Chapter 2

Intelligent Agents

Chapter 2
Intelligent Agents

- What is an agent ?
  - An **agent** is anything that **perceiving** its environment through **sensors** and **acting** upon that environment through **actuators(device)**
  - Example:
    - Human is an agent
    - A robot is also an agent with cameras and motors
    - A thermostat detecting room temperature.

Intelligent Agents

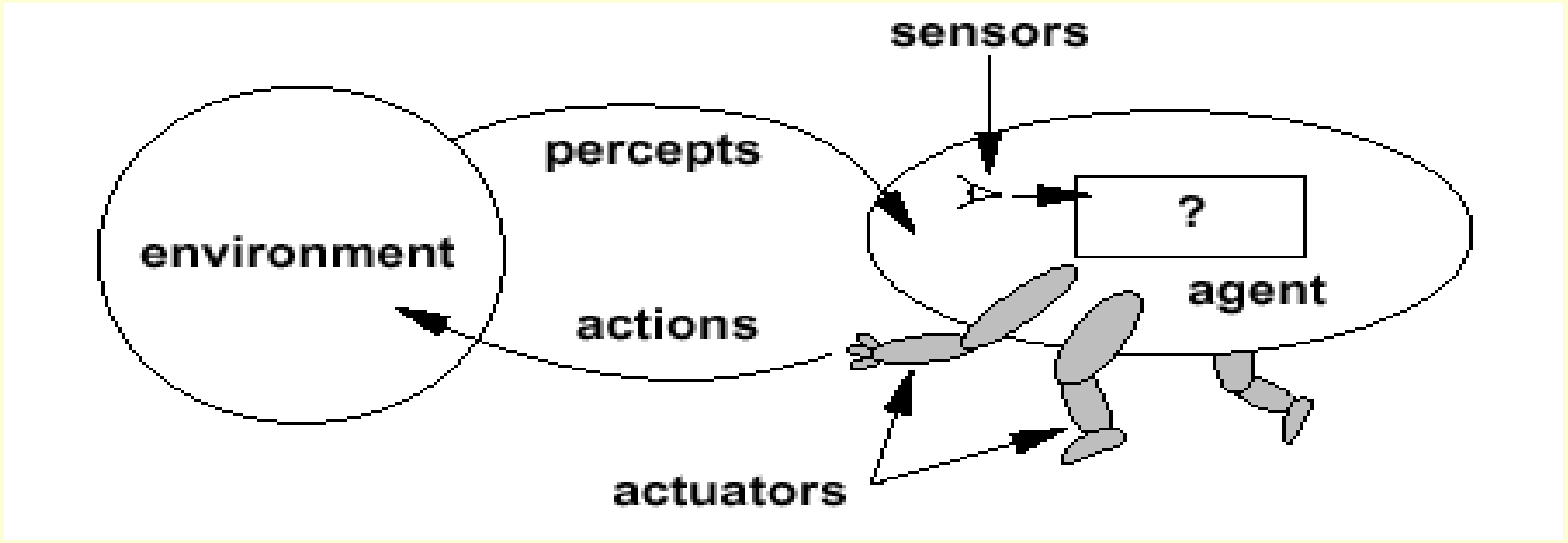
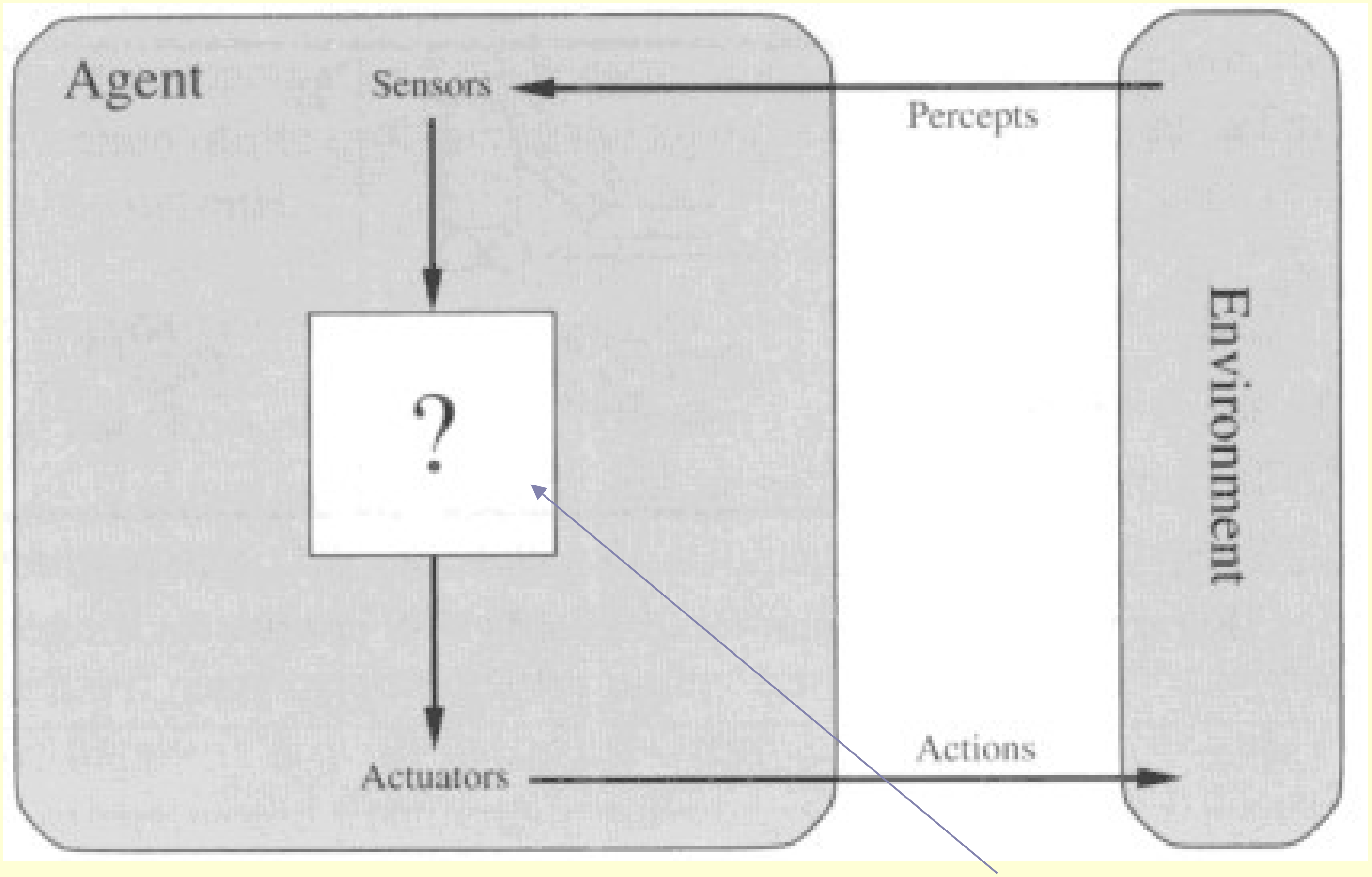


Diagram of an agent



What AI should fill

Simple Terms

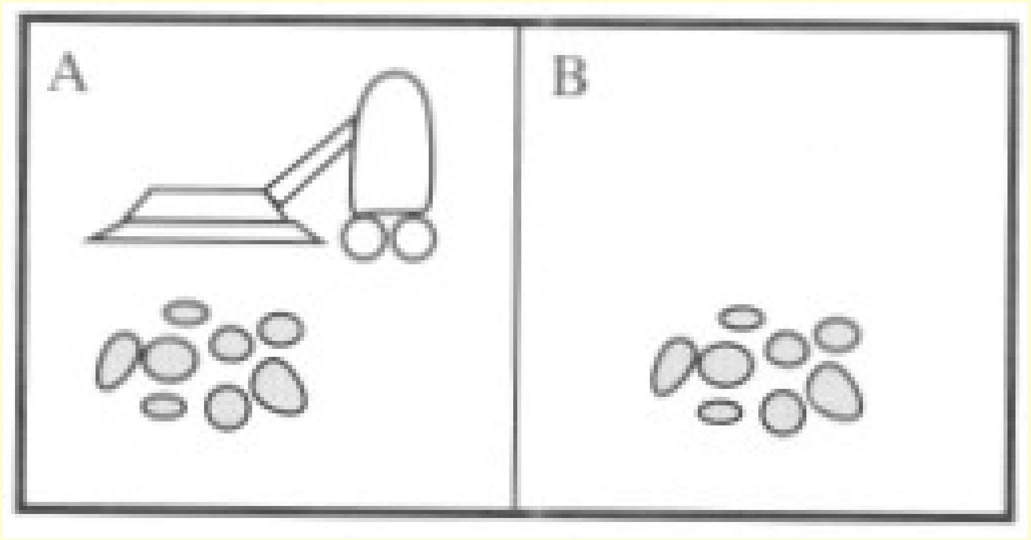
- Percept
  - Agent's perceptual inputs at any given instant
- Percept sequence
  - Complete history of everything that the agent has ever perceived.
- Sensor:
  - device that converts physical events or characteristics into the electrical signals
- Actuator:
  - device that converts the electrical signals into the physical events / characteristics.

Agent function & program

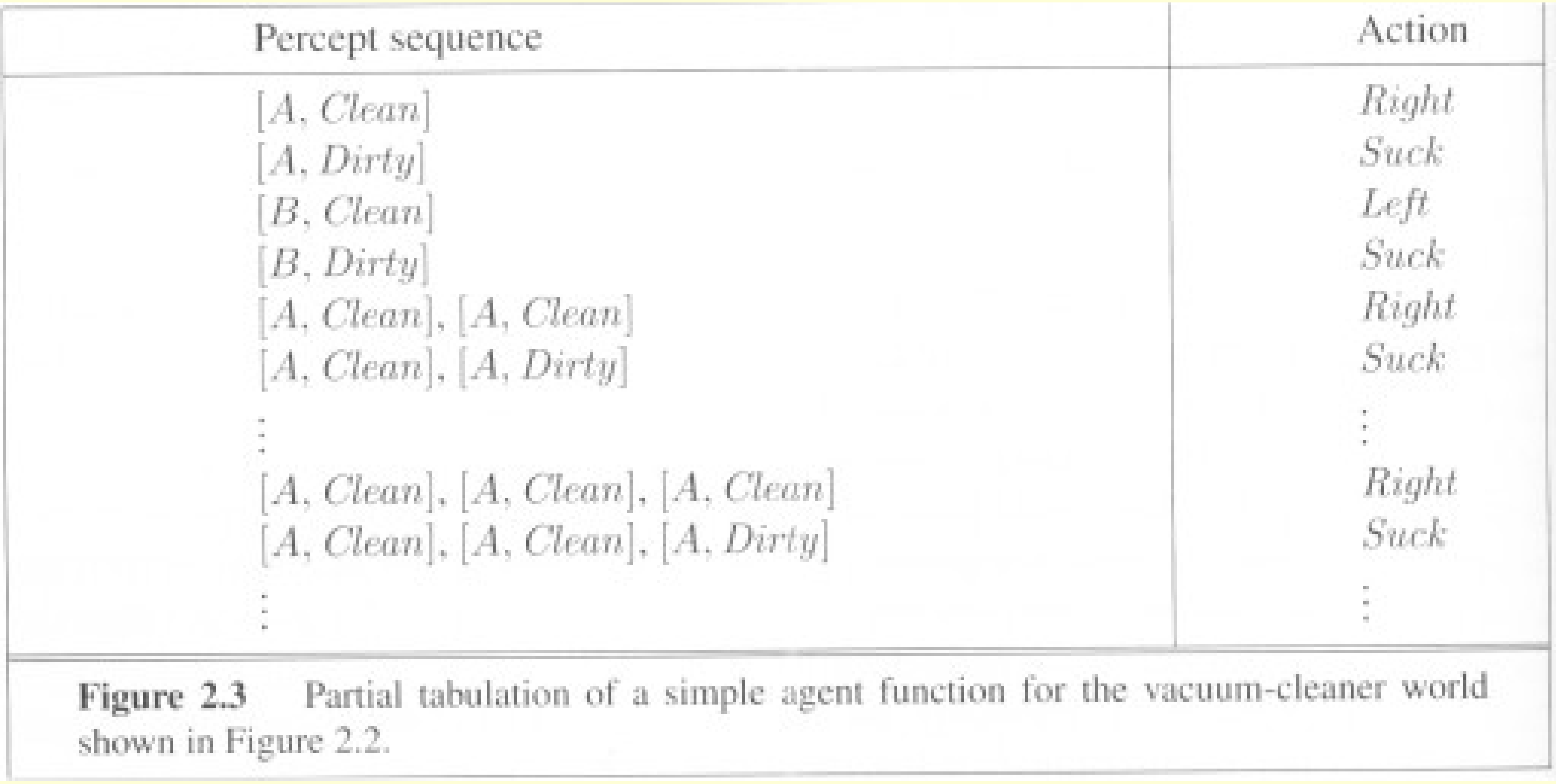
- Agent's behavior is *mathematically* described by
  - **Agent function**
  - A function mapping any given percept sequence to an action
- *Practically* it is described by
  - An **agent program**
  - The real implementation

Vacuum-cleaner world

- Perception: Clean or Dirty? where it is in?
- Actions: Move left, Move right, suck, do nothing



Vacuum-cleaner world



Percept sequence	Action
[*A*, *Clean*]	*Right*
[*A*, *Dirty*]	*Suck*
[*B*, *Clean*]	*Left*
[*B*, *Dirty*]	*Suck*
[*A*, *Clean*], [*A*, *Clean*]	*Right*
[*A*, *Clean*], [*A*, *Dirty*]	*Suck*
⋮	⋮
[*A*, *Clean*], [*A*, *Clean*], [*A*, *Clean*]	*Right*
[*A*, *Clean*], [*A*, *Clean*], [*A*, *Dirty*]	*Suck*
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

Program implements the agent function tabulated in Fig. 2.3

```
Function Reflex-Vacuum-Agent([location,status])
return an action
    If status =  Dirty then return Suck
    else if location = A then return Right
    else if location = B then return left
```

Concept of Rationality

- Rational agent
  - One that does the right thing
  - = every entry in the table for the agent function is correct (rational).
- What is correct?
  - The actions that cause the agent to be most successful
  - So we need ways to measure success.

Performance measure

- Performance measure
  - An objective function that determines
    - How the agent does successfully
    - E.g., 90% or 30% ?
- An agent, based on its percepts
  - → action sequence :

    if desirable, it is said to be performing well.
  - No universal performance measure for all agents

Performance measure

- A general rule:
  - Design performance measures according to
    - What one actually wants in the environment
    - Rather than how one thinks the agent should behave
- E.g., in vacuum-cleaner world
  - We want the floor clean, no matter how the agent behave
  - We don’t restrict how the agent behaves

Rationality

- What is rational at any given time depends on four things:
  - The performance measure defining the criterion of success
  - The agent’s prior knowledge of the environment
  - The actions that the agent can perform
  - The agents’s percept sequence up to now

Rational agent

- For each possible percept sequence,
  - an rational agent should select
    - an action expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has
- E.g., an exam
  - Maximize marks, based on

    the questions on the paper & your knowledge

Rationality

Rationality in an agent is defined by its ability to maximize its performance measure based on its percept sequence, prior knowledge, available actions, and the performance measure itself.

A rational agent does not require omniscience but should gather information, learn from experiences, and adapt its actions accordingly.

Rationality involves choosing actions that enhance future percepts and making informed decisions rather than acting perfectly.

An autonomous rational agent should learn and improve over time, reducing dependence on prior knowledge from its designer.

Example of a rational agent

- Performance measure
  - Awards one point for each clean square
    - at each time step, over 10000 time steps
- Prior knowledge about the environment
  - The geography of the environment
  - Only two squares
  - The *effect* of the actions

Example of a rational agent

- Actions that can perform
  - Left, Right, Suck and NoOp
- Percept sequences
  - Where is the agent?
  - Whether the location contains dirt?

- Under this circumstance, the agent is rational.

Omniscience

- An omniscient agent
  - Knows the *actual* outcome of its actions in advance
  - No other possible outcomes
  - However, impossible in real world
- An example
  - crossing a street but died of the fallen cargo door from 33,000ft → irrational?

Omniscience

- Based on the circumstance, it is rational.
- As rationality maximizes
  - *Expected* performance
- Perfection maximizes
  - *Actual* performance
- Hence rational agents are not *omniscient*.

Learning

- Does a rational agent depend on only current percept?
  - No, the past percept sequence should also be used
  - This is called <u>**learning**</u>
  - After experiencing an episode, the agent
    - should adjust its behaviors to perform better for the same job next time.

Autonomy

- If an agent just relies on the prior knowledge of its designer rather than its own percepts then the agent lacks ***<u>autonomy</u>***

  <u>A rational agent should be autonomous- it should learn what it can to compensate for partial or incorrect prior knowledge.</u>
- E.g., a clock
  - No input (percepts)
  - Run only but its own algorithm (prior knowledge)
  - No learning, no experience, etc.

Software Agents

- Sometimes, the environment may not be the real world
  - E.g., flight simulator, video games, Internet
  - They are all artificial but very complex environments
  - Those agents working in these environments are called
    - Software agent (softbots)
    - Because all parts of the agent are software

Task environments

- Task environments are the problems
  - While the rational agents are the solutions
- Specifying the task environment
  - PEAS description as fully as possible
    - Performance
    - Environment
    - Actuators
    - Sensors

In designing an agent, the first step must always be to specify the task environment as fully as possible.

- automated taxi driver

Task environments

- Performance measure
  - How can we judge the automated driver?
  - Which factors are considered?
    - getting to the correct destination
    - minimizing fuel consumption
    - minimizing the trip time and/or cost
    - minimizing the violations of traffic laws
    - maximizing the safety and comfort, etc.

Task environments

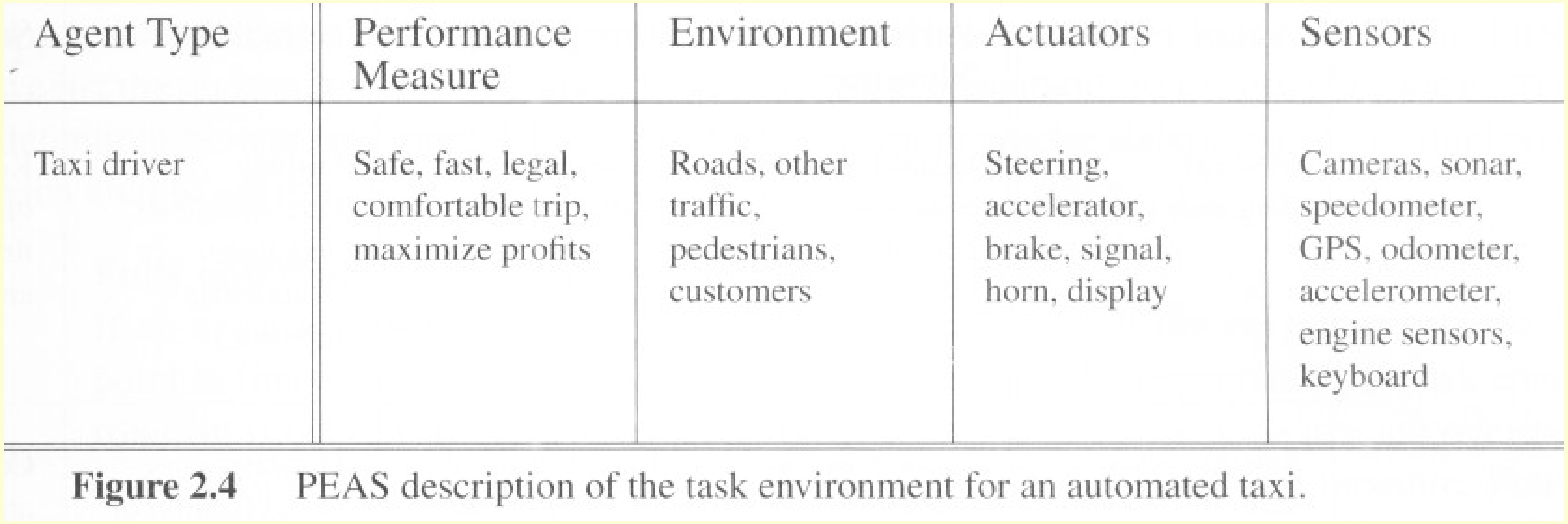
- Environment
  - A taxi must deal with a variety of roads
  - Traffic lights, other vehicles, pedestrians, stray animals, road works, police cars, etc.
  - Interact with the customer

Task environments

- Actuators (for outputs)
  - Control over the accelerator, steering, gear shifting and braking
  - A display to communicate with the customers
- Sensors (for inputs)
  - Detect other vehicles, road situations
  - GPS (Global Positioning System) to know where the taxi is
  - Many more devices are necessary

Task environments

- A sketch of automated taxi driver



Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

Properties of task environments

- Fully observable vs. Partially observable
  - If an agent's sensors give it access to the complete state of the environment at each point in time then the environment is <u>effectively and fully observable</u>
    - if the sensors detect all aspects
    - That are relevant to the choice of action
    - Chess

- Partially observable - driving

  An environment might be Partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

  <u>**Example:**</u>

  - A local dirt sensor of the cleaner cannot tell
  - Whether other squares are clean or not

Properties of task environments

- Deterministic vs. stochastic
  - next state of the environment Completely determined by the current state and the actions executed by the agent, then the environment is deterministic, otherwise, it is Stochastic.   - chess
  - Strategic environment: deterministic except for actions of other agents

    -Cleaner and taxi driver are:

    - Stochastic because of some un observable aspects → noise or unknown

Properties of task environments

- Episodic vs. sequential
  - An episode = agent’s single pair of <u>perception & action</u>
  - The quality of the agent’s action does not depend on other episodes
    - Every episode is independent of each other
  - Episodic environment is simpler - image analysis
    - The agent does not need to think ahead
- Sequential
  - Current action may affect all future decisions

    -Ex. Taxi driving and chess.

Properties of task environments

- Static vs. dynamic
  - A dynamic environment is always changing over time -traffic
    - E.g., the number of people in the street
  - While static environment - chess
    - E.g., the destination
- Semi-dynamic
  - environment is not changed over time
  - but the agent’s performance score does

Properties of task environments

- Discrete vs. continuous

If an environment consists of a finite number of actions that can be deliberated in the environment to obtain the output, it is said to be a discrete environment.

The game of chess is discrete as it has only a finite number of moves. The number of moves might vary with every game, but still, it’s finite.

E.g., Chess game

- Continuous: Taxi driving

Properties of task environments

- Single agent VS. multiagent
  - Playing a crossword puzzle – single agent
  - Chess playing – two agents
  - Competitive multiagent environment
    - Chess playing
  - Cooperative multiagent environment
    - Automated taxi driver
    - Avoiding collision

Properties of task environments

- Known vs. unknown

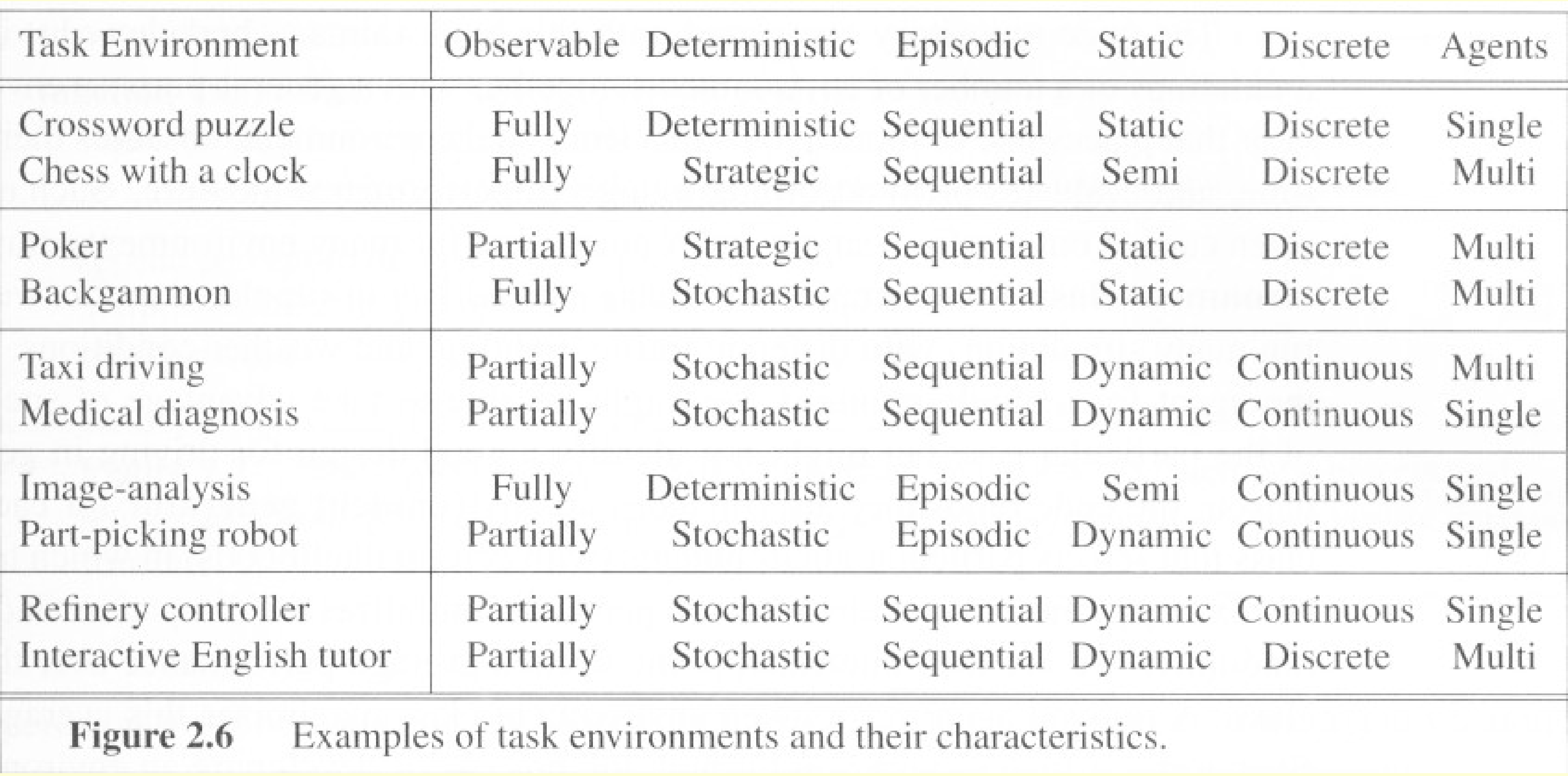
  This distinction refers not to the environment itslef but to the agent’s state of knowledge about the environment.

  -In known environment, the outcomes for all actions are given. (example: solitaire card games).

  - If the environment is unknown, the agent will have to learn how it works in order to make good decisions.

  Online Marketplaces: In an e-commerce platform, sellers and buyers (agents) may have varying behaviors, preferences, and strategies. An agent (like a recommendation system) must adapt to these unknown factors.

Examples of task environments



Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Figure 2.6 Examples of task environments and their characteristics.

Structure of agents

Structure of agents

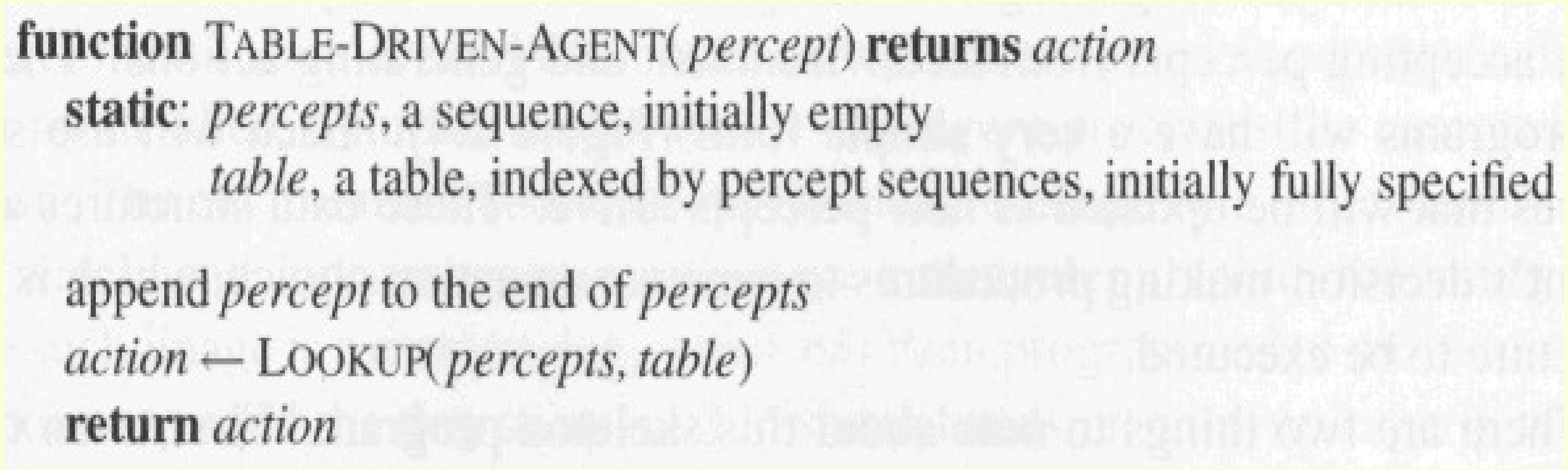
- Agent = architecture + program
  - Architecture = some sort of computing device (sensors + actuators)
  - (Agent) Program = some function that implements the agent mapping = “?”
  - Agent Program = Job of AI

Agent programs

- Input for Agent Program
  - Only the current percept
- Input for Agent Function
  - The entire percept sequence
  - The agent must remember all of them
- Implement the agent program as
  - A look up table (agent function)

Agent programs

- Skeleton design of an agent program



```
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
```

Agent Programs

- P = the set of possible percepts
- T= lifetime of the agent
  - The total number of percepts it receives
- Size of the look up table $\sum_{t=1}^{T} |P|^t$
- Consider playing chess
  - P =10, T=150
  - Will require a table of at least 10^{150} entries

Agent programs

- Despite of huge size, look up table does what we want.
- The key challenge of AI
  - Find out how to write programs that, to the extent possible, produce rational behavior
    - From a small amount of code
    - Rather than a large amount of table entries
  - E.g., a five-line program of Newton’s Method
  - V.s. huge tables of square roots, sine, cosine, …

Types of agent programs

- Four types
  - Simple reflex agents
  - Model-based reflex agents
  - Goal-based agents
  - Utility-based agents

Simple reflex agents

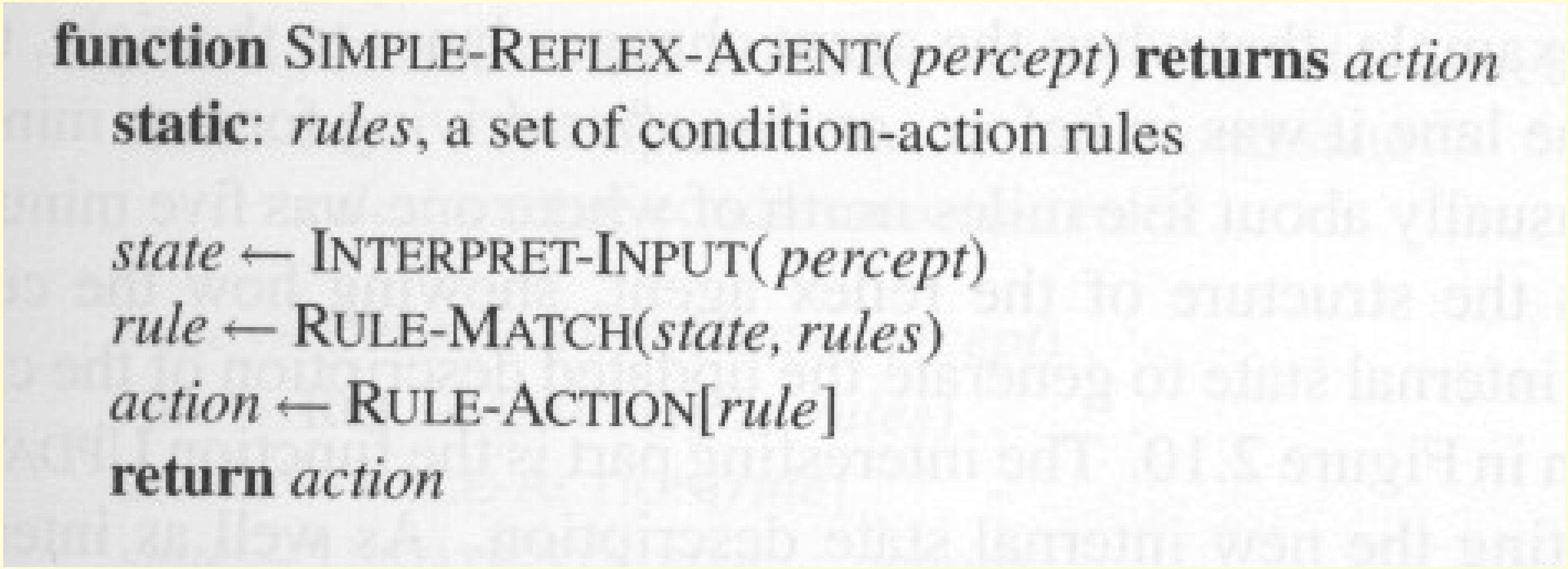
- It uses just ***condition-action rules***

Simple reflex agents operate based on the current percept, ignoring the rest of the percept history.

They follow a condition-action rule: if a certain condition is true, then perform a specific action

- The rules are like the form “if … then …”
- efficient but have narrow range of applicability
- Because knowledge sometimes cannot be stated explicitly
- Work only
  - if the environment is fully observable

Simple reflex agents



```
function SIMPLE-REFLEX-AGENT(percept) returns action
    static: rules, a set of condition-action rules

    state ← INTERPRET-INPUT(percept)
    rule ← RULE-MATCH(state, rules)
    action ← RULE-ACTION[rule]
    return action
```

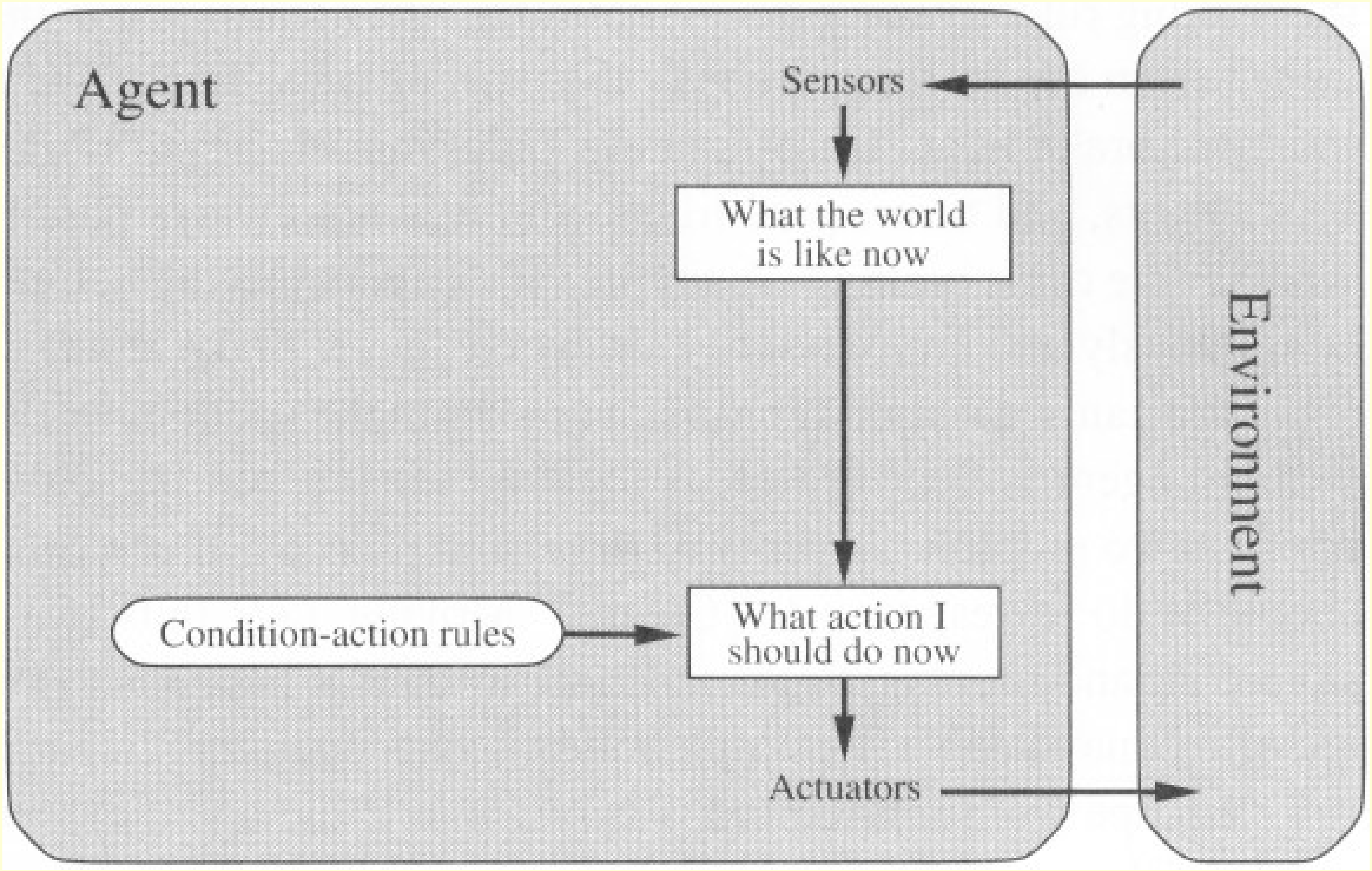
Advantages:

Simple to design and implement.
Fast decision-making.

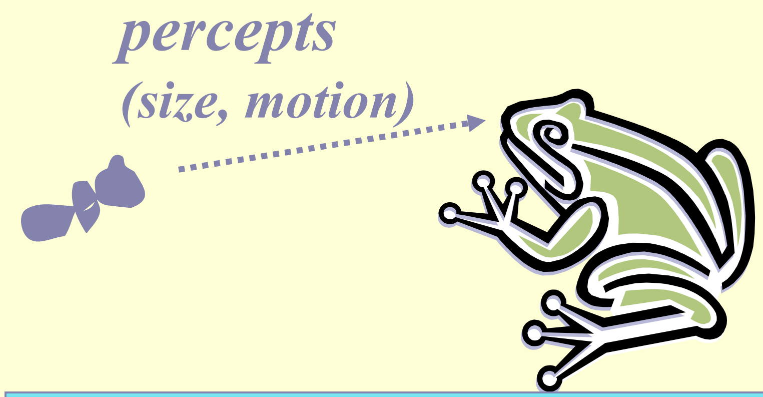
Disadvantages:

Limited intelligence and adaptability.
Cannot handle situations where the correct action depends on the history of percepts.

Simple reflex agents (2)



A Simple Reflex Agent in Nature



RULES:

(1) If small moving object,
then activate SNAP

(2) If large moving object,
then activate AVOID and inhibit SNAP

ELSE (not moving) then NOOP

needed for completeness

Action: SNAP or AVOID or NOOP(No Operation)

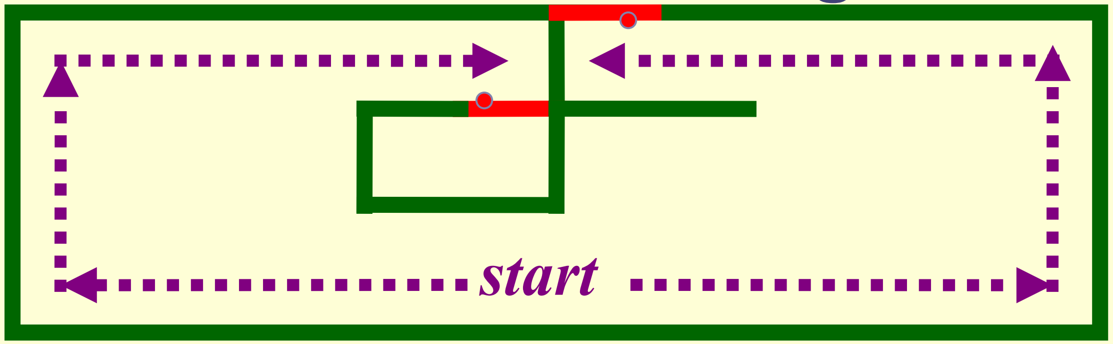
Model-based Reflex Agents

- For the world that is partially observable
  - the agent has to keep track of an internal state
    - That depends on the percept history
    - Reflecting some of the unobserved aspects
    - E.g., driving a car and changing lane
- Requiring two types of knowledge
  - How the world evolves independently of the agent
  - How the agent's actions affect the world

Example Table Agent With Internal State

IF	THEN
Saw an object ahead, and turned right, and it's now clear ahead	Go straight
Saw an object Ahead, turned right, and object ahead again	Halt
See no objects ahead	Go straight
See an object ahead	Turn randomly

Example Reflex Agent With Internal State: **Wall-Following**

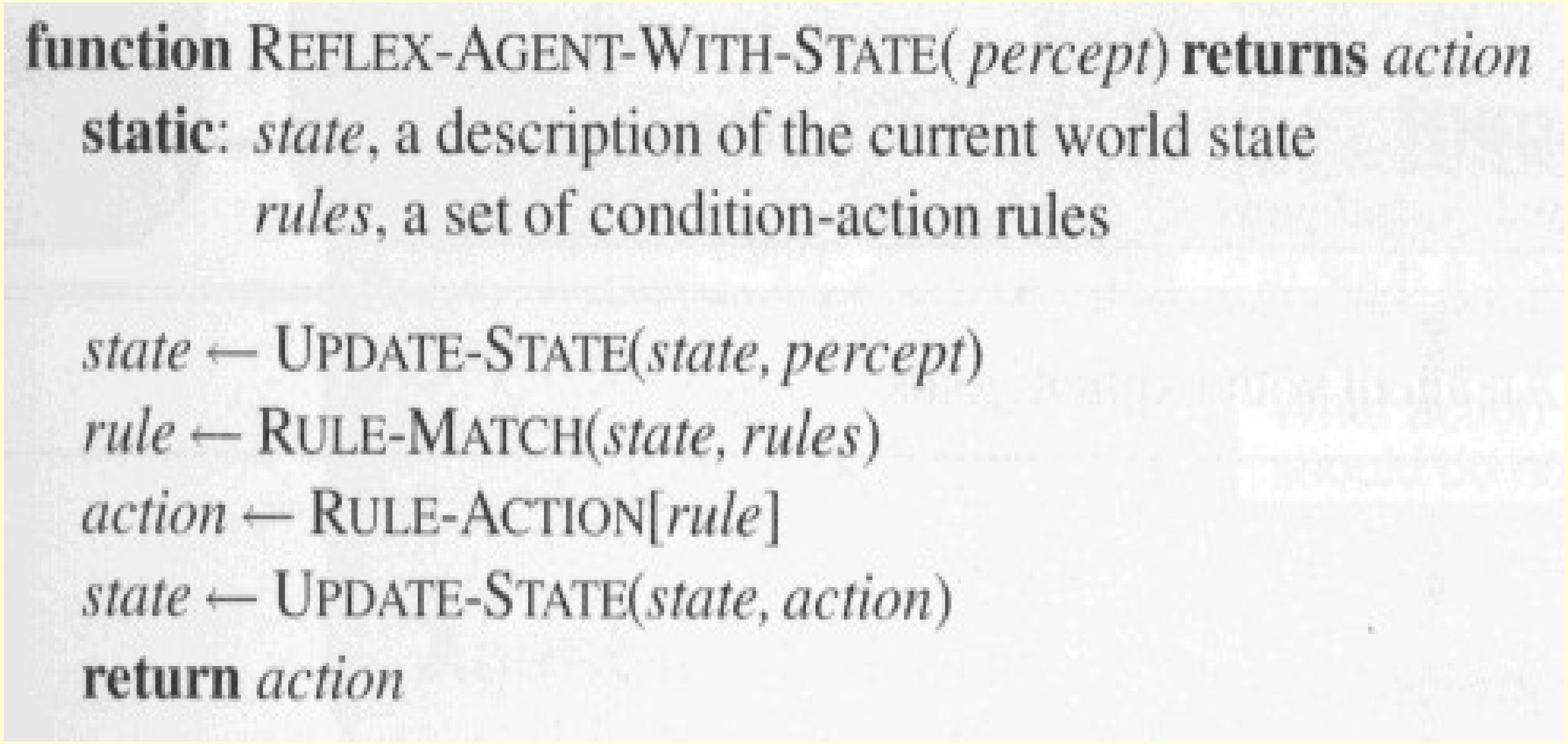


Actions: left, right, straight, open-door

Rules:

1. If open(left) & open(right) and open(straight) then choose randomly between right and left
2. If wall(left) and open(right) and open(straight) then straight
3. If wall(right) and open(left) and open(straight) then straight
4. If wall(right) and open(left) and wall(straight) then left
5. If wall(left) and open(right) and wall(straight) then right
6. If wall(left) and door(right) and wall(straight) then open-door
7. If wall(right) and wall(left) and open(straight) then straight.
8. (Default) Move randomly

Model-based Reflex Agents

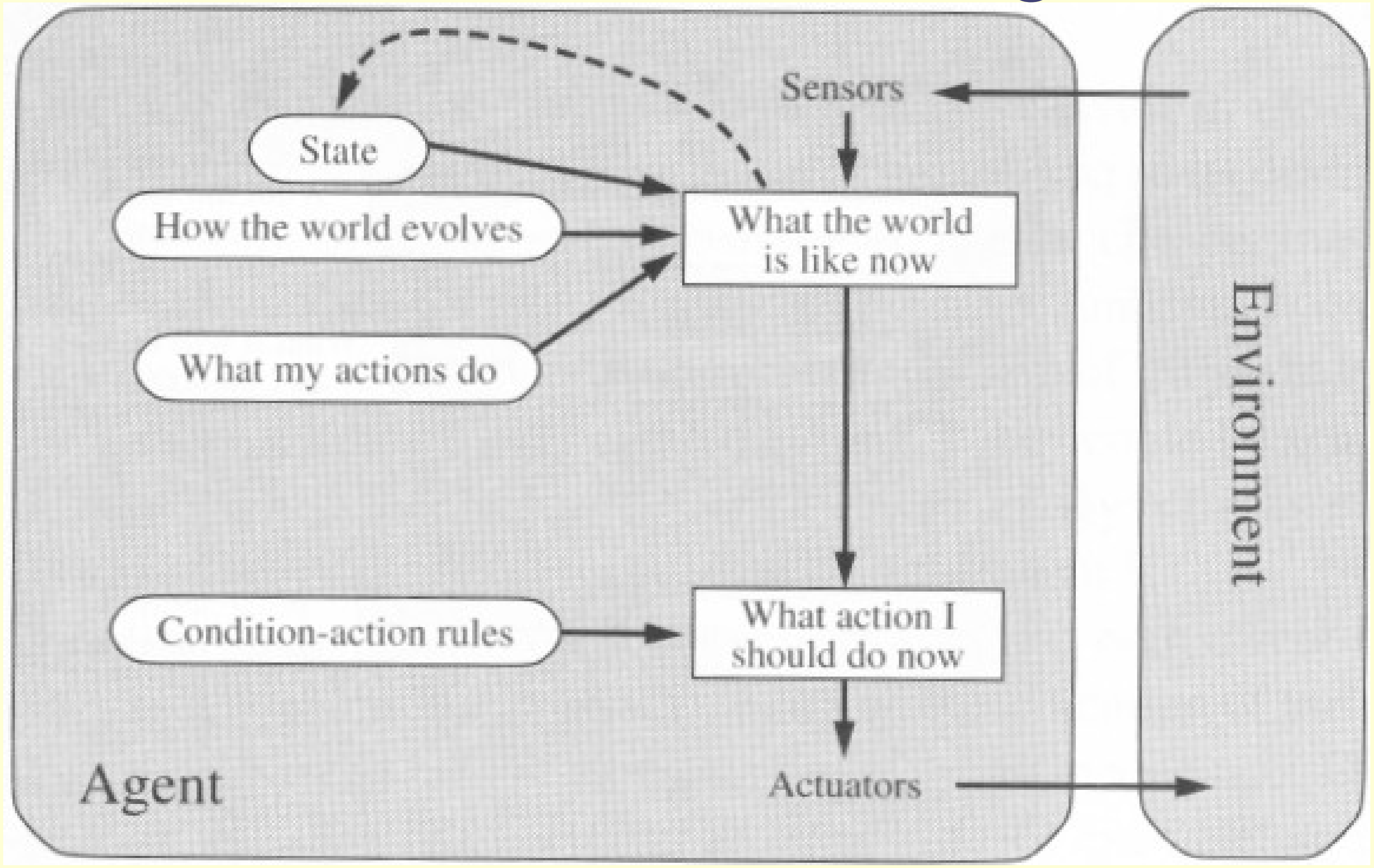


```
function REFLEX-AGENT-WITH-STATE(percept) returns action
    static: state, a description of the current world state
            rules, a set of condition-action rules

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

The agent is with memory

Model-based Reflex Agents



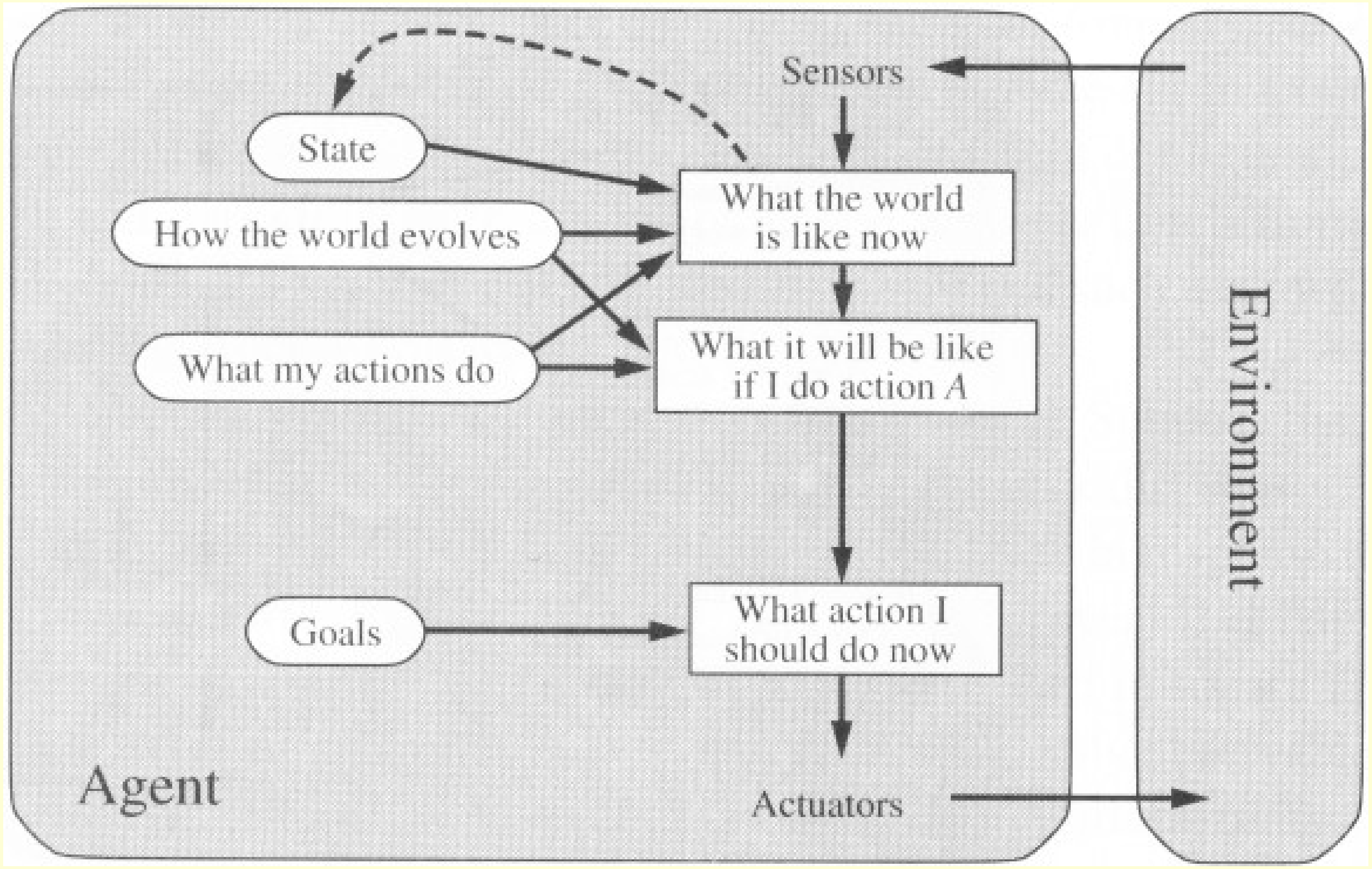
Goal-based agents

- Current state of the environment is always not enough
- The goal is another issue to achieve
  - Judgment of rationality / correctness
- Actions chosen → goals, based on
  - the current state
  - the current percept

Goal-based agents

- Conclusion
  - Goal-based agents are less efficient
  - but more flexible
    - Agent ← Different goals ← different tasks
  - Search and planning
    - two other sub-fields in AI
    - to find out the action sequences to achieve its goal

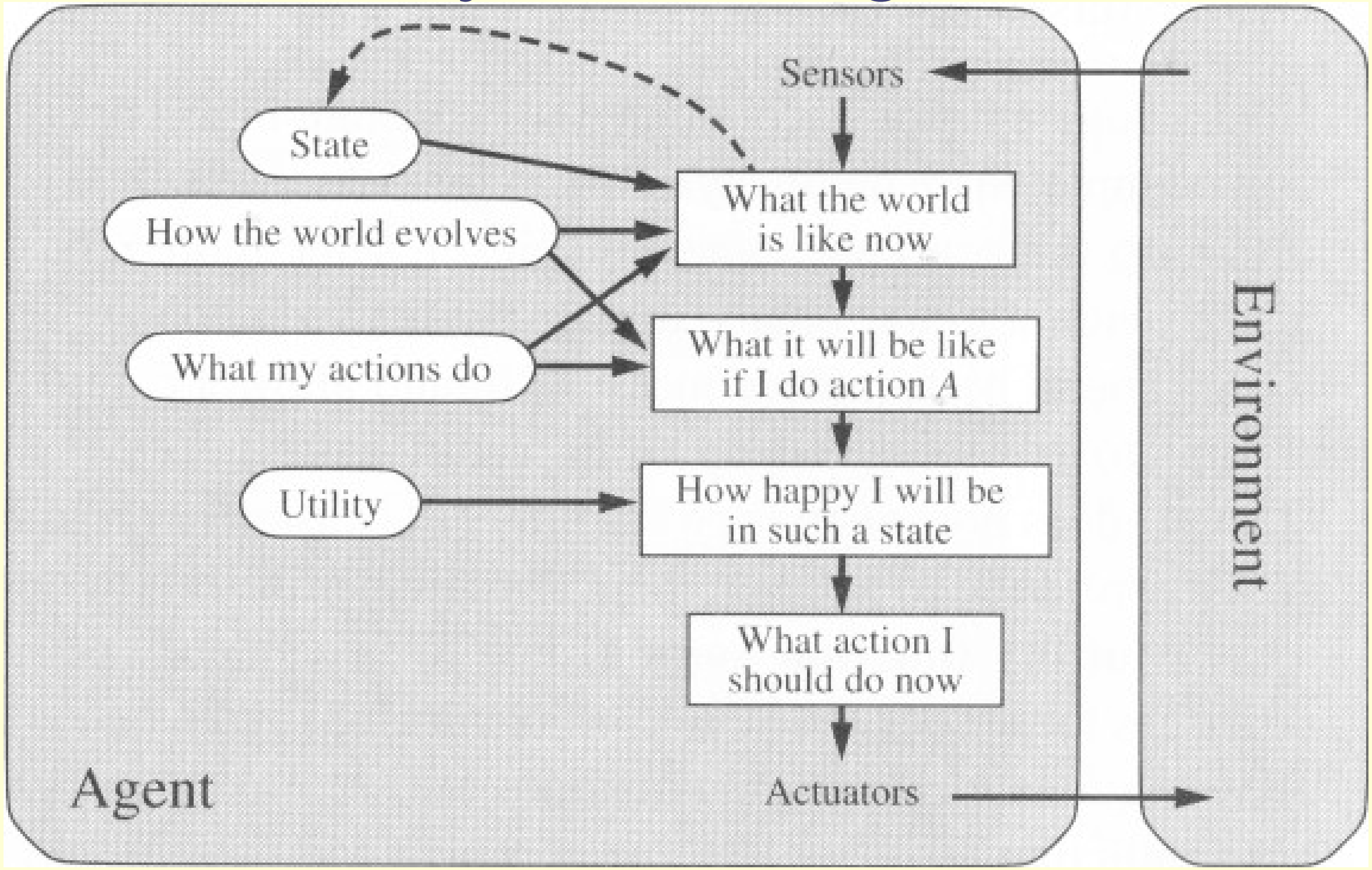
Goal-based agents


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Utility-based agents

- Goals alone are not enough
  - to generate **high-quality** behavior
  - E.g. meals in Canteen, good or not ?
- Many action sequences → the goals
  - some are better and some worse
  - If goal means success,
  - then **utility** means the degree of success (how successful it is)

Utility-based agents


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Actuators
Agent
Environment

Utility-based agents

- it is said state A has higher utility
  - If state A is more preferred than others
- Utility is therefore a function
  - that maps a state onto a real number
  - the degree of success

Utility-based agents

- Utility has several advantages:
  - When there are conflicting goals,
    - Only some of the goals but not all can be achieved
    - utility describes the appropriate trade-off
  - When there are several goals
    - None of them are achieved **certainly**
    - utility provides a way for the decision-making

Learning Agents

- After an agent is programmed, can it work immediately?
  - No, it still need teaching
- In AI,
  - Once an agent is done
  - We teach it by giving it a set of examples
  - Test it by using another set of examples
- We then say the agent learns
  - A learning agent

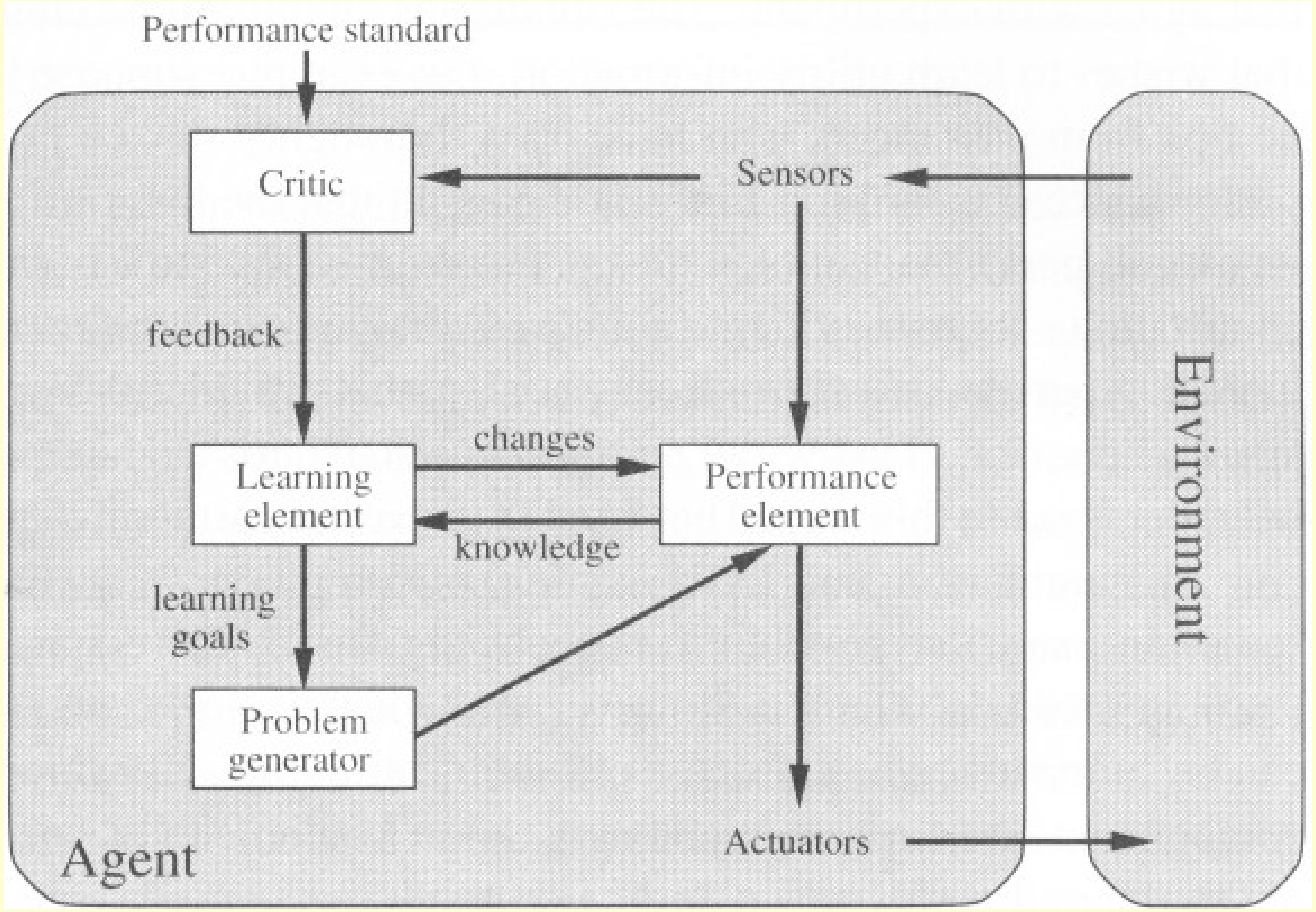
Learning Agents

- Four conceptual components
  - Learning element
    - Making improvement
  - Performance element
    - Selecting external actions
  - Critic
    - Tells the Learning element how well the agent is doing with respect to fixed performance standard.

(Feedback from user or examples, good or not?)

  - Problem generator
    - Suggest actions that will lead to new and informative experiences.

Learning Agents



Key points:

Formulate, Search, Execute: This design involves formulating a goal, searching for a solution, and then executing the solution. Once a solution is found, the agent performs the recommended actions.

Problem-Solving Agents: These agents assume a static, observable, discrete, and deterministic environment, making it easier to formulate and solve problems without considering environmental changes.

Problem Definition: A problem is defined by four components:

Initial State: The starting point of the agent.
Successor Function: Defines the possible actions and resulting states.
Goal Test: Determines if a state is a goal state.
Path Cost: Measures the cost of a path from the initial state to the goal state.
State Space: The combination of the initial state and successor function defines the state space, which can be visualized as a graph with nodes (states) and arcs (actions).

Solution: A solution is a path from the initial state to a goal state, with the optimal solution having the lowest path cost.

Goal State: (2, 2)
Actions: "right", "up"
The solution found by the search might be a sequence of actions: seq = ["right", "right", "up", "up"]
Execute Actions:

First Iteration:

seq = ["right", "right", "up", "up"]
action <- FIRST(seq): The first action is "right".
Execute "right" to move to (1, 0).
seq <- REST(seq): Update sequence to seq = ["right", "up", "up"].
Second Iteration:

seq = ["right", "up", "up"]
action <- FIRST(seq): The first action is "right".
Execute "right" to move to (2, 0).
seq <- REST(seq): Update sequence to seq = ["up", "up"].
Third Iteration:

seq = ["up", "up"]
action <- FIRST(seq): The first action is "up".
Execute "up" to move to (2, 1).
seq <- REST(seq): Update sequence to seq = ["up"].
Fourth Iteration: