Search

Luke Zettlemoyer

With slides from
Dan Klein, Stuart Russell, Andrew Moore, Dan Weld
Announcements

- **Project 0: Python Tutorial**
  - Online, but not graded

- **Project 1: Search**
  - On the web by tomorrow.
  - Start early and ask questions. *It’s longer than most!*
Outline

- Agents that Plan Ahead

- Search Problems

- Uninformed Search Methods (part review for some)
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search

- Heuristic Search Methods (new for all)
  - Best First / Greedy Search
An agent is an entity that perceives and acts.

A rational agent selects actions that maximize its utility function.

Characteristics of the percepts, environment, and action space dictate techniques for selecting rational actions.

Search -- the environment is: fully observable, single agent, deterministic, episodic, discrete
Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - Do not consider the future consequences of their actions
  - Act on how the world IS

- Can a reflex agent be rational?
- Can a non-rational agent achieve goals?
Famous Reflex Agents
Goal Based Agents

- **Goal-based agents:**
  - Plan ahead
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Act on how the world WOULD BE
Search thru a
Problem Space / State Space

• Input:
  ▪ Set of states
  ▪ Operators [and costs]
  ▪ Start state
  ▪ Goal state [test]

• Output:
  • Path: start ⇒ a state satisfying goal test
  • [May require shortest path]
  • [Sometimes just need state passing test]
Example: Simplified Pac-Man

- **Input:**
  - A state space
  - A successor function
  - A start state
  - A goal test

- **Output:**
Ex: Route Planning: Romania → Bucharest

- **Input:**
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)

- **Output:**
Example: N Queens

- **Input:**
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)

- **Output**
Algebraic Simplification

\[ \partial^2_r u = - \left[ E' - \frac{l(l+1)}{r^2} - r^3 \right] u(r) \]
\[ e^{-2s} (\partial^3_s - \partial_s) u(s) = - \left[ E' - l(l+1)e^{-2s} - e^{2s} \right] u(s) \]
\[ e^{-2s} \left[ e^{\frac{1}{2}s} \left( e^{-\frac{3}{2}s} u(s) \right)' - \frac{1}{4} u \right] = - \left[ E' - l(l+1)e^{-2s} - e^{2s} \right] u(s) \]
\[ e^{-2s} \left[ e^{\frac{1}{2}s} \left( e^{-\frac{3}{2}s} u(s) \right)'' \right] = - \left[ E' - \left( l + \frac{1}{2} \right)^2 e^{-2s} - e^{2s} \right] u(s) \]
\[ v'' = - e^{2s} \left[ E' - \left( l + \frac{1}{2} \right)^2 e^{-2s} - e^{2s} \right] v \]

- **Input:**
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)

- **Output:**
State Space Graphs

- **State space graph:**
  - Each node is a state
  - The successor function is represented by arcs
  - Edges may be labeled with costs
- We can rarely build this graph in memory (so we don’t)

Ridiculously tiny search graph for a tiny search problem
State Space Sizes?

- **Search Problem:** Eat all of the food
- **Pacman positions:** $10 \times 12 = 120$
- **Pacman facing:** up, down, left, right
- **Food Count:** 30
- **Ghost positions:** 12
Search Strategies

- **Blind Search**
  - Depth first search
  - Breadth first search
  - Iterative deepening search
  - Uniform cost search

- **Informed Search**

- **Constraint Satisfaction**

- **Adversary Search**
A search tree:
- Start state at the root node
- Children correspond to successors
- Nodes contain states, correspond to PLANS to those states
- Edges are labeled with actions and costs
- For most problems, we can never actually build the whole tree
Example: Tree Search

State Graph:

What is the search tree?
We construct both on demand – and we construct as little as possible.

Each NODE in the search tree is an entire PATH in the problem graph.
States vs. Nodes

- Nodes in state space graphs are problem states
  - Represent an abstracted state of the world
  - Have successors, can be goal / non-goal, have multiple predecessors
- Nodes in search trees are plans
  - Represent a plan (sequence of actions) which results in the node’s state
  - Have a problem state and one parent, a path length, a depth & a cost
  - The same problem state may be achieved by multiple search tree nodes

Search Nodes

Problem States

- Depth 5
- Depth 6
Building Search Trees

- **Search:**
  - Expand out possible plans
  - Maintain a *fringe* of unexpanded plans
  - Try to expand as few tree nodes as possible
General Tree Search

- **Important ideas:**
  - Fringe
  - Expansion
  - Exploration strategy

- **Main question:** which fringe nodes to explore?

---

function TREE-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

Detailed pseudocode is in the book!
Review: Depth First Search

Strategy: expand deepest node first
Implementation: Fringe is a LIFO queue (a stack)
Review: Depth First Search

Expansion ordering:
(d,b,a,c,a,e,h,p,q,q,r,f,c,a,G)
Strategy: expand shallowest node first

Implementation: Fringe is a FIFO queue
Review: Breadth First Search

Expansion order:
(S,d,e,p,b,c,e,h,r,q,a,a,h,r,p,q,f,p,q,f,q,c,G)
Search Algorithm Properties

- **Complete?** Guaranteed to find a solution if one exists?
- **Optimal?** Guaranteed to find the least cost path?
- **Time complexity?**
- **Space complexity?**

Variables:

<table>
<thead>
<tr>
<th>n</th>
<th>Number of states in the problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>The maximum branching factor B (the maximum number of successors for a state)</td>
</tr>
<tr>
<td>C*</td>
<td>Cost of least cost solution</td>
</tr>
<tr>
<td>d</td>
<td>Depth of the shallowest solution</td>
</tr>
<tr>
<td>m</td>
<td>Max depth of the search tree</td>
</tr>
</tbody>
</table>
DFS

- Infinite paths make DFS incomplete…
  - How can we fix this?
  - Check new nodes against path from S
- Infinite search spaces still a problem

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>No</td>
<td>No</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
<td>No</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

Diagram:
- START
- a
- b
- GOAL

- DFS:
  - Depth First Search
  - No
  - No
  - Infinite
  - Infinite
DFS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y if finite</td>
<td>N</td>
<td>O(b^m)</td>
<td>O(bm)</td>
</tr>
<tr>
<td>w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Or graph search – next lecture.
When is BFS optimal?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
</tbody>
</table>

DFS with Path Checking

- Complete: Y
- Optimal: N
- Time: $O(b^m)$
- Space: $O(bm)$

BFS

- Complete: Y
- Optimal: Y*
- Time: $O(b^d)$
- Space: $O(b^d)$

Diagram:
- d tiers
- 1 node
- $b$ nodes
- $b^2$ nodes
- $b^d$ nodes
- $b^m$ nodes
Memory a Limitation?

- **Suppose:**
  - 4 GHz CPU
  - 6 GB main memory
  - 100 instructions / expansion
  - 5 bytes / node
  - 400,000 expansions / sec
    - Memory filled in 300 sec ... 5 min
Comparisons

- When will BFS outperform DFS?
- When will DFS outperform BFS?
Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.
   …and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(bd)$</td>
</tr>
</tbody>
</table>
### Speed

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8 Puzzle</strong></td>
<td>$10^5$</td>
<td>.01 sec</td>
<td>$10^5$</td>
<td>.01 sec</td>
</tr>
<tr>
<td><strong>2x2x2 Rubik’s</strong></td>
<td>$10^6$</td>
<td>.2 sec</td>
<td>$10^6$</td>
<td>.2 sec</td>
</tr>
<tr>
<td><strong>15 Puzzle</strong></td>
<td>$10^{13}$</td>
<td>6 days</td>
<td>$10^{17}$</td>
<td>20k yrs</td>
</tr>
<tr>
<td><strong>3x3x3 Rubik’s</strong></td>
<td>$10^{19}$</td>
<td>68k yrs</td>
<td>$10^{20}$</td>
<td>574k yrs</td>
</tr>
<tr>
<td><strong>24 Puzzle</strong></td>
<td>$10^{25}$</td>
<td>12B yrs</td>
<td>$10^{37}$</td>
<td>$10^{23}$ yrs</td>
</tr>
</tbody>
</table>

**Why the difference?**
- Rubik has higher branch factor
- 15 puzzle has greater depth

---

Slide adapted from Richard Korf presentation
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.
Uniform Cost Search

Expand cheapest node first:
Fringe is a priority queue
Uniform Cost Search

Expansion order:
(S,p,d,b,e,a,r,f,e,G)
A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pq.push(key, value)</td>
<td>inserts (key, value) into the queue.</td>
</tr>
<tr>
<td>pq.pop()</td>
<td>returns the key with the lowest value, and removes it from the queue.</td>
</tr>
</tbody>
</table>

You can decrease a key’s priority by pushing it again.

Unlike a regular queue, insertions aren’t constant time, usually $O(\log n)$.

We’ll need priority queues for cost-sensitive search methods.
## Uniform Cost Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(b^{C*/\varepsilon}$</td>
<td>$O(b^{C*/\varepsilon}$</td>
</tr>
</tbody>
</table>

### Diagram

```
C*/\varepsilon tiers
```

The diagram illustrates the concept of $C*/\varepsilon$ tiers in the context of uniform cost search algorithms, highlighting the progression through the tiers with decreasing cost as the search progresses deeper into the graph.
Uniform Cost Issues

- Remember: explores increasing cost contours

- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location
Uniform Cost: Pac-Man

- Cost of 1 for each action
- Explores all of the states, but one
Search Heuristics

- Any estimate of how close a state is to a goal
- Designed for a particular search problem

Examples: Manhattan distance, Euclidean distance
Best First / Greedy Search

Expand closest node first: Fringe is a priority queue
Best First / Greedy Search

- Expand the node that seems closest…

- What can go wrong?
Best First / Greedy Search

- A common case:
  - Best-first takes you straight to the (wrong) goal

- Worst-case: like a badly-guided DFS in the worst case
  - Can explore everything
  - Can get stuck in loops if no cycle checking

- Like DFS in completeness (finite states w/ cycle checking)
To Do:

- Look at the course website:
  - http://www.cs.washington.edu/cse473/12sp
- Do the readings (Ch 3)
- Do PS0 if new to Python
- Start PS1, when it is posted