UNIT 2: SEARCH ALGORITHMS

1. RANDOM SEARCH

1.1 Introduction to Search in Artificial Intelligence

Search is one of the most fundamental techniques used in Artificial Intelligence to solve problems. Many AI problems can be viewed as searching through a set of possible states to find a path from the initial state to the goal state. Search algorithms systematically explore the state space to identify a solution. Some search techniques use intelligent strategies, while others use simple or uninformed approaches.

1.2 Meaning of Random Search

Random Search is one of the simplest search techniques in Artificial Intelligence. In this method, the agent selects actions randomly without using any specific strategy or heuristic guidance. It does not consider past states or future consequences in a structured way. Instead, it explores the search space by randomly choosing available options until the goal state is found.

In simple words, Random Search tries different possibilities by chance rather than by using logical reasoning or systematic exploration.

1.3 Working of Random Search

The working of Random Search is straightforward. The algorithm starts from the initial state. At each step, it randomly selects one of the possible actions and moves to a new state. This process continues until either the goal state is reached or a predefined stopping condition occurs.

The basic steps involved are:

- Start from the initial state.
- Generate all possible actions from the current state.
- Select one action randomly.
- Move to the new state.
- Repeat the process until the goal is found or the search is terminated.

Since the choices are random, the algorithm may revisit the same states multiple times and may not guarantee finding the optimal solution.

1.4 Characteristics of Random Search

Random Search has certain characteristics that distinguish it from other search techniques. It is simple to implement and does not require additional information such as heuristics or path costs.

Important characteristics include:

- No use of heuristic or informed guidance.
- Selection of actions is completely random.
- May revisit the same state multiple times.
- Does not guarantee optimal solution.

- May take a long time to find the goal in large state spaces.

1.5 Advantages of Random Search

Although Random Search is not efficient for complex problems, it has some advantages. Its simplicity makes it easy to understand and implement. It does not require additional memory for storing explored states or maintaining complex data structures.

The main advantages are:

- Simple and easy to implement.
- Does not require heuristic knowledge.
- Useful when no information about the problem structure is available.

1.6 Limitations of Random Search

Random Search has several limitations that make it unsuitable for most real-world AI problems. Because it does not follow a systematic approach, it may waste time exploring irrelevant states.

Major limitations include:

- Inefficient for large search spaces.
- No guarantee of finding a solution.
- May loop indefinitely without progress.
- Does not ensure optimal or shortest path solution.

1.7 Example of Random Search

Consider a simple maze where an agent needs to reach the exit. In Random Search, the agent chooses any available direction randomly at each step. It may move forward, backward, left, or right without planning. Eventually, it might reach the exit, but it may also wander aimlessly or take a very long path.

This example shows that Random Search can sometimes find a solution, but it is not efficient compared to systematic search methods.

SUMMARY OF THE TOPIC

Random Search is the simplest form of search technique in Artificial Intelligence. It selects actions randomly without using heuristics or structured exploration. Although it is easy to implement, it is inefficient for large or complex problems and does not guarantee optimal solutions. Random Search is mainly useful for understanding basic search concepts and for situations where no additional problem information is available.

2. SEARCH WITH OPEN LIST AND CLOSED LIST

2.1 Introduction to Open List and Closed List Concept

In many search algorithms, especially graph search algorithms, it is important to keep track of which states have been explored and which states are yet to be explored. To manage this process efficiently, two important data structures are used: the Open List and the Closed List. These lists help in organizing the search process and avoiding repeated exploration of the same states.

The use of open and closed lists makes the search more systematic and efficient compared to random exploration. It is commonly used in algorithms such as Breadth First Search, Depth First Search, and A*.

2.2 Meaning of Open List

The Open List contains all the nodes that have been generated but not yet expanded. These nodes are waiting to be explored. Whenever a new state is discovered during the search process, it is added to the open list.

The open list acts like a queue or stack depending on the search strategy being used. For example, in Breadth First Search, it behaves like a queue, while in Depth First Search, it behaves like a stack.

Important points about the Open List:

- Contains nodes that are discovered but not yet expanded.
- Stores frontier nodes of the search tree or graph.
- Its structure depends on the search algorithm used.
- Helps in selecting the next node to explore.

2.3 Meaning of Closed List

The Closed List contains nodes that have already been expanded. Once a node is removed from the open list and expanded, it is added to the closed list. This prevents the algorithm from revisiting the same state again.

The closed list is especially important in graph search problems where cycles may exist. Without a closed list, the algorithm might enter an infinite loop by repeatedly visiting the same states.

Important points about the Closed List:

- Contains nodes that have already been explored.
- Prevents repeated expansion of the same state.
- Helps avoid infinite loops in cyclic graphs.
- Improves efficiency by reducing redundancy.

2.4 Working of Search with Open and Closed Lists

The search process using open and closed lists follows a systematic approach. Initially, the open list contains only the initial state, and the closed list is empty. The algorithm repeatedly selects a node from the open list, checks whether it is the goal state, and if not, expands it to generate its successor states.

The general steps are:

- Initialize the open list with the initial state.
- Keep the closed list empty at the beginning.
- Remove a node from the open list for expansion.
- If it is the goal state, stop the search.
- Otherwise, generate successor states.
- Add unexplored successors to the open list.
- Move the expanded node to the closed list.

This process continues until the goal is found or the open list becomes empty.

2.5 Advantages of Using Open and Closed Lists

Using open and closed lists provides several advantages in search algorithms. It ensures systematic exploration and reduces unnecessary computation.

Major advantages include:

- Avoids revisiting already explored states.
- Prevents infinite loops in cyclic graphs.
- Improves time efficiency.
- Provides better organization of the search process.

2.6 Example for Better Understanding

Consider a route-finding problem between cities. Initially, the starting city is placed in the open list. When it is expanded, its neighboring cities are added to the open list. The starting city is then moved to the closed list. If one of the neighboring cities leads back to the starting city, it will not be added again because the starting city is already in the closed list. This avoids repeated exploration.

SUMMARY OF THE TOPIC

Search with Open List and Closed List is a systematic approach used in many AI search algorithms. The open list contains nodes that are yet to be explored, while the closed list stores already expanded nodes. This method prevents repeated exploration, avoids infinite loops, and improves efficiency. Understanding open and closed lists is essential for implementing graph search algorithms effectively.

3. DEPTH FIRST SEARCH AND BREADTH FIRST SEARCH

3.1 Introduction to Systematic Search Strategies

Depth First Search (DFS) and Breadth First Search (BFS) are two fundamental uninformed search algorithms used in Artificial Intelligence. Unlike Random Search, these algorithms explore the state space in a systematic manner. They do not use heuristic information but follow a fixed rule for selecting the next node to expand. Both algorithms are widely used for traversing trees and graphs.

3.2 Depth First Search (DFS)

Depth First Search is a search algorithm that explores as far as possible along a branch before backtracking. It follows a depth-oriented strategy, meaning it goes deep into the search tree until it reaches a leaf node or a dead end, and then it backtracks to explore other branches.

DFS uses a stack data structure for implementation. In many programming languages, it can also be implemented using recursion because recursion internally uses a stack.

Working steps of Depth First Search:

- Start with the initial node and push it onto the stack.
- Pop the top node from the stack and check if it is the goal state.
- If it is not the goal, generate its successors.
- Push the successors onto the stack.

- Repeat the process until the goal is found or the stack becomes empty.

DFS may go very deep in the search tree even if the goal is located near the root. It does not guarantee the shortest path solution.

3.3 Advantages of Depth First Search

Depth First Search has certain advantages that make it useful in some situations. It requires less memory compared to Breadth First Search because it only needs to store the nodes along the current path.

Major advantages include:

- Low memory requirement.
- Simple implementation using stack or recursion.
- Useful for problems where solutions are deep in the tree.

3.4 Limitations of Depth First Search

Despite its advantages, DFS has some important limitations. It may get stuck in an infinite path if the graph contains cycles and no proper visited list is maintained.

Major limitations include:

- Does not guarantee optimal solution.
- May not terminate in infinite-depth spaces.
- Can explore very deep unnecessary paths.

3.5 Breadth First Search (BFS)

Breadth First Search is a search algorithm that explores all nodes at the current depth level before moving to the next level. It follows a breadth-oriented strategy, meaning it expands nodes layer by layer starting from the root.

BFS uses a queue data structure for implementation. The first node added to the queue is the first one to be expanded.

Working steps of Breadth First Search:

- Start with the initial node and add it to the queue.
- Remove the front node from the queue and check if it is the goal state.
- If not, generate its successors.
- Add all successors to the rear of the queue.
- Repeat the process until the goal is found or the queue becomes empty.

BFS systematically explores all nodes at each level, which ensures that the shallowest solution is found first.

3.6 Advantages of Breadth First Search

Breadth First Search has important advantages, especially in problems where the shortest path is required in an unweighted graph.

Major advantages include:

- Guarantees shortest path in unweighted graphs.
- Complete, meaning it will find a solution if one exists.
- Systematic level-by-level exploration.

3.7 Limitations of Breadth First Search

The main disadvantage of BFS is its high memory requirement. It must store all nodes at the current level, which can be very large in wide search trees.

Major limitations include:

- High memory consumption.
- Slower in very deep search spaces.
- Not suitable for extremely large branching factors.

3.8 Comparison Between DFS and BFS

Depth First Search and Breadth First Search differ in strategy, memory usage, and optimality. DFS explores depth-wise using a stack, while BFS explores level-wise using a queue.

Key comparison points:

- DFS uses stack; BFS uses queue.
- DFS requires less memory; BFS requires more memory.
- DFS does not guarantee shortest path; BFS guarantees shortest path in unweighted graphs.
- DFS may go deep unnecessarily; BFS explores level by level.

SUMMARY OF THE TOPIC

Depth First Search and Breadth First Search are fundamental uninformed search algorithms in Artificial Intelligence. DFS explores deep into the search tree before backtracking and uses a stack, while BFS explores level by level and uses a queue. DFS is memory efficient but does not guarantee optimal solutions. BFS guarantees the shortest path in unweighted graphs but requires more memory. Understanding both algorithms is essential for mastering search techniques in AI.

4. HEURISTIC SEARCH

4.1 Introduction to Heuristic Search

Heuristic Search is an informed search technique used in Artificial Intelligence to improve the efficiency of problem solving. Unlike uninformed search algorithms such as BFS and DFS, heuristic search uses additional knowledge about the problem to guide the search process. This additional knowledge is called a heuristic.

A heuristic is a function that estimates how close a given state is to the goal state. By using heuristic information, the algorithm can explore more promising paths first instead of blindly searching through all possible states.

4.2 Meaning of Heuristic Function

A heuristic function is usually denoted by h(n), where n represents a node in the search tree. The value of h(n) provides an estimate of the cost required to reach the goal from node n. The better the heuristic function, the more efficient the search becomes.

For example, in a route-finding problem between cities, the straight-line distance between the current city and the destination can be used as a heuristic value.

Important properties of heuristic functions:

- Provides an estimate of remaining cost.
- Guides the search toward the goal.
- Should be easy to compute.
- Better heuristics lead to faster solutions.

4.3 Need for Heuristic Search

In large or complex problems, uninformed search algorithms may take a long time to find a solution because they explore many unnecessary states. Heuristic search reduces the search space by prioritizing states that appear closer to the goal.

This makes heuristic search more efficient in terms of time and sometimes memory. It is especially useful in problems such as route finding, puzzle solving, and game playing.

4.4 Types of Heuristic Search

There are several algorithms that use heuristic information. Some of the most common heuristic search techniques include Best First Search and A* Search. These algorithms use heuristic values to decide which node should be expanded next.

In these algorithms, nodes with better heuristic values (closer to the goal) are given higher priority.

4.5 Advantages of Heuristic Search

Heuristic Search provides significant advantages over uninformed search methods. By using problem-specific knowledge, it reduces unnecessary exploration.

Major advantages include:

- Faster solution compared to uninformed search.
- Reduces search space.
- More efficient for complex problems.
- Useful in real-world applications such as navigation systems.

4.6 Limitations of Heuristic Search

Although heuristic search is efficient, it also has some limitations. The performance depends heavily on the quality of the heuristic function. If the heuristic is poorly designed, the algorithm may behave like uninformed search or may produce suboptimal solutions.

Major limitations include:

- Requires good heuristic knowledge.
- Designing an accurate heuristic can be difficult.
- May not guarantee optimal solution unless specific conditions are satisfied.

4.7 Example for Better Understanding

Consider a puzzle-solving problem such as the 8-puzzle. A heuristic function can count the number of misplaced tiles compared to the goal state. The algorithm then prefers states with fewer misplaced tiles, as they are likely closer to the solution.

This example shows how heuristic information helps the algorithm move in a direction that seems more promising instead of exploring all possibilities.

SUMMARY OF THE TOPIC

Heuristic Search is an informed search technique that uses additional knowledge in the form of a heuristic function to guide the search process. The heuristic function estimates the distance from the current state to the goal state. By prioritizing promising paths, heuristic search reduces the search space and improves efficiency. However, its performance depends on the quality of the heuristic function. Heuristic search plays a crucial role in solving complex real-world AI problems.

5. BEST FIRST SEARCH

5.1 Introduction to Best First Search

Best First Search is an informed search algorithm that expands the most promising node according to a specific evaluation function. It is an extension of heuristic search where the next node to be explored is selected based on the best heuristic value. The main idea is to always choose the node that appears closest to the goal.

Best First Search uses a priority queue data structure. The node with the best evaluation value is given the highest priority and is expanded first.

5.2 Evaluation Function in Best First Search

In Best First Search, an evaluation function f(n) is used to decide which node should be expanded next. In the simplest form of Best First Search, the evaluation function is equal to the heuristic function.

That means: $f(n) = h(n)$, where $h(n)$ estimates the cost from the current node to the goal state. The node with the smallest heuristic value is expanded first.

Important points about the evaluation function:

- Uses heuristic value to guide search.
- Smaller value means closer to goal.
- Stored in a priority queue.
- Determines the order of node expansion.

5.3 Working of Best First Search

The algorithm starts by placing the initial node in the open list (priority queue). The node with the lowest heuristic value is selected for expansion. Its successors are generated and added to the open list based on their heuristic values.

The general steps are:

- Initialize the open list with the initial node.
- Select the node with the smallest heuristic value.
- Check if it is the goal state.
- If not, generate its successors.
- Add successors to the priority queue.
- Move the expanded node to the closed list.

This process continues until the goal state is found or the open list becomes empty.

5.4 Advantages of Best First Search

Best First Search is more efficient than uninformed search methods because it uses heuristic guidance. It focuses on promising paths and reduces unnecessary exploration.

Major advantages include:

- Faster than BFS and DFS in many cases.
- Reduces search space using heuristic guidance.
- Simple implementation using priority queue.

5.5 Limitations of Best First Search

The performance of Best First Search depends heavily on the heuristic function. If the heuristic is not accurate, the algorithm may explore wrong paths first.

Major limitations include:

- Does not guarantee optimal solution.
- May get trapped in local optimum.
- Performance depends on heuristic quality.
- May require large memory to store nodes.

5.6 Example for Better Understanding

Consider a route-finding problem between cities. Suppose we use straight-line distance to the destination as the heuristic. Best First Search will always choose the city that appears closest to the goal according to this estimate. Although this strategy may reach the goal quickly, it may not always find the shortest path.

SUMMARY OF THE TOPIC

Best First Search is an informed search algorithm that selects the most promising node based on a heuristic evaluation function. It uses a priority queue to expand nodes with the smallest heuristic value first. While it is

faster and more efficient than uninformed search in many cases, it does not guarantee optimal solutions. The effectiveness of Best First Search depends largely on the quality of the heuristic function.

6. A* ALGORITHM

6.1 Introduction to A* Algorithm

The A* Algorithm is one of the most important and widely used search algorithms in Artificial Intelligence. It is an informed search technique that combines the advantages of Uniform Cost Search and Heuristic Search. A* is designed to find the optimal path from the initial state to the goal state while minimizing the total cost.

Unlike simple Best First Search, A* considers both the cost already spent to reach a node and the estimated cost remaining to reach the goal. Because of this balanced approach, A* is both complete and optimal under certain conditions.

6.2 Evaluation Function of A* Algorithm

The A* algorithm uses an evaluation function represented as f(n). The function f(n) is defined as:

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

Here, g(n) represents the actual cost from the initial state to the current node n, and h(n) represents the heuristic estimate of the cost from node n to the goal state. The value f(n) represents the estimated total cost of the solution path that passes through node n.

Important components of the evaluation function:

- g(n) – Actual path cost from start node to current node.
- h(n) – Estimated cost from current node to goal.
- f(n) – Estimated total cost of solution through node n.

6.3 Working of A* Algorithm

The A* algorithm maintains two lists: an open list and a closed list. The open list contains nodes that are yet to be explored, and the closed list contains nodes that have already been expanded.

The general steps of A* are:

- Initialize the open list with the start node.
- Calculate $f(n) = g(n) + h(n)$ for the start node.
- Select the node with the smallest f(n) value from the open list.
- If the selected node is the goal state, stop the search.
- Otherwise, generate its successor nodes.
- For each successor, calculate g(n), h(n), and f(n).
- Add new successors to the open list if not already explored.
- Move the expanded node to the closed list.

This process continues until the goal is found or the open list becomes empty.

6.4 Properties of A* Algorithm

A* has important theoretical properties that make it very useful in practical applications.

Important properties include:

- Complete – It will always find a solution if one exists.
- Optimal – It finds the least-cost solution if the heuristic is admissible.
- Efficient – More efficient than uninformed search in many cases.
- Uses both actual and estimated costs.

6.5 Admissible and Consistent Heuristic

For A* to guarantee optimality, the heuristic function must be admissible. A heuristic is admissible if it never overestimates the true cost to reach the goal. In other words, h(n) should always be less than or equal to the actual remaining cost.

A heuristic is consistent (or monotonic) if the estimated cost is always less than or equal to the step cost plus the heuristic value of the next node. Consistency ensures that the f(n) value never decreases along a path.

6.6 Advantages of A* Algorithm

A* algorithm offers several advantages compared to other search methods.

Major advantages include:

- Finds optimal solution with admissible heuristic.
- More efficient than BFS and DFS in many problems.
- Widely used in pathfinding and navigation systems.
- Balances exploration and cost efficiency.

6.7 Limitations of A* Algorithm

Despite its strengths, A* also has some limitations. The main limitation is its high memory usage because it stores all generated nodes in memory.

Major limitations include:

- High memory consumption.
- Performance depends on heuristic quality.
- May become slow for extremely large state spaces.

6.8 Example for Better Understanding

Consider a route-finding problem between two cities. The value g(n) represents the distance already traveled from the starting city, and h(n) represents the straight-line distance to the destination. The A* algorithm selects the city with the smallest total estimated distance f(n). By combining actual and estimated costs, A* efficiently finds the shortest path.

SUMMARY OF THE TOPIC

The A* Algorithm is a powerful informed search technique that uses the evaluation function f(n) = g(n) + h(n). It combines actual path cost and heuristic estimate to find the optimal solution. If the heuristic is admissible and consistent, A* guarantees optimality. Although it requires more memory compared to some other algorithms, it is widely used in real-world applications such as navigation systems and game development due to its efficiency and reliability.

7. GAME SEARCH

7.1 Introduction to Game Search in Artificial Intelligence

Game Search is a search technique used in Artificial Intelligence for decision-making in competitive environments such as board games and strategy games. Unlike simple search problems where there is only one agent trying to reach a goal, game search involves two or more players with opposing objectives. Each player tries to maximize their own benefit while minimizing the opponent's advantage.

Examples of such games include Chess, Tic-Tac-Toe, Checkers, and other turn-based strategy games. In these problems, the AI agent must consider not only its own possible moves but also the possible responses of the opponent.

7.2 Game Tree Representation

Game problems are represented using a Game Tree. A game tree is a tree structure where nodes represent game states and edges represent possible moves. The root node represents the current state of the game, and each level of the tree alternates between the moves of different players.

Terminal nodes represent final states of the game, such as win, lose, or draw. Each terminal node is assigned a utility value that indicates the outcome of the game from the perspective of the AI agent.

7.3 Minimax Algorithm

The Minimax algorithm is a fundamental game search algorithm used for two-player zero-sum games. In a zero-sum game, one player's gain is equal to the other player's loss. The Minimax algorithm assumes that both players play optimally.

The main idea of the Minimax algorithm is that one player tries to maximize the utility value while the opponent tries to minimize it. Therefore, the algorithm alternates between maximizing and minimizing levels in the game tree.

The working of the Minimax algorithm can be described as follows:

- Generate the complete game tree up to a certain depth.
- Assign utility values to terminal nodes.
- At MAX level, choose the maximum value from child nodes.
- At MIN level, choose the minimum value from child nodes.
- Propagate these values upward until the root node is reached.
- Select the move corresponding to the best value at the root.

Here, MAX represents the AI agent trying to maximize its score, and MIN represents the opponent trying to minimize the AI's score. By applying this rule recursively, the algorithm determines the best possible move assuming optimal play from both sides.

7.4 Properties of Minimax Algorithm

The Minimax algorithm has several important properties. It guarantees the optimal decision if the opponent also plays optimally. However, it can be computationally expensive because the size of the game tree grows exponentially with depth.

Important properties include:

- Complete – Finds a solution if the tree is finite.
- Optimal – Provides best move under optimal opponent play.
- Time complexity grows exponentially with depth.
- Memory usage depends on implementation method.

7.5 Limitations of Minimax Algorithm

The major limitation of the Minimax algorithm is its high computational cost. For complex games like chess, the number of possible game states is extremely large. Exploring the complete game tree is not feasible in practice.

To overcome this limitation, depth-limited search and evaluation functions are used. Additionally, Alpha-Beta Pruning is applied to reduce the number of nodes evaluated.

7.6 Alpha-Beta Pruning

Alpha-Beta Pruning is an optimization technique used to improve the efficiency of the Minimax algorithm. It reduces the number of nodes evaluated in the game tree without affecting the final decision.

The idea behind Alpha-Beta Pruning is that if a move is found to be worse than a previously examined move, there is no need to explore it further. This allows the algorithm to prune or cut off large parts of the tree.

Two important values are maintained during the search:

- Alpha – The best (highest) value found so far at MAX level.
- Beta – The best (lowest) value found so far at MIN level.

During the search process, if the value of a node becomes worse than the current alpha or beta value, further exploration of that branch is stopped. This pruning does not change the final result but significantly reduces computation.

7.7 Advantages of Alpha-Beta Pruning

Alpha-Beta Pruning provides significant performance improvements over the basic Minimax algorithm.

Major advantages include:

- Reduces number of nodes evaluated.
- Speeds up decision-making process.
- Allows deeper search within same time limit.

- Maintains optimality of Minimax.

7.8 Example for Better Understanding

Consider a simple Tic-Tac-Toe game. The AI player (MAX) evaluates all possible moves. Using Minimax, it calculates possible outcomes assuming the opponent (MIN) also plays optimally. With Alpha-Beta Pruning, if the AI discovers that a branch will result in a worse outcome than a previously evaluated move, it stops exploring that branch. This reduces computation while still selecting the optimal move.

SUMMARY OF THE TOPIC

Game Search is used in competitive environments where multiple players make decisions. The Minimax algorithm determines the optimal move by assuming that both players act optimally. It alternates between maximizing and minimizing levels in a game tree. However, Minimax can be computationally expensive. Alpha-Beta Pruning improves efficiency by eliminating branches that cannot influence the final decision. Together, these techniques form the foundation of AI-based game-playing systems.