Artificial Intelligence

3. Solving Problems By Searching

**Definition**

- **Goal Formulation**
  - Given situations we should adopt the "goal"
  - 1st step in problem solving!
  - It is a set of world states, only those in which the goal is satisfied
  - Action causes transition between world states

- **Problem Formulation**
  - Process of deciding what actions and states to consider, and follows goal formulation

---

**Formulation of simple problem-solving agent**

Function \textsc{simple-prob-solv-agent}(p) returns an action

![Formulation of simple problem-solving agent](image)

---

**Examples (1) traveling**

On holiday in Taif

- **Formulate Goal**
  - Be after two days in Paris

- **Formulate Problem**
  - States: various cities
  - Actions: drive/fly between cities

- **Find Solution**
  - Sequence of cities: e.g., Taif, Jeddah, Riyadh, Paris

---

**Examples (2) Vacuum World**

- 8 possible world states
- 3 possible actions: Left/Right/ Suck
- Goal: clean up all the dirt= state(7) or state(8)

- World is accessible ➔ agent’s sensors give enough information about which state it is in (so, it knows what each of its action does), then it calculate exactly which state it will be after any sequence of actions. \textit{Single-State problem}

- World is inaccessible ➔ agent has limited access to the world state, so it may have no sensors at all. It knows only that initial state is one of the set \{1,2,3,4,5,6,7,8\}. \textit{Multiple-States problem}

---

**Problem types**

- Deterministic, accessible ➔ single-state problem
- Deterministic, inaccessible ➔ multiple-state problem
- Non-deterministic, inaccessible ➔ contingency problem
- Unknown state space ➔ exploration problem ("online")
Problem Definition

- **Initial state**: description of an action
- **Operator**: all states reachable from the initial state by any sequence action
- **State space**: sequence of actions leading from one state to another
- **Goal test**: which the agent can apply to a single state description to determine if it is a goal state
- **Path cost function**: assign a cost to a path which is the sum of the costs of the individual actions along the path.

Vacuum World

- **Single state**: start in #5. Solution?
- **Multiple state**: start in {1,2,3,4,5,6,7,8} e.g., Right goes to {2,4,6,8}. Solution?
- **Contingency**: start in #5
  - Murphy’s Law: Suck can dirty a clean carpet
  - Local sensing: dirt location only
  - Solution?

States: S1, S2, S3, S4, S5, S6, S7, S8
- **Operators**: Go Left, Go Right, Suck
- **Goal test**: no dirt left in both squares
- **Path Cost**: each action costs 1.

Real-world problems

- Routine finding
  - Routing in computer networks
  - Automated travel advisory system
  - Airline travel planning system
    - Goal: the best path between the origin and the destination
- Travelling Salesperson problem (TSP)
  - Is a famous touring problem in which each city must be visited exactly once.
  - Goal: shortest tour

Data Structure for Search Tree

- **DataType Node**: data structure with 5 components
- **Components**:
  - State
  - Parent-node
  - Operator
  - Depth
  - Path-cost
**Search Strategies**

The strategies are evaluated based on 4 criteria:

1. **Completeness**: always find solution when there is one
2. **Time Complexity**: how long does it take to find a solution
3. **Space Complexity**: how much memory does it need to perform the search
4. **Optimality**: does the strategy find the highest-quality 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?

- Contingency
  - Nondeterministic: Suck may dirty a clean carpet
  - Partially observable: location, dirt
  - Percept: \([L, \text{Clean}]\), i.e., start in #
  - Solution? (Right, if dirt then Suck)

---

**Example: vacuum world**

- Sensorless, start in \(1,2,3,4,5,6,7,8\) e.g., Right goes to \(2,4,6,8\)
  - Solution?

- Contingency
  - Nondeterministic: Suck may dirty a clean carpet
  - Partially observable: location, dirt
  - Percept: \([L, \text{Clean}]\), i.e., start in #
  - Solution? (Right, if dirt then Suck)

---

**Single-state problem formulation**

A problem is defined by four items:

1. **Initial state** e.g., "at Arad" (initial, start state)
2. **Successor function** \(S(x) = \text{set of action–state pairs} \)
   - e.g., \(S(\text{Arad}) = \{\text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rightarrow \ldots\} \)
3. **Goal test**, Can be
   - \(x \in \text{goal}\)
   - \(x = \text{goal} \)
   - \(x = \text{\textit{goal}} \)
   - \(x = \text{Endgame}(x)\)
4. **Path cost** (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - \(c(x,y)\) 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: Left, Right, Suck
- goal test? no dirt at all locations
- path cost? 1 per action

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: robotic assembly

- states? of the object to be assembled
- actions? continuous motions of robot joints
- goal test? complete assembly
- 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)

```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
```

Example: Romania

Implementation: general tree search

```plaintext
Function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe = INITIAL-STATE(problem)
loop do
  If fringe is empty then return failure
  node = REMOVE-FRONT(fringe)
  If GOAL-TEST(node) then return SOLUTION(node)
  fringe = EXPAND(node, problem, fringe)
end loop
```

Function EXPAND(node, problem, fringe) returns a set of nodes

- successors = the empty set
- For each action, result in SUCCESSORS(node, action, problem) do
  - a new NODE
  - Precond(node) = node, Action = action, Result = result
  - Path-Cost = Path-Cost(node) + Step-Cost(node, action, a)
  - Depth = Depth(node) + 1
  - Add node to successors
return successors
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.
- 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

Properties of breadth-first search

- Complete? Yes (if $b$ is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^{d+1})$
- 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)

Breadth-first search tree sample

Branching factor: number of nodes generated by a node parent (we called here "b")

Here after $b=2$

Breadth First Algorithm

```c
void breadth () {
    queue = [];//initialize the empty queue
    state = root_node;//initialize the start state
    while (! is_goal (state)) {
        add_to_back_of_queue (successors(state));
        if queue == []
            return FAILURE;
        state = queue[0];//state=first item in queue
        remove_first_item_from (queue);
    }
    return SUCCESS;
}
```

Breadth First Complexity

- The root \( \rightarrow \) generates \( b \) new nodes
- Each of which \( \rightarrow \) generates \( b \) more nodes
- So, the maximum number of nodes expended before finding a solution at level \( "d" \), it is : \( 1 + b + b^2 + b^3 + \ldots + b^d \)
- Complexity is exponential = \( O(b^d) \)

Time and memory requirement in Breadth-first

<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 sec</td>
<td>1 Kb</td>
</tr>
<tr>
<td>4</td>
<td>11.111</td>
<td>11 sec</td>
<td>1 Mb</td>
</tr>
<tr>
<td>6</td>
<td>10^4</td>
<td>18 minutes</td>
<td>111 Mb</td>
</tr>
<tr>
<td>8</td>
<td>10^8</td>
<td>31 hours</td>
<td>11 GB</td>
</tr>
<tr>
<td>10</td>
<td>10^10</td>
<td>128 days</td>
<td>1 Tb</td>
</tr>
<tr>
<td>12</td>
<td>10^12</td>
<td>35 years</td>
<td>111 Tb</td>
</tr>
<tr>
<td>14</td>
<td>10^14</td>
<td>3500 years</td>
<td>111.111 Tb</td>
</tr>
</tbody>
</table>

Assume branching factor $b=10$, 1000 nodes explored/sec and 100 bytes/node
Example (Routing Problem)

A

B

C

SG

1

5

15

10

Solution of the Routing problem using Breadth-first search

Sol= empty & Cost = infinity

New solution found better than the current

Sol= {S,B,G} & Cost = 10

C will not be expanded as its cost is greater than the current solution

Solution of the Routing problem using Uniform Cost search

Sol= empty & Cost = infinity

New solution found better than the current

Sol= {S,B,G} & Cost = 10

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 ≥ ε
- Time? # of nodes with g ≤ cost of optimal solution, O(b^{ceiling(C*/ε)})
- Space? # of nodes with g ≤ cost of optimal solution, O(b^{ceiling(C*/ε)})
- 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
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-first search tree sample
Depth First Complexity

- Let $b$: is the branching factor
- Let $m$: maximum depth to find solution
- So, the maximum number of nodes expended before finding a solution at level "m", it is:
  \[ 1 + b + b^2 + \ldots + b^m \]
- Memory need = $b^m$
- Complexity in worst case = $O(b^m)$ as "Breadth-First"
- Complexity in best case = $O(b^m)$ which is excellent!

Depth First Algorithm

```c
void depth ()
{
    queue[];
    //initialize the empty queue
    state = root_node;
    //initialize the start state
    while (!Is_goal( state ))
    {
        if queue == []
            return FAILURE;
        state = queue[0];
        //state=first item in queue
        remove_first_item_from (queue);
    }
    return SUCCESS;
}
```

Time and memory requirement in Depth-first

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time (best case)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 milliseck</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>0.02 sec</td>
<td>2 Kb</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>0.04 sec</td>
<td>4 Kb</td>
</tr>
<tr>
<td>6</td>
<td>10 * 6</td>
<td>0.06 sec</td>
<td>6 Kb</td>
</tr>
<tr>
<td>8</td>
<td>10 * 8</td>
<td>0.08 sec</td>
<td>8 Kb</td>
</tr>
<tr>
<td>10</td>
<td>10 * 10</td>
<td>0.1 sec</td>
<td>10 Kb</td>
</tr>
<tr>
<td>12</td>
<td>10 * 12</td>
<td>0.12 sec</td>
<td>12 Kb</td>
</tr>
<tr>
<td>14</td>
<td>10 * 14</td>
<td>0.14 sec</td>
<td>14 Kb</td>
</tr>
</tbody>
</table>

Assume branching factor $b=10$; 1000 nodes explored/sec and 100 bytes/node.

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 → single-state problem
- Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
- Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
- Percepts provide new information about current state
- Often interleaves search, execution
- Unknown state space → exploration problem

Depth-limited search

= depth-first search with depth limit $l$, i.e., nodes at depth $l$ have no successors

Recursive implementation:

```c
Recursive Depth-Limited Search(problemp, limit) returns value of path if
Recursive(DLSearch(Make-Problem(Make-InitialState(), problem), limit))

Recursive(Recursive DLSearch(problemp, limit) returns value of path if
result = Recursive DLSearch(solution, limit) if result = Success then return result
else if result = Failure then return result
else if limit = 0 then return Failure
else if result = Success then return result
else if limit > 0 then return Recursive DLSearch(solution, limit)
```
Iterative deepening search

Function Iterative-Deepening-Search(problem) returns a solution, or failure.
Input: problem, a problem
for depth = 0 to ∞ do
    result = Deep-Limited-Search(problem, depth)
if result ≠ cutoff then return result

Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:
$$NDLS = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
$$NIDS = (d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d$$

For $b = 10$, $d = 5$,
- $NDLS = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
- $NIDS = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
- Overhead = $(123,456 - 111,111)/111,111 = 11\%$
Properties of iterative deepening search

- **Complete?** Yes
- **Time?** \((d+1)b^0 + d b^1 + (d-1)b^2 + ... + b^d = O(b^d)\)
- **Space?** \(O(bd)\)
- **Optimal?** Yes, if step cost = 1

Summary of algorithms

<table>
<thead>
<tr>
<th>Criteria</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))</td>
<td>(O(b^m))</td>
<td>(O(b^d))</td>
<td>(O(b^d))</td>
<td>(O(b^d))</td>
</tr>
<tr>
<td>Space</td>
<td>(O(b^d))</td>
<td>(O(b^m))</td>
<td>(O(bm))</td>
<td>(O(bm))</td>
<td>(O(bm))</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 GraphSearch(problem, fringe) returns a solution, or failure
   fringe = Insert(Make-Nodes(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
         for each state in Expand(node, problem) do
            fringe = Insert(Add-State(state, fringe), fringe)
   end loop
```

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