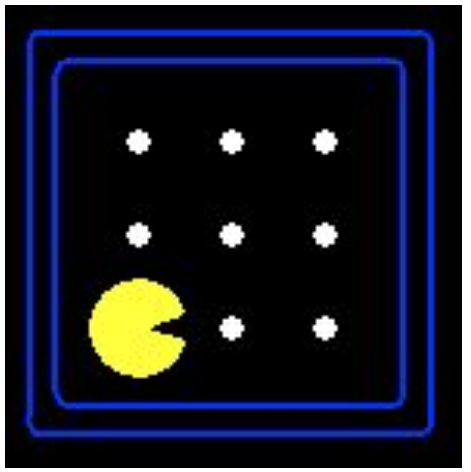
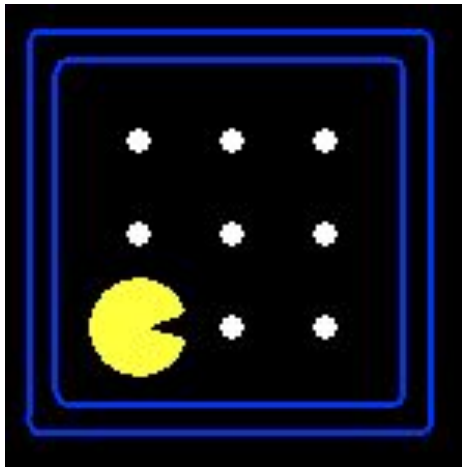
Reminder: Partially observable Pacman

- 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)



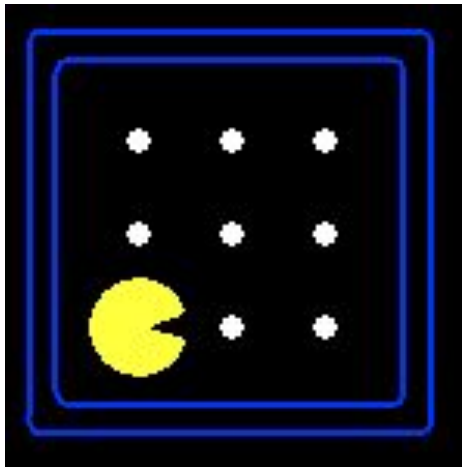
Pacman's knowledge base: Map

- Pacman knows where the walls are:
  - $\text{Wall}_{0,0} \wedge \text{Wall}_{0,1} \wedge \text{Wall}_{0,2} \wedge \text{Wall}_{0,3} \wedge \text{Wall}_{0,4} \wedge \text{Wall}_{1,4} \wedge \ldots$
- Pacman knows where the walls aren't!
  - $\neg\text{Wall}_{1,1} \wedge \neg\text{Wall}_{1,2} \wedge \neg\text{Wall}_{1,3} \wedge \neg\text{Wall}_{2,1} \wedge \neg\text{Wall}_{2,2} \wedge \ldots$



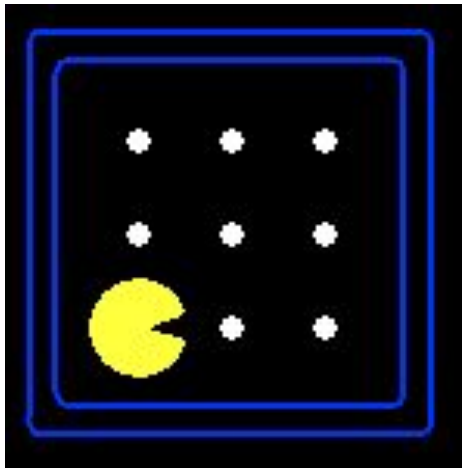
Pacman's knowledge base: Initial state

- Pacman doesn't know where he is!
- But he knows he's somewhere!
  - $At\_1,1\_0 \vee At\_1,2\_0 \vee At\_1,3\_0 \vee At\_2,1\_0 \vee \ldots$
- And he knows he's not where the walls are!
  - $\neg At\_0,0\_0 \wedge \neg At\_0,1\_0 \wedge \neg At\_0,2\_0 \wedge \ldots$
- And he knows he's not in two places at once!
  - $\neg(At\_1,1\_0 \wedge At\_1,2\_0) \wedge \neg(At\_1,1\_0 \wedge At\_1,3\_0) \wedge \ldots$



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 …)]

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)

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
```

Some reasoning tasks

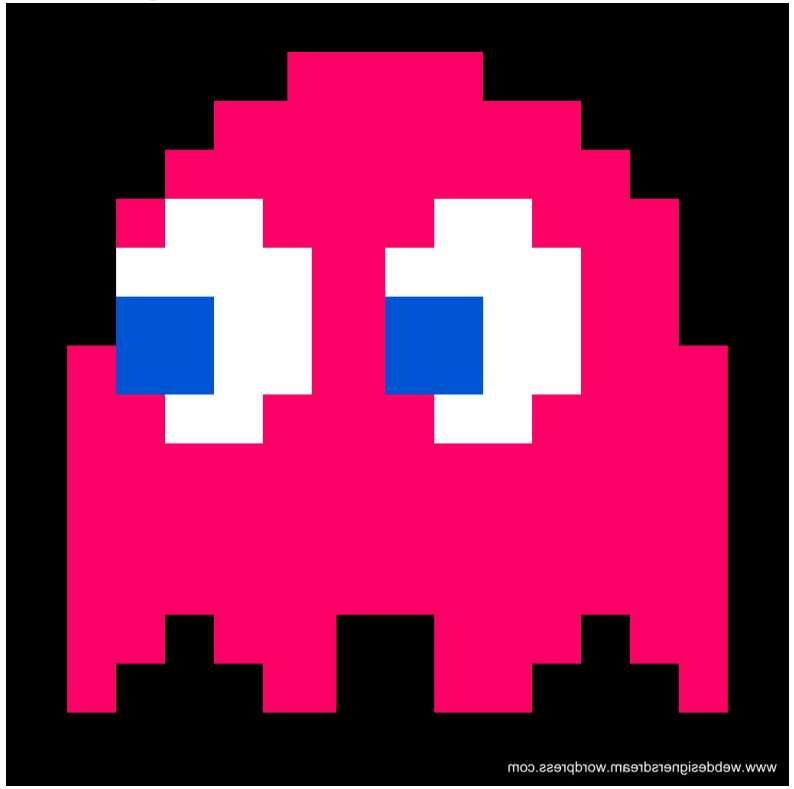
- Localization with a map and local sensing:
  - Given an initial KB, plus a sequence of percepts and actions, where am I?
- Mapping with a location sensor:
  - Given an initial KB, plus a sequence of percepts and actions, what is the map?
- Simultaneous localization and mapping:
  - Given …, where am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

CS 188: Artificial Intelligence

Inference in Propositional Logic



Instructors: Stuart Russell and Dawn Song

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) ⇒ 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
```

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 $\models q$, q is true in every model of KB, including ***m***

Resolution (briefly)

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

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

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

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 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 ∨ 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
  - 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???***

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 solvers (usually based on DPLL) are usually very efficient in practice