Chapter 3: Solving Problems by Searching
The General Idea

Initial State $\rightarrow$ Goal State

$\text{Search} \rightarrow \text{Action}_i \rightarrow \text{State}_i$
A Simple Problem-Solving Agent

\begin{verbatim}
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
  inputs: p, a percept
  static: s, an action sequence, initially empty
           state, some description of the current world state
           g, a goal, initially null
           problem, a problem formulation

  state ← UPDATE-STATE(state, p)
  if s is empty then
    g ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, g)
    s ← SEARCH(problem)
    action ← RECOMMENDATION(s, state)
    s ← REMAINDER(s, state)
  return action
\end{verbatim}
A Simple Problem-Solving Agent

```plaintext
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
inputs: p, a percept

static: s, an action sequence, initially empty
  state, some description of the current world state
  g, a goal, initially null
  problem, a problem formulation

state ← UPDATE-STATE(state, p)
if s is empty then
  g ← FORMULATE-GOAL(state)
  problem ← FORMULATE-PROBLEM(state, g)
  s ← SEARCH(problem)
action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
return action
```
class SimpleProblemSolvingAgent:

    def __init__(self):
        self.s = []
        self.state = None
        self.g = None
        self.problem = None

    def respond(self, p):
        if len(self.s) == 0:  # initially and perhaps later
            self.g = self.formulateGoal()
            self.problem = self.formulateProblem()
            self.s = self.Search()
        action = self.Recommendation()
        self.s = self.Remainder()
        return action
Terminology

• **operator** – an action taken to reach a state

• **state space** – the set of all possible states reachable from the initial state

• **goal test** – does state satisfy goal?

• **path** - sequence of actions leading from one state to another

• **path cost** – cost associated with a particular path (e.g., flying more expensive than driving)

• **search cost** – time/space required to find a solution
**Classic (Toy) Problems**

**The 8-Puzzle**

- **States**: location of each tile
- **Operators**: move blank up, right, left, or down
- **Goal test**: state looks like figure at right
- **Path cost**: total number of moves
The 8-Queens Problem

- **Goal test**: 8 queens on board, none under attack
- **Path cost**: zero (???)
8-Queens Variant #1

- **States**: arrangement of 0-8 queens on board
- **Operators**: add a queen to any square (till there are 8 on the board)
- **Search cost**: $64^8$ possible configs to check!

*Hint: We shouldn't put a queen on an attacked position...*
8-Queens Variant #2

- **States**: arrangement of 0-8 queens on board with none under attack
- **Operators**: place a queen in the leftmost empty column s.t. it is not attacked
- **Search cost**: 2057 possible sequences (one at right fails)

Conclusion: A better formulation of the problem can make a big difference!
Classic (Toy) Problems

Missionaries & Cannibals

• **States** : \(<M,C,B>\) on starting side, where 
  \(M = \#\text{missionaries},\ C = \#\text{cannibals},\ B = \#\text{boats}\)

• **Operators** : From each side, take either 1 M, 1 C, 2 M, 2 C, or one of each across.

• **Goal test** : \(<0,0,0>\)

• **Path cost** : \# of crossings
Classic (Toy) Problems

Towers of Hanoi

- **States**: \(<P_1, P_2, P_3>\) where \(P_i\) is configuration of \(i\)th peg
- **Operators**: Move a disc from \(P_i\) to \(P_j\) without putting a larger disc on a smaller one.
- **Goal test**: \(<0,0,P_3>\)
- **Path cost**: # of moves
Real-World Problems

- **Route finding**: go from New York to San Francisco for under $200 with the minimum 
  # of stopovers.

- **Traveling Salesman Problem**: Visit every 
  major city in a region without visiting any city 
  twice.

- **VLSI layout**: lay out circuit to minimize area 
  and connection lengths

- **Robot navigation**: continuous route-finding

- **Assembly sequencing**: putting together complex 
  objects
3.3 Searching for Solutions

- Many search spaces can be represented as trees.
- Each node in the tree represents a state.
- Each branch in the tree represents an action.
A General Tree-Search Algorithm

```plaintext
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)
```

- Search tree is just a list (stack, queue) - expanding a node means getting its children and replacing it with them
- **INSERTALL** function implements a strategy (more later)
- Let’s look at the **EXPAND** function…. 
A General Tree-Search Algorithm: Expanding Nodes

Now we can turn to search strategies…

function \texttt{Expand}(node, problem) returns a set of nodes

\texttt{successors} $\leftarrow$ the empty set

\texttt{for each action, result in Successor-Fn(problem, State[\texttt{node}]) do} \\
\hspace{2em} \texttt{s} $\leftarrow$ a new Node \\
\hspace{2em} \texttt{Parent-Node}[$\texttt{s}$] $\leftarrow$ \texttt{node}; \texttt{Action}[$\texttt{s}$] $\leftarrow$ \texttt{action}; \texttt{State}[$\texttt{s}$] $\leftarrow$ \texttt{result} \\
\hspace{2em} \texttt{Path-Cost}[$\texttt{s}$] $\leftarrow$ \texttt{Path-Cost}[$\texttt{node}$] + \texttt{Step-Cost}($\texttt{node}$, \texttt{action}, $\texttt{s}$) \\
\hspace{2em} \texttt{Depth}[$\texttt{s}$] $\leftarrow$ \texttt{Depth}[$\texttt{node}$] + 1 \\
\hspace{2em} add \texttt{s to successors} \\
\texttt{return} \texttt{successors}
3.4 Search Strategies

**Issues:**

- **Completeness:** if there's a solution, will strategy find it? (c.f. Gödel's Incompleteness Theorem; Perceptron Convergence Theorem)
- **Time complexity:** How long does it take?
- **Space complexity:** How much memory needed?
- **Optimality:** Will we get the best solution?
This can be implemented by having **INSERTALL** put the new nodes at the end of the list - *i.e.*, a queue (FIFO)
Breadth-First Search

- **Complete?** Yes (considers all paths of a given length)
- **Optimal?** Yes (always finds shortest path), provided path cost increases with path length
- **Time complexity:** \( O(b^d) \), for branching factor \( b \), depth \( d \)
- **Space complexity:** same
Breadth-First Search

- Memory requirements dominate (though 111 MB no big deal nowadays!)
- Time is still very expensive

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 millisecond</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>.1 seconds</td>
<td>11 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 minutes</td>
<td>111 megabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
<td>11 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 terabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3500 years</td>
<td>11,111 terabytes</td>
</tr>
</tbody>
</table>
• This can be implemented by having INSERTALL put the new nodes at the front of the list - *i.e.*, a stack (LIFO)
Depth-First Search

- **Complete?** No (can get stuck in loops or “infinite” paths)
- **Optimal?** No (best solution may be higher up in another branch)
- **Time complexity**: $O(b^m)$, for branching factor $b$, max depth $m$
- **Space complexity**: $O(bm)$ (versus $b^d$ for breadth-first, where $d = \text{depth of shallowest goal}$)
- **Conclusion**: Avoid for search trees with large or infinite maximum depth
Implementation: Python

- Search class with abstract `insertAll` (strategy) method
- `DepthFirstSearch` subclass implements `insertAll` as stack (insert at front of list)
- `BreadthFirstSearch` implements it as queue (insert at back of list)
- Don’t need to pass in fringe, just problem
- Data-structure translation example:
  \[
  \text{STATE}[s] \leftarrow \text{result} \text{ translates to } \text{s.state} = \text{result}
  \]
Implementation: LISP/Scheme (FYI)

• As in Python, use list for stack/queue
• Pass in strategy function as a parameter
• Use recursion instead of do-loop
Depth-limited Search

• Depth-first search with a depth limit
• Often know limit; e.g., for route, total number of cities
• Complete? Yes
• Optimal? No (best solution may be higher up in another branch)
• **Time complexity:** $O(b^l)$, for branching factor $b$, depth limit $l$
• **Space complexity:** $O(bl)$
Depth-limited Search

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

- Depth-limited search with increasing limits
- Complete? Yes
- Optimal? Yes
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

```python
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

Limit = 0

Limit = 1

Limit = 2

- \( Q \): Doesn't repetition waste time?
- \( A \): Not really: most nodes are at bottom.
  - So only slightly more expensive than depth-limited search.

- Doesn't repetition waste time?
- Not really: most nodes are at bottom.
- So only slightly more expensive than depth-limited search.
Uniform-Cost Search

- Expand “cheapest” node first
- So order fringe nodes by cost
- A subclass of “best-first” search
- Other famous best-first is A*
- Complete? Yes, assuming cost is well defined
- Optimal? Yes
- Time complexity: $O(b^x)$
- Space complexity: $O(b^x)$

Where $x$ is proportional to cost of optimal solution
# Comparing Search Strategies

<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>
Postscript: Avoiding Repeated States

• For some problems (8 Queens), not an issue
• Others (Missionaries & Cannibals; Mazes) are reversible and so can repeat states indefinitely.

• Even without reverses, can have exponential search from linear path:
Avoiding Repeated States: Solutions

• Don't return to a state that you just came from (cheap but not guaranteed).
• Don't create paths with cycles (more expensive, but better).
• Don't generate a state that's been generated before (most expensive, but guaranteed).
• If you do need to repeat a state (e.g., in maze), maintain a list of visited states and put non-visited states at front of fringe (e.g. by sorting).