Problem solving and search

Chapter 3, Sections 1–5

Problem-solving agents

Restricted form of general agent:

function SIMPLE-PROBLEM-SOLVER-AGENT(p) returns an action
inputs: p, a percept

1. state: a an action sequence, initially empty
2. goal: a description of the current world state
3. problem: a problem formulation

a. if state is empty then
   if goal is achieved then
      return goal
   else
      return RECOMMENDATION(state, goal) = SEARCH PROBLEM(state, goal)

Note: this is off-line problem solving.

Online problem solving involves acting without complete knowledge of the problem and solution.

Example: Romania

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

Formulate goal:
be in Bucharest

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

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

Problem types

Deterministic, accessible \(\implies\) single-state problem
Deterministic, inaccessible \(\implies\) multiple-state problem

Non-deterministic, inaccessible \(\implies\) contingency problem
must use sensors during execution
solution is a tree or policy
often interleave search, execution

Unknown state space \(\implies\) exploration problem ("online")
Example: vacuum world

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

Multiple-state, start in {1, 2, 3, 4, 5, 6, 7, 8} e.g., Right goes to {2, 4, 6, 8}. Solution??

Contingency, start in #5

Murphy's Law: Suck can dirty a clean carpet

Local sensing: dirt, location only. Solution??

Example: The 8-puzzle

states??: integer locations of tiles (ignore intermediate positions)
operators??: 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]

Single-state problem formulation

A problem is defined by four items:

initial state e.g., "at Arad"
operators (or successor function \( S(x) \)) e.g., \( \text{Arad} \rightarrow \text{Zerind} \) \( \text{Arad} \rightarrow \text{Sibiu} \) etc.
goal test, can be explicit, e.g., \( x = \text{"at Bucharest"} \)
implicit, e.g., \( N_0 \text{Dirt}(x) \)
path cost (additive)
e.g., sum of distances, number of operators executed, etc.

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

Selecting a state space

Real world is absurdly complex

\( \Rightarrow \) state space must be abstracted for problem solving

(Abstract) state = set of real states

(Abstract) operator = complex combination of real actions e.g., "Arad \( \rightarrow \) 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 = set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!
Example: vacuum world state space graph

states??: integer dirt and robot locations (ignore dirt amounts)
operators??: Left, Right, Suck
goal test??: no dirt
path cost??: 1 per operator

Example: robotic assembly

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

Search algorithms

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

function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
begin
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

General search example

Arau

Zerind Sibiu Timisoara

Arad Oradea Rimnicu Vilcea Fagaras
Implementation of search algorithms

function GENERAL-SEARCH problem, QUEUE-FN returns a solution, or failure
  \begin{align}
  &\text{mark} = \text{MARK-QUEUE-MARK-NODE(INITIAL-STATE[problem])} \\
  \text{loop do} & \\
  &\text{if} \text{mark} \text{ is empty} \text{ then return failure} \\
  &\text{mark} = \text{REGENERATE-CHILDREN(marked)} \\
  &\text{if} \text{GOAL-TEST[problem] applied to STATE(mark) returns true} \text{ then return mark} \\
  &\text{mark} = \text{QUEUE-FN(marked, EXPAND(mark, OPERATOR[problem])]} \\
  \text{end}
\end{align}

Implementation contd: 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 OPERATORS (or 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:

- 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:

\[
\text{QUEUEFN} = \text{put successors at end of queue}
\]
Properties of breadth-first search

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

**Time??** $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in $d$

**Space??** $O(b^d)$ (keeps every node in memory)

**Optimal??** Yes (if cost = 1 per step); not optimal in general

*Space* is the big problem; can easily generate nodes at 1MB/sec
so 24hrs = 86GB.

---

**Romania with step costs in km**

<table>
<thead>
<tr>
<th>City</th>
<th>Bucharest</th>
<th>Craiova</th>
<th>Doljani</th>
<th>Fagaras</th>
<th>Giurgiu</th>
<th>Iasi</th>
<th>Neamt</th>
<th>Oradea</th>
<th>Pitesti</th>
<th>Sibiu</th>
<th>Timisoara</th>
<th>Vatra Dornei</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucharest</td>
<td>0</td>
<td>140</td>
<td>151</td>
<td>226</td>
<td>178</td>
<td>253</td>
<td>193</td>
<td>242</td>
<td>241</td>
<td>329</td>
<td>374</td>
<td></td>
</tr>
<tr>
<td>Craiova</td>
<td></td>
<td>366</td>
<td>101</td>
<td>92</td>
<td>85</td>
<td>97</td>
<td>80</td>
<td>98</td>
<td>99</td>
<td>92</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>Doljani</td>
<td></td>
<td></td>
<td>118</td>
<td>101</td>
<td>94</td>
<td>86</td>
<td>80</td>
<td>96</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Fagaras</td>
<td></td>
<td></td>
<td></td>
<td>151</td>
<td>92</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Giurgiu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>178</td>
<td>92</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Iasi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>253</td>
<td>92</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Neamt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Oradea</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Pitesti</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Sibiu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Timisoara</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Vatra Dornei</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Bucharest**

- 200 K
- 240 K
- 280 K
- 320 K
- 360 K

**Craiova**

- 150 K
- 200 K
- 250 K
- 300 K
- 350 K

**Doljani**

- 250 K
- 300 K
- 350 K
- 400 K

**Fagaras**

- 100 K
- 150 K
- 200 K
- 250 K

**Giurgiu**

- 350 K
- 400 K
- 450 K
- 500 K

**Iasi**

- 300 K
- 350 K
- 400 K
- 450 K

**Neamt**

- 500 K
- 550 K
- 600 K
- 650 K

**Oradea**

- 250 K
- 300 K
- 350 K
- 400 K

**Pitesti**

- 350 K
- 400 K
- 450 K
- 500 K

**Sibiu**

- 100 K
- 150 K
- 200 K
- 250 K

**Timisoara**

- 500 K
- 550 K
- 600 K
- 650 K

**Vatra Dornei**

- 550 K
- 600 K
- 650 K
- 700 K

---

**Final Thoughts**

- Breadth-first search is useful for finding the shortest path.
- It is complete and optimal if the cost is 1 per step.
- It can handle large problems with limited memory.

---

**Applications**

- Road networks
- Telecommunication networks
- Computer networks
- Air traffic control
-基因组学
**Uniform-cost search**

Expand least-cost unexpanded node

Implementation:

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

**Properties of uniform-cost search**

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

Time?? \# of nodes with \( g \leq \) cost of optimal solution

Space?? \# of nodes with \( g \leq \) cost of optimal solution

Optimal?? Yes

**Depth-first search**

Expand deepest unexpanded node

Implementation:

\[ \text{QUERINGFN} = \text{insert successors at front of queue} \]
DFS on a depth-3 binary tree

I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)
DFS on a depth-3 binary tree, contd.
### Properties of depth-first search

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

### Depth-limited search

= depth-first search with depth limit \(l\)

**Implementation:**

Nodes at depth \(l\) have no successors.

### Iterative deepening search

**Function:** `ITERATIVE-DEEPENING-SEARCH(problem)` returns a solution sequence.

**Inputs:** problem.

for \(depth = 0\) to \(\infty\) do
  \(result \leftarrow \) DEPTH-LIMITED-SEARCH(problem, depth)
  if \(result\) is not \(\) fails then return \(result\)
end

### Iterative deepening search \(l = 0\)

Start

### Iterative deepening search \(l = 1\)

Start
Properties of iterative deepening search

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

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