School of EECS
Washington State University
Overview

- Problem-solving agent
- Formulating problems
- Search
- Uninformed search
- Informed (heuristic) search
- Heuristics
- Admissibility
Problem–Solving Agent

- Goal–based
- Atomic state representation
- Assume solution is a fixed sequence of actions
- Rationality: Achieve goal (minimize cost)
- Search for sequence of actions achieving goal
Environment Assumptions

- Observable
  - Agent always knows what state it is in

- Deterministic
  - Each action has one possible outcome (next state)

- Discrete
  - Each state has finite number of applicable actions

- Known
  - Agent knows which state each action will lead to

- Once solution sequence known, execute blindly (ignore percepts) until completion
Wumpus World Example

- Initial state →
- Goal state
  - Any state where agent has gold and not in cave
- Solution?
function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns an action

persistent: seq, an action sequence initially empty
  state, some description of the current world state
  goal, a goal, initially null
  problem, a problem formulation

state ← UPDATE-STATE (state, percept)
if seq is empty then
  goal = FORMULATE-GOAL (state)
  problem = FORMULATE-PROBLEM (state, goal)
  seq = SEARCH (problem)
  if seq = failure then return a null action
action = FIRST (seq)
seq = REST (seq)
return action
Well-Defined Problems

- State representation (atomic, but…)
- Action definitions (action: state \(\rightarrow\) state)
- Five parts
  - Initial state
  - Actions
  - Transition model
  - Goal test
  - Path cost
Well-Defined Problems (5 parts)

1. **Initial state**
2. **Actions**
   - ACTIONS(s) returns set of actions applicable to state s
3. **Transition model**
   - RESULT(s,a) returns state resulting from taking action a in state s
   - **Successor state** is any state reachable from the current state by a single action

- **State space** is set of all states reachable from the initial state by any sequence of actions
- State space forms a **directed graph** of nodes (states) and edges (actions)
- **Path** in state space is a sequence of states connected by actions
Vacuum World State Space
Well-Defined Problems (5 parts)

4. **Goal test**
   - True for any state satisfying goal

5. **Path cost**
   - Sum of the costs of the individual actions along the path
   - **Step cost** $c(s,a,s')$ is the cost of taking action $a$ in state $s$ to reach state $s'$
   - Non-negative

- **Solution** is sequence of actions leading from the initial state to a goal state
- **Optimal solution** is a solution with minimal path cost
Vacuum World Problem

- **State representation**
  - Location of vacuum: Left, Right
  - Cleanliness of each room: Clean, Dirty
  - Example state: (Left, Clean, Clean)
  - How many unique states?

- **Initial state**: Any state

- **Actions**: Left, Right, Suck

- **Transition model**
  - E.g., Result((Left, Dirty, Clean), Suck) = (Left, Clean, Clean)

- **Goal test**: State = (?, Clean, Clean)

- **Path cost**
  - Number of actions in solution (step cost = 1)
8-Puzzle

- **State**: Location of each tile (and blank)
  - E.g., (B, 1, 2, 3, 4, 5, 6, 7, 8)
  - How many states?
- **Initial state**: Any state
- **Actions**: Move blank Up, Down, Left or Right
- **Transition model**
- **Goal test**: State matches Goal State
- **Path cost**: Number of steps in path (step cost = 1)
Search

- Search tree
  - Root node is initial state
  - Node branches for each applicable move from node’s state
  - Frontier consists of the leaf nodes that can be expanded
  - Repeated states (*)
  - Goal state
Search Demo

- Code
- Nice 8-puzzle search web app
  - [http://tristanpenman.com/demos/n-puzzle](http://tristanpenman.com/demos/n-puzzle)
- Code
  - [http://github.com/tristanpenman/n-puzzle](http://github.com/tristanpenman/n-puzzle)

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

Initial State

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

Goal State
Real–World Search Problems

- Route finding
- Touring
- Circuit layouts
- Robot navigation
- Assembly sequencing
- Chemical design

- Most of AI can be cast as a search problem
Route Finding Example

- Romania road map
- Initial state: Arad
- Goal state: Bucharest
Route Finding Example
Search Tree

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

Duplicate

Bucharest
Tree Search

function Tree-Search (problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the node, adding the resulting nodes to the frontier

- Search strategy determines how nodes are chosen for expansion
- Suffers from repeated state generation
function GRAPH-SEARCH (problem) returns a solution, or failure
initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  add the node to the explored set
  expand the node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set

- Keep track of explored set to avoid repeated states
- Changes from TREE-SEARCH highlighted
Graph Search Example

Oradea no longer expanded
function CHILD-NODE (problem, parent, action) returns a node
return a node with
  STATE = problem.RESULT (parent.STATE, action),
  PARENT = parent,
  ACTION = action,
  PATH-COST = parent.PATH-COST +
               problem.STEP-COST (parent.STATE, action)
Implementation

- Frontier is a queue or stack
  - How nodes are added/removed defines search strategy
- Explored set is a hash table
  - Can be large (# unique states)
  - Key is some canonical state representation
Measuring Performance

- **Completeness**
  - Is the search algorithm guaranteed to find a solution if one exists?

- **Optimality**
  - Does the search algorithm find the optimal solution?

- **Time and space complexity**
  - Branching factor $b$ (maximum successors of a node)
  - Depth $d$ of shallowest goal node
  - Maximum path length $m$
  - Complexity $O(b^d)$ to $O(b^m)$
Uninformed Search Strategies

- No preference over states based on “closeness” to goal
- Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
Breadth–First Search

- Expand shallowest nodes in frontier
- Frontier is a simple queue
  - Dequeue nodes from front, enqueue nodes to back
  - First–In, First–Out (FIFO)
function **BREADTH-FIRST-SEARCH** *(problem)* returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

frontier ← FIFO queue with node as only element
explored ← empty set

loop do
iif EMPTY(frontier) then return failure

node ← DEQUEUE(frontier) // choose shallowest node in frontier
add node.STATE to explored

for each action in problem.ACTIONS(node.STATE) do
child = CHILD-NODE(problem, node, action)

if child.STATE is not in explored or frontier then
  if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)

frontier ← ENQUEUE(child, frontier)
Breadth-First Search

8-puzzle demo

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

Initial State

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

Goal State
Breadth-First Search

- Complete?
- Optimal?
- Time complexity
  - Number of nodes generated (worst case)
    \[ \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d) \]
- Space complexity
  - \( O(b^{d-1}) \) nodes in explored set
  - \( O(b^d) \) nodes in frontier
  - Total \( O(b^d) \)
Exponential complexity $O(b^d)$

For $b=4$, 1KB/node, 1M nodes/sec

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16</td>
<td>0.02 ms</td>
<td>16 KB ($10^3$)</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>0.26 ms</td>
<td>256 KB ($10^3$)</td>
</tr>
<tr>
<td>8</td>
<td>65,536</td>
<td>0.07 sec</td>
<td>65 MB ($10^6$)</td>
</tr>
<tr>
<td>16</td>
<td>4.3B</td>
<td>71.6 min</td>
<td>4.3 TB ($10^{12}$)</td>
</tr>
<tr>
<td>20</td>
<td>$10^{12}$</td>
<td>12.7 days</td>
<td>1 PetaByte ($10^{15}$)</td>
</tr>
<tr>
<td>30</td>
<td>$10^{18}$</td>
<td>366 centuries</td>
<td>1 ZettaByte ($10^{21}$)</td>
</tr>
</tbody>
</table>
Uniform–Cost Search

- Expand node $n$ with lowest path cost $g(n)$
- Frontier is a priority queue
  - Queue partially ordered by path cost
  - Lowest path cost node always at the front
function **UNIFORM-COST-SEARCH** *(problem)* returns a solution, or failure

node ← a node with `STATE = problem.INITIAL-STATE`, `PATH-COST = 0`

frontier ← priority queue ordered by `PATH-COST`, with `node` as only element

explored ← empty set

loop do
  if `EMPTY(frontier)` then return failure
  node ← `DEQUEUE(frontier)`  // choose lowest cost node in `frontier`
  if `problem.GOAL-TEST(node.STATE)` then return `SOLUTION(node)`
  add `node.STATE` to `explored`
  for each `action` in `problem.ACTIONS(node.STATE)` do
    child = `CHILD-NODE(problem, node, action)`
    if `child.STATE` is not in `explored` or `frontier` then
      `frontier` ← `ENQUEUE(child, frontier)`
    else if `child.STATE` is in `frontier` with higher `PATH-COST` then
      replace that `frontier` node with `child`

Why not check Goal-Test here?

Why is this test necessary?
Uniform–Cost Search

- Example (Sibiu → Bucharest)
Uniform–Cost Search

- Complete?
- Optimal?
- Time and space complexity
  - \( b = \) branching factor
  - \( \varepsilon = \) minimum step cost (>0)
  - \( C^* = \) cost of optimal solution

\[
O(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor})
\]
Depth-First Search

- Always expand the deepest node
- Frontier is a simple stack
  - Push nodes to front, pop nodes from front
  - Last-In, First-Out (LIFO)
- Otherwise, same code as BFS
- Or, implement recursively
Depth-First Search

DEMO
Depth–First Search

- Tree–Search version
  - Not complete (infinite loops)
  - Not optimal

- Graph–Search version
  - Complete
  - Not optimal

- Time complexity (m = max depth): $O(b^m)$

- Space complexity
  - Tree–search: $O(bm)$
  - Graph–search: $O(b^m)$
function **DEPTH-LIMITED-SEARCH** *(problem, limit)* returns a solution, or failure/cutoff
return **RECURSIVE-DLS** *(MAKE-NODE *(problem.INITIAL-STATE)*, problem, limit)*

function **RECURSIVE-DLS** *(node, problem, limit)* returns a solution, or failure/cutoff
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
else if limit = 0 then return cutoff
else
cutoff_occurred ← false
for each action in problem.ACTIONS(node.STATE) do
  child = CHILD-NODE(problem, node, action)
  result ← RECURSIVE-DLS (child, problem, limit – 1)
  if result = cutoff then cutoff_occurred ← true
  else if result ≠ failure then return result
if cutoff_occurred then return cutoff else return failure
Depth-Limited Search

- Limit DFS depth to $l$
- Still incomplete, if $l < d$
- Non-optimal if $l > d$
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$
Iterative–Deepening Search

- Run `DEPTH-LIMITED-SEARCH` iteratively with increasing depth limit

```plaintext
function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure
for depth = 0 to ∞ do
    result = DEPTH-LIMITED-SEARCH (problem, depth)
    if result ≠ cutoff then return result
```
Iterative–Deepening Search

Limit = 0

Limit = 1

Limit = 2

Limit = 3

DEMO
Iterative–Deepening Search

- Complete?
- Optimal?
- Space complexity: $O(bd)$
- Time complexity
  \[
  \sum_{i=0}^{d-1} (d - i)b^{i+1} = (d)b + (d - 1)b^2 + \ldots + (1)b^d = O(b^d)
  \]
  - Nodes at depth $d = \text{all nodes at depths 1 to (d−1)}$
- Iterative deepening best uninformed search when solution depth unknown
Bidirectional Search

- Search forward from initial state and backward from goal state
- Meet (hopefully) in the middle
- Each search has complexity $O(b^{d/2}) \ll O(b^d)$
- Replace goal test with frontier intersection
- How to reverse actions?
## Uninformed Search Strategies

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes(^1)</td>
<td>Yes(^1,2)</td>
<td>No</td>
<td>No</td>
<td>Yes(^1)</td>
<td>Yes(^1,4)</td>
</tr>
<tr>
<td>Time</td>
<td>O(b(^d))</td>
<td>O(b(^1+\lceil C*/\varepsilon \rceil))</td>
<td>O(b(^m))</td>
<td>O(b(^l))</td>
<td>O(b(^d))</td>
<td>O(b(^d/2))</td>
</tr>
<tr>
<td>Space</td>
<td>O(b(^d))</td>
<td>O(b(^1+\lceil C*/\varepsilon \rceil))</td>
<td>O(bm)</td>
<td>O(b(^l))</td>
<td>O(bd)</td>
<td>O(b(^d/2))</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes(^3)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes(^3)</td>
<td>Yes(^3,4)</td>
</tr>
</tbody>
</table>

1. Complete if b is finite
2. Complete if step costs \(\geq \varepsilon > 0\)
3. Optimal if step costs all the same
4. If both directions use BFS
Informed (Heuristic) Search

- Guided by problem-specific knowledge other than the problem formulation
- Problem-specific knowledge usually expressed as heuristics
Heuristic function $h(n)$ estimates the cost of the path from state $n$ to a goal state
- E.g., 8-puzzle
  - Number of tiles out
  - Euclidean distance of each tile
  - City-block (Manhattan) distance of each tile
- Non-negative function
- For goal node $h(n)=0$

Recall path cost $g(n)$ is the cost so far from the initial state to state $n$

Evaluation function $f(n) = g(n) + h(n)$ estimates the total cost of a solution going through state $n$
Best–First Search

- Choose next frontier node with smallest $f(n)$
- Depth–first search = Best–first search with
  - $f(n) = g(n) + h(n) = ?$
- Breadth–first search = Best–first search with
  - $f(n) = g(n) + h(n) = ?$
- Uniform–cost search = Best–first search with
  - $f(n) = g(n) + h(n) = ?$
Heuristic Search Strategies

- Greedy best-first search
- A* search
Greedy Best–First Search

- Best–first search with \( f(n) = h(n) \)
- Example: Route–finding problem
  - \( h(n) = \) straight–line distance from city \( n \) to goal city

Straight–line distances to Bucharest:

<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>Drobeta</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 Best-First Search Example: Arad to Bucharest

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Fagaras
Greedy Best-First Search Example: Arad to Bucharest
Greedy Best-First Tree Search Example: Iasi to Fagaras

SLD to Fagaras
Neamt 200
Iasi 220
Vaslui 230
Greedy Best–First Search

- Complete?
- Optimal?
- Time and space complexity: $O(b^m)$
  - $b =$ branching factor
  - $m =$ maximum depth of search space
  - Worst case
    - Good heuristic can substantially improve
A* Search

- $f(n) = g(n) + h(n)$
  - Estimated cost of solution through $n$
- Same as Uniform-Cost search using $f(n)$
- Complete and optimal under some constraints on $h(n)$
- Example: Route-finding using SLD

History: A* generalizes over algorithms A1 and A2, which were heuristic extensions to Dijkstra’s shortest path algorithm.
A* Search Example: Arad to Bucharest

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Rimnicu Vilea

Artificial Intelligence
A* Search Example: Arad to Bucharest (cont.)

(e) After expanding Fagaras

(f) After expanding Pitesti
For A* tree search to be optimal, h(n) must be admissible
  ◦ A heuristic function h(n) is admissible if it never over-estimates the cost of reaching the goal from n
  ◦ E.g., Straight-line distance for route finding
  ◦ E.g., Tiles out of place in 8-puzzle

For A* graph search to be optimal, heuristic must further satisfy triangle inequality (also called consistent or monotonic)
  ◦ A heuristic function h(n) satisfies the triangle inequality if h(n) ≤ cost(n,a,n’) + h(n’)

Optimality of A*
A* Search

- Complete and optimal?
  - Yes, if heuristic is admissible

- Time and space complexity?
  - Still \( O(b^d) \) worst case
  - Space is typically the bottleneck

- A* is optimally efficient
  - No other algorithm using the same consistent heuristic is guaranteed to expand fewer nodes
Heuristics Revisited

- Why not use \( h(n) = 1 \)?
- How to measure quality of heuristic?
- **Effective branching factor** \( b^* \)
  - Assume \( A^* \) generates \( N \) nodes to find solution at depth \( d \)
  - What branching factor needed for a uniform tree of depth \( d \) to include \( N+1 \) nodes?
  - \( N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d \)
  - Ideally, \( b^* = 1 \)
- E.g., \( N=52 \), \( d=5 \), \( b^*=1.92 \)
Heuristics Revisited

- E.g., 8-puzzle
  - $h_1 = \text{tiles out of place}$
  - $h_2 = \text{sum of tiles’ city block distances}$

\[
\begin{align*}
&h_1 = 8 \\
&h_2 = 3 + 1 + 2 + 2 + 3 + 2 + 2 + 3 = 18 \\
&\text{Solution cost} = 26
\end{align*}
\]
Values averaged over 100 8-puzzle problems for each d

Note: \( b^*(h_2) \leq b^*(h_1) \)

<table>
<thead>
<tr>
<th>d</th>
<th>IDS</th>
<th>A*(h₁)</th>
<th>A*(h₂)</th>
<th>IDS</th>
<th>A*(h₁)</th>
<th>A*(h₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>2.45</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
<td>2.87</td>
<td>1.48</td>
<td>1.45</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
<td>2.73</td>
<td>1.34</td>
<td>1.30</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
<td>2.80</td>
<td>1.33</td>
<td>1.24</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
<td>2.79</td>
<td>1.38</td>
<td>1.22</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
<td>2.78</td>
<td>1.42</td>
<td>1.24</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>539</td>
<td>113</td>
<td>-</td>
<td>1.44</td>
<td>1.23</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>1301</td>
<td>211</td>
<td>-</td>
<td>1.45</td>
<td>1.25</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>3056</td>
<td>363</td>
<td>-</td>
<td>1.46</td>
<td>1.26</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>7276</td>
<td>676</td>
<td>-</td>
<td>1.47</td>
<td>1.27</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>18094</td>
<td>1219</td>
<td>-</td>
<td>1.48</td>
<td>1.28</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>39135</td>
<td>1641</td>
<td>-</td>
<td>1.48</td>
<td>1.26</td>
</tr>
</tbody>
</table>
Heuristics Revisited

- Heuristic $h_2$ dominates $h_1$ if, for all nodes $n$, $h_2(n) \geq h_1(n)$
- Implies A* using $h_2$ will typically generate fewer nodes than A* using $h_1$
- “City block distance” dominates “misplaced tiles”
- In general, want $h(n)$ to be consistent and close to true solution cost from node $n$
  - But still be fast to compute
Relaxed problems
  ◦ $h(n) = \text{cost of solution to relaxed problem}$
  ◦ E.g., 8-puzzle where you can swap tiles

Subproblems
  ◦ $h(n) = \text{cost of solution to subproblem}$
  ◦ E.g., get half the tiles in correct position

Learning from experience
  ◦ Collect experience as (state, solution cost) pairs
  ◦ Learn $h(n): \text{state} \rightarrow \text{solution cost}$
Problem–solving agent
Formulating problems
Search
Uninformed search (Iterative–Deepening)
Informed (heuristic) search (A*)
Admissible heuristics