Problem-solving agents

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
static: seq, an action sequence, initially empty
  state, some description of the current world state
  goal, a goal, initially null
  problem, a problem formulation
state — UPDATE-STATE(state, percept)
if seq is empty then
  goal — FORMULATE-GOAL(state)
  problem — FORMULATE-PROBLEM(state, goal)
  seq — SEARCH(problem)
action — FIRST(seq); seq — REST(seq)
return action

Note: this is offline problem solving; solution executed "eyes closed."
Online problem solving involves acting without complete knowledge.
Problems formulated in terms of atomic states

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:
be in Bucharest

Formulate problem:
states: various cities
actions: drive between cities

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

Assignment 0 due midnight Thursday 9/8
Assignment 1 posted, due 9/20 (online or in box in 283)
Problem types

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

Non-observable \(\implies\) sensorless problem (a.k.a. conformant)
- 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 contingent plan or a policy
- Often interleave search, execution

Unknown state space \(\implies\) exploration problem ("online")

Example: vacuum world

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

Sensorless, start in \(\{1, 2, 3, 4, 5, 6, 7, 8\}\)
- \(\text{e.g., } \text{Right goes to } \{2, 4, 6, 8\}\). Solution??

Contingency, start in #5
- Murphy’s Law: \(\text{Suck}\) can dirty a clean carpet
- Local sensing: dirt, location only.
- Solution??

Initial belief state is \(\{5, 7\}\)
- \(\text{Right, if dirt then Suck}\)

Example: vacuum world

Single-state problem formulation

A problem is defined by four items:
- initial state e.g., “at Arad”
- successor function \(S(x) = \text{set of action-state pairs}\)
  - \(\text{e.g., } S(\text{Arad}) = \{(\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}), \ldots\}\)
- 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.
  - \(c(x, a, y)\) is the step cost, assumed to be \(\geq 0\)

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

(Abstract) solution
  = sequence of abstract actions
  = set of real paths that are solutions in the real world
Each abstract action should be “easier” than the original problem!
Example: The 8-puzzle

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

**Start State**

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

**Goal State**

*states??*: integer locations of tiles (ignore intermediate positions)
*actions??*: move blank left, right, up, down (ignore unjamming etc.)
*goal test??*: = goal state (given)
*path cost??*: 1 per move

[Note: optimal solution of n-Puzzle family is NP-hard]

Example: robotic assembly

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

**Start State**

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

**Goal State**

*states??*: real-valued coordinates of robot joint angles and parts of the object to be assembled
*actions??*: continuous motions of robot joints
*goal test??*: complete assembly with no robot included!
*path cost??*: time to execute
Tree search algorithms

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

function Tree-Search(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
end

Tree search example

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.

Implementation: general tree search

function Tree-Search(problem, fringe) returns a solution, or failure
fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
loop do
    if fringe is empty then return failure
    node ← Remove-Front(fringe)
    if Goal-Test(problem, State[node]) then return Solution(node)
    fringe ← InsertAll(Expand(node, problem), fringe)
end

function Expand(node, problem) returns a set of nodes
successors ← the empty set; state ← State[node]
for each action, result in Successor-Fn(problem, state) do
    s ← a new Node
    Parent-Node[s] ← node; Action[s] ← action; State[s] ← result
    Path-Cost[s] ← Path-Cost[node] + Step-Cost(state, action, result)
    Depth[s] ← Depth[node] + 1
    add s to successors
return successors
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
- $C^*$: path cost of the least-cost solution
- $m$: maximum depth of the state space (may be $\infty$)

### 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

### Breadth-first search

Expand shallowest unexpanded node

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

- Expand shallowest unexpanded node
- **Implementation**: fringe is a FIFO queue, i.e., new successors go at end

- 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?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost, lowest first

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:
\[ fringe = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

$\textit{fringe} = \text{LIFO queue, i.e., put successors at front}$
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?? \( O(m) \), i.e., linear space!

Optimal?? No

Depth-limited search

= depth-first search with depth limit \( l \),
returns \( cutoff \) if any path is cut off by depth limit

Recursive implementation:

function \( \text{DEPTH-LIMITED-SEARCH}(\text{problem, limit}) \) returns \( \text{soln/fail/cutoff} \)

function \( \text{RECURSIVE-DLS}(\text{node, problem, limit}) \) returns \( \text{soln/fail/cutoff} \)
cutoff-occurred? \( \leftarrow \) false
if \( \text{GOAL-TEST}(\text{problem, State[node]}) \) then return node
else if \( \text{DEPTH}[\text{node}] = \text{limit} \) then return cutoff
derf for each \( \text{successor} \) in \( \text{EXPAND}(\text{node, problem}) \) do
if result = cutoff then cutoff-occurred? \( \leftarrow \) true
derf else if result \( \neq \) failure then return result
if cutoff-occurred? then return cutoff else return failure
Iterative deepening search

function Iterative-Deepening-Search(problem) returns a solution
inputs: problem, a problem
for depth = 0 to ∞ do
    result = Depth-Limited-Search(problem, depth)
    if result ≠ cutoff then return result
end

Properties of iterative deepening search

Complete??
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? No, unless step costs are constant

Can be modified to explore uniform-cost tree

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

\[
N(\text{IDS}) = 50 + 400 + 3000 + 20000 + 100000 = 123,450
\]

\[
N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 999,990 = 1,111,100
\]

IDS does better because other nodes at depth \(d\) are not expanded

BFS can be modified to apply goal test when a node is generated

Summary of algorithms

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

Repeated states

Failure to detect repeated states can cause exponentially more work!
Graph search

function \textsc{Graph-Search}(\texttt{problem}, \texttt{fringe}) returns a solution, or failure
\begin{center}
\texttt{closed} — an empty set
\texttt{fringe} — \textsc{Insert}(\textsc{Make-Node}(\texttt{Initial-State}(\texttt{problem})), \texttt{fringe})
loop do
    if \texttt{fringe} is empty then return failure
    \texttt{node} — \textsc{Remove-Front}(\texttt{fringe})
    if \texttt{Goal-Test}(\texttt{problem}, \texttt{State}(\texttt{node})) then return \texttt{node}
    if \texttt{State}(\texttt{node}) is not in \texttt{closed} then
        add \texttt{State}(\texttt{node}) to \texttt{closed}
        \texttt{fringe} — \textsc{InsertAll}(\textsc{Expand}(\texttt{node}, \texttt{problem}), \texttt{fringe})
end
\end{center}

Use hash table for \texttt{closed} — constant-time lookup!

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

Graph search can be exponentially more efficient than tree search