Problem Solving Agents & Problem Formulation

AIMA 2.3, 3.1-3
Outline for today’s lecture

- *Defining Task Environments (AIMA 2.3)*
- Environment types
- Formulating Search Problems
- Search Fundamentals
Task environments

- To design a rational agent we need to specify a task environment
  - a problem specification for which the agent is a solution

- PEAS: to specify a task environment
  - Performance measure
  - Environment
  - Actuators
  - Sensors
**PEAS: Specifying an automated taxi driver**

**Performance measure:**
- ?

**Environment:**
- ?

**Actuators:**
- ?

**Sensors:**
- ?
**PEAS:** Specifying an automated taxi driver

**Performance measure:**
- safe, fast, legal, comfortable, maximize profits

**Environment:**
- roads, other traffic, pedestrians, customers

**Actuators:**
- steering, accelerator, brake, signal, horn

**Sensors:**
- cameras, sonar, speedometer, GPS
**PEAS: Medical diagnosis system**

- **Performance measure:** Healthy patient, minimize costs, lawsuits

- **Environment:** Patient, hospital, staff

- **Actuators:** Screen display (form including: questions, tests, diagnoses, treatments, referrals)

- **Sensors:** Keyboard (entry of symptoms, findings, patient's answers)
Outline for today’s lecture

• Defining Task Environments
• Environment types (also AIMA 2.3)
• Formulating Search Problems
• Search Fundamentals
Environment types: Definitions I

- **Fully observable** (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.

- **Deterministic** (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent.
  - If the environment is deterministic except for the actions of other agents, then the environment is *strategic*.

- **Episodic** (vs. sequential): The agent's experience is divided into atomic "episodes" during which the agent perceives and then performs a single action, and the choice of action in each episode depends only on the episode itself.
Environment types: Definitions II

- **Static** (vs. dynamic): The environment is unchanged while an agent is deliberating.
  - The environment is *semidynamic* if the environment itself does not change with the passage of time but the agent's performance score does.

- **Discrete** (vs. continuous): A limited number of distinct, clearly defined percepts and actions.

- **Single agent** (vs. multiagent): An agent operating by itself in an environment.

*(See examples in AIMA, however I don’t agree with some of the judgments)*
Environment Restrictions for Now

- We will assume environment is
  - Static
  - Fully Observable
  - Deterministic
  - Discrete
The rational agent designer’s goal

- Goal of AI practitioner who designs rational agents: given a PEAS task environment,

1. Construct agent function $f$ that maximizes (the expected value of) the performance measure,

2. Design an agent program that implements $f$ on a particular architecture
Outline for today’s lecture

- Defining Task Environments
- Environment types
- *Formulating Search Problems* (AIMA, 3.1-3.2)
- Search Fundamentals
Example search problem: 8-puzzle

- **Formulate goal**
  - Pieces to end up in order as shown…

- **Formulate search problem**
  - **States**: configurations of the puzzle (9! configurations)
  - **Actions**: Move one of the movable pieces (≤4 possible)
  - **Performance measure**: minimize total moves

- **Find solution**
  - Sequence of pieces moved: 3,1,6,3,1,…
Example search problem: holiday in Romania

You are here

You need to be here
Holiday in Romania II

- **On holiday in Romania; currently in Arad**
  - Flight leaves tomorrow from Bucharest
- **Formulate goal**
  - Be in Bucharest
- **Formulate search problem**
  - States: various cities
  - Actions: drive between cities
  - Performance measure: minimize distance
- **Find solution**
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, …
More formally, a problem is defined by:

1. A set of *states* $S$
2. An *initial state* $s_i \in S$
3. A *set of actions* $A$
   - $\forall s \; \text{Actions}(s) =$ the set of actions that can be executed in $s$, that are *applicable* in $s$.
4. *Transition Model*: $\forall s \; \forall a \in \text{Actions}(s) \; \text{Result}(s, a) \rightarrow s_r$
   - $s_r$ is called a *successor* of $s$
   - $\{s_i\} \cup \text{Successors}(s_i)^* =$ *state space*
5. *Goal test* $\text{Goal}(s)$
   - Can be implicit, e.g. $\text{checkmate}(x)$
   - $s$ is a *goal state* if $\text{Goal}(s)$ is *true*
6. *Path cost* (additive)
   - e.g. sum of distances, number of actions executed, ...
   - $c(x,a,y)$ is the step cost, assumed $\geq 0$
     - (where action $a$ goes from state $x$ to state $y$)
Solution

A *solution* is a sequence of actions from the *initial state* to a *goal state*.

**Optimal Solution:**
A solution is *optimal* if no solution has a lower path cost.
Hard subtask: Selecting a state space

- Real world is absurdly complex
  State space must be *abstracted* for problem solving

- (abstract) \textit{State} = set (equivalence class) of real world states

- (abstract) \textit{Action} = equivalence class of combinations of real world actions
  - e.g. \textit{Arad} $\rightarrow$ \textit{Zerind} represents a complex set of possible routes, detours, rest stops, etc
  - The abstraction is valid if the path between two states is reflected in the real world

- Each abstract action should be “easier” than the real problem
Art: Formulating a Search Problem

Decide:

- Which properties matter & how to represent
  - Initial State, Goal State, Possible Intermediate States
- Which actions are possible & how to represent
  - Operator Set: Actions and Transition Model
- Which action is next
  - Path Cost Function

*Formulation greatly affects combinatorics of search space and therefore speed of search*
Example: 8-puzzle

- States??
- Initial state??
- Actions??
- Transition Model??
- Goal test??
- Path cost??
Example: 8-puzzle

- **States??** List of 9 locations- e.g., [7,2,4,5,-,6,8,3,1]
- **Initial state??** [7,2,4,5,-,6,8,3,1]
- **Actions??** \{Left, Right, Up, Down\}
- **Transition Model??** ...
- **Goal test??** Check if goal configuration is reached
- **Path cost??** Number of actions to reach goal
Example: Missionaries & Cannibals

Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten. *(AIMA problem 3.9)*

How shall they cross the river?
Formulation: Missionaries & Cannibals

• **How to formalize:**
  • *Initial state*: all M, all C, and boat on one bank
  • *Actions*: ??
  • *Transition Model*: ??
  • *Goal test*: True if all M, all C, and boat on other bank
  • *Cost*: ??

**Remember:**

• **Representation:**
  • *States*: Which properties matter & how to represent
  • *Actions & Transition Model*: Which actions are possible & how to represent
  • *Path Cost*: Deciding which action is next
# Missionaries and Cannibals

**States:** \((CL, ML, BL)\)

<table>
<thead>
<tr>
<th>Initial</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>331</td>
<td>000</td>
</tr>
</tbody>
</table>

**Actions:**

<table>
<thead>
<tr>
<th>Travel Across</th>
<th>Travel Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>-101</td>
<td>101</td>
</tr>
<tr>
<td>-201</td>
<td>201</td>
</tr>
<tr>
<td>-011</td>
<td>011</td>
</tr>
<tr>
<td>-021</td>
<td>021</td>
</tr>
<tr>
<td>-111</td>
<td>111</td>
</tr>
</tbody>
</table>
Outline for today’s lecture

- Defining Task Environments
- Environment types
- Formulating Search Problems
- *Search Fundamentals (AIMA 3.3)*
Useful Concepts

- **State space**: the set of all states reachable from the initial state by *any* sequence of actions
  - When several operators can apply to each state, this gets large very quickly
  - Might be a proper subset of the set of configurations

- **Path**: a sequence of actions leading from one state $s_j$ to another state $s_k$

- **Frontier**: those states that are available for expanding (for applying legal actions to)

- **Solution**: a path from the initial state $s_i$ to a state $s_f$ that satisfies the goal test
Basic search algorithms: *Tree Search*

- Generalized algorithm to solve search problems (Review from CIS 121)
  - *Enumerate in some order all possible paths from the initial state*
  - Here: search through *explicit tree generation*
    - ROOT = initial state.
    - Nodes in search tree generated through *transition model*
  - In general search generates a *graph* (same state through multiple paths), but we’ll just look at *trees in lecture*
    - Tree search treats different paths to the same node as distinct
function \text{TREE-SEARCH}(\text{problem}, \text{strategy}) \text{ return a solution or failure}

Initialize frontier to the initial state of the problem

do
\begin{align*}
\text{if the frontier is empty then return} & \text{ failure} \\
\text{choose leaf node for expansion according to strategy} & \text{ & remove from frontier} \\
\text{if node contains goal state then return} & \text{ solution} \\
\text{else} & \text{ expand the node and add resulting nodes to the frontier}
\end{align*}
8-Puzzle: States and Nodes

- **State** is a (representation of a) **physical configuration**
- A **node** is a data structure constituting **part of a search tree**
  - Also includes *parent, children, depth, path cost* \( g(x) \)
  - Here *node* = \(<\text{state, parent-node, children, action, path-cost, depth}>\)
- States do not have parents, children, depth or path cost!

**The EXPAND function**
- uses the Actions and Transition Model to create the corresponding states
  - creates new nodes,
  - fills in the various fields
8-Puzzle Search Tree

• (Nodes show state, parent, children - leaving Action, Cost, Depth Implicit)

• Suppressing useless “backwards” moves
Problem: Repeated states

- Failure to detect *repeated states* can turn a linear problem into an *exponential* one!
Solution: Graph Search!

- **Graph search**
  - Optimal but memory inefficient

- Mod from tree search: Check to see if a node has been visited before adding to search queue
  - must keep track of all possible states (can use a lot of memory)
  - e.g., 8-puzzle problem, we have $9!/2 \approx 182K$ states
Graph Search vs Tree Search

**function** TREE-SEARCH(*problem*) **returns** a solution, or failure
initialize the frontier using the initial state of *problem*

**loop**
if the frontier is empty then **return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state then **return** the corresponding solution
expand the chosen node, adding the resulting nodes to the frontier

**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
initialize the frontier using the initial state of *problem*
initialize the explored set to be empty

**loop**
if the frontier is empty then **return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state then **return** the corresponding solution
add the node to the explored set
expand the chosen node, adding the resulting nodes to the frontier
only if not in the frontier or explored set

**Figure 3.7** An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.