Questions?
Outline

- Chapter 3 - Solving Problems by Searching
  - Uninformed Search Strategies
    - Breadth-first Search
    - Uniform-cost Search
    - Depth-first Search
    - Depth-limited Search
    - Iterative-deepening Search
Recall: Searching for Solutions

- Recall a solution to a problem is an action sequence. An agent searches among the possible action sequences for one that solves the problem.

- Structure this process as a search tree with the initial state as the root, the edges as the actions, and the nodes corresponding to the states in the state space of the problem.
Recall: Searching for Solutions

- As each node is "visited", test to see if it is a goal state. If so, have found a solution.
- Otherwise, expand the node by applying each legal action to current state. This generates a new set of states.
- The essential differences in search algorithms is choosing which unexplored state should be visited next. The set of these leaf nodes is called the frontier.
Romania Map Search Tree

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu
Graph-Search Function

Receives: \textit{problem}; Returns: \textit{solution}, or failure

1. Initialize frontier using initial state of \textit{problem}

2. \textbf{Initialize the explored set to empty}

3. Loop
   3.1. If the frontier is empty then return failure
   3.2. Choose a leaf node and remove it from frontier
   3.3. If the node contains a goal state then return corresponding solution
   3.4. \textbf{Add node to the explored set}
   3.5. Expand the chosen node, adding resulting nodes to the frontier, \textbf{only if not in the frontier or explored set}
Tree node is a structure containing
- State: state in state space the node corresponds to
- Parent: node in search tree that generated this node
- Action: action applied to the parent to generate this node
- Path-Cost: cost of the path from the initial state to this node, as indicated by the parent pointers. It is traditionally denoted $g(n)$
Infrastructure for Search Algorithms

Child-Node (constructor) function

Receives: problem, parent (node), action
Returns: a node

2. Parent = parent
3. Action = action
A solution is formed by following the parent pointers from the node containing a goal state back to the initial state.
The frontier is implemented using a "queue" with operations

- Empty? \((q)\) – returns true if there are no elements
- Pop \((q)\) – removes the first element and returns it
- Insert \((element, q)\) – insert an element and returns the resulting queue

Different implementations used in each search strategy
Measuring Performance

- Search algorithms can be compared in 4 ways
  - Completeness: Is the algorithm guaranteed to find a solution when there is one?
  - Optimality: Does the strategy find the optimal one?
  - Time complexity: How long does it take to find a solution?
  - Space complexity: How much memory is needed to perform the search?
Measuring Performance

- For complexity measures, problem size is expressed in terms of:
  - \( b \) - branching factor
  - \( d \) - depth of shallowest goal node
  - \( m \) - the maximum length of any path in state space

- For time, often use number of nodes generated during the search

- For space, often use maximum number of nodes stored in memory
Measuring Performance

- **Search cost** is one factor in determining effectiveness. Could also consider **total cost** – search cost combined with path cost of the solution.

- Use total cost to find optimal tradeoff point between computation and finding a shorter path. Unfortunately, costs usually are in different units. E.g., Romanian map search cost in milliseconds and path cost in kilometers.
Uninformed Search

- Uninformed (or blind) search strategies have no additional information beyond that provided in the problem definition.
  - Can generate successor states
  - Can distinguish a goal state from a non-goal state
- Strategies are distinguished by order in which nodes are expanded. Will look at several.
Breadth-First Search (BFS)

- **Breadth-first search** expands the root node, then all of the successors of the root node, then all of their successors, etc. In general, all the nodes at a given depth are expanded before any nodes at the next level are expanded.

- This is an instance of the general Graph-Search algorithm where the shallowest node is expanded first.

- How would the frontier queue be implemented?
Breadth-First-Search Function

Receives: \textit{problem}; Returns: \textit{solution}, or failure

1. Create a \textit{node} with node.State = problem.Initial-State and node.Path-Cost = 0
2. If problem.Goal-Test(node) then return Solution(node)
3. Create \textit{frontier} as FIFO queue and insert node
4. Initialize the \textit{explored} to empty
5. Loop
   5.1. If Empty?(frontier) then return failure
   5.2. node = Pop (frontier)
   5.3. Add node.State to \textit{explored}
   5.4. For each \textit{action} in Actions(node.State) do
       5.4.1. child = ChildNode(problem, node, action)
       5.4.2. If child.State is not in \textit{explored} or \textit{frontier} then
           5.4.2.1. If problem.Goal-Test(child.State) then
                       return Solution(child)
           5.4.2.2. \textit{frontier} = Insert(child, \textit{frontier})
Breadth-First Search (BFS)

- One slight tweak to the general graph search algorithm is to apply the goal test as nodes are generated, rather than when it is selected for expansion.
- Evaluation:
  - Complete?
  - Optimal?
  - Space complexity?
  - Time complexity?
BFS Time & Space

\[ b = 10; \ 1M \text{ nodes/second}; \ 1000 \text{ bytes/node} \]

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 ms</td>
<td>1 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>11 ms</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>(10^6)</td>
<td>1.1 s</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>(10^8)</td>
<td>2 min</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>(10^{10})</td>
<td>3 h</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>(10^{12})</td>
<td>13 d</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>(10^{14})</td>
<td>3.5 y</td>
<td>99 petabytes</td>
</tr>
<tr>
<td>16</td>
<td>(10^{16})</td>
<td>350 y</td>
<td>10 exabyte</td>
</tr>
</tbody>
</table>
Uniform-Cost Search (UCS)

- When all step costs are equal, BFS is optimal since it expands the shallowest node. Extend this idea to **uniform-cost search** (UCS) which expands the node with the **lowest path cost** $g(n)$.
- Also want to replace a frontier node when find lower cost path to the node's state.
- How would the frontier be implemented for this search?
Uniform-Cost-Search Function

Receives: problem; Returns: solution, or failure

1. Create a node with node.State = problem.Initial-State and node.Path-Cost = 0
2. Create frontier as priority queue ordered by Path-Cost and insert node
3. Initialize the explored to empty
4. Loop
   4.1. If Empty?(frontier) then return failure
   4.2. node = Pop (frontier)
   4.3. If problem.Goal-Test(node.State) then return Solution(node)
   4.4. Add node.State to explored
   4.5. For each action in Actions(node.State) do
       4.5.1. child = ChildNode(problem, node, action)
       4.5.2. If child.State is not in explored or frontier then frontier = Insert(child, frontier)
       4.5.3. Else if child.State is in frontier with higher Path-Cost then replace that frontier node with child
Uniform-Cost Search (UCS)

- Test for goal state when selected for expansion because it may not be lowest cost when generated.
- Later generated paths may be lower cost and need to be retained.
Uniform-Cost Search (UCS)

- Evaluation:
  - Complete?
  - Optimal?
  - Space complexity?
  - Time complexity?
Depth-First Search (DFS)

- **Depth-first search** expands the *deepest* node in the frontier. It searches down a path to the leaf nodes, then "backs up" to the next deepest unexplored node.

- How would the frontier be implemented for this search?

- Can be implemented with Tree-Search or Graph-Search. Alternatively, it can be implemented as a recursive function.
Depth-First Search (DFS)
Depth-First Search (DFS)

- Evaluation:
  - Complete?
  - Optimal?
  - Space complexity?
  - Time complexity?
Depth-Limited Search (DLS)

- **Depth-limited search** (DLS) alleviates the problem of infinite paths in DFS by limiting the depth of the search.
- All nodes at the depth limit $l$ are treated as if they have no successors. When this happens a **cutoff** failure occurs for that branch.
- Time complexity becomes $O(b^l)$ and space complexity becomes $O(b^l)$. Still not complete or optimal.
Iterative-Deepening Depth-First Search (IDS)

- **Iterative-deepening search** combines the characteristics of DFS and BFS. Call DLS with depths from 0 to $\infty$.
- Explores frontier like BFS, but with time and space complexity like DFS.
Iterative-Deepening Depth-First Search (IDS)
### Comparison of Uninformed Search Strategies

<table>
<thead>
<tr>
<th>Criterion</th>
<th>BFS</th>
<th>UCS</th>
<th>DFS</th>
<th>DLS</th>
<th>IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Sometimes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Sometimes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^d)$</td>
<td>$O(b^{1+C*/\varepsilon})$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^d)$</td>
<td>$O(b^{1+C*/\varepsilon})$</td>
<td>$O(bm)$</td>
<td>$(O(b^l))$</td>
<td>$O(bd)$</td>
</tr>
</tbody>
</table>

- When using Tree-Search.
- Graph-Search makes DFS complete for finite spaces and space and time complexity bounded by size of state space.