Announcements

- **Project 0: Python Tutorial**
  - Due tomorrow!
  - There is a lab **Wednesday from 3pm-5pm in Soda 275**
  - The lab time is optional, but P0 itself is not
  - On submit, you should get email from the autograder

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

- **Self-Diagnostic on web**

- **Sections: can go to any, but have priority in your own**

---

**CS 188: Artificial Intelligence**

**Fall 2011**

**Lecture 2: Queue-Based Search**

**8/30/2011**

Dan Klein – UC Berkeley

Multiple slides from Stuart Russell, Andrew Moore
Today

- 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)
    - Greedy Search

Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world’s current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS

- Can a reflex agent be rational?

[demo: reflex optimal / loop]
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
  - Consider how the world WOULD BE

Search Problems

- A search problem consists of:
  - A state space
  - A successor function (with actions, costs)
  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state
Example: Romania

- State space:
  - Cities
- Successor function:
  - Roads: Go to adj city with cost = dist
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

State Space Graphs

- State space graph: A mathematical representation of a search problem
  - For every search problem, there’s a corresponding state space graph
  - The successor function is represented by arcs

- We can rarely build this graph in memory (so we don’t)

Ridiculously tiny search graph for a tiny search problem
What’s in a State Space?

The world state specifies every last detail of the environment.

A search state keeps only the details needed (abstraction).

- Problem: Pathing
  - States: (x, y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x, y) = END

- Problem: Eat-All-Dots
  - States: {(x, y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- How many
  - World states?
    - $120 \times (2^{30}) \times (12^2) \times 4$
  - States for pathing?
    - 120
  - States for eat-all-dots?
    - $120 \times (2^{30})$
Search Trees

A search tree:
- This is a “what if” tree of plans and outcomes
- Start state at the root node
- Children correspond to successors
- Nodes contain states, correspond to PLANS to those states
- For most problems, we can never actually build the whole tree

Another Search Tree

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?

Detailed pseudocode is in the book!

Example: Tree Search
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.

Review: Depth First Search

Strategy: expand deepest node first
Implementation: Fringe is a LIFO stack
Review: Breadth First Search

Strategy: expand shallowest node first
Implementation: Fringe is a FIFO queue

Search Tiers

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>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Number of states in the problem (huge)</td>
</tr>
<tr>
<td>$b$</td>
<td>The average branching factor $B$ (the average number of successors)</td>
</tr>
<tr>
<td>$C^*$</td>
<td>Cost of least cost solution</td>
</tr>
<tr>
<td>$s$</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

<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>N</td>
<td>N</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

- Infinite paths make DFS incomplete…
- How can we fix this?

**With cycle checking, DFS is complete.**

<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+1}))</td>
<td>(O(bm))</td>
</tr>
</tbody>
</table>

- When is DFS optimal?

* Or graph search – next lecture.
### BFS

<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>$O(b^{m+1})$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>$O(b^{m+1})$</td>
<td>$O(b^s)$</td>
</tr>
</tbody>
</table>

- When is BFS optimal?

### Comparisons

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

Iterative deepening: BFS using 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</td>
<td>Y</td>
<td>N</td>
<td>(O(b^{m+1}))</td>
<td>(O(bm))</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>(O(b^{s+1}))</td>
<td>(O(b^s))</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>N*</td>
<td>(O(b^{s+1}))</td>
<td>(O(bs))</td>
</tr>
</tbody>
</table>

Costs on Actions

Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.
We will quickly cover an algorithm which does find the least-cost path.
Uniform Cost Search

Expand cheapest node first:
Fringe is a priority queue (priority: cumulative cost)

Priority Queue Refresher

- 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+1}))</td>
<td>(O(b^m))</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N</td>
<td>(O(b^{s+1}))</td>
<td>(O(b^s))</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>

\(C^*/\varepsilon\) tiers

* UCS can fail if actions can get arbitrarily cheap

---

### 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

[demo: search demo empty]
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 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)
Search Gone Wrong?