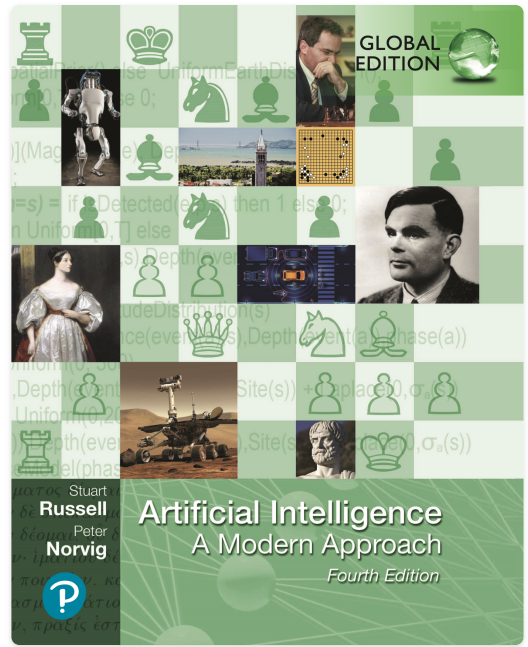
AI: A Modern Approach - 1

bifnudozhao@tencent.com

Outline

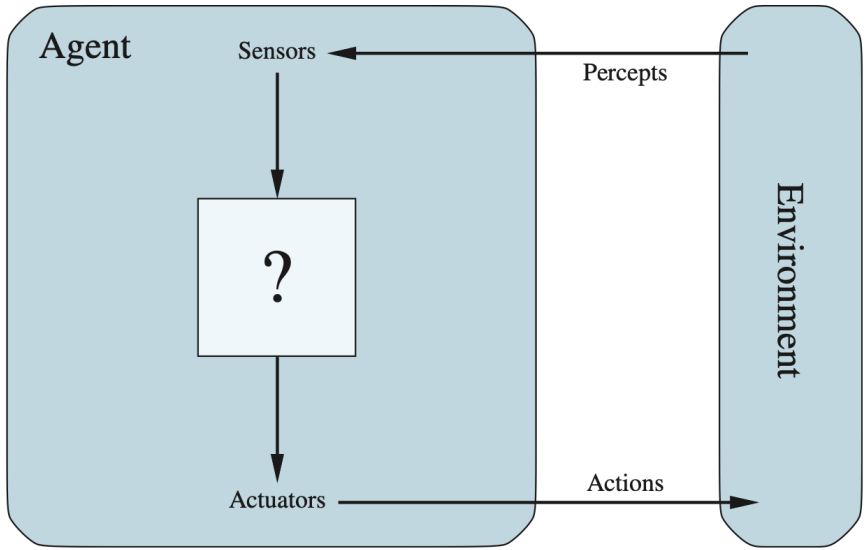
- Intelligent Agents
- Solving Problems by Searching
- Demos



Artificial Intelligence: A Modern Approach

by Stuart Russell, Peter Norvig

Intelligent Agents



An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.

The Nature of Environment

Task environments, which are essentially the “problems” to which rational agents are the “solutions”.

Specifying the Task Environment

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi Driver	Save, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Camera, radar, speedometer, GPS, engine sensors

Properties of Task Environment

- **Fully observable vs. partially observable**: Whether an agent's sensors give it access to the complete state of the environment at each point in time.
- **Single-agent vs. multiagent**: Chess is a competitive multiagent environment. Auto-driving is a partially cooperative multiagent environment.
- **Deterministic vs. nondeterministic**: Whether the next state of the environment is completely determined by the current state and the action executed by the agents.
- **Episodic vs. sequential**: Whether the agent's experience is divided into atomic episodes. In sequential environments, the current decision could affect all future decisions.
- **Static vs. dynamic**: Whether the environment can change while an agent is deliberating.
- **Discrete vs. continuous**: The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.
- **Known vs. unknown**: whether the outcomes for all actions are given.

The Structure of Agents

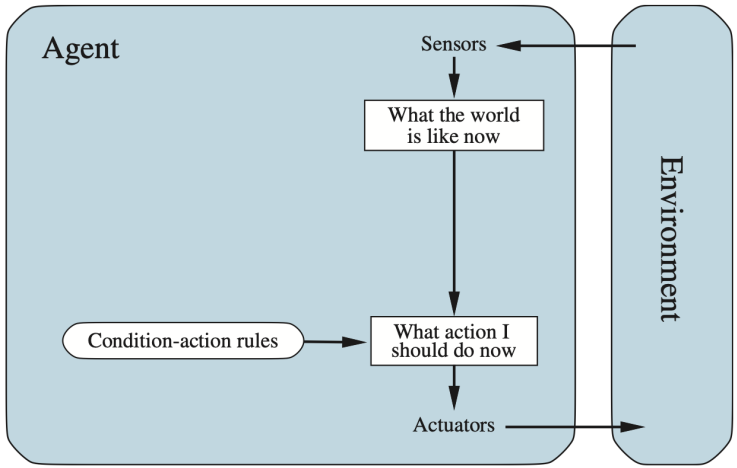
The job of AI is to design an **agent program** that implements the agent function —— the mapping from percepts to actions. The program is assumed to run on some sort of computing device with physical sensors and actuators —— which is called **agent architecture**.

$$agent = architecture + program$$

Agent Programs

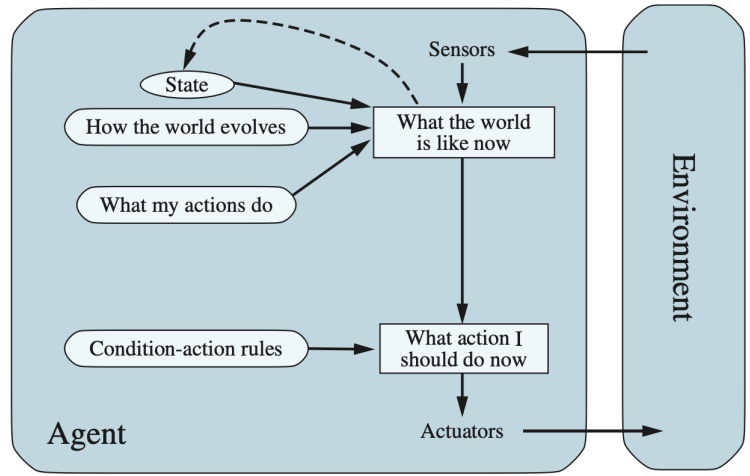
```
const percepts: Percept[] = []
const table: { [perceptsId: string]: Action } = {}

function TABLE_DRIVEN_AGENT(percept): Action {
  percepts.push(percept)
  const action = LOOK_UP(percepts, table)
  return action
}
```

Simple Reflex Agents



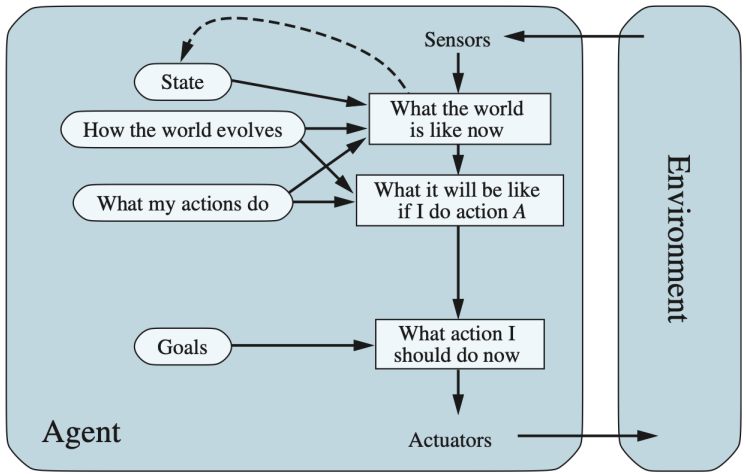
These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

Model-based Reflex Agents



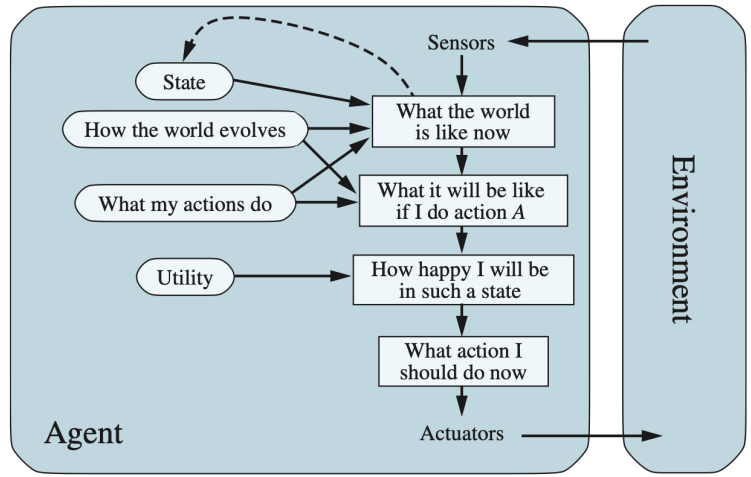
The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

Goal-Based Agents



As well as a current state description, the agent needs some sort of goal information that describes situations that are desirable. **Search** and **planning** are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

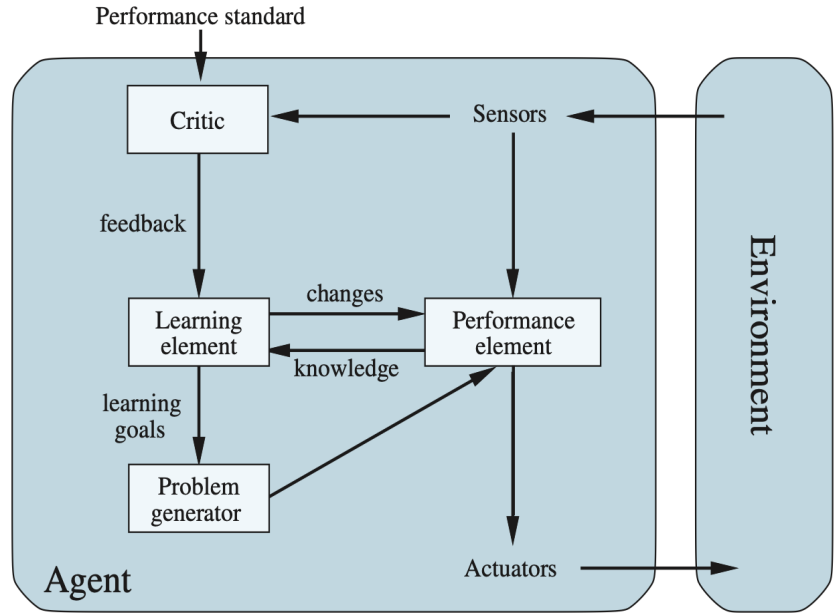
Utility-Based Agents



Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to the quality of being useful. An agent's **utility function** is essentially an internalization of the performance measure.

General-Learning Agents

- **Learning element**: responsible for making improvements
- **Performance element**: responsible for selecting external actions
- **Critic element**: tells the learning elemenet how well the agent is doing respect to a fixed performance standard
- **Problem Generator**: responsible for suggesting actions that will lead to new and informative experiences



A general learning agent. The “performance element” box represents what we have previously considered to be the whole agent program. Now, the “learning element” box gets to modify that program to improve its performance.

Problem-Solving Agents

- **Goal formulation**: Goals organize behavior by limiting the objectives and hence the actions to be considered.
- **Problem formulation**: The agent devises a description of the states and actions necessary to reach the goal —— an abstract model of the relevant part of the world.
- **Search**: The agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution**.
- **Execution**: The agent can now execute the actions in the solution, one at a time.

Search Problems and Solutions

- **state space**: A set of possible states that the environment can be in.
- **initial state**: that the agent starts in.
- A set of one or more **goal states**.
- The **actions** available to the agent. Given a state s, $\text{ACTIONS}(s)$ returns a finite set of actions that can be executed in s. We say that each of these actions is **applicable** in s.
- A **transition model**, which describes what each action does. $\text{RESULT}(s, a)$ returns the state that results from doing action a in state s.
- An **action cost function**, denoted by $\text{ACTION-COST}(s, a, s')$, that gives the numeric cost of applying action a in state s to reach s'.

General Problem Definition

```
export class Problem<State extends StateNode> {
  initial: State
  goal: State
  actions: Action[]
  states: State[]

  // determine whether the given node is goal
  isGoal(node: State): boolean

  // takes an action on the given state and gives a result
  result(node: State, action: Action): State

  // calculates the action cost
  actionCost(before: State, action: Action, after: State): number

  // expands a node
  expand(node: State): State[]
}
```

Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure. One type of algorithms superimposes a **search tree** over the state space graph, forming various paths from the initial state, trying to find a path that reaches a goal state.

Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the three corresponds to the initial state of the problem.

- Uninform Search Strategies
  - Breadth-First Search
  - Dijkstra's Search
  - Depth-First Search
  - Bidirectional Search
- Informed Search Strategies
  - Greedy Best-First Search
  - A* Search
  - Weighted A* Search

Best-First Search

A general approach for expand the frontier is called best-first search, in which a node n is chosen, with minimum value fo some evaluation function, $f(n)$.

By employing different $f(n)$ functions, we get different specific algorithms.

```
export function* bestFirstSearch<State>(
  problem: Problem<State>, f: any
) {
  let node = problem.initial
  const frontier = new PriorityQueue<State>(f)
  frontier.push(node)
  const reached: any = {}
  reached[node.id] = true

  while (!frontier.isEmpty) {
    node = frontier.shift()
    if (problem.isGoal(node)) return node

    const children = problem.expand(node)
    for (let i = 0; i < children.length; ++i) {
      const child = children[i]
      if (problem.isGoal(child)) return child
      if (!reached[child.id]) {
        frontier.add(child)
        reached[child.id] = true
      }

      yield { frontier, reached }
    }
  }
}
```

Summary of Different Search Algorithms

Searches can be considered that evaluates states by combining g and h in various ways.

Algorithm	Evaluation Function $f(n)$	Weight
A* search	$g(n) + h(n)$	$(W = 1)$
Dijkstra's search	$g(n)$	$(W = 0)$
Greedy best-first search	$h(n)$	$(W = \infty)$
Weighted A* search	$g(n) + W \times h(n)$	$(1 < W < \infty)$

Important

The priority queue is maintained by the evaluation function $f(n)$

```
1   const frontier = new PriorityQueue<State>(f)
```

Examples - 2D Map Search

Definition of a 2D map search problem.

The 2D map search problem extends the basic problem.

```
class MapSearch2D extends Problem<MapSearch2DState>
```

Action generator.

```
move(dir: Vector2) {
  return (node: MapSearch2DState) => {
    const newNode = new MapSearch2DState({ state: new Vector2(node.state.x, node.state.y).add(dir) })
    if (this.isOutside(newNode) || this.barrier[newNode.getId()]) return
    return newNode
  }
}
```

If diagonal move is not allow, then generate `moveUp`, `moveLeft`, `moveRight`, `moveDown`. If diagonal move is allow, then generate diagonal moves.

```
  setAllowDiagonal(allowDiagonal: boolean) {
    this.allowDiagonal = allowDiagonal
    this.actions = _.union([
      this.move(new Vector2(0, -1)), // up
      this.move(new Vector2(-1, 0)), // left
      this.move(new Vector2(1, 0)),  // right
      this.move(new Vector2(0, 1)),  // down
    ], this.allowDiagonal ? [
      this.move(new Vector2(-1, -1)), // up left
      this.move(new Vector2(-1, 1)), // down left
      this.move(new Vector2(1, -1)), // up right
      this.move(new Vector2(1, 1)), // down right
    ] : [])
  }
```

Examples - 8 Puzzle

Definition of a 8 puzzle problem solver.

The 8 puzzle problem solver extends the basic problem.

```
class EightPuzzle extends Problem<EightPuzzleState>
```

Basic action generator. Swaps the empty slot with a target slot.

```
move(dir: Vector2) {
  return (node: EightPuzzleState) => {
    const newSlot = node.state.slot.add(dir)
    if (this.isOutside(newSlot)) return
    const board = _.cloneDeep(node.state.board)
    board[node.state.slot.x][node.state.slot.y] = board[newSlot.x][newSlot.y]
    board[newSlot.x][newSlot.y] = '*'
    const newNode = new EightPuzzleState({ state: { board, slot: newSlot } })
    return newNode
  }
}
```

Demo