CS 188: Artificial Intelligence
Spring 2011

Lecture 2: Queue-Based Search
1/24/2010

Pieter Abbeel – UC Berkley
Many slides from Dan Klein

Announcements

- Project 0: Python Tutorial
  - Due Friday 5pm.
  - Lab session Wednesday 3-5pm in 271 Soda
  - The lab time is optional, but P0 itself is not
  - On submit, you should get email from the autograder

- Project 1: Search
  - Out today, due next week Friday 5pm.
  - Start early and ask questions. It’s longer than most!

- Sections starting this week Thursday and Friday
  - See course webpage for details

Today

- Reflex Agents
- Agents that Plan Ahead
  - Formalization: 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

Reminder

- Only a very small fraction of AI is about making computers play games intelligently
- Recall: computer vision, natural language, robotics, machine learning, computational biology, etc.

- That being said: games tend to provide relatively simple example settings which are great to illustrate concepts and learn about algorithms which underlie many areas of AI

Reflex Agent

- 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
- Act on how the world is
- Can a reflex agent be rational?

A reflex agent for pacman

4 actions: move North, East, South or West

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food
A reflex agent for pacman (2)

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food

A reflex agent for pacman (3)

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food
  - But, if other options are available, exclude the direction we just came from

A reflex agent for pacman (4)

- While(food left)
  - If can keep going in the current direction, do so
  - Otherwise:
    - Sort directions according to the amount of food
    - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from

A reflex agent for pacman (5)

- While(food left)
  - If can keep going in the current direction, do so
  - Otherwise:
    - Sort directions according to the amount of food
    - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from

Reflex Agent

- 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
- Act on how the world IS
- Can a reflex agent be rational?

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 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
**Example: Romania**

- **State space:**
  - Cities
- **Successor function:**
  - Go to adjacent city with cost = distance
- **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!)

**State Space Sizes?**

- **Search Problem:** Eat all of the food
  - Pacman positions: 10 x 12 = 120
  - Food count: 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**

- **Search:**
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible
- **Main question:** which fringe nodes to explore?
**Example: Tree Search**

- State Graphs vs. Search Trees
  - Each NODE in the search tree is an entire PATH in the problem graph.
  - We construct both on demand – and we construct as little as possible.

**Review: Depth First Search**
- Strategy: expand shallowest 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>n</th>
<th>Number of states in the problem</th>
</tr>
</thead>
<tbody>
<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.*

When is DFS 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>
</tbody>
</table>

BFS

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

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.

Uniform Cost Search

Expand cheapest node first:
Fringe is a priority queue

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.
**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 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 (in nodes)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N</td>
<td>$O(d^n)$</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^n)$</td>
<td>$O(b^n)$</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(b^n\log b)$</td>
<td>$O(b^n)$</td>
</tr>
</tbody>
</table>

* 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

---

**Search Heuristics**

- Any *estimate* of how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance

---

**Heuristics**

---

**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?

- Can we leverage the heuristic information in a more sound way?

→ A* search

We will cover that on Wednesday!