Solving Problem by Searching

Chapter 3
Outline

- Problem-solving agents
- Problem formulation
- Example problems
- Basic search algorithms – blind search
- Heuristic search strategies
- Heuristic functions
Problem-solving agents

```
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 do
    goal ← FORMULATE-GOAL( state )
    problem ← FORMULATE-PROBLEM( state, goal )
    seq ← SEARCH( problem )
    action ← FIRST( seq )
    seq ← REST( seq )
return action
```
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 \(\rightarrow\) single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence

- Non-observable \(\rightarrow\) sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence

- Nondeterministic and/or partially observable \(\rightarrow\) contingency problem
  - Percepts provide **new** information about current state
  - Often **interleave** search, execution

- Unknown state space \(\rightarrow\) exploration problem
Example: vacuum world

- Single-state, start in #5.
  Solution?
Example: vacuum world

- **Single-state**, start in #5. **Solution?** [Right, Suck]

- **Sensorless**, start in \{1,2,3,4,5,6,7,8\} e.g., Right goes to \{2,4,6,8\} **Solution?**
Example: vacuum world

- Sensorless, 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
  - Nondeterministic: Suck may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: [L, Clean], i.e., start in #5 or #7
  Solution?
Example: vacuum world

- **Sensorless**, 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**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: \[L, \text{Clean}\], i.e., start in #5 or #7
  
  **Solution?** [Right, if dirt then Suck]
Problem formulation

A (complete state) formulation of a problem: 6 items:

- **States**: the set of all states
- **initial state** e.g., “In(Arad)“
- **actions** – Actions(s): actions that can be performed in state s
- **transition model (successor function)**: \( \text{Result}(s, a) = s' \)
  
  e.g., \( \text{Result}(\text{In(Arad)}, \text{Go(Zerind)}) = \text{In(Zerind)} \)
- **goal test**, can be
  - **explicit**, e.g., \( x = \text{"In(Bucharest)"} \)
  - **implicit**, e.g., \( \text{Checkmate}(x) \)
- **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
Vacuum world state space graph

- states?
- actions?
- transition model?
- goal test?
- path cost?
Vacuum world state space graph

- **states?** integer dirt and robot location
- **actions?** Left, Right, Suck
- **transition model?** shown by the above graph
- **goal test?** no dirt at all locations
- **path cost?** 1 per action
Example: The 8-puzzle

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

- **states?** locations of tiles
- **actions?** move blank left, right, up, down
- **goal test?** = goal state (given)
- **path cost?** 1 per move

[Note: optimal solution of n-Puzzle family is NP-hard]
Example: The 8-queens problem

- states?
- actions?
- goal test?
- path cost?
Example: The 8-queens problem

- An incremental formulation:
  - **Initial state?** Empty board
  - **actions?** Place a queen in left-most empty column s. t. it is not attacked
  - **transition model?** The result board
  - **goal test?** When all 8 queens are placed
  - **path cost?**
Tree search algorithms

Basic idea:

- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```plaintext
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
```
Tree search example
Tree search example
Tree search example
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 SOLUTION(node)
    fringe ← INSERT-ALL(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
Tree search and graph search

**Function TREE-SEARCH**(problem) **returns** a solution, or failure

Initialize the frontier (open set) using the initial state of problem

loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

**Function GRAPH-SEARCH**(problem) **returns** a solution, or failure

Initialize the frontier using the initial state of problem

Initialize the explored set (closed set) to be empty

loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if the result node is not in the frontier or explored set
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 state, parent node, action, path cost $g(x)$, depth.

The **Expand** function creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.
Search strategies

A search 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
- **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 length of any path in the state space (may be $\infty$) – maximum depth of the search tree
Uninformed (blind) search strategies

- **Uninformed** search 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
- The goal test is performed when a node is generated
- 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
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?** $1 + b + b^2 + b^3 + \ldots + b^d + \cdots = O(b^d)$
- **Space?** $O(b^d)$ (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step)
- **Space** is the bigger problem (more than time)
Uniform-cost (graph) search

- Expand least-cost unexpanded node

**Implementation:**
- $frontier = \text{queue ordered by path cost } g(n)\text{ for each node } n$

- Essentially the same as breadth-first graph search with two modifications:
  - The goal test is performed when a node is selected for expansion
  - A test is added in case a better path is found to a node currently in the frontier – the better path replaces the worse one
Uniform-cost search

Search for the shortest path from Sibiu to Bucharest
Uniform-cost search

**Complete?** Yes, if step cost $\geq \epsilon > 0$

**Time?** # of nodes with $g \leq \text{cost of optimal solution}$ is $O(b^{\lceil C^*/\epsilon \rceil})$ where $C^*$ is the cost of the optimal solution – so the time complexity is $O(b^{1+\lceil C^*/\epsilon \rceil})$

**Space?** # of nodes with $g \leq C^*$ is $O(b^{\lceil C^*/\epsilon \rceil})$, so the space complexity is $O(b^{1+\lceil C^*/\epsilon \rceil})$

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

- Expand deepest unexpanded node

Implementation:
- \textit{frontier} = LIFO queue, i.e., put successors at front
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - \textit{frontier} = 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:**
  - $\textit{frontier} = \text{LIFO queue, i.e., put successors at front}$
Depth-first search

- Expand deepest unexpanded node

**Implementation:**
- \textit{frontier} = LIFO queue, i.e., put successors at front
Depth-first search

- Expand deepest unexpanded node

Implementation:

- *frontier* = LIFO queue, i.e., put successors at front
Depth-first search

- Expand deepest unexpanded node

**Implementation:**

- \textit{frontier} = LIFO queue, i.e., put successors at front

![Diagram of a tree structure demonstrating depth-first search with nodes A, C, F, G, L, M, N, O, and branch points indicated. The root node A is expanded, followed by its children, and so on. The path from A to C is highlighted to show the depth-first search process.]
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - \textit{frontier} = LIFO queue, i.e., put successors at front
Properties of depth-first search

- **Complete?** For tree search: No - it fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path for tree search, or use graph search
    - 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:
Iterative deepening search

```plaintext
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```
Iterative deepening search $l = 0$
Iterative deepening search $l = 1$
Iterative deepening search $l = 2$
Iterative deepening search $l = 3$
Iterative deepening search

- Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:
  \[ N_{DLS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d \]

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  \[ N_{IDS} = d b^{\lfloor d \rfloor} + (d-1)b^{\lfloor d-1 \rfloor} + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d \]

  For $b = 10$, $d = 5$,
  \[ N_{DLS} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 \]
  \[ N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \]

  Overhead = \( \frac{(123,450 - 111,110)}{111,110} \) = 11%
Properties of iterative deepening search

- **Complete?** Yes
- **Time?** \( d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d) \)
- **Space?** \( O(bd) \)
- **Optimal?** Yes, if step cost = 1
# 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>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^*}/\epsilon)$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^*}/\epsilon)$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>


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