INTelligent Agents

Chapter 2

Outline

◊ PAGE (Percepts, Actions, Goals, Environment)
◊ Environment types
◊ Agent functions and programs
◊ Agent types
◊ Vacuum world
◊ we want to design rational agents - how to define success
Must first specify the setting for intelligent agent design

Consider, e.g., the task of designing an automated taxi:

**Percepts**
- video, accelerometers, gauges, engine sensors, keyboard, GPS, ...

**Actions**
- steer, accelerate, brake, horn, speak/display, ...

**Goals**
- safety, reach destination, maximize profits, obey laws, passenger comfort, ...

**Environment**
- US urban streets, freeways, traffic, pedestrians, weather, customers, ...

◊ Internet Shopping Agents - percepts, actions, goals, environment
Rational agents

Without loss of generality, “goals” specifiable by performance measure defining a numerical value for any environment history (modulo what can the agent perceive)

Rational action: whichever action maximizes the expected value of the performance measure given the percept sequence to date

Rational $\neq$ omniscient
Rational $\neq$ clairvoyant
Rational $\neq$ successful

♦ accessible vs. unaccessible ♦ deterministic vs. undeterministic ♦ episodic vs. noepisodic ♦ static vs. dynamic ♦ discrete vs. continuous

Environment types

<table>
<thead>
<tr>
<th>Accessible??</th>
<th>Deterministic??</th>
<th>Episodic??</th>
<th>Static??</th>
<th>Discrete??</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solitaire</td>
<td>Backgammon</td>
<td>Internet shopping</td>
<td>Taxi</td>
<td></td>
</tr>
</tbody>
</table>
Environment types

<table>
<thead>
<tr>
<th></th>
<th>Solitaire</th>
<th>Backgammon</th>
<th>Internet shopping</th>
<th>Taxi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible??</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Deterministic??</td>
<td>Yes</td>
<td>No</td>
<td>Partly</td>
<td>No</td>
</tr>
<tr>
<td>Episodic??</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Static??</td>
<td>Yes</td>
<td>Semi</td>
<td>Semi</td>
<td>No</td>
</tr>
<tr>
<td>Discrete??</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

The environment type largely determines the agent design.

The real world is (of course) inaccessible, stochastic, sequential, dynamic, continuous.

Agent functions and programs

An agent is completely specified by the agent function mapping percept sequences to actions.

(In principle, one can supply each possible sequence to see what it does. Obviously, a lookup table would usually be immense.)

Aim: find a way to implement the rational agent function concisely.

An agent program takes a single percept as input, keeps internal state:

```plaintext
function SKELETON-AGENT( percept) returns action
  static: memory, the agent’s memory of the world
  memory ← UPDATE-MEMORY(memory, percept)
  action ← CHOOSE-BEST-ACTION(memory)
  memory ← UPDATE-MEMORY(memory, action)
  return action
```
Agent types

Four basic types in order of increasing generality:
- simple reflex agents
- reflex agents with state
- goal-based agents
- utility-based agents

Simple reflex agents

◊ lookup table out of question get-input, rule-match, rule-action
Reflex agents with state

Agent

State
How the world evolves
What my actions do
Condition-action rules

Environment

Sensors
What the world is like now

Effectors
What action I should do now

get-input, update-state, rule-match, rule-action, update-state

Goal-based agents

Agent

State
How the world evolves
What my actions do
Goals

Environment

Sensors
What the world is like now

Effectors
What action I should do now

What it will be like if I do action A

needs a goal, involves some search, planning ahead
Utility-based agents

Agent

- State
- Sensors
- What the world is like now
- What it will be like if I do action A
- Utility
- How happy I will be in such a state
- What action I should do now
- Effectors

Environment

explicit utility - measure how happy agent is to be in a state

Chapter 2  13

The vacuum world

Percepts (<bump> <dirt> <home>)

Actions shutoff forward suck (turn left) (turn right)

Goals (performance measure on environment history)
- +100 for each piece of dirt cleaned up
- -1 for each action
- -1000 for shutting off away from home

Environment
- grid, walls/obstacles, dirt distribution and creation, agent body
- movement actions work unless bump into wall
- suck actions put dirt into agent body (or not)


Chapter 3, Sections 1–5  14
Problem solving and search

Chapter 3, Sections 1–5

Outline

♦ Problem-solving agents (problem, goal and means of achieving it)
♦ Problem types
♦ Problem formulation
♦ Example problems
♦ autonomous agents, games, proving theorems, path finding problems
♦ Basic search algorithms (search for solution - what space ? )

Notion of the problem space - set of states - set of operators - how to get from the initial state to the final state
Problem-solving agents

Restricted form of general agent:

```plaintext
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
    inputs: p, a percept
    static: s, an action sequence, initially empty
            state, some description of the current world state
            g, a goal, initially null
            problem, a problem formulation
    state ← UPDATE-STATE(state, p)
    if s is empty then
        g ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, g)
        s ← SEARCH(problem)
        action ← RECOMMENDATION(s, state)
        s ← REMAINDER(s, state)
    return action
```

Note: this is offline problem solving. 

Online problem solving involves acting without complete knowledge of the problem and solution.

Example: The 8-puzzle

```
Start State
5 4
6 1 8
7 3 2

Goal State
1 2 3
8 4
7 6 5
```

states??
operators??
goal test??
path cost??
Example: The 8-puzzle

<table>
<thead>
<tr>
<th>Start State</th>
<th>Goal State</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4</td>
<td>1 2 3</td>
</tr>
<tr>
<td>6 1 8</td>
<td>8 4</td>
</tr>
<tr>
<td>7 3 2</td>
<td>7 6 5</td>
</tr>
</tbody>
</table>

states?: integer locations of tiles (ignore intermediate positions)
operators?: move blank left, right, up, down (ignore unjamming etc.)
goal test?: = goal state (given)
path cost?: 1 per move

[Note: optimal solution of \(n\)-Puzzle family is NP-hard]

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:
be in Bucharest

Formulate problem:
states: various cities
operators: drive between cities

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
**Problem types**

- **Deterministic, accessible** $\Rightarrow$ *single-state problem*
- **Deterministic, inaccessible** $\Rightarrow$ *multiple-state problem*
- **Nondeterministic, inaccessible** $\Rightarrow$ *contingency problem*
  - must use sensors during execution
  - solution is a *tree* or *policy*
  - often *interleave* search, execution
- **Unknown state space** $\Rightarrow$ *exploration problem* (“online”)
Example: vacuum world

Single-state, start in #5. Solution??

Multiple-state, start in \{1, 2, 3, 4, 5, 6, 7, 8\} e.g., Right goes to \{2, 4, 6, 8\}. Solution??

Contingency, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only. Solution??

Single-state problem formulation

A problem is defined by four items:

initial state e.g., “at Arad”

operators (or successor function \(S(x)\))
e.g., Arad \(\rightarrow\) Zerind Arad \(\rightarrow\) Sibiu etc.

goal test, can be
explicit, e.g., \(x = \text{“at Bucharest”}\)
implicit, e.g., \(\text{NoDirt}(x)\)

path cost (additive)
e.g., sum of distances, number of operators executed, etc.

A solution is a sequence of operators leading from the initial state to a goal state
Selecting a state space

Real world is absurdly complex
⇒ state space must be abstracted for problem solving

(Abstract) state = set of real states

(Abstract) operator = complex combination of real actions
e.g., “Arad → Zerind” represents a complex set
of possible routes, detours, rest stops, etc.
For guaranteed realizability, any real state “in Arad”
must get to some real state “in Zerind”

(Abstract) solution =
set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!

Example: vacuum world state space graph

states??
operators??
goal test??
path cost??
Example: vacuum world state space graph

- **states**: integer dirt and robot locations (ignore dirt amounts)
- **operators**: Left, Right, Suck
- **goal test**: no dirt
- **path cost**: 1 per operator

Example: robotic assembly

- **states**: real-valued coordinates of robot joint angles, parts of the object to be assembled
- **operators**: continuous motions of robot joints
- **goal test**: complete assembly with no robot included!
- **path cost**: time to execute
Search algorithms

Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end

Implementation of search algorithms

function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
nodes ← Make-Queue(Make-Node(Initial-State[problem]))
loop do
  if nodes is empty then return failure
  node ← REMOVE-FRONT(nodes)
  if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
  nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
end
Implementation contd: states vs. nodes

A state is a (representation of) a physical configuration.
A node is a data structure constituting part of a search tree includes parent, children, depth, path cost \( g(x) \).
States do not have parents, children, depth, or path cost!

The Expand function creates new nodes, filling in the various fields and using the Operators (or SuccessorFn) of the problem to create the corresponding states.

Search strategies

A strategy is defined by picking the order of node expansion.

Strategies are evaluated along the following dimensions:
- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of
- \( b \)—maximum branching factor of the search tree
- \( d \)—depth of the least-cost solution
- \( m \)—maximum depth of the state space (may be \( \infty \))
Uninformed search strategies

*Uninformed* strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search