Solving problems by searching

Chapter 3
Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
Example: vacuum world

- Single-state, start in #5.

Solution?
Example: vacuum world

- **Single-state**, start in #5.
  Solution? \([\text{Right, Suck}]\)

- **Sensorless**, start in \(\{1,2,3,4,5,6,7,8\}\) e.g., \(\text{Right}\) 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: \([L, \text{Clean}]\), 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**]
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
Vacuum world state space graph

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

- **states?** integer dirt and robot location
- **actions?** *Left, Right, Suck*
- **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]
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 depth of the state space (may be \( \infty \))
Uninformed search strategies

- **Uninformed** search strategies use only the information available in the problem definition, and successors can only be distinguished as goal and non-goal states (no “prior idea” how good a successor state is). Otherwise: informed search or heuristics

- 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

- 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?** $1 + b + b^2 + b^3 + ... + b^d + b(b^d - 1) = O(b^{d+1})$
  - Worse case, the goal is at the very last node of depth $d$, so no need to expand it.

- **Space?** $O(b^{d+1})$ (keeps every node in memory)

- **Optimal?** Yes (if cost = 1 per step)

- **Space** is the bigger problem (more than time)
Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation:**
  - fringe = queue ordered by path cost
- 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:**
- 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

```
    A
   / \  
  B   C
 /\  /\  /
D  E F G
 |  |  |  |
I  J  K  L
 |  |  |  |
H  O  M  N
```
Depth-first search

- Expand deepest unexpanded node

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

[Diagram of a tree structure]
Depth-first search

- Expand deepest unexpanded node

**Implementation:**

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

i.e., nodes at depth /have no successors

```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 SOLUTION(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, 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 $l \neq 0$
Iterative deepening search / = 1

Limit = 1

Diagram showing iterative deepening search process with a limit of 1.
Iterative deepening search / = 2

Limit = 2

Diagram of iterative deepening search with limit 2.
Iterative deepening search \( l = 3 \)
Iterative deepening search

- Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:

$$N_{IDS} = (d+1)b^0 + db^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- Overhead $= (123,456 - 111,111)/111,111 = 11\%$
Properties of iterative deepening search

- **Complete?** Yes
- **Time?** \((d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)\)
- **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>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</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!
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
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