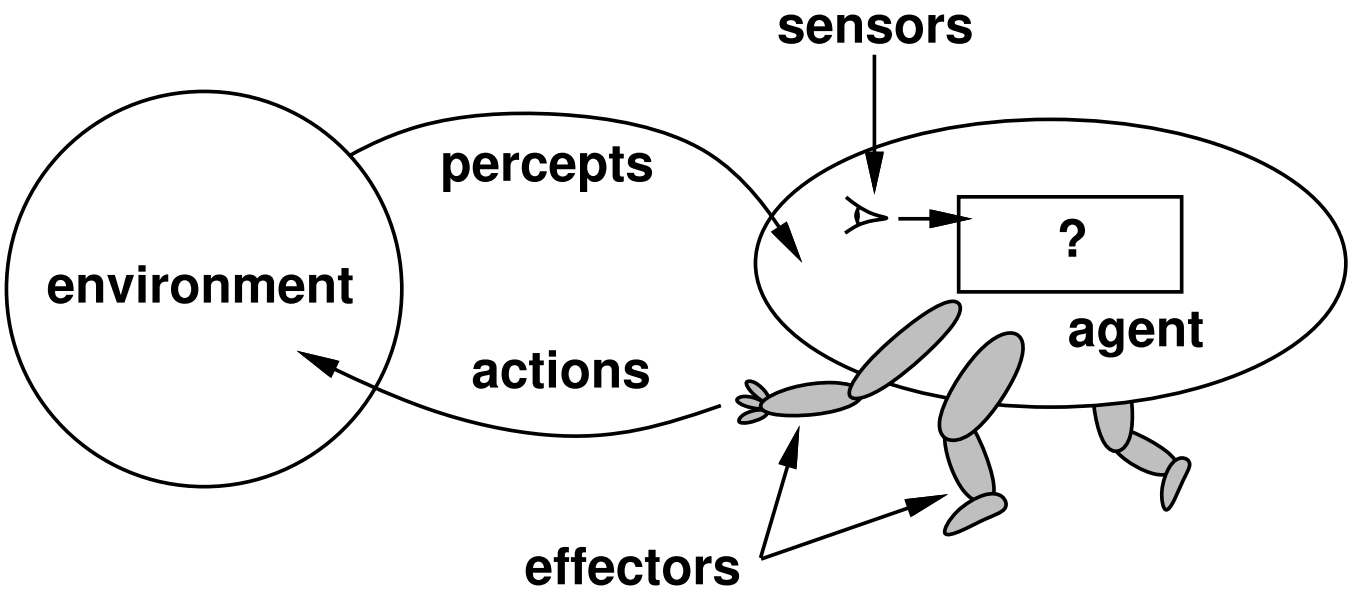
Agents and Environments



Example: Vacuum Cleaner Agent

- **Agent:** Robot vacuum cleaner
- **Environment:** Rooms of your apartment
- **Sensors:**
  - Dirt sensor: Detects when floor is dirty in front of the robot.
  - Bump sensor: Detects when it has bumped into something
  - Power sensor: Measures amount of power in batteries
  - Bag sensor: Amount of space remaining in dirt bag
- **Effectors:**
  - Motorized wheels
  - Suction motor
  - Plug into wall? Empty dirt bag?
- **Percepts:** "Floor is dirty"
- **Actions:** "Forward, 1 foot-per-second"

Rational Agent

- **Performance Measure**. Criteria for determining the quality of an agent.
  Example: Vacuum Cleaner Agent: dirt collected in 8 hour shift.
- **Avoiding Omniscience**. An omniscient agent is one that can predict the future perfectly. Our performance criterion should not require omniscience.
  Example: Door falls off jetliner and hits you as you are crossing the street.
- **Agent**. Mapping from percept sequences to actions.

  Definition of Ideal Rational Agent: *For each percept sequence, choose the action that maximizes expected value of the performance measure given only builtin knowledge and the percept sequence.*

Policy or Ideal Mapping

- **policy**: a mapping from percept sequences to actions.
  Every agent implements some policy.
- **agent programming**: designing and implementing good policies.

Ways of implementing policies

- **Tables**.
- **Rules**.
- **Search Procedures**.
- **Learning Algorithms**.

Implementing Agents using Tables

```
function TABLE-DRIVEN-AGENT(percept) returns action
  static: percepts, a sequence, initially empty
          table, a table, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Problems with this Approach

- **Space**. For chess, this would require 35^{100} entries.
- **Design Difficulty**. The designer would need to anticipate how the agent should respond to every possible percept sequence.

Avoiding Tables

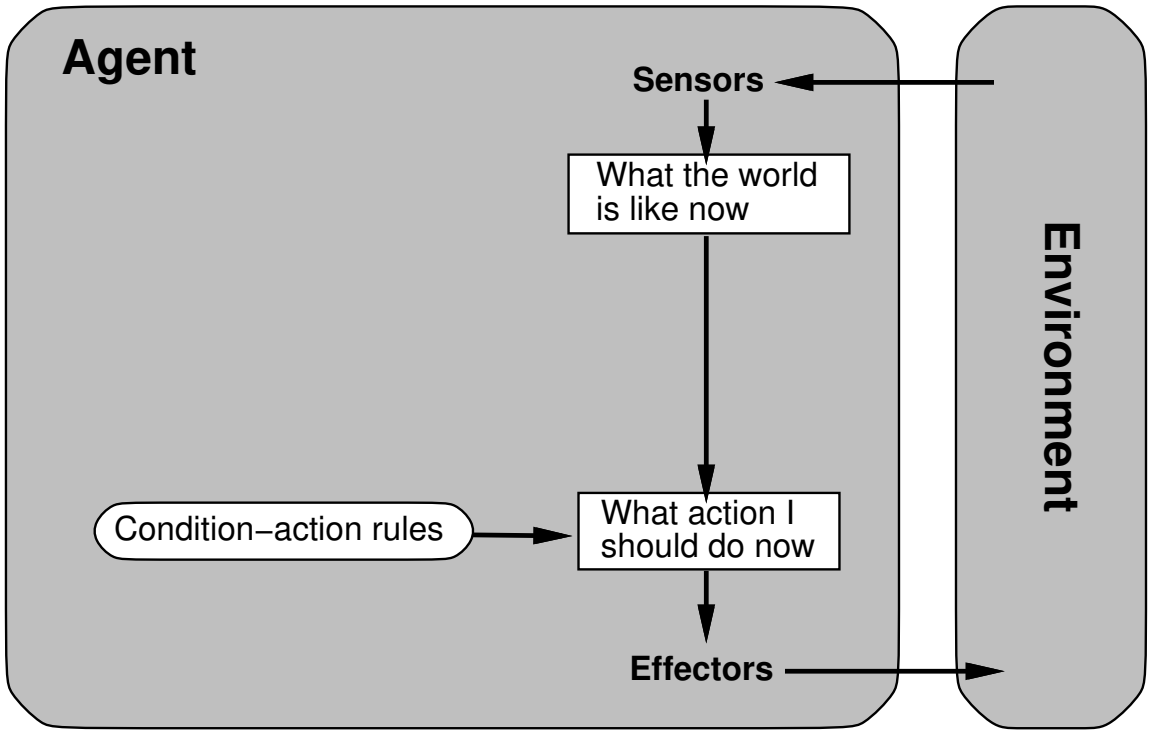
- **Compact Representations of the Table**. Many cells in the table will be identical.
  - **Markov Environments**. Example: In Chess, only the current board position matters, so all previous percepts dictate the same move.
    Environments where this is always true are called *Markov environments*.
  - **Irrelevant Percepts**. Example: If car in front of you slows down, you should apply the brakes. The color and model of the car, the music on the radio, the weather, and so on are all irrelevant.
- **Summarizing the Percept Sequence**. By analyzing the percept sequence, we can compute a model of the current state of the world. With this state, the agent can act as if the world is a Markov environment.
- **Computing the Table Dynamically**. In many cases, the agent can compute the desired action rather than looking it up in the table. This is a tradeoff of CPU time to gain space.

Example of Compact Representation: Implementing Agents using Rules

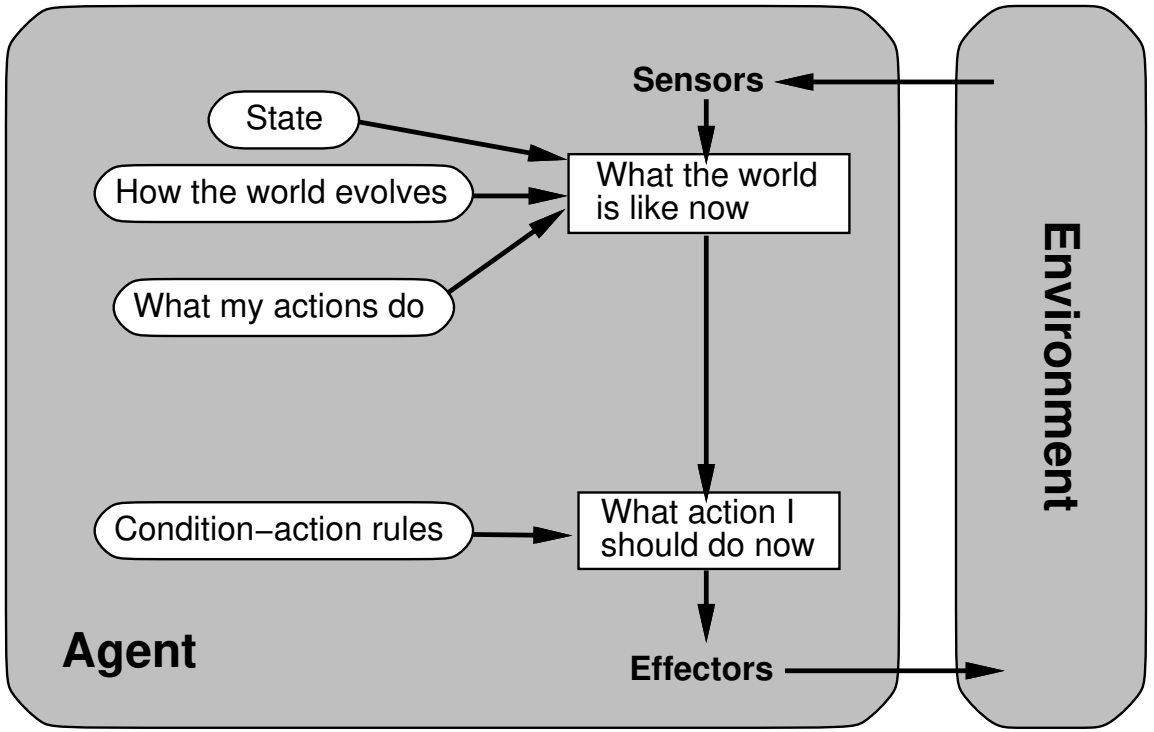
Rules express regularities in the table.

- **if** *car-in-front-is-braking* **then** *initiate-braking*

Notice that this rule only needs the last few percepts and ignores all irrelevant percepts.



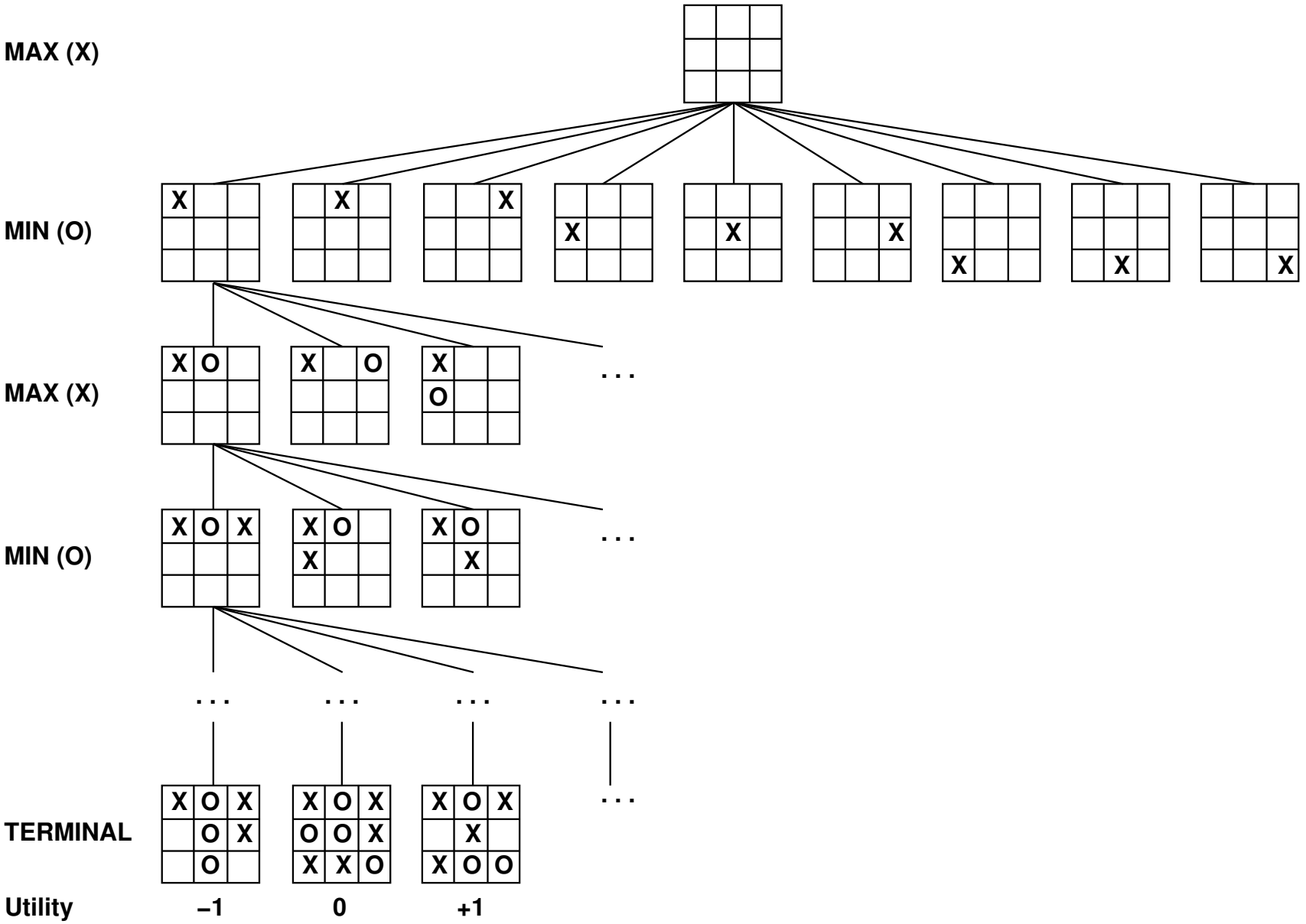
Example of Summarizing the Percept Sequence: Tracking the Environment



function REFLEX-AGENT-WITH-STATE(*percept*) **returns** *action*
  static: *state*, a description of the current world state
    rules, a set of condition-action rules

  state ← UPDATE-STATE(*state, percept*)
  rule ← RULE-MATCH(*state, rules*)
  action ← RULE-ACTION[*rule*]
  state ← UPDATE-STATE(*state, action*)
  return *action*

Example of Computing Table Dynamically

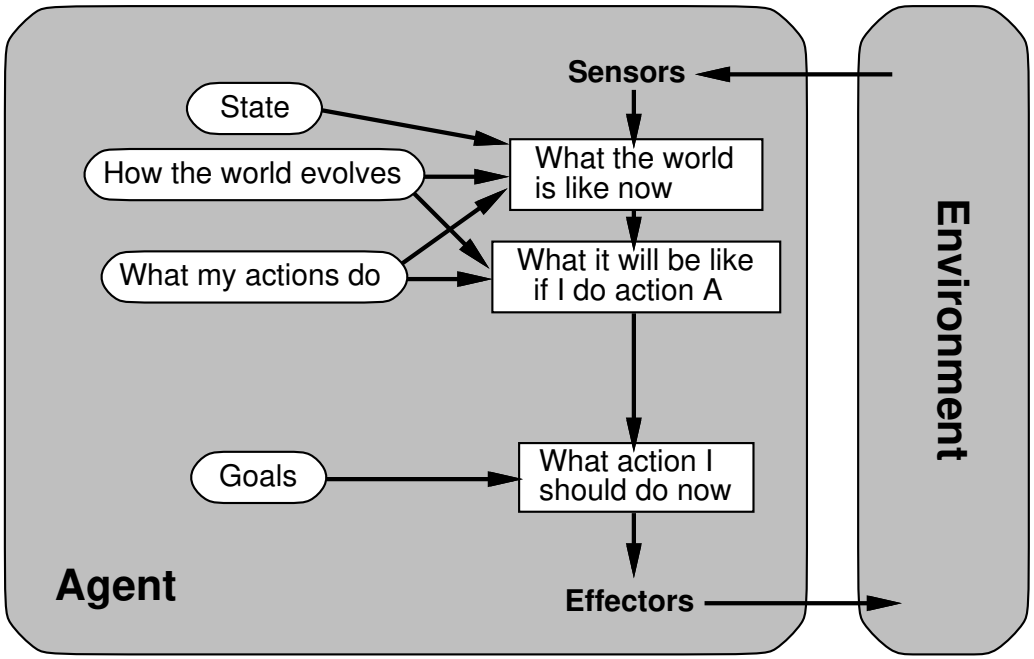


Problems with Computing Table Dynamically

- **Search space may be exponentially large**.
  Choosing the best action may be computationally intractable.
- **World may change while we are searching**.
  In a dynamic environment, we must act promptly. We may not have time to consider all possible action sequences.
- **Knowledge of the world may be incomplete or wrong**.
  We may not be able to accurately predict the future.

Goal-Based Agents

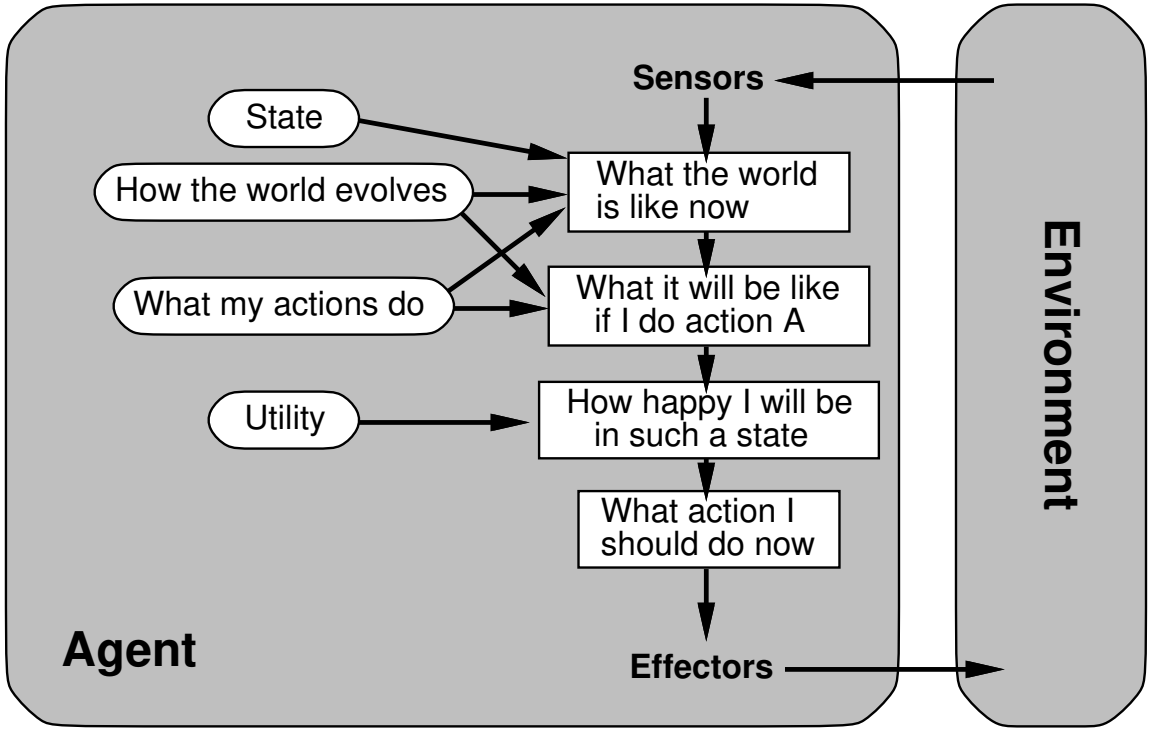
- **Generate possible sequences of actions**
- **Predict resulting states**
- **Assess goals in each resulting state**
- **Choose an action that will achieve the goal**



Notice that we can reprogram the agent simply by changing the goals.

Utility-Based Agents

Goals are *qualitative*. In some applications, we want to make *quantitative* comparisons of actions. For this, we need *utilities*. This is especially important for making tradeoffs.



Environments

The best approach to designing an agent depends on the complexity of its environment and the performance measure.

- **Accessible** vs. **Inaccessible**. Percepts observe whole state (Markov environment).
- **Deterministic** vs. **Nondeterministic**. Actions cause deterministic changes.
- **Episodic** vs. **Nonepisodic**. The performance measure concerns only a finite sequence of actions (no long-term issues).
- **Static** vs. **Dynamic**. The world does not change while the agent is thinking.
- **Discrete** vs. **Continuous**. Percepts and actions are discrete.

Advantages of Simpler Environments

- **Accessible** vs. **Inaccessible**. Policy can be based on only most recent percept.
- **Deterministic** vs. **Nondeterministic**. Consideration of the effects of actions is simpler.
- **Episodic** vs. **Nonepisodic**. Do not need to look ahead beyond end of episode.
- **Static** vs. **Dynamic**. Can afford lots of computing to make decisions.
- **Discrete** vs. **Continuous**. Interpretation of percepts and consideration of actions are simpler.