CS 343: Artificial Intelligence

Informed Search

Prof. Scott Niekum

University of Texas at Austin

[These slides based on ones created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]
Today

- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search

- Graph Search
Recap: Search

- **Search problem:**
  - States (configurations of the world)
  - Actions and costs
  - Successor function (world dynamics)
  - 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)
  - Optimal: finds least-cost plans
Example: Pancake Problem

Cost: Number of pancakes flipped
Example: Pancake Problem

State space graph with costs as weights
General Tree Search

function TREE-SEARCH(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
    if there are no candidates for expansion then return failure
    choose a leaf 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

Action: flip top two
Cost: 2

Path to reach goal:
Flip four, flip three
Total cost: 7
All these search algorithms are the same except for fringe strategies

- Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
- Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
- Can even code one implementation that takes a variable queuing object
Uninformed Search
Uniform Cost Search

- Strategy: expand lowest path cost
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every “direction”
  - No information about goal location
Video of Demo Contours UCS Empty
Video of Demo Contours UCS Pacman Small Maze
Informed Search
A heuristic is:
- A function that *estimates* how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance for pathing
Example: Heuristic Function

\[ h(x) \]

Straight-line distance
to Bucharest

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>266</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hiasuva</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</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>Rimnicu Vâlcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timișoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Example: Heuristic Function

Heuristic: the number of the largest pancake that is still out of place
Greedy Search
Example: Heuristic Function

$h(x)$
Greedy Search

- Expand the node that seems closest...

- What can go wrong?
Greedy Search

- **Strategy:** expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state

- **Best case:**
  - Best-first takes you straight to the nearest goal

- **A common case:**
  - Suboptimal route to goal due to imperfect heuristic
  - Does not lead to nearest goal

- **Worst-case:** like a badly-guided DFS
Video of Demo Contours Greedy (Empty)
Video of Demo Contours Greedy (Pacman Small Maze)
A* Search
A* Search
Combining UCS and Greedy

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

![Diagram showing the combination of UCS and Greedy search algorithms. The diagram includes nodes labeled $S$, $a$, $b$, $c$, $d$, $e$, and $G$, with edges and costs illustrating the path costs and heuristic values.](image)

- **A* Search** orders by the sum: $f(n) = g(n) + h(n)$
When should $A^*$ terminate?

- Should we stop when we enqueue a goal?
  - No: only stop when we expand a goal
Is A* Optimal?

- What will A* do here?
- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!
Idea: Admissibility

Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe.

Admissible (optimistic) heuristics can still help to delay the evaluation of bad plans, but never overestimate the true costs.
A heuristic $h$ is **admissible** (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

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.
Optimality of A* Tree Search
Optimality of A* Tree Search

Assume:
- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:
- A will exit the fringe before B
Optimality of A* Tree Search: Blocking

Proof:
- Imagine B is on the fringe
- Some ancestor $n$ of A is on the fringe, too (maybe A!)
- Claim: $n$ will be expanded before B
  1. $f(n)$ is less or equal to $f(A)$

\[ f(n) = g(n) + h(n) \]
\[ f(n) \leq g(A) \]
\[ g(A) = f(A) \]

Definition of f-cost
Admissibility of $h$
$h = 0$ at a goal
Optimality of A* Tree Search: Blocking

Proof:
- Imagine B is on the fringe
- Some ancestor $n$ of A is on the fringe, too (maybe A!)
- Claim: $n$ will be expanded before B
  1. $f(n)$ is less or equal to $f(A)$
  2. $f(A)$ is less than $f(B)$

\[ g(A) < g(B) \]
\[ f(A) < f(B) \]

B is suboptimal

$h = 0$ at a goal
Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor $n$ of A is on the fringe, too (maybe A!)
- Claim: $n$ will be expanded before B
  1. $f(n)$ is less or equal to $f(A)$
  2. $f(A)$ is less than $f(B)$
  3. $n$ expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal
Properties of A*
Properties of A*
UCS vs A* Contours

- Uniform-cost expands equally in all “directions”
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality
Video of Demo Contours (Empty) -- UCS
Video of Demo Contours (Empty) -- Greedy
Video of Demo Contours (Empty) – A*
Pacman - A*
Pacman - Greedy
Comparison

Greedy  Uniform Cost  A*
Guess algorithm (DFS / BFS / UCS / Greedy / A*)
Guess algorithm (DFS / BFS / UCS / Greedy / A*)
Guess algorithm (DFS / BFS / UCS / Greedy / A*)
Guess algorithm (DFS / BFS / UCS / Greedy / A*)
Guess algorithm (DFS / BFS / UCS / Greedy / A*)
A* Applications

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

[Diagram of a video game interface]
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.
Example: 8 Puzzle

- What are the states?
- How many states?
- What are the actions?
- How many successors 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

Average nodes expanded when the optimal path has...

<table>
<thead>
<tr>
<th>Steps</th>
<th>UCS</th>
<th>TILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>...4 steps</td>
<td>112</td>
<td>13</td>
</tr>
<tr>
<td>...8 steps</td>
<td>6,300</td>
<td>39</td>
</tr>
<tr>
<td>...12 steps</td>
<td>3.6 x 10^6</td>
<td>227</td>
</tr>
</tbody>
</table>
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 is it admissible?

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

---

<table>
<thead>
<tr>
<th>Average nodes expanded when the optimal path has...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...4 steps</td>
</tr>
<tr>
<td><strong>TILES</strong></td>
</tr>
<tr>
<td><strong>MANHATTAN</strong></td>
</tr>
</tbody>
</table>
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
- As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself
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
Graph Search
Failure to detect repeated states can cause exponentially more work.
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 + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

- **Important:** store the closed set as a set, not a list

- **Can graph search wreck completeness?** Why/why not?

- **How about optimality?**
A* Graph Search Gone Wrong?

**State space graph**

- **S** (h=2) to **A** (h=4): 1
- **S** (h=2) to **B** (h=1): 1
- **B** (h=1) to **C** (h=1): 2
- **A** (h=4) to **C** (h=1): 1
- **C** (h=1) to **G** (h=0): 3

**Search tree**

- **S** (0+2)
  - **A** (1+4)
    - **C** (2+1)
      - **G** (5+0)
  - **B** (1+1)
    - **C** (3+1)
    - **G** (6+0)
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - **Admissibility:** heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from } A \text{ to } G \]
  - **Consistency:** heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost}(A \text{ to } C) \]
    i.e. if the true cost of an edge from A to C is X, then the h-value should not decrease by more than X between A and C.

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
  - A* graph search is optimal
Sketch: consider what A* does with a consistent heuristic:

- Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
- Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
- Result: A* graph search is optimal
Optimality

- **Tree search:**
  - $A^*$ is optimal if heuristic is admissible
  - 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, most natural admissible heuristics tend to be consistent, especially if from relaxed problems
A*: Summary
A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(make-node(initial-state[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
end
function GraphSearch(problem, fringe) return a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
        add STATE[node] to closed
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
end