CS 188: Artificial Intelligence
Fall 2008

Lecture 2: Queue-Based Search
9/2/2008

Dan Klein – UC Berkeley
Many slides from either Stuart Russell or Andrew Moore

Announcements

- Written assignments:
  - One mini-homework each week (1-2 problems)
  - We'll drop your two lowest scores
  - First one posted soon!

- Lab Wednesday 11am to 4pm in Soda 275
  - Learn Python
  - Come whatever times you like

- Project 1.1 posted soon, too, due 9/12

Today

- Agents that Plan Ahead
- Search Problems
- Uniformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Heuristic Search Methods
  - Greedy Search
  - A* Search

Reflex Agents

- Reflex agents:
  - Choose action based on current percept and memory
  - May have memory or a model of the world’s current state
  - Do not consider the future consequences of their actions
  - Can a reflex agent be rational?

Goal Based Agents

- Goal-based agents:
  - Plan ahead
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions

Search Problems

- A search problem consists of:
  - A state space
  - A successor function
  - 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
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 labeled with states, correspond to PLANS to those states
    - So, have to find ways of using only the important parts of the tree!

State Space Graphs

- There's some big graph in which
  - Each state is a node
  - Each successor is an outgoing arc
- Important: For most problems we could never actually build this graph
- How many states in Pacman?

Example: Romania

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?
State Graphs vs Search Trees

Each node in the search tree is an entire path in the problem graph.

We almost always construct both on demand – and we construct as little as possible.

Review: Depth First Search

Strategy: expand deepest node first
Implementation: Fringe is a LIFO stack

Each node in the search tree is an entire path in the problem graph.

Review: Breadth First Search

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

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</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.

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 w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>O(b^m)</td>
<td>O(bm)</td>
</tr>
</tbody>
</table>

- When is DFS optimal?
**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 <em>w/ Path Checking</em></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>$O(b^m)$</td>
<td>$O(b)$</td>
</tr>
</tbody>
</table>

- When is BFS optimal?

**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. (DFS gives up on any path of length 2)
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 <em>w/ Path Checking</em></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>$O(b^m)$</td>
<td>$O(b)$</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>N*</td>
<td>$O(b^{m+1})$</td>
<td>$O(bm)$</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 Queue Refresher**

- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:
  - `pq.push(key, value)`: inserts (key, value) into the queue.
  - `pq.pop()`: returns the key with the lowest value, and removes it from the queue.

- You can promote or demote keys by resetting their priorities
- Unlike a regular queue, insertions into a priority queue are not constant time, usually $O(\log n)$
- We'll need priority queues for most 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>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(b)$</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(C^* b^{m+1})$</td>
<td>$O(b^{m+1})$</td>
</tr>
</tbody>
</table>

We’ll talk more about uniform cost search’s failure cases later…

**Uniform Cost Problems**

- **Remember: explores increasing cost contours**
  - The good: UCS is complete and optimal!
  - The bad:
    - Explores options in every "direction"
    - No information about goal location

**Heuristics**

Expand the node that seems closest…

**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?

Extra Work?

Failure to detect repeated states can cause exponentially more work. Why?

Graph Search

In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)

Graph Search

Very simple fix: never expand a state type twice

Can this wreck completeness? Why or why not?

How about optimality? Why or why not?

Some Hints

Graph search is almost always better than tree search (when not?)

The collection of already-expanded state is sometimes called a “closed lists” – but they are really “closed sets.” Don’t implement them with lists (use sets)!

Nodes are conceptually paths, but better to represent with a state, cost, and reference to parent node (usually with an object bundling these three things)