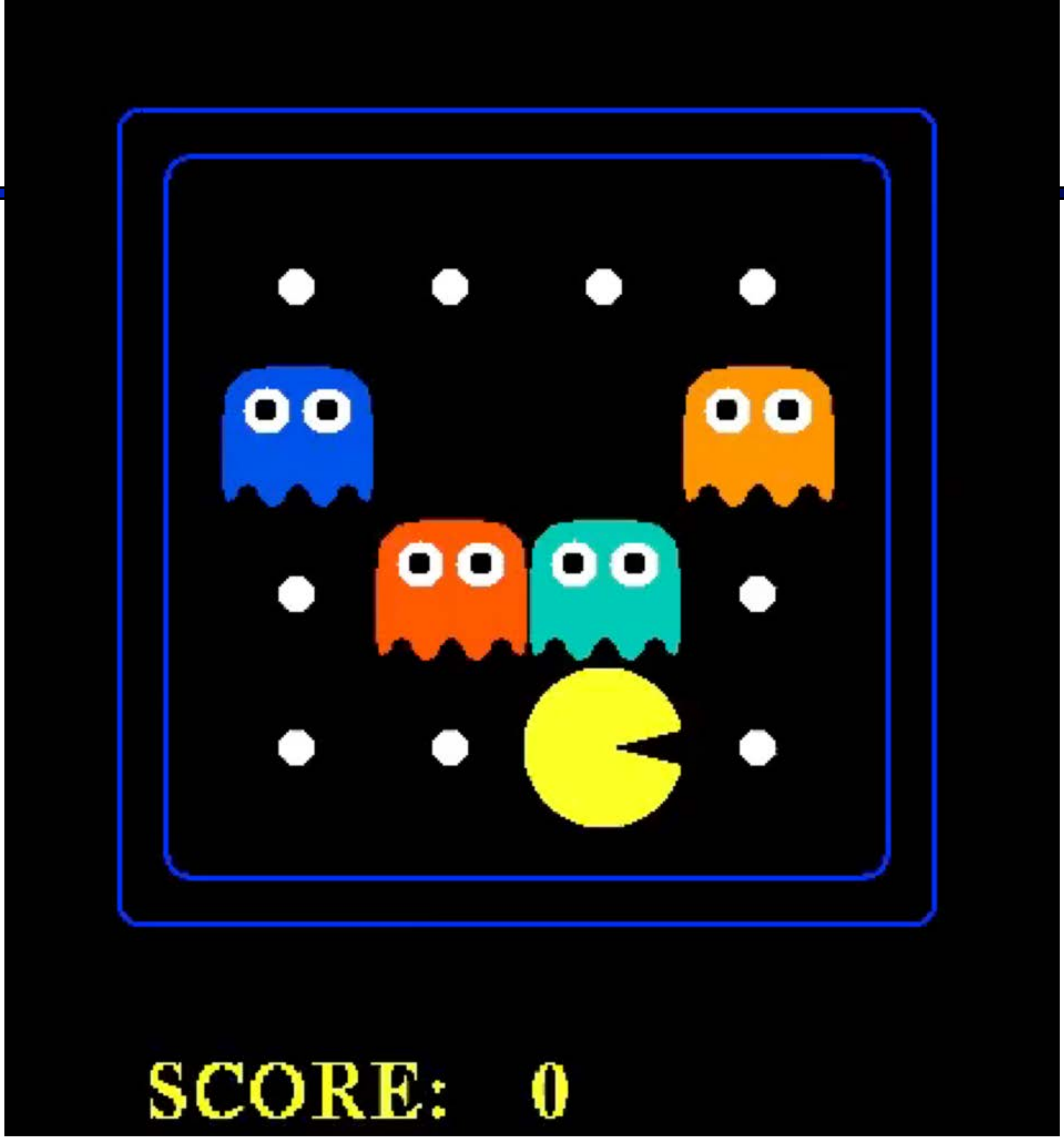
- Circuit verification: does this VLSI circuit compute the right answer?
- Software verification: does this program compute the right answer?
- Software synthesis: what program computes the right answer?
- Protocol verification: can this security protocol be broken?
- Protocol synthesis: what protocol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

Planning as satisfiability

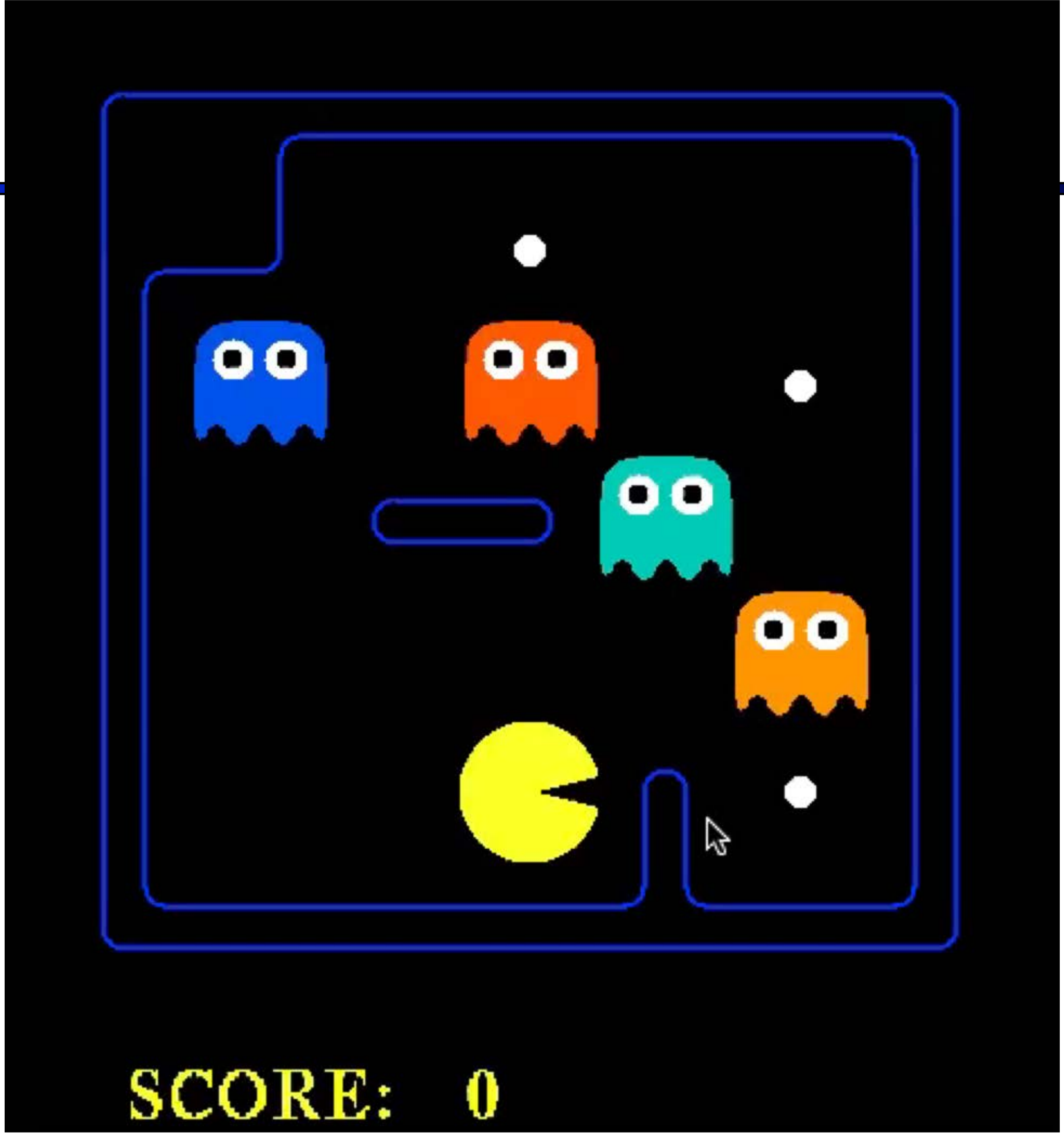
- Given a hyper-efficient SAT solver, can we use it to make plans?
- Yes, for fully observable, deterministic case:
  - planning problem is solvable iff there is some satisfying assignment
  - solution obtained from truth values of action variables
- For $T = 1$ to ∞,
  - Initialize the KB with **PacPhysics** for T time steps
  - Assert goal is true at time T
- Read off action variables from SAT-solver solution

Basic PacPhysics for Planning

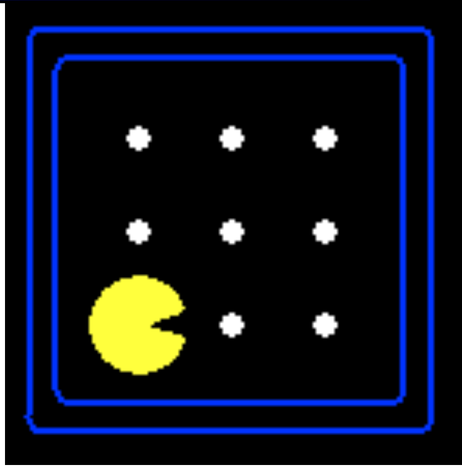
- **Map**: where the walls are and aren't, where the food is and isn't
- **Initial state**: Pacman start location (exactly one place), ghosts
- **Actions**: Pacman does exactly one action at each step
- **Transition model**:
  - <at x,y_t> ⇔ [at x,y_t-1 and stayed put] v [next to x,y_t-1 and moved to x,y]
  - <food x,y_t> ⇔ [food x,y_t-1 and not eaten]
  - <ghost$_B$ x,y_t> ⇔ [.....]


SCORE: 0


SCORE: 0


SCORE: 0

Reminder: Partially observable Pacman



- Perceives wall/no-wall in each direction at each time
- Variables: Blocked_W_0, Blocked_N_0, …, Blocked_W_1, …
- Basic question: where am I?

State estimation

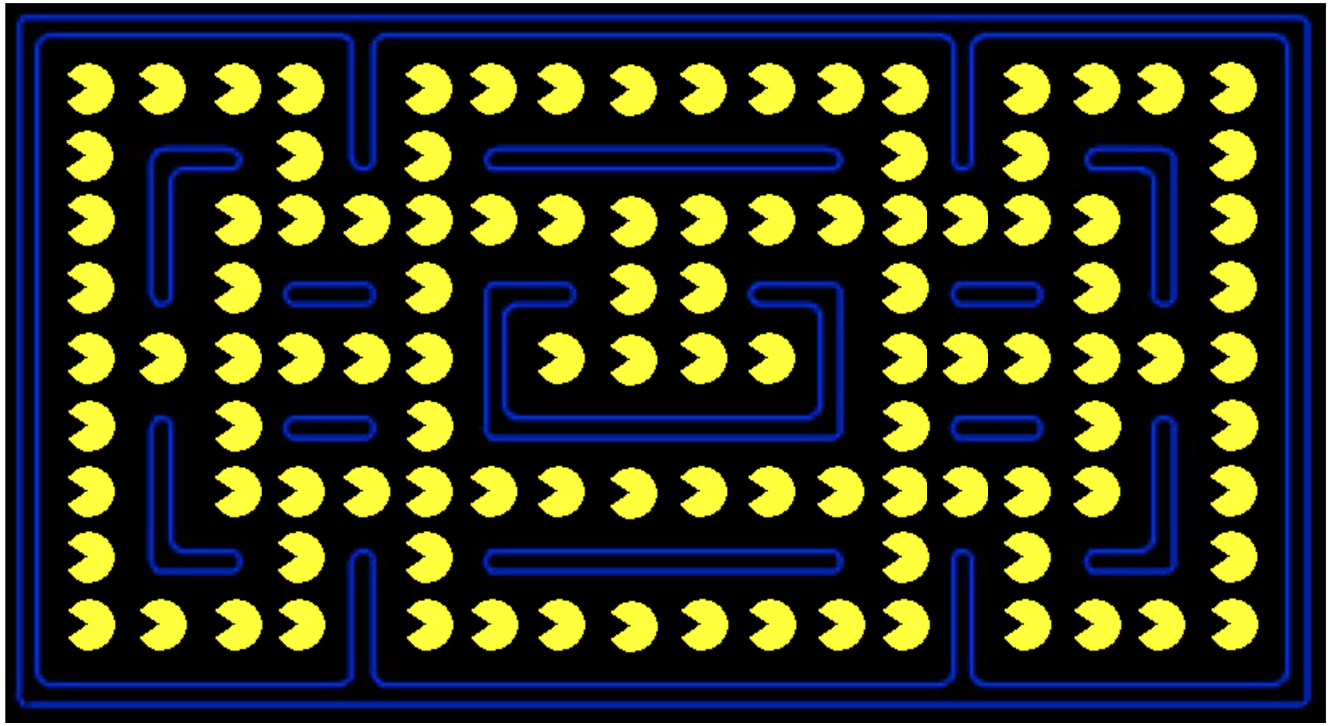
- ***State estimation*** means keeping track of what's true now, given a history of observations and actions
- A logical agent can just ask itself!
  - E.g., ask whether KB ∧ <actions> ∧ <percepts> |= At_2,2_6
- This is "lazy": it analyzes one's whole life history at each step!
- A more "eager" form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

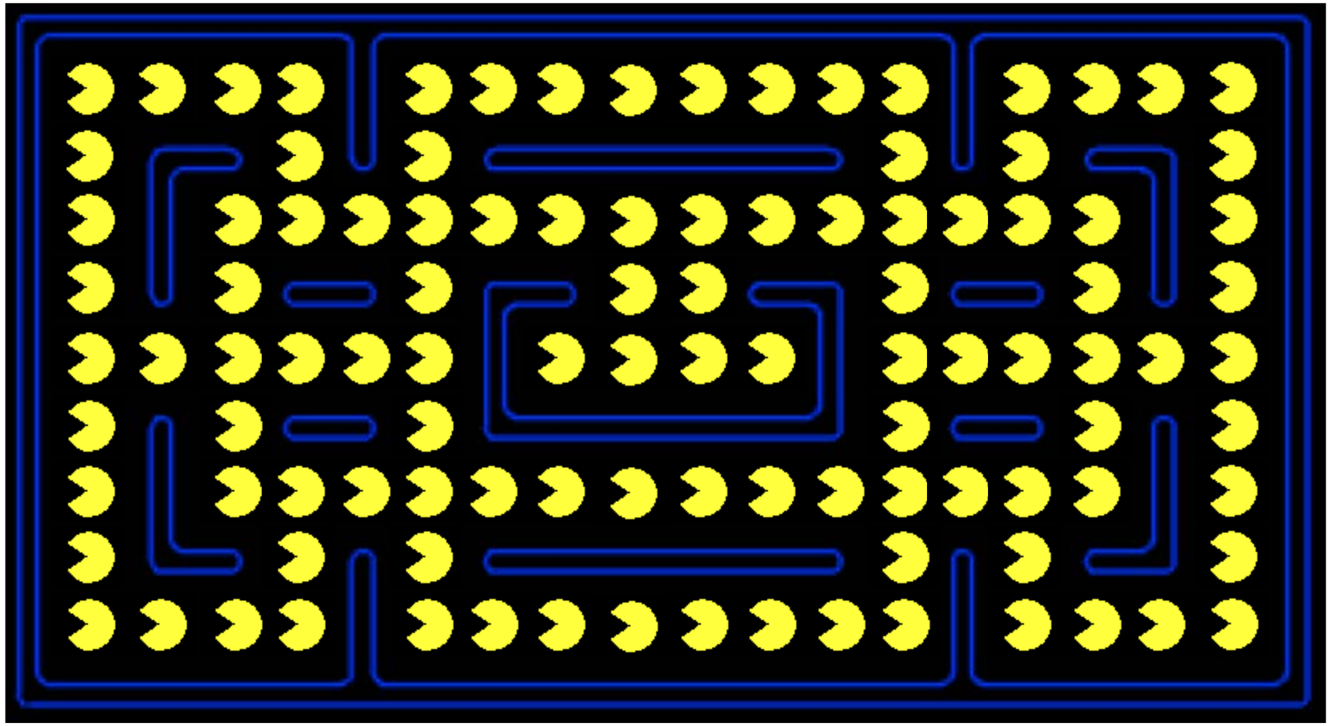
Localization demo

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



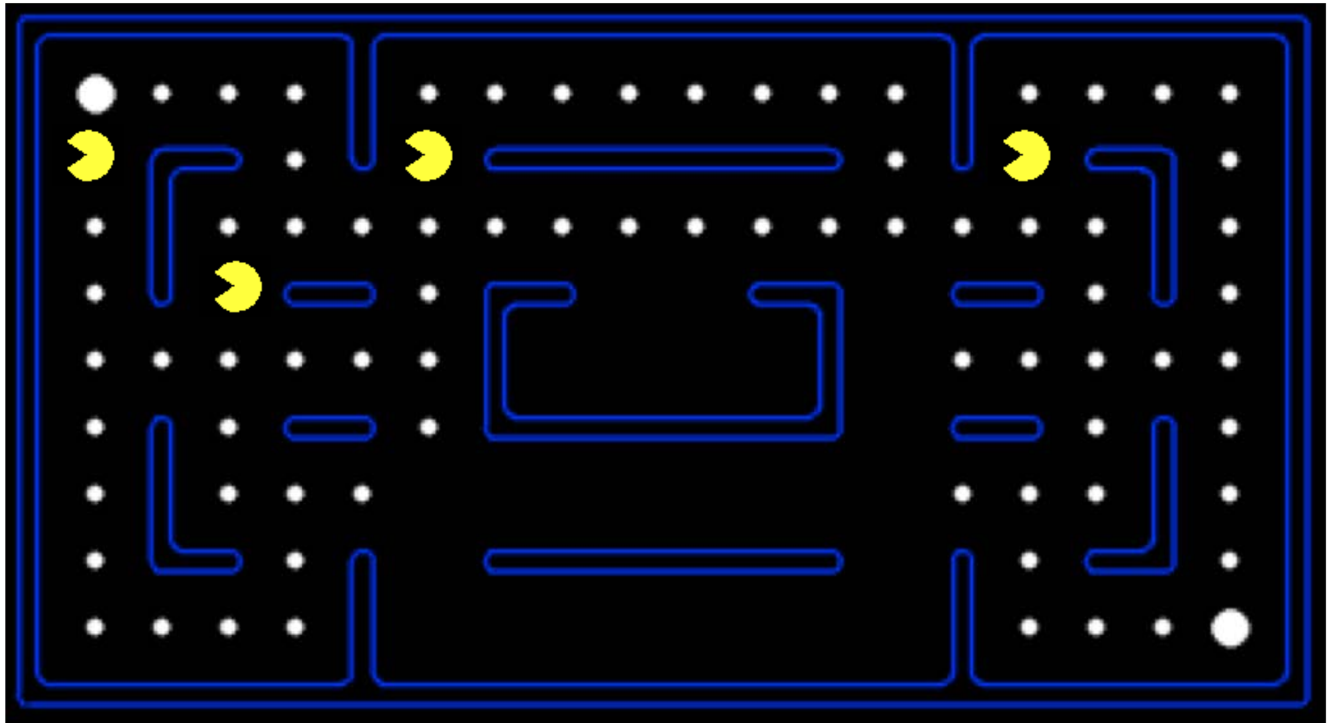
Localization demo

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



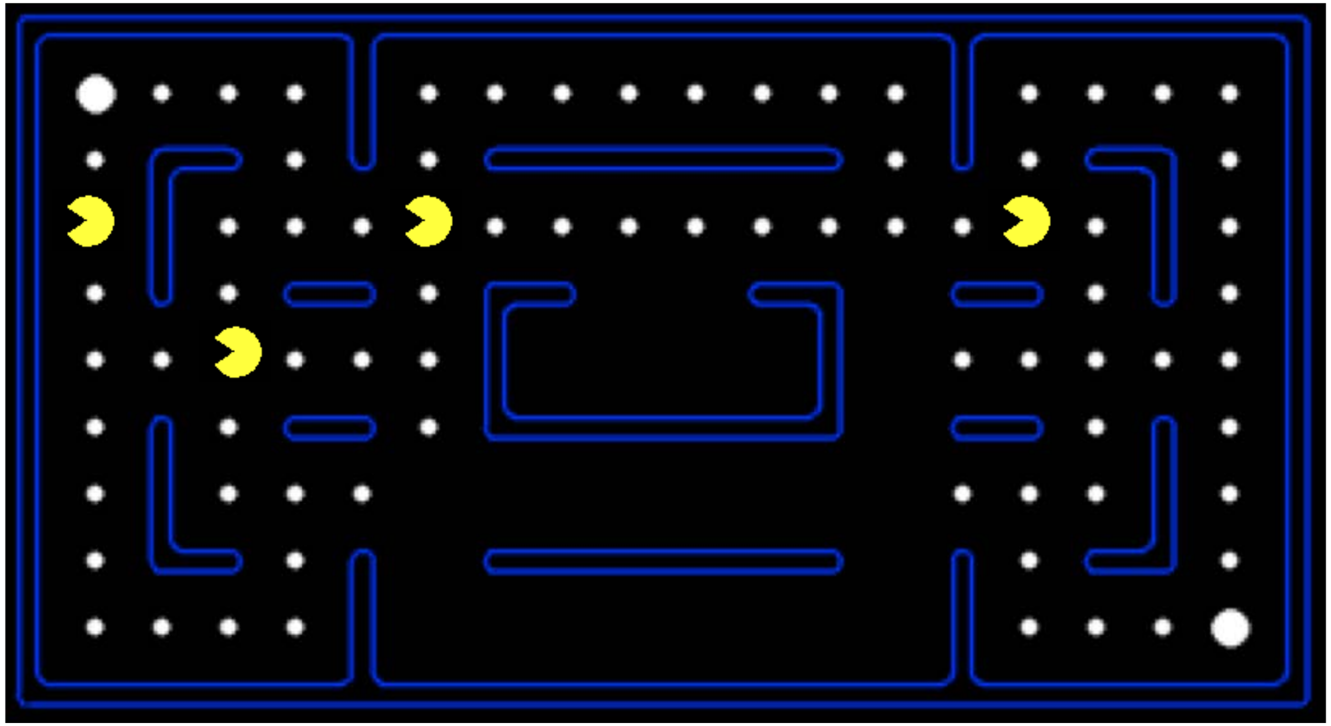
Localization demo

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



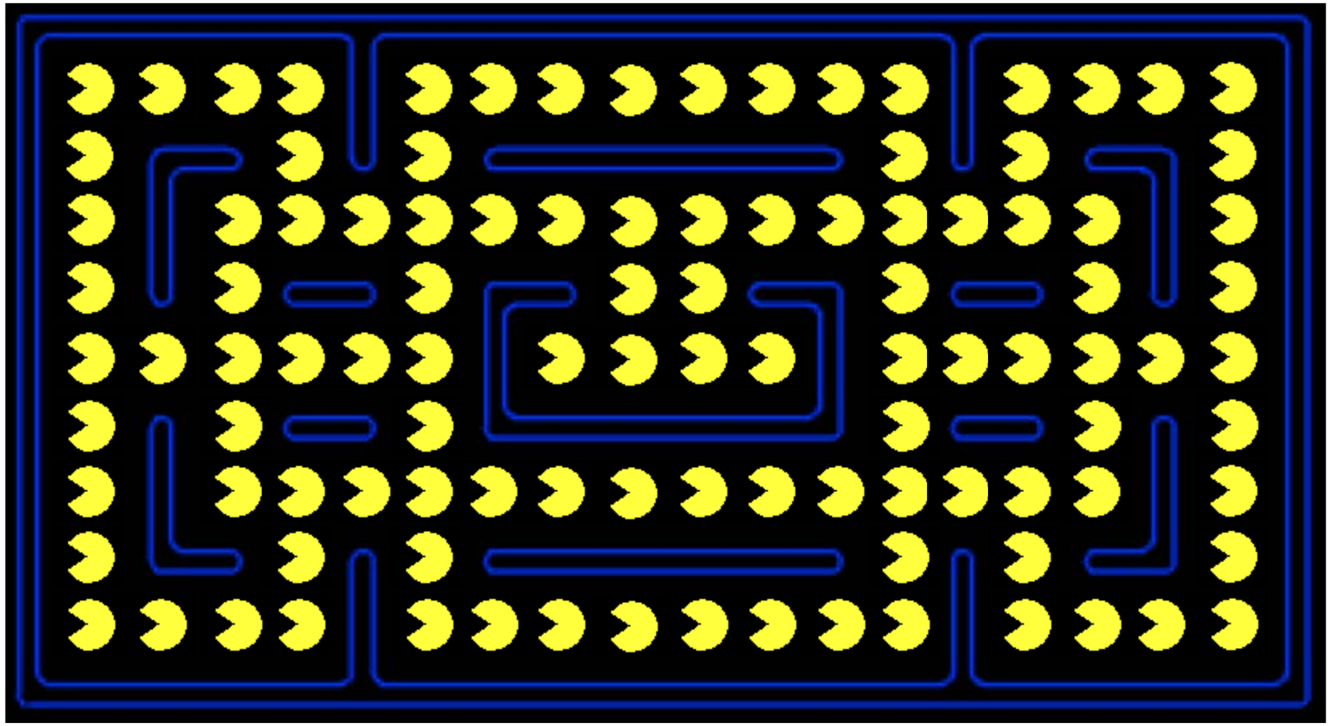
Localization demo

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



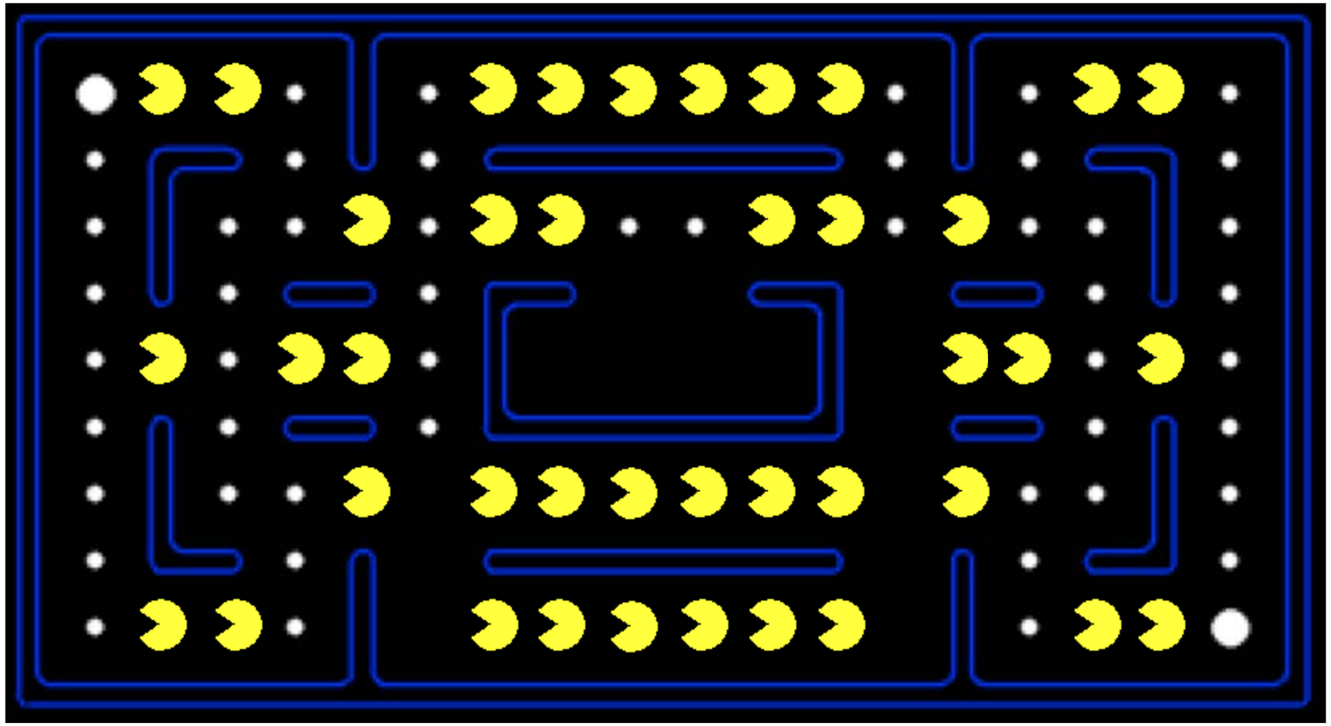
Localization demo

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



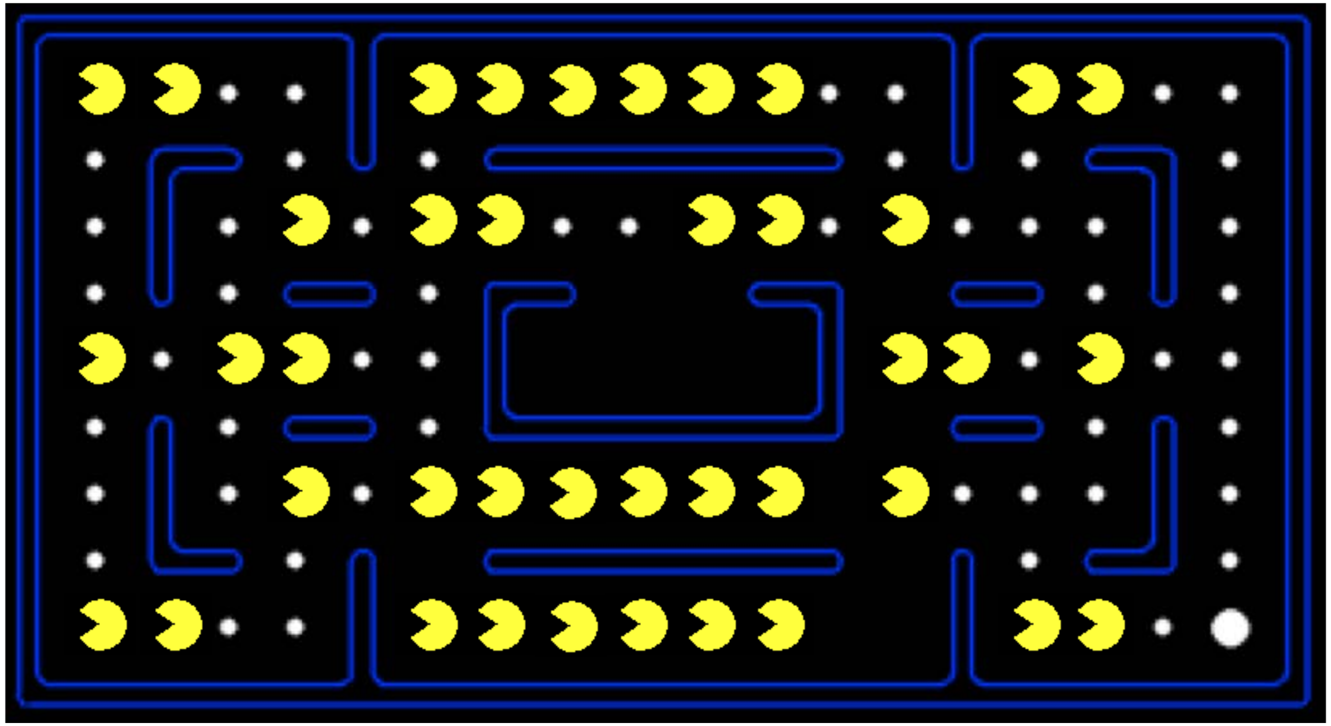
Localization demo

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



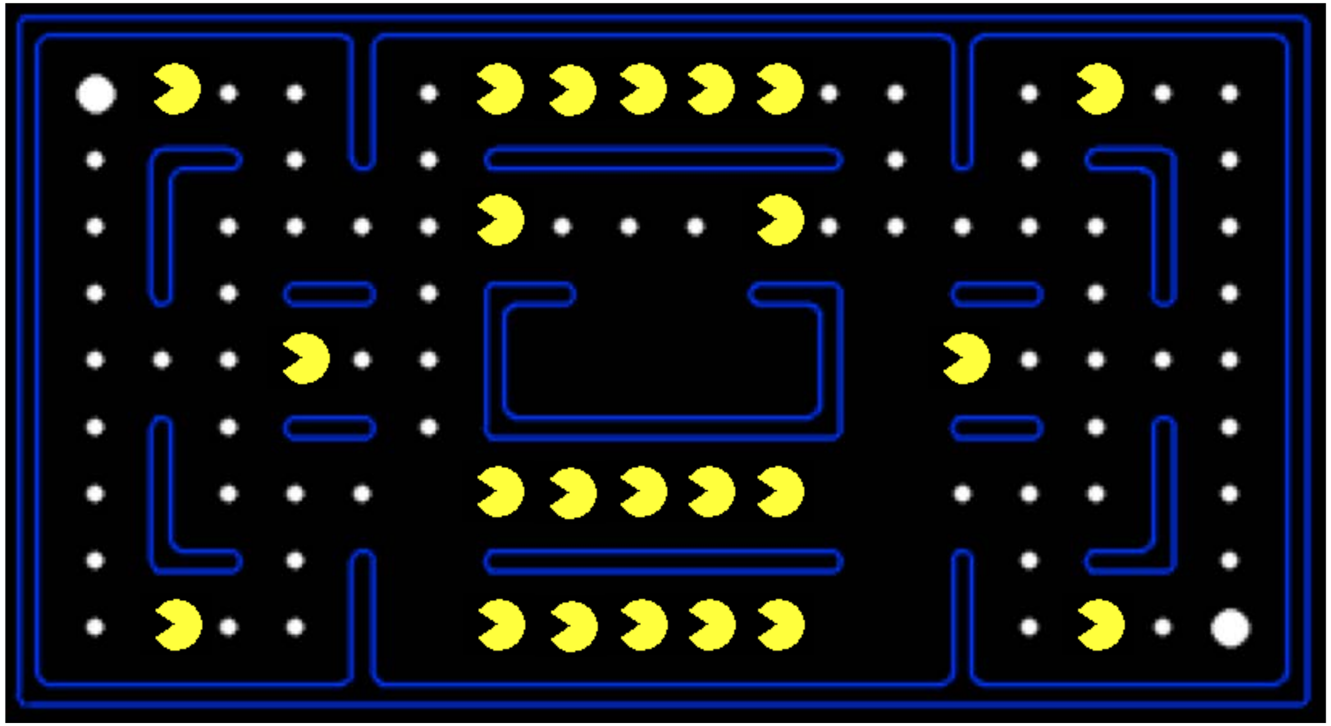
Localization demo

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



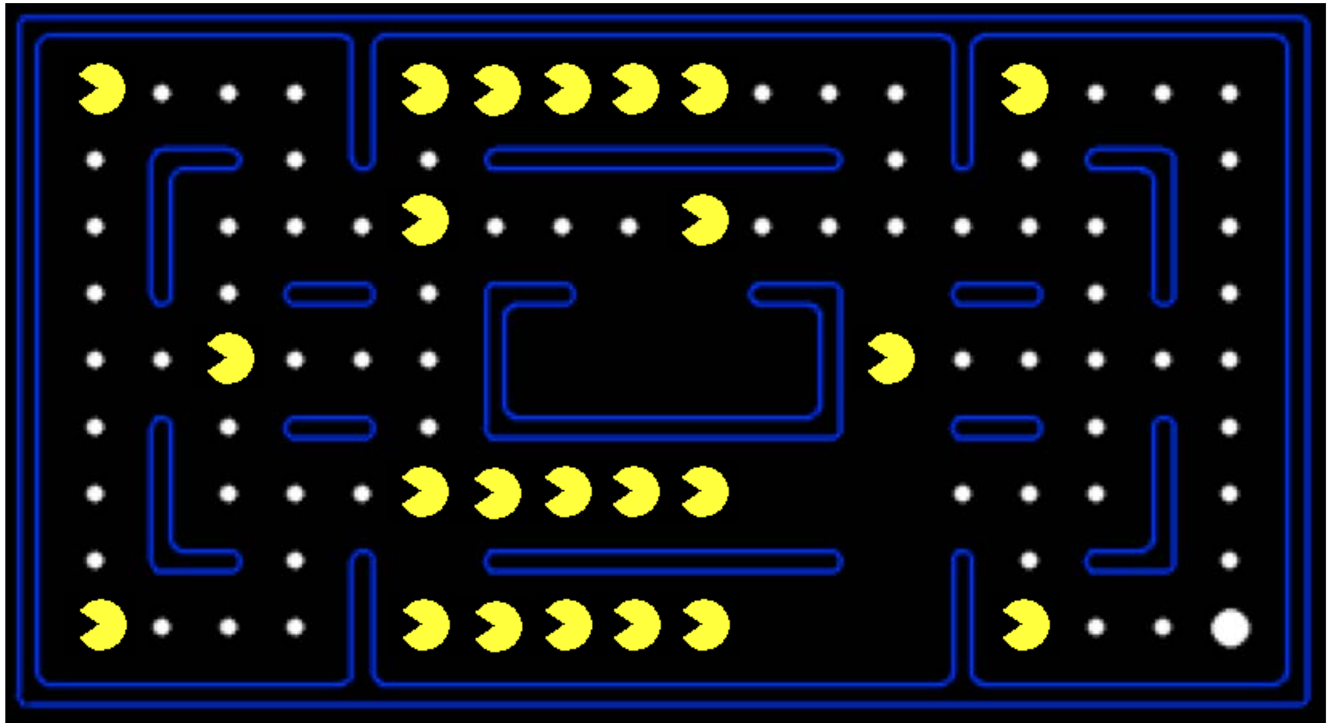
Localization demo

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



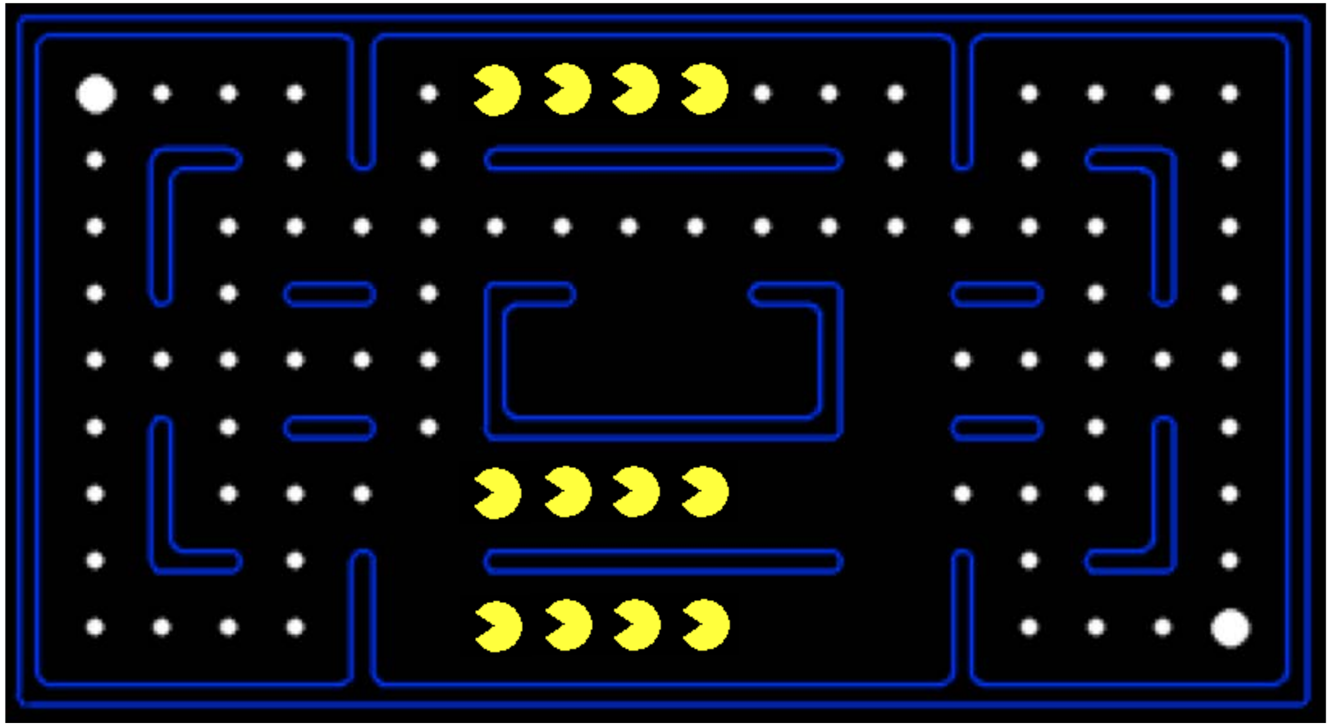
Localization demo

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



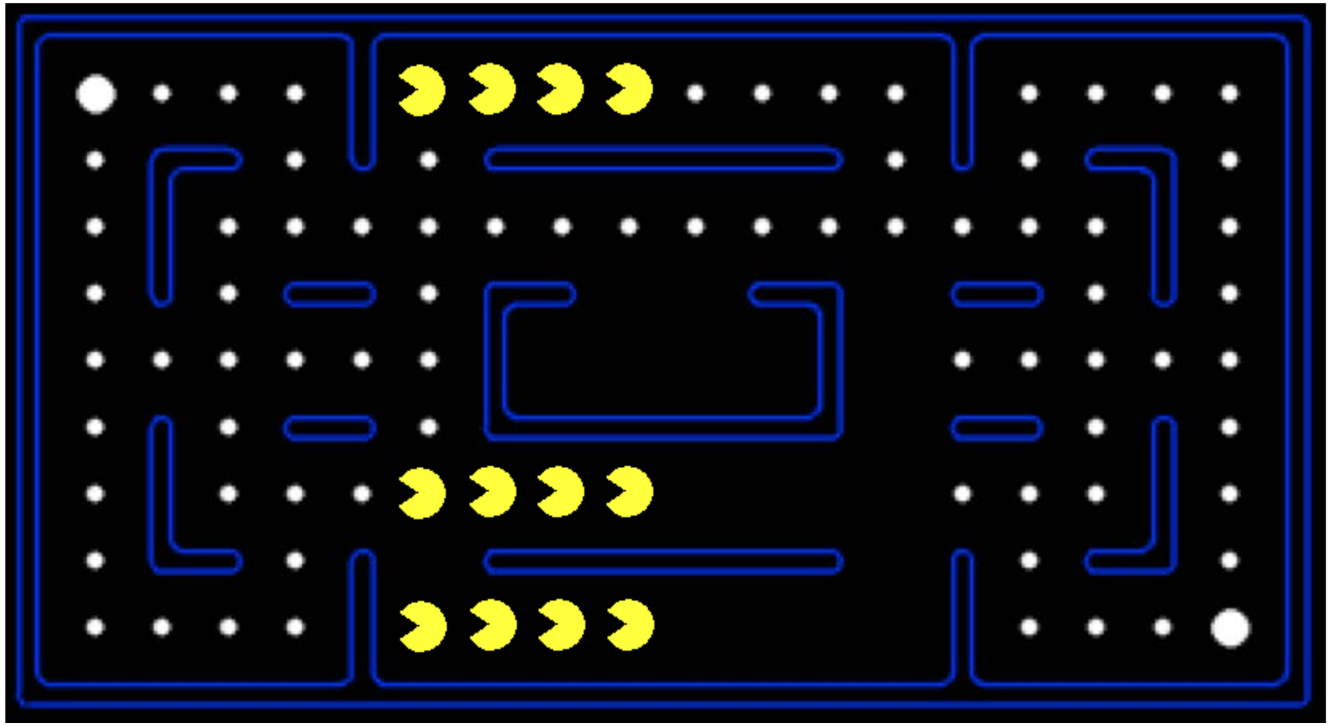
Localization demo

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

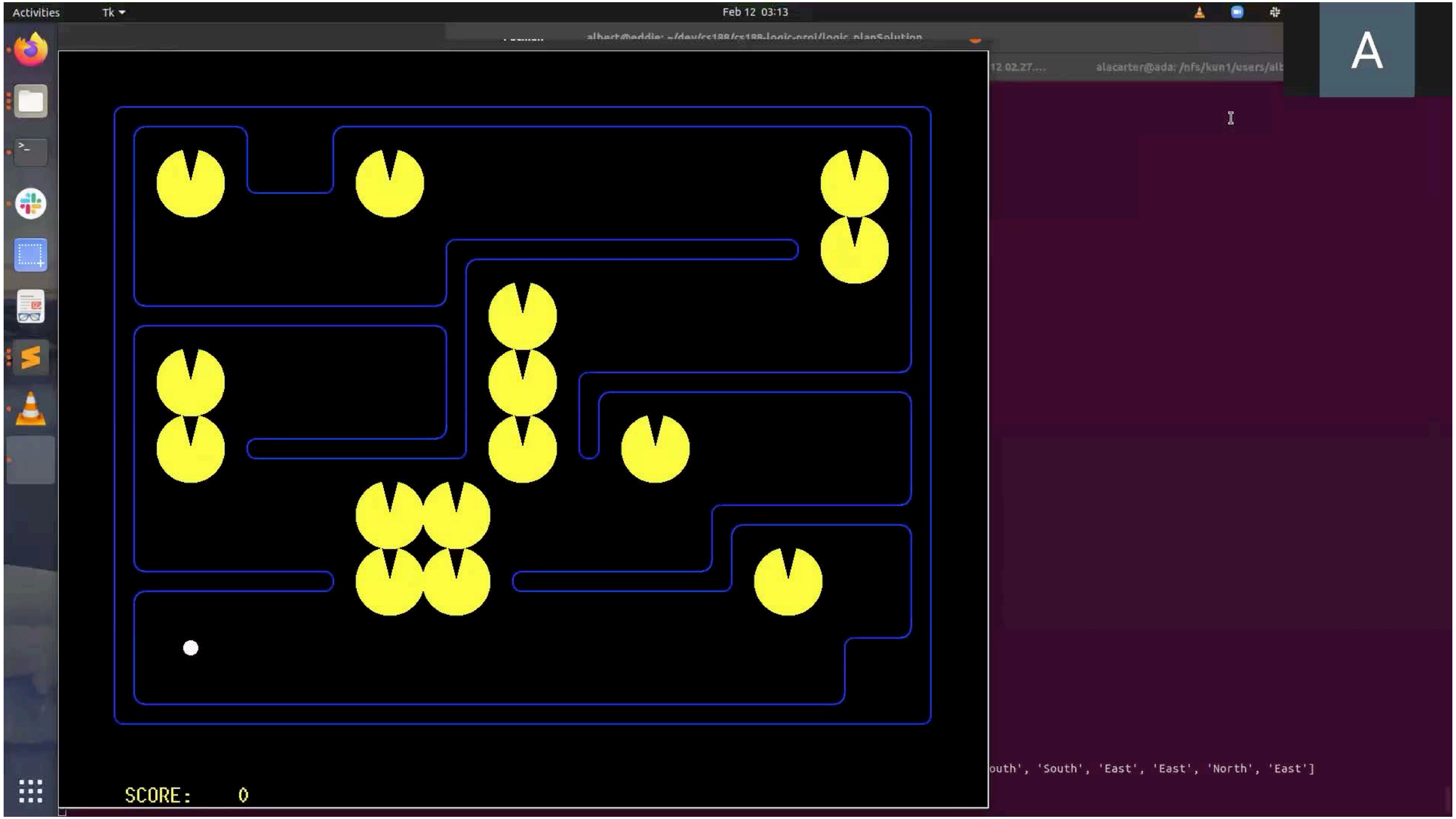


Localization demo

- 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 

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

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 $True \Rightarrow X$)

Forward chaining: example

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \wedge C \rightarrow D$
4. $D \wedge E \rightarrow Q$
5. $A \wedge D \rightarrow Q$
6. A

… Is Q true or false?

Forward chaining: example

… init:

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \wedge C \rightarrow D$
4. $D \wedge E \rightarrow Q$
5. $A \wedge D \rightarrow Q$
6. A

- *count*: $[1, 1, 2, 2, 2, 0]$
- *inferred*: $\{A : F, B : F, C : F, D : F, E : F, Q : F\}$
- *agenda*: $[A]$

… iteration 0:

… Is Q true or false?

Forward chaining: example

1. $A \to B$
2. $A \to C$
3. $B \wedge C \to D$
4. $D \wedge E \to Q$
5. $A \wedge D \to Q$
6. A

… Is Q true or false?

… init:

- *count*: $[1, 1, 2, 2, 2, 0]$
- *inferred*: $\{A : F, B : F, C : F, D : F, E : F, Q : F\}$
- *agenda*: $[A]$

… iteration 0:

- *count*: $[0, 0, 2, 2, 1, 0]$
- *inferred*: $\{A : T, B : F, C : F, D : F, E : F, Q : F\}$
- *agenda*: $[B, C]$

Forward chaining: example

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \wedge C \rightarrow D$
4. $D \wedge E \rightarrow Q$
5. $A \wedge D \rightarrow Q$
6. A

… Is Q true or false?

… iteration 0:

- *count*: $[0, 0, 2, 2, 1, 0]$
- *inferred*: $\{A : T, B : F, C : F, D : F, E : F, Q : F\}$
- *agenda*: $[B, C]$

… iteration 1:

- *count*: $[0, 0, 1, 2, 1, 0]$
- *inferred*: $\{A : T, B : T, C : F, D : F, E : F, Q : F\}$
- *agenda*: $[C]$

Forward chaining: example

1. $A \to B$
2. $A \to C$
3. $B \wedge C \to D$
4. $D \wedge E \to Q$
5. $A \wedge D \to Q$
6. A

… Is Q true or false?

… iteration 2:

- *count*: $[0, 0, 0, 2, 1, 0]$
- *inferred*: $\{A : T, B : T, C : T, D : F, E : F, Q : F\}$
- *agenda*: $[D]$

Forward chaining: example

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \wedge C \rightarrow D$
4. $D \wedge E \rightarrow Q$
5. $A \wedge D \rightarrow Q$
6. A

… iteration 3:

- *count*: $[0,0,0,1,0,0]$
- *inferred*: $\{A:T, B:T, C:T, D:T, E:F, Q:F\}$
- *agenda*: $[Q]$

… Is Q true or false?

Properties of forward chaining

- Theorem: FC is sound and complete for definite-clause KBs
- Soundness: follows from soundness of Modus Ponens (easy to check)
- Completeness: see proof on p. 230-1.
- Runs in ***linear*** time using two simple indexing 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
- Very commonly used in database (Datalog) systems for updating

Backward chaining (briefly)

- Backward chaining works the other way around:
  - Start from the goal Y to prove
  - Find implications $X_1 \wedge X_2 \wedge \ldots X_n \Rightarrow Y$ with Y on the right-hand side
  - Set up X_1, X_2, …, X_n as subgoals to prove recursively (stop at known facts)
- Theorem: BC is sound and complete for definite-clause KBs
- Linear-time with suitable indexing and caching of subgoals
- BC with first-order definite clauses is the basis of *logic programming*

Resolution (briefly)

- The resolution inference rule takes two implication sentences (of a particular form) and infers a new implication sentence:
- Example: $A \wedge B \wedge C \Rightarrow \mathbf{U} \vee V$

$$\frac{\begin{array}{l} A \wedge B \wedge C \Rightarrow \mathbf{U} \vee V \\ D \wedge E \wedge \mathbf{U} \Rightarrow X \vee Y \end{array}}{A \wedge B \wedge C \wedge D \wedge E \Rightarrow V \vee X \vee Y}$$

- Resolution is complete for propositional logic
- Exponential time in the worst case

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