Last time: Problem-Solving

- **Problem solving:**
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution

- **Problem formulation:**
  - Initial state
  - ?
  - ?
  - ?
  - ?

- **Problem types:**
  - single state: accessible and deterministic environment
  - multiple state: ?
  - contingency: ?
  - exploration: ?
Last time: Problem-Solving

- **Problem solving:**
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution

- **Problem formulation:**
  - Initial state
  - Operators
  - Goal test
  - Path cost

- **Problem types:**
  - single state: accessible and deterministic environment
  - multiple state: ?
  - contingency: ?
  - exploration: ?
Last time: Problem-Solving

- **Problem solving:**
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution

- **Problem formulation:**
  - Initial state
  - Operators
  - Goal test
  - Path cost

- **Problem types:**
  - single state: accessible and deterministic environment
  - multiple state: inaccessible and deterministic environment
  - contingency: inaccessible and nondeterministic environment
  - exploration: unknown state-space
Last time: Finding a solution

**Solution:** is ???

**Basic idea:** offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

```plaintext
Function General-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state 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 resulting nodes to the search tree
    end
```

Last time: Finding a solution

Solution: is a sequence of operators that bring you from current state to the goal state.

Basic idea: offline, systematic exploration of simulated state-space by generating successors of explored states (expanding).

Function General-Search(problem, strategy) returns a solution, or failure

initialize the search tree using the initial state 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 resulting nodes to the search tree
end

Strategy: The search strategy is determined by ???
Last time: Finding a solution

Solution: is a sequence of operators that bring you from current state to the goal state

Basic idea: offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

Function General-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state 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 resulting nodes to the search tree
    end

Strategy: The search strategy is determined by the order in which the nodes are expanded.
**A Clean Robust Algorithm**

**Function** UniformCost-Search(problem, Queuing-Fn) **returns** a solution, or failure

- open ← make-queue(make-node(initial-state[problem]))
- closed ← [empty]

**loop do**

- **if** open is empty **then return** failure
- currnode ← Remove-Front(open)
- **if** Goal-Test[problem] applied to State(currnode) **then return** currnode
- children ← Expand(currnode, Operators[problem])
- **while** children not empty

  **[… see next slide …]**

**end**
- closed ← Insert(closed, currnode)
- open ← Sort-By-PathCost(open)

**end**
A Clean Robust Algorithm

[... see previous slide ...]

children ← Expand(currnode, Operators[problem])

while children not empty

    child ← Remove-Front(children)

    if no node in open or closed has child’s state

        open ← Queuing-Fn(open, child)

    else if there exists node in open that has child’s state

        if PathCost(child) < PathCost(node)

            open ← Delete-Node(open, node)
            open ← Queuing-Fn(open, child)

    else if there exists node in closed that has child’s state

        if PathCost(child) < PathCost(node)

            closed ← Delete-Node(closed, node)
            open ← Queuing-Fn(open, child)

end

[... see previous slide ...]
Last time: search strategies

**Uninformed:** Use only information available in the problem formulation

- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

**Informed:** Use heuristics to guide the search

- Best first
- A*
Evaluation of search strategies

• Search algorithms are commonly evaluated according to the following four criteria:
  • **Completeness:** does it always find a solution if one exists?
  • **Time complexity:** how long does it take as a function of number of nodes?
  • **Space complexity:** how much memory does it require?
  • **Optimality:** does it guarantee the least-cost solution?

• Time and space complexity are measured in terms of:
  • $b$ – max branching factor of the search tree
  • $d$ – depth of the least-cost solution
  • $m$ – max depth of the search tree (may be infinity)
Last time: uninformed search strategies

Uninformed search:
Use only information available in the problem formulation
- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening
This time: informed search

**Informed search:**

Use heuristics to guide the search

- Best first
- A*
- Heuristics
- Hill-climbing
- Simulated annealing
Best-first search

• Idea:
  use an evaluation function for each node; estimate of "desirability"
⇒ expand most desirable unexpanded node.

• Implementation:

  QueueingFn = insert successors in decreasing order of desirability

• Special cases:
  greedy search
  A* search
Romania with step costs in km

<table>
<thead>
<tr>
<th>City</th>
<th>Step Cost</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>Dobroasia</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>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Medea</td>
<td>241</td>
</tr>
<tr>
<td>Neamț</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>98</td>
</tr>
<tr>
<td>Rimnicu Vâlcea</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

• Estimation function:
  \[ h(n) = \text{estimate of cost from } n \text{ to goal (heuristic)} \]

• For example:
  \[ h_{SLD}(n) = \text{straight-line distance from } n \text{ to Bucharest} \]

• Greedy search expands first the node that appears to be closest to the goal, according to \( h(n) \).
Greedy search example

Arad

366
Properties of Greedy Search

- Complete?
- Time?
- Space?
- Optimal?
Properties of Greedy Search

- **Complete?** No – can get stuck in loops
  e.g., Iasi > Neamt > Iasi > Neamt > …
  Complete in finite space with repeated-state checking.

- **Time?** \(O(b^m)\) but a good heuristic can give
dramatic improvement

- **Space?** \(O(b^m)\) – keeps all nodes in memory

- **Optimal?** No.
A* search

• Idea: avoid expanding paths that are already expensive

evaluation function: \( f(n) = g(n) + h(n) \) with:
  \( g(n) \) – cost so far to reach \( n \)
  \( h(n) \) – estimated cost to goal from \( n \)
  \( f(n) \) – estimated total cost of path through \( n \) to goal

• A* search uses an admissible heuristic, that is,
  \( h(n) \leq h^*(n) \) where \( h^*(n) \) is the true cost from \( n \).
  For example: \( h_{SLD}(n) \) never overestimates actual road distance.

• Theorem: A* search is optimal
A* search example
AIMA Slides ©Stuart Russell and Peter Norvig, 1998

Chapter 4, Sections 1, 2, 4
Optimality of A* (standard proof)

Suppose some suboptimal goal $G_2$ has been generated and is in the queue. Let $n$ be an unexpanded node on a shortest path to an optimal goal $G_1$.

\[
f(G_2) = g(G_2) \quad \text{since } h(G_2) = 0
\]
\[
> g(G_1) \quad \text{since } G_2 \text{ is suboptimal}
\]
\[
\geq f(n) \quad \text{since } h \text{ is admissible}
\]

Since $f(G_2) > f(n)$, A* will never select $G_2$ for expansion.
Optimality of A* (more useful proof)

Lemma: A* expands nodes in order of increasing $f$ value

Gradually adds “$f$-contours” of nodes (cf. breadth-first adds layers)
Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
How do the contours look like when \( h(n) = 0 \)?
Properties of A*

- Complete?
- Time?
- Space?
- Optimal?
Properties of A*

• Complete? Yes, unless infinitely many nodes with $f \leq f(G)$

• Time? Exponential in $[(\text{relative error in } h) \times (\text{length of solution})]$]

• Space? Keeps all nodes in memory

• Optimal? Yes – cannot expand $f_{i+1}$ until $f_i$ is finished
Proof of lemma: pathmax

For some admissible heuristics, \( f \) may decrease along a path.

E.g., suppose \( n' \) is a successor of \( n \)

\[
\begin{align*}
n & \quad g=5 \quad h=4 \quad f=9 \\
1 & \\
n' & \quad g'=6 \quad h'=2 \quad f'=8
\end{align*}
\]

But this throws away information!
\( f(n) = 9 \Rightarrow \) true cost of a path through \( n \) is \( \geq 9 \)
Hence true cost of a path through \( n' \) is \( \geq 9 \) also

Pathmax modification to A*:
Instead of \( f(n') = g(n') + h(n') \), use \( f(n') = \max(g(n') + h(n'), f(n)) \)

With pathmax, \( f \) is always nondecreasing along any path.
Admissible heuristics

E.g., for the 8-puzzle:

\[ h_1(n) = \text{number of misplaced tiles} \]
\[ h_2(n) = \text{total Manhattan distance} \]

(i.e., no. of squares from desired location of each tile)

Start State

Goal State

\[
\begin{array}{ccc}
5 & 4 & \\
6 & 1 & 8 \\
7 & 3 & 2 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
8 & & 4 \\
7 & 6 & 5 \\
\end{array}
\]

\[
h_1(S) = ?? \]
\[
h_2(S) = ?? \]
Admissible heuristics

E.g., for the 8-puzzle:

\[ h_1(n) = \text{number of misplaced tiles} \]
\[ h_2(n) = \text{total Manhattan distance} \]

(i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
5 & 4 & \text{\textcolor{gray}{gray}} \\
6 & 1 & 8 \\
7 & 3 & 2 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 2 & 3 \\
8 & \text{\textcolor{gray}{gray}} & 4 \\
7 & 6 & 5 \\
\end{array}
\]

Start State

Goal State

\[ h_1(S) = ?? \quad 7 \]
\[ \frac{h_2(S)}{h_2(S)} = ?? \quad 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18 \]
Relaxed Problem

• Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.

• If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.

• If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.
Next time

• Iterative improvement
• Hill climbing
• Simulated annealing