Reminders

✦ Homework 2 will be released today
✦ I have help hours Thursday from 4:30 - 5:30pm
✦ Reading for next Tuesday: YLLATAILY Chapter 3-4
✦ Tutor help hours start next week!
YLLATAILY Chapters 1-2
Big Ideas

Rule-based programming

- Pro: we understand the rules the program is using
- Con: we have to write the rules

Supervised learning

- Pro: AI generates its own rules
- Con: hard to understand *why* it's doing what it's doing
Big Ideas

Signs of AI Doom:

- The problem is too hard
- The problem is not what we thought it was
- There are sneaky shortcuts
- The AI tried to learn from garbage data
Big Ideas

AI Weaknesses

- Remembering things
- Planning ahead
- Data- and computation-intensive
Example Tasks

- Self-driving cars
- Recipe generation
- Résumé screening
- Cockroach farming
- Tic-tac-toe
- Image recognition
- Joke generation
- Super Mario
- Writing news articles

Inability to adapt

- How does it know what tastes good?
- Possible biases from training data
- Keywords: easy but not useful
- Sneaky shortcuts
- Sneaky shortcuts + bad data
- Learning word correlations
Recap
Given a goal, an AI agent must decide what the best action to take is in order to reach this goal.

For complex tasks, this can mean:

- gathering information
- coming up a set of possible actions
- weighing the best action
- acting
- updating and adapting based on changes to the environment
Agent Complexity

**Problem-solving agent**: capable of considering a sequence of actions that form a path to a goal state (planning ahead).

If I wail, the human might refill my bowl. Then I can eat more.
Search
Example search problem: Holiday in Romania

You are here

You need to be here

Slides adapted from Chris Callison-Burch
Holiday in Romania

On holiday in Romania; currently in Arad

- Flight leaves tomorrow from Bucharest

Formulate **goal**

Be in Bucharest

Formulate **search problem**

**States:** cities

**Actions:** driving between cities

**Cost:** distance (between cities)

Find **solution**

Sequence of cities
Example search problem: 8-puzzle

Formulate **goal**

- State: Where tiles are in order (as shown on right)

Formulate **search problem**

- States: Ways to arrange tiles (9!)
- Actions: Move a tile: up, down, left, right
- Cost: # of moves

Find **solution**

- Sequence of pieces moved w/ the direction they move
Search Algorithms
Basic search algorithms: *Tree Search*

Generalized algorithm to solve search problems

**Enumerate in some order all possible paths from the initial state**

- Search through explicit tree generation
  - Root: initial state
  - Nodes: states (generated through transition model)

Tree search treats different paths to the same state (node) as distinct
Generalized tree search

initialize frontier to the start state
do
  if the frontier is empty then return failure
  choose the next node to expand according to strategy
  if node is goal, return solution
  else expand the node
    for each child:
      if child has not been visited, add to frontier
Generalized tree search

function TREE-SEARCH(problem, strategy) return a solution or failure
  Initialize frontier to the initial state of the problem
  do
    if the frontier is empty then return failure
    choose leaf node for expansion according to strategy & remove from frontier
    if node contains goal state then return solution
    else expand the node and add resulting nodes to the frontier

The strategy determines search process!
Search Tree

Root node = start state

Expanded nodes:
- Alderaan
- Starkiller Base
- Onderon
- Endor
- Ryloth
- Tatooine
- Coruscant

Frontier

Choose leaf node from frontier for expansion according to the search strategy

Determines the search process

Slides adapted from Chris Callison-Burch
States Versus Nodes

State: representation of a physical configuration of the environment

Node: a data structure with several fields:

\[(\text{state}, \text{parent node}, \text{children}, \text{action}, \text{cost}, \text{depth})\]

States don't have costs, parents, or depths!
8-Puzzle **Search Tree**

(Nodes show state, parent, children - leaving *Action*, *Cost*, *Depth* Implicit)
Problem: Repeated states

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

Graph search
- Simple 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 do**
  **if** the frontier is empty **then return** failure
  choose a leaf nose 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 do**
  **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 node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier only if not in the frontier of explored set
Uninformed Search
Uninformed Search

Uses only information available in problem definition

Informally:

*Uninformed search:* All non-goal nodes in frontier look equally good

*Informed search:* Some non-goal nodes can be ranked above others.

Slides adapted from Chris Callison-Burch
Breadth-First Search
Breadth-first search

Idea:
  • Expand *shallowest* unexpanded node

Implementation:
  • *frontier* is FIFO (First-In-First-Out) Queue:
    • Put successors at the *end* of *frontier* successor list.
Breadth-first search

```plaintext
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
        return failure
```
Breadth-first search

function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(State=s', Parent=node, Action=action, Path-Cost=cost)

Node data structure contains variables like the state, a pointer to its parent node, the action that was used to create this state, and the path cost.

The Python yield keyword means that we don’t have to pre-compute a list of all successors.
Breadth-first search

\begin{verbatim}
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
        return failure
\end{verbatim}

Subtle: Node inserted into queue only after testing to see if it is a goal state
bfs(x):
make a new queue called q
mark x visited
push x onto q

while q not empty:
    pop q into x
    for each y in x connections
        if y not visited:
            mark y visited
            push y onto q
Properties of breadth-first search

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes (if $b$ is finite)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes if cost = 1 per step</td>
</tr>
<tr>
<td>Time Complexity?</td>
<td>$1 + b + b^2 + b^3 + \ldots = O(b^d)$</td>
</tr>
<tr>
<td>Space Complexity?</td>
<td>$O(b^d)$</td>
</tr>
</tbody>
</table>

$b$ = branching factor
d = depth
Exponential Space (and time) Is Not Good...

- Exponential complexity uninformed search problems cannot be solved for any but the smallest instances.
- (Memory requirements are a bigger problem than execution time.)

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>NODES</th>
<th>TIME</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>0.11 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 petabytes</td>
</tr>
</tbody>
</table>

Assumes b=10, 1M nodes/sec, 1000 bytes/node