CS 188: Artificial Intelligence

Search Problems

Instructors: Saagar Sanghavi, Nicholas Tomlin

University of California, Berkeley

(slides adapted from Dan Klein, Pieter Abbeel, Anca Dragan, Stuart Russell)
Utilities and Rationality

Rational Preferences

MEU Principle

Orderability: \((A > B) \lor (B > A) \lor (A \sim B)\)

Transitivity: \((A > B) \land (B > C) \implies (A > C)\)

Continuity: \((A > B > C) \implies \exists p \ [p, A; 1-p, C] \sim B\)

Substitutability: \((A \sim B) \implies [p, A; 1-p, C] \sim [p, B; 1-p, C]\)

Monotonicity: \((A > B) \implies (p \geq q) \iff [p, A; 1-p, B] \succeq [q, A; 1-q, B]\)
Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
Agents that Plan
Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world’s current state
  - Do not consider the future consequences of their actions
  - *Consider how the world IS*

- Can a reflex agent be rational?
Video of Demo Reflex Optimal
Video of Demo Reflex Odd
Planning Agents

- Planning agents:
  - Ask "what if"
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE

- Optimal vs. complete planning

- Planning vs. replanning
Video of Demo Mastermind
Search Problems
Search Problems

- A search problem consists of:
  - A state space
  - A successor function (with actions, costs)
  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state
Search Problems Are Models
Example: Traveling in Romania

- **State space:**
  - Cities

- **Successor function:**
  - Roads: Go to adjacent city with cost = distance

- **Start state:**
  - Arad

- **Goal test:**
  - Is state == Bucharest?

- **Solution?**
What’s in a State Space?

The world state includes every last detail of the environment

A search state keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

- Problem: Eat-All-Dots
  - States: {(x,y), dot boolean}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false
State Space Sizes?

- **World state:**
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- **How many**
  - World states?
    - $120 \times (2^{30}) \times (12^2) \times 4$
  - States for pathing?
    - 120
  - States for eat-all-dots?
    - $120 \times (2^{30})$
State Space Graphs and Search Trees
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a state space graph, each state occurs only once!

- We can rarely build this full graph in memory (it’s too big), but it’s a useful idea
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a state space graph, each state occurs only once!

- We can rarely build this full graph in memory (it’s too big), but it’s a useful idea
Search Trees

- A search tree:
  - A “what if” tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

```
This is now / start
Possible futures
```

```
“N”, 1.0  “E”, 1.0
```

```
This is now / start
Possible futures
```

```
“N”, 1.0  “E”, 1.0
```

```
This is now / start
Possible futures
```

```
“N”, 1.0  “E”, 1.0
```

```
This is now / start
Possible futures
```

```
“N”, 1.0  “E”, 1.0
```

```
This is now / start
Possible futures
```
State Space Graphs vs. Search Trees

Each NODE in the search tree is an entire PATH in the state space graph. We construct only what we need on demand.
Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

How big is its search tree (from S)?
Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

How big is its search tree (from S)?

Important: Lots of repeated structure in the search tree!
Tree Search
Search Example: Romania
Searching with a Search Tree

- **Search:**
  - Expand out potential plans (tree nodes)
  - Maintain a **fringe** of partial plans under consideration
  - Try to expand as few tree nodes as possible
General Tree Search

**Important ideas:**
- Fringe
- Expansion
- Exploration strategy

**Main question:** which fringe nodes to explore?

```plaintext
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
```
Example: Tree Search
Example: Tree Search
Depth-First Search
Depth-First Search

Strategy: expand a deepest node first

Implementation:
Fringe is a LIFO stack
Search Algorithm Properties
Search Algorithm Properties

- **Complete:** Guaranteed to find a solution if one exists?
- **Optimal:** Guaranteed to find the least cost path?
- **Time complexity?**
- **Space complexity?**

**Cartoon of search tree:**
- $b$ is the branching factor
- $m$ is the maximum depth
- Solutions at various depths

**Number of nodes in entire tree?**
- $1 + b + b^2 + \ldots + b^m = O(b^m)$
Depth-First Search (DFS) Properties

- **What nodes DFS expand?**
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$

- **How much space does the fringe take?**
  - Only has siblings on path to root, so $O(bm)$

- **Is it complete?**
  - m could be infinite, so only if we prevent cycles (more later)

- **Is it optimal?**
  - No, it finds the “leftmost” solution, regardless of depth or cost
Breadth-First Search
Breadth-First Search

Strategy: expand a shallowest node first

Implementation:
Fringe is a FIFO queue
Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be $s$
  - Search takes time $O(b^s)$

- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$

- Is it complete?
  - $s$ must be finite if a solution exists, so yes!

- Is it optimal?
  - Only if costs are all 1 (more on costs later)
Video of Demo Maze Water DFS/BFS (part 1)
Video of Demo Maze Water DFS/BFS (part 2)
Iterative Deepening

- Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution...
  - Run a DFS with depth limit 2. If no solution...
  - Run a DFS with depth limit 3. ..... 

- Isn’t that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!
BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.
Uniform Cost Search
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs $C^*$ and arcs cost at least $\varepsilon$, then the “effective depth” is roughly $C^*/\varepsilon$
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)

- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!

- Is it optimal?
  - Yes! (Proof via A*)
Uniform Cost Issues

- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location

- We’ll fix that soon!
Video of Demo Empty UCS
Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 1)
Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 2)
Demo Maze with Deep/Shallow Water --- DFS, BFS, or UCS? (part 3)
The One Queue

- All these search algorithms are the same except for fringe strategies
  - DFS: Fringe is a Stack
  - BFS: Fringe is a Queue
  - UCS: Fringe is a PriorityQueue
  - Can even code one implementation that takes a variable queuing object
Up next: Informed Search

- Uninformed Search
  - DFS
  - BFS
  - UCS

- Informed Search (Heuristics)
  - Greedy Search
  - A* Search
Search Heuristics

- **A heuristic is:**
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Pathing?
  - Examples: Manhattan distance, Euclidean distance
Example: Heuristic Function

<table>
<thead>
<tr>
<th>City</th>
<th>Distance to Bucharest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobrogea</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>Hirssova</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>Pitesti</td>
<td>98</td>
</tr>
<tr>
<td>Rimenicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</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>

$h(x)$
Greedy Search
Greedy Search

- Expand the node that seems closest…
  - Move to smallest heuristic value

- Is it optimal?
  - No. Resulting path to Bucharest is not the shortest!
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* $h(n)$

- **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?
  - No: only stop when we dequeue a goal
A* Demo, with s = 0, goal = 6. (Credit: Josh Hug)

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

<table>
<thead>
<tr>
<th>#</th>
<th>distTo</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>- 1</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
<td>- 3</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>- 15</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>- 5</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>- ∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
<td>- 0</td>
</tr>
</tbody>
</table>

h(v, goal) is arbitrary. In this example, it’s the min weight edge out of each vertex.

Fringe: [(1: ∞), (2: ∞), (3: ∞), (4: ∞), (5: ∞), (6: ∞)]
A* Demo, with $s = 0$, goal = 6.

Insert all vertices into fringe $PQ$, storing vertices in order of $d(source, v) + h(v, goal)$.

Repeat: Remove best vertex $v$ from $PQ$, and relax all edges pointing from $v$.

<table>
<thead>
<tr>
<th>#</th>
<th>distTo</th>
<th>edgeTo</th>
<th>$h(v, goal)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>-</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Fringe: $[(1: 5), (2: 16), (3: \infty), (4: \infty), (5: \infty), (6: \infty)]$
A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Fringe: [(2: 16), (3: ∞), (4: ∞), (5: ∞), (6: ∞)]
A* Demo, with \( s = 0 \), goal = 6.

Insert all vertices into fringe \( PQ \), storing vertices in order of \( d(\text{source}, v) + h(v, \text{goal}) \).

Repeat: Remove best vertex \( v \) from \( PQ \), and relax all edges pointing from \( v \).

<table>
<thead>
<tr>
<th>#</th>
<th>distTo</th>
<th>edgeToh(v, goal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>- 1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0 3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0 15</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>1 2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1 1</td>
</tr>
<tr>
<td>5</td>
<td>( \infty )</td>
<td>- ( \infty )</td>
</tr>
<tr>
<td>6</td>
<td>( \infty )</td>
<td>- 0</td>
</tr>
</tbody>
</table>

Fringe: \([(4: 6), (3: 15), (2: 16), (5: \infty), (6: \infty)]\)

Which vertex is removed next?
A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of \( d(\text{source, v}) + h(v, \text{goal}) \).

Repeat: Remove best vertex \( v \) from PQ, and relax all edges pointing from \( v \).

- Give distTo, edgeTo, \( h(v, \text{goal}) \), and fringe after relaxation.

<table>
<thead>
<tr>
<th>#</th>
<th>distTo</th>
<th>edgeTo</th>
<th>( h(v, \text{goal}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>4</td>
<td>( \infty )</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Fringe: \([(6: 10), (3: 15), (2: 16), (5: \infty)]\)
A* Demo, with $s = 0$, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of $d$(source, $v$) + $h$(v, goal).

Repeat: Remove best vertex $v$ from PQ, and relax all edges pointing from $v$.

Next vertex to be dequeued is our target, so we’re done!

Fringe: [(6: 10), (3: 15), (2: 16), (5: $\infty$)]
A* Demo, with $s = 0$, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of $d(\text{source}, v) + h(v, \text{goal})$.

Repeat: Remove best vertex $v$ from PQ, and relax all edges pointing from $v$.

#     distTo     edgeTo$h(v, \text{goal})$
0     0          - 1
1     2          0 3
2     1          0 15
3     13         1 2
4     5          1 1
5     9          4 $\infty$
6     10         4 0

Observations:
- Not every vertex got visited.
- Result is not a shortest paths tree for vertex zero (path to 3 is suboptimal!), but that’s OK because we only care about path to 6.
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
Idea: Admissibility

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

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs.
A heuristic $h$ is *admissible* (optimistic) iff:

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

where $h^*(n)$ 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 \( f(A) \)

\[
f(n) = g(n) + h(n) \quad \text{Definition of f-cost}
\]

\[
f(n) \leq g(A) \quad \text{Admissibility of h}
\]

\[
g(A) = f(A) \quad \text{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 \( f(B) \)

\[
\begin{align*}
g(A) &< g(B) & \text{B is suboptimal} \\
f(A) &< f(B) & h = 0 \text{ at a goal}
\end{align*}
\]
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*

Uniform-Cost

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

[Demo: contours UCS / greedy / A* empty (L3D1)]
[Demo: contours A* pacman small maze (L3D5)]
Video of Demo Contours (Empty) -- UCS
Video of Demo Contours (Empty) -- Greedy
Video of Demo Contours (Empty) – A*
Demo Contours (Pacman Small Maze) – A*
Comparison

Greedy  Uniform Cost  A*
Creating Heuristics
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

<table>
<thead>
<tr>
<th>Average nodes expanded when the optimal path has...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...4 steps</td>
</tr>
<tr>
<td>UCS</td>
</tr>
<tr>
<td>TILES</td>
</tr>
</tbody>
</table>

Statistics from Andrew Moore
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 is it admissible?

  \[ 3 + 1 + 2 + \ldots = 18 \]

- \( h(\text{start}) = \)

<table>
<thead>
<tr>
<th></th>
<th>...4 steps</th>
<th>...8 steps</th>
<th>...12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
<tr>
<td>MANHATTAN</td>
<td>12</td>
<td>25</td>
<td>73</td>
</tr>
</tbody>
</table>
8 Puzzle III

- 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
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.
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 + 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

Search tree

Closed Set: S B C A
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(v) \leq h^*(v) \text{ for all } v \in V \]
    Underestimate the true cost to the goal!
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(v) - h(u) \leq d(u, v) \text{ for all } (u, v) \in E \]
    Underestimate the weight of every edge!

- **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
Optimality of A* Search

- With a admissible heuristic, Tree A* is optimal.
- With a consistent heuristic, Graph A* is optimal.
  - With h=0, the same proof shows that UCS is optimal.
Optimality of A* Graph Search
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 it comes from a relaxed problem
Tree Search Pseudo-Code

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 **Graph-Search**(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
Search and Models

- Search operates over models of the world
  - The agent doesn’t actually try all the plans out in the real world!
  - Planning is all “in simulation”
  - Your search is only as good as your models...
Search Gone Wrong?