Chapter 2: State Space Search for Problem Solving

State space search is a foundational concept in Artificial Intelligence (AI) that allows us to formalize and solve problems by exploring possible states and transitions. In this chapter, we start with the basic principles and gradually move to more advanced topics. Whether you're new to the field or aiming to deepen your understanding, this chapter covers all essential aspects—from problem representation to detailed analyses of search algorithms.

2.1 Introduction

2.1.1 What Is State Space Search?

State space search is the process of exploring all possible configurations (or states) of a problem until a solution is found. In this context, a state is a complete description of a particular situation. For example, in a maze, a state might describe your current location; in a puzzle, it might represent the arrangement of pieces.

2.1.2 Why Is It Important?

- **Fundamental to AI:**
  Many AI systems solve problems—such as planning moves in a game, route finding, or scheduling—by exploring a state space.
- **General-Purpose:**
  With a proper representation, the same search algorithms can be applied to a wide variety of problems.
- **Types of Problems:**
  - **Configuration Problems:** Find a state that meets specific conditions (e.g., N-Queens, Sudoku, map colouring).
  - **Planning Problems:** Find a sequence of moves that transforms the start state into a goal state (e.g., the water jug problem or route finding).

2.2 Representing Problems in a State Space

2.2.1 Elements of a State Space

- **State:**
  A complete description of a situation (e.g., a list representing the colours of regions on a map or the water levels in jugs).
- **Operators (Moves):**
  Actions that transition one state to another. These are captured by a function called **MoveGen**.
- **State Space Graph (or Tree):**
  The implicit graph formed by states (nodes) and operators (edges).
- **Goal Test:**
  A function (called **GoalTest**) that checks whether a state satisfies the problem's goal.

2.2.2 Example: Map Colouring Problem

Imagine a map with regions labeled A, B, C, D, and E. Each region must be coloured such that no two adjacent regions have the same colour.

- **Representation:**
  Each state is an assignment of colours to regions.

- **MoveGen:**
  A function that, given a partially coloured map, returns new states by assigning a colour to one uncoloured region.
- **GoalTest:**
  Checks if every region is coloured and adjacent regions have different colours.

This problem can also be viewed as a constraint satisfaction problem (CSP), but here we focus on its formulation as a state space search.

2.3 Example Problems in State Space Search

2.3.1 The Water Jug Problem

- **Problem Statement:**
  You have three jugs with capacities 8, 5, and 3 liters. The start state is (8, 0, 0), meaning the 8-liter jug is full, and the others are empty. The goal is to measure, for example, 4 liters in one of the jugs.
- **Operators:**
  Moves include pouring water from one jug to another until the source is empty or the destination is full.
- **State Representation:**
  Represented as a triple, e.g., (8, 0, 0), (3, 5, 0), etc.
- **GoalTest:**
  Returns true if any jug has exactly 4 liters.

2.3.2 The Eight Puzzle

- **Problem Statement:**
  A 3×3 grid with eight numbered tiles and one blank space. A tile adjacent to the blank space can slide into it.
- **Operators:**
  Moves are typically labeled R (right), L (left), U (up), and D (down).
- **State Representation:**
  A configuration of the grid.
- **GoalTest:**
  Checks whether the tiles are arranged in the desired order.

2.3.3 The Man, Goat, Lion, Cabbage (MGLC) Problem

- **Problem Statement:**
  A man must transport a lion, a goat, and a cabbage across a river using a boat that can carry only one item at a time, without leaving the goat with the lion or the cabbage with the goat.
- **State Representation:**
  This can be represented as two lists (one for each bank) or as a description of which side each object is on.
- **Operators:**
  Moves consist of the man moving alone or with one item.
- **GoalTest:**
  Checks if all items have safely been transferred to the other side.

2.3.4 Other Problems

Similar state space formulations apply to:

- **N-Queens Problem:** Place queens on an (N \times N) board so that no two queens attack each other.

- **Traveling Salesman Problem:** Find the shortest tour that visits each city exactly once.
- **Maze Navigation:** Find a path through a maze from a start point to an exit.

2.4 General Purpose State Space Search Methods

2.4.1 Domain-Independent Search Algorithms

Rather than writing a custom program for each problem, we develop general algorithms that accept a domain description via functions such as MoveGen and GoalTest. This allows you to “plug in” different problems into the same search framework.

2.4.2 OPEN and CLOSED Lists

- **OPEN (Frontier):**
  A set (or list) of candidate states to be explored next.
- **CLOSED (Visited):**
  A set (or list) of states that have already been explored to avoid loops.
- **Data Structures:**
  Lists (or queues/stacks) and tuples (or node pairs) are used to represent nodes and track parent pointers for path reconstruction.

2.5 Uninformed (Blind) Search Algorithms

2.5.1 Simple Search (Generate and Test)

A basic approach where you:

1. Start from the initial state.
2. Generate all possible successors using MoveGen.
3. Test each successor with GoalTest.
4. Continue until a goal state is found or the state space is exhausted.

Issues: Without proper loop checking (i.e., using CLOSED), the algorithm may fall into infinite loops.

2.5.2 Depth-First Search (DFS)

- **Approach:**
  DFS explores as deeply as possible along one branch before backtracking.
- **Implementation:**
  Uses a **stack** (OPEN treated as a stack) so that the newest node is expanded first.
- **Pseudocode Outline:**
  1. Initialize OPEN with the start state (with parent pointer null).
  2. While OPEN is not empty, pick the node at the head; if it passes GoalTest, reconstruct and return the path; otherwise, add its unvisited neighbors (after checking with CLOSED) to OPEN.
- **Properties:**
  - Space: Linear
  - May get stuck in deep paths or infinite loops if the state space is infinite.

2.5.3 Breadth-First Search (BFS)

- **Approach:**
  BFS explores the state space level by level.

- **Implementation:**
  Uses a **queue** (OPEN treated as a FIFO list) so that the oldest node is expanded first.
- **Pseudocode Outline:**
  1. Initialize OPEN with the start state.
  2. While OPEN is not empty, remove the node at the head; if it is the goal, reconstruct and return the path; otherwise, add its neighbors (not already in OPEN or CLOSED) to the end of OPEN.
- **Properties:**
  - Space: Exponential
  - Guarantees finding the shortest path (in terms of number of moves)

2.5.4 Variations and Case Studies

Different variations exist depending on whether nodes already in CLOSED or OPEN are re-added or pruned. These variations affect the generated search tree and the efficiency of the algorithm.

2.6 Depth-First Iterative Deepening (DFID)

2.6.1 Motivation

DFID combines the space efficiency of DFS with the optimality (shortest path guarantee) of BFS. It repeatedly performs depth-bounded DFS, increasing the depth bound gradually until a goal is found.

2.6.2 Depth-Bounded DFS

- **Concept:**
  Perform DFS only up to a certain depth (d). If the goal is not found, increment (d) and restart the search.
- **Node Representation:**
  Use a triple: (node, parent, depth).

2.6.3 DFID Algorithm Overview

1. Set an initial depth bound (e.g., 0 or 1).
2. Perform depth-first search with that bound.
3. If the goal is found, return the corresponding path.
4. If not, increment the bound and repeat.
5. Use ancillary functions such as removeseen (to avoid revisiting nodes) and makePairs (to store parent pointers).

2.6.4 Loop Checking and Path Reconstruction

- **Loop Checking:**
  When using a CLOSED list to prune nodes, DFID may sometimes miss shorter paths. One solution is to allow re-expansion of nodes even if they appear in CLOSED, while still storing parent pointers.
- **Path Reconstruction:**
  To reconstruct the path, you can either store the entire path with each node or maintain node pairs (current node, parent) and trace back once the goal is reached.

2.6.5 Analysis and Trade-Offs

- **Extra Work:**
  DFID re-examines nodes at shallower depths in each iteration. For large branching factors, this extra work is only a constant factor more than BFS.

- **Space Efficiency:**
  DFID uses linear space (like DFS) while guaranteeing completeness and optimality in finite state spaces.

2.7 Node Representation and Data Structures

2.7.1 Node Pairs and Triples

- **Node Pairs:**
  Represent each state as a pair (current state, parent state) to help reconstruct the path.
- **Node Triples:**
  For DFID, a triple (state, parent, depth) is used to track the node's depth.

2.7.2 Lists and Tuples

- **Lists:**
  Used for the OPEN and CLOSED lists. Operations such as "head" (first element), "tail" (remaining elements), and concatenation (using colon for cons and plus-plus for append) are fundamental.
- **Tuples:**
  Used for representing node pairs/triples. Built-in functions like first, second, and third allow access to tuple elements.

2.7.3 Tie-Breaking and Node Order

The order in which nodes are added to and removed from OPEN (stack for DFS, queue for BFS) determines the search's behavior. Tie-breaking rules can affect node order, which is especially important in automated grading and precise algorithm implementations.

2.8 Analysis of Search Algorithms

2.8.1 Time Complexity

- **Exponential Growth:**
  With a branching factor (b) and a goal at depth (d), the worst-case number of nodes inspected is (O(b^d)) for both DFS and BFS.
- **DFID Overhead:**
  Although DFID repeats work for each depth level, the extra work is only a small constant factor compared to BFS for large (b).

2.8.2 Space Complexity

- **DFS:**
  Requires only linear space proportional to the maximum depth.
- **BFS:**
  Requires exponential space, proportional to the number of nodes at the deepest level.
- **DFID:**
  Retains the linear space advantage of DFS.

2.8.3 Quality and Completeness

- **BFS:**
  Guarantees finding the shortest path (optimal solution) and is complete if a solution exists in a finite state space.

- **DFS:**
  May not find the shortest path and can fail on infinite state spaces.
- **DFID:**
  Combines the advantages of DFS and BFS by guaranteeing the shortest path (if one exists) while using linear space.

2.8.4 Combating Combinatorial Explosion

The rapid exponential growth of the search tree—known as combinatorial explosion—can be mitigated through:

- **Effective Pruning:** Using CLOSED lists to avoid re-expansion.
- **Heuristics:** Later topics will introduce heuristic search to guide the search more effectively.
- **Iterative Deepening:** DFID limits memory usage while ensuring all nodes up to a given depth are explored.

2.9 Summary and Conclusion

In this chapter, we have explored the fundamentals of state space search in AI. We began by defining the concept of a state space, key functions (MoveGen and GoalTest), and how problems—such as map colouring, the water jug problem, the eight puzzle, and the man, goat, lion, cabbage problem—are represented as search problems.

We then detailed general search methods that are domain independent, using OPEN (frontier) and CLOSED (visited) lists to manage the search process. We covered two classic uninformed search algorithms:

- **Depth-First Search (DFS):**
  Uses a stack, is space efficient, but may get trapped in deep or infinite paths and does not guarantee the shortest path.

- **Breadth-First Search (BFS):**
  Uses a queue, guarantees the shortest path, and is complete for finite state spaces but requires exponential space.

Next, we introduced **Depth-First Iterative Deepening (DFID),** which repeatedly performs depth-limited DFS with increasing depth bounds. DFID combines the linear space requirements of DFS with the optimality and completeness of BFS.

We also discussed node representation using pairs and triples, as well as the use of lists and tuples for implementing these algorithms. Finally, we analyzed the trade-offs in time and space complexity, solution quality, and completeness, and discussed strategies for managing combinatorial explosion.

2.10 Exercises and Further Reading

1. **Implementation Tasks:**
   - Write a **MoveGen** function for the Water Jug problem (state represented as a triple).
   - Write a **GoalTest** function that returns true when any jug contains exactly 4 liters.
2. **Simulation Exercises:**
   - Simulate DFS and BFS on a small state space (e.g., a simple graph with nodes S, A, B, C, D, G) and track the order in which nodes are expanded.
3. **DFID Experiment:**

    - Implement DFID (with and without CLOSED list pruning) for a puzzle such as the Eight Puzzle and compare the number of nodes generated.

4. **Analysis Questions:**
    - Compare the space complexity of DFS and BFS for a state space with branching factor 4 and depth 5.
    - Discuss how the use of heuristics (to be introduced in later chapters) could further improve search performance.

Further Reading:

- Deepak Khemani, *A First Course in Artificial Intelligence*, McGraw-Hill Education (India), 2013.
- Russell, Stuart, and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd Edition, Prentice Hall, 2009.
- Additional articles on combinatorial explosion and heuristic search methods.