Problem Solving Search

Here’s your chapter with improved formatting for better readability:

Chapter 3: Problem Solving Agents

Dr. Ammar Masood

Department of Cyber Security, Air University Islamabad

Contents

Problem Solving as Search

One way to address these issues is to view goal attainment as problem solving, and to see problem solving as a search through a state space.

For example, in chess, a *state* is a board configuration.

Important Concepts in Problem Solving

- The solution to any problem is a fixed sequence of actions.
- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A **search algorithm** takes a problem as input and returns a solution in the form of an action sequence.

- Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase**.
- After formulating a goal and a problem to solve, the agent calls a **search procedure** to solve it.

Problem Solving by Searching

A problem can be formally defined by five components:

1. **Initial state** – The state the agent starts in.
2. **Actions** – A description of the possible actions available to the agent.
3. **Transition model** – Describes what each action does.
4. **Goal test** – Determines whether a given state is a goal state.
5. **Path cost function** – Assigns a numeric cost to each path.

A Simple Problem-Solving Agent

Algorithm:

sql

```sql
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent:
        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
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
        if seq = failure then
```

```
            return a null action
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

End Goal?

- A solution to a problem is an **action sequence** that leads from the initial state to a goal state.
- **Solution quality** is measured by the **path cost function**, and an **optimal solution** has the lowest path cost among all solutions.

Example Problems

Toy Problems (Illustrate problem-solving methods)

- **Concise, exact description**
- Can be used to compare performance
- Examples:
  - 8-Puzzle
  - 8-Queens Problem
  - Cryptarithmetic
  - Vacuum World
  - Missionaries and Cannibals
  - Simple Route Finding

Real-World Problems (Practical application)

- More difficult, often without a single, agreed-upon description.
- Examples:
  - Route Finding
  - Touring and Traveling Salesperson Problem (TSP)
  - VLSI Layout

- Robot Navigation
- Assembly Sequencing

Problem Formulation

- **Toy Problem**: Designed for research and algorithm comparison.
- **Real-World Problem**: Solutions have practical applications but often lack a single precise description.

Example: Vacuum World

- **States**: Determined by agent location and dirt locations.
- **Initial State**: Any state can be designated as the initial state.
- **Actions**: Left, Right, Suck (Up/Down for larger environments).
- **Transition Model**: Defines effects of actions.
- **Goal Test**: All squares must be clean.
- **Path Cost**: Each step costs 1.

Real-World Problems

Route Finding

- Defined by **locations** and **transitions** between them.
- Applications:
  - Routing in computer networks
  - Automated travel advisory systems
  - Airline travel planning

Traveling Salesperson Problem (TSP)

- **Goal**: Find the shortest tour that visits all cities.

- **Challenges**:
  - Requires keeping track of visited cities.
  - **NP-hard** problem; extensive research exists to optimize solutions.

Example: Romania

- You are on holiday in Romania, currently in Arad.
- Your **goal**: Reach Bucharest before your flight tomorrow.

Problem Formulation

- **States**: Various cities.
- **Actions**: Drive between cities.
- **Solution**: A sequence of cities, e.g., *Arad → Sibiu → Fagaras → Bucharest*.

What is a Solution?

- A **sequence of actions** that transforms the initial state into a goal state.
- Sometimes, the solution is just the goal state itself (e.g., molecular structure inference from mass spectrometry).

State Space Representation

- **The real world is complex** → We must abstract the state space for problem-solving.
- **Abstract State** = Set of real states.
- **Abstract Action** = Complex combination of real actions.
- **Abstract Solution** = A set of real paths that are solutions in the real world.

Search Concepts

Expanded Nodes vs. Frontier

- **Frontier**: Set of nodes that have been reached but not yet expanded.
- **Interior**: Nodes that have been expanded.
- **Exterior**: States that have not been reached.

Example: The 8-Puzzle

- **States**: Locations of tiles.
- **Actions**: Move left, right, up, down.
- **Goal Test**: Check if the state matches the goal state.
- **Path Cost**: Each move costs 1.

Search Elements & Structure

Search Tree Elements

- **Root Node**: Starting state of the problem.
- **Branches (Edges)**: Possible actions leading to new states.
- **Nodes**: Represent states in the search space.
- **Leaf Nodes**: Terminal states with no further expansion.
- **Goal Node**: A leaf node that satisfies the goal condition.
- **Path Cost**: Cumulative cost of reaching a node from the root.

Search Algorithm Basics

- The **search tree** is built from the initial state, with actions forming branches.
- The **search strategy** determines the order in which states are expanded.

Search Algorithm Infrastructure

- **Each node (n) contains:**

- **n.STATE**: The state in the search space.
- **n.PARENT**: The parent node in the search tree.
- **n.ACTION**: The action applied to generate the node.
- **n.PATH-COST**: The cost of reaching this node.

Implementation: States vs. Nodes

- A **state** represents a physical configuration.
- A **node** is part of the search tree, containing state, parent node, action, path cost, and depth.
- The **Expand function** generates new nodes based on the given state.

Search Strategies

Defining a Search Strategy

- **A search strategy determines the order of node expansion.**
- **Evaluation Criteria:**
  1. **Completeness** – Does it always find a solution if one exists?
  2. **Time Complexity** – Number of nodes generated.
  3. **Space Complexity** – Maximum number of nodes in memory.
  4. **Optimality** – Does it always find the least-cost solution?

Complexity Notation

- **b** = Maximum branching factor of the search tree.
- **d** = Depth of the least-cost solution.
- **m** = Maximum depth of the state space (possibly infinite).

This refined format keeps all original content intact while improving readability for exam preparation. Let me know if you need any additional edits! 🚀

Here's your content with improved formatting:

Problem Solving by Searching

Contents

- Problem Solving by Searching
- Problem Formulation
- Search Strategies
- Uninformed Search Strategies

Uninformed Search Strategies

Uninformed search strategies explore the search space without prior knowledge about the goal's location, only using the problem definition.

Types of Uninformed Search:

1. **Breadth-First Search (BFS)**
2. **Uniform Cost Search (UCS)**
3. **Depth-First Search (DFS)**
4. **Depth-Limited Depth-First Search (DLS)**
5. **Iterative Deepening Depth-First Search (IDDFS)**

Breadth-First Search (BFS)

- Explores all nodes at the current depth before moving deeper.
- Uses a **queue (FIFO)** for node expansion.

- **Guaranteed** to find the shortest path in an unweighted graph.
- **High memory consumption** as it stores all nodes at a given depth.

Properties of BFS:

- **Branching factor (b):** Number of successors per node.
- **Level progression:** Root → b nodes → b^2 nodes → b^3 nodes → ... → bd nodes
- **Total nodes at depth d:**

$$1 + b + b^2 + b^3 + \ldots + b^d$$

- **Time and Space Complexity:** O(b^d)
- **Completeness:** Yes (if b is finite)
- **Optimality:** Yes (if cost = 1 per step)
- **Space Complexity:** The major issue (higher than time complexity).

Real-World Example Breakdown

- **Branching factor (b) = 10**
- **Processing speed:** 1 million nodes/sec
- **Memory per node:** 1 KB
- **At depth d = 10:**
  - Time: < 3 hours
  - Memory required: **10 TB**
- **At depth d = 14:**
  - Processing time ≈ **3.5 years**

Challenges:

- **Exponential complexity** makes BFS impractical for:
  - Large datasets
  - Deep searches
  - High branching factors

Solution:

Use **informed search strategies** for real-world applications.

Uniform Cost Search (UCS)

- Also called **Dijkstra's Algorithm** in theoretical computer science.
- Unlike BFS, which expands nodes in increasing depth order, UCS expands nodes in order of **path cost**.

Depth-First Search (DFS)

- Explores as deep as possible along a branch before backtracking.
- Uses a **stack (LIFO)** for node expansion.

Properties of DFS:

- **Completeness:** No (fails in infinite-depth spaces or loops).
  - Solution: Modify to avoid repeated states → complete in finite spaces.
- **Optimality:** No (not guaranteed to find the shortest path).
- **Time Complexity:** O(b^m), where m = max depth.
- **Space Complexity:** O(bm)

Backtracking Optimization:

- Instead of generating all successors at once, **backtracking** generates **one successor at a time**.
- Reduces memory usage to **O(m)** instead of **O(bm)**.

Depth-Limited Depth-First Search (DLS)

- **DFS with a predefined depth limit (L).**
- Prevents infinite loops by stopping at the depth limit.
- If the goal is beyond L, it **may fail**.

Complexity:

- **Time Complexity:** O(b^L)
- **Space Complexity:** O(bL)

Example Use Case:

- On a **map of Romania**, there are **20 cities**.
- If a solution exists, the longest path must be **at most 19 steps**.
- **Setting L = 19** ensures completeness.

Iterative Deepening Depth-First Search (IDDFS)

- Repeatedly applies **DLS with increasing depth limits**.
- Combines **DFS's low memory use** with **BFS's completeness**.
- **Guaranteed to find the shortest path.**

Complexity:

- **Time Complexity:** O(b^d) (some nodes are re-explored but remains efficient).
- **Space Complexity:** O(bd)

When is IDDFS Used?

- Commonly applied in **large search spaces** where BFS's memory usage is impractical.

IDDFS Algorithm:

- Repeatedly applies **depth-limited search** with **increasing limits**.
- Terminates when:
  1. A solution is found.
  2. The depth-limited search returns failure (no solution exists).

Node Generation Comparison:

- **Depth-Limited Search (DLS) to depth d with branching factor b:**

$$N_{DLS} = 1 + b + b^2 + ... + b^{d-1} + b^d$$

- **Iterative Deepening Search (IDS) to depth d:**

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

- **For b = 10, d = 5:**
  - **DLS nodes generated:** 111,111
  - **IDS nodes generated:** 123,456
  - **Overhead:** 11%

Properties of IDDFS:

- **Completeness:** Yes
- **Time Complexity:** O(b^d)
- **Space Complexity:** O(bd)
- **Optimality:** Yes (if step cost = 1)

Bidirectional Search

- Searches **simultaneously from the start** and **from the goal**, hoping the two searches meet.
- **Significant reduction in search complexity:**
  - Instead of O(b^d), it requires **O(b^(d/2) + b^(d/2))**, which is significantly smaller.

Example:

- Find a path from vertex **0 to 14**.
- Two searches are executed:
  - One from vertex **0**
  - One from vertex **14**
- When they meet at **vertex 7**, the search terminates.

Bidirectional Best-First Search

- Maintains **two frontiers** and **two tables of reached states**.

- The node expanded **next** is the one with the **minimum evaluation function** from either frontier.
- **Once paths meet, they are joined to form a solution.**
- **The first solution found is not necessarily the best**—search continues until termination criteria are met.

Summary

- **Problem formulation** involves abstracting real-world details into a **state space**.
- **Uninformed search strategies** explore the search space without **prior knowledge** of the goal.
- **Iterative deepening search** is often the best uninformed search method:
  - Uses **only linear space**
  - Requires **not much more time** than other uninformed algorithms.

This version keeps all the content intact but makes it **well-structured, readable, and easy to review** for your exam preparation. Let me know if you need further refinements! 🚀