Chapter 3

Solving Problems by Searching

Problem-Solving Agents

• Reflex agents cannot work well in those environments
  - state/action mapping too large
  - take too long to learn

• Problem-solving agent
  - is one kind of goal-based agent
  - decides what to do by finding sequences of actions that lead to desirable states
Problem Solving Agents (cont.)

- **Formulation**
  - Goal formulation (final state)
  - Problem formulation (decide what actions and states to consider)
- **Search** (look for solution i.e. action sequence)
- **Execution** (follow states in solution)

Assume the environment is **static, observable, discrete, deterministic**

---

Well-defined Problems and Solutions

- **A problem** can be defined by
  - Initial state: \( \text{In(Arad)} \)
  - Possible actions (or successor function)
    - \( \langle \text{Go(Sibiu)}, \text{In(Sibiu)} \rangle, \langle \text{Go(Timi)}, \text{In(Timi)} \rangle, \langle \text{Go(Zerind)}, \text{In(Zerind)} \rangle \)  
  - Goal test: \( \text{In(Bucharest)} \)
  - Path cost function
- **Step cost**: \( c(x, a, y) \) taking action \( a \) to go from state \( x \) to state \( y \)
- **Optimal solution**
  - the lowest path cost among all solutions
Example: Romania

Example Problems

- **Toy problems**
  Vacuum World, 8-Puzzle, 8-Queens Problem, Cryptarithmetic, Missionaries and Cannibals

- **Real-world problems**
  Route finding, Touring problems
  Traveling salesman problem,
  VLSI layout, Robot navigation,
  Assembly sequencing,
  Protein Design, Internet Searching
Vacuum World

- States:
  - agent location, each location might or might not contain dirt
  - \# of possible states = $2^n \cdot 2^2 = 8$
- Initial state: any possible state
- Successor function: possible actions (left, Right, Suck)
- Goal test: check whether all the squares are clean
- Path cost: the number of steps, each step cost 1
The 8-Puzzle

• States: location of each of the eight tiles and the blank tile
  \# of possible states = 9!/2 = 181,440  --- 9!/4 ???
• Initial state: any state
• Successor function: blank moves (left, Right, Up, Down)
• Goal test: check whether the state match as the goal configuration
• Path cost: the number of steps, each step cost 1

The 8-Queens

• Incremental formulation vs. complete-state formulation
  • States-I-1: 0-8 queens on board
  • Successor function-I-1:
    add a queen to any square
    \# of possible states = (64*63*…*57= 3*10^{14} )
  • States-I-2: 0-8 non-attacking queens on board
  • Successor function-I-2:
    add a queen to a non-attacking square in the left-most empty column
    \# of possible states = 2057  --- ???
• Goal test: 8 queens on board, none attacked
• Path cost: of no interest (since only the final state count)
Robotic Assembly

- **States:** real-valued coordinates of robot joint angles
  parts of the object to be assembled
- **Successor function:** continuous motions of robot joints
- **Goal test:** complete assembly
- **Path cost:** time to execute

Basic Search Algorithms

- How do we find the solutions of previous problems?
  - Search the state space (remember complexity of space depends on state representation)
  
  - Here: search through *explicit* tree generation
    ROOT = initial state.
    Nodes and leafs generated through successor function.
  
  - In general search generates a *graph* (same state through multiple paths)
Tree Search Algorithms

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

Simple Tree Search Example

function TREE-SEARCH(problem, strategy) return a solution or failure
Initialize search tree to the initial state of the problem
loop do
    if no candidates for expansion then return failure
    choose leaf node for expansion according to strategy
    if node contains goal state then return solution
    else expand the node and add resulting nodes to the search tree
end
Simple tree search example

function TREE-SEARCH(problem, strategy) return a solution or failure
  Initialize search tree to the initial state of the problem
  loop do
    if no candidates for expansion then return failure
    choose leaf node for expansion according to strategy
    if node contains goal state then return solution
    else expand the node and add resulting nodes to the search tree
  end

Determines search process!
State Space vs. Search Tree

- **State**: a (representation of) a physical configuration
- **Node**: a data structure belong to a search tree
  < State, Parent-Node, Action, Path-Cost, Depth >
- **Fringe**: contains generated nodes which are not yet expanded.

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
        node ← Remove-Front(fringe)
        if fringe is empty then return failure
        if Goal-Test(problem[State[node]]) then return Solution(node)
        fringe ← Insert(All(Expand(node, problem)), fringe)
    return 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
  - Completeness --- Is it guaranteed that a solution will be found (if one exists)?
  - Time complexity --- How long does it take to find a solution?
  - Space complexity --- How much memory is needed to perform a search?
  - Optimality --- Is the best solution found when several solutions exist?

- 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

use only the information available in the problem definition
do not use state information to decide the order on which nodes are expanded

- **Breadth-first search**
  - `Tree-Search(problem, FIFO-Queue())`
  - Expand the shallowest node in the fringe.
  - It is both optimal and complete.

- **Uniform-cost search**

- **Depth-first search**
  - Expand the deepest node in the fringe.
  - It is neither complete nor optimal. Why?

- **Depth-limited search**

- **Iterative deepening search**
  - Try all possible depth limits in depth limited search.
  - It is both optimal and complete.
Breadth-First Search

- Tree-Search(problem, FIFO-Queue())
- Fringe is a FIFO queue, i.e., new successors go at end
- Expand the shallowest unexpanded node

Analysis of Breadth-First Search

Complete?? Yes (if $b$ is finite)

Time?? $1 + b + b^2 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1})$
expand all but the last node at level $d$ i.e. exp. in $d$

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

Optimal?? Yes (if cost = 1 per step);
ot optimal in general (unless actions have different cost)
Analysis of Breadth-First Search (cont.)

- **Space** is the bigger problem (more than time)
  
  If \( d = 8 \), it will take 31 hours with 1 terabytes.

- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>NODES</th>
<th>TIME</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1100</td>
<td>0.11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>4</td>
<td>111100</td>
<td>11 seconds</td>
<td>106 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>( 10^7 )</td>
<td>19 minutes</td>
<td>10 gigabytes</td>
</tr>
<tr>
<td>8</td>
<td>( 10^9 )</td>
<td>31 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>( 10^{11} )</td>
<td>129 days</td>
<td>101 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>( 10^{13} )</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>( 10^{15} )</td>
<td>3523 years</td>
<td>1 exabyte</td>
</tr>
</tbody>
</table>

Uniform Cost Search

- Each node \( n \) has a path cost \( g(n) \).
- Expand *lowest path cost* unexpanded node.
- **Fringe** is queue ordered by path cost.
- It will find the *cheapest* solution provided that
  \[ \forall n. g(\text{Successor}(n)) \geq g(n) \]
  
  i.e. Every operator has a *nonnegative* cost.
- Equivalent to *breadth-first search* if step costs all equal.
Uniform Cost Search (cont.)

Analysis of Uniform Cost Search

**Complete??** Yes (if step cost >= \( \varepsilon \))

**Time??** \( O(b^{\lceil C^*/\varepsilon \rceil}) \)

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

**Space??** \( O(b^{\lceil C^*/\varepsilon \rceil}) \)

**Optimal??** Nodes expanded in increasing order of \( g(n) \)

Yes, if complete
Depth-First Search

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node

Depth-First Search (cont.-1)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node
Depth-First Search (cont.-2)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node

Depth-First Search (cont.-3)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node
Depth-First Search (cont.-4)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node

Depth-First Search (cont.-5)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node
Depth-First Search (cont.-6)

- *Tree-Search(problem, LIFO-Queue())*
- *Fringe* is a LIFO queue, 
  i.e., stack, put successors at front.
- Expand the *deepest* unexpanded node

![Depth-First Search Diagram](image1)

Depth-First Search (cont.-7)

- *Tree-Search(problem, LIFO-Queue())*
- *Fringe* is a LIFO queue,  
  i.e., stack, put successors at front.
- Expand the *deepest* unexpanded node

![Depth-First Search Diagram](image2)
Depth-First Search (cont.-8)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node

Depth-First Search (cont.-9)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node
Depth-First Search (cont.-10)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node

Depth-First Search (cont.-11)

- Tree-Search(problem, LIFO-Queue())
- Fringe is a LIFO queue, i.e., stack, put successors at front.
- Expand the deepest unexpanded node
Analysis of Depth-First Search

**Complete??**
- No: fails in indefinite-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 $d$
  (m: the maximum depth of any node,
  $d$: depth of the shallowest solution)
- But if solutions are dense, may be much faster than breadth-first

**Space??**
- $O(bm)$, i.e., linear space

**Optimal??**
- No

If $d = 12$, $b = 10$,
- Space: 10 petabytes for BFS; 118 KB for DFS

---

Analysis of Depth-First Search (cont.)

- **Time requirement:**
  - Goal at the far left: $d+1$
  - Goal at the far right: $\frac{b^{d+1} - 1}{b - 1}$
  - Average: (How is the average derived?)
    \[
    \frac{b^{d+1} + bd + b - d - 2}{2(b - 1)}
    \]
# Depth-limited Search

The unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit \( l \).

Failure (no solution) / Cutoff (no solution within the depth limit)

### Recursive implementation:

```plaintext
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
```

---

## Analysis of Depth-Limit Search

**Complete?** Yes, if \( l \geq d \)

if \( l \leq d \), the shallowest goal is beyond the depth limit.

**Time?** \( b^0 + b^1 + b^2 + \ldots + b^l = O(b^l) \)

**Space?** \( O(bl) \)

**Optimal?** No

DFS can be viewed as a special case of depth-limit search with \( l = \infty \).

**Diameter** of state space is a better depth limit, which leads to a more efficient depth-limit search e.g. diameter = 9, for the map of Romania of 20 cities
Iterative Deepening Search

function Iterative-Deepening-Search(problem) returns a solution
inputs: problem, a problem
for depth = 0 to \( \infty \) do
  result \leftarrow \text{Depth-Limited-Search}(\text{problem}, \text{depth})
  if result \neq \text{cutoff} then return result
end

Iterative Deepening Search (cont.-1)

Limit = 0

\[ \bullet \]
Iterative Deepening Search (cont.-2)

Limit = 1

Iterative Deepening Search (cont.-3)

Limit = 2
Iterative Deepening Search (cont-4)

Analysis 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.

If $b = 10$, $d = 5$,

- $N_{max} = 1 + 10 + 100 + 1,000 + 10,000 + 999,990 = 1,111,101$
- $N_{max} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
- $N_{max} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

Iterative deepening is faster than breadth-first search, despite the repeated generation of states.

Overhead = (123,456 - 111,111)/111,111 = 11%

Iterative deepening is the preferred uninformed search when there is a large search space and the depth of the solution is not known.
Bidirectional Search

- Search forward from the initial state.
  Generate successors to the current node.
- Search backward from the goal.
  Generate predecessors to the current node.
- Stop when two searches meet in the middle.

Bidirectional Search (cont.-1)

- Two simultaneous searches from start an goal.
  - Motivation: \( b^{d/2} + b^{d/2} \neq b^d \)
- Check whether the node belongs to the other fringe before expansion.
- Space complexity is the most significant weakness.
- Complete and optimal if both searches are BF.
Bidirectional Search (cont.-2)

Issues:
- If all operators are reversible, \(\text{predecessor}(n) = \text{successor}(n)\)
- Multiple goal states
- Cost of checking if a node exists
- Search strategy for each half?

Comparing Uninformed 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>(\frac{C^{*}}{c})</td>
<td>(b)</td>
<td>(b)</td>
<td>(b)</td>
</tr>
<tr>
<td>Space</td>
<td>(\frac{b^{d+1}}{c})</td>
<td>(\frac{C^{*}}{c})</td>
<td>(b)</td>
<td>(bd)</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>

- \(b\): the branching factor
- \(d\): the depth of the shallowest solution
- \(m\): the maximum depth of the search tree
- \(l\): the depth limit
Comparing Uninformed Search Strategies (cont.)

Issues considered in selecting search strategies:
- size of the search space
- depth of the solution
- solution density
- finite vs. infinite depth
- any vs. all solutions
- optimality?
- predecessors?

Avoid Repeated States

- Do not return to the previous state.
- Do not create paths with cycles.
- Do not generate the same state twice.
  - Store states in a hash table.
  - Check for repeated states.
Graph Search

- To modify “Tree Search Algorithm” by adding
  closed list --- to store every expanded node

```
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 node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
    end
```