Unit 3: Search

José Luis Ruiz Reina
José Antonio Alonso
Francisco J. Martín Mateos
Agustín Riscos

Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Inteligencia Artificial, 2012-13
Definition of a problem as a state space

- Before searching for the solution to a problem:
  - Specification of the problem

- Elements of the problem:
  - Which is the initial situation where we start?
  - Which is the final goal?
  - How to describe all possible intermediate situations or states that we could encounter?
  - Which elementary steps or actions are available to modify the state and how do they work?

- Specifying a problem as a state space consists on describing in a precise way each one of such components
  - Advantage: general procedures for searching solutions
  - Independent of the problem
Protocol for solving problems

Problem $\xrightarrow{\text{Abstraction}}$ Problem specification

Abstraction

Solution $\xrightarrow{\text{Interpretation}}$ Implementation on some programming language

Aplication of algorithms for searching the solution
Formulating the 8-puzzle problem

- A 3x3 board with 8 numbered tiles on it (thus leaving a blank free space of the same size of a tile). A tile adjacent to the blank space can slide into it. The game consists on transforming the initial position into the final one by means of a sequence of tile slidings. In particular, let us consider the following initial and final states:

```
Initial state

2 8 3
1 6 4
7 5

Final state

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

- **State**: description of a **possible** situation of the problem
  - Abstraction of the properties

- **Importance of a good representation of states**
  - Only information being relevant for the problem should be considered
  - Depending on the chosen representation the number of states will vary, influencing the performance of the procedures for searching solutions

- **Example of the 8-puzzle**: Elements of the representation:
  - relevant: localization of each tile and the space;
  - irrelevant: the material the tiles are made of, colors,...
States Representation

• Example of the 8-puzzle: State representations

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

  • Description of the exact position of each one of the tiles

• Representation vs. implementation
  • Tuples: \((2 \ 8 \ 3 \ 1 \ 6 \ 4 \ 7 \ H \ 5), \ (2 \ 8 \ 3 \ 4 \ 5 \ H \ 7 \ 1 \ 6)\)
  • Nested lists: \(((2 \ 8 \ 3)(1 \ 6 \ 4)(7 \ H \ 5))\)
  • Dictionaries: \{"firstleft":2, "firstcenter":8, ...\}

• Number of states: \(9! = 362,880\).
Actions

• Actions:
  • Represent a finite set of basic operators that transform a states into another one

• Elements describing an action
  • Applicability: precondition and postcondition
  • Resulting state after the application of an (applicable) action over a state

• Criterion for selecting actions.
  • Depends on the representation of the states
  • Preferably representacions with lower number of actions

• Example: Actions for the 8-puzzle:
  • One for each possible movement of a tile: 32.
  • One for each possible movement of the space: 4.
Actions

• Actions for the 8-puzzle
  • Move the space up
  • Move the space down
  • Move the space to the right
  • Move the space to the left

• Description of the action “Move the space up”
  • Applicability: it is applicable over states where the space is not in the first (upper) row
  • Result of applying the action: the space and the tile situated over it swap their positions

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

• Analogously for the remaining three actions
Initial State

- Initial State
  - A state describing the situation when the game begins
- Initial State for the 8-puzzle example

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

• Description of the goal
  • Usually, a set of states, which will be called *final*
  • Eventually (not always) a single final state

• Example of the 8-puzzle (only one final state)

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

• Approaches for defining final states:
  • Enumerative
  • Declarative
Solutions to a problem

Definition

- Sequence of actions to be performed in order to achieve a goal
- Sequence of actions s.t. a final state is reached if they are applied starting from the initial state

Example

A solution to the 8-puzzle problem

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

( up up left down right)
Solutions to a problem

• Types of problems:
  • Searching for a solution.
  • Determining whether there exists a solution and finding a final state.
  • Searching for any solution as fast as possible.
  • Search for all solutions.
  • Search for the shortest solution.
  • Search for the solution of least cost.
State space as a graph

- A state space can be seen as a directed graph
  - Nodes \equiv\ States
  - Successors of a state: those obtained from it by applying any applicable action

- 8-puzzle Example
Elements of the implementation

- Choosing a representation (data structure):
  - for the states
  - for the actions

- An implementation of a problem as a state space consists on:
  - A variable *INITIAL-STATE*
    - Stores the representation of the initial state
  - A function IS-FINAL-STATE(STATE)
    - Checks whether a given state is final or not
  - A function ACTIONS(STATE).
    - Returns the applicable actions for a given state
  - A function APPLY(ACTION, STATE)
    - Yields the state resulting from applying a given (applicable) action over a given state
The Farmer’s problem

- The Farmer’s problem can be stated as follows:
  - A farmer is by a river with a wolf, a goat and a cabbage.
  - He wishes all of them to cross the river.
  - There’s a boat that allows only to bring one thing at a time.
  - The wolf will eat the goat if the farmer is not present.
  - The goat will eat the cabbage if the farmer is not present.
- Information about the states: river bank (left or right) where each element is
  - The river bank of the boat is redundant information
Formulating the Farmer’s problem

- Representation of the states: \((x, y, z, u)\) in \(\{l, r\}^4\).
  - Number of states: 16.
- Initial State: \((l, l, l, l)\).
- Final State (unique): \((r, r, r, r)\).
- Actions:
  - The farmer crosses alone.
  - The farmer crosses with the wolf.
  - The farmer crosses with the goat.
  - The farmer crosses with the cabbage.
Formulating the Farmer’s problem

- Applicability of the actions
  - Precondition (for the last three): the two of them must be in the same river bank
  - Poscondition: the resulting state should not have wolf and goat together, neither goat and cabbage together, except if the farmer is also on their river bank

- State resulting of applying the action
  - Swaping the bank of the elements that go on the boat
The water jug problem

• The water jug problem can be stated as follows:
  • We have two jugs, of 3 and 4 liters respectively.
  • None of them has any measuring markers on it.
  • There is a tap that can be used to fill the jugs with water.
  • Figure out the way to get exactly 2 liters of water into the larger jug (the 4-liter one).

• Representation of the states: \((xy)\) being \(x\) in \(\{0,1,2,3,4\}\) and \(y\) in \(\{0,1,2,3\}\).

• Number of states: 20.
**Formulation of the water jug problem**

- **Initial State:** (0 0).
- **Final States:** every state of the form (2 y).
- **Actions:**
  - Fill the 4-liter jug at the tap.
  - Fill the 3-liter jug at the tap.
  - Empty the 4-liter jug to the drain.
  - Empty the 3-liter jug to the drain.
  - Fill the 4-liter jug by pouring the 3-liter jug in.
  - Fill the 3-liter jug by pouring the 4-liter jug in.
  - Empty the 3-liter jug by pouring it into the 4-liter jug.
  - Empty the 4-liter jug by pouring it into the 3-liter jug.
Formulation of the water jug problem

- Application of actions on a state \((x, y)\)
- Action “Fill the 3-liter jug at the tap”
  - Applicability: \(y < 3\) (precondition)
  - Resulting State: \((x, 3)\)
- Action “Fill the 4-liter jug by pouring the 3-liter jug in”
  - Applicability: \(x < 4\), \(y > 0\), \(x+y > 4\) (precondition)
  - Resulting State: \((4, x+y-4)\)
- Action “Empty the 3-liter jug by pouring it into the 4-liter jug”
  - Applicability: \(y > 0\), \(x+y \leq 4\) (precondition)
  - Resulting State: \((x+y, 0)\)
- Analogously for the rest of the actions
The trip problem

• The trip problem can be stated as follows:
  • We are currently at one of the Andalousian capital cities (e.g. Sevilla).
  • We wish to go to another capital (e.g. Almería).
  • The bus service only connects neighbouring capitals.
Formulation of the trip problem

- 8 possible states:
  - Almería, Cádiz, Córdoba, Granada, Huelva, Jaén, Málaga, Sevilla
- Initial State: Sevilla.
- Final State: Almería.
- Actions:
  - Go to Almería, Go to Cádiz, Go to Córdoba, Go to Granada, Go to Huelva, Go to Jaén, Go to Málaga, Go to Sevilla.
- Example: application of “Go to Málaga” onto a state x
  - Applicability: x and Málaga must be neighbouring capitals
  - Resulting State: Málaga
Real-life problems

- Real-life problems that can be formulated and solved as state spaces:
  - Routing problems in computer networks
  - Aerial routes for traveling by plane
  - The Travelling Salesman Problem
  - Microchip design
  - Components assembly
  - Movement of robots
Recall: Protocol for solving problems

**Problem** → **Abstraction** → **Problem specification**

**Solution** → **Interpretation** →
- **Application of algorithms for searching the solution**
- **Implementation on some programming language**
Searching solutions on state spaces

- Goal: to find a sequence of actions that, if applied from the initial state, will lead to a final state
- Basic idea: exploring the *graph* of the state space
  - At each step the current state is analyzed (starting from the initial state)
  - If the current state is final, then halt (gathering the sequence of actions)
  - Otherwise, get the successors of the current state (*expand*)
  - *Select* a new current state, *keeping* the remaining ones for further analysis (if needed)
  - Repeat the process while there are states to analyze
- The selection of the current state for the next step determines a searching *strategy*
Search trees

- The process just described can be seen as the incremental construction of a search tree.
- Example on the water jug problem:
Search trees

- Nodes of a search tree, components:
  - State
  - Parent node (pointer)
  - Action (applied to the parent)
  - Depth (number of actions since the root node)

- Root node of a search tree: corresponding to the initial state

- Leaf nodes of a search tree:
  - Nodes that do not yield new nodes when being expanded
  - Nodes yet to be considered (and eventually expand)

- Note that it is easy to build the path (from the root node) to the node

- State space vs. Search tree:
  - Search tree nodes: state + how to reach it
  - The tree is incrementally built, and reflects a search process over the graph of the state space
Recall: implementation of a problem via state space

- Choosing a representation (data structure):
  - for the states
  - for the actions

- An implementation of a problem as a state space consists on:
  - A variable *INITIAL-STATE*
    - Stores the representation of the initial state
  - A function IS-FINAL-STATE(STATE)
    - Checks whether a given state is final or not
  - A function ACTIONS(STATE).
    - Returns the applicable actions for a given state
  - A function APPLY(ACTION, STATE)
    - Yields the state resulting from applying a given (applicable) action over a given state
Implementation of the search (auxiliary functions)

• Search nodes: state + parent + action + depth
  • Access functions: STATE(NODE), PARENT(NODE), ACTION(NODE), DEPTH(NODE)

• Successors of a node:

FUNCTION SUCCESSOR(NODE,ACTION)
  1. Let SUCCESSOR-STATE be equal to APPLY(ACTION,STATE(NODE))
  2. Return a node whose state is SUCCESSOR-STATE, whose parent is NODE, whose action is ACTION and whose depth is DEPTH(NODE)+1

FUNCTION SUCCESSORS(NODE)
  1. Let SUCCESSORS be initially empty
  1. For each ACTION in ACTIONS(STATE(NODE)),
     include SUCCESSOR(NODE,ACTION) into SUCCESSORS
  3. Return SUCCESSORS
Implementation of a general search procedure

FUNCTION GENERAL-SEARCH()
1. Let OPEN be the "queue" formed by just the initial node (i.e. the node whose state is *INITIAL-STATE*);
   Let CLOSED be initially empty
2. While OPEN is not empty,
   2.1 Let CURRENT be the first node in OPEN
   2.2 Let OPEN be the rest of OPEN
   2.3 Add the node CURRENT to CLOSED.
   2.4 If IS-FINAL-STATE(STATE(CURRENT)),
      2.4.1 return the node CURRENT and halt.
   2.4.2 otherwise,
      2.4.2.1 Let NEW-SUCCESSORS be the list of nodes from SUCCESSORS(CURRENT) whose state is neither in OPEN nor in CLOSED
      2.4.2.2 Let OPEN be equal to MANAGE-QUEUE (OPEN, NEW-SUCCESSORS)
3. Return FAIL.
General search procedure: comments

• The implementation shown is independent of the problem
• OPEN can be seen as a “queue” containing the nodes who wait to be analyzed (frontier of the search)
• CLOSED contains nodes already analyzed:
  • Allows to avoid starting searches from analyzed states
  • In particular, prevents cycles in the search process
  • In some problems it is not needed
• MANAGE-QUEUE(OPEN, NEW-SUCCESSORS):
  • Adds NEW-SUCCESSORS to OPEN, and orders the result according to some criterion
  • Different definitions of this function determine different search algorithms (search strategies)
  • Uninformed or blind search vs. informed search
Search algorithms. Properties to be studied

- Completeness: if there exists a solution, does it find it?
- Optimal or minimal solution: does it return the solution with least number of steps or the lowest cost?
- Time complexity: how long it takes to find a solution?
- Space complexity: how much memory is required?
Observations about the complexity

- Always using the notation $O$, considering the worst case
- Given with respect to the size of the input problem: $r$ (max number of successors of a node, called *branching factor*), $p$ (min depth of a solution) and $m$ (max depth in the search tree)
- Time complexity: number of analyzed nodes
- Space complexity: max size of OPEN (and CLOSED) during the search process
Implementation of breadth-first-search

- In breadth-first-search, OPEN is handled as a queue (FIFO):

  FUNCTION BREADTH-FIRST-SEARCH()
  1. Let OPEN be the "queue" formed by just the initial node (i.e. the node whose state is *INITIAL-STATE*);
     Let CLOSED be initially empty
  2. While OPEN is not empty,
     2.1 Let CURRENT be the first node in OPEN
     2.2 Let OPEN be the rest of OPEN
     2.3 Add the node CURRENT to CLOSED.
     2.4 If IS-FINAL-STATE(STATE(CURRENT)),
        2.4.1 return the node CURRENT and halt.
        2.4.2 otherwise,
           2.4.2.1 Let NEW-SUCCESSORS be the list of nodes
                     from SUCCESSORS(CURRENT) whose state is neither
                     in OPEN nor in CLOSED
           2.4.2.2 Let OPEN be equal to the result of appending
                     NEW-SUCCESSORS at the end of OPEN
  3. Return FAIL.
Search tree for Breadth-first-search

Water jug problem
Search tree for Breadth-first-search

Water jug problem

(0,0)

(4,0)

(0,3)

(4,3)

(1,3)

(3,0)

(3,3)

(4,2)

(0,2)

(2,0)
Search tree for Breadth-first-search

Water jug problem

(0,0)

(4,0)

(4,3)

(1,3)

(0,3)

(3,0)

(1,0)

(0,1)

(4,1)

(2,3)

(3,3)

(4,2)

(0,2)

(2,0)
Search tree for Breadth-first-search

Water jug problem
Search tree for Breadth-first-search

Water jug problem

State space representation Basic search techniques on state spaces Informed search using heuristic strategies
Search tree for Breadth-first-search

Water jug problem
Search tree for Breadth-first-search

Water jug problem

(0,0)

(4,0)

(4,3)

(1,3)

(1,0)

(0,1)

(4,1)

(4,2)

(0,2)

(2,0)

(3,3)

(3,0)
Search tree for Breadth-first-search

Water jug problem

(4,0) → (0,0) → (0,3) → (3,0) → (3,3) → (4,2)

(1,3) → (1,0) → (0,1) → (4,1) → (2,3) → (0,2) → (2,0)
Search tree for Breadth-first-search

Water jug problem
Search tree for Breadth-first-search

Water jug problem

(0,0) - (4,0)

(4,3) - (1,3) - (1,0) - (0,1) - (4,1) - (2,3)

(0,3) - (3,0) - (3,3) - (4,2) - (0,2) - (2,0)
Search tree for Breadth-first-search

Water jug problem
Search tree for Breadth-first-search

Water jug problem

(0,0) → (4,0) → (1,3) → (1,0) → (0,1) → (4,1) → (2,3)

(0,3) → (3,0) → (3,3) → (4,2) → (0,2) → (2,0)
### Table for Breadth-first-search

#### Water jug problem

<table>
<thead>
<tr>
<th>Node</th>
<th>Current</th>
<th>Successors</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0 0)</td>
<td>(4 0) (0 3)</td>
<td>(4 0) (0 3)</td>
</tr>
<tr>
<td>2</td>
<td>(4 0)</td>
<td>(4 3) (1 3)</td>
<td>(0 3) (4 3) (1 3)</td>
</tr>
<tr>
<td>3</td>
<td>(0 3)</td>
<td>(3 0)</td>
<td>(4 3) (1 3) (3 0)</td>
</tr>
<tr>
<td>4</td>
<td>(4 3)</td>
<td></td>
<td>(1 3) (3 0)</td>
</tr>
<tr>
<td>5</td>
<td>(1 3)</td>
<td>(1 0)</td>
<td>(3 0) (1 0)</td>
</tr>
<tr>
<td>6</td>
<td>(3 0)</td>
<td>(3 3)</td>
<td>(1 0) (3 3)</td>
</tr>
<tr>
<td>7</td>
<td>(1 0)</td>
<td>(0 1)</td>
<td>(3 3) (0 1)</td>
</tr>
<tr>
<td>8</td>
<td>(3 3)</td>
<td>(4 2)</td>
<td>(0 1) (4 2)</td>
</tr>
<tr>
<td>9</td>
<td>(0 1)</td>
<td>(4 1)</td>
<td>(4 2) (4 1)</td>
</tr>
<tr>
<td>10</td>
<td>(4 2)</td>
<td>(0 2)</td>
<td>(4 1) (0 2)</td>
</tr>
<tr>
<td>11</td>
<td>(4 1)</td>
<td>(2 3)</td>
<td>(0 2) (2 3)</td>
</tr>
<tr>
<td>12</td>
<td>(0 2)</td>
<td>(2 0)</td>
<td>(2 3) (2 0)</td>
</tr>
<tr>
<td>13</td>
<td>(2 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

State space representation

Basic search techniques on state spaces

Informed search using heuristic strategies
Properties of Breadth-first-search

- Time complexity $O(b^d)$, where:
  - $b$: branching factor.
  - $d$: depth of the shortest solution.
- Space complexity $O(b^d)$.
- Completeness.
- Returns a solution with the minimum number of actions.
**Limitations of Breadth-first-search**

- Exponential cost in time and space, makes it often not feasible in practice
- Assuming branching \( r = 10 \), \( 10^6 \) nodes per sec. and 1000 bytes per node:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes(\sim)</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>0.11 ms.</td>
<td>107 Kb</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 ms.</td>
<td>10.6 Mb</td>
</tr>
<tr>
<td>6</td>
<td>(10^6)</td>
<td>1.1 s.</td>
<td>1 Gb</td>
</tr>
<tr>
<td>8</td>
<td>(10^8)</td>
<td>2 min.</td>
<td>103 Gb</td>
</tr>
<tr>
<td>10</td>
<td>(10^{10})</td>
<td>3 hours</td>
<td>10 Tb</td>
</tr>
<tr>
<td>12</td>
<td>(10^{12})</td>
<td>13 days</td>
<td>1 Pb</td>
</tr>
<tr>
<td>14</td>
<td>(10^{14})</td>
<td>3.5 years</td>
<td>99 Pb</td>
</tr>
<tr>
<td>16</td>
<td>(10^{16})</td>
<td>350 years</td>
<td>10 Eb</td>
</tr>
</tbody>
</table>
Implementation of Depth-first-search

- Depth-first-search, OPEN is handled as a pile (LIFO):

  FUNCTION DEPTH-FIRST-SEARCH()
  1. Let OPEN be the "pile" formed by just the initial node (i.e.
      the node whose state is *INITIAL-STATE*);
      Let CLOSED be initially empty
  2. While OPEN is not empty,
     2.1 Let CURRENT be the first node in OPEN
     2.2 Let OPEN be the rest of OPEN
     2.3 Add the node CURRENT to CLOSED.
     2.4 If IS-FINAL-STATE(STATE(CURRENT)),
           2.4.1 return the node CURRENT and halt.
           2.4.2 otherwise,
           2.4.2.1 Let NEW-SUCCESSORS be the list of nodes
                   from SUCCESSORS(CURRENT) whose state is neither
                   in OPEN nor in CLOSED
           2.4.2.2 Let OPEN be equal to the result of appending
                   NEW-SUCCESSORS at the beginning of OPEN
  3. Return FAIL.
Search tree for Depth-first-search

Water jug problem
Search tree for Depth-first-search

Water jug problem
Search tree for Depth-first-search

Water jug problem

(4,0) -> (0,0) -> (0,3)
(4,0) -> (1,3) -> (1,0)
(4,0) -> (4,1) -> (2,3)
Search tree for Depth-first-search

Water jug problem

(0,0) → (4,0) → (4,3) → (4,1) → (0,1) → (0,0)

(0,0) → (1,3) → (1,0) → (4,1) → (2,3) → (0,3)
Search tree for Depth-first-search

Water jug problem
Search tree for Depth-first-search

Water jug problem
Search tree for Depth-first-search

Water jug problem
Search tree for Depth-first-search

Water jug problem
## Table for Depth-first-search

**Water jug problem**

<table>
<thead>
<tr>
<th>Node</th>
<th>Current</th>
<th>Successors</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0 0)</td>
<td>((4 0) (0 3))</td>
<td>((0 0))</td>
</tr>
<tr>
<td>2</td>
<td>(4 0)</td>
<td>((4 3) (1 3))</td>
<td>((4 0) (0 3))</td>
</tr>
<tr>
<td>3</td>
<td>(4 3)</td>
<td>()</td>
<td>((1 3) (0 3))</td>
</tr>
<tr>
<td>4</td>
<td>(1 3)</td>
<td>((1 0))</td>
<td>((1 0) (0 3))</td>
</tr>
<tr>
<td>5</td>
<td>(1 0)</td>
<td>((0 1))</td>
<td>((0 1) (0 3))</td>
</tr>
<tr>
<td>6</td>
<td>(0 1)</td>
<td>((4 1))</td>
<td>((4 1) (0 3))</td>
</tr>
<tr>
<td>7</td>
<td>(4 1)</td>
<td>((2 3))</td>
<td>((2 3) (0 3))</td>
</tr>
<tr>
<td>8</td>
<td>(2 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Properties of Depth-first search

- Not complete (it may not halt)
  - Completeness for finite spaces
- When it halts, the output is not necessarily a minimal solution.
- Time complexity $O(b^m)$, where:
  - $b$: branching factor.
  - $m$: max depth during the search.
- Space complexity:
  - for the implementation shown before (where the list CLOSED is used), $O(b^m)$
Observations about the space complexity

• For the implementation shown before:
  • Max size of the list OPEN is $O(r \cdot m)$
  • The exponential complexity is caused by the list CLOSED

• Detection of cycles in the search:
  • For some state spaces it is not possible to get cyclic paths (and hence the list CLOSED is not needed to avoid cycles)
  • In general, in order to detect a cycle it suffices to store the nodes of the branch that is currently being explored
  • Therefore, it is easy to implement a depth-first-search that is still able to detect cycles, but decreasing the space complexity to $O(r \cdot m)$

• Thus, in general we shall consider that the space complexity of depth-first-search is $O(r \cdot m)$
Depth-limited search

- An attempt to avoid the incompleteness of depth-first search.
- Idea: not exploring paths beyond a given length limit.
- Problem: may still not be complete, e.g. if the limit is lower than the length of the shortest solution.
- However, in many cases an appropriate limit can be estimated in advance.
Implementation of Depth-limited search

FUNCTION DEPTH-LIMITED-SEARCH(LIMIT)
1. Let OPEN be the "pile" formed by just the initial node (i.e. the node whose state is *INITIAL-STATE*);
   Let CLOSED be initially empty
2. While OPEN is not empty,
   2.1 Let CURRENT be the first node in OPEN
   2.2 Let OPEN be the rest of OPEN
   2.3 Add the node CURRENT to CLOSED.
   2.4 If IS-FINAL-STATE(STATE(CURRENT)),
      2.4.1 return the node CURRENT and halt.
      2.4.2 otherwise, if DEPTH(CURRENT) < LIMIT,
      2.4.2.1 Let NEW-SUCCESSORS be the list of nodes from SUCCESSORS(CURRENT) whose state is neither in OPEN nor in CLOSED
      2.4.2.2 Let OPEN be equal to the result of appending NEW-SUCCESSORS at the beginning of OPEN
3. Return FAIL.
Properties of Depth-limited search

- Time complexity $O(b^l)$, where:
  - $b$: branching factor
  - $l$: depth limit
- Space complexity: $O(rl)$
- Not complete, in general
  - When the limit is too small
  - Even if the limit is greater than the length of a solution, the implementation shown may not find any solution, because of CLOSED
  - The modified version not using CLOSED is complete when the limit is greater than the length of a solution
- Not always finds a minimal solution
Iterative deepening depth-first search

- When a bound on the length of the solution is not known, an option to avoid incompleteness is to run several limited depth-first searches, gradually increasing the limit.
- Implementation of Iterative deepening search

FUNCTION ITERATIVE-DEEPENING-SEARCH(INITIAL-LIMIT)
1. Let N=INITIAL-LIMIT
2. If DEPTH-LIMITED-SEARCH(N) does not return FAIL,
   2.1 return the obtained result and halt.
   2.2 otherwise, set N to N+1 and goto step 2.
Search tree for Iterative deepening search

Water jug problem

- Initial state: (0 0)
- Goal state: (0 0) with combinations 1, 2, 5, 11, 19, 29, 41
- Successor states:
  - (4 0) with combinations 3, 6, 12, 20, 30, 42
  - (0 3) with combinations 4, 9, 16, 25, 36
  - (4 3) with combinations 7, 13, 21, 31, 43
  - (1 3) with combinations 8, 14, 22, 32, 44
  - (3 0) with combinations 10, 17, 26, 37
  - (1 0) with combinations 15, 23, 33, 45
  - (3 3) with combinations 18, 27, 38
  - (0 1) with combinations 24, 34, 46
  - (4 2) with combinations 28, 39
  - (4 1) with combinations 35, 47
  - (0 2) with combinations 40
  - (2 3) with combinations 48
  - (2 0)
Properties of Iterative deepening search

- Time complexity $O(b^d)$, where:
  - $b$: branching factor.
  - $d$: depth of the shortest solution.
- Space complexity $O(bd)$.
- Completeness.
- Returns a solution with the minimum number of actions (in the version where CLOSED is not used)
Properties of Iterative deepening search

- Iterative deepening search is the preferred uninformed search for huge search spaces and there’s no clue about depth of solution.
- Redundancy in expansion is little price for completeness.
  - Moreover, such redundancy is not significant.

Example, \( b = 10 \) and \( d = 5 \):

- Number of nodes analyzed (Depth-limited):
  \[ 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111 \]
- Number of nodes analyzed (Iterative deepening):
  \[ 1 + 11 + 111 + 111 + 111 + 111,111 = 123,456 \]
- Only 10% increase.
- Reason: the majority of the nodes belong to the lowest level of the tree.
## Comparison of uninformed search methods

<table>
<thead>
<tr>
<th></th>
<th>Breadth-first</th>
<th>Depth-first</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>$O(b^d)$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$O(b^d)$</td>
<td>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td><strong>Complete?</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes, if $l \geq d$</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Minimal?</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- $b$: branching factor
- $d$: depth of the shortest solution
- $m$: max depth during the search
- $l$: depth limit
Informed search

- Blind or uninformed search (Breadth-first, Depth-first,...): have no knowledge concerning how to reach the goal.
- Informed search: apply *knowledge* to the search process to make it more efficient.
- The knowledge will be given through a function *estimating* the “kindness” of states:
  - Give preference to better states.
  - Sort the pile OPEN, according to their estimated kindness.
  - Purpose: to reduce the search tree, gaining *efficiency* in practice.
Concept of heuristic

- **Heuristic:**
  - from Greek *heuriskein*, to discover: *Eureka!*
  - Merriam-Webster Dictionary: “involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods”.
  - Informally: method for solving problems that may not guarantee the solution, but works fine in general.
  - In our case, a heuristic will be a numerical function defined over states.

- **Heuristic function, `heuristic(state)`:**
  - Estimates the “distance” to the goal.
  - Always greater or equal than 0.
  - Value on final states: 0.
  - Value $\infty$ is allowed.

- All the specific knowledge about the problem that will be used is encoded in the heuristic function.
Recall: the trip problem

- To go from Sevilla to Almería, using a sequence of connections between neighbouring capitals
- 8 possible states:
  - Almería, Cádiz, Córdoba, Granada, Huelva, Jaén, Málaga, Sevilla
- Initial State: Sevilla.
- Final State: Almería.
- Actions:
  - Go to Almería, Go to Cádiz, Go to Córdoba, Go to Granada, Go to Huelva, Go to Jaén, Go to Málaga, Go to Sevilla.
An heuristic for the trip problem

• Coordinates:

Almeria : (409.5, 93 )
Granada : (309 ,127.5)
Malaga : (232.5, 75 )
Cadiz : ( 63 , 57 )
Huelva : ( 3 ,139.5)
Sevilla : ( 90 ,153 )
Cordoba : (198 ,207 )
Jaen : (295.5,192 )

• Heuristic function (Euclidean distance):

\[
\text{heuristic(state)} = \text{distance(coordinates(state), coordinates(almeria))}
\]
Recall: 8-puzzle problem

- A 3x3 board with 8 numbered tiles on it (thus leaving a blank free space of the same size of a tile). A tile adjacent to the blank space can slide into it. The game consists on transforming the initial position into the final one by means of a sequence of tile slidings. In particular, let us consider the following initial and final states:

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

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

- States: every possible board configuration
- Actions: up, down, left, right (movements of the space)
First heuristic for the 8-puzzle problem

- 8-puzzle problem (first heuristic):
  - \text{heuristic}(\text{state}): \text{number of pieces out of their correct positions (w.r.t. final state)}

- Example:

\begin{tabular}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 5 & \\
\end{tabular}

\textbf{H = 4}

\begin{tabular}{ccc}
1 & 2 & 3 \\
8 & 4 & \\
7 & 6 & 5 \\
\end{tabular}

\textbf{H = 0}
Second heuristic for the 8-puzzle problem

- 8-puzzle problem (second heuristic):
  - \text{heuristic(state)}: sum of the Manhattan distances of each tile to its position in the final state

- Example:

\begin{align*}
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 5 & \\
\end{array} & \quad \begin{array}{ccc}
1 & 2 & 3 \\
8 & 4 & \\
7 & 6 & 5 \\
\end{array} \\
H = 5 & \quad H = 0
\end{align*}
Idea of the Greedy best-first search

- **Best-first search:**
  - Prioritize the analysis of nodes with lower heuristic.
  - Sort the open queue w.r.t. heuristic, from lower to greater
  - The expanded node is always the one that is *estimated* to be “closer” to the goal

- The performance strongly depends on the quality of the heuristic used.
Implementation of the Greedy best-first search

FUNCTION BEST-FIRST-SEARCH()
1. Let OPEN be the "pile" formed by just the initial node (i.e. the node whose state is *INITIAL-STATE*);
   Let CLOSED be initially empty
2. While OPEN is not empty,
   2.1 Let CURRENT be the first node in OPEN
   2.2 Let OPEN be the rest of OPEN
   2.3 Add the node CURRENT to CLOSED.
   2.4 If IS-FINAL-STATE(STATE(CURRENT)),
      2.4.1 return the node CURRENT and halt.
      2.4.2 otherwise,
         2.4.2.1 Let NEW-SUCCESSORS be the list of nodes from SUCCESSORS(CURRENT) whose state is neither in OPEN nor in CLOSED
         2.4.2.2 Let OPEN be equal to the result of appending NEW-SUCCESSORS to OPEN, and sort the result in increasing order of heuristic value
3. Return FAIL.
Search tree for Best-first search

The trip problem

Sevilla

H = 325.08

Cádiz
H = 348.37

Córdoba
H = 240.27

Huelva
H = 409.15

Málaga
H = 177.91

Granada
H = 106.26

Almería
H = 0

Jaén
H = 150.99
Search tree for Best-first search

8-puzzle problem (1st heuristic)
Search tree for Best-first search

8-puzzle problem (2nd heuristic)
Properties of the Best-first search

- Time complexity $O(b^m)$, where:
  - $b$: branching factor.
  - $m$: max depth during the search.
- Space complexity: $O(b^m)$.
- In practice, the time and space required depend on the instance of the problem being solved and the quality of the chosen heuristic.
- Not completeness, in general.
  - For example, a bad heuristic could lead to an infinite branch.
- Not minimal (not necessarily finds a minimal solution).
  - The heuristic could guide the search towards a non-minimal solution.
State spaces with cost

- In some problems, there exists a (positive) cost associated to the application of actions.
  - \( \text{cost-of-applying-action}(\text{state}, \text{action}) \).
  - Assuming action is applicable on state.

- Cost associated to a path:
  - Sum of the costs of applying each one of its actions.
  - An *optimal solution* is a solution having the minimal cost.

- In some problems, there is not an explicit cost
  - In such case, \( \text{cost-of-applying-action}(\text{state}, \text{action})=1 \), and the optimal solutions coincide with the minimal ones.

- Cost in the trip problem: Euclidean distance between state and apply(action, state)
Idea of the optimal search

- Analyze on the first place nodes with the lower costs.
- That is, sort the OPEN queue by increasing cost.
- In this way, when the algorithm reaches a state for the first time, the associated cost is the lowest possible cost.
  - In particular, the first time that a final state is reached, we have found an optimal solution.
- This is a *blind* or *uninformed search*:
  - Does not use knowledge as a guide towards the goal
  - Particular case: breadth-first search.
Implementation of the optimal search

- Search node: add a field for “cost of the path”
  - Access function: PATH-COST(node)
- Successors of a node with cost:

FUNCTION SUCCESSOR(NODE, ACTION)
1. Let SUCCESSOR-STATE be equal to APPLY(ACTION, STATE(NODE))
2. Return a node whose state is SUCCESSOR-STATE, whose parent is NODE, whose action is ACTION, whose depth is DEPTH(NODE)+1, and whose cost is PATH-COST(NODE) added to COST-OF-APPLYING-ACTION(STATE(NODE), ACTION)

FUNCTION SUCCESSORS(NODE)
1. Let SUCCESSORS be initially empty
1. For each ACTION in ACTIONS(STATE(NODE)), include SUCCESSOR(NODE, ACTION) into SUCCESSORS
3. Return SUCCESSORS
FUNCTION OPTIMAL-SEARCH()

1. Let OPEN be the "queue" formed by just the initial node (i.e. the node whose state is *INITIAL-STATE*, empty path, and cost 0);
   Let CLOSED be initially empty
2. While OPEN is not empty,
   2.1 Let CURRENT be the first node in OPEN
       Let OPEN be the rest of OPEN
   2.2 Add the node CURRENT to CLOSED.
   2.3 If IS-FINAL-STATE(STATE(CURRENT)),
       2.3.1 return the node CURRENT and halt.
       2.3.2 otherwise,
           2.3.2.1 Let NEW-SUCCESSORS be the list of nodes
                    from SUCCESSORS(CURRENT) whose state is neither
                    in OPEN nor in CLOSED, or else their cost is
                    lower than any other previous node with the same
                    state.
           2.3.2.2 Let OPEN be equal to the result of appending
                    NEW-SUCCESSORS to OPEN, and sort the result in
                    increasing order of cost of the paths in the nodes.
3. Return FAIL.
Search tree for optimal search

The trip problem
Properties of optimal search

- Completeness (provided that every individual cost is greater than some $\epsilon > 0$)
- Space and Time complexity: $O(r^{1+[C/\epsilon]})$, where $C$ is the cost of an optimal solution.
  - Exponential (it is a generalization of breadth-first search)
- Always finds optimal solution (if exists).
- Except for very small state spaces, in practice this search is not feasible, due to the enormous space and time required.
Use of heuristics for searching optimal solutions

- Can we use best-first search to accelerate the search and still keep guaranteeing optimal solutions? In general, NO.

- Example 1:

  - Solution found by best-first search: I-A-C-E-G (suboptimal)
  - Cause: the heuristic overestimates the real cost
Use of heuristics for searching optimal solutions

- Example 2:

- Solution found by best-first search: I-B-D-G (suboptimal)
- Cause: we have ignored *the costs* of the current path
Idea of the A* search

- Objective of the A* search:
  - obtain good solutions (optimal).
  - improve efficiency (reducing the search tree).
- Idea: each node $n$ gets assigned a value $f(n) = g(n) + h(n)$
  - $g(n)$: cost of the path leading to $n$
  - $h(n)$: heuristic of the node, estimation of the cost of an optimal path from $n$ up to a final state
  - $f(n)$: estimation of the total cost of an optimal solution passing through $n$
- Always select the node having the lowest $f$
  - sorting the OPEN queue in increasing order w.r.t. $f$
Implementation of the A* search

FUNCTION A-STAR-SEARCH()
1. Let OPEN be the "queue" formed by just the initial node
   (i.e. the node whose state is *INITIAL-STATE*, empty path,
   cost 0, and cost-plus-heuristic HEURISTIC(*INITIAL-STATE*));
   Let CLOSED be initially empty
2. While OPEN is not empty,
   2.1 Let CURRENT be the first node in OPEN
      Let OPEN be the rest of OPEN
   2.2 Add the node CURRENT to CLOSED.
2.3 If IS-FINAL-STATE(STATE(CURRENT)),
   2.3.1 return the node CURRENT and halt.
   2.3.2 otherwise,
      2.3.2.1 Let NEW-SUCCESSORS be the list of nodes
               from SUCCESSORS(CURRENT) whose state is neither
               in OPEN nor in CLOSED, or else their cost is lower
               than any other previous node with the same state
      2.3.2.2 Let OPEN be equal to the result of appending
               NEW-SUCCESSORS to OPEN, and sort the result in
               increasing order of cost+heuristic of the nodes
3. Return FAIL.
Search tree for A* search

The trip problem

- **Sevilla**
  - **Cádiz**: C+H = 448.09
  - **Córdoba**: C+H = 361.01
  - **Jaén**: C+H = 370.38
- **Huelva**: C+H = 497.19
  - **Granada**: C+H = 361.49
  - **Almería**: C+H = 361.49
- **Málaga**: C+H = 340.36
Properties of A*

- Let $h^*(n)$ be the cost of an optimal path from STATE($n$) until a final state
  - $f^*(n) = g(n) + h^*(n)$ is the total cost of an optimal solution passing through $n$
- In practice, we cannot know $h^*$
- We use a heuristic function $h$ that estimates $h^*$
- Possibilities:
  - For every node $n$, $h(n) = 0$: optimal search, no reduction on the search tree at all
  - For every node $n$, $h(n) = h^*(n)$: perfect estimation, there is no search branching
  - For every node $n$, $0 \leq h(n) \leq h^*(n)$: admissible heuristic, optimal solution guaranteed
  - For at least one node $n$, $h(n) > h^*(n)$: we cannot guarantee that the solution to be found is optimal
Properties of A*

- When using an admissible heuristic:
  - It is complete.
  - Always finds an optimal solution.
- Space and Time complexity: in the worst case, just like the optimal search
- In practice, the actual time and space required depend on the particular problem and the quality of the heuristic used
Proof of A* being optimal

Let us suppose that the A* algorithm uses an admissible heuristic $h$, and let us prove that a node $m$ belonging to a non-optimal solution can never be the first node of the OPEN queue.

Let $p$ be a node corresponding to an optimal solution. It can be easily proved that there must exist a node $n$ in the OPEN queue whose path is a subpath from the path of the solution $p$. It is sufficient now to prove that $n$ is located before $m$ in the OPEN queue (i.e., prove that $f(n) < f(m)$).

By definition, $f(n) = g(n) + h(n)$. Since $h$ is admissible, $h(n) \leq h^*(n)$. On the other hand, since $n$ and $p$ belong to the same path to the optimal solution, by definition of $g$ and $h^*$, we have $g(n) + h^*(n) = g(p)$. Therefore, we deduce $f(n) \leq g(p)$. Taking into account that $m$ is suboptimal and $h(m) = 0$ (because $m$ is final), we conclude that $g(p) < g(m) = f(m)$. Thus, we have proved that $f(n) < f(m)$ holds.
Inventing heuristics

- Recall: estimation of the minimum remaining cost until reaching a final state
- Comparing heuristics:
  - If for every node \( n \), \( 0 \leq h_1(n) \leq h_2(n) \leq h^*(n) \), then \( h_2 \) is more informed than \( h_1 \) and improves the efficiency
- Techniques to find heuristics
  - Relaxation of the problem
  - Combination of admissible heuristics
  - Use of statistical information
- Efficient evaluation of the heuristic function
Maze problem

- In the following maze, we can move from a cell to an adjacent one (up, down, left, right), except for the cases where there is a wall between them.

- Objective: to go from I to F

- As a final exercise, we will solve the problem by applying depth-first, best-first, and A* search algorithms.
Maze problem as a state space problem

- Initial state: I
- Final state: F
- Actions: up, down, left, right
- Applicability and result of the application
  - “Up” is applicable on a state if there exist a cell upwards (according to the map) and there is no wall in this direction; the result of applying the action is the upper cell, and its cost is 1
  - Analogously for the rest of actions
  - Consider that the list of successors is ordered alphabetically
- Heuristic (admissible): “Manhattan distance from the state to cell F”
- During the search, in case of equal heuristic value, select nodes in alphabetical order
Search tree for Depth-first-search

Maze problem

- Solution found: I-Q-R-T-K-M-F
  - Not optimal
  - Nodes analyzed: 13
Search tree for Best-first search

Maze problem

- Solution found: I-Q-R-T-K-M-F
  - Not optimal
  - Nodes analyzed: 7
  - The heuristic has been useful for reducing the search space
• Solution found: I-W-K-M-F
  • Optimal (admissible heuristic)
  • Nodes analyzed: 8
  • Although in this case the number of analyzed nodes increases (a wrong path has been partially explored), the optimal solution is guaranteed
  • Note that two different nodes have been generated having the same state K (but having different paths)
Bibliography I

  - Ch. 3: “Solving problems by searching”.

  - Ch. 3: “Structures and strategies for state space search”
  - Ch. 4: “Heuristic search”
Bibliography II

  • Ch. 7: “Uninformed Search”
  • Ch. 8: “Heuristic Search”

• Poole, D.; Mackworth, A. and Goebel, R. Computational Intelligence (A Logical Approach) (Oxford University Press, 1998).
  • Ch. 4: “Searching”