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:

```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 ← 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
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 interleave 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 or #7
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 or #7  
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**

![Diagram of vacuum world state space graph]
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

Goal State

states??
actions??
goal test??
path cost??
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</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>8 3 1</td>
<td>5 6</td>
</tr>
<tr>
<td></td>
<td>7 8</td>
</tr>
</tbody>
</table>

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??
Example: The 8-puzzle

Start State

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

Goal State

```
1 2 3
4 5 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??:
Example: robotic assembly

**states??**: real-valued coordinates of robot joint angles
parts of the object to be assembled

**actions??**: 
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??:
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 **with no robot included!**

*path cost??*: 
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 **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. expanding states)

**function** \( \text{Tree-Search}(\text{problem}, \text{strategy}) \) **returns** a solution, or failure

initialize the search tree using the initial state of \( \text{problem} \)

**loop** do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to \( \text{strategy} \)
  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
Tree search example
Tree search example
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

```
A

B          C

d
D  E  F  G
```
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

![SearchTreeDiagram]
Breadth-first search

Expand shallowest unexpanded node

Implementation: fringe is a FIFO queue, i.e., new successors go at end

Diagram shows a tree with nodes A, B, C, D, E, F, and G. Node A is the root, with B and C as its children. B has children D and E, and C has children F and G.
Properties of breadth-first search

Complete??
<table>
<thead>
<tr>
<th>Properties of breadth-first search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complete</strong></td>
</tr>
<tr>
<td><strong>Time</strong></td>
</tr>
</tbody>
</table>
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??**
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, \(O(b^{\lceil C^*/\epsilon \rceil})\)

where \(C^*\) is the cost of the optimal solution

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

**Optimal??** Yes—nodes expanded in increasing order of \(g(n)\)
Depth-first search

Expand deepest unexpanded node

**Implementation:**

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
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:**

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

Expand deepest unexpanded node

**Implementation:**

\emph{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:**

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
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:**

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

Expand deepest unexpanded node

**Implementation:**

$fringe = \text{LIFO queue, i.e., put successors at front}$
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
    - \( \Rightarrow \) 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:**

```plaintext
function DEPTH-LIMITED-SEARCH( problem, limit ) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

def 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
```

Chapter 3  61
Iterative deepening search

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
Iterative deepening search $l = 0$
Iterative deepening search $l = 1$
Iterative deepening search $l = 2$

Limit = 2
Iterative deepening search $l = 3$

Limit = 3

Chapter 3  66
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
### Summary of algorithms

<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>$b^{\lceil C*/\epsilon \rceil}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^{\lceil C*/\epsilon \rceil}$</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 state checking

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

Failure to detect repeated states can turn a linear problem into an exponential one!
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 \textsc{Graph-Search}( \textit{problem}, \textit{fringe} ) returns a solution, or failure

\begin{itemize}
  \item \textit{closed} $\leftarrow$ an empty set
  \item \textit{fringe} $\leftarrow$ \textsc{Insert}(\textsc{Make-Node}($\text{Initial-State}[\textit{problem}]$), \textit{fringe})
\end{itemize}

\begin{algorithm}
\begin{algorithmic}
\Loop
  \If {\textit{fringe} is empty} then \textbf{return} failure \\
  \State \textit{node} $\leftarrow$ \textsc{Remove-Front}(\textit{fringe}) \\
  \If {\textsc{Goal-Test}(\textit{problem}, \text{State}[\textit{node}])} then \textbf{return} \textit{node} \\
  \If {\text{State}[\textit{node}] is not in \textit{closed}} then \\
    \State add \text{State}[\textit{node}] to \textit{closed} \\
    \State \textit{fringe} $\leftarrow$ \textsc{InsertAll}(\textsc{Expand}(\textit{node}, \textit{problem}), \textit{fringe})
  \EndIf
\EndIf
\EndLoop
\end{algorithmic}
\end{algorithm}

Is this optimal?

- BFS in \textsc{InsertAll}
- DFS in \textsc{InsertAll}
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.
Complexity of BFS and DFS

Complexity of BFS and DFS is linear in the number of states $V$

In particular, Dijkstra’s algorithm for single source shortest paths is $\Theta(E + V \log V)$, i.e. polynomial in $V$

However, $V$ is $b^m$ in many cases.

e.g. chess, theorem proving, scheduling problems