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:

```java
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 interleave search, execution

Unknown state space $\implies$ exploration problem (“online”)
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
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: The 8-puzzle

![Start State](image1)

![Goal State](image2)

- **states??**
- **actions??**
- **goal test??**
- **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**

**goal test**

**path cost**
Example: The 8-puzzle

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Start State \quad Goal State

**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

Start State

Goal State

**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**: 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 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
  - Arad
  - Fagaras
  - Oradea
  - Rimnicu Vilcea
- Timisoara
  - Arad
  - Lugoj
- Zerind
  - Arad
  - Oradea
Tree search example
Tree search example
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 `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(node, action, s)
  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

---

Chapter 3
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

Expand shallowest unexpanded node

Implementation:

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

\( fringe = \) 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, \( O(b[C^*/\epsilon]) \)

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:

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

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

Expand deepest unexpanded node

**Implementation:**

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

![Depth-first search diagram](image)
**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:**

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??
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
   \[ \Rightarrow \text{complete in finite spaces} \]

**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:

```
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 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 $l = 1$

Limit = 1

Diagram showing the iterative deepening search process with nodes A, B, and C.
Iterative deepening search $l = 2$

Limit = 2
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:

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

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-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>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</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!
**Graph search**

**function**  \textsc{Graph-Search}(problem, fringe)  \textbf{returns} a solution, or failure

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

**loop** do

- if \textit{fringe} is empty then **return** failure

- \textit{node} ← \textsc{Remove-Front}(\textit{fringe})

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

- if \text{State}[	extit{node}] is not in \textit{closed} then

  - add \text{State}[	extit{node}] to \textit{closed}

  - \textit{fringe} ← \textsc{InsertAll}([\textsc{Expand}(\textit{node}, problem), \textit{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