Game Engineering
CS420-2011F-12
Artificial Intelligence

David Galles

Department of Computer Science
University of San Francisco
Artificial Intelligence

- AI in games is a huge field
  - Creating a believable world
    - Characters with their own apparent goals and desires, especially in RPGs and open world games
    - Opponents that seem to think and plan
  - Simulating human players
    - Chess players, FPS “bots”, strategy game opponents, etc
Most AI is Faked ...

- ... which in unsurprising, since most everything is faked, if possible
- Don’t need to have intelligent enemies, just need to appear intelligent
- Surprisingly large quantity is done with Finite State Machines
12-2: Finite state machines

• Each entity has a number of states, that represent behaviors
  • Patrolling, advancing to a position, searching, running away, finding cover, etc

• Each behavior can be relatively simple

• Transitions between behaviors can be triggered by timers, scripting, “sensing” by entities, etc
Creating a stealth-based action game (Thief, Splinter Cell, Metal Gear Solid, etc)
- Patrol state (traversing between waypoints)
- Alerted state (simple search pattern)
- Attacking state (advance towards player, attack)

Each behavior is relatively simple, well-managed transitions between them (especially scripted transitions) can lead to very intelligent-seeming enemies. Add in some random audio cues, and the enemies can seem quite smart ...
12-4: Pathfinding

- One aspect of traditional AI that is commonly used in games is pathfinding
  - RTS units getting from home base to place they are attacking
  - Enemies attacking player in a maze-style game
  - Bots finding shortest route to powerups / other players / etc in FPSs
- First step: Simplifying the problem
12-5: Pathfinding

• Navigating a real-life (or even complex simulated) environment is tricky
• Vastly simplify the search space, make it a standard CS-style graph
  • Waypoint System
  • Navigation Mesh
• 2D games (RTS, etc), can be easier – just use a grid
12-6: Pathfinding

• OK, so we’ve simplified the problem to searching for a path in a (potentially very complicated) graph
  • Verticies (places AI can go)
  • Edges (links between verticies, cost – often just a distance, can be more complicated)
• How do we efficiently search the graph?
12-7: Breadth-First Search

- Examine all nodes that are 1 unit away
- Examine all nodes that are 2 units away
- ...
- Examine all nodes that are $n$ units away

(Examples)
A few more wrinkles:

- Searching a graph instead of a tree
- Get to the same node in more than one way
- Once we’ve found shortest path to a path to a node, don’t need to consider any other paths
Breadth-First Search

- Maintain two data structures
  - “Open List” – search horizon
  - “Closed list” – nodes we’ve already found the shortest path to, don’t need to examine again
void BFS(Graph G, Vertex v) {

    Queue Q = new Queue();
    Closed = new ClosedList();

    Q.enqueue(v);
    while (!Q.empty()) {
        nextV = Q.dequeue()
        if (v not in Closed)
            { 
                Closed.Add(v);
                foreach (Vertex neighbor adjacent to v in G)
                    Q.enqueue(neighbor);
            }
    }
}
Breadth-First Search

- Problem #1 with BFS:
  - Assumes uniform edge cost
  - Not actually true with most graphs we will be searching
- Solution?
Best-first Search

- Uniform-cost search
  - Store node \textit{and cost to get to node} in queue
  - Use a priority queue instead of a standard queue
  - Always choose the cheapeast node to expand
    - “Expand” means examine children of node
Uniform-Cost Search

- **Uniform-Cost Pseudocode**

  ```python
  enqueue(initialState)
  do
      node = priority-dequeue()
      if (node not in closed list)
          add node to closed list
          if goalTest(node)
              return node (potentially path as well)
          else
              children = successors(node)
              for child in children
                  priority-enqueue(child, dist(child))
  ```

- **dist** is the cost of the path from the initial state to the child node

(EXAMPLES!)
Problem with Uniform cost search

To find a goal that is 100 units away from the start, we examine all nodes that are 100 units away from the start

RTS example on board

Make a minor change to Uniform cost search, make it much more general
enqueue(initialState)
do
    node = priority-dequeue()
    if (node not in closed list)
        add node to closed list
        if goalTest(node)
            return node (potentially path as well)
    else
        children = successors(node)
        for child in children
            priority-enqueue(child, f(child))

• \( f(n) \) is a function that describes how “good” a node is
(Almost) all searches are instances of best-first, with different evaluation functions $f$.

What functions $f$ would yield the following searches:
- Depth-First Search
- Breadth-First Search
- Uniform Cost Search
12-17: **Best-first Search**

- (Almost) all searches are instances of best-first, with different evaluation functions $f$
- What functions $f$ would yield the following searches:
  - Breadth-First Search $f(n) = \text{depth}(n)$
  - Depth-First Search $f(n) = -\text{depth}(n)$
  - Uniform Cost Search $f(n) = g(n)$ (actual cost to get to n)
A Heuristic Function $h(n)$ is an estimate of how much it would cost to get to the solution from node $n$.

- $h(n)$ is not perfect
  - What could we do if $h$ was perfect?

Example heuristic: Route planning: straight-line distance to the goal.

- How could we use a heuristic function as part of best-first search to find a goal quickly?
12-19: Greedy Search

- Best-First search with $f(n) = h(n)$
- Route-planning example: Always travel to the city that looks like it is closest to out destination
12-20: Greedy Search Example

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</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>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</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>100</td>
</tr>
<tr>
<td>Rimnicu 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>
## Greedy Search Example

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</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>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</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>100</td>
</tr>
<tr>
<td>Rimnicu 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>

(A, 336)
(S, 253), (T, 329), (Z, 374)
(F, 176), (RV, 193), (T, 329), (A, 336), (Z, 374), (O, 380)
(B, 0), (RV, 193), (S, 253), (T, 329), (A, 336), (Z, 374), (O, 380)

Solution: A → S → F → B

Optimal: A → S → RV → P → B
Greedy Search Problems

• Optimal solution can involve moving ‘away’ from goal
  • Sliding tile puzzle: “undo” a partial solution
  • Rubic’s cube: “Mess up” part of cube to solve

• Not really moving away from goal – as a measure of the number of moves to a solution, you are actually getting closer to the goal. Only relative to your heuristic function are you going backwards
  • Perfect $h = $ no need to search
Greedy Search Problems

- Greedy search has similar strengths / weaknesses to DFS
  - Expands a linear number of nodes
  - Not optimal
  - May not even necessarily find goal (depending upon the heuristic function)
- What are the flaws of greedy search?
- How could we fix them?
A* search is a combination of uniform cost search and greedy search.

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

- \( g(n) \) = current path cost
- \( h(n) \) = heuristic estimate of distance to goal.

Favors nodes with best estimated total cost to goal.

If \( h(n) \) satisfies certain conditions, A* is both complete (always finds a solution) and optimal (always finds the best solution).
A* Search Example

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</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>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</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>100</td>
</tr>
<tr>
<td>Rimnicu 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>
12-26: A* Search Example

- Arad = 0 + 366 = 366
- (dequeue A: g = 0)  S = 140 + 253 = 393, T = 118 + 329 = 447, Z = 75 + 374 = 449
- (dequeue S: g = 140)  RV = 220 + 193 = 413, F = 239 + 176 = 415, T = 118 + 329 = 447, Z = 374 + 75 = 449, A = 280 + 336 = 616, O = 291 + 380 = 671,
- (dequeue RV: g = 220)  F = 239 + 176 = 415, P = 317 + 100 = 417, T = 118 + 329 = 447, Z = 374 + 75 = 449, C = 366 + 160 = 526, S = 300 + 253 = 553, A = 280 + 336 = 616, O = 291 + 380 = 671
- (dequeue F: g = 239)  P = 317 + 100 = 417, T = 118 + 329 = 447, Z = 374 + 75 = 449, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, S = 338 + 253 = 591, A = 280 + 336 = 616, O = 291 + 380 = 671
12-27: A* Search Example

- (dequeue P: g = 317) T = 118 + 329 = 447, Z = 374 + 75 = 449, B = 518 + 0 = 518, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671

- (dequeue T: g = 118) Z = 374 + 75 = 449, L = 229 + 244 = 473, B = 518 + 0 = 518, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671

- (dequeue Z: g = 75) L = 229 + 244 = 473, A = 150 + 336 = 486, B = 518 + 0 = 518, O = 146 + 380 = 526, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671
12-28: A* Search Example


- (dequeue B: g = 518) solution. A -> S -> RV -> P -> B
12-29: A* Example II
12-30: A* Example II
12-31: A* Example II

5 4 3 2 3 2
A --- B --- C --- G --- H --- J
| | | | | |
D --- E --- F --- I --- K --- L
3 2 1 4 1 0
12-32: A* Example III

Start

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Goal
12-33: A* Example III
12-34: A* Example III

Start

Goal

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
</tbody>
</table>
12-35: A* Example IV

Node expansion order for BFS, Uniform Cost, Greedy, A*

h() values in yellow
edge costs in white
A* Example IV

- **BFS:**
  - AGBJHCEKDIML (goal found)
  - (Other orderings are possible)
A* Example IV

- Uniform Cost Search:
  - ABCGEJDHFIMKL
  - (Other orderings are possible)
A* Example IV

- Greedy
  - AJKL
  - (Other orderings are possible)
A* Example IV

- A*
  - ABCFIEMJL
  - (Other orderings are possible)
A* is optimal (finds the shortest solution) as long as our $h$ function is **admissible**.

- Admissible: always underestimates the cost to the goal.

Proof: When we dequeue a goal state, we see $g(n)$, the actual cost to reach the goal. If $h$ underestimates, then a more optimal solution would have had a smaller $g + h$ than the current goal, and thus have already been dequeued.

Or: If $h$ overestimates the cost to the goal, it’s possible for a good solution to “look bad” and get buried further back in the queue.
Notice that we can’t discard repeated states.
  - We could always keep the version of the state with the lowest $g$

More simply, we can also ensure that we always traverse the best path to a node first.

A *monotonic* heuristic guarantees this.

A heuristic is monotonic if, for every node $n$ and each of its successors $(n')$, $h(n)$ is less than or equal to $stepCost(n, n') + h(n')$.
  - In geometry, this is called the triangle inequality.
12-42: Optimality of A*

- SLD is monotonic. (In general, it’s hard to find realistic heuristics that are admissible but not monotonic).
- Corollary: If \( h \) is monotonic, then \( f \) is nondecreasing as we expand the search tree.
- Alternative proof of optimality.
- Notice also that UCS is A* with \( h(n) = 0 \)
- A* is also *optimally efficient*
  - No other complete and optimal algorithm is guaranteed to expand fewer nodes.
• Is $h()$ admissible?
• Is $h()$ monotonic?
12-44: \textbf{A* Example II}

Node: Queue:

\[
\text{[(A f = 17, g = 0, h = 17)]}
\]
Node: Queue :

A  [(C f = 22, g = 7, h = 15), (B f = 28, g = 8, h = 20)]
12-46: A* Example II

Node: Queue:
C [(D f = 23, g = 15, h = 8), (B f = 28, g = 8, h = 20)]
Node: Queue:
D [(I f = 26, g = 20, h = 6), (F f = 27, g = 21, h = 6),
   (B f = 28, g = 8, h = 20), (E f = 28, g = 20, h = 8)]
A* Example II

Node: Queue:

I [(F f = 27, g = 21, h = 6), (B f = 28, g = 8, h = 20),
   (E f = 28, g = 20, h = 8), (G f = 30 g = 26, h = 4)]
12-49: A* Example II

Node: Queue:
F [(B f = 28, g = 8, h = 20), (E f = 28, g = 20, h = 8),
(G f = 30 g = 26, h = 4), (G f = 30 g = 26 h = 4)]
**12-50: A* Example II**

Node: Queue:

B: [(E f = 28, g = 20, h = 8), (E f = 29, g = 21, h = 8),
   (G f = 30, g = 26, h = 4), (G f = 30, g = 26, h = 4)]
Node: Queue :
E [(E f = 29, g = 21, h = 8), (G f = 30, g = 26, h = 4),
(G f = 30, g = 26, h = 4), (H f = 31, g = 31, h = 0)]
(next E can be discarded)
Node: Queue:
G        [(G f = 30, g = 26, h = 4), (H f = 30, g = 30, h = 0),
            (H f = 31, g = 31, h = 0)]
(next G can be discarded)
12-53: A* Example II

Goal

Start

Node: Queue :

H. Goal. [(H f = 31, g = 31, h = 0)]

Solution: A, C, D, I, G, H (or A, C, D, F, G, H)
Topologically, we can imagine A* creating a set of contours corresponding to $f$ values over the search space.

A* will search all nodes within a contour before expanding.

This allows us to prune the search space.

We can chop off the portion of the search tree corresponding to Zerind without searching it.
A* has one big weakness - Like BFS, it potentially keeps an exponential number of nodes in memory at once.

Iterative deepening A* is a workaround
- IDS was depth-limited search – IDA* is f-limited search
- Each iteration, increase bound to smallest value that allows search to continue
12-56: Iterative Deepening A* (IDA*)

```plaintext
f-limited-DFS(node, limit)
    if g(n) + h(n) > limit
        return fail, g(node) + h(node)
    if goalTest(node)
        return node, g(node)
    children = successor(node)
    smallestFail = MAX_VALUE
    for child in children
        sol, cost = depth-limited-DFS(child, limit)
        if sol != fail
            return sol, cost
        smallestFail = min(cost, smallestFail)
    return smallestFail, fail
```
Iterative Deepening A* (IDA*)

ida-star(node)
    limit = h(node)
    while true
        sol, limit = f-limited-DFS(node, limit)
        if (sol != fail)
            return sol
12-58: IDA* Example

Start: A

Edges:
- A to B: (3)
- A to C: (2)
- C to D: (2)
- D to E: (8)
- E to F: (1)
- E to G: (1)
- F to G: (1)
- G to H: (0)
- H to Goal: (0)

Edge Costs:
- A to B: 1
- B to E: 1
- C to D: 1
- D to E: 1
- E to G: 1
- G to I: 1
- I to H: 1

Values:
- (x) h value
- y edge cost

Goal: H
IDA*

- Works well in works with discrete-valued step costs
  - Preferably with steps having the same cost
- Each iteration brings in a large section of nodes
- What is the worst case performance for IDA*?
- When does the worst case occur?
12-60: **SMA***

- Run regular A*, with a fixed memory limit
- When limit is reached, discard node with highest f value of discarded node is assigned to the parent
  - Use the discarded node to get a better f value for parent
  - ’remember’ the value of that branch
  - If all other branches get higher f value, regenerate

- SMA* is complete and optimal
- Very hard problems can case SMA* to thrash, repeatedly regenerating branches
12-61: DFB&B

- Depth-First Branch and Bound
  - Run f-limited DFS, with limit set to infinity
  - When a goal is found, don’t stop – record it, and set limit to the goal depth
  - Keep going until all branches are searched or pruned.

- We will use something similar in 2-player games

- (DFB&B not in the text)
12-62: DFB&B
12-63: DFB&B
12-64: **DFB&B**

- What kinds of problems might Depth-First Branch and Bound work well for?
- Is DFB&B Complete? Optimal?
- How could we improve performance?
What kinds of problems might Depth-First Branch and Bound work well for?
- Optimization: Finding a solution is easy, finding the best is hard (TSP)

Is DFB&B Complete? Optimal?
- If we can find a solution easily, it is complete and optimal

How could we improve performance?
- Examine children in increasing g() value
Some nice features:

- Quickly find a solution
- Best solution so far gradually gets better
- Run DFB&B until it finishes (we have an optimal solution), or we run out of time (use the best so far)
While A* is optimally efficient, actual performance depends on developing accurate heuristics. Ideally, $h$ is as close to the actual cost to the goal ($h^*$) as possible while remaining admissible. Developing an effective heuristic requires some understanding of the problem domain.
12-68: Effective Heuristics - 8-puzzle

• $h_1$ - number of misplaced tiles.
  • This is clearly admissible, since each tile will have to be moved at least once.

• $h_2$ - *Manhattan distance* between each tile’s current position and goal position.
  • Also admissible - best case, we’ll move each tile directly to where it should go.

• Which heuristic is better?
• $h_2$ is better.
  • We want $h$ to be as close to $h^*$ as possible.
• If $h_2(n) > h_1(n)$ for all $n$, we say that $h_2$ dominates $h_1$.
• We would prefer a heuristic that dominates other known heuristics.
• So how do we find a good heuristic?
• Solve a relaxed version of the problem.
  • 8-puzzle:
    • Tile can be moved from A to B if:
      • A is adjacent to B
      • B is blank
    • Remove restriction that A is adjacent to B
      • Misplaced tiles
    • Remove restriction that B is blank
      • Manhattan distance
Finding a heuristic

- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
  - Romania path-finding
    - Add an extra road from each city directly to goal
    - (Decreases restrictions on where you can move)
  - Straight-line distance heuristic
Finding a heuristic

- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
  - Traveling Salesman
    - Connected graph
    - Each node has 2 neighbors
  - Minimum Cost Spanning Tree Heuristic
Finding a heuristic

- Solve subproblems
  - Cost of getting a subset of the tiles in place (ignoring the cost of moving other tiles)
- Save these subproblems in a database (could get large, depending upon the problem)
Finding a heuristic

- Using subproblems

Start State

Goal State
Finding a heuristic

- Number of heuristics $h_1, h_2, \ldots h_k$
- No one heuristic dominates any other
  - Different heuristics have different performances with different states
- What can you do?
Finding a heuristic

- Number of heuristics $h_1, h_2, \ldots, h_k$
- No one heuristic dominates any other
  - Different heuristics have different performances with different states
- What can you do?
  - $h(n) = \max(h_1(n), h_2(n), \ldots, h_k(n))$
Summary

- Problem-specific heuristics can improve search.
- Greedy search
- A*
- Memory limited search (IDA*, SMA*)
- Developing heuristics
  - Admissibility, monotonicity, dominance