Search Algorithms

1. Understanding Search Problems

A search problem is a computational problem where an agent must find a sequence of actions that lead from an initial state to a goal state. Identifying a search problem involves recognizing:

- **Initial State**: Where the search begins.
- **Goal State**: The desired outcome.
- **State Space**: The set of all possible states.
- **Actions**: Possible moves or transitions.
- **Path Cost**: The cost associated with a sequence of actions.

Concepts:

- **Agent**: An entity making decisions to solve the search problem.
- **State**: A representation of a configuration in the problem space.
- **Actions**: The possible moves the agent can take from a given state.

2. Solving Search Problems

Solution and Optimal Solution

A solution to a search problem is a sequence of actions leading from the initial state to the goal state. The **optimal solution** is the one with the least cost or shortest path.

3. Depth-First Search (DFS)

DFS explores as far down a branch as possible before backtracking.

Algorithm:

```python
# Depth-First Search Implementation in Python
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, start, visited=set()):
        if start not in visited:
            print(start, end=' ')
            visited.add(start)
            for neighbor in self.graph[start]:
                self.dfs(neighbor, visited)

# Example Usage
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)

g.dfs(0)
```

4. Breadth-First Search (BFS)

BFS explores all neighbors of a node before moving to the next level.

Algorithm:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Example Usage
graph = {
    0: [1, 2],
    1: [3, 4],
    2: [5],
    3: [],
    4: [],
    5: []
}
bfs(graph, 0)
```

5. Greedy Best-First Search

This heuristic-based search prioritizes nodes based on an estimated cost function.

Algorithm (Pseudocode):

1. Initialize an open list with the start node.
2. Remove the node with the lowest heuristic value.
3. If the node is the goal, return the path.
4. Expand the node's neighbors and add them to the open list.
5. Repeat until a solution is found.

6. Adversarial Search (Game Playing)

Adversarial search is used in games where opponents compete.

Minimax Algorithm

Minimax is used in two-player games to minimize the opponent's maximum possible gain.

Pseudocode:

```
def minimax(depth, node_index, maximizing_player, values, alpha, beta):
    if depth == 0:
        return values[node_index]

    if maximizing_player:
        best = float('-inf')
        for i in range(2):
            val = minimax(depth - 1, node_index * 2 + i, False, values, alpha,
beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = float('inf')
        for i in range(2):
            val = minimax(depth - 1, node_index * 2 + i, True, values, alpha,
beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best
```

Alpha-Beta Pruning (α-β Pruning)

Alpha-beta pruning is an optimization technique for the **Minimax algorithm**, which is used in decision-making and game theory (e.g., chess, tic-tac-toe). The goal of alpha-beta pruning is to reduce the number of nodes evaluated in the minimax tree by "pruning" branches that **will not affect the final decision**.

How it Works

- Alpha (α) represents the **best (highest) value** a **maximizing** player can guarantee.
- Beta (β) represents the **best (lowest) value** a **minimizing** player can guarantee.
- During tree traversal, if we find a move that guarantees a worse outcome than a previously evaluated move, we **stop evaluating that branch**.

This pruning helps the algorithm ignore parts of the tree that do not influence the final result, reducing the number of computations.

Alpha-Beta Pruning Conditions

1. If a **maximizing** node finds a value **greater than or equal to** beta, it stops searching its children.
   - **Prune when:** α≥β\alpha \geq \beta
2. If a **minimizing** node finds a value **less than or equal to** alpha, it stops searching its children.
   - **Prune when:** α≥β\alpha \geq \beta

Example of Alpha-Beta Pruning

Consider a two-player game tree where:

- Max (✓) tries to maximize the score.
- Min (✗) tries to minimize the score.

Without Pruning:

```
      Max (✓)
     /       \
   Min(✗)    Min(✗)
  /    \      /    \
✓(3)  ✓(5)  ✓(2)
```

Depth-Limited Minimax

Depth-limited minimax is a variation of the **Minimax algorithm** that limits the depth of the search tree to a fixed value. Instead of exploring the entire game tree to determine the best possible move, the algorithm only searches up to a specific depth and uses a heuristic evaluation to estimate the value of leaf nodes beyond that depth.

This approach is useful in games where the state space is large, and exploring the entire tree is computationally expensive or impractical. By setting a fixed depth limit, you can balance between **computation time** and **strategic depth** (the number of moves ahead considered by the algorithm).

How it Works

1. **Tree Traversal:** Similar to the standard minimax algorithm, depth-limited minimax performs a recursive search of the game tree, but it halts the search when it reaches a certain **depth limit**.
2. **Evaluation at Leaves:** Once the search reaches the depth limit, it uses an **evaluation function** (heuristic) to estimate the desirability of the leaf node (the state of the game at that point).
3. **Pruning:** After evaluating all leaf nodes at the depth limit, the algorithm backpropagates the evaluations up the tree, applying the minimax rule (maximize for the maximizing player, minimize for the minimizing player) until it reaches the root.
4. **Depth-Limited Decision:** At the root, the algorithm selects the best move based on the evaluations computed up to the limited depth.

Advantages of Depth-Limited Minimax

- **Improved Efficiency:** By limiting the depth, it significantly reduces the number of nodes to explore, thus reducing computational cost.
- **Faster Decision Making:** It allows the AI to make decisions more quickly since it doesn't need to search all possible moves deep into the game tree.
- **Strategic Balance:** You can control how far ahead the AI will think by adjusting the depth limit.

Disadvantages

- **Inaccuracy:** Limiting the depth might result in suboptimal decisions, as the algorithm cannot foresee long-term consequences that are beyond the depth limit.
- **Heuristic Dependency:** The evaluation function becomes more critical, as it is used to evaluate leaf nodes at the limited depth, and if the heuristic is not accurate, the decision quality can degrade.

Example

Let’s say you are playing a game like **tic-tac-toe**, and you limit the search depth to 2 moves (i.e., look ahead 2 moves).

- At depth 0 (root), it's the maximizing player's turn.
- The algorithm generates all possible moves for the maximizing player.
- At depth 1, the minimizing player will respond, and the algorithm generates all possible moves for the minimizing player.
- At depth 2, the maximizing player will respond again, but since this is the depth limit, the algorithm stops generating further moves and evaluates the leaf nodes based on a heuristic evaluation function (e.g., 1 for a win, -1 for a loss, 0 for a draw).

The algorithm will then backtrack and choose the best move for the maximizing player based on the evaluated values at depth 2.

Depth-Limited Minimax Pseudocode

```
def depth_limited_minimax(node, depth, maximizing_player):
    if depth == 0 or game_over(node):
        return evaluate(node)  # Use heuristic evaluation at depth limit

    if maximizing_player:
        max_eval = -infinity
        for child in children(node):
            eval = depth_limited_minimax(child, depth - 1, False)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = infinity
        for child in children(node):
            eval = depth_limited_minimax(child, depth - 1, True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Adjusting the Depth Limit

- **Shallow depth limits** (e.g., depth = 1 or 2) will result in faster decision-making but may lead to poor strategic choices, as it only looks at short-term moves.
- **Deeper depth limits** (e.g., depth = 3 or 4) will give more strategic insight, but the computation time will increase.