CS 188: Artificial Intelligence
Fall 2009

Lecture 3: A* Search
9/3/2009

Pieter Abbeel – UC Berkeley
Many slides from Dan Klein

Announcements

- Assignments:
  - Project 0 (Python tutorial): due Friday 1/28 at 4:59pm
  - Project 1 (Search): due Friday 2/4 at 4:59pm
    - Watch for office hour specifics — GSI project Czar!
  - Still looking for project partners? — Come to front after lecture.
  - Try pair programming, not divide-and-conquer
  - Account forms available up front during break and after lecture
- Lecture Videos: will be linked from lecture schedule
- Sections start tomorrow
  - Have fun solving exercises! Solutions will be posted online on Friday after last section.
  - After 2 weeks of section we will evaluate potential overcrowdedness issues and find a solution

Today

- Time and space complexity of DFS and BFS
- Iterative deepening --- “best of both worlds”
- Uniform cost search
- Greedy search
- A* search
  - Heuristic design
  - Admissibility, Consistency
- Tree search → Graph search

Recap: Search

- Search problem:
  - States (configurations of the world)
  - Successor function: a function from states to lists of (state, action, cost) triples; drawn as a graph
  - Start state and goal test
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)
- Search Algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)

General Tree Search

Function TREE-SEARCH (problem, strategy) returns a solution, or failure:
initiate the search tree using the initial state of problem
lookup:
  if there are no candidates for expansion then return failure
  choose a list 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?
**Example Search Tree**

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

**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?

<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^{s+1}) )</td>
<td>( O(b^{s+1}) )</td>
</tr>
</tbody>
</table>

**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>N</td>
<td>Y</td>
<td>( O(b^s) )</td>
<td>( O(b^s) )</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>

- 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.
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)</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>O(b^m-1)</td>
<td>O(b^m-1)</td>
</tr>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N*</td>
<td>O(b^m-1)</td>
<td>O(b^m-1)</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

Cost contours

Priority Queue Refresher

- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:
  - push(key, value) inserts (key, value) into the queue.
  - 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(b^m)</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>O(b^m-1)</td>
<td>O(b^m-1)</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>O(b^m^2)</td>
<td>O(b^m^2)</td>
</tr>
</tbody>
</table>

C/viriders

* 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
Uniform Cost Search Example

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)
Combining UCS and Greedy

- Uniform-cost orders by path cost, or backward cost $g(n)$
- Best-first orders by goal proximity, or forward cost $h(n)$

A* Search orders by the sum: $f(n) = g(n) + h(n)$

Example: Tag Grenager

When should A* terminate?

- Should we stop when we enqueue a goal?
  - No: only stop when we dequeue a goal

Is A* Optimal?

- What went wrong?
  - Actual bad goal cost < estimated good goal cost
  - We need estimates to be less than actual costs!

Admissible Heuristics

- A heuristic $h$ is **admissible** (optimistic) if:
  $$h(n) \leq h^*(n)$$
  where $h^*(n)$ is the true cost to a nearest goal

Examples:

- Coming up with admissible heuristics is most of what’s involved in using A* in practice.
Properties of A*

**Uniform-Cost**

- $b$

**A***

- $b$

UCS vs A* Contours

- Uniform-cost expanded in all directions

- A* expands mainly toward the goal, but does hedge its bets to ensure optimality

Example: Explored States with A*

Heuristic: manhattan distance ignoring walls

Comparison

- Greedy

- Uniform Cost

- A star

Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics

- Often, admissible heuristics are solutions to relaxed problems, with new actions (“some cheating”) available

- Inadmissible heuristics are often useful too (why?)

Example: 8 Puzzle

- What are the states?
- How many states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?
8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- \( h(\text{start}) = 8 \)

This is a relaxed-problem heuristic

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>UCS</th>
<th>TILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tiles misplaced</td>
<td>112</td>
<td>13</td>
</tr>
</tbody>
</table>

Average nodes expanded when optimal path has length...

<table>
<thead>
<tr>
<th>Steps</th>
<th>4 steps</th>
<th>8 steps</th>
<th>12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS</td>
<td>6.300</td>
<td>3.6 x 10^6</td>
<td></td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
</tbody>
</table>

8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total Manhattan distance

Why admissible?

- \( h(\text{start}) = 3 + 1 + 2 + \ldots \)
- \( = 18 \)

Average nodes expanded when optimal path has length...

<table>
<thead>
<tr>
<th>Steps</th>
<th>4 steps</th>
<th>8 steps</th>
<th>12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS</td>
<td>12</td>
<td>25</td>
<td>73</td>
</tr>
</tbody>
</table>

Trivial Heuristics, Dominance

- Dominance: \( h_a \geq h_b \) if \( \forall n : h_a(n) \geq h_b(n) \)
- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible
    - \( h(n) = \max(h_a(n), h_b(n)) \)

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic

Other A* Applications

- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- ...

Tree Search: 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?)

Optimality of A* Graph Search

Proof:
- New possible problem: nodes on path to \( G' \) that would have been in queue aren’t, because some worse \( n' \) for the same state as some \( n \) was dequeued and expanded first (disaster!)
- Take the highest such \( n \) in tree
- Let \( p \) be the ancestor which was on the queue when \( n' \) was expanded
- Assume \( f(p) < f(n) \)
- \( f(n') = f(n) \) because \( n' \) is suboptimal
- \( p \) would have been expanded before \( n' \)
- So \( n \) would have been expanded before \( n' \), too
- Contradiction!

Optimality

- Tree search:
  - \( A^* \) optimal if heuristic is admissible (and non-negative)
  - UCS is a special case (\( h = 0 \))

- Graph search:
  - \( A^* \) optimal if heuristic is consistent
  - UCS optimal (\( h = 0 \) is consistent)
  - Consistency implies admissibility
  - In general, natural admissible heuristics tend to be consistent

Consistency

- Wait, how do we know parents have better f-values than their successors?
- Couldn’t we pop some node \( n \), and find its child \( n' \) to have lower f value?

YES:

- What can we require to prevent these inversions?
- Consistency: \( c(n, a, n') \geq h(n) - h(n') \)
- Real cost must always exceed reduction in heuristic

Summary: A*

- \( A^* \) uses both backward costs and (estimates of) forward costs
- \( A^* \) is optimal with admissible heuristics
- Heuristic design is key: often use relaxed problems