CS 188: Artificial Intelligence
Spring 2006

Lecture 2: Queue-Based Search
8/31/2006

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

Announcements

- Lab Friday 1-5pm in Soda 275
  - Learn Python
  - Start on Project 1.1: Mazeworld
  - Come for whatever times you like

- No sections this Monday

- Project 1.1 posted due 9/8
  - You can do most of it after today
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 an 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
    - "N", 1.0
    - "E", 1.0
  - A start state and a goal test
- A solution is a sequence of actions which transform the start state to a goal state
Search Trees

- A search tree:
  - This is a “what if” tree
  - Current state at the root node
  - Children correspond to successors
  - Nodes labeled with states, correspond to PATHS to those states
  - For most problems, can never actually build the whole tree
    - 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?

Laughably tiny search graph for a tiny search problem
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
States vs. Nodes

- Problem graphs have problem states
  - Have successors

- Search trees have search nodes
  - Have parents, children, depth, path cost, etc.
  - Expand uses successor function to create new search tree nodes
  - The same problem state may be in multiple search tree nodes

General Tree Search

```plaintext
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
```

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?

Detailed pseudocode is in the book!
Example: Tree Search

State Graphs vs Search Trees

We almost always 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 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>Variables</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?
**DFS**

- 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(b^m)</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 w/ Path Checking</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>
</tbody>
</table>

- When is BFS optimal?
Comparisons

- When will BFS outperform DFS?
- When will DFS outperform BFS?

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:

<table>
<thead>
<tr>
<th>Function</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 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

- What will UCS do for this graph?

- What does this mean for completeness?

---

### 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>$O(b^{m+1})$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>UCS</td>
<td>$Y^*$</td>
<td>$Y$</td>
<td>$O(C^* b^{C^*/2})$</td>
<td>$O(b^{C^*/2})$</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

Extra Work?

- Failure to detect repeated states can cause exponentially more work. Why?
In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)

Very simple fix: never expand a node twice

```python
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(Make-Node(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, State[node]) then return node
        if State[node] is not in closed then
            add State[node] to closed
            fringe ← INSERT-ALL(Expand(node, problem), fringe)
    end
```

Can this wreck correctness? Why or why not?
Search Gone Wrong?

Best-First / Greedy Search
Best-First / Greedy Search

- Expand the node that seems closest...

What can go wrong?

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)