Types of agents

Reflex agent
- Consider how the world IS
- Choose action based on current percept
- Do not consider the future consequences of actions

Goal-directed agent
- Consider how the world WOULD BE
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Must formulate a goal

Source: D. Klein, P. Abbeel
Outline of today’s lecture

1. How to turn ANY problem into a SEARCH problem:
   1. Initial state, goal state, transition model
   2. Actions, path cost

2. General algorithm for solving search problems
   1. First data structure: a frontier list
   2. Second data structure: a search tree
   3. Third data structure: a “visited states” list
Search

• We will consider the problem of designing goal-based agents in fully observable, deterministic, discrete, static, known environments
Search

We will consider the problem of designing goal-based agents in fully observable, deterministic, discrete, known environments:

- The agent must find a sequence of actions that reaches the goal.
- The performance measure is defined by (a) reaching the goal and (b) how “expensive” the path to the goal is:
  - The agent doesn’t know the performance measure. This is a goal-directed agent, not a utility-directed agent.
  - The programmer (you) DOES know the performance measure. So you design a goal-seeking strategy that minimizes cost.
- We are focused on the process of finding the solution; while executing the solution, we assume that the agent can safely ignore its percepts (static environment, open-loop system).
Search problem components

• **Initial state**
• **Actions**
• **Transition model**
  • What state results from performing a given action in a given state?
• **Goal state**
• **Path cost**
  • Assume that it is a sum of nonnegative *step costs*

• The **optimal solution** is the sequence of actions that gives the *lowest* path cost for reaching the goal
Knowledge Representation: State

• State = description of the world
  • Must have enough detail to decide whether or not you’re currently in the initial state
  • Must have enough detail to decide whether or not you’ve reached the goal state
  • Often but not always: “defining the state” and “defining the transition model” are the same thing
Example: Romania

- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

**Initial state**
- Arad

**Actions**
- Go from one city to another

**Transition model**
- If you go from city A to city B, you end up in city B

**Goal state**
- Bucharest

**Path cost**
- Sum of edge costs (total distance traveled)
State space

• The initial state, actions, and transition model define the state space of the problem
  • The set of all states reachable from initial state by any sequence of actions
  • Can be represented as a directed graph where the nodes are states and links between nodes are actions

• What is the state space for the Romania problem?
Example: Vacuum world

- **States**
  - Agent location and dirt location
  - How many possible states?
  - What if there are $n$ possible locations?
    - The size of the state space grows exponentially with the “size” of the world!

- **Actions**
  - Left, right, suck

- **Transition model**
Vacuum world state space graph
Complexity of the State Space

• Many “video game” style problems can be subdivided:
  • There are M different things your character needs to pick up: $2^M$ different world states
  • There are N locations you can be in while carrying any subset of those M objects: total number of world states = $O(2^M N)$

• Why a maze is nice: you don’t need to pick anything up
  • Only N different world states to consider
Example: The 8-puzzle

- **States**
  - Locations of tiles
    - 8-puzzle: 181,440 states (9!/2)
    - 15-puzzle: ~10 trillion states
    - 24-puzzle: ~$10^{25}$ states

- **Actions**
  - Move blank left, right, up, down

- **Path cost**
  - 1 per move

- **Finding the optimal solution of n-Puzzle is NP-hard**
Example: Robot motion planning

- **States**
  - Real-valued joint parameters (angles, displacements)
- **Actions**
  - Continuous motions of robot joints
- **Goal state**
  - Configuration in which object is grasped
- **Path cost**
  - Time to execute, smoothness of path, etc.
Traveling Salesman Problem

• Goal: visit every city in the United States
• Path cost: total miles traveled
• Initial state: Champaign, IL
• Action: travel from one city to another
• Transition model: when you visit a city, mark it as “visited.”
Outline of today’s lecture

1. How to turn ANY problem into a SEARCH problem:
   1. Initial state, goal state, transition model
   2. Actions, path cost

2. General algorithm for solving search problems
   1. First data structure: a frontier list
   2. Second data structure: a search tree
   3. Third data structure: a “visited states” list
Search

• Given:
  • Initial state
  • Actions
  • Transition model
  • Goal state
  • Path cost

• How do we find the optimal solution?
  • How about building the state space and then using Dijkstra’s shortest path algorithm?
    • Complexity of Dijkstra’s is $O(E + V \log V)$, where $E$ is the number of transitions, $V$ is the number of states
    • Usually, $V = O(2^{\# \text{variables you need to keep track of}})$
    • Therefore the shortest-path algorithm is exponential in the # variables
Tree Search: Basic idea

• Let’s begin at the start state and expand it by making a list of all possible successor states
• Maintain a frontier or a list of unexpanded states
• At each step, pick a state from the frontier to expand (EXPAND = list all of the other states that can be reached from this state)
• Keep going until you reach a goal state
• BACK-TRACE: go back up the tree; list, in reverse order, all of the actions you need to perform in order to reach the goal state.
• ACT: the agent reads off the sequence of necessary actions, in order, and does them.
Search tree

• “What if” tree of sequences of actions and outcomes
• The root node corresponds to the starting state
• The children of a node correspond to the successor states of that node’s state
• A path through the tree corresponds to a sequence of actions
  • A solution is a path ending in the goal state
• Nodes vs. states
  • A state is a representation of the world, while a node is a data structure that is part of the search tree
    • Node has to keep pointer to parent, path cost, possibly other info
Knowledge Representation: States and Nodes

- **State** = description of the world
  - Must have enough detail to decide whether or not you’re currently in the *initial state*
  - Must have enough detail to decide whether or not you’ve reached the *goal state*
  - Often but not always: “defining the state” and “defining the transition model” are the same thing

- **Node** = a point in the search tree
  - Private data: ID of the state reached by this node
  - Private data: the ID of the parent node, and of all child nodes
Search: Computational complexity

• In the typical case, your search algorithm will need to expand about half of all of the world-states

• The number of world-states is exponential in the number of sub-tasks you need to perform, $V = O\{2^M\}$

• Therefore, in the typical case, search is exponential complexity

• Most of what we discuss, in the next three lectures, will be methods for limiting the mantissa and/or limiting the exponent.
Tree Search Algorithm Outline

- Initialize the **frontier** using the **starting state**
- While the frontier is not empty
  - Choose a frontier node according to **search strategy** and take it off the frontier
  - If the node contains the **goal state**, return solution
  - Else **expand** the node and add its children to the frontier
Tree search example

Start: Arad
Goal: Bucharest
Tree search example

Start: Arad  Goal: Bucharest

Start: Arad  Goal: Bucharest
Tree search example

Start: Arad
Goal: Bucharest
Tree search example

Start: Arad
Goal: Bucharest
Tree search example

Start: Arad
Goal: Bucharest
Tree search example

Start: Arad
Goal: Bucharest
Handling repeated states

• Initialize the **frontier** using the **starting state**
• While the frontier is not empty
  • Choose a frontier node according to **search strategy** and take it off the frontier
  • If the node contains the **goal state**, return solution
  • Else **expand** the node and add its children to the frontier

• To handle repeated states:
  • Every time you expand a node, add that state to the **explored set**; do not put explored states on the frontier again
  • Every time you add a node to the frontier, check whether it already exists in the frontier with a higher path cost, and if yes, replace that node with the new one
Tree search w/o repeats

Start: Arad
Goal: Bucharest
Tree search w/o repeats

Explored:
Arad

Start: Arad    Goal: Bucharest
Tree search example

Explored:
Arad
Sibiu

Start: Arad
Goal: Bucharest
Tree search example

Explored:
Arad
Sibiu
Rimnicu Vilces

Start: Arad
Goal: Bucharest

<table>
<thead>
<tr>
<th>City</th>
<th>Straight-line distance to Bucharest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobroța</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Făgărăș</td>
<td>176</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iași</td>
<td>226</td>
</tr>
<tr>
<td>Lăcătani</td>
<td>244</td>
</tr>
<tr>
<td>Mehedia</td>
<td>241</td>
</tr>
<tr>
<td>Neamț</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitești</td>
<td>10</td>
</tr>
<tr>
<td>Rimnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timișoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vâlcea</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Tree search example

Explored:
Arad
Sibiu
Rimnicu Vilces
Fagaras

Start: Arad
Goal: Bucharest
Our first computational savings: avoid repeated states

• Complexity if you allow repeated states: mantissa = number of states you can transition to from any source state, exponent = depth of the tree

• Complexity if you don’t allow repeated states: never larger than the total number of states in the world

• For a maze search: # states = # positions you can reach, therefore avoiding repeated states might be all the computational savings you need

• For a task with multiple sub-tasks, e.g., search a maze while cleaning dirt: # states is exponential in the # sub-tasks, therefore we still need better algorithms. That’s the topic for next week.