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

DFS and BFS diagrams:

- **DFS:**
  - m tiers
  - s tiers

- **BFS:**
  - 1 node
  - b nodes
  - b^2 nodes
  - b^s nodes
  - b^m nodes
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>w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>N*</td>
<td>O(b^{i+1})</td>
<td>O(b^{i+1})</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>N*</td>
<td>O(b^i)</td>
<td>O(bs)</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:

<table>
<thead>
<tr>
<th>Operation</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 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 w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^n)$</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>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(b^{C^*/\epsilon + 1})$</td>
<td>$O(b^{C^*/\epsilon + 1})$</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

<table>
<thead>
<tr>
<th></th>
<th>Straight-line distance to Bucharest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Brașov</td>
<td>461</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Constanța</td>
<td>506</td>
</tr>
<tr>
<td>Craiova</td>
<td>166</td>
</tr>
<tr>
<td>Drobeta</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Făgăraș</td>
<td>178</td>
</tr>
<tr>
<td>Iași</td>
<td>77</td>
</tr>
<tr>
<td>Iasi</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>344</td>
</tr>
<tr>
<td>Medias</td>
<td>24</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitești</td>
<td>98</td>
</tr>
<tr>
<td>Râmnicu Vâlcei</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>553</td>
</tr>
<tr>
<td>Târgu Mureș</td>
<td>229</td>
</tr>
<tr>
<td>Vâlcea</td>
<td>96</td>
</tr>
<tr>
<td>Vișeu</td>
<td>169</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
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* $h(n)$

```
S  1  a
  1  3  d
  1  2  G

h=6  h=5  h=2  h=0
```

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

Example: Teg Grenager

---

When should A* terminate?

- **Should we stop when we enqueue a goal?**

```
S  2  A  2  G
  h=3

h=2  h=0
```

- **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

- Example:

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

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

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

[Demo: contours UCS / $A^*$]
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: \( h_a \geq h_c \) 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 \( n' \)
- So \( n \) 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:

\[
\begin{align*}
&g = 10 \\
&h = 10 \\
&B \quad h = 0 \quad h = 8 \\
&A \quad h = 0 \\
&G \quad h = 8
\end{align*}
\]

- 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
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

**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