Example: Romania

- **Problem**: On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest. Find a short route to drive to Bucharest.

- **Formulate problem**:
  - **states**: various cities
  - **actions**: drive between cities
  - **solution**: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem types

- Deterministic, fully observable $\implies$ single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence

- Non-observable $\implies$ conformant problem
  - Agent may have no idea where it is; solution (if any) is a sequence

- Nondeterministic and/or partially observable $\implies$ contingency problem
  - percepts provide new information about current state
  - solution is a tree or policy
  - often interleave search, execution

- Unknown state space $\implies$ exploration problem ("online")
Problem Solving

We will start by considering the simpler cases in which the following holds.

- The agent’s world (environment) is representable by a discrete set of states.
- The agent’s actions are representable by a discrete set of operators.
- The world is static and deterministic.

Example: Vacuum World

Single-state, start in #5.
Solution??

Conformant, start in
{1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}.
Solution??
[Right, Suck, Left, Suck]

Contingency, start in #5

Example: Vacuum World

Single-state, start in #5.
Solution?? [Right, Suck]

Conformant, start in
{1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}.
Solution??
[Right, Suck]
**Single-state problem formulation**

- A **problem** is defined by four items:
  - *initial state* e.g., “at Arad”
  - *successor function* $S(x)$ = set of action–state pairs e.g., $S(\text{Arad}) = \{\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}\}, \ldots$
  - *goal test*, can be **explicit**, e.g., $x = \text{“at Bucharest”}$ **implicit**, e.g., $\text{NoDirt}(x)$
  - *path cost* (additive) e.g., sum of distances, number of actions executed, etc. Usually given as $c(x, a, y)$, the *step cost* from $x$ to $y$ by action $a$, assumed to be $\geq 0$.

- A **solution** is a sequence of actions leading from the initial state to a goal state

---

**Example: Vacuum World**

**Single-state**, start in #5.

**Solution??** [Right, Suck]

**Conformant**, start in

{1, 2, 3, 4, 5, 6, 7, 8}

**Solution??**

[Right, Suck, Left, Suck]

**Contingency**, start in #5

---

**State space graph of vacuum world**

![State space graph](image)

**Selecting a State Space**

- Real world is absurdly complex ⇒ state space must be **abstracted** for problem solving

- (Abstract) state = set of real states

- (Abstract) action = complex combination of real actions e.g., “Arad → Zerind” represents a complex set of possible routes, detours, rest stops, etc.

  - For guaranteed realizability, **any** real state “in Arad” must get to **some** real state “in Zerind”.

  - Each abstract action should be “easier” than the original problem!

- (Abstract) solution = set of real paths that are solutions in the real world
Formulating Problem as a Graph

In the graph
- each node represents a possible state;
- a node is designated as the initial state;
- one or more nodes represent goal states, states in which the agent's goal is considered accomplished.
- each edge represents a state transition caused by a specific agent action;
- associated to each edge is the cost of performing that transition.

State space graph of vacuum world

- states??: integer dirt and robot locations (ignore dirt amounts)
- actions??: Left, Right, Suck, NoOp
- goal test??: no dirt
- path cost??: 1 per action (0 for NoOp)

Problem Solving as Search

Search space: set of states reachable from an initial state \( S_0 \) via a (possibly empty/finite/infinite) sequence of state transitions.

To achieve the problem's goal
- search the space for a (possibly optimal) sequence of transitions starting from \( S_0 \) and leading to a goal state;
- execute (in order) the actions associated to each transition in the identified sequence.

Depending on the features of the agent's world the two steps above can be interleaved.

Search Graph

How do we reach a goal state?

There may be several possible ways. Or none!

Factors to consider:
- cost of finding a path;
- cost of traversing a path.
**Problem Solving as Search**

- Reduce the original problem to a search problem.
- A solution for the search problem is a path initial state–goal state.
- The solution for the original problem is either
  - the sequence of actions associated with the path or
  - the description of the goal state.

**Example: The 8-puzzle**

It can be generalized to 15-puzzle, 24-puzzle, or \((n^2 - 1)\)-puzzle for \(n \geq 6\).

**States:** configurations of tiles

**Operators:** move one tile Up/Down/Left/Right

- There are 9! = 362,880 possible states (all permutations of \(\{\square, 1, 2, 3, 4, 5, 6, 7, 8\}\)).
- There are 16! possible states for 15-puzzle.
- Not all states are directly reachable from a given state. (In fact, exactly half of them are reachable from a given state.)

How can an artificial agent represent the states and the state space for this problem?
Formulating the 8-puzzle Problem

States: each represented by a 3 × 3 array of numbers in [0...8], where value 0 is for the empty cell.

States:  

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

becomes  

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5
\end{array}
\]

Preconditions and Effects

Example: \( Op_{(3,2,R)} \)

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5
\end{array} \xrightarrow{Op_{(3,2,R)}} \begin{array}{ccc}
2 & 8 & 3 \\
\text{Preconditions: } A[3, 2] = 0 \\
\text{Effects: } \\
A[3, 3] \leftarrow \text{Effects}
\end{array}
\]

We have 24 operators in this problem formulation . . .  20 too many!

Problem Formulation

1. Choose an appropriate data structure to represent the world states.
2. Define each operator as a precondition/effects pair where the
   - precondition holds exactly in the states the operator applies to,
   - effects describe how a state changes into a successor state by the application of the operator.
3. Specify an initial state.
4. Provide a description of the goal (used to check if a reached state is a goal state).

Formulating the 8-puzzle Problem

Operators: 24 operators of the form \( Op_{(r,c,d)} \)

where \( r, c \in \{1, 2, 3\}, \ d \in \{L, R, U, D\} \).

\( Op_{(r,c,d)} \) moves the empty space at position \((r, c)\) in the direction \(d\).

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5
\end{array} \xrightarrow{Op_{(3,2,L)}} \begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5
\end{array}
\]
A Better Formulation

Operators: 4 operators of the form \( O_{d} \) where 
\( d \in \{L, R, U, D\} \).

\( O_{d} \) moves the empty space in the direction \( d \).

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5 \\
\end{array}
\]

\( \xrightarrow{O_L} \)

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5 \\
\end{array}
\]

States: each represented by a pair \((A, (i, j))\) where:
- \( A \) is a \( 3 \times 3 \) array of numbers in \([0 \ldots 8]\)
- \((i, j)\) is the position of the empty space \((0)\) in the array.

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

becomes

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5 \\
\end{array}
\]

Preconditions and Effects

Example: \( O_{L} \)

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5 \\
\end{array}
\]

\( \xrightarrow{O_L} \)

\[
\begin{array}{ccc}
2 & 8 & 3 \\
1 & 6 & 4 \\
7 & 0 & 5 \\
\end{array}
\]

Let \((r_0, c_0)\) be the position of 0 in \( A \).

Preconditions: \( c_0 > 1 \)

Effects:

\[
\begin{align*}
A[r_0, c_0] & \leftarrow A[r_0, c_0 - 1] \\
A[r_0, c_0 - 1] & \leftarrow 0 \\
(r_0, c_0) & \leftarrow (r_0, c_0 - 1)
\end{align*}
\]

Half states are not reachable?

Can this be done?

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

\( \xrightarrow{\text{any steps}} \)

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

$1,000 award for anyone who can do it!
The Water Jugs Problem

Get exactly 2 gallons of water into the 4gl jug.

The Water Jugs Problem: Operators

F4: fill jug4 from the pump.
  \[ \text{precond: } J_4 < 4 \quad \text{effect: } J'_4 = 4 \]

E4: empty jug4 on the ground.
  \[ \text{precond: } J_4 > 0 \quad \text{effect: } J'_4 = 0 \]

E4-3: pour water from jug4 into jug3 until jug3 is full.
  \[ \text{precond: } J_3 < 3, \quad J_4 \geq J_3 \quad \text{effect: } J'_3 = J_3 - (3 - J_3), \quad J'_4 = J_4 - (3 - J_3) \]

P3-4: pour water from jug3 into jug4 until jug4 is full.
  \[ \text{precond: } J_4 < 4, \quad J_3 \geq J_4 \quad \text{effect: } J'_4 = J_4 - (4 - J_4), \quad J'_3 = J_3 - (4 - J_4) \]

E3-4: pour water from jug3 into jug4 until jug3 is empty.
  \[ \text{precond: } J_3 + J_4 < 4, \quad J_3 > 0 \quad \text{effect: } J'_3 = J_3 + J_4, \quad J'_4 = 0 \]

... Natalia’s notes...

Half states are not reachable?

Let the 8-puzzle be represented by \((a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)\). We say \((a_i, a_j)\) is an inversion if neither \(a_i\) nor \(a_j\) is blank, \(i < j\) and \(a_i > a_j\).

The first one has 0 inversions and the second has 1.

Claim: # of inversions modulo two remains the same after each move.

The Water Jugs Problem

States: Determined by the amount of water in each jug.

State Representation: Two real-valued variables, \(J_3, J_4\), indicating the amount of water in the two jugs, with the constraints:

\[ 0 \leq J_3 \leq 3, \quad 0 \leq J_4 \leq 4 \]

Initial State Description

\[ J_3 = 0, \quad J_4 = 0 \]

Goal State Description:

\[ J_4 = 2 \quad \Leftarrow \text{non exhaustive description} \]
Real-World Search Problems

- Route Finding
  (computer networks, airline travel planning system, …)
- Travelling Salesman Optimization Problem
  (package delivery, automatic drills, …)
- Layout Problems
  (VLSI layout, furniture layout, packaging, …)
- Assembly Sequencing
  (assembly of electric motors, …)
- Task Scheduling
  (manufacturing, timetables, …)

The Water Jugs Problem

<table>
<thead>
<tr>
<th>Problem</th>
<th>Search Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₃ = 0</td>
<td>J₄ = 0</td>
</tr>
<tr>
<td>J₃ = 0</td>
<td>J₄ = 4</td>
</tr>
<tr>
<td>J₃ = 0</td>
<td>J₄ = 3</td>
</tr>
<tr>
<td>J₃ = 3</td>
<td>J₄ = 0</td>
</tr>
<tr>
<td>J₃ = 3</td>
<td>J₄ = 4</td>
</tr>
<tr>
<td>J₃ = 3</td>
<td>J₄ = 1</td>
</tr>
</tbody>
</table>

More on Graphs

A graph is a set of notes and edges (arcs) between them.

A graph is directed if an edge can be traversed only in a specified direction.

When an edge is directed from \( n_i \) to \( n_j \)
- it is univocally identified by the pair \( (n_i, n_j) \)
- \( n_i \) is a parent (or predecessor) of \( n_j \)
- \( n_j \) is a child (or successor) of \( n_i \)

Problem Solution

- Problems whose solution is a description of how to reach a goal state from the initial state:
  - \( n \)-puzzle
  - route-finding problem
  - assembly sequencing
- Problems whose solution is simply a description of the goal state itself:
  - 8-queen problem
  - scheduling problems
  - layout problems
Directed Graphs

A path, of length \( k \geq 0 \), is a sequence \( \langle (n_1, n_2), (n_2, n_3), \ldots, (n_k, n_{k+1}) \rangle \) of \( k \) successive edges. Ex: \( \langle \rangle, \langle (S, D) \rangle, \langle (S, D), (D, E) \rangle, \langle (E, B) \rangle \)

For \( 1 \leq i < j \leq k + 1 \),
- \( N_i \) is a \textit{ancestor} of \( N_j \); \( N_j \) is a \textit{descendant} of \( N_i \).

A graph is \textit{cyclic} if it has a path starting from and ending into the same node. Ex: \( \langle (A, D), (D, E), (E, A) \rangle \)

\(^a\) Note that a path of length \( k > 0 \) contains \( k + 1 \) nodes.

Tree Search Algorithms

Basic idea: offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. \textit{expanding} states)

\[
\text{function TREE-SEARCH( problem, strategy) returns a solution, or failure}
\]
\[
\text{initialize the search tree using the initial state of problem}
\]
\[
\text{loop do}
\]
\[
\text{if there are no candidates for expansion then return failure}
\]
\[
\text{choose a leaf node for expansion according to strategy}
\]
\[
\text{if the node contains a goal state then return the solution}
\]
\[
\text{else expand the node and add the resulting nodes to the search tree}
\]

From Graphs to Trees

To unravel a graph into a tree choose a root node and trace every path from that node until you reach a leaf node or a node already in that path.

- must remember which nodes have been visited
- a node may get duplicated several times in the tree
- the tree has infinite paths only if the graph has infinite non-cyclic paths.
Implementation: states vs. nodes

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

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

The **EXPAND** function creates new nodes, filling in the various fields and using the **SUCCESSORFn** of the problem to create the corresponding states.
Uninformed Search Strategies

*Uninformed* strategies use only the information available in the problem definition

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

Search Strategies

- A strategy is defined by picking the *order of node expansion*. Strategies are evaluated along the following dimensions:
  - *completeness*—does it always find a solution if one exists?
  - *time complexity*—number of nodes generated/expanded
  - *space complexity*—maximum number of nodes in memory
  - *optimality*—does it always find a least-cost solution?

Time and space complexity are measured in terms of

- $b$—maximum branching factor of the search tree
- $d$—depth of the least-cost solution
- $m$—maximum depth of the state space (may be $\infty$)

Breadth-First Search

Expand shallowest unexpanded node

**Implementation:** fringe is a FIFO queue, i.e., new successors go at end

Artificial Intelligence – p.45/89

Breadth-First Search

Expand shallowest unexpanded node

**Implementation:** fringe is a FIFO queue, i.e., new successors go at end

Artificial Intelligence – p.47/89
Breadth-First Search

Expand shallowest unexpanded node

Implementation: fringe is a FIFO queue, i.e., new successors go at end

Properties of Breadth-First Search

Complete?? Yes (if $b$ is finite)

Time??

Complete??

Yes (if $b$ is finite)
Properties of Breadth-First Search

- **Complete:** Yes (if \( b \) is finite)
- **Time:** \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}), \) i.e.,
  exp. in \( d \)
- **Space:** \( O(b^{d+1}) \) (keeps every node in memory)
- **Optimal:** Yes (if cost = 1 per step); not optimal in general

It is the big problem; can easily generate nodes at 10MB/sec so 24hrs = 860GB.
**Uniform-Cost Search**

Expand least-cost unexpanded node

Implementation: fringe = queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost \( \geq \epsilon \)

Time?? \# of nodes with \( g \leq \) cost of optimal solution, \( O(b^{C^*/\epsilon}) \) where \( C^* \) is the cost of the optimal solution

Space?? \# of nodes with \( g \leq \) cost of optimal solution, \( O(b^{C^*/\epsilon}) \)

Optimal?? Yes—nodes expanded in increasing order of \( g(n) \)

---

**Depth-First Search**

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front

---

Depth-First Search

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front

---

**Depth-First Search**

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front

---

**Depth-First Search**

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front
Depth-First Search

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front
Depth-First Search

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front
Properties of depth-first search

Complete??

No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path ⇒
complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than breadth-first

Space??

Depth-First Search

Expand deepest unexpanded node

Implementation: fringe = LIFO queue, i.e., put successors at front

Properties of depth-first search

Complete??

No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path ⇒
complete in finite spaces

Time??
Properties of depth-first search

- **Complete??** No: fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along path ⇒ complete in finite spaces
- **Time??** $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than breadth-first
- **Space??** $O(bm)$, i.e., linear space!
- **Optimal??** No

Iterative Deepening Search

function Iterative-Deepening-Search (problem) return soln
  for depth from 0 to MAX-INT do
    result := Depth-Limited-Search(problem, depth)
    if (result != cutoff) then return result
  end for
end function

Depth-Limited Search

= depth-first search with depth limit $l$, i.e., nodes at depth $l$ have no successors

function Depth-Limited-Search (problem, limit) return soln/fail/cutoff
  return Recursive-DLS(Make-Node(Initial-State(problem)), problem, limit)
end function

function Recursive-DLS (node, problem, limit) return soln/fail/cutoff
  cutoff-occurred := false;
  if (Goal-State(problem, State(node))) then return node;
  else if (Depth(node) == limit) then return cutoff;
  else for each successor in Expand(node, problem) do
    result := Recursive-DLS(successor, problem, limit)
    if (result == cutoff) then cutoff-occurred := true;
    else if (result != fail) then return result;
  end for
  if (cutoff-occurred) then return cutoff; else return fail;
end function
Iterative deepening search $l = 0$

Iterative deepening search $l = 1$

Iterative deepening search $l = 2$

Iterative deepening search $l = 3$
Properties of iterative deepening search

- Complete: Yes
- Time: \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)
- Space: \(O(bd)\)
- Optimal: Yes

\[
(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)
\]
**Summary of Algorithms**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-Limited</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>Yes, if ( l \geq d )</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>( b^{d+1} )</td>
<td>( b^{C^*/\epsilon} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^{d+1} )</td>
<td>( b^{C^*/\epsilon} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Properties of iterative deepening search**

- **Complete??** Yes
- **Time??** \( (d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d) \)
- **Space??** \( O(bd) \)
- **Optimal??** Yes, if step cost = 1 Can be modified to explore uniform-cost tree

Numerical comparison for \( b = 10 \) and \( d = 5 \), solution at far right:

\[
N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\]

**Summary**

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

**Repeated states**

Failure to detect repeated states can turn a linear problem into an exponential one!

Artificial Intelligence – p.85/89
Example: Romania

For a given strategy, what is the order of nodes to be generated (or stored), and expanded?
With or without checking duplicated nodes?

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