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)

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

Office hours, Section

- Drop-in lab times: Wed 1/26 4-5pm in 271 Soda
- Office hours posted on the course website
- Sections starting this week:
  - Working through exercises are key for your understanding
  - Section handout contains several exercises similar to written 1
  - Solutions will be posted Wed 1pm (after last section)
  - Section 101: Tue 3-4pm
  - Section 104: Tue 4-5pm
  - Section 102: Wed 11-noon
  - Section 103: Wed noon-1pm

Today

- Iterative deepening
- Uniform cost search
- **A* Search**
- Heuristic Design

Announcements

- **Assignments:**
  - Project 0 (Python tutorial): due Thursday 1/28
  - Written 1 (Search): due Thursday 1/28
  - Project 1 (Search): to be released today, due Thursday 2/4
    - You don’t need to submit answers to P1 discussion questions
  - 5 slip days for projects; up to two per deadline
  - Try pair programming, not divide-and-conquer
- **Study materials**
  - Slides, Section materials, Assignments
  - Book

CS 188: Artificial Intelligence
Fall 2009

Lecture 3: A* Search
9/3/2009

Pieter Abbeel – UC Berkeley
Many slides from Dan Klein
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(bm)</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N^s</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^√c)</td>
<td>O(b^√c)</td>
</tr>
</tbody>
</table>

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.

Costs on Actions

Uniform Cost Search

Expand cheapest node first:
Fringe is a priority queue

expand

Cost contours

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>pq.push(key, value)</th>
<th>inserts (key, value) into the queue.</th>
</tr>
</thead>
<tbody>
<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 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)</td>
<td>O(bm)</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N^s</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^√c)</td>
<td>O(b^√c)</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

Best First / Greedy Search

- Expand the node that seems closest…

What can go wrong?

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: $f(n) = g(n) + h(n)$

Heuristics

- 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)

When should $A^*$ terminate?

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

Example: Toy Grenager
### 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

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

### Optimality of A*: Blocking

**Notation:**
- \( g(n) = \) cost to node \( n \)
- \( h(n) = \) estimated cost from \( n \) to the nearest goal (heuristic)
- \( f(n) = g(n) + h(n) = \) estimated total cost via \( n \)
- \( G^* = \) a lowest cost goal node
- \( G = \) another goal node

**Properties:**
- We’d have to pop a suboptimal goal \( G \) off the fringe before \( G^* \)
- \( G^* \) will be popped before \( G \)

### Optimality of A*: Blocking

**Proof:**
- What could go wrong?
- We’d have to pop a suboptimal goal \( G \) off the fringe before \( G^* \)
- This can’t happen:
  - Imagine a suboptimal goal \( G \) is on the queue
  - Some node \( n \) which is a subpath of \( G^* \) must also be on the fringe (why?)
  - \( n \) will be popped before \( G \)

### Properties of A*

**Uniform-Cost**

**A**

- Uniform-cost expanded in all directions
- \( A^* \) expands mainly toward the goal, but does hedge its bets to ensure optimality

### UCS vs A* Contours

- Uniform-cost expanded in all directions
- \( A^* \) expands mainly toward the goal, but does hedge its bets to ensure optimality
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, where new actions are available
- Inadmissible heuristics are often useful too (why?)

Trivial Heuristics, Dominance

- Dominance: \( h_a \geq h_i \) if
  \[ \forall n : h_a(n) \geq h_c(n) \]
- 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?)

Graph Search

- Idea: never expand a state twice
- How to implement:
  - Tree search + list of expanded states (closed list)
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state is new
- Python trick: store the closed list as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?
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’, too
- Contradiction!

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:

Consistency implies admissibility

In general, natural admissible heuristics tend to be consistent

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)
- 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