CHAPTER 3: SECTIONS 1–5

Problem solving and search
Outline

Basic search algorithms

Example problems

Problem formulation

Problem types

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

Note: this is offline problem solving.

```
function Simple-Problem-Solving-Agent(\(d\)) returns an action

Restricted form of General agent:

```

return action

s' \rightarrow REMAINDER(s, state)

action \rightarrow RECOMMENDATION(s, state)

s \rightarrow SEARCH\(\)\(p\)\(r\)\(v\)\(a\)\(l\)\(c\)\(e\)

problem \rightarrow FORMULATE-PROBLEM(state, \(g\))

\(g\) \rightarrow FORMULATE-GOAL(state)

if \(s\) is empty then

state \rightarrow UPDATE-STATE(state, \(p\))

problem, a problem formulation

\(g\) a goal, initially null

state, some description of the current world state

static: \(s\) an action sequence, initially empty

impulse: \(p\) a percent

return action

```
sequence of cities: e.g., Arad, Sibiu, Făgăraș, Bucharest

Find solution:

operators: drive between cities
states: various cities

Formulate problem:

be in Bucharest

Formulate goal:

Flight leaves tomorrow from Bucharest

On holiday in Romania; currently in Arad.

Example: Romania
Problem Types

Online exploration problem

Unknown state space

Often interfere with execution

Solution is a tree or policy

Must use sensors during execution

Non-deterministic, inaccessible

Deterministic, inaccessible

Multiple-state problem

Non-deterministic, inaccessible

Deterministic, accessible

Single-state problem
Example: Vacuum World
A solution is a sequence of operators executed, e.g., sum of distances run by operators executed, etc.

A problem is defined by four items:

1. Initial state, e.g., "at Arad"
2. Operators or successor functions $S(x)$
3. Implicit, e.g., $NO\text{\ small}(x)$
4. Explicit, e.g., $x = \text{at Bucharest}$

A problem can be a goal test, e.g., Sibiu $\leftarrow$ Arad $\leftarrow$ Zerind $\leftarrow$ Sibiu, etc.

path cost (additive)

Single-state problem formulation
Each abstract action should be "easier" than the original problem.

\[(\text{abstract}) \text{ solution} = (\text{abstract}) \text{ solution of set of real paths that are solutions in the real world} \]

For guaranteed realizability, any real state in A is a complex set of possible routes, detours, rest stops, etc.

\[(\text{abstract}) \text{ operator} = \text{complex combination of real actions} \]

Real world is absurdly complex.

Selecting a state space
Example: The 8-puzzle
[Note: optimal solution of $u$-Puzzle family is NP-hard]

$\text{path cost?} = 1$ per move

$\text{goal test?} = \text{goal state (given)}$

$\text{operators?} = \text{move blank left, right, up, down (ignore unsatisfying etc.)}$

$\text{states?} = \text{integer locations of tiles (ignore intermediate positions)}$

Example: The 8-Puzzle
Example: Vacuum world state space graph
path cost: 1 per operator

goal test: no dirt

operators: Left, Right, Such

states: integer dirt and robot locations (ignore dirt amounts)

Example: Vacuum World State Space Graph
Example: Robotic Assembly

path cost: time to execute

goal test: complete assembly with no robot included

operators: continuous motions of robot joints

parts of the object to be assembled

robot joint angles

states: real-valued coordinates of
end

else expand the node and add the resulting nodes to the search tree

if the node contains a goal state then return the corresponding solution

if there is a legal node for expansion according to strategy

if there are no candidates for expansion then return failure

loop do

initialize the search tree using the initial state of problem

function (general-search (problem, strategy)) return a solution, or failure

(k-a. expanding states)

by generating successors of already-expanded states

offline, simulated exploration of state space

Basic idea:

Search algorithms
function General-Search(problem, initial-state) returns a solution, or failure

end

if Goal-Test(problem, node) succeeds then
    return node

if nodes is empty then
    failure

nodes → make-queue(make-node(initial-state, problem))
node → remove-front(nodes)
if queues-expanding Functions Operators [problem]
    then nodes → queue-queue(node, queues, Operators [problem])
    end

loop do
    node → remove-front(nodes)
    if Goal-Test(problem, node) succeeds then return node

    nodes → queue-queue(node, queues, Operators [problem])

end
corresponding states. Using the OPERATORS (or SUCCESSORS) of the problem to create the
The EXPAND function creates new nodes, filling in the various fields and

States do not have parents, children, depth, or path cost.
States includes parent, children, depth, path cost \( g(x) \).
A node is a data structure constituting part of a search tree.
A state is a (representation of) a physical configuration.

Implementation cont'd: states vs. nodes
Search strategies

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

Time and space complexity are measured in terms of:

- optimality—does it always find a least-cost solution?
- space complexity—maximum number of nodes in memory
- time complexity—number of nodes generated/expanded
- completeness—does it always find a solution if one exists?

A strategy is defined by picking the order of node expansion.
Iterative deepening search
Depth-limited search
Depth-first search
Uniform-cost search
Breadth-first search

In the problem definition, uniform search strategies use only the information available.
Implementation:
Expand shallowest unexpanded node

Breadth-First Search
Zerind
Sibiu
Timisoara
Arad

-
Space is the big problem: can easily generate nodes at 1MB/sec

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

Space? \( (pq)O \) (keeps every node in memory)

Time? \( p^q + \cdots + q^3 + q + q + q + q + q \)

Complete? Yes (if \( q \) is finite)

Properties of breadth-first search
Romania with step costs in km
Uniform-cost search

And

\[ \text{Enqueue} = \text{Insert in order of increasing path cost} \]

Implementation:

Expand least-cost unexpanded node
Zerind

Sibiu

Timisoara

75

140

118
Properties of uniform-cost search:

- **Optimal**: Yes
- **Space**: # of nodes with cost of optimal solution ≤ \( b \)
- **Time**: # of nodes with cost of optimal solution ≤ \( b \)
- **Complete**: Yes, if step cost \( \geq e \)
Implementation:

Expand deepest unexpanded node
Need a finite, non-cyclic search space (or repeated-state checking)

\[ \text{Zerind} \rightarrow \text{Sibiu} \rightarrow \text{Timisoara} \rightarrow \text{Arad} \rightarrow \text{Oradea} \rightarrow \text{Sibiu} \rightarrow \text{Zerind} \]

\[ \vdots \]
O

DES on a depth-3 binary tree
DFS on a depth-3 binary tree, cont.
Properties of depth-first search
Nodes at depth \( l \) have no successors.

Implementation:

\[ \text{depth-first search with depth limit} \]

Depth-limited search
function Iterative-Deepening-Search (Problem) returns a solution sequence

end

if result ≠ cutoff then return result

result ← Depth-Limited-Search (Problem, depth)

for depth → 0 to ∞ do


input: Problem, a problem

Iterative deepening search
And

Iterative deepening search = 0
Can be modified to explore uniform-cost tree

Optimal? Yes, if step cost = 1

Space

\[(pq)O = pq + \cdots + q(1-p) + qp + q(1+p)\]

Time

Complete? Yes

Properties of iterative deepening search
Chapter 3: Sections 1-5

Summary

Variety of uninformed search strategies to define a state space that can feasibly be explored.

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

Problem formulation usually requires abstracting away real-world details.