CS 188: Artificial Intelligence

Inference in Propositional Logic

Quiz 1:

Is the following sentence true in the following model?

Model: {A: True, B: False, C: True, D: True, E: True}

Sentence: $((\neg A \vee B) \Rightarrow \neg(D \vee E)) \wedge (C \Leftrightarrow C)$

Quiz 2:

Do the following sentences entail H?

A, C, D, F, G, $F \wedge G \Rightarrow D$, $B \wedge A \wedge G \Rightarrow E$,

$A \wedge E \Rightarrow H$, $C \wedge D \wedge E \Rightarrow A$,

$F \wedge G \Rightarrow E$, $B \Rightarrow E$, $L \Rightarrow B$, $B \Rightarrow H$

Slides mostly from Stuart Russell

University of California, Berkeley

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

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
- 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

Reminder about definite clauses: $X_1 \wedge X_2 \wedge \ldots \wedge X_n \Rightarrow Y$

```
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 symbols 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
```

Forward chaining algorithm: Example

Sentences: A, B, D, $A \wedge B \Rightarrow C$, $C \wedge D \Rightarrow E$ Query: F

	A	B	C	D	E	F		$A \wedge B \Rightarrow C$	$C \wedge D \Rightarrow E$
Inferred?	F	F	F	F	F	F	Count?	2	2
Agenda?	x	x		x					

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 proof:

    1. FC reaches a fixed point where no new atomic sentences are derived
    2. Consider the final set of known-to-be-true symbols as a model ***m*** (other ones false)
    3. Every clause in the original KB is true in ***m***

        Proof: Suppose a clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in ***m***
        Then $a_1 \wedge \ldots \wedge a_k$ is true in ***m*** and b is false in ***m***
        Therefore the algorithm has not reached a fixed point!

    4. Hence ***m*** is a model of KB
    5. If KB |= q, q is true in every model of KB, including ***m***

Backward chaining

- Puzzle to try at home: develop a “Backward chaining algorithm”
- Then look it up to see if you’ve reproduced the standard algorithm
- Idea:
  - Keep track of what you’re trying to prove
  - Look for sentences that might get you there

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 possible worlds
  - iff $\neg(\alpha \Rightarrow \beta)$ is false in all possible worlds
  - iff $\alpha \wedge \neg\beta$ is false in all possible 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 expressed as a ***conjunction*** of ***clauses***
- Each clause is a ***disjunction***
- Each literal is a symbol or a n
- (A v ¬B v ¬C) ∧ (¬A) ∧ (¬D … v F
- Convert anything to CNF with standard transf…tions!
  - 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)

Replace biconditional by two implications

Replace α ⇒ β by ¬α v β

Distribute v over ∧

Distributivity

- $(A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C)$
- (I’m in SF or I’m in Berkeley) and I’m alive
- (I’m in SF and I’m alive) or (I’m in Berkeley and I’m alive)
- $(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$
- $\neg(A \wedge B) = (\neg A \vee \neg B)$
- It’s not the case that (I’m alive and I’m kicking)
- (I’m not alive) or (I’m not kicking)
- $\neg(A \vee B) = (\neg A \wedge \neg B)$

Reduction to CNF

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

1. Get rid of $\Leftrightarrow$
   - Replace $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - α and β could be long expressions, not just symbols
2. Get rid of $\Rightarrow$
   - Replace $\alpha \Rightarrow \beta$ with $\beta \vee \neg\alpha$
3. Distribute $\neg$s to lower levels
   - Replace $\neg(\alpha \wedge \neg\beta)$ with $\neg\alpha \vee \beta$
4. If any $\vee$s are at a higher level than $\wedge$s, distribute down
   - Replace $(\alpha \wedge \beta \wedge \gamma) \vee (\varepsilon \wedge \delta)$ with…
   - $(\alpha \vee \varepsilon) \wedge (\beta \vee \varepsilon) \wedge (\gamma \vee \varepsilon) \wedge (\alpha \vee \delta) \wedge (\beta \vee \delta) \wedge (\gamma \vee \delta)$

Depth first search solver

clauses

Reminder of conjunctive normal form: $(A \lor B) \land (A \lor \neg C \lor D) \land (C \lor \neg B) \land (B)$

function DFSS(clauses, symbols, partmodel={}) **returns** true or false

  if every clause in clauses is true in partmodel **then return** true

  if some clause in clauses is false in partmodel **then return** false

  Tricks go here

  P ← First(symbols); rest ← Rest(symbols)

  return or(DFSS(clauses, rest, partmodel∪{P=true})**,**

        DFSS(clauses, rest, partmodel∪{P=false}))

Efficient SAT solvers

- DPLL (Davis-Putnam-Logemann-Loveland) is the core of modern solvers
- Recursive depth-first search over partial models with some extras:
  - ***Early termination***: stop if

    clauses

    - all clauses are satisfied; e.g., $(A \lor B) \land (A \lor \neg C)$ is satisfied by $\{A=\text{true}\}$
    - any clause is falsified; e.g., $(A \lor B) \land (A \lor \neg C)$ is falsified if $\{A=\text{false}, B=\text{false}\}$
  - ***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 positive in every clause of $(A \lor B) \land (A \lor \neg C) \land (C \lor \neg B)$ so set it to true
  - ***Unit clauses***: if clause has one unresolved literal, set symbol to satisfy clause
    - E.g., if A=false, $(A \lor B) \land (\neg B \lor \neg C)$ becomes $(\text{false} \lor B) \land (\neg B \lor \neg C)$, so set B=true
    - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.

DPLL algorithm

clauses

Reminder of conjunctive normal form: $(A \vee B) \wedge (A \vee \neg C \vee D) \wedge (C \vee \neg B) \wedge (B)$

```
function DPLL(clauses, symbols, partmodel={}) returns true or false
  if every clause in clauses is true in partmodel then return true
  if some clause in clauses is false in partmodel then return false
  clauses ← every clause in clauses that is not true (or false)
  P, value ←FIND-PURE-SYMBOL(symbols, clauses, partmodel)
  if P is non-null then return DPLL(clauses, symbols–P, partmodelU{P=value})
  P, value ←FIND-UNIT-CLAUSE(clauses, partmodel)      *clause with 1 unresolved literal
  if P is non-null then return DPLL(clauses, symbols–P, partmodelU{P=value})
  P ← First(symbols); rest ← Rest(symbols)
  return or(DPLL(clauses, rest, partmodelU{P=true}),
            DPLL(clauses, rest, partmodelU{P=false}))
```

DPLL Example

- Start with $(A \lor B) \land (\neg A \lor \neg C) \land (C \lor \neg B) \land (A \lor B \lor C)$
- No pure symbols or unit clauses

- $(T \lor B) \land (\neg T \lor \neg C) \land (C \lor \neg B) \land (T \lor B \lor C)$
- Remove satisfied clauses
- $(\neg T \lor \neg C) \land (C \lor \neg B)$
- B is a pure symbol – set it to false
- $(\neg T \lor \neg C) \land (C \lor \neg F)$
- Remove satisfied clauses
- $(\neg T \lor \neg C)$
- C is a pure symbol (also a unit clause btw because the clause has one *unresolved* literal)
- $(\neg T \lor \neg F)$
- All clauses satisfied!

- $(F \lor B) \land (\neg F \lor \neg C) \land (C \lor \neg B) \land (F \lor B \lor C)$

Question: what partial model are we dealing with at this point? (What is a partial model?)

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 in positive / negative form
    - 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 ~100,000,000 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???***

Resolution (briefly)

- Every CNF clause can be written as
  - Conjunction of symbols $\Rightarrow$ disjunction of symbols
  - $A \vee B \vee \neg C \vee \neg D \quad = \quad C \wedge D \Rightarrow A \vee B$
- The resolution inference rule takes two such clauses and infers a new one by **resolving** complementary symbols:
- Example: $A \wedge B \wedge C \Rightarrow \mathbf{U} \vee V$

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

- Sentence unsatistfiable iff repeated resolution produces $() \Rightarrow ()$
- Resolution is complete for propositional logic, but exp-time

A knowledge-based agent

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

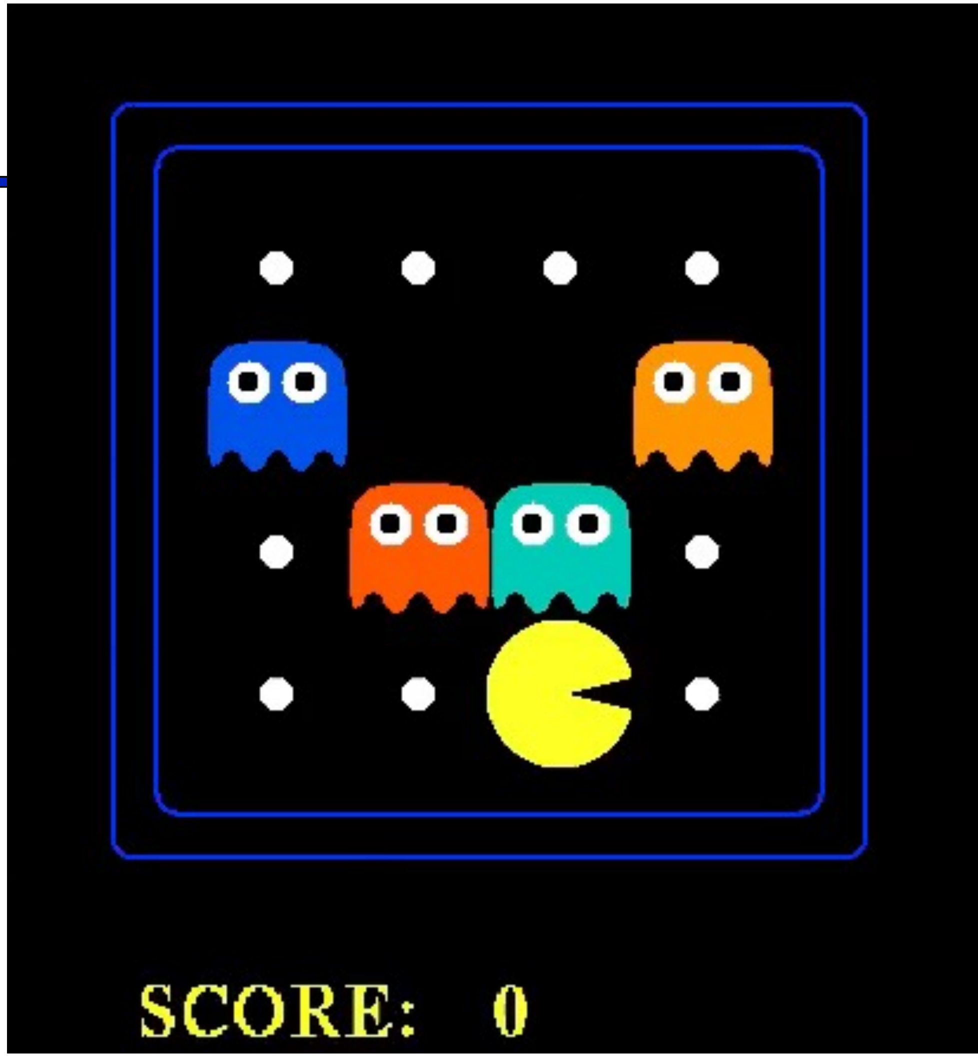
TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
action ← ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t←t+1
return action

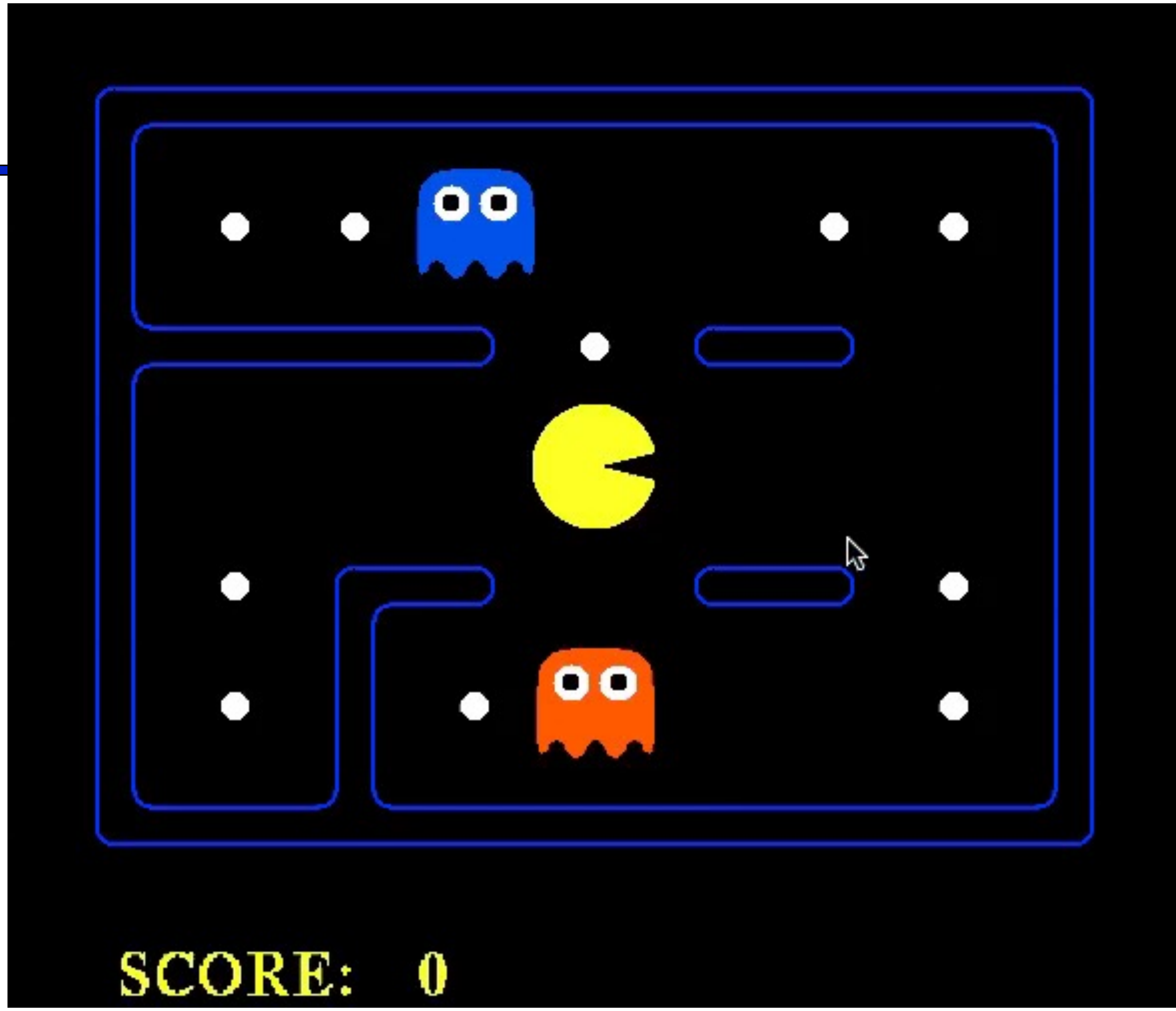
Planning as satisfiability

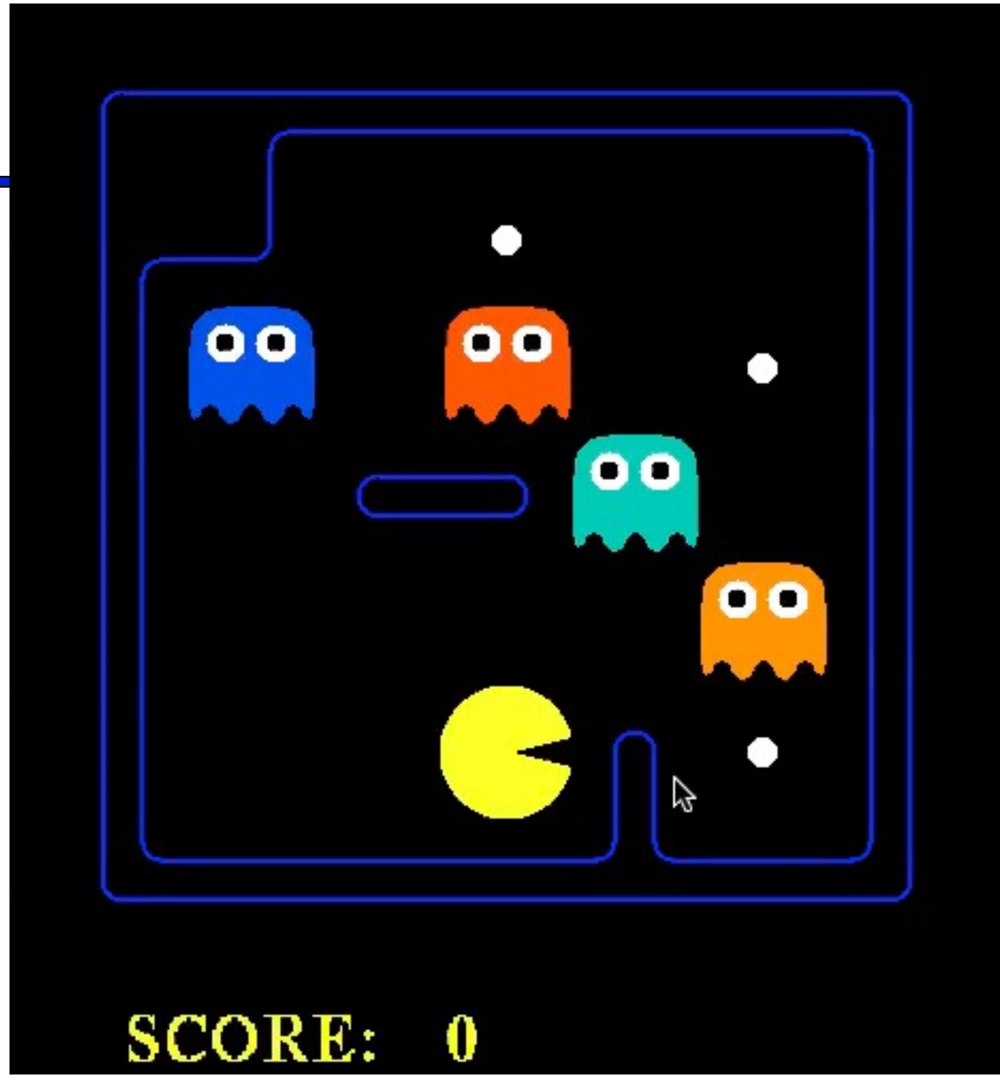
- Given a hyper-efficient SAT solver, can we use it to make plans?
- Yes, for **fully observable, deterministic** case
- For $T = 1$ to ∞,
  - Initialize the KB with **PacPhysics** for T time steps
  - Assert goal is true at time T
- Planning problem is solvable iff there is some satisfying assignment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

Basic PacPhysics for Planning

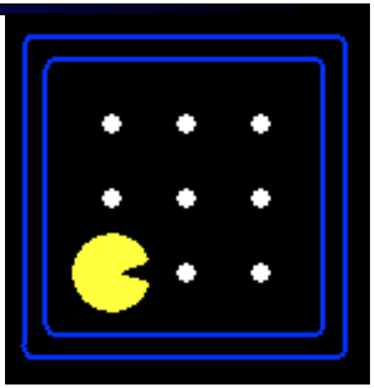
- **Map**: where the walls are and aren't, where the food is and isn't
- **Initial state**: Pacman start location (exactly one place), ghosts
- **Actions**: Pacman does exactly one action at each step
- **Transition model**:
  - <at x,y_t> ⇔ [at x,y_t-1 and stayed put] v [next to x,y_t-1 and moved to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


SCORE: 0


SCORE: 0


SCORE: 0

Reminder: Partially observable Pacman



- Basic question: where am I?
- 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 , …
- Sensor model:
  - Blocked_W_0 $\Leftrightarrow$ ((At_1,1_0 $\wedge$ Wall_0,1) v
    (At_1,2_0 $\wedge$ Wall_0,2) v
    (At_1,3_0 $\wedge$ Wall_0,3) v ….)
- Map: where are the walls
- Initial state: Pacman definitely somewhere
- Domain constraints: e.g. only one action per timestep
- Transition model: how state variables change (or don't)

State estimation

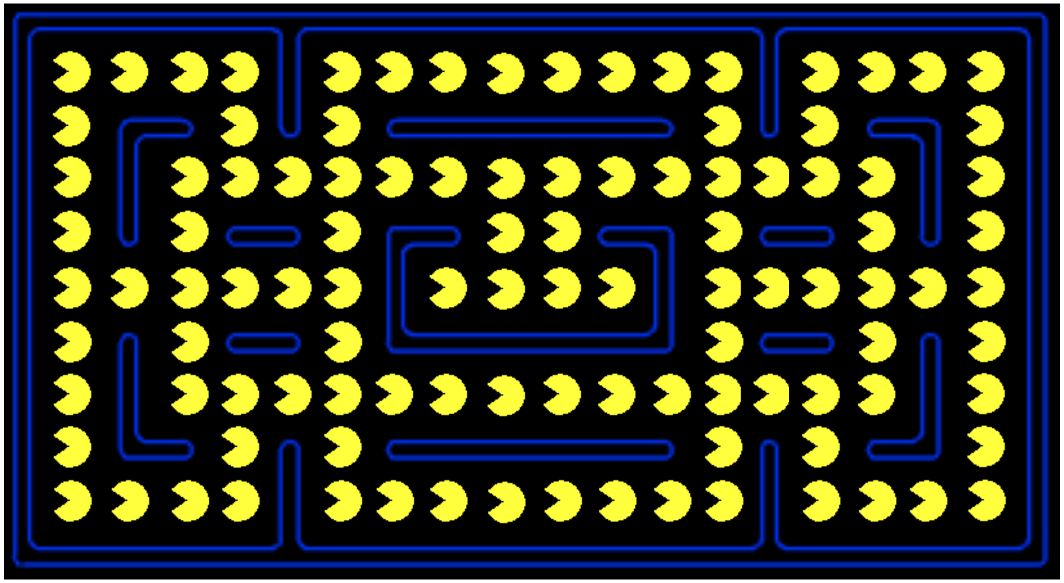
- ***State estimation*** means keeping track of what's true now
- A logical agent can just ask itself!
  - E.g., ask whether KB ∧ <actions> ∧ <percepts> |= At_2,2_6
- This is "lazy": it analyzes one's whole life history at each step!
- A more "eager" form of state estimation:
  - After each action and 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 $\wedge$ action_t-1 $\wedge$ percept_t $\models$ At_x,y_t, add At_x,y_t to KB
      - If KB $\wedge$ action_t-1 $\wedge$ percept_t $\models$ $\neg$ At_x,y_t, add $\neg$ At_x,y_t to KB
    - Choose an action
- Pacman's ***possible*** locations are those that are not provably false

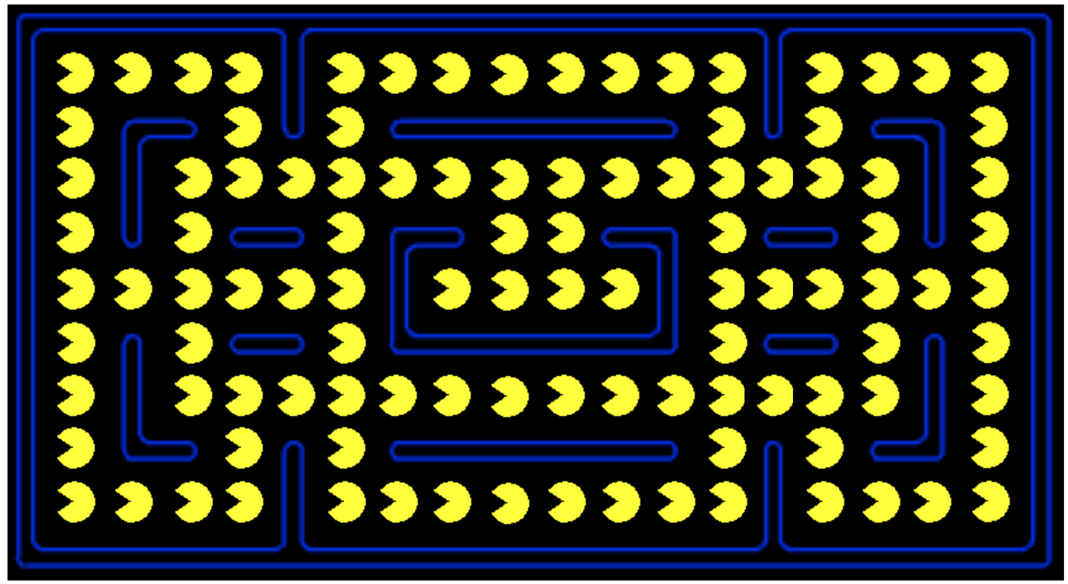
Localization demo

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



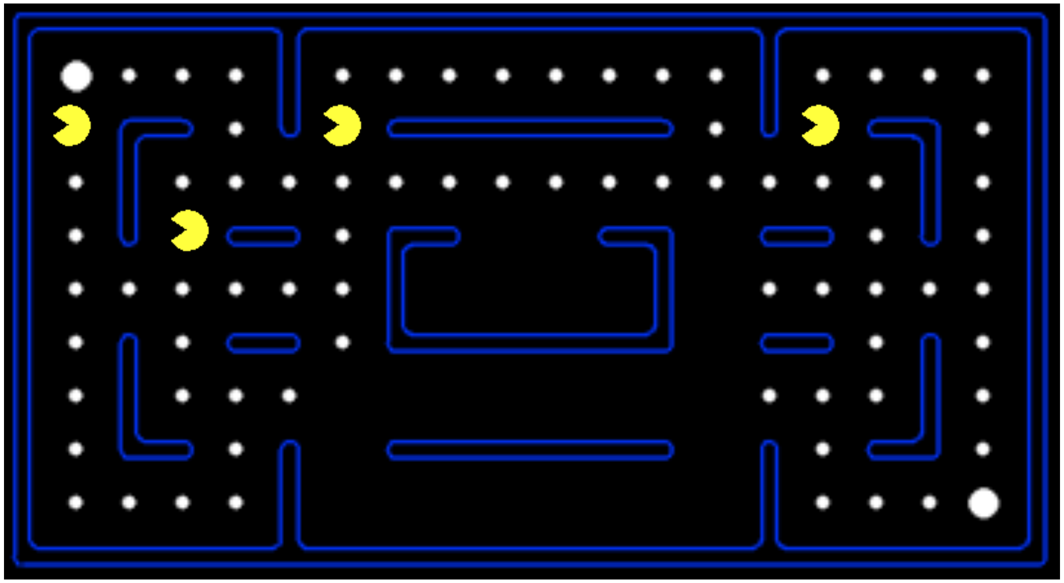
Localization demo

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



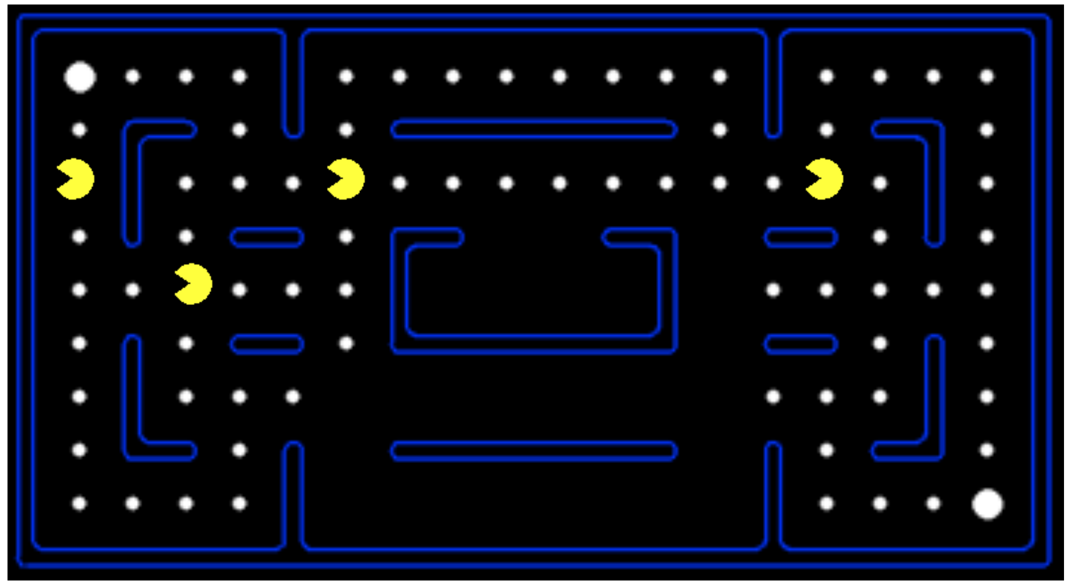
Localization demo

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



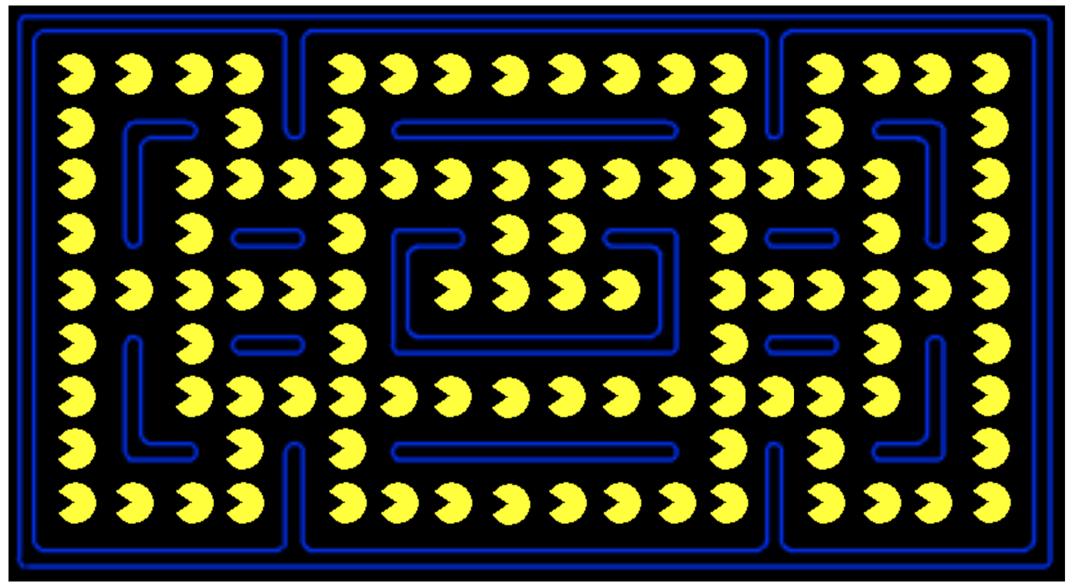
Localization demo

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



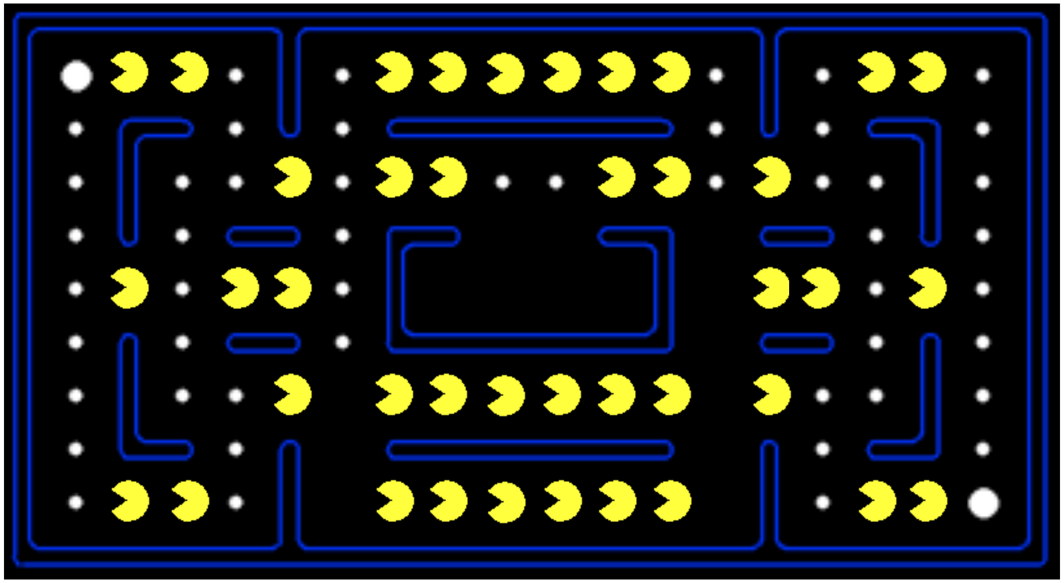
Localization demo

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

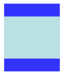


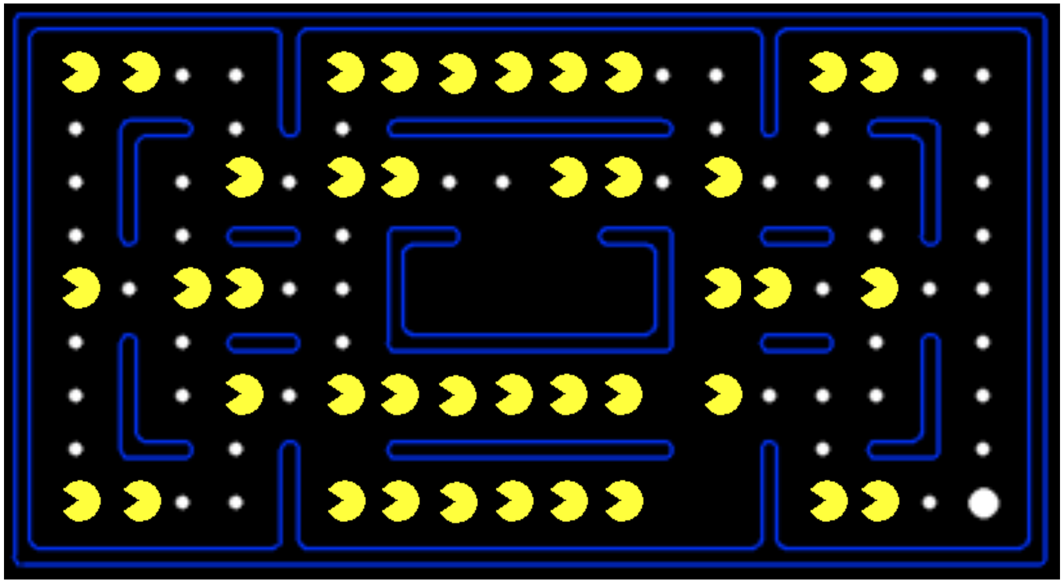
Localization demo

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

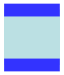


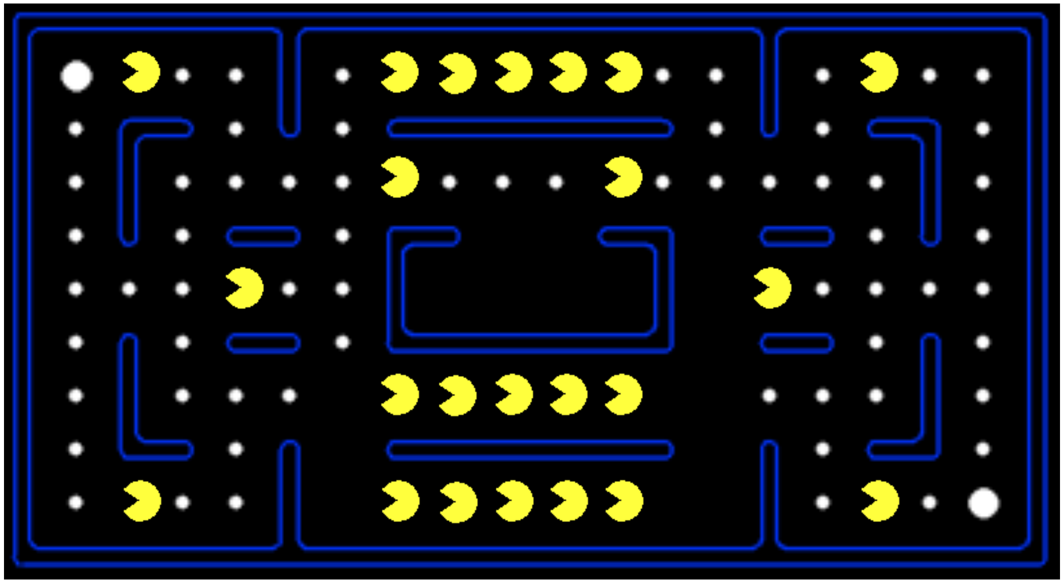
Localization demo

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

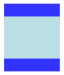
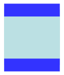


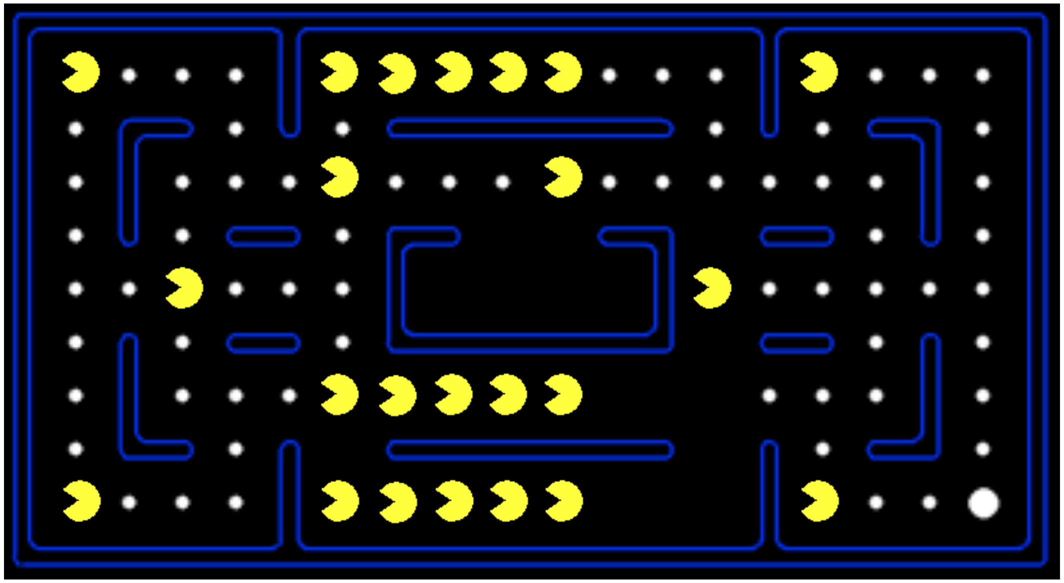
Localization demo

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

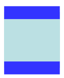


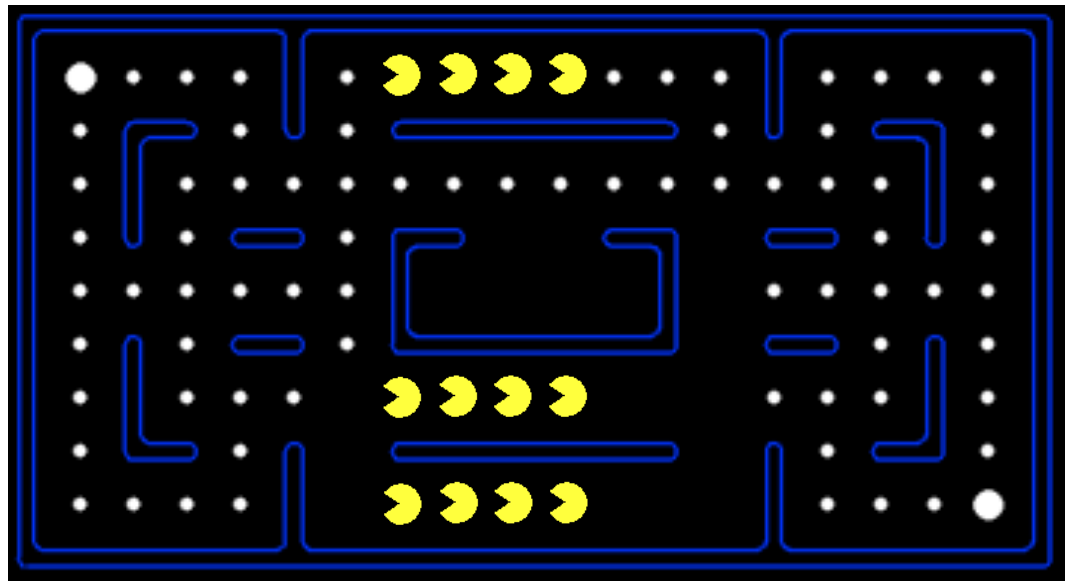
Localization demo

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



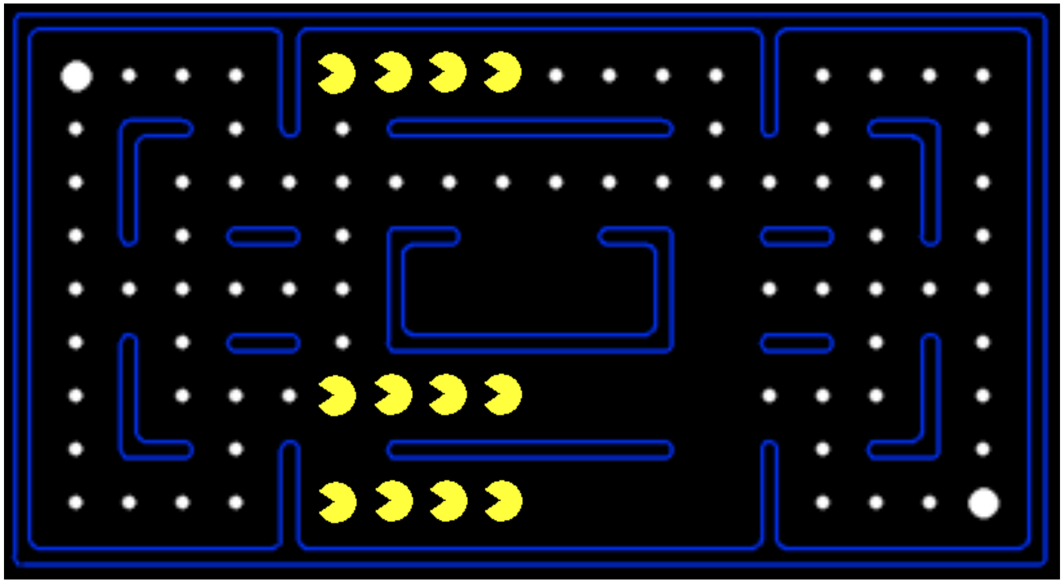
Localization demo

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

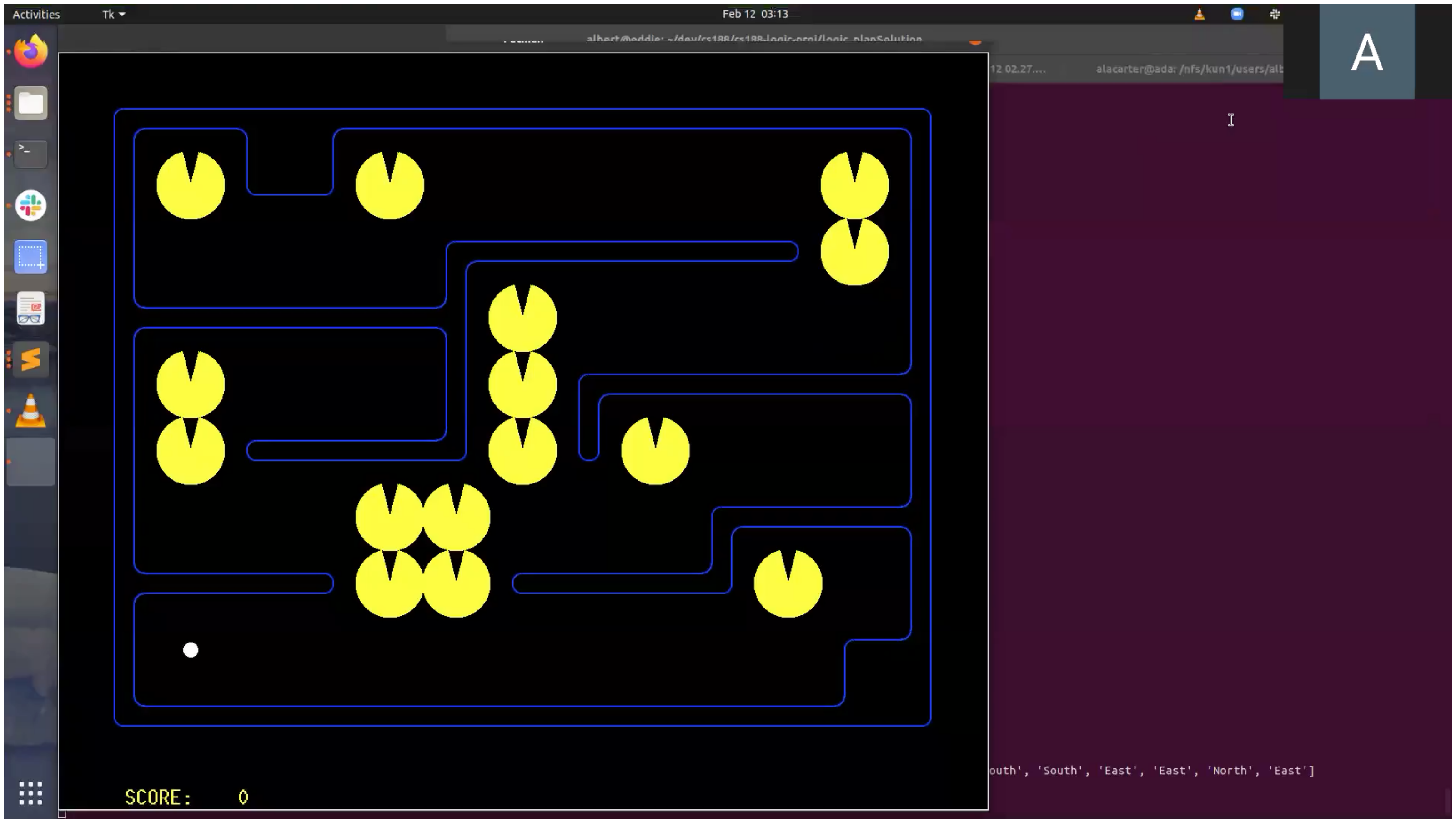


Localization demo

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

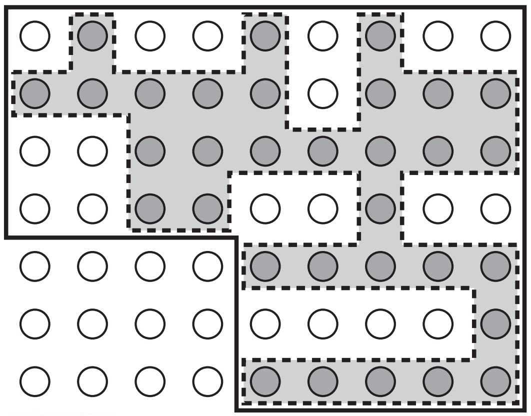


Localization with random movement



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
  - E.g., the study at time t was flawed or meat causes cancer
- Exact state estimation is intractable in general
  - Requires keeping track of properties of combinations of state variables



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

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