Remember: Implementation of search algorithms

Function General-Search(problem, Queuing-Fn) returns a solution, or failure
nodes ← make-queue(make-node(initial-state[problem]))

loop do
  if nodes is empty then return failure
  node ← Remove-Front(nodes)
  if Goal-Test[problem] applied to State(node) succeeds then return node
  nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
end
Remember: Implementation of search algorithms

**Queuing-Fn** *(queue, elements)* is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

Encapsulating *state* information in *nodes*

A *state* is a (representation of) a physical configuration.

A *node* is a data structure constituting part of a search tree includes parent, children, depth, path cost *g(x)*.

*States do not have parents, children, depth, or path cost!*

The **Expand** function creates new nodes, filling in the various fields and using the **Operators** (or **SuccessorFn** of the problem to create the corresponding states.
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 function of num. 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)
\begin{itemize}
  \item In our complexity analysis, we do not take the built-in \textit{loop-detection} into account.
  \item The results only ‘formally’ apply to the variants of our algorithms \textbf{WITHOUT} loop-checks.
  \item Studying the effect of the loop-checking on the complexity is hard:
    \begin{itemize}
      \item overhead of the checking \textbf{MAY} or \textbf{MAY NOT} be compensated by the reduction of the size of the tree.
    \end{itemize}
  \end{itemize}

Also: our analysis \textbf{DOES NOT} take the length (space) of representing paths into account !!
Uninformed search strategies

Use only information available in the problem formulation

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

Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

\[ \text{QUEUEINGFN} = \text{put successors at end of queue} \]

Move downwards, level by level, until goal is reached.
Example: Traveling from Arad To Bucharest

Breadth-first search
Breadth-first search

Breadth-first search
Properties of breadth-first search

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:**
  - **Time complexity:**
  - **Space complexity:**
  - **Optimality:**

Uniform-cost search

Expand least-cost unexpanded node

**Implementation:**

\[
\text{QUEUEINGFn} = \text{insert in order of increasing path cost}
\]

So, the queueing function keeps the node list sorted by increasing path cost, and we expand the first unexpanded node (hence with smallest path cost)

A refinement of the breadth-first strategy:

Breadth-first = uniform-cost with path cost = node depth
Romania with step costs in km

Uniform-cost search

<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>Dobrota</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>Hursova</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>580</td>
</tr>
<tr>
<td>Pitești</td>
<td>98</td>
</tr>
<tr>
<td>Râmnicu Vâlcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Uziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Properties of uniform-cost search

- Completeness: ?
- Time complexity: ?
- Space complexity: ?
- Optimality: ?

Implementation of uniform-cost search

- Initialize Queue with root node (built from start state)
- Repeat until Queue is empty/node 1 has Goal state:
  - Remove first node from front of Queue
  - Expand node (find its children)
  - Reject those children that have already been considered, to avoid loops
  - Add remaining children to Queue, in a way that keeps entire queue sorted by increasing path cost
- If Goal was reached, return success, otherwise failure
Caution!

- Uniform-cost search not optimal if it is terminated when *any* node in the queue has goal state.

  - Uniform cost returns the path with cost 102 (if any goal node is considered a solution), while there is a path with cost 25.

Question:

- Do we have the previous problem in this example?
In class, we saw that the search may fail or be sub-optimal if:

- no loop detection: then algorithm runs into infinite cycles
  \[(A \rightarrow B \rightarrow A \rightarrow B \rightarrow \ldots)\]

- not queueing-up a node that has a state which we have already visited:
  may yield suboptimal solution

- simply avoiding to go back to our parent: looks promising, but we have not proven that it works

Solution?

Is that enough??
Example

Example Illustrating Uninformed Search Strategies

Breadth-First Search Solution

Example Illustrating Uninformed Search Strategies
Uniform-Cost Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

Note: Queuing in Uniform-Cost Search

**Problem:** wasting (but not incorrect) to queue-up three nodes with G:

- Although different paths, but for sure that the one with smallest path cost (9 in the example) is the first one in the queue.

**Solution:** refine the queuing function by:

**Is that it??**
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 ...]
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Black = open queue
Grey = closed queue
#: The step in which the node expanded

Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Black = open queue
Grey = closed queue
#: The step in which the node expanded

Insert expanded nodes
Such as to keep open queue sorted
Node 2 has 2 successors: one with state B and one with state S.

We have node #1 in closed with state S; but its path cost 0 is smaller than the path cost obtained by expanding from A to S. So we do not queue-up the successor of node 2 that has state S.

Node 4 has a successor with state C and Cost smaller than node #3 in open that also had state C; so we update open to reflect the shortest path.
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>6</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>G</td>
<td>7</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>

Goal reached

---

### More examples...

- See the great demos at:

Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{QUEUEING FN} = \text{insert successors at front of queue} \]
Depth-first search

![Diagram showing depth-first search]

Depth-first search

![Diagram showing depth-first search with additional nodes]

23
Depth-first search

Properties of depth-first search

- Completeness: ?
- Time complexity: ?
- Space complexity: ?
- Optimality: ?

Remember:

\[ b = \text{branching factor} \]
\[ m = \text{max depth of search tree} \]
Avoiding repeated states

In increasing order of effectiveness and computational overhead:

1. do not return to state we come from, i.e., expand function will skip possible successors that are in same state as node’s parent.

2. do not create paths with cycles, i.e., expand function will skip possible successors that are in same state as any of node’s ancestors.

3. do not generate any state that was ever generated before, by keeping track (in memory) of every state generated, unless the cost of reaching that state is lower than last time we reached it.
Examples: Avoiding repeated states

Assembly example