Chapter 3

Problem solving and search
Outline

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

Restricted form of general agent:

```plaintext
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.
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
Problem types

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

Non-observable $\implies$ conformant problem
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 interpolate search, execution

Unknown state space $\implies$ exploration problem (“online”)
Example: vacuum world

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

1
2
3
4
5
6
7
8
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\}. Solution??
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\}. Solution??
[Right, Suck, Left, Suck]

Contingency, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
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\}. Solution??
[Right, Suck, Left, Suck]

Contingency, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
[Right, if dirt then Suck]
A problem is defined by four items:

initial state e.g., “at Arad”

successor function $S(x) = \text{set of action-state pairs}$
e.g., $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \ldots\}$

goal test, can be
explicit, e.g., $x = \text{“at Bucharest”}$
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 =
  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??
actions??
goal test??
path cost??
Example: vacuum world state space graph

**states??**: integer dirt and robot locations (ignore dirt amounts etc.)

**actions??**

**goal test??**

**path cost??**
**Example: vacuum world state space graph**

- **states**: integer dirt and robot locations (ignore dirt amounts etc.)
- **actions**: Left, Right, Suck, NoOp
- **goal test**
- **path cost**
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??**
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

**Start State**

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

**Goal State**

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

- **states??**
- **actions??**
- **goal test??**
- **path cost??**
Example: The 8-puzzle

states: integer locations of tiles (ignore intermediate positions)
actions
goal test
path cost
Example: The 8-puzzle

states: integer locations of tiles (ignore intermediate positions)

actions: move blank left, right, up, down (ignore unjamming etc.)

goal test

path cost
Example: The 8-puzzle

**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**: 

Start State:

```
7  2  4
5  6
8  3  1
```

Goal State:

```
1  2  3
4  5  6
7  8
```
Example: The 8-puzzle

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 parts of the object to be assembled

**actions**?: continuous motions of robot joints

**goal test**?: complete assembly

**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

Arad

Sibiu

Fagaras

Oradea

Rimnicu Vilcea

Timisoara

Arad

Lugoj

Zerind

Arad

Oradea
Tree search example

Arad

Sibiu

Târgoviște

Zerind

Arad

Fagaras

Oradea

Rimnicu Vilcea

Arad

Lugoj

Sibiu

Timisoara

Oradea

Zerind
Tree search example

```
    Arad
   /   \\
Sibiu     Timisoara     Zerind
/    /  \\/    \    \   /    /
Arad  Fagaras  Oradea  Rimnicu Vilcea  Arad  Lugoj  Arad  Oradea
```

Chapter 3   27
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 node
  fringe ← InsertAll(Expand(node, problem), fringe)

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
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

![Breadth-first search diagram](image)
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

![Breadth-first search diagram]

- **A**
  - **B**
    - **D**
    - **E**
  - **C**
    - **F**
    - **G**
Properties of breadth-first search

Complete??
Properties of breadth-first search

**Complete**
Yes (if \( b \) is finite)

**Time**
??
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**?
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**
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:**

\(\text{fringe} = \text{queue ordered by path cost, lowest first}\)

Equivalent to breadth-first if step costs all equal

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

**Time??** # of nodes with \( g \leq \) 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:**

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

Expand deepest unexpanded node

**Implementation:**

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

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

Expand deepest unexpanded node

**Implementation:**

\( fringe = \) LIFO queue, i.e., put successors at front

![Diagram of a depth-first search tree](image)
Depth-first search

Expand deepest unexpanded node

**Implementation:**

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

![Diagram of Depth-first search tree](image)
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Properties of depth-first search

Complete??
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??**
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**??
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??** $O(bm)$, i.e., linear space!

**Optimal??**
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??** $O(bm)$, 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:

\[
\text{function } \text{Depth-Limited-Search}( \text{problem}, \text{limit} ) \text{ returns } \text{soln/fail/cutoff}
\]

\[
\text{Recursive-DLS}( \text{Make-Node(Initial-State[problem]), problem, limit} )
\]

\[
\text{function } \text{Recursive-DLS}( \text{node, problem, limit} ) \text{ returns } \text{soln/fail/cutoff}
\]

\[
\text{cutoff-occurred? } \leftarrow \text{false}
\]

\[
\text{if Goal-Test(problem, State[node]) then return node}
\]

\[
\text{else if Depth[node] = limit then return cutoff}
\]

\[
\text{else for each successor in Expand(node, problem) do}
\]

\[
\text{result } \leftarrow \text{Recursive-DLS(successor, problem, limit)}
\]

\[
\text{if result = cutoff then cutoff-occurred? } \leftarrow \text{true}
\]

\[
\text{else if result } \neq \text{failure then return result}
\]

\[
\text{if cutoff-occurred? then return cutoff else return failure}
\]
Iterative deepening search

**function** \textsc{Iterative-Deepening-Search}(problem) \textbf{returns} a solution

**inputs:** problem, a problem

\begin{verbatim}
for depth ← 0 to ∞ do
    result ← \textsc{Depth-Limited-Search}(problem, depth)
    if result ≠ cutoff then return result
end
\end{verbatim}
Iterative deepening search $l = 0$

Limit = 0
Iterative deepening search $l = 1$

Limit = 1

Diagram showing the iterative deepening search process for $l = 1$. The diagram illustrates the search algorithm starting from node A, exploring up to depth 1 and stopping at the first level ofNodes B and C.
Iterative deepening search $l = 2$

Limit = 2

Diagram showing the iterative deepening search with a limit of 2, illustrating the exploration of the search space through multiple iterations, each with an increasing depth limit.
Iterative deepening search $l = 3$

Limit = 3
Properties of iterative deepening search

Complete??
Properties of iterative deepening search

**Complete?? Yes**

**Time??**
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??**
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**
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:

\[
\begin{align*}
N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\end{align*}
\]

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**
<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 $l \geq d$</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$b^{[C^*/\epsilon]}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^{[C^*/\epsilon]}$</td>
<td>$bm$</td>
<td>$bl$</td>
<td>$bd$</td>
</tr>
<tr>
<td>Optimal?</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!

A

B

C

D

A

B

C

C

B

C

C
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

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