Artificial Intelligence

2. State Space Search

prof. dr. sc. Bojana Dalbelo Bašić
doc. dr. sc. Jan Šnajder

University of Zagreb
Faculty of Electrical Engineering and Computing (FER)

Academic Year 2014/2015
Motivation

- Many analytical problems can be solved by searching through a space of possible states
- Starting from an initial state, we try to reach a goal state
- Sequence of actions leading from initial to goal state is the solution to the problem
- The issues: large number of states and many choices to make in each state
- Search must be performed in a systematic manner
Typical problems...
Formal description of the problem

- Let $S$ be a set of states (state space)
- A search problem consists of initial state, transitions between states, and a goal state (or many goal states)
Formal description of the problem

- Let $S$ be a set of states (state space)
- A search problem consists of initial state, transitions between states, and a goal state (or many goal states)

**Search problem**

$$problem = (s_0, succ, goal)$$

1. $s_0 \in S$ is the **initial state**
2. $succ : S \rightarrow \wp(S)$ is a **successor function** defining the state transitions
3. $goal : S \rightarrow \{\top, \bot\}$ is a **test predicate** to check if a state is a goal state
Formal description of the problem

- Let \( S \) be a set of states (state space).
- A search problem consists of initial state, transitions between states, and a goal state (or many goal states).

Search problem

\[
\text{problem} = (s_0, \text{succ}, \text{goal})
\]

1. \( s_0 \in S \) is the **initial state**
2. \( \text{succ} : S \to \wp(S) \) is a **successor function** defining the state transitions
3. \( \text{goal} : S \to \{\top, \bot\} \) is a **test predicate** to check if a state is a goal state

The successor function can be defined either explicitly (as a map from input to output states) or implicitly (using a set of **operators** that act on a state and transform it into a new state).
An example: A journey through Istria

How to reach Pula from Buzet?
An example: A journey through Istria

How to reach Pula from Buzet?

\[ \text{problem} = (s_0, \text{succ}, \text{goal}) \]

\[ s_0 = \text{Pula} \]

\[ \text{succ}(\text{Pula}) = \{\text{Barban, Medulin, Vodnjan}\} \]

\[ \text{succ}(\text{Vodnjan}) = \{\text{Kanfanar, Pula}\} \]

\[ \vdots \]

\[ \text{goal}(\text{Buzet}) = \top \]

\[ \text{goal}(\text{Motovun}) = \bot \]

\[ \text{goal}(\text{Pula}) = \bot \]

\[ \vdots \]
Why Buzet?

Giant truffle omelette
Another example: 8-puzzle

initial state:

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

What sequence of actions leads to the goal state?

goal 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>
```
Another example: 8-puzzle

initial state:

```
8  7
6  5  4
3  2  1
```

goal state:

```
1  2  3
4  5  6
7  8
```

What sequence of actions leads to the goal state?

\[
\text{problem} = (s_0, \text{succ}, \text{goal})
\]

\[
s_0 = \begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}
\]

\[
\text{succ}\left(\begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}\right) = \left\{\begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}, \begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}, \begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}\right\}
\]

\[
\text{goal}\left(\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8
\end{bmatrix}\right) = \top
\]

\[
\text{goal}\left(\begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}\right) = \bot
\]

\[
\text{goal}\left(\begin{bmatrix}
8 & 7 \\
6 & 5 & 4 \\
3 & 2 & 1
\end{bmatrix}\right) = \bot
\]

\[
\vdots
\]
State space search – the basic idea

- State space search amounts to a search through a **directed graph** (digraph)
- Graph nodes = states
- Arcs (directed edges) = transitions between states
- Graph may be defined explicitly or implicitly
- Graph may contain cycles
State space search – the basic idea

- State space search amounts to a search through a **directed graph** (digraph)
- Graph nodes = states
- Arcs (directed edges) = transitions between states
- Graph may be defined explicitly or implicitly
- Graph may contain cycles
- If we also need the transition costs, we work with a **weighted directed graph**
Search tree

- By searching through a digraph, we gradually construct a search tree.
- We do this by expanding one node after the other: we use the successor function to generate the descendants of each node.
Search tree

- By searching through a digraph, we gradually construct a **search tree**
- We do this by **expanding** one node after the other: we use the successor function to generate the descendants of each node
- **Open nodes** or “the front”: nodes that have been generated, but have not yet been expanded
- **Closed nodes**: already expanded nodes
Search tree

- By searching through a digraph, we gradually construct a **search tree**
- We do this by **expanding** one node after the other: we use the successor function to generate the descendants of each node
- **Open nodes** or “the front”: nodes that have been generated, but have not yet been expanded
- **Closed nodes**: already expanded nodes

**Search strategy**

The **search strategy** is defined by the order in which the nodes are expanded. Different orders yield different strategies.
State space vs. search tree

- Search tree is **created** by searching through the state space
State space vs. search tree

Search tree is created by searching through the state space.

Search tree can be infinite even if the state space is finite.
State space vs. search tree

- Search tree is **created** by searching through the state space
- Search tree can be infinite even if the state space is finite

**NB:** state space contains cycles $\Rightarrow$ search tree is infinite
State vs. node

- Node \( n \) is a data structure comprising the search tree
- A node stores a state, as well as some additional data:
State vs. node

- Node $n$ is a data structure comprising the search tree
- **A node stores a state**, as well as some additional data:

---

**Node data structure**

\[ n = (s, d) \]

- $s$ – state
- $d$ – depth of the node in the search tree
State vs. node

- Node $n$ is a data structure comprising the search tree
- **A node stores a state**, as well as some additional data:

**Node data structure**

$$n = (s, d)$$

- $s$ – state
- $d$ – depth of the node in the search tree

**state**$(n) = s$, **depth**$(n) = d$

**initial**$(s) = (s, 0)$
General search algorithm

```plaintext
function search(s₀, succ, goal)
    open ← [ initial(s₀) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n, succ) do
            insert(m, open)
    return fail
```

- `removeHead(l)` – removes the first element from a nonempty list `l`
- `expand(n, succ)` – expands node `n` using successor function `succ`
- `insert(n, l)` – inserts node `n` into list `l`
General search algorithm

function search(s₀, succ, goal)
    open ← [initial(s₀)]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n, succ) do
            insert(m, open)
    return fail

Q: Is this a deterministic algorithm?
Node expansion

- When expanding a node, we must update all components stored within it:

\[
\text{function}\ expand(n,\ succ) \\
\quad \text{return}\ \{ (s, \text{depth}(n) + 1) \mid s \in \text{succ}(\text{state}(n)) \} \\
\]

- The function gets more complex as we store more data in a node (e.g., a pointer to the parent node)
Comparing problems and algorithms

Problem properties:

- $|S|$ – number of states
- $b$ – search tree branching factor
- $d$ – depth of the optimal solution in the search tree
- $m$ – maximum depth of the search tree (possibly $\infty$)
Comparing problems and algorithms

Problem properties:
- \(|S|\) – number of states
- \(b\) – search tree branching factor
- \(d\) – depth of the optimal solution in the search tree
- \(m\) – maximum depth of the search tree (possibly \(\infty\))

Algorithm properties:
1. **Completeness** – an algorithm is complete iff it finds a solution whenever the solution exists
Comparing problems and algorithms

Problem properties:

- $|S|$ – number of states
- $b$ – search tree branching factor
- $d$ – depth of the optimal solution in the search tree
- $m$ – maximum depth of the search tree (possibly $\infty$)

Algorithm properties:

1. Completeness – an algorithm is complete iff it finds a solution whenever the solution exists
2. Optimality (admissibility) – an algorithm is optimal iff the solution it finds is optimal (has the smallest cost)
Comparing problems and algorithms

Problem properties:

- \(|S|\) – number of states
- \(b\) – search tree branching factor
- \(d\) – depth of the optimal solution in the search tree
- \(m\) – maximum depth of the search tree (possibly \(\infty\))

Algorithm properties:

1. **Completeness** – an algorithm is complete iff it finds a solution whenever the solution exists
2. **Optimality** (admissibility) – an algorithm is optimal iff the solution it finds is optimal (has the smallest cost)
3. **Time complexity** (number of generated nodes)
4. **Space complexity** (number of stored nodes)
Search strategies

There are two types of strategies:

- **Blind** (uninformed) search
- **Heuristic** (directed, informed) search
Search strategies

There are two types of strategies:

- **Blind** (uninformed) search
- **Heuristic** (directed, informed) search

Today we focus on blind search.
Blind search

1. Breadth-first search (BFS)
2. Uniform cost search
3. Depth-first search (DFS)
4. Depth-limited search
5. Iterative deepening search
6. Bidirectional search
Breadth-first search

- The simplest blind search strategy
- Upon expanding the root node, we expand its children, then we expand their children, etc.
- In general, nodes at level $d$ are expanded only after all nodes at depth $d - 1$ have been expanded, i.e., we search **level-by-level**
Breadth-first search

- The simplest blind search strategy
- Upon expanding the root node, we expand its children, then we expand their children, etc.
- In general, nodes at level $d$ are expanded only after all nodes at depth $d - 1$ have been expanded, i.e., we search **level-by-level**
Breadth-first search

- The simplest blind search strategy
- Upon expanding the root node, we expand its children, then we expand their children, etc.
- In general, nodes at level \( d \) are expanded only after all nodes at depth \( d - 1 \) have been expanded, i.e., we search level-by-level

```
A, B, C, D, E, F, G, H, ...
```

```
A
 / \  /
B  C
 / \ / \  /
D E F G
 / \ / \ / \  /
H I J K L M N O
```
We get the BFS strategy if we always insert the generated nodes at the end of the open nodes list.

**Breadth-first search**

```plaintext
function breadthFirstSearch(s0, succ, goal)
    open ← [ initial(s0) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n, succ) do
            insertBack(m, open)
    return fail
```

List `open` now functions as a queue (FIFO).

---

Dalbelo Bašić, Šnajder (UNIZG FER)  AI – State space search  Academic Year 2014/2015  18 / 52
Breadth-first search – example of execution

\( open = [(Pula, 0)] \)
Breadth-first search – example of execution

0 \( \text{open} = [\, (Pula, 0) \,] \)

1 \( \text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\} \)
\( \text{open} = [\, (Vodnjan, 1), (Barban, 1), (Medulin, 1) \,] \)
Breadth-first search – example of execution

0 \hspace{1em} \textit{open} = [(Pula, 0)]

1 \hspace{1em} \text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\}
\hspace{1em} \textit{open} = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)]

2 \hspace{1em} \text{expand}(Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\}
\hspace{1em} \textit{open} = [(Barban, 1), (Medulin, 1), (Kanfanar, 2), (Pula, 2)]
Breadth-first search – example of execution

0 \hspace{10pt} open = [(Pula, 0)]

1 expand(Pula, 0) = {(Vodnjan, 1), (Barban, 1), (Medulin, 1)}
open = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)]

2 expand(Vodnjan, 1) = {(Kanfanar, 2), (Pula, 2)}
open = [(Barban, 1), (Medulin, 1), (Kanfanar, 2), (Pula, 2)]

3 expand(Barban, 1) = {(Labin, 2), (Pula, 2)}
open = [(Medulin, 1), (Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2)]
Breadth-first search – example of execution

0  \( \text{open} = [(Pula, 0)] \)

1  \( \text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\} \)
    \( \text{open} = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)] \)

2  \( \text{expand}(Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\} \)
    \( \text{open} = [(Barban, 1), (Medulin, 1), (Kanfanar, 2), (Pula, 2)] \)

3  \( \text{expand}(Barban, 1) = \{(Labin, 2), (Pula, 2)\} \)
    \( \text{open} = [(Medulin, 1), (Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2)] \)

4  \( \text{expand}(Medulin, 1) = \{(Pula, 2)\} \)
    \( \text{open} = [(Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2), (Pula, 2)] \)
Breadth-first search – example of execution

0 \hspace{1em} open = [(Pula, 0)]

1 \hspace{1em} expand(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\}
   \hspace{1em} open = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)]

2 \hspace{1em} expand(Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\}
   \hspace{1em} open = [(Barban, 1), (Medulin, 1), (Kanfanar, 2), (Pula, 2)]

3 \hspace{1em} expand(Barban, 1) = \{(Labin, 2), (Pula, 2)\}
   \hspace{1em} open = [(Medulin, 1), (Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2)]

4 \hspace{1em} expand(Medulin, 1) = \{(Pula, 2)\}
   \hspace{1em} open = [(Kanfanar, 2), (Pula, 2), (Labin, 2), (Pula, 2), (Pula, 2)]

5 \hspace{1em} expand(Kanfanar, 2) = \{(Baderna, 3), (Rovinj, 3), (Vodnjan, 3)(Zminj, 3)\}
   \hspace{1em} open = [(Pula, 2), (Labin, 2), (Pula, 2), (Pula, 2), (Pula, 2), (Baderna, 3), \ldots]
Breadth-first search – properties

- Breadth-first search is **complete** and **optimal**
- Each search step expands a node from the shallowest level, thus the strategy must be optimal (assuming constant transition costs)

**Time complexity:**

\[ 1 + b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b) = \mathcal{O}(b^{d+1}) \]

(for nodes at the last level, all successors are generated, except for the very last node, which is the goal node)

**Space complexity:** \( \mathcal{O}(b^{d+1}) \)

- Exponential complexity (especially space complexity) is the biggest downside of BFS
- E.g. \( b = 4, d = 16 \), \( 10 \text{ B/nodes} \rightarrow 43 \text{ GB} \)
- BFS is applicable only to small problems
A quick reminder: Asymptotic algorithm complexity

- Asymptotic complexity of a function: the behavior of function $f(n)$ when $n \rightarrow \infty$ expressed in terms of more simple functions

**“Big-O” notation**

$$\mathcal{O}(g(n)) = \{ f(n) \mid \exists c, n_0 \geq 0 \text{ such that } \forall n \geq n_0. 0 \leq f(n) \leq c \cdot g(n) \}$$

- By convention, we write $f(n) = \mathcal{O}(g(n))$ instead of $f(n) \in \mathcal{O}(g(n))$
- This is the upper complexity bound (worst-case complexity)
- Lower complexity bound is not defined here.
  Hence, e.g., $n = \mathcal{O}(n)$, $n = \mathcal{O}(n^2)$, …
  (in principle we are interested in the least upper bound)
- $\Theta(g(n))$ defines the upper and the lower bound (tight bounds)
Transition costs

- If operations (state transitions) differ in cost, we can modify the successor functions to also return the transition costs for each generated state:

\[
\text{succ} : S \rightarrow \mathcal{P}(S \times \mathbb{R}^+)
\]
Transition costs

- If operations (state transitions) differ in cost, we can modify the successor functions to also return the transition costs for each generated state:

\[
\text{succ} : S \rightarrow \wp(S \times \mathbb{R}^+)
\]

- In each node, instead of node’s depth, we now store the total path cost from the initial state to the given node:

\[
n = (s, c), \quad g(n) = c
\]
Transition costs

- If operations (state transitions) differ in cost, we can modify the successor functions to also return the transition costs for each generated state:

\[ \text{succ} : S \rightarrow \wp(S \times \mathbb{R}^+) \]

- In each node, instead of node’s depth, we now store the **total path cost** from the initial state to the given node:

\[ n = (s, c), \quad g(n) = c \]

- We also need to modify the node expansion function so that it updates the path cost:

```
function expand(n, succ)
    return \{ (s, g(n) + c) \mid (s, c) \in \text{succ}(\text{state}(n)) \}
```
An example: A journey through Istria

How to reach Pula from Buzet?

\[
\text{problem} = (s_0, \text{succ}, \text{goal})
\]

\[
s_0 = \text{Pula}
\]

\[
\text{succ} (\text{Pula}) = \{(\text{Barban}, 28), (\text{Medulin}, 9), (\text{Vodnjan}, 12)\}
\]

\[
\text{succ} (\text{Vodnjan}) = \{(\text{Kanfanar}, 29), (\text{Pula}, 12)\}
\]

... goal(\text{Buzet}) = \top

goal(\text{Motovun}) = \bot

goal(\text{Pula}) = \bot

...
Uniform cost search

- Similar to BFS, but accounts for transition costs

### Uniform cost search

**function** uniformCostSearch(s₀, succ, goal)

1. \( \text{open} \leftarrow [\text{initial}(s₀)] \)
2. **while** \( \text{open} \neq [] \) **do**
   1. \( n \leftarrow \text{removeHead(} \text{open} \text{)} \)
   2. **if** \( \text{goal(state}(n)) \) **then return** \( n \)
   3. **for** \( m \in \text{expand}(n, \text{succ}) \) **do**
      1. \( \text{insertSortedBy}(g, m, \text{open}) \)
3. **return** fail

- \( \text{insertSortedBy}(f, n, l) \) – inserts node \( n \) into a list \( l \) sorted by increasing values of \( f(n) \)
- List \( \text{open} \) now functions as a **priority queue**
Uniform cost search – example of execution

0 \quad open = [(Pula, 0)]
Uniform cost search – example of execution

0. \( \text{open} = \{ (Pula, 0) \} \)

1. \( \text{expand}(Pula, 0) = \{ (Vodnjan, 12), (Barban, 28), (Medulin, 9) \} \)
   \( \text{open} = \{ (Medulin, 9), (Vodnjan, 12), (Barban, 28) \} \)
Uniform cost search – example of execution

0  $open = [(Pula, 0)]$

1  $\text{expand}(Pula, 0) = \{(Vodnjan, 12), (Barban, 28), (Medulin, 9)\}$
   $open = [(Medulin, 9), (Vodnjan, 12), (Barban, 28)]$

2  $\text{expand}(Medulin, 9) = \{(Pula, 18)\}$
   $open = [(Vodnjan, 12), (Pula, 18), (Barban, 28)]$
Uniform cost search – example of execution

0. $open = [(Pula, 0)]$

1. $expand(Pula, 0) = \{(Vodnjan, 12), (Barban, 28), (Medulin, 9)\}$
   $open = [(Medulin, 9), (Vodnjan, 12), (Barban, 28)]$

2. $expand(Medulin, 9) = \{(Pula, 18)\}$
   $open = [(Vodnjan, 12), (Pula, 18), (Barban, 28)]$

3. $expand(Vodnjan, 12) = \{(Kanfanar, 41), (Pula, 24)\}$
   $open = [(Pula, 18), (Pula, 24), (Barban, 28), (Kanfanar, 41)]$
Uniform cost search – example of execution

0. $\text{open} = [(Pula, 0)]$

1. $\text{expand}(Pula, 0) = \{(Vodnjan, 12), (Barban, 28), (Medulin, 9)\}$
   $\text{open} = [(Medulin, 9), (Vodnjan, 12), (Barban, 28)]$

2. $\text{expand}(Medulin, 9) = \{(Pula, 18)\}$
   $\text{open} = [(Vodnjan, 12), (Pula, 18), (Barban, 28)]$

3. $\text{expand}(Vodnjan, 12) = \{(Kanfanar, 41), (Pula, 24)\}$
   $\text{open} = [(Pula, 18), (Pula, 24), (Barban, 28), (Kanfanar, 41)]$

4. $\text{expand}(Pula, 18) = \{(Vodnjan, 30), (Barban, 46), (Medulin, 27)\}$
   $\text{open} = [(Pula, 24), (Medulin, 27), (Barban, 28), (Vodnjan, 30), \ldots]$

Q: Will this algorithm eventually reach the goal state (Buzet)?
Q: Will it find the shortest path to Buzet?
Uniform cost search – example of execution

0 \quad open = [(Pula, 0)]

1 \quad expand(Pula, 0) = {(Vodnjan, 12), (Barban, 28), (Medulin, 9)}
   \quad open = [(Medulin, 9), (Vodnjan, 12), (Barban, 28)]

2 \quad expand(Medulin, 9) = {(Pula, 18)}
   \quad open = [(Vodnjan, 12), (Pula, 18), (Barban, 28)]

3 \quad expand(Vodnjan, 12) = {(Kanfanar, 41), (Pula, 24)}
   \quad open = [(Pula, 18), (Pula, 24), (Barban, 28), (Kanfanar, 41)]

4 \quad expand(Pula, 18) = {(Vodnjan, 30), (Barban, 46), (Medulin, 27)}
   \quad open = [(Pula, 24), (Medulin, 27), (Barban, 28), (Vodnjan, 30), \ldots ]

Q: Will this algorithm eventually reach the goal state (Buzet)?
Q: Will it find the shortest path to Buzet?
Uniform cost search – properties

- Uniform cost search is **complete** and **optimal**
- If $C^*$ is the optimal (minimum) cost to reach the goal state and $\varepsilon$ is the minimum transition cost, then the goal state is at depth $d = \left\lfloor \frac{C^*}{\varepsilon} \right\rfloor$ in the search tree
- Time and space complexity: $O(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor})$
Depth-first search

- Depth-first search (DFS) always expands the deepest node in the search tree
- The search returns to the upper level nodes only after reaching the leaf node (a node without descendants)
Depth-first search

- Depth-first search (DFS) always expands the deepest node in the search tree.
- The search returns to the upper level nodes only after reaching the leaf node (a node without descendants).

```
    A
   / \
  B   C
 /\   /\  
D E F G
/\ /\ /\  
H I J K L M N O
```
Depth-first search

- Depth-first search (DFS) always expands the deepest node in the search tree.
- The search returns to the upper level nodes only after reaching the leaf node (a node without descendants).

```
A, B, D, H, I, E, J, K, C, ...
```

![Diagram of depth-first search](image)
Depth-first search – implementation

- We get DFS strategy if we insert the generated nodes at the beginning of the open nodes list

Depth-first search

```plaintext
function depthFirstSearch(s₀, succ, goal)
    open ← [ initial(s₀) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n, succ) do
            insertFront(m, open)
    return fail
```

- List open now functions as a stack (LIFO)
Depth-first search – example of execution

0 \( \text{open} = [(Pula, 0)] \)
Depth-first search – example of execution

0 \hspace{1em} open = [(Pula, 0)]

1 \hspace{1em} expand(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\}
   open = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)]
Depth-first search – example of execution

0. $open = [(Pula, 0)]$

1. $\text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\}$
   $open = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)]$

2. $\text{expand}(Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\}$
   $open = [(Kanfanar, 2), (Pula, 2), (Barban, 1), (Medulin, 1)]$
Depth-first search – example of execution

0 open = [(Pula, 0)]

1 expand(Pula, 0) = {(Vodnjan, 1), (Barban, 1), (Medulin, 1)}
open = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)]

2 expand(Vodnjan, 1) = {(Kanfanar, 2), (Pula, 2)}
open = [(Kanfanar, 2), (Pula, 2), (Barban, 1), (Medulin, 1)]

3 expand(Kanfanar, 2) =
{(Baderna, 3), (Rovinj, 3), (Vodnjan, 3), (Zminj, 3)}
open = [(Baderna, 3), (Rovinj, 3), (Vodnjan, 3), (Zminj, 3), (Pula, 2), ...]
Depth-first search – example of execution

0. \( \text{open} = [(Pula, 0)] \)

1. \( \text{expand}(Pula, 0) = \{(Vodnjan, 1), (Barban, 1), (Medulin, 1)\} \)
   \( \text{open} = [(Vodnjan, 1), (Barban, 1), (Medulin, 1)] \)

2. \( \text{expand}(Vodnjan, 1) = \{(Kanfanar, 2), (Pula, 2)\} \)
   \( \text{open} = [(Kanfanar, 2), (Pula, 2), (Barban, 1), (Medulin, 1)] \)

3. \( \text{expand}(Kanfanar, 2) = \{(Baderna, 3), (Rovinj, 3), (Vodnjan, 3), (Zminj, 3)\} \)
   \( \text{open} = [(Baderna, 3), (Rovinj, 3), (Vodnjan, 3), (Zminj, 3), (Pula, 2), \ldots] \)

4. \( \text{expand}(Baderna, 3) = \{(Porec, 4), (Visnjan, 4), (Pazin, 4), (Kanfanar, 4)\} \)
   \( \text{open} = [(Porec, 4), (Visnjan, 4), (Pazin, 4), (Kanfanar, 4), (Baderna, 3), \ldots] \)
   \( \vdots \)

Q: Is this the only possible execution trace?
Depth-first search – properties

- Depth-first search is less memory-demanding than BFS
- **Space complexity**: $O(bm)$, where $m$ is the maximum tree depth
- **Time complexity**: $O(b^m)$
  (unfavorable if $m \gg d$)
- **Completeness**: no, because it might loop infinitely by cycling
- **Optimality**: no, because it does not search level-by-level

DFS should be avoided if the maximum search tree depth is large or infinite
Depth-first search – properties

- Depth-first search is less memory-demanding than BFS
- **Space complexity**: $\mathcal{O}(bm)$, where $m$ is the maximum tree depth
- **Time complexity**: $\mathcal{O}(b^m)$
  (unfavorable if $m \gg d$)
- **Completeness**: no, because it might loop infinitely by cycling
Depth-first search – properties

- Depth-first search is less memory-demanding than BFS
- **Space complexity**: $O(bm)$, where $m$ is the maximum tree depth
- **Time complexity**: $O(b^m)$
  (unfavorable if $m \gg d$)
- **Completeness**: no, because it might loop infinitely by cycling
- **Optimality**: 
Depth-first search – properties

- Depth-first search is less memory-demanding than BFS
- **Space complexity:** $O(bm)$, where $m$ is the maximum tree depth
- **Time complexity:** $O(b^m)$
  (unfavorable if $m \gg d$)
- **Completeness:** no, because it might loop infinitely by cycling
- **Optimality:** no, because it does not search level-by-level
Depth-first search – properties

- Depth-first search is less memory-demanding than BFS
- **Space complexity**: $O(bm)$, where $m$ is the maximum tree depth
- **Time complexity**: $O(b^m)$
  (unfavorable if $m \gg d$)
- **Completeness**: no, because it might loop infinitely by cycling
- **Optimality**: no, because it does not search level-by-level
- DFS should be avoided if the maximum search tree depth is large or infinite
Depth first search – recursive implementation

- We can avoid using the open list:

```plaintext
Depth first search (recursive implementation)

function depthFirstSearch(s, succ, goal)
    if goal(s) then return s
    for m ∈ succ(s) do
        r ← depthFirstSearch(m, succ, goal)
        if r ≠ fail then return r
    return fail
```

- We use the system stack instead of explicitly using the open list
- Space complexity reduces to $O(m)$
Depth-limited search

- Like depth-first search, but stops at a given depth

\[
\begin{align*}
k = 0: & \quad A \\
k = 1: & \quad A, B, C \\
k = 2: & \quad A, B, D, E, C, F, G
\end{align*}
\]
Depth-limited search – implementation

- The node is expanded only if above the depth limit $k$:

```plaintext
Depth-limited search

function depthLimitedSearch(s_0, succ, goal, k)
    open ← [ initial(s_0) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return s
        if depth(n) < k then
            for m ∈ expand(n, succ) do
                insertFront(m, open)
        end if
    end while
    return fail
```

Dalbelo Bašić, Šnajder (UNIZG FER)  AI – State space search  Academic Year 2014/2015  33 / 52
Depth-limited search – properties

- **Space complexity**: $O(bk)$, where $k$ is the depth limit
- **Time complexity**: $O(b^k)$
Depth-limited search – properties

- **Space complexity**: $O(bk)$, where $k$ is the depth limit
- **Time complexity**: $O(b^k)$
- **Completeness**: Yes, provided $d \leq k$

This algorithm is useful if we know the solution depth $d$ (we can set $k = |S|$ for reasonably sized state spaces).
Depth-limited search – properties

- **Space complexity**: $O(bk)$, where $k$ is the depth limit
- **Time complexity**: $O(b^k)$
- **Completeness**: yes, provided $d \leq k$
Depth-limited search – properties

- **Space complexity**: $O(bk)$, where $k$ is the depth limit
- **Time complexity**: $O(b^k)$
- **Completeness**: yes, provided $d \leq k$
- **Optimality**: no, because it does not search level-by-level

This algorithm is useful if we know the solution depth $d$ (we can set $k = \lvert S \rvert$ for reasonably-sized state spaces)
Depth-limited search – properties

- **Space complexity**: $O(bk)$, where $k$ is the depth limit
- **Time complexity**: $O(b^k)$
- **Completeness**: yes, provided $d \leq k$
- **Optimality**: no, because it does not search level-by-level
Depth-limited search – properties

- **Space complexity**: $O(bk)$, where $k$ is the depth limit
- **Time complexity**: $O(b^k)$
- **Completeness**: yes, provided $d \leq k$
- **Optimality**: no, because it does not search level-by-level

This algorithm is useful if we know the solution depth $d$ (we can set $k = |S|$ for reasonably-sized state spaces)
Iterative deepening search

- Avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- Effectively combines the advantages of DFS and BFS
Iterative deepening search

- Avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- Effectively combines the advantages of DFS and BFS
Iterative deepening search

- Avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- Effectively combines the advantages of DFS and BFS
Iterative deepening search

- Avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- Effectively combines the advantages of DFS and BFS
Iterative deepening search

- Avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- Effectively combines the advantages of DFS and BFS

A, A, B, C, A, B, D, E, C, F, G

A, B, D, H, I, E, J, K, ...
Iterative deepening search

- Avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- Effectively combines the advantages of DFS and BFS
Iterative deepening search – implementation

Iterative deepening search

```plaintext
function iterativeDeepeningSearch(s₀, succ, goal)
    for k ← 0 to ∞ do
        result ← depthLimitedSearch(s₀, succ, goal, k)
        if result ≠ fail then return result
    return fail
```
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.

Time complexity: $O(b^d)$

Space complexity: $O(bd)$

Completeness: yes, because it uses a depth limit and gradually increases it.

Optimality: yes, because it searches level-by-level.

Iterative deepening search is the recommended strategy for problems with big search spaces and unknown solution depths.
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.

- **Time complexity**: $O(b^d)$
- **Space complexity**: $O(bd)$
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.
- **Time complexity**: $O(b^d)$
- **Space complexity**: $O(bd)$
- **Completeness**: Yes, because it uses a depth limit and gradually increases it.
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.
- **Time complexity**: $O(b^d)$
- **Space complexity**: $O(bd)$
- **Completeness**: yes, because it uses a depth limit and gradually increases it.
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.
- **Time complexity**: $O(b^d)$
- **Space complexity**: $O(bd)$
- **Completeness**: yes, because it uses a depth limit and gradually increases it.
- **Optimality**:
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.

**Time complexity**: $O(b^d)$

**Space complexity**: $O(bd)$

**Completeness**: yes, because it uses a depth limit and gradually increases it.

**Optimality**: yes, because it searches level-by-level.
Iterative deepening search – properties

- At first glance, the strategy seems utterly inefficient: the same nodes are expanded many times over again.
- In most cases this is not a problem: the majority of nodes are positioned at lower levels, so repeated expansion of the remaining higher level nodes is not problematic.

**Time complexity**: $O(b^d)$

**Space complexity**: $O(bd)$

**Completeness**: yes, because it uses a depth limit and gradually increases it.

**Optimality**: yes, because it searches level-by-level.

Iterative deepening search is the recommended strategy for problems with big search spaces and unknown solution depths.
Iterative deepening search – complexity

- Number of nodes generated by BFS is

\[ 1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b) \]

thus asymptotic time complexity is \( O(b^{d+1}) \)
Iterative deepening search – complexity

- Number of nodes generated by BFS is

\[ 1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b) \]

thus asymptotic time complexity is \( O(b^{d+1}) \)

- Number of nodes generated by iterative deepening search is

\[ (d + 1)1 + db + (d - 1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \]

thus asymptotic time complexity is \( O(b^d) \)
Iterative deepening search – complexity

- Number of nodes generated by BFS is

\[ 1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b) \]

thus asymptotic time complexity is \( \mathcal{O}(b^{d+1}) \)

- Number of nodes generated by iterative deepening search is

\[ (d + 1)1 + db + (d - 1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d \]

thus asymptotic time complexity is \( \mathcal{O}(b^d) \)

- The difference decreases as branching factor \( b \) increases

- E.g. for \( b = 2 \): 100% more nodes are generated
  for \( b = 3 \): 50% more nodes are generated
  for \( b = 10 \): 11% more nodes are generated
Bidirectional search

- Two searches are run simultaneously: one from the initial state to the goal state, another from the goal state to the initial state.
- In the worst case, search ends when two fronts meet halfway in between.

E.g., if BFS is used for both directions, space and time complexity is \( O(2^{b/2}) = O(b^{d/2}) \) = Significant savings!

Drawbacks: applicable only if the problem has (1) a small number of goal states and (2) all operators are invertible.
Bidirectional search

- Two searches are run simultaneously: one from the initial state to the goal state, another from the goal state to the initial state.
- In the worst case, search ends when two fronts meet halfway in between.
- E.g., if BFS is used for both directions, space and time complexity is $O(2b^{d/2}) = O(b^{d/2})$.
  Significant savings!
Bidirectional search

- Two searches are run simultaneously: one from the initial state to the goal state, another from the goal state to the initial state.
- In the worst case, search ends when two fronts meet halfway in between.
- E.g., if BFS is used for both directions, space and time complexity is $O(2b^{d/2}) = O(b^{d/2})$
  
  Significant savings!
- Drawbacks: applicable only if the problem has (1) a small number of goal states and (2) all operators are invertible.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Complete</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{d+1})$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Uniform cost</td>
<td>$O(b^{1+[C^*]/\epsilon})$</td>
<td>$O(b^{1+[C^*]/\epsilon})$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DFS</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Depth-limited</td>
<td>$O(b^k)$</td>
<td>$O(bk)$</td>
<td>Yes, if $d \leq k$</td>
<td>No</td>
</tr>
<tr>
<td>Iter. deepening</td>
<td>$O(b^d)$</td>
<td>$O(bd)$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>$O(b^{d/2})$</td>
<td>$O(b^{d/2})$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(b\) – branching factor, \(d\) – optimal solution depth, 
\(m\) – maximum tree depth \((m \geq d)\), \(k\) – depth limit

- All algorithms have **exponential time complexity**!
- DFS (and its variants) has lower space complexity than BFS
Reconstruction of the solution

- Node data structure needs to store a pointer back to parent node:

\[ n = (s, d, p), \ \text{parent}(n) = p \]

function expand(n, succ)
    return \{ (s, depth(n) + 1, n) \mid s \in \text{succ(state}(n))) \}
Reconstruction of the solution

- Node data structure needs to store a pointer back to parent node:
\[ n = (s, d, p), \text{ parent}(n) = p \]

- Function `expand(n, succ)`
  ```
  \text{return} \{ (s, \text{depth}(n) + 1, n) \mid s \in \text{succ(state}(n)) \}
  ```

- We can now follow the pointers pointing back from the goal state:

Path reconstruction

- Function `path(n)`
  ```
  p \leftarrow \text{parent}(n)
  \text{if} \ p = \text{null} \text{ then return} \ [\text{state}(n)]
  \text{return} \text{insertBack}(\text{state}(n), \text{path}(p))
  ```
Reconstruction of the solution

- Node data structure needs to store a pointer back to parent node:

\[ n = (s, d, p), \quad \text{parent}(n) = p \]

```python
function expand(n, succ)
    return \{ (s, depth(n) + 1, n) | s ∈ succ(state(n)) \}
```

- We can now follow the pointers pointing back from the goal state:

**Path reconstruction**

```python
function path(n)
    p ← parent(n)
    if p = null then return [state(n)]
    return insertBack(state(n), path(p))
```

- Time complexity is \( O(d) \)
- **NB:** We obviously need to keep the closed nodes in memory!
Avoiding repeated states (1)

Solution 1:
prevent the algorithm to return to the state it came from

General search algorithm (revised)

function search(s_0, succ, goal)

open ← [ initial(s_0) ]

while open ≠ [ ] do

n ← removeHead(open)

if goal(state(n)) then return n

for m ∈ expand(n) do

if state(m) ≠ state(parent(n)) then

insert(m, open)

return fail

Q: Does this prevent the algorithm to loop infinitely?
A: Generally not! It only prevents cycles of length 2 (transpositions)
Avoiding repeated states (1)

- **Solution 1:** prevent the algorithm to return to the state it came from

---

**General search algorithm (revised)**

```plaintext
function search(s₀, succ, goal)
    open ← [ initial(s₀) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n) do
            if state(m) ≠ state(parent(n)) then insert(m, open)
        return fail
```

Q: Does this prevent the algorithm to loop infinitely?
A: Generally not! It only prevents cycles of length 2 (transpositions).
Avoiding repeated states (1)

- **Solution 1:** prevent the algorithm to return to the state it came from

### General search algorithm (revised)

```plaintext
function search(s₀, succ, goal)
    open ← [ initial(s₀) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n) do
            if state(m) ≠ state(parent(n)) then insert(m, open)
        return fail
```

- **Q:** Does this prevent the algorithm to loop infinitely?
Avoiding repeated states (1)

- **Solution 1:** prevent the algorithm to return to the state it came from

**General search algorithm (revised)**

```plaintext
function search(s₀, succ, goal)
    open ← [initial(s₀)]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n) do
            if state(m) ≠ state(parent(n)) then insert(m, open)
        return fail
```

- **Q:** Does this prevent the algorithm to loop infinitely?
  - **A:** Generally not! It only prevents cycles of length 2 (transpositions)
Avoiding repeated states (2)

- **Solution 2:** prevent paths containing cycles

**General search algorithm (revised)**

```plaintext
function search(s₀, succ, goal)
    open ← [ initial(s₀) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n) do
            if state(m) ∉ path(n) then insert(m, open)
        return fail
```

Prevents deadlock in cycles (thus ensures algorithm completeness)

Does not prevent repeated states at different paths of the search tree

Increases the time complexity by a factor of $O(d)$
Avoiding repeated states (2)

- **Solution 2:** prevent paths containing cycles

## General search algorithm (revised)

```plaintext
function search(s₀, succ, goal)
    open ← [ initial(s₀) ]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n) do
            if state(m) ∉ path(n) then insert(m, open)
        return fail
```

- Prevents deadlock in cycles (thus ensures algorithm completeness)
- Does not prevent repeated states at different paths of the search tree
- Increases the time complexity by a factor of $O(d)$
Avoiding repeated states (3)

- **Solution 3**: prevent the repetition of any state whatsoever

---

**General search algorithms with visited states list**

```plaintext
function search(s₀, succ, goal)
  open ← [initial(s₀)]
  visited ← ∅
  while open ≠ [] do
    n ← removeHead(open)
    if goal(state(n)) then return n
    visited ← visited ∪ {state(n)}
    for m ∈ expand(n) do
      if state(m) ∉ visited then insert(m, open)
  return fail

NB: Visited states list (set) contains states, not tree nodes
```
Avoiding repeated states – remarks

- Using visited states list ensures that the algorithm is complete (deadlock in cycles cannot occur)
Avoiding repeated states – remarks

- Using visited states list ensures that the algorithm is complete (deadlock in cycles cannot occur)
- Moreover, using visited states list may decrease space and time complexity:

\[ O(b + 1) \rightarrow O(\min(b + 1, b|S|)) \]

where \(|S|\) is the size of the state space (in practice it is often the case that \(b|S| < b\)).
Avoiding repeated states – remarks

- Using visited states list ensures that the algorithm is complete (deadlock in cycles cannot occur)
- Moreover, using visited states list may decrease space and time complexity:
  Because no state is ever repeated, complexity $O(b^{d+1})$ reduces to $O(\min(b^{d+1}, b|S|))$, where $|S|$ is the size of the state space (in practice it is often the case that $b|S| < b^d$)
Avoiding repeated states – remarks

- Using visited states list ensures that the algorithm is complete (deadlock in cycles cannot occur)
- Moreover, using visited states list may **decrease space and time complexity**:
  Because no state is ever repeated, complexity $O(b^{d+1})$ reduces to $O(\min(b^{d+1}, b|S|))$, where $|S|$ is the size of the state space (in practice it is often the case that $b|S| < b^d$)
- Visited states list is usually implemented as a hash table
Avoiding repeated states – remarks

- Using visited states list ensures that the algorithm is complete (deadlock in cycles cannot occur)
- Moreover, using visited states list may **decrease space and time complexity**:
  Because no state is ever repeated, complexity $O(b^{d+1})$ reduces to $O(\min(b^{d+1}, b|S|))$, where $|S|$ is the size of the state space (in practice it is often the case that $b|S| < b^d$)
- Visited states list is usually implemented as a hash table (enables lookup “state($m$) $\notin$ visited” in $O(1)$)
An example: Missionaries and cannibals (1)

Missionaries and cannibals problem

Three missionaries and three cannibals must be brought over by boat from one side of the river to the other. At no time should the missionaries be outnumbered by the cannibals on either side of the river. The boat can carry up to two passengers and cannot move by itself. We are looking for a solution with the fewest possible number of steps.
Missionaries and cannibals problem

Three missionaries and three cannibals must be brought over by boat from one side of the river to the other. At no time should the missionaries be outnumbered by the cannibals on either side of the river. The boat can carry up to two passengers and cannot move by itself. We are looking for a solution with the fewest possible number of steps.

- Which search algorithm should we use?
- How should we represent the problem?
An example: Missionaries and cannibals (2)

\[ \text{problem} = (s_0, \text{succ}, \text{goal}) \]

\[ s_0 = (3, 3, L) \]
An example: Missionaries and cannibals (2)

\[
problem = (s_0, \text{succ}, \text{goal})
\]

1. \(s_0 = (3, 3, L)\)
   - number of missionaries on the left side of the coast, \(\{0, 1, 2, 3\}\)
An example: Missionaries and cannibals (2)

\[ \text{problem} = (s_0, \text{succ}, \text{goal}) \]

1. \[ s_0 = (3, 3, L) \]
   - number of missionaries on the left side of the coast, \{0, 1, 2, 3\}
   - number of cannibals on the left side of the coast, \{0, 1, 2, 3\}
An example: Missionaries and cannibals (2)

\[ problem = (s_0, \text{succ}, \text{goal}) \]

1. \[ s_0 = (3, 3, L) \]

- number of missionaries on the left side of the coast, \( \{0, 1, 2, 3\} \)
- number of cannibals on the left side of the coast, \( \{0, 1, 2, 3\} \)
- position of the boat, \( \{L, R\} \)
An example: Missionaries and cannibals (2)

\[
\text{problem} = (s_0, \text{succ}, \text{goal})
\]

1. \(s_0 = (3, 3, L)\)
   - number of missionaries on the left side of the coast, \(\{0, 1, 2, 3\}\)
   - number of cannibals on the left side of the coast, \(\{0, 1, 2, 3\}\)
   - position of the boat, \(\{L, R\}\)

2. \(\text{succ}(m, c, b) = \{s \mid s \in \text{Succs}, \text{safe}(s)\}\)

\[
\text{Succs} = \{f(\Delta m, \Delta c) \mid (\Delta m, \Delta c) \in \text{Moves}\}
\]

\[
\text{Moves} = \{(1, 1), (2, 0), (0, 2), (1, 0), (0, 1)\}
\]

\[
f(\Delta m, \Delta c) = \begin{cases} 
(m - \Delta m, c - \Delta c, R) & \text{if } b = L \\
(m + \Delta m, c + \Delta c, L) & \text{otherwise}
\end{cases}
\]

\[
\text{safe}(m, c, b) = (m = 0) \lor (m = 3) \lor (m = c)
\]
An example: Missionaries and cannibals (2)

\[ \text{problem} = (s_0, \text{succ}, \text{goal}) \]

1. \[ s_0 = (3, 3, L) \]
   - number of missionaries on the left side of the coast, \( \{0, 1, 2, 3\} \)
   - number of cannibals on the left side of the coast, \( \{0, 1, 2, 3\} \)
   - position of the boat, \( \{L, R\} \)

2. \[ \text{succ}(m, c, b) = \{ s \mid s \in \text{Succs}, \text{safe}(s) \} \]
   \[ \text{Succs} = \{ f(\Delta m, \Delta c) \mid (\Delta m, \Delta c) \in \text{Moves} \} \]
   \[ \text{Moves} = \{ (1, 1), (2, 0), (0, 2), (1, 0), (0, 1) \} \]
   \[ f(\Delta m, \Delta c) = \begin{cases} (m - \Delta m, c - \Delta c, R) & \text{if } b = L \\ (m + \Delta m, c + \Delta c, L) & \text{otherwise} \end{cases} \]
   \[ \text{safe}(m, c, b) = (m = 0) \vee (m = 3) \vee (m = c) \]

3. \[ \text{goal}(m, c, b) = \begin{cases} \top & \text{if } (m = c = 0) \wedge (b = R) \\ \bot & \text{otherwise} \end{cases} \]
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?

Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
- Should we avoid repeated states?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
- Should we avoid repeated states?
- What are the properties of our problem?
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
- Should we avoid repeated states?
- What are the properties of our problem?
  - $|S| = ?$
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate \texttt{goal}?
- Should we avoid repeated states?
- What are the properties of our problem?
  - $|S| = ?$
  - $b = ?$
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
- Should we avoid repeated states?
- What are the properties of our problem?
  - $|S| = ?$
  - $b = ?$
  - $d = ?$
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate \texttt{goal}?
- Should we avoid repeated states?
- What are the properties of our problem?
  - $|S| = ?$
  - $b = ?$
  - $d = ?$
  - $m = ?$
An example: Missionaries and cannibals (3)

- Have we described all that is relevant for the problem?
- Have we abstracted away the unimportant details?
- Do we generate all possible moves?
- Are all moves that we generate legal?
- Do we generate undesirable states?
- Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?
- Should we avoid repeated states?
- What are the properties of our problem?
  - $|S| =$?
  - $b =$?
  - $d =$?
  - $m =$?
- Is this a difficult problem?
Lab assignment: Jealous husbands problem

Implement the jealous husbands problem and solve it using BFS.

The problem is as follows. Three jealous husbands and their wives need to cross a river using a single boat. At no time should any of the women be left in company with any of the men, unless her husband is present. The boat can carry up to two passengers and cannot move by itself. We are looking for an optimal solution.

The optimal solution is the one with the fewest number of steps. The program should print out the solution by listing a sequence of states and operators needed to reach the goal state from the initial state.

Q: How would you define $S$ and $succ$?
Lab assignment: Countdown numbers game

Implement the countdown numbers game and solve it using iterative deepening search.

The problem is as follows. Use the six given numbers to arithmetically calculate a randomly chosen number. Only the four arithmetic operations of addition, subtraction, multiplication, and division may be used, and no fractions may be introduced into the calculation. Each number may be used at most once.

The initial six numbers should be taken as input from the user. The target number should be generated randomly from 100 to 999.

If the exact expression is not found, the program should print out the expression that evaluates to a value that is the closest to the target value.

Q: How would you define $S$ and $\text{succ}$?
Playground

- Implementation of blind search algorithms (thanks to Marko Čupić)
  - http://www.fer.unizg.hr/_download/repository/pretrazivanja4.zip
Implementation of blind search algorithms (thanks to Marko Čupić)
▶ http://www.fer.unizg.hr/_download/repository/pretrazivanja4.zip

You can focus on formulating the tasks as state search problems
▶ State encoding
▶ Generating state transitions (i.e., the Succ function)
▶ Goal state predicate
Wrap-up

- Many problems can be solved by state space search
- Problems differ in the **number of states**, **branching factor** and solution **depth**
- Desirable algorithm properties are **completeness, optimality** and **small space complexity**
- Unfortunately, all search algorithms have **exponential time complexity**
- When state space is large, **iterative deepening search** is the recommended search strategy
- Take care of cycles (repeated states)

Next topic: **Heuristic search**