Chapter 3

Searching Techniques in AI

In artificial intelligence (AI), searching techniques refer to algorithms and strategies used to explore and navigate through problem spaces in order to find solutions or make decisions. These techniques are often applied in various AI applications, such as problem-solving, planning, game playing, path finding, and more.

In the field of artificial intelligence, various types of search algorithms are used to explore and find solutions in different problem-solving scenarios. These search algorithms can be categorized into three main types: uninformed search (blind search) and informed search (heuristic search) and adversarial search.

3.1. Uninformed Search (Blind Search):

Uninformed search algorithms explore the search space without using any additional information about the problem other than the initial state and the available actions. These algorithms are less efficient than informed search algorithms, but they guarantee completeness and can be useful when no heuristic information is available.

Some common types of uninformed search algorithms include:

a. Depth-First Search (DFS): Explores as far as possible along a branch before backtracking. It's memory-efficient but might not guarantee the shortest path.

b. Depth-Limited Search (DLS): A variant of DFS that limits the depth of exploration.

c. Breadth-First Search (BFS): Explores nodes level by level, moving outward from the start node. Guarantees the shortest path.

d.

e. Iterative Deepening Search (IDS): Combines features of BFS and DLS by performing a series of DLS searches with increasing depths.

3.11 Depth-First Search (DFS): - DFS explores as far as possible along each branch before backtracking. It starts at the root node and goes as deep as possible in the tree or graph before backtracking to explore other branches.

Depth-First Search (DFS) is a searching algorithm used in artificial intelligence and computer science to explore all the possible paths in a tree or graph data structure. It starts from a given source node and explores as far as possible along each branch before backtracking.

DFS Algorithm Steps:-

1. Initialize a stack to keep track of nodes to be explored.
2. Push the starting node onto the stack.
3. While the stack is not empty:

   a. Pop a node from the top of the stack. This node becomes the current node.

   b. If the current node is the goal node or the desired solution, terminate the search and return the solution.

   c. If the current node has not been visited:

      i. Mark the node as visited.

      ii. Expand the current node to find its neighboring nodes or children.

      iii. Push the unvisited neighboring nodes onto the stack.

   d. If all neighbors of the current node have been visited, backtrack by popping the next node from the stack and making it the current node.

Depth-First Search (DFS) is a powerful searching algorithm, but it also has several limitations and shortcomings that can affect its performance in certain scenarios:

1. Completeness: DFS is not guaranteed to find a solution if one exists. In certain cases, it might get stuck in infinite loops or explore only one branch of the search space.

2. Optimality: DFS does not guarantee finding the shortest path to the goal. It might find a solution but not necessarily the most optimal one.

3. Memory Usage: DFS can use a lot of memory in deeply branching or infinite search spaces. It stores a potentially large number of nodes on the stack during exploration, which can lead to stack overflow errors or excessive memory consumption.

4. Lack of Guidance: DFS does not use any information about the goal or the potential paths to the goal. It blindly explores down a path until it reaches a leaf node or encounters a dead-end, regardless of how promising that path might be.

5. Vulnerability to Infinite Paths: If there are infinite paths in the search space, DFS might get stuck exploring one of these paths indefinitely, never finding a solution or even terminating.

6. Non-Uniform Path Costs: In graphs with non-uniform edge costs, DFS may explore paths with higher costs before exploring paths with lower costs. This can lead to suboptimal solutions.

7. Backtracking Overhead: Backtracking in DFS can introduce overhead as nodes need to be revisited and re-expanded when alternative paths are explored. This can slow down the search process, especially in cases with deep paths.

8. Lack of Information Sharing: DFS operates independently on each path and doesn't communicate information between parallel paths. This can lead to redundancy in exploration and suboptimal use of computational resources.

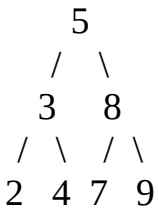
9. Limited Use in Search Trees: DFS can be problematic in search trees with many levels and a wide branching factor. It may exhaustively explore one branch before considering other branches, which can lead to inefficient searches.

10. No Guarantee of Shortest Path: DFS might find a solution early in the search, but that solution might not be the shortest path to the goal. It may need to continue searching to potentially find a better solution.

To mitigate these limitations, variations of DFS like Iterative Deepening Depth-First Search (IDDFS) and strategies like depth limits, cycle detection, and heuristics can be employed. Additionally, for certain problems, other searching algorithms like Breadth-First Search (BFS) or informed searches like A* can be more suitable, depending on the problem's characteristics and requirements.

A simple example of using Depth-First Search (DFS) in AI to search for a target value in a binary tree data structure.

Binary Tree Example:



DFS Algorithm Steps:

>Start at the root node (5).

>Check if the current node's value is the target value we are looking for. If yes, the search is successful, and we return the node.

>If the current node is not the target value, mark it as visited.

>Recursively apply DFS on the left child of the current node (3).

>If the left child doesn't contain the target value, backtrack and explore the right child of the current node (8).

>Continue this process until the target value is found or all nodes have been visited.

Searching for Target Value:

Let's say we are searching for the target value 7 in the binary tree.

>Start at the root node (5).

>Check the current node's value (5) - Not the target value, mark it as visited.

>Move to the left child (3).

>Check the current node's value (3) - Not the target value, mark it as visited.

>Move to the left child (2).

>Check the current node's value (2) - Not the target value, mark it as visited.

>No left child for node 2, backtrack to the parent (3) and explore its right child (4).

>Check the current node's value (4) - Not the target value, mark it as visited.

>No left or right child for node 4, backtrack to the parent (3).

>No right child for node 3, backtrack to the parent (5).

>Explore the right child of the root node (8).

>Check the current node's value (8) - Not the target value, mark it as visited.

>Move to the left child (7).

>Check the current node's value (7) - Found the target value (7), the search is successful, and we return the node.

Result:

The DFS algorithm successfully found the target value 7 in the binary tree and returned the corresponding node (7).

DFS explores as far as possible along each branch before backtracking, which can be efficient in certain cases. However, it's essential to consider the structure of the data and the specific problem requirements when choosing the appropriate searching algorithm.

DFS Program in Python

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def dfs_search(root, target):
    if root is None:
        return None

    if root.value == target:
        return root

    # Recursive DFS on the left subtree
    left_result = dfs_search(root.left, target)
    if left_result:
        return left_result

    # Recursive DFS on the right subtree
    right_result = dfs_search(root.right, target)
    if right_result:
        return right_result

    return None

# Creating a binary tree:
#         5
#       /   \
```

```
#      3     8
#     / \   / \
#    2   4 7   9

root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(8)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.left = TreeNode(7)
root.right.right = TreeNode(9)

target_value = 7
result_node = dfs_search(root, target_value)

if result_node:
    print(f"Target value {target_value} found in the binary tree.")
else:
    print(f"Target value {target_value} not found in the binary tree.")
```

OUTPUT

```
Target value 7 found in the binary tree.
```

In this example, we define a simple TreeNode class to represent nodes in the binary tree. The dfs_search function performs the Depth-First Search by recursively traversing the tree in a depth-first manner. If the target value is found, it returns the node containing the value; otherwise, it returns None.

Keep in mind that this is a basic demonstration of DFS for a binary tree. In real-world scenarios, data structures and algorithms for searching may vary depending on the complexity and structure of the problem.

3.12 Depth-Limited Search: - Depth-Limited Search is a variant of DFS that limits the maximum depth of the search. It helps control the search space when infinite paths exist or to improve efficiency in large state spaces.

Depth-Limited Search (DLS) is a variant of Depth-First Search (DFS), an uninformed search algorithm used in artificial intelligence to explore a graph or tree data structure. DLS limits the depth of exploration, preventing the algorithm from going beyond a certain depth level in the search tree. This limitation helps to avoid infinite loops and ensures that the algorithm terminates, even in cases where DFS might get stuck in an infinite branch.

Depth-Limited Search Algorithm Steps:

1. Initialize the search with the starting node and set the depth limit.
2. Perform a DFS up to the specified depth limit.
3. If the goal node is found within the depth limit, return success.
4. If the depth limit is reached and the goal is not found, return failure.
5. If the goal is not found at the current depth limit, increase the depth limit and repeat the search.

Depth-Limited Search Characteristics:

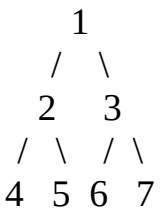
1. Completeness: DLS is not complete because it might miss a solution if it's beyond the specified depth limit. However, increasing the depth limit can increase the likelihood of finding a solution, at the cost of increased computational resources.
2. Optimality: DLS is not guaranteed to find the optimal solution. If the optimal solution is beyond the depth limit, DLS might terminate without finding it.
3. Time Complexity: The time complexity of DLS depends on the branching factor of the search tree, the depth limit, and the structure of the problem. Increasing the depth limit can significantly impact the time complexity.
4. Memory Usage: Like DFS, DLS uses memory to store the nodes on the current path. Memory usage increases with the depth limit and the depth of the search tree.

Applications of Depth-Limited Search in AI:

1. Solving puzzles and games where solutions are expected to be within a certain depth.
2. Navigating through decision trees or game trees, where deep exploration is unnecessary or impractical.
3. Exploring paths in situations where DFS might get stuck due to infinite branches.

Example of Depth-Limited Search:

Consider a binary tree where each node has two children, and the goal is to find a specific target value. Let's say we're performing a depth-limited search with a depth limit of 3:



In this example, if we're searching for the value "7" using a depth limit of 3, DLS would explore nodes up to depth 3 and stop once the limit is reached. It would traverse the nodes 1 → 3 → 7 and successfully find the target value.
Depth-Limited Search is useful when we have constraints on the depth of exploration, which can be common in scenarios where deep exploration is unnecessary or impractical due to time or memory constraints.

3.13 Breadth-First Search (BFS): - Breadth-First Search (BFS) is an most commonly used algorithm in artificial intelligence and computer science. Explores the graph or tree, starting from its base and expanding the process from layer to layer. BFS ensures that all nodes at the same level are checked before moving on to the next level node.

BFS Algorithm Steps:

1. Initialize a queue (FIFO) to keep track of nodes to be explored.
2. Enqueue the starting node into the queue.
3. While the queue is not empty:
    a. Dequeue a node from the front of the queue. This node becomes the current node.
    b. If the current node is the goal node or the desired solution, the search is successful.
    c. If the current node has not been visited:
        i. Mark the node as visited.
        ii. Enqueue all unvisited neighboring nodes of the current node into the queue.
    d. Repeat steps a to c until the queue is empty or the goal is found.

BFS Characteristics:

a. Completeness: BFS is guaranteed to find the shortest path in an unweighted graph or tree if one exists.
b. Optimality: BFS finds the shortest path, but it might be less efficient compared to other algorithms in terms of memory usage and time complexity.

c. Memory Usage: BFS uses more memory compared to Depth-First Search (DFS) as it stores all nodes at a given level before moving to the next level.

d. Time Complexity: The time complexity of BFS is generally proportional to the number of nodes in the graph or tree.

Applications of BFS in AI:

BFS has various applications in AI and computer science, including:

i. Solving puzzles and games where finding the shortest path is crucial.

ii. Navigating maze-like environments to find an exit.

iii. Performing shortest path routing in network systems.

iv. Traversing hierarchical structures like decision trees in AI planning.

v. Web crawling and searching in web-related applications.

Example of BFS: -

Consider a simple example of using BFS to find the shortest path from the starting point "S" to the goal point "G" in a grid:

S . . . . .
. # # . # .
. . # . # .
 # . # . # .
 # . . . # G

Starting from "S," BFS will explore the grid layer by layer until it finds the goal "G." The search will yield the shortest path in this case.

Python program

Simple Python program to that demonstrates the Breadth-First Search (BFS) algorithm to find the shortest path between two points in a grid-based environment. In this example, we'll consider a grid where "S" is the starting point and "G" is the goal point. We'll use BFS to find the shortest path from the starting point to the goal point.

```
from collections import deque

# Define the grid with "S" as the starting point and "G" as the goal point
grid = [
    ['S', '.', '.', '.', '.'],
```

```
    ['.', '#', '#', '.', '#'],
    ['.', '.', '#', '.', '#'],
    ['#', '.', '#', '.', '#'],
    ['#', '.', '.', '.', 'G']
]

# Define the dimensions of the grid
rows = len(grid)
cols = len(grid[0])

# Define possible moves: up, right, down, left
moves = [(-1, 0), (0, 1), (1, 0), (0, -1)]

def is_valid(x, y):
    return 0 <= x < rows and 0 <= y < cols and grid[x][y] != '#'

def bfs(start_x, start_y, goal_x, goal_y):
    queue = deque([(start_x, start_y)])
    visited = set([(start_x, start_y)])
    parent = {}

    while queue:
        x, y = queue.popleft()

        if x == goal_x and y == goal_y:
            break

        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if is_valid(new_x, new_y) and (new_x, new_y) not in visited:
                queue.append((new_x, new_y))
                visited.add((new_x, new_y))
                parent[(new_x, new_y)] = (x, y)

    if (goal_x, goal_y) not in parent:
        return None  # No path found

    path = [(goal_x, goal_y)]
    while (goal_x, goal_y) in parent:
        goal_x, goal_y = parent[(goal_x, goal_y)]
        path.append((goal_x, goal_y))

    return path[::-1]  # Reverse the path to start from "S"

# Find the shortest path from "S" to "G" using BFS
start_x, start_y = 0, 0
goal_x, goal_y = 4, 4
shortest_path = bfs(start_x, start_y, goal_x, goal_y)

if shortest_path:
```

```
    print("Shortest path:")
    for x, y in shortest_path:
        print(f"({x}, {y})", end=" ")
else:
    print("No path found.")
```

OUTPUT

Shortest path:

(0, 0) (1, 0) (2, 0) (3, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4)

In this example, the BFS algorithm explores the grid layer by layer, starting from "S," and finds the shortest path to the goal "G." The resulting path is printed, showing the sequence of coordinates that make up the shortest path.

Breadth-First Search (BFS) is a powerful searching algorithm with many advantages, it also comes with its own set of limitations and drawbacks that can impact its performance in certain scenarios:

1. Memory Usage: BFS can consume a significant amount of memory, especially in search spaces with a high branching factor or deep levels. It stores all nodes at a given level before moving on to the next level, which can lead to high memory requirements.

2. Time Complexity: In some cases, BFS can be slower than other algorithms due to its need to explore all nodes at each level before moving deeper. The time complexity can increase exponentially with the branching factor and depth of the search space.

3. Infinite Spaces: BFS is not suitable for infinite search spaces or spaces with cycles. It can get stuck in an infinite loop if there are cycles, which prevents it from finding a solution or terminating.

4. Lack of Guidance: BFS explores all nodes at the current level before moving to the next level. It doesn't take into account any information about the potential paths to the goal, which can lead to suboptimal paths in cases where some paths seem more promising.

5. Non-Uniform Path Costs: BFS treats all edges as having equal weight. In graphs with non-uniform edge costs, BFS might find a solution, but it might not be the shortest path.

6. Space for Explored Nodes: BFS keeps all explored nodes in memory until the search is complete, which can lead to memory constraints, especially if the search space is large.

7. Complex Data Structures: Implementing BFS efficiently often requires the use of data structures like queues, which can introduce overhead in terms of memory and computational resources.

8. Delayed Solutions: BFS might not find a solution quickly if the goal is located far away from the starting point. It needs to explore nodes layer by layer, and the shortest path may be located at a deeper level.

9. Lack of Parallelism: BFS doesn't naturally lend itself to parallel processing, as it relies on the sequential exploration of levels. This can limit its use in scenarios where parallelism is desirable.

To address these limitations, it's important to carefully consider the characteristics of the problem at hand and choose the appropriate searching algorithm. For problems with large search spaces, deep levels, or infinite paths, BFS might not be the optimal choice. In such cases, other algorithms like Depth-First Search (DFS), Iterative Deepening Depth-First Search (IDDFS), or informed searches like A* might provide better solutions or performance

3.13 a: Beam Search: - Beam Search is a variant of BFS that only keeps a fixed number of best candidates (called the beam width) at each level. It is commonly used in AI applications with large search spaces.

Beam Search is a heuristic search algorithm used in artificial intelligence and natural language processing. It's a search strategy that focuses on exploring a limited set of the most promising paths, aiming to find a solution more efficiently compared to traditional uninformed search algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS).

Beam Search Algorithm Steps:

1. Initialize the search with the starting node or state.
2. Create a beam or set that can hold a limited number of nodes or states, typically called the "beam width."
3. Generate successor nodes or states from the current node.
4. Evaluate the successor nodes using a heuristic function and select the top "beam width" nodes to keep in the beam.
5. If a goal state is found among the successor nodes, terminate and return the solution.
6. Repeat steps 3 to 5 for each level of the search tree, gradually expanding the search.

Beam Search Characteristics:

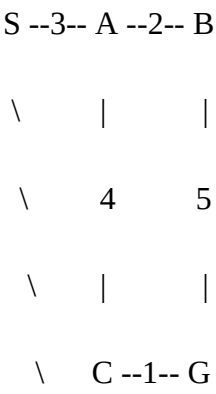
1. Completeness: Beam Search is not guaranteed to find a solution. It can miss solutions that are not among the top nodes in the beam.
2. Optimality: Beam Search does not guarantee finding the optimal solution. It often sacrifices optimality to gain efficiency by focusing on a limited set of paths.
3. Memory Usage: Beam Search limits memory usage by only keeping a small set of nodes in the beam. This can be advantageous in memory-constrained situations.
4. Time Complexity: The time complexity of Beam Search is influenced by the beam width and the branching factor of the search space. A wider beam explores more paths but also increases computational requirements.
5. Heuristic Dependency: The effectiveness of Beam Search relies on the quality of the heuristic function used to evaluate successor nodes. A good heuristic helps select promising paths.

Applications of Beam Search in AI:

1. Natural language processing tasks like language generation and machine translation.
2. Speech recognition, where hypotheses can be explored more efficiently using a limited set of possibilities.
3. Some optimization problems where exploring a large search space is impractical.

Example of Beam Search:

Consider a scenario where you are trying to find the shortest path from the starting city "S" to the goal city "G" in a graph. Each city is connected by edges with associated distances.



In this example, if you're using Beam Search with a beam width of 2, you would start from the "S" node, generate successor nodes, evaluate them using a heuristic, and select the top 2 nodes to keep in the beam. The algorithm would continue to explore the path that leads to "A" and "C," and eventually, it might find the optimal path "S" → "A" → "C" → "G."

Beam Search is a useful technique when you want a balance between exploring multiple paths and efficiency. However, keep in mind its limitations in terms of completeness and optimality.

3.14 Uniform Cost Search (Dijkstra's Algorithm): - Uniform Cost Search explores nodes based on their path cost from the start node. It guarantees finding the shortest path in a graph with non-negative edge weights.

UCS Algorithm Steps:

1. Initialize a priority queue (usually implemented with a min-heap) to keep track of nodes to be explored, with their path cost as the priority.
2. Enqueue the starting node with a path cost of 0 into the priority queue.
3. While the priority queue is not empty:
    a. Dequeue the node with the smallest path cost from the priority queue. This node becomes the current node.
    b. If the current node is the goal node, the search is successful.
    c. Otherwise, if the current node has not been visited:
        i. Mark the node as visited.
        ii. Enqueue all unvisited neighboring nodes with their updated path costs into the priority queue.

d. Repeat steps a to c until the goal is found or the priority queue is empty.

UCS Characteristics:

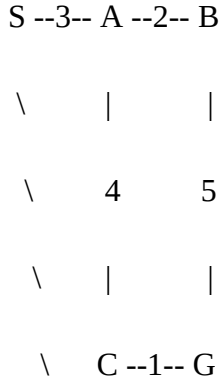
1. Completeness: UCS is complete and will find the shortest path if one exists, as long as the edge costs are non-negative.
2. Optimality: UCS guarantees finding the optimal solution, i.e., the shortest path with the lowest total cost.
3. Memory Usage: UCS stores nodes in the priority queue, which requires more memory than uninformed search algorithms like BFS or DFS.
4. Time Complexity: The time complexity of UCS can vary depending on the implementation, but in general, it's higher than BFS.

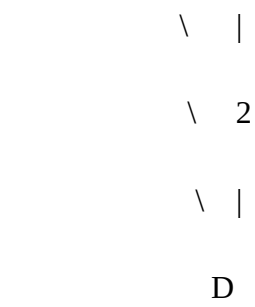
Applications of UCS in AI:

1. Pathfinding in games and robotics.
2. Network routing and transportation planning.
3. Shortest route computation in maps and navigation systems.
4. Resource allocation and scheduling problems.

Example of UCS (Dijkstra's Algorithm):

Consider a weighted graph where nodes represent cities, and edges represent distances between cities. We want to find the shortest path from the starting city "S" to the goal city "G." The edge weights represent distances between cities.





In this example, the UCS algorithm would prioritize exploring paths with lower costs, leading to the shortest path from "S" to "G" being found. The optimal path would be "S" -> "A" -> "C" -> "G" with a total cost of 6.

Dijkstra's Algorithm is a foundation for more advanced algorithms like A* and can be applied to various problems where finding the shortest path is crucial.

Python program

Simple Python program that implements Uniform Cost Search (Dijkstra's Algorithm) to find the shortest path between nodes in a graph. In this example, we'll represent the graph using an adjacency list and find the shortest path from a starting node to a goal node.

```
import heapq

# Define the graph as an adjacency list
graph = {
    'S': [('A', 3), ('D', 2)],
    'A': [('S', 3), ('B', 2), ('C', 4)],
    'B': [('A', 2), ('G', 5)],
    'C': [('A', 4), ('G', 1)],
    'D': [('S', 2), ('C', 2)],
    'G': [('B', 5), ('C', 1)]
}

def dijkstra(start, goal):
    # Priority queue to store nodes to be explored, with their path cost
    priority_queue = [(0, start)]
    # Dictionary to keep track of the shortest path distances
    shortest_distances = {node: float('inf') for node in graph}
    shortest_distances[start] = 0
    # Dictionary to store parent nodes for constructing the path
    parents = {}

    while priority_queue:
```

```
        cost, current = heapq.heappop(priority_queue)

        if current == goal:
            break

        if cost > shortest_distances[current]:
            continue

        for neighbor, edge_cost in graph[current]:
            new_cost = cost + edge_cost
            if new_cost < shortest_distances[neighbor]:
                shortest_distances[neighbor] = new_cost
                heapq.heappush(priority_queue, (new_cost, neighbor))
                parents[neighbor] = current

    # Construct the shortest path
    path = []
    node = goal
    while node != start:
        path.insert(0, node)
        node = parents[node]
    path.insert(0, start)
    return path

# Find the shortest path from 'S' to 'G' using Dijkstra's Algorithm
start_node = 'S'
goal_node = 'G'
shortest_path = dijkstra(start_node, goal_node)

print("Shortest path:", shortest_path)
```

OUTPUT

```
Shortest path: ['S', 'D', 'C', 'G']
```

In this example, the graph is represented as an adjacency list, where each node is associated with its neighboring nodes and the corresponding edge costs. The dijkstra function implements Dijkstra's Algorithm using a priority queue to explore nodes with the lowest path cost. The shortest path is constructed by tracing back from the goal node to the start node using the parents dictionary.

This program demonstrates the basic principles of Dijkstra's Algorithm for finding the shortest path in a graph. Keep in mind that this example uses a simplified graph, and in real-world scenarios, more complex data structures and optimizations might be needed.

Uniform Cost Search (UCS) also has certain limitations and drawbacks that can affect its applicability in specific scenarios:

1. Non-Negative Edge Costs: Dijkstra's Algorithm requires that all edge costs in the graph be non-negative. If negative edge costs are present, the algorithm can produce incorrect results.

2. Limited Applicability in Negative Weight Graphs: While Dijkstra's Algorithm can handle non-negative edge costs, it's not suitable for graphs with negative edge costs or cycles containing negative total costs. Negative edge costs can lead to incorrect shortest path calculations.

3. Infeasible for Large Graphs: Dijkstra's Algorithm can be inefficient for graphs with a large number of nodes and edges. Its time complexity is influenced by the number of nodes and edges, which can result in slow performance for large-scale problems.

4. Memory Usage: Dijkstra's Algorithm uses a priority queue to store nodes, which can require significant memory, especially in graphs with many nodes. In scenarios with limited memory, this can be a drawback.

5. Time Complexity: Although Dijkstra's Algorithm guarantees finding the shortest path, its time complexity can be relatively high in dense graphs with many nodes and edges. More efficient algorithms like A* can be more suitable in such cases.

6. Lack of Heuristics: Dijkstra's Algorithm doesn't utilize heuristics to guide the search. This means it might explore paths that appear promising due to lower edge costs, even if those paths lead further away from the goal.

7. Complex Edge Costs: In some scenarios, edge costs might not accurately represent the true cost or time of traversal. For example, in real-world transportation networks, traffic congestion and delays can impact the actual traversal time.

8. Pathfinding in Dynamic Environments: Dijkstra's Algorithm assumes a static graph with fixed edge costs. In dynamic environments where edge costs change over time (e.g., real-time traffic updates), the algorithm may not produce optimal results.

9. Efficiency in Sparse Graphs: In graphs with low branching factors and sparse connectivity, Dijkstra's Algorithm can be overkill. Other algorithms like BFS or A* might provide comparable or better performance.

To mitigate these limitations, it's important to carefully consider the characteristics of the problem and the graph structure. Depending on the nature of the problem, the presence of

negative edge costs, and the available resources, other algorithms like A* (informed search) or Bellman-Ford (for graphs with negative edge costs) might be more appropriate choices.

3.2. Informed Search (Heuristic Search):

Informed search algorithms make use of additional information beyond the current state and the goal, typically provided by a heuristic function. This extra information guides the search toward more promising paths and is particularly useful in solving complex problems efficiently.

Some common types of informed search algorithms include:

a. Greedy Best-First Search: Expands nodes based solely on the heuristic value (h-value), aiming to reach the goal quickly. Might not guarantee optimal solutions.

b. A Search:* Combines the cumulative cost to reach a node (g-value) and the heuristic estimate of the cost to reach the goal (h-value). It finds optimal solutions when using admissible and consistent heuristics.

c. IDA (Iterative Deepening A): A memory-efficient variant of A* that uses depth-first search while keeping the memory usage under control.

d. RBFS (Recursive Best-First Search): Expands nodes using a best-first approach but maintains backup values for efficient backtracking.

3.21 Greedy Best-First Search: - Greedy best first search is a search algorithm used in artificial intelligence and computer science. It differs from the best first search algorithm, which uses a heuristic function to estimate the distance from the current source to the destination. Heuristic function is used to guide the search towards the goal. It selects the node that appears to be closest to the goal based on the heuristic, without considering the path cost.

Greedy Best-First Search Algorithm Steps:

1. Initialize a priority queue with the starting node, where the priority is determined by the heuristic value (h) of the node (h(n)).
2. While the priority queue is not empty:
    a. Dequeue the node with the highest heuristic value.
    b. If the current node is the goal node, the search is successful.
    c. Otherwise, expand the current node by generating its child nodes.

d. Enqueue the child nodes into the priority queue, ordered by their heuristic values.

e. Repeat steps a to d until the goal is found or the priority queue is empty.

Greedy Best-First Search Characteristics:

1. Completeness: Greedy Best-First Search is not guaranteed to find a solution, as it can get stuck in local optima and ignore promising paths that might lead to the goal.

2. Optimality: Greedy Best-First Search is not guaranteed to find the optimal solution, even if a solution is found. It tends to prioritize nodes that appear promising based solely on the heuristic, which can lead to suboptimal paths.

3. Memory Usage: Similar to other informed search algorithms, Greedy Best-First Search stores nodes in a priority queue. The memory usage can be high, particularly if the heuristic values don't provide effective guidance.

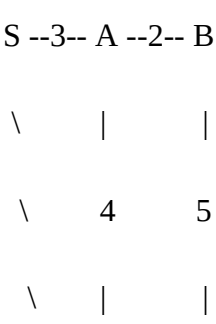
4. Time Complexity: The time complexity of Greedy Best-First Search can vary significantly depending on the quality of the heuristic function. In some cases, it might converge to a solution quickly, while in others, it might exhaustively explore parts of the search space.

Applications of Greedy Best-First Search in AI:

1. Path finding algorithms in games and robotics, where finding a solution quickly is more important than ensuring optimality.
2. Certain optimization problems where a local optimum is acceptable.

Example of Greedy Best-First Search:

Consider a simple graph where nodes represent cities, and the goal is to find the shortest path from the starting city "S" to the goal city "G." The edges between cities have weights representing distances.



\ C --1-- G

\ |

\ 2

In this example, Greedy Best-First Search would prioritize exploring paths that seem closer to the goal based on the heuristic (e.g., straight-line distance), even if the total path cost is higher. The algorithm might choose to explore "S" -> "A" -> "B" -> "G," which might not be the optimal path.

Python program that demonstrates the Greedy Best-First Search algorithm. In this example, we'll use a graph represented as an adjacency list and a heuristic function to estimate distances between nodes. We'll find the path from a starting node to a goal node using the Greedy Best-First Search algorithm.

```
import heapq

# Define the graph as an adjacency list
graph = {
    'S': [('A', 3), ('D', 2)],
    'A': [('S', 3), ('B', 2), ('C', 4)],
    'B': [('A', 2), ('G', 5)],
    'C': [('A', 4), ('G', 1)],
    'D': [('S', 2), ('C', 2)],
    'G': [('B', 5), ('C', 1)]
}

# Define the heuristic function (estimated distance to goal)
heuristic = {
    'S': 7,
    'A': 5,
    'B': 2,
    'C': 1,
    'D': 8,
    'G': 0
}

def greedy_best_first_search(start, goal):
    priority_queue = [(heuristic[start], start)]
    visited = set()

    while priority_queue:
```

```
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            return visited

        visited.add(current)

        for neighbor, _ in graph[current]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (heuristic[neighbor],
neighbor))

    return visited

# Find the path using Greedy Best-First Search
start_node = 'S'
goal_node = 'G'
visited_nodes = greedy_best_first_search(start_node, goal_node)

print("Visited nodes:", visited_nodes)
```

OUTPUT

```
Visited nodes: {'S', 'A', 'B', 'G'}
```

In this example, the heuristic function estimates the distance from each node to the goal node "G." The greedy_best_first_search function uses the heuristic values to prioritize exploring nodes that appear to be closer to the goal. The algorithm explores nodes in a greedy manner, which can lead to a suboptimal path, as it doesn't consider the total path cost.

Keep in mind that this program uses a simplified graph and heuristic function. In practice, the quality of the heuristic and the characteristics of the graph can significantly impact the performance and quality of the solution produced by Greedy Best-First Search.

Limitations and drawbacks that can impact GBFS applicability and performance:

1. Completeness: Greedy Best-First Search is not guaranteed to find a solution, even if one exists. It can get stuck in local optima or follow paths that seem promising based on the heuristic but lead to dead ends.

2. Optimality: Greedy Best-First Search does not guarantee finding the optimal solution. It tends to prioritize nodes that have low heuristic values, but these nodes might not necessarily lead to the best overall path.

3. Lack of Information: The algorithm relies heavily on the heuristic function, often ignoring other important factors such as the actual path cost or constraints. This can lead to suboptimal solutions if the heuristic is inaccurate or doesn't capture the full problem complexity.

4. Heuristic Dependency: The effectiveness of Greedy Best-First Search heavily relies on the quality of the heuristic function. If the heuristic function doesn't accurately estimate distances or costs, the algorithm's performance can be compromised.

5. Informed Bias: Because Greedy Best-First Search prioritizes nodes solely based on heuristic values, it might ignore paths that are initially more costly but ultimately lead to a better solution. This can result in suboptimal paths in some cases.

6. Sensitivity to Initial State: The initial state of the search can significantly affect the path chosen by the algorithm. Slight changes in the starting point can lead to entirely different paths being explored.

7. No Backtracking: Greedy Best-First Search does not backtrack or reconsider previously explored paths. Once it chooses a path, it doesn't reassess its decisions, which can lead to missed opportunities for finding better paths.

8. Exploration Imbalance: The algorithm can exhibit an exploration imbalance, favoring certain parts of the search space while ignoring others. This can lead to inefficient exploration and overlooked potential solutions.

9. High Memory Usage: Similar to other informed search algorithms, Greedy Best-First Search requires memory to store nodes and their heuristic values in the priority queue. This can lead to high memory usage, particularly in large search spaces.

10. Inefficiency in Certain Graphs: Greedy Best-First Search might not perform well in graphs with complex structures or when there are many paths to consider. It can focus on a single path and miss better alternatives.

Greedy Best-First Search is a quick heuristic-based algorithm that might work well in situations where fast decision-making is more important than finding the absolute best solution. However, its limitations, especially its lack of completeness and optimality guarantees, should be carefully considered when choosing it for a specific problem.

3.22 A Algorithm: - A* is an informed search algorithm that combines elements of Uniform Cost Search and Greedy Best-First Search. It uses a heuristic function to guide exploration while

considering both the path cost and the estimated cost to the goal. A* guarantees finding the shortest path if the heuristic is admissible.

A algorithm* in the context of artificial intelligence. A* (pronounced "A star") is a widely used informed search algorithm that efficiently finds the shortest path from a starting node to a goal node in a weighted graph. It combines the strengths of Dijkstra's Algorithm and Greedy Best-First Search by considering both the actual cost from the start node (the "g" value) and a heuristic estimate of the cost to reach the goal node (the "h" value).

A Algorithm Steps:*

1. Initialize two lists: an open list (to store nodes to be explored) and a closed list (to store nodes that have been visited).
2. Enqueue the starting node to the open list with a "g" value of 0 and calculate its heuristic "h" value.
3. While the open list is not empty:
    a. Dequeue the node with the lowest "f" value (f = g + h) from the open list. This node becomes the current node.
    b. If the current node is the goal node, the search is successful.
    c. Otherwise, generate the child nodes of the current node and calculate their "g" and "h" values.
    d. For each child node, if it's already in the open list with a lower "f" value or in the closed list, skip it.
    e. If the child node is not in the open list or has a higher "f" value, enqueue it to the open list.
    f. Move the current node to the closed list.
4. If the open list becomes empty and the goal node is not reached, there's no path.

A Algorithm Characteristics:

1. Completeness: A* is complete and will find the shortest path if one exists, as long as the heuristic is admissible (never overestimates the actual cost).

2. Optimality: A* guarantees finding the optimal solution if the heuristic is both admissible and consistent (satisfies the triangle inequality).

3. Memory Usage: A* uses memory to store nodes in the open and closed lists. Its memory usage can be influenced by the branching factor and graph size.

4. Time Complexity: The time complexity of A* depends on the efficiency of the heuristic function. In practice, it's often faster than uninformed search algorithms like Dijkstra's Algorithm.

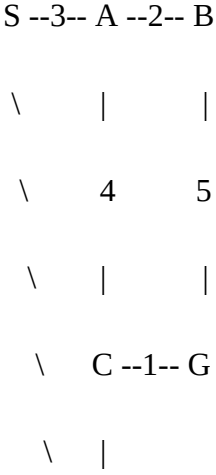
5. Heuristic Requirement: A* requires a heuristic function that provides a reasonable estimate of the cost from a given node to the goal. The quality of the heuristic can greatly impact the algorithm's performance.

Applications of A Algorithm in AI:

1. Pathfinding in games, robotics, and navigation systems.
2. Route planning for transportation networks.
3. Resource allocation and scheduling problems.
4. Graph-based optimization problems.

The A* algorithm efficiently combines informed and uninformed search strategies to find optimal solutions in a variety of scenarios. Its performance and effectiveness depend on the quality of the heuristic function and the characteristics of the problem's search space.

Python program that demonstrates the A* algorithm.



\ 2

\ |

D

In this example, we'll use a graph represented as an adjacency list, and we'll find the shortest path from a starting node to a goal node using the A* algorithm. The heuristic function used here estimates the straight-line distance between nodes.

```
import heapq

# Define the graph as an adjacency list with edge costs
graph = {
    'S': [('A', 3), ('D', 2)],
    'A': [('S', 3), ('B', 2), ('C', 4)],
    'B': [('A', 2), ('G', 5)],
    'C': [('A', 4), ('G', 1)],
    'D': [('S', 2), ('C', 2)],
    'G': [('B', 5), ('C', 1)]
}

# Define the heuristic function (straight-line distance to goal)
heuristic = {
    'S': 7,
    'A': 5,
    'B': 2,
    'C': 1,
    'D': 8,
    'G': 0
}

def a_star(start, goal):
    open_list = [(heuristic[start], 0, start)]  # (h, g, node)
    closed_list = set()
    parent = {start: None}
    g_value = {node: float('inf') for node in graph}
    g_value[start] = 0

    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == goal:
            return reconstruct_path(parent, goal)
```

```
        closed_list.add(current)

        for neighbor, edge_cost in graph[current]:
            tentative_g = g + edge_cost
            if tentative_g < g_value[neighbor]:
                parent[neighbor] = current
                g_value[neighbor] = tentative_g
                f_value = tentative_g + heuristic[neighbor]
                if neighbor not in closed_list:
                    heapq.heappush(open_list, (f_value, tentative_g,
neighbor))

    return None  # No path found

def reconstruct_path(parent, goal):
    path = [goal]
    while parent[goal] is not None:
        goal = parent[goal]
        path.insert(0, goal)
    return path

# Find the shortest path using A* algorithm
start_node = 'S'
goal_node = 'G'
shortest_path = a_star(start_node, goal_node)

if shortest_path:
    print("Shortest path:", shortest_path)
else:
    print("No path found.")
```

OUTPUT

Shortest path: ['S', 'A', 'C', 'G']

In this example, the A* algorithm uses the given heuristic function and edge costs to find the shortest path from the starting node "S" to the goal node "G." The priority queue open_list is used to prioritize nodes based on their "f" values, which are the sum of the actual path cost ("g" value) and the heuristic estimate ("h" value). The parent dictionary is used to reconstruct the shortest path once the goal is reached.

This program demonstrates the basic principles of the A* algorithm for finding optimal paths in a graph. Keep in mind that the effectiveness of A* heavily depends on the quality of the heuristic function and the characteristics of the problem's search space.

A* algorithm is a widely used and effective informed search algorithm, it's important to be aware of its limitations and potential drawbacks in certain scenarios:

1. Heuristic Quality: A* heavily relies on the quality of the heuristic function. If the heuristic is inaccurate or doesn't provide a good estimate of the actual cost, A* might not perform as expected. An admissible and consistent heuristic is crucial for optimality guarantees.

2. Completeness: A* is not guaranteed to find a solution if the search space is infinite or if there are cycles with negative edge costs. Additionally, if the heuristic overestimates the true cost, A* might not find a solution even if one exists.

3. Optimality and Heuristic Influence: A* guarantees optimal solutions when the heuristic is admissible and consistent. However, if the heuristic is not well-tuned or if it overestimates the true cost, A* might not produce the optimal path.

4. Time Complexity: The time complexity of A* can be high in some cases, particularly if the branching factor of the search space is large. The performance can be impacted by the heuristic quality and the structure of the graph.

5. Memory Usage: A* uses memory to store nodes in the open and closed lists. The memory consumption can be significant, especially for large search spaces, as it stores all generated nodes until a solution is found.

6. Duplicating States: A* can potentially generate and explore duplicate states (nodes with the same configuration). Proper handling of duplicate states is important to avoid inefficiencies.

7. Tie-Breaking: When multiple nodes have the same "f" value, A* requires a tie-breaking strategy to determine which node to expand first. The choice of tie-breaking strategy can impact the search's behavior.

8. Complex Heuristic Design: Designing a good heuristic function can be challenging. In some cases, obtaining accurate heuristics might be difficult or even impossible, limiting A*'s effectiveness.

9. Limited Exploration: A* focuses on exploring the most promising paths based on the heuristic. This can lead to limited exploration of other paths that might yield better solutions in certain cases.

10. Dynamic Environments: A* assumes a static environment where edge costs remain constant. In dynamic environments with changing costs, A* might not perform optimally.

11. Performance Trade-offs: In some cases, algorithms like Dijkstra's Algorithm or Greedy Best-First Search might be more suitable, depending on the specific problem's characteristics and requirements.

To address these limitations, it's important to carefully select or design an appropriate heuristic function and consider the nature of the problem at hand. In some cases, alternative search algorithms or modifications to A* might be better suited to overcome these limitations.

3.3. Adversarial search

Adversarial search, also known as game tree search, is a type of search algorithm used in artificial intelligence to make decisions in competitive environments, such as games. It involves predicting the opponent's moves and making optimal decisions to maximize one's own chances of winning. Adversarial search algorithms are particularly relevant in two-player games where opponents take turns to make moves.

The goal of adversarial search is to find the best move for the player in consideration, assuming the opponent will make moves that are strategically beneficial for them. The algorithms aim to explore the possible moves and outcomes in the game tree to make informed decisions.

Some common algorithms for adversarial search include:

a. Minimax Algorithm: The minimax algorithm is a fundamental technique in adversarial search. It works by recursively evaluating the possible outcomes of each move and assigning a value to each state. The player aims to maximize their own utility while assuming that the opponent aims to minimize it. The algorithm selects the move that leads to the maximum possible utility, considering the opponent's best responses.

b. Alpha-Beta Pruning: Alpha-beta pruning is an optimization technique used with the minimax algorithm to reduce the number of nodes that need to be evaluated. It eliminates

branches in the game tree that are guaranteed to not affect the final decision. This technique significantly speeds up the search process.

c. Monte Carlo Tree Search (MCTS): MCTS is an adaptive algorithm that combines random simulations and tree search. It is used in games with large state spaces and complex decision trees. MCTS has been particularly successful in games like Go and Chess.

d. Negamax Algorithm: Negamax is a simplification of the minimax algorithm. It takes advantage of the fact that the values of the game tree nodes have the same sign and are essentially negations of each other. Negamax simplifies the computation of the value of each node.

Adversarial search algorithms play a crucial role in game-playing AI systems, allowing them to make strategic decisions by considering both the player's and the opponent's potential moves and outcomes. These algorithms are used in games ranging from traditional board games to modern video games, and their principles have applications in various decision-making scenarios beyond gaming.

These search algorithms form the basis for solving a wide range of problems in AI, including pathfinding, puzzle solving, game playing, optimization, and more. The choice of algorithm depends on factors such as problem characteristics, available heuristic information, computational resources, and the desire for optimality or efficiency.

3.4 Iterative Deepening Depth-First Search (IDDFS): -

IDDFS is a combination of DFS and BFS. It repeatedly applies DFS with increasing depth limits until the goal is found. It has the advantages of both BFS (completeness) and DFS (space efficiency).

IDDFS Algorithm Steps:

1. Initialize the depth limit to 0.
2. Perform a DFS search with the given depth limit.
3. If the goal is not found at the current depth, increment the depth limit and repeat step 2.
4. Continue increasing the depth limit and performing DFS searches until the goal is found or a maximum depth is reached.

IDDFS Characteristics:

1. Completeness: IDDFS is complete, meaning it will find a solution if one exists, just like BFS and DFS.

2. Optimality: IDDFS, like BFS, guarantees finding the shortest path if one exists in an unweighted graph or tree.

3. Memory Usage: IDDFS uses memory comparable to DFS due to its depth-first nature and limited memory requirements.

4. Time Complexity: The time complexity of IDDFS is generally similar to BFS but can be higher in practice due to multiple iterations.

Advantages of IDDFS:

1. IDDFS combines the advantages of both DFS (memory efficiency) and BFS (optimal solution).

2. It gradually explores the search space, making it suitable for problems where the depth of the solution is unknown.

3. It avoids the excessive memory usage of BFS for deep search spaces.

Python program that demonstrates the Iterative Deepening Depth-First Search (IDDFS) algorithm. In this example, we'll use IDDFS to find the shortest path between a starting point "S" and a goal point "G" in a grid-based environment.

```
# Define the grid with "S" as the starting point and "G" as the goal point
grid = [
    ['S', '.', '.', '.', '.'],
    ['.', '#', '#', '.', '#'],
    ['.', '.', '#', '.', '#'],
    ['#', '.', '#', '.', '#'],
    ['#', '.', '.', '.', 'G']
]

# Define the dimensions of the grid
rows = len(grid)
cols = len(grid[0])

# Define possible moves: up, right, down, left
moves = [(-1, 0), (0, 1), (1, 0), (0, -1)]

def is_valid(x, y):
```

```
    return 0 <= x < rows and 0 <= y < cols and grid[x][y] != '#'

def iddfs(start_x, start_y, goal_x, goal_y, max_depth):
    for depth_limit in range(max_depth + 1):
        visited = set()
        if dls(start_x, start_y, goal_x, goal_y, depth_limit, visited):
            return True
    return False

def dls(x, y, goal_x, goal_y, depth_limit, visited):
    if depth_limit == 0 and x == goal_x and y == goal_y:
        return True
    if depth_limit > 0:
        visited.add((x, y))
        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if is_valid(new_x, new_y) and (new_x, new_y) not in visited:
                if dls(new_x, new_y, goal_x, goal_y, depth_limit - 1,
visited):
                    return True
    return False

# Find the shortest path from "S" to "G" using IDDFS
start_x, start_y = 0, 0
goal_x, goal_y = 4, 4
max_depth = 10  # Maximum depth limit
path_found = iddfs(start_x, start_y, goal_x, goal_y, max_depth)

if path_found:
    print("Path found!")
else:
    print("No path found.")
```

OUTPUT

```
Path found!
```

In this example, the IDDFS algorithm iteratively performs Depth-Limited Search (DLS) with increasing depth limits. It searches for a path from the starting point "S" to the goal point "G" in the grid. The dls function implements the Depth-Limited Search, and the iddfs function orchestrates the iterative process.

Keep in mind that IDDFS is generally useful for scenarios with large or unknown depths, where the goal might be located deep within the search space. The max_depth parameter sets an upper limit on the depth explored by the algorithm. It's important to choose an appropriate value for max_depth based on the problem's characteristics to balance exploration and performance.

Limitations of IDDFS:

1. IDDFS might revisit nodes multiple times, leading to potentially slower exploration compared to BFS.
2. It can be less efficient than other optimized search algorithms like A* if heuristics are available.

IDDFS is particularly useful in scenarios where memory is limited, and you want to ensure an optimal solution while avoiding the high memory requirements of BFS. However, its efficiency depends on the branching factor of the search tree and the specific problem at hand.

Implementing IDDFS often involves integrating DFS with a loop that iteratively increases the depth limit until a solution is found or a maximum depth is reached. This allows the algorithm to gradually explore the search space while maintaining low memory usage.

3.5 Some of the recently used algorithms in AI are

1. Monte Carlo Tree Search (MCTS): While MCTS has been around for a while, it continues to be an active area of research and is widely used in game-playing AI systems. MCTS is particularly effective in domains with large state spaces and complex decision trees. It's used in game AI, robotics, and other decision-making applications.

2. Reinforcement Learning (RL) and Deep Learning: Recent advancements in deep reinforcement learning have led to the development of more sophisticated search algorithms. These algorithms combine the power of deep neural networks with reinforcement learning techniques to search for optimal policies in complex environments.

3. AlphaGo and AlphaZero: These breakthrough algorithms, developed by DeepMind, combine Monte Carlo Tree Search with deep neural networks and reinforcement learning. They have demonstrated exceptional performance in playing complex games like Go and Chess, and they showcase the potential of combining different search and learning techniques.

4. Neural Architecture Search (NAS): While not a traditional search algorithm, NAS involves the automated design of neural network architectures. It uses search techniques to explore the vast space of possible network architectures to find those that perform well on specific tasks.

5. Differential Evolution: This optimization technique has gained popularity in solving complex optimization problems. It involves iteratively improving a population of candidate solutions by combining mutation, crossover, and selection operations.

6. Particle Swarm Optimization (PSO): PSO is a population-based optimization technique that simulates the social behavior of birds or fish to explore solution spaces. It's used in optimization tasks and has potential applications in various AI domains.

7. Hyperparameter Optimization: While not strictly a search algorithm, hyperparameter optimization techniques, such as Bayesian optimization and genetic algorithms, are used to find the best hyperparameters for machine learning models. These techniques aim to efficiently explore the hyperparameter space to improve model performance.

It's important to note that the latest advancements in search algorithms often leverage the power of machine learning and AI techniques to create more effective and efficient approaches. Research is ongoing, and new algorithms or improvements to existing ones can emerge rapidly in this field. If you're looking for the very latest developments, I recommend checking recent research papers, conference proceedings, and AI-related news sources.

3.6 Logic in AI

In artificial intelligence, logic forms the foundation for representing and reasoning about knowledge and information. There are two main branches of logic used in AI: propositional logic and predicate logic (also known as first-order logic).

1. Propositional Logic:

Propositional logic deals with propositions, which are statements that are either true or false. It involves creating logical expressions using logical connectives such as AND, OR, NOT, and implications (IF-THEN). In propositional logic, variables represent propositions, and truth values (true or false) are assigned to these variables.

Propositional logic is useful for representing simple relationships and making logical inferences, but it lacks the expressive power to handle more complex scenarios involving quantification and relationships between objects.

2. Predicate Logic (First-Order Logic):

Predicate logic is a more expressive form of logic that allows for the representation of relationships between objects, quantification, and more complex logical statements. It extends propositional logic by introducing predicates, functions, variables, and quantifiers.

>Predicates: Predicates represent relationships between objects. For example, "Larger(x, y)" could represent that "x" is larger than "y."

>Quantifiers: Quantifiers allow you to express statements about all or some objects. The two main quantifiers are:

>>Universal Quantifier (∀): "For all" or "Every." For example, ∀x Cat(x) represents "Every x is a cat."

>>Existential Quantifier (∃): "There exists" or "Some." For example, ∃x Dog(x) represents "There exists an x that is a dog."

>Functions: Functions can take objects as inputs and produce other objects as outputs. For example, Age(john) = 30 represents "The age of John is 30."

Predicate logic is more expressive and can represent complex relationships, make more nuanced inferences, and model a wider range of scenarios, making it a crucial tool in AI knowledge representation and reasoning.

Both propositional and predicate logic are used in various AI applications, such as knowledge representation, natural language processing, expert systems, automated reasoning, and more. The choice between the two depends on the complexity of the domain being modeled and the types of relationships that need to be captured.