Chapter 3

Outline

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

Problem-solving agents

Restricted form of general agent:

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.

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

Example Romania

Deterministic, fully observable ⇒ single-state problem
Non-deterministic, non-observable ⇒ conformant problem
Non-deterministic, partially observable ⇒ contingency problem
Unknown state space ⇒ exploration problem

Problem types

Deterministic, fully observable ⇒ single-state problem
Agent knows exactly which state it will be in; solution is a sequence
Non-observable ⇒ conformant problem
Agent may have no idea where it is; solution (if any) is a sequence
Nondeterministic and/or partially observable ⇒ contingency problem
Percepts provide new information about current state
Solution is a contingent plan or a policy
Often interleave search, execution
Unknown state space ⇒ exploration problem ("online")

Chapter 3 3

Example: Romania

Arad
Sibiu
Fagaras
Bucharest
Timisoara
Lugoj
Mehadia
Dobreta
Craiova
Sibiu
Fagaras
Pitesti
Vaslui
Iasi
Rimnicu Vilcea
Bucharest

71
75
118
111
70
75
120
151
140
99
80
97
101
92
87
86
90
98
142
101
92
87
86
90
98
142

Chapter 3 4

Example: Romania

Giurgiu
Urziceni
Hirsova
Eforie
Neamt
Oradea
Zerind

Chapter 3 5

Non-deterministic
⇒ single-state problem
Agent knows exactly which state it will be in; solution is a sequence

Problem types

Deterministic, fully observable =⇒ single-state problem
Agent knows exactly which state it will be in; solution is a sequence
Non-observable =⇒ conformant problem
Agent may have no idea where it is; solution (if any) is a sequence
Nondeterministic and/or partially observable =⇒ contingency problem
Percepts provide new information about current state
Solution is a contingent plan or a policy
Often interleave search, execution
Unknown state space =⇒ exploration problem ("online")
Each abstract action should be "easier" than the original problem.

A solution is a sequence of actions

A problem is defined by four items:

- Initial state: e.g., "at Arad"
- Successor function: \( S(x) \) = set of action-state pairs
- Goal test: can be explicit, e.g., \( x \) = "at Bucharest"
- Path cost (additive)

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

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.

An abstract state = set of real states

Real world is already complex.

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\}.

Example: Vacuum World

- Single-state
- Start in #5.

Solution?

- Right, Suck, Left, Suck
- Contingency, start in #5 or #7

Murphy's Law: Suck can dirty a clean carpet.

Local sensing: dirt, location only.

Example: Vacuum World

- Single-state
- Start in #5.

Solution?

- Right, if dirt then Suck

Chapter 3

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
- Goal test: can be explicit, e.g., \( x \) = "at Bucharest"
- Path cost (additive)

A path cost \( c(x,a,y) \) is the step cost, assumed to be \( \geq 0 \).

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\}.

Example: Vacuum World

- Single-state
- Start in #5.

Solution?

- Right, Suck

Chapter 3

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 = set of real paths that are solutions in the real world.

Each abstract action should be "easier" than the original problem.

Chapter 3
Example: Vacuum world state space graph

Example: The 8-puzzle

Example: The 8-puzzle

Example: The 8-puzzle
Example: The 8-puzzle

Start State: Goal State:

5 1 3 4 6 7 8
5 1 3 4 6 7 8

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

states: real-valued coordinates of robot joint angles

actions: continuous motions of robot joints

goal test: parts of the object to be assembled

path cost: real-valued coordinates of robot joint angles

Chapter 3

Tree search algorithms

Basic idea:

Function Tree-Search

end

The root node and the root node to the root

If the root contains a goal state then return the coordinating solution

Choose a leaf node for expansion according to a strategy

If there are no candidates for expansion then return failure

else

do

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

end

Chapter 3

Tree search example

Example: robotic assembly
Chapter 3

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 in terms of
  - $b$ — maximum depth of the search tree
  - $d$ — depth of the least-cost solution
  - $m$ — maximum number of nodes in memory
- **Space complexity**: maximum number of nodes stored in memory.
- **Optimality**: does it always find a least-cost solution?

Time and space complexity are measured in terms of $b$ and $m$.

Strategies are evaluated along the following dimensions:

- A strategy is defined by picking the order of node expansion.
- Heuristic functions are never used.
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Uninformed search strategies use only the information available in the problem definition.

### Breadth-first search

Expand shallowest unexpanded node.

**Implementation**

- **Fringe**: is a FIFO queue. i.e., new successors go at end.

---

**Search strategies**

---

Tree search example
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? $1 + b + b^2 + b^3 + \ldots + b^d + b^{(b^d - 1)} = O(b^d + 1)$, 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

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, $O(b^{\lceil C^*/\epsilon \rceil})$

where $C^*$ is the cost of the optimal solution

Space??

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

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:
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

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

Expanded deepest unexpanded node
Depth-limited search = depth-first search with depth limit, i.e., nodes at depth \( l \) have no successors.

### Recursive implementation:

```plaintext
function Depth-Limited-Search(problem, limit)
returns soln/fail/cutoff

Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit)
returns soln/fail/cutoff

cutoff-occurred? ← false
if Goal-Test(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 ≠ failure
    then return result
if cutoff-occurred?
then return cutoff
else return failure
```

---

Iterative deepening search

```plaintext
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
```

---

Chapter 3 61

Iterative deepening search

<table>
<thead>
<tr>
<th>l</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 0 )</td>
</tr>
<tr>
<td>1</td>
<td>( 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( 2 )</td>
</tr>
<tr>
<td>3</td>
<td>( 3 )</td>
</tr>
</tbody>
</table>

---

Iterative deepening search: starts with depth 1, then depth 2, and so on until a solution is found.
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: \(O(b^d)\)

Optimal: Yes, if step cost = 1

Can be modified to explore uniform-cost trees.

Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-First</th>
<th>Depth-Limited</th>
<th>Depth-First</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes (if step cost = 1)</td>
</tr>
<tr>
<td>Time</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
</tr>
<tr>
<td>Space</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
<td>(b^d + 1)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes (if step cost = 1)</td>
</tr>
</tbody>
</table>
Repeated state checking

Depth-first search: Is checking current node w.r.t. path stored in memory enough? i.e. Is linear space sufficient?

No! Can only detect looping paths, not all repeated states. Need exponential space to store all visited nodes.

Graph search function

\[
\text{Graph-Search}(\text{problem}, \text{fringe}) \quad \text{returns} \quad \text{a solution, or failure}
\]

\[
\begin{align*}
\text{closed} & \leftarrow \text{an empty set} \\
\text{fringe} & \leftarrow \text{Insert}(\text{Make-Node}(\text{Initial-State}[\text{problem}]), \text{fringe}) \\
\text{loop do} & \\
& \quad \text{if fringe is empty then return failure} \\
& \quad \text{node} \leftarrow \text{Remove-Front}(\text{fringe}) \\
& \quad \text{if Goal-Test}(\text{problem}, \text{State}[\text{node}]) \text{ then return node} \\
& \quad \text{if State}[\text{node}] \text{ is not in closed then} \\
& \quad \quad \text{add State}[\text{node}] \text{ to closed} \\
& \quad \text{fringe} \leftarrow \text{InsertAll}(\text{Expand}(\text{node}, \text{problem}), \text{fringe}) \\
\end{align*}
\]

Is this optimal?
- BFS in INSERTALL
- DFS in INSERTALL

Need exponential space to store all visited nodes. Not can only detect looping paths, not all repeated states.

Chapter 3 73

Repetitive state checking

Chapter 3 74

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

Chapter 3 75

Repeated state checking

Depth-first search: Is checking current node w.r.t. path stored in memory enough? i.e. Is linear space sufficient?

No! Can only detect looping paths, not all repeated states. Need exponential space to store all visited nodes.

Graph search function

\[
\text{Graph-Search}(\text{problem}, \text{fringe}) \quad \text{returns} \quad \text{a solution, or failure}
\]

\[
\begin{align*}
\text{closed} & \leftarrow \text{an empty set} \\
\text{fringe} & \leftarrow \text{Insert}(\text{Make-Node}(\text{Initial-State}[\text{problem}]), \text{fringe}) \\
\text{loop do} & \\
& \quad \text{if fringe is empty then return failure} \\
& \quad \text{node} \leftarrow \text{Remove-Front}(\text{fringe}) \\
& \quad \text{if Goal-Test}(\text{problem}, \text{State}[\text{node}]) \text{ then return node} \\
& \quad \text{if State}[\text{node}] \text{ is not in closed then} \\
& \quad \quad \text{add State}[\text{node}] \text{ to closed} \\
& \quad \text{fringe} \leftarrow \text{InsertAll}(\text{Expand}(\text{node}, \text{problem}), \text{fringe}) \\
\end{align*}
\]

Is this optimal?
- BFS in INSERTALL
- DFS in INSERTALL

Need exponential space to store all visited nodes. Not can only detect looping paths, not all repeated states.

Chapter 3 76

Graph search

Chapter 3 77

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. Lower memory demands can result in smaller memory requirements. Graph search can be exponentially more efficient than tree search.

Chapter 3 78

Complexity of BFS and DFS

Complexity of BFS and DFS is linear in the number of states. In particular, Dijkstra's algorithm for single source shortest paths is \(\Theta(V + E \log V)\), i.e. polynomial in \(V\). However, \(V\) is \(b^m\) in many cases.

e.g. chess, theorem proving, scheduling problems.