B A Saran
22011102012
B.Tech CSE(IoT)-A

ARTIFICIAL INTELLIGENCE LAB

SEARCH ALGORITHMS

AIM: To implement the search algorithms.

ALGORITHM:

1. Depth-First Search (DFS)

1. Initialize a stack with the start node and an empty path.
2. Pop the last node from the stack.
3. If the node is the goal, return the path.
4. If unvisited, mark the node as visited and push its neighbors onto the stack.
5. Repeat until the goal is found or the stack is empty.

2. Breadth-First Search (BFS)

1. Initialize a queue with the start node and an empty path.
2. Dequeue the first node in the queue.
3. If the node is the goal, return the path.
4. If unvisited, mark the node as visited and enqueue its neighbors.
5. Repeat until the goal is found or the queue is empty.

3. British Museum Search (BMS)

1. Initialize a stack to keep track of paths and nodes to explore.
2. Pop the last node and its path from the stack.
3. If the node is the goal, add the path to the results list.
4. Push all unvisited neighbors of the current node onto the stack.
5. Repeat until all possible paths to the goal are found.

4. Hill Climbing (HC)

1. Initialize with the start node and empty path.
2. Check if the current node is the goal.
3. Add the node to the path and select the neighbor with the lowest heuristic.

4. Move to the selected neighbor, mark as visited, and repeat.
5. Stop if the goal is reached or no further moves are possible.

5. Beam Search (BS)

1. Initialize a beam with the start node.
2. Sort beam nodes by heuristic value and limit it to a predefined width.
3. For each node in the beam, add unvisited neighbors to the new beam.
4. If a goal node is reached, return the path.
5. Continue with the new beam until the goal is found.

6. Oracle Search

1. Initialize a stack to track paths and nodes.
2. Pop a node and its path from the stack.
3. If the node is the goal, store the path.
4. For each unvisited neighbor, add it to the stack with the updated path.
5. Repeat until all paths to the goal are found.

7. Oracle Search with Heuristics

1. Initialize a stack with paths and cumulative costs.
2. Pop the last node and path from the stack.
3. If the node is the goal, store the path and cost.
4. For each unvisited neighbor, calculate the new cost and add it to the stack.
5. Continue until all paths to the goal are explored.

8. Branch and Bound (BB)

1. Initialize a queue with the start node, path, and a cost of zero.
2. Dequeue the node with the lowest cost.
3. If the node is the goal and cost is lowest, update the best path.
4. Enqueue unvisited neighbors with updated cumulative costs.
5. Repeat until the goal with the lowest cost is found.

9. Extended Branch and Bound (EL)

1. Initialize a queue and visited list.
2. Dequeue the node with the lowest cost.

3. If the node is the goal with the lowest cost, update the best path.
4. Mark the node as visited, then enqueue neighbors with updated costs.
5. Repeat until the goal with the minimum cost is found.

10. Branch and Bound with Heuristics (EH)

1. Initialize a priority queue with the start node, path, and cost.
2. Dequeue the node with the lowest cost.
3. If the node is the goal with the best cost, save the path.
4. For each neighbor, calculate the new cost with the heuristic and enqueue.
5. Continue until the goal with the lowest cost is found.

11. A* Search

1. Initialize a priority queue with the start node and its heuristic cost.
2. Dequeue the node with the lowest total cost (cost + heuristic).
3. If the node is the goal, return the path.
4. For each neighbor, calculate the new cost and heuristic and update the queue.
5. Repeat until the goal with the lowest total cost is found.

12. Best-First Search

1. Initialize a priority queue with the start node based on its heuristic.
2. Dequeue the node with the lowest heuristic value.
3. If the node is the goal, return the path.
4. For each neighbor, add it to the queue based on heuristic value.
5. Repeat until the goal node is reached.

13. AO* Search

1. Initialize paths and costs with the start node.
2. Select the node with the lowest cost.
3. If the node is the goal, return the path.
4. Generate neighbors and update paths and costs.
5. Repeat until the goal path is found.

CODE:

```
def DFS(graph, start, goal):
    visited = set()
    stack = [(start, [start])]  # Stack for the DFS
    while stack:
        node, path = stack.pop()  # Get the last node added
        if node == goal:  # Check if we reached the goal
            return path
        if node not in visited:  # If not visited, mark it
            visited.add(node)
            # Add neighbors to the stack
            for neighbor in reversed(graph.get(node, {})):
                if neighbor not in visited:
                    stack.append((neighbor, path + [neighbor]))
    return None

def BFS(graph, start, goal):
    visited = set()
    queue = [(start, [start])]  # Queue for BFS
    while queue:
        node, path = queue.pop(0)  # Get the first node added
        if node == goal:  # Check if we reached the goal
            return path
        if node not in visited:  # If not visited, mark it
            visited.add(node)
            # Add neighbors to the queue
            for neighbor in graph.get(node, {}):
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))
    return None

def BMS(graph, start, goal):
    all_paths = []  # To store all paths found
```

```
    stack = [(start, [start])]  # Stack for BMS
    while stack:
        node, path = stack.pop()  # Get the last node added
        if node == goal:  # Check if we reached the goal
            all_paths.append(path)  # Store the path
        # Add neighbors to the stack
        for neighbor in graph.get(node, {}):
            if neighbor not in path:
                stack.append((neighbor, path + [neighbor]))
    return all_paths

def HC(graph, start, goal, heuristics):
    path = []  # To store the path taken
    visited = set()
    current_node = start  # Start from the initial node

    while current_node != goal:  # Until we reach the goal
        path.append(current_node)
        visited.add(current_node)

        # Get all unvisited neighbors
        neighbors = [n for n in graph.get(current_node, {}) if n not in visited]
        if not neighbors:  # No unvisited neighbors
            return None

        # Choose the neighbor with the lowest heuristic value only
        current_node = min(neighbors, key=lambda n: heuristics.get(n, float('inf')))

    path.append(goal)  # Add the goal to the path
    return path

def BS(graph, start, goal, heuristics, beam_width=3):
    beam = [(start, [start])]  # Initial beam
```

```
    while beam:
        # Sort the beam based on heuristic values and restrict to beam_width
        beam = sorted(beam, key=lambda x: heuristics.get(x[0], float('inf')))[:beam_width]
        new_beam = []

        for node, path in beam:
            if node == goal:  # Check if we reached the goal
                return path

            # Add neighbors to the new beam
            for neighbor in graph.get(node, {}):
                if neighbor not in path:  # Avoid revisiting nodes in the current path
                    new_beam.append((neighbor, path + [neighbor]))

        beam = new_beam  # Update the beam with new candidates

    return None

def Oracle(graph, start, goal):
    all_paths = []  # To store all paths found
    stack = [(start, [start])]  # Stack for Oracle
    while stack:
        node, path = stack.pop()  # Get the last node added
        if node == goal:  # Check if we reached the goal
            all_paths.append(path)  # Store the path
        # Add neighbors to the stack
        for neighbor in graph.get(node, {}):
            if neighbor not in path:
                stack.append((neighbor, path + [neighbor]))
    return all_paths

def OracleH(graph, start, goal, heuristics):
```

```
    all_paths = []  # To store all paths found
    stack = [(start, [start], 0)]  # Stack for Oracle with heuristics
    while stack:
        node, path, cost = stack.pop()  # Get the last node added
        if node == goal:  # Check if we reached the goal
            all_paths.append((path, cost))  # Store the path and cost
        # Add neighbors to the stack
        for neighbor in graph.get(node, {}):
            if neighbor not in path:
                stack.append((neighbor, path + [neighbor], cost + heuristics.get(neighbor,0)))  # Add cost of edge
    return all_paths

def BB(graph, start, goal):
    best_path = None  # To store the best path found
    best_cost = float('inf')  # Start with an infinitely large cost
    queue = [(0, start, [start])]  # Queue for Branch and Bound
    while queue:
        cost, node, path = queue.pop(0)  # Get the first node added
        if node == goal and cost < best_cost:  # Check if we reached the goal and if it's the best cost
            best_path = path
            best_cost = cost
        # Add neighbors to the queue
        for neighbor in graph.get(node, {}):
            if neighbor not in path:
                new_cost = cost + graph[node][neighbor]  # Use the edge weight
                queue.append((new_cost, neighbor, path + [neighbor]))
                queue.sort()  # Sort by cost
    return best_path

def EL(graph, start, goal):
    best_path = None  # To store the best path found
    best_cost = float('inf')  # Start with an infinitely large cost
    queue = [(0, start, [start])]  # Queue for Branch and Bound with Extended List
```

```
    visited = set()  # Set of visited nodes
    while queue:
        cost, node, path = queue.pop(0)  # Get the first node added
        if node == goal and cost < best_cost:  # Check if we reached the goal and if it's the best cost
            best_path = path
            best_cost = cost
        if node not in visited:  # If not visited, mark it
            visited.add(node)
            # Add neighbors to the queue
            for neighbor in graph.get(node, {}):
                if neighbor not in path:
                    new_cost = cost + graph[node][neighbor]  # Use the edge weight
                    queue.append((new_cost, neighbor, path + [neighbor]))
                    queue.sort()  # Sort by cost
    return best_path

def EH(graph, start, goal, heuristics):
    best_path = None  # To store the best path found
    best_cost = float('inf')  # Start with an infinitely large cost
    queue = [(0, start, [start])]  # Queue for Branch and Bound with Heuristics
    while queue:
        cost, node, path = queue.pop(0)  # Get the first node added
        if node == goal and cost < best_cost:  # Check if we reached the goal and if it's the best cost
            best_path = path
            best_cost = cost
        # Add neighbors to the queue
        for neighbor in graph.get(node, {}):
            if neighbor not in path:
                new_cost = cost + graph[node][neighbor] + heuristics.get(neighbor, 0)  # Use weight + heuristic
                queue.append((new_cost, neighbor, path + [neighbor]))
                queue.sort()  # Sort by cost
    return best_path
```

```
def Astar(graph, start, goal, heuristics):
    best_cost = {start: 0}  # Dictionary to track the best cost to each node
    queue = [(0 + heuristics.get(start, 0), start, [start])]  # Start with the start node

    while queue:
        queue.sort()
        cost, node, path = queue.pop(0)  # Get the node with the lowest cost

        if node == goal:  # Check if we reached the goal
            return path

        for neighbor in graph.get(node, {}):
            new_cost = best_cost[node] + graph[node][neighbor]  # Actual cost to reach the neighbor

            if neighbor not in best_cost or new_cost < best_cost[neighbor]:
                best_cost[neighbor] = new_cost
                total_cost = new_cost + heuristics.get(neighbor, 0)  # Total cost = actual cost + heuristic
                queue.append((total_cost, neighbor, path + [neighbor]))

    return None

def BestFirstSearch(graph, start, goal, heuristics):
    queue = [(heuristics.get(start, 0), start, [start])]  # Initialize the queue
    while queue:
        _, node, path = queue.pop(0)  # Get the first node added
        if node == goal:  # Check if we reached the goal
            return path
        for neighbor in graph.get(node, {}):
            if neighbor not in path:
                queue.append((heuristics.get(neighbor, 0), neighbor, path + [neighbor]))
                queue.sort()  # Sort by heuristic
    return None
```

```
def AOstar(graph, start, goal, heuristics):
    def recur_ao(node, path):
        # Check if we reached the goal
        if node == goal:
            return path + [node]

        # If the node has been expanded already, skip re-expanding it
        if node in solved_nodes:
            return path

        # Find all possible branches for the node and evaluate the best path
        best_subpath, min_cost = None, float('inf')

        for subpath in graph.get(node, []):
            sub_cost = 0
            sub_solution = []

            # Calculate the cost for each subpath
            for child in subpath:
                if child in heuristics:
                    sub_cost += heuristics[child]
                else:
                    # If there's no heuristic, assume a high default cost
                    sub_cost += float('inf')
                sub_solution.append(child)

            # Choose the path with the lowest cost
            if sub_cost < min_cost:
                min_cost = sub_cost
                best_subpath = sub_solution

        # Recursively expand the best subpath
        final_path = []
```

```
        for subnode in best_subpath:
            sub_result = recur_ao(subnode, path + [node])
            if sub_result:
                final_path.extend(sub_result)

        # Mark the node as solved and update its path
        solved_nodes.add(node)
        return final_path

    # Initialize the set of solved nodes
    solved_nodes = set()
    return recur_ao(start, [])

# Dictionary of algorithms
def select_search_algorithm(algorithm, graph, start, goal, heuristics=None):
    algorithms = {
        "DFS": DFS, "BFS": BFS, "BMS": BMS, "Hill Climbing": HC, "Beam Search": BS,
        "Oracle": Oracle, "Oracle with Heuristics": OracleH, "Branch and Bound": BB,
        "Branch and Bound Extended List": EL, "Branch and Bound with Heuristics": EH,
        "A* Search": Astar, "Best-First Search": BestFirstSearch, "AO* Search": AOstar
    }
    if algorithm in algorithms:
        if algorithm in ["Hill Climbing", "Beam Search", "Oracle with Heuristics", "Branch and Bound with
Heuristics", "A* Search", "Best-First Search"]:
            return algorithms[algorithm](graph, start, goal, heuristics)
        else:
            return algorithms[algorithm](graph, start, goal)
    else:
        print("Invalid algorithm selection.")
        return None

# Test execution
weighted_graph = {
```

```
    'S': {'A': 3, 'B': 5},
    'A': {'S': 3, 'B': 4, 'D': 3},
    'B': {'S': 5, 'A': 4, 'C': 4},
    'C': {'B': 4, 'E': 6},
    'D': {'A': 3, 'G': 5},
    'E': {'C': 6},
    'G': {'D': 5}
}

heuristics = {
    'S': 10, 'A': 7.5, 'B': 6, 'C': 7.5, 'D': 5, 'E': 4, 'G': 0
}

start_node = 'S'
goal_node = 'G'

print("Choose an algorithm:")
print("Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,")
print("         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,")
print("         A* Search, Best-First Search, AO* Search")
algorithm_choice = input("Enter the algorithm name: ")

path = select_search_algorithm(algorithm_choice, weighted_graph, start_node, goal_node, heuristics)
if path:
    print(f"Path found by {algorithm_choice}: {path}")
else:
    print(f"No path found by {algorithm_choice}")
```

OUTPUT:

DFS:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: DFS
Path found by DFS: ['S', 'A', 'D', 'G']
```

BFS:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: BFS
Path found by BFS: ['S', 'A', 'D', 'G']
PS C:\Users\saran\Downloads\AI-main\AI-main\CIA1>
```

BMS:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: BMS
Path found by BMS: [['S', 'B', 'A', 'D', 'G'], ['S', 'A', 'D', 'G']]
```

Hill Climbing:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Hill Climbing
Path found by Hill Climbing: ['S', 'B', 'A', 'D', 'G']
```

Beam Search:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Beam Search
Path found by Beam Search: ['S', 'A', 'D', 'G']
```

Oracle:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Oracle
Path found by Oracle: [['S', 'B', 'A', 'D', 'G'], ['S', 'A', 'D', 'G']]
```

Oracle with Heuristics:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Oracle with Heuristics
Path found by Oracle with Heuristics: [(['S', 'B', 'A', 'D', 'G'], 18.5), (['S', 'A', 'D', 'G'], 12.5)]
```

Branch and Bound:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Branch and Bound
Path found by Branch and Bound: ['S', 'A', 'D', 'G']
```

Branch and Bound Extended List:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Branch and Bound Extended List
Path found by Branch and Bound Extended List: ['S', 'A', 'D', 'G']
```

Branch and Bound with Heuristics:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Branch and Bound with Heuristics
Path found by Branch and Bound with Heuristics: ['S', 'A', 'D', 'G']
```

A* Search:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: A* Search
Path found by A* Search: ['S', 'A', 'D', 'G']
```

Best-First Search:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: Best-First Search
Path found by Best-First Search: ['S', 'A', 'D', 'G']
```

AO* Search:

```
Choose an algorithm:
Options: DFS, BFS, BMS, Hill Climbing, Beam Search, Oracle, Oracle with Heuristics,
         Branch and Bound, Branch and Bound Extended List, Branch and Bound with Heuristics,
         A* Search, Best-First Search, AO* Search
Enter the algorithm name: AO* Search
Path found by AO* Search: ['S', 'A', 'D', 'G']
```

<u>RESULT:</u> All the search algorithms are implemented.