ABHYASMITRA.IN
STUDY NOTES

ARTIFICIAL INTELLIGENCE

Unit II — Problem Solving: State Space Approach and Search Strategies

Topics Covered (09 Hours)

State Space Search & Tower of Hanoi Problem
Informed (Heuristic) Search Strategies & Evaluation Functions
Greedy Best-First Search (Logic & Complexity)
A* Search (Algorithm, Admissibility, & Consistency)
Iterative-Deepening A* (IDA*) Search
Heuristic Functions (Design & Dominance)
Local Search & Optimization Problems
Hill-Climbing Search (Varieties & Pitfalls)
Simulated Annealing (Physics Analogy & Logic)
Local Beam Search & Online Search Agents in Unknown Worlds

Compiled in a simple, easy-to-understand, book style format
for students of Computer Science and Engineering

TABLE OF CONTENTS

2.1 State Space Search: Tower of Hanoi

In Artificial Intelligence, problem-solving is often framed as a search process. A problem-solving agent is a goal-based agent that decides what to do by finding sequences of actions that lead to desirable states. Before an agent can start searching for a solution, it must formulate the problem clearly. The primary mechanism for doing this is called State Space Formulation.

Five Components of a Search Problem

- **Initial State:** The starting condition of the environment. For example, in a navigation task, it is the city where the journey begins.
- **Actions:** A set of possible actions available to the agent. Given a state, the function Actions(s) returns the set of actions that can be executed in that state.
- **Transition Model:** A description of what each action does. Mathematically, it is represented as a function Result(s, a) that returns the state reached by performing action a in state s.
- **Goal Test:** A condition that determines whether a given state is a goal state. This can be explicit (e.g., 'being in Bucharest') or implicit (e.g., 'having checkmate in chess').
- **Path Cost:** A function that assigns a numeric cost to a path of actions. The step cost of performing action a to go from state s to s' is denoted by c(s, a, s'). The goal is usually to find a solution path with the lowest cost.

The State Space

Together, the initial state, actions, and transition model define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions. We can visualize the state space as a directed graph where the nodes are states and the links are actions.

Tower of Hanoi Problem Formulation

The Tower of Hanoi is a classic mathematical puzzle that serves as an excellent illustration of state space search. The puzzle consists of three pegs (A, B, and C) and N disks of different sizes that can slide onto any peg. The puzzle starts with the disks stacked on peg A in decreasing order of size, forming a cone. The objective is to move the entire stack to peg C, obeying three simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one stack and placing it on top of another stack or on an empty peg.
- No larger disk may be placed on top of a smaller disk.

Formulating Tower of Hanoi as a State Space Search:

- **States:** A state is represented by the positions of the N disks on the three pegs. A common representation is an N-element tuple, where the i-th element indicates which peg (A, B, or C) disk i is currently on (with disk 1 being the smallest and disk N being the largest).
- **Initial State:** All disks are on peg A. For N=3, this is the state (A, A, A).
- **Actions:** Move the top disk of peg x to peg y, provided disk x is smaller than the top disk on peg y, and peg x is not empty.
- **Transition Model:** Executing Move(x, y) transfers the top disk of peg x to peg y, changing the tuple's state. For example, moving disk 1 from peg A to peg B in the initial state results in state (B, A, A).
- **Goal Test:** All disks are on peg C. For N=3, this is the state (C, C, C).
- **Path Cost:** Each move costs 1 unit. The total path cost is equal to the total number of moves.

For N disks, the state space contains exactly 3^N unique states. The state space graph forms a beautiful fractal structure known as a Sierpiński triangle. The optimal path to solve the Tower of Hanoi requires exactly 2^N - 1 moves.

2.2 Informed (Heuristic) Search Strategies

Uninformed search strategies—like Breadth-First Search (BFS) and Depth-First Search (DFS)—are systematic but blind. They expand nodes based purely on the graph structure without knowing if one node is more promising than another. Informed (heuristic) search strategies use domain-specific knowledge to search more efficiently.

Heuristic Function

A heuristic function, denoted by h(n), is an evaluation function that estimates the cost of the cheapest path from the current node n to a goal node.

Heuristic functions are problem-specific. If n is a goal node, then h(n) must equal 0.

Heuristics act as guides. Instead of exploring every path equally, the agent uses h(n) to prioritize nodes that appear closer to the goal. This reduces the search space dramatically, turning exponential time searches into manageable, goal-focused computations.

2.3 Greedy Best-First Search

Greedy Best-First Search is one of the simplest informed search algorithms. It evaluates nodes by using the heuristic function directly as the node selection criterion:

$$f(n) = h(n)$$

At each step, Greedy Best-First Search expands the node that is estimated to be closest to the goal state, based on the assumption that this immediate step will lead to the target quickest (hence the name 'greedy').

A Trace Example: Road Map Navigation

Consider finding a path from Arad to Bucharest using the straight-line distance heuristic, h_SLD(n). Straight-line distance is a heuristic because it estimates the remaining distance, even though roads are winding and the actual road distance is longer. When at Arad, the neighbors are Sibiu, Timisoara, and Zerind. The heuristic values are h_SLD(Sibiu)=253, h_SLD(Timisoara)=329, and h_SLD(Zerind)=374. Greedy Best-First Search immediately chooses Sibiu because 253 is the lowest estimate. It then looks at Sibiu's neighbors and continues making locally optimal choices.

Evaluation of Greedy Best-First Search

Criterion	Status	Detailed Explanation
Completeness	No	It can get stuck in infinite loops in graph search if it follows a path that loops back, unless we keep track of visited states.
Time Complexity	O(b^m)	In the worst case, it can explore all paths. However, with a good heuristic, the complexity can be reduced to O(b * d).
Space Complexity	O(b^m)	It keeps all generated nodes in the frontier memory queue, requiring space proportional to the search depth.
Optimality	No	It is not optimal. Since it only looks at the estimated cost to the goal, it ignores the path cost already spent to reach the current node, leading to longer paths.

2.4 A* Search

The most widely known form of best-first search is A* Search (pronounced 'A-star'). Unlike Greedy Best-First Search, A* avoids expanding paths that are already expensive. It does this by combining the cost to reach the node, g(n), and the estimated cost to the goal from that node, h(n):

$$f(n) = g(n) + h(n)$$

Here, f(n) represents the estimated total cost of the cheapest path from the start node to the goal that passes through node n. A* expands nodes in order of increasing f(n).

Conditions for Optimality

For A* to find the absolute shortest path (to be optimal), the heuristic function must meet certain mathematical conditions depending on the search space structure.

1. Admissibility (For Tree Search)

An admissible heuristic is one that never overestimates the actual cost to reach the goal. Admissible heuristics are optimistic—they always think the goal is closer or exactly as close as it actually is. Formally, for every node n, h(n) <= h*(n), where h*(n) is the true cost to reach the goal from n.

2. Consistency / Monotonicity (For Graph Search)

A heuristic is consistent (or monotonic) if, for every node n and every successor n' generated by action a, the estimated cost of reaching the goal from n is no greater than the step cost to get to n' plus the estimated cost of reaching the goal from n':

h(n) <= c(n, a, n') + h(n')

This is a form of the triangle inequality: the straight-line distance from A to the goal cannot be longer than the path from A to B plus the straight-line distance from B to the goal. Every consistent heuristic is also admissible.

Properties of A Search*

- Completeness: Yes, provided the step costs are greater than some tiny positive value epsilon, and the branching factor is finite.
- Time Complexity: O(b^d) in the worst case, where d is the depth. If the heuristic is highly accurate, A* can run in polynomial time.
- Space Complexity: O(b^d) because A* keeps all generated nodes in memory. This is the biggest practical limitation of A* search.
- Optimality: Yes, A* is optimal when using an admissible heuristic for tree search, and a consistent heuristic for graph search.

2.5 Iterative-deepening A* (IDA*)

Although A* is optimal, its space complexity is a major bottleneck because it stores all generated nodes in memory. Iterative-deepening A* (IDA*) solves this memory problem by combining the space efficiency of Depth-First Search (DFS) with the informed guidance of A*.

In standard Iterative Deepening Search, the cutoff is the depth limit (1, 2, 3, etc.). In IDA*, the cutoff limit is the f-cost ($f(n) = g(n) + h(n)$).

How IDA Operates*

- The initial threshold is set to the f-cost of the start node, which is $f(start) = g(start) + h(start) = 0 + h(start)$.
- A depth-first search is initiated from the start node. The search will backtrack if it encounters any node whose f-cost exceeds the current threshold.
- If the search finds the goal within the threshold, the algorithm terminates successfully.
- If the goal is not found, the threshold is updated to the smallest f-cost of all nodes that exceeded the previous threshold, and a new DFS is started.

> **NOTE**
> IDA* has a space complexity of only $O(b * d)$ because it performs depth-first searches, which only require storing the current path in memory. This is a massive improvement over A*'s $O(b^d)$ space requirement, allowing search agents to solve large-scale problems with limited computer memory.

2.6 Heuristic Functions

The performance of A* and other heuristic search algorithms depends heavily on the quality of the heuristic function. A good heuristic guides the search directly to the goal, while a poor heuristic can reduce the algorithm to a blind search.

Designing Heuristics: Relaxed Problems

One of the most effective ways to design an admissible heuristic is by defining a relaxed problem. A relaxed problem is one that has fewer restrictions on the available actions. The cost of an optimal solution to a relaxed problem is always an admissible heuristic for the original problem.

For example, in the 8-puzzle, the rules state that a tile can move from square A to square B if B is blank and adjacent. If we relax these rules:

- **Relaxation 1 (Tiles can move anywhere):** This leads to the Misplaced Tiles heuristic (counting how many tiles are in the wrong place).
- **Relaxation 2 (Tiles can move to any adjacent square, even if occupied):** This leads to the Manhattan Distance heuristic (sum of the horizontal and vertical distances of each tile from its goal position).

Heuristic Dominance

If we have two admissible heuristics, h1 and h2, we say that h2 dominates h1 if:

h2(n) >= h1(n) for all nodes n

A dominant heuristic is always better because it is closer to the true cost, meaning it will expand fewer nodes during the A* search. For the 8-puzzle, the Manhattan Distance heuristic dominates the Misplaced Tiles heuristic.

2.7 Local Search and Optimization Problems

In many problems, the path to the goal is completely irrelevant. We do not care about the sequence of steps it took to get there; we only care about the final configuration itself. Examples include the 8-Queens puzzle, circuit board layouts, and network routing optimizations.

Local search algorithms operate using a single current state rather than exploring paths. They move only to neighboring states. Because they do not retain a search tree, they have two major advantages: they use almost no memory, and they can find reasonable solutions in massive, continuous state spaces where systematic algorithms are useless.

The State-Space Landscape

To understand local search, we visualize the state space as a landscape with hills (representing state value or utility) and valleys (representing cost). The goal is to find the highest peak (global maximum) if we are maximizing, or the lowest valley (global minimum) if we are minimizing.

Landscape Feature	Description	Challenge for Algorithms
Global Maximum	The highest point in the entire landscape, representing the absolute best solution.	The ultimate target for optimization.
Local Maximum	A peak that is higher than all its immediate neighbors, but lower than the global maximum.	Algorithms can get stuck here, thinking they have reached the best solution when they haven't.
Plateau	A flat area of the state space where neighboring states have the same evaluation value.	The algorithm has no direction to guide its next step, leading to random wandering.
Ridge	A sequence of local maxima forming a line, where the slope drops steeply on either side.	It is difficult to navigate because small steps in any coordinate direction lead down the slope.

2.8 Hill-climbing search

Hill-climbing search is a simple loop that continuously moves in the direction of increasing value—that is, uphill. It terminates when it reaches a peak where no neighbor has a higher value. It is often compared to climbing a mountain in a thick fog with amnesia.

Three Major Drawbacks of Simple Hill-Climbing

- **1. Local Maxima:** Because the algorithm only moves uphill, once it reaches a local maximum, it stops. Every single move from this state leads downwards, so the agent stays stuck, missing the global maximum.
- **2. Ridges:** A ridge results in a sequence of local peaks. Unless the algorithm can move diagonally along the ridge, moving in standard grid directions will go down, halting the search.
- **3. Plateaus:** A plateau is a flat area. If the agent reaches a plateau, it cannot determine which way is uphill and wanders aimlessly, wasting computational time.

Variations to Overcome Limitations

- **Stochastic Hill-Climbing:** Chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the move.
- **First-Choice Hill-Climbing:** Generates successors randomly until one is found that is better than the current state, which is useful when a state has thousands of neighbors.
- **Random-Restart Hill-Climbing:** Conducts a series of hill-climbing searches from randomly generated initial states, running until a goal is found. This is remarkably effective and is guaranteed to find the global optimum if given enough attempts.

2.9 Simulated annealing

Simulated Annealing is a powerful local search algorithm that avoids getting stuck in local maxima by occasionally accepting 'bad' moves (moves that decrease value). The algorithm is inspired by metallurgy, specifically the process of annealing—cooling a liquid metal slowly so that it settles into a low-energy, highly ordered crystalline structure.

The Core Logic of Simulated Annealing

Instead of choosing the best move, simulated annealing chooses a random move. If the random move improves the situation, it is always accepted. If the move is worse, the algorithm accepts it with a probability P:

$$e^{\Delta E / T}$$

Where dE is the change in state value (which is negative for a bad move), and T is the temperature. The temperature T starts high and slowly decreases according to a cooling schedule.

When T is high (hot), the probability of accepting bad moves is very high, allowing the agent to jump out of local maxima and explore the landscape. As T approaches zero (cooling), the probability of accepting bad moves drops to zero, and the algorithm behaves like a standard greedy hill-climber, settling into the nearest peak.

2.10 Local beam search

Keeping track of just a single state can make local search fragile. Local Beam Search addresses this by keeping track of k states instead of just one. The search begins with k randomly generated states.

How Local Beam Search Works

- At each step, the algorithm generates all the successors of all k current states.
- If any of these successors is the goal state, the algorithm terminates successfully.
- If not, it selects the k best successors from the complete list of generated successors and repeats the process.

NOTE
It is important to realize that Local Beam Search is different from running k independent hill-climbing searches in parallel. In parallel hill-climbing, the paths do not communicate. In Local Beam Search, information is shared: if one state finds a highly promising path, it generates many good successors, and the algorithm will naturally drop less promising paths to focus its resources on the better area.

2.11 Online Search Agents and Unknown Environments

So far, we have focused on offline search. An offline search agent computes a complete solution path first using its model of the world, and then executes that path without looking at the environment (acting with its eyes closed).

In contrast, an Online Search Agent interleaves computation and action: it takes an action, observes the environment, and then decides what action to take next. Online search is necessary when the environment is unknown, dynamic, or stochastic.

Challenges of Online Search

- **Dead Ends:** A state from which the agent can no longer reach the goal, or from which it cannot return to the start. In an unknown environment, an agent might accidentally enter a dead end (e.g., driving off a cliff).
- **Competitive Ratio:** A metric used to evaluate online agents. It compares the path cost of the online agent (which doesn't know the map in advance) with the path cost of an optimal offline agent that has full knowledge of the map.
- **Exploration vs. Exploitation:** The agent must balance exploiting what it already knows to minimize cost versus exploring unknown areas to discover better paths.

Unit Summary

This unit explored problem-solving via the state space approach and various search strategies. The key concepts are summarized below.

- A search problem is defined by an initial state, actions, a transition model, a goal test, and a path cost.
- The Tower of Hanoi has 3^N states, and its state space graph forms a Sierpiński triangle fractal. The optimal path requires 2^N - 1 moves.
- Informed (heuristic) search strategies use problem-specific knowledge via a heuristic function h(n) to prioritize nodes closer to the goal.
- Greedy Best-First Search expands nodes with the lowest heuristic value (f(n) = h(n)). It is neither complete nor optimal.
- A* Search combines path cost and heuristic estimate (f(n) = g(n) + h(n)). It is optimal if the heuristic is admissible (for tree search) and consistent (for graph search).
- Iterative-deepening A* (IDA*) solves A*'s memory issues by using f-cost as a threshold limit for a series of depth-first searches.
- Admissible heuristics can be generated by relaxing the rules of the original problem. A heuristic dominates another if its estimates are consistently higher (closer to the true cost).
- Local search algorithms are used when the path to the goal does not matter. They keep only a single state in memory and move to neighbors.
- Hill-climbing search moves uphill but can get trapped by local maxima, ridges, and plateaus. Random restarts help find global optimums.
- Simulated Annealing accepts bad moves with a probability e^(dE/T) that decreases over a cooling schedule, allowing exploration of the state landscape.
- Local Beam Search tracks k states in parallel, sharing promising successors among them.
- Online search agents interleave computation and action, navigating unknown worlds where competitive ratios and dead ends are major challenges.