Problem solving and search

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 ← RECOMMENDATION(seq, state)
seq ← REMAINDER(seq, state)
return action
```

Note: this is offline problem solving; solution executed “eyes closed.”
Online problem solving involves acting without complete knowledge.

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”)
Example: vacuum world

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

Conformant, start in \{1, 2, 3, 4, 5, 6, 7, 8\}. Solution??

Contingency, start in #5
Murphy’s Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.

Solution??

\[ \text{Right, Suck, Left, Suck} \]

Contingency, start in #5

Murphy’s Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.

Solution??

\[ \text{Right, if dirt then Suck} \]

---

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., \( S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \ldots\} \)
- **goal test**, can be
  - explicit, e.g., \( x = \) “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

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

(Abstract) state = set of real states

(Abstract) action = 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 etc.)
- **actions**: Left, Right, Suck, NoOp
- **goal test**: no dirt
- **path cost**: 1 per action (0 for NoOp)

Example: vacuum world state space graph

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

Example: vacuum world state space graph

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

Example: The 8-puzzle

- **states**:
- **actions**: Left, Right, Suck, NoOp
- **goal test**:
- **path cost**:

Start State

Goal State
Example: The 8-puzzle

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

**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: The 8-puzzle

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

**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

---

Example: Robotic assembly

**states**?: real-valued coordinates of robot joint angles

**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. exploring states)

**function** 

Tree-Search( problem, strategy ) returns a solution, or failure

1. Initialize the search tree using the initial state of problem
2. loop do
   3. if there are no candidates for expansion then return failure
   4. choose a leaf node for expansion according to strategy
   5. if the node contains a goal state then return the corresponding solution
   6. else expand the node and add the resulting nodes to the search tree
end
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 successor function 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 node
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
end loop

function EXPAND(node, problem) returns a set of nodes
successors ← the empty set
for each action, result in SUCCESSOR-FN(problem, State(node)) do
  s ← a new Node
  Parent-Node[s] ← node; Action[s] ← action; State[s] ← result
  Path-Cost[s] ← Path-Cost[node] + Step-Cost(State[node], action, result)
  Depth[s] ← Depth[node] + 1
  add s to successors
return successors
**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 $\infty$)

---

**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

---

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

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:

\[ \text{fringe} = \text{queue ordered by path cost, lowest first} \]

Equivalent to breadth-first if step costs all equal

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

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

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

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:

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

Diagram of depth-first search:

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
- N
- O
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?? $O(lm)$, i.e., linear space!

Optimal?? No

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

Recursive implementation:

```python
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

function \textsc{Iterative-Deepening-Search}(\textit{problem}) returns a solution
  \begin{itemize}
  \item \textit{inputs:} \textit{problem}, a problem
  \item \textbf{for} \textit{depth} ← 0 to ∞
  \item \textit{result} ← \textsc{Depth-Limited-Search}(\textit{problem}, \textit{depth})
  \item \textbf{if} \textit{result} \neq \textit{cutoff} \textbf{then} return \textit{result}
  \item \textbf{end}
  \end{itemize}
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 leaf:

\[ N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \]

\[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 99,990 = 1,111,100 \]

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

---

**Repeated states**

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

function Graph-Search(problem, fringe) returns a solution, or failure

  closed ← an empty set
  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 node
  if State[node] is not in closed then
    add State[node] to closed
    fringe ← InsertAll(Expand(node, problem), fringe)
end

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