Informed search

Lirong Xia

Spring, 2017
Last class

Search problems

- state space graph: modeling the problem
- search tree: scratch paper for solving the problem

Uninformed search

- BFS
- DFS
Today’s schedule

- More on uninformed search
  - iterative deepening: BFS + DFS
  - uniform cost search (UCS)
  - searching backwards from the goal

- Informed search
  - best first (greedy)
  - A*
Combining good properties of BFS and DFS

- **Iterative deepening DFS:**
  - Call limited depth DFS with depth 0;
  - If unsuccessful, call with depth 1;
  - If unsuccessful, call with depth 2;
  - Etc.

- Complete, finds shallowest solution
- Flexible time-space tradeoff
- May seem wasteful timewise because replicating effort
Costs on Actions
Uniform-Cost Search (UCS)

- BFS: finds shallowest solution
- **Uniform-cost search (UCS):** work on the lowest-cost node first
  - so that all states in the fringe have more or less the same cost, therefore “uniform cost”
- If it finds a solution, it will be an optimal one
- Will often pursue lots of short steps first
- Project 1 Q3
Example
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>

• Insertions aren’t constant time, usually
• We’ll need priority queues for cost-sensitive search methods
Uniform-Cost Search

➢ The good: UCS is
  • complete: if a solution exists, one will be found
  • optimal: always find the solution with the lowest cost
  • really?
    ▪ yes, for cases where all costs are positive

➢ The bad
  • explores every direction
  • no information about the goal location
Searching backwards from the goal

- Switch the role of START and GOAL
- Reverse the edges in the problem graph
- Need to be able to compute *predecessors* instead of successors
Informed search

- So far, have assumed that no non-goal state looks better than another
- Unrealistic
  - Even without knowing the road structure, some locations seem closer to the goal than others
- Makes sense to expand seemingly closer nodes first
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)

• Strategy: expand a node that you think is closest to a goal state
  • **Heuristic function**: estimate of distance to nearest goal for each state
  • $h(n)$
  • Often easy to compute

• A common case:
  • Best-first takes you straight to the (wrong) goal

• Worst-case: like a badly-guided DFS
Example

\[ h(b) = h(c) = h(d) = h(e) = h(f) = h(g) = 1 \]
\[ h(a) = 2 \]
A more problematic example

Greedy will choose SbcdG
Optimal: SaG
Should we stop when the fringe is empty?

h(b) = h(c) = h(d) = 1
h(a) = 2
h(G) = 0
A*: Combining UCS and Greedy

- **Uniform-cost** orders by path cost $g(n)$
- **Greedy** orders by goal proximity, or forward cost $h(n)$

*A* search orders by the sum:

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

![Graph](image)
When should A* terminate?

• Should we stop when we add a goal to the fringe?

• No: only stop when a goal pops from the fringe

![Diagram of A* algorithm with nodes S, a, b, and c, and their heuristic values h(S)=7, h(a)=2, h(b)=0, h(G)=0, and edge weights 2, 2, 3.](image)
**A\(^*\)**

- Never expand a node whose state has been visited
- Fringe can be maintained as a *priority* queue
- Maintain a set of visited states
- \(fringe := \{\text{node corresponding to initial state}\}\)
- **loop:**
  - if fringe empty, declare failure
  - choose and remove the top node \(v\) from fringe
  - check if \(v\)'s state \(s\) is a goal state; if so, declare success
  - if \(v\)'s state has been visited before, skip
  - if not, expand \(v\), insert resulting nodes with \(f(v)=g(v)+h(v)\) to fringe
Problem: overestimate the true cost to a goal
    • so that the algorithm terminates without expanding the actual optimal path
Admissible Heuristics

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

• Examples:
  • $h(n) = 0$
  • $h(n) = h^*(n)$
  • another example

• Coming up with admissible heuristics is most challenging part in using A* in practice
Is Admissibility enough?

*Problem:* the first time to visit a state may not via the shortest path to it
Consistency of Heuristics

- Stronger than admissibility
- Definition:
  \[ \text{cost}(a \to c) + h(c) \geq h(a) \]
  - Triangle inequality
  - Be careful about the edges
- Consequences:
  - The \( f \) value along a path never decreases
  - so that the first time to visit a state corresponds to the shortest path
Violation of consistency

\[ h(S) = 7 \]
\[ h(a) = 0 \]
\[ h(b) = 6 \]
\[ h(c) = 0 \]
\[ h(G) = 0 \]

\[ b \rightarrow c \]
Correctness of A*

- A* finds the optimal solution for consistent heuristics
  - but not for the admissible heuristics (as in the previous example)
- However, if we do not maintain a set of “visited” states, then admissible heuristics are good enough
Proof

Proof by contradiction: suppose the output is incorrect.

- Let the optimal path be $S \rightarrow a \rightarrow \ldots \rightarrow G$
  - for simplicity, assume it is unique
- When $G$ pops from the fringe, $f(G)$ is no more than $f(v)$ for all $v$ in the fringe.
- Let $v$ denote the state along the optimal path in the fringe
  - guaranteed by consistency
- Contradiction
  - addimisbility: $f(G) \leq f(v) \leq g(v) + h^*(v)$
Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
Example: 8 Puzzle

- What are the states?
- How many states?
- What are the actions?
- What should the costs be?
8 Puzzle I

- Heuristic: Number of tiles misplaced

- Why is it admissible?

- \( h(\text{START}) = 8 \)

Start State

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Goal State

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Average nodes expanded when optimal path has length....

<table>
<thead>
<tr>
<th></th>
<th>...4 steps</th>
<th>...8 steps</th>
<th>...12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS</td>
<td>112</td>
<td>6,300</td>
<td>3,600,000</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?
- $h(\text{START}) = 3+2+\ldots=18$
8 Puzzle III

- Total Manhattan distance for grid 1 and 2
- Why admissible?
- Is it consistent?
- $h(\text{START}) = 3+1=4$
8 Puzzle IV

• How about using the actual cost as a heuristic?
  • Would it be admissible?
  • Would we save on nodes expanded?
  • What’s wrong with it?

• With A*: a trade-off between quality of estimate and work per node!
Other A* Applications

- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- Voting!
Features of A*

- A* uses both backward costs and (estimates of) forward costs
- A* computes an optimal solution with consistent heuristics
- Heuristic design is key
Search algorithms

- Maintain a fringe and a set of visited states

- \textit{fringe} := \{node corresponding to initial state\}

- \textit{loop:}
  - if fringe empty, declare failure
  - choose and remove the top node \( v \) from fringe
  - check if \( v \)'s state \( s \) is a goal state; if so, declare success
  - if \( v \)'s state has been visited before, skip
  - if not, expand \( v \), insert resulting nodes to fringe

- Data structure for fringe
  - FIFO: BFS
  - Stack: DFS
  - Priority Queue: UCS (value = \( h(n) \)) and A* (value = \( g(n)+h(n) \))
Take-home message

No information

- BFS, DFS, UCS, iterative deepening,
  backward search

With some information

- Design heuristics $h(n)$
- A*, make sure that $h$ is consistent
- Never use Greedy!
You can start to work on UCS and A* parts

Start early!

Ask questions on Piazza