Sachita S Limaye
2320030134
Section-6

Start coding or generate with AI.

AIML PYTHON FILE (ALGORITHMS)

```
#bfs code:
graph={
    '5':['3','7'],
    '3':['2','4'],
    '7':['8'],
    '2':[],
    '4':['8'],
    '8':[]
}

def bfs(visited,graph,node):
    visited.append(node)
    queue.append(node)

    while queue:
        m=queue.pop(0)
        print(m,end=" ")

        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

visited=[]
queue=[]

print("Graph using BFS:")
bfs(visited, graph,'5')
```

```
Graph using BFS:
5 3 7 2 4 8
```

```
# DFS code
def dfs(visited, graph, node):
    stack=[]
    stack.append(node)

    while stack:
        m=stack.pop()
        if m not in visited:
            print(m,end=" ")
            visited.append(m)
        for neighbour in graph[m]:
            if neighbour not in visited:
                stack.append(neighbour)
visited=[]

print("Graph using DFS:")
dfs(visited, graph,'5')
```

```
Graph using DFS:
5 7 8 3 4 2
```

```
# Iterative deepning search code:
def dls(node, depth, graph, visited):
    if depth == 0:
        return False
    if node not in visited:
        print(node, end=" ")
        visited.append(node)
    for neighbour in graph[node]:
        if neighbour not in visited:
            if dls(neighbour, depth - 1, graph, visited):
                return True
    return False

def ids(graph, start, max_depth):
    for depth in range(1, max_depth + 1):
        print(f"\nDepth: {depth}")
        visited = []
        if dls(start, depth, graph, visited):
            return True
    return False
start_node = '5'
max_depth = 4
ids(graph, start_node, max_depth)
```

```
Depth: 1
5
Depth: 2
5 3 7
Depth: 3
5 3 2 4 7 8
Depth: 4
5 3 2 4 8 7
False
```

```
import heapq

def best_first_search(graph, start, goal, h):
    open_list = []
    heapq.heappush(open_list, (h[start], start))
    visited = set()

    while open_list:
        _, current_node = heapq.heappop(open_list)
        print(current_node, end=" ")

        if current_node == goal:
            print("\nGoal found!")
            return True

        visited.add(current_node)

        for neighbor, cost in graph[current_node]:
            if neighbor not in visited:
                heapq.heappush(open_list, (h[neighbor], neighbor))

    print("\nGoal not found.")
    return False
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 5), ('E', 2)],
    'C': [('F', 2)],
    'D': [],
    'E': [('G', 5)],
    'F': [('G', 1)],
    'G': []
}

h = {
    'A': 7,
    'B': 6,
    'C': 4,
    'D': 9,
    'E': 5,
    'F': 3,
    'G': 0
}

print("Best First Search:")
best_first_search(graph, 'A', 'G', h)
```

```
Best First Search:
A C F G
Goal found!
True
```

```
import heapq

def a_star_search(graph, start, goal, h):
    open_list = []
    heapq.heappush(open_list, (0 + h[start], 0, start))
    came_from = {}
    cost_so_far = {start: 0}

    while open_list:
        _, current_cost, current_node = heapq.heappop(open_list)
        print(current_node, end=" ")

        if current_node == goal:
            print("\nGoal found!")
            return True

        for neighbor, cost in graph[current_node]:
            new_cost = current_cost + cost
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + h[neighbor]
                heapq.heappush(open_list, (priority, new_cost, neighbor))
                came_from[neighbor] = current_node

    print("\nGoal not found.")
    return False

graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 5), ('E', 2)],
    'C': [('F', 2)],
    'D': [],
    'E': [('G', 5)],
    'F': [('G', 1)],
    'G': []
}

h = {
    'A': 7,
    'B': 6,
    'C': 4,
    'D': 9,
    'E': 5,
    'F': 3,
    'G': 0
}

print("A* Search:")
a_star_search(graph, 'A', 'G', h)
```

```
A* Search:
A B C E F G
Goal found!
True
```

```
def minimax(position, depth, is_maximizing, scores, max_depth):
    if depth == max_depth:
        return scores[position]

    if is_maximizing:
        best_score = float('-inf')
        for child in get_children(position):
            score = minimax(child, depth + 1, False, scores, max_depth)
            best_score = max(best_score, score)
        return best_score
    else:
        best_score = float('inf')
        for child in get_children(position):
            score = minimax(child, depth + 1, True, scores, max_depth)
            best_score = min(best_score, score)
        return best_score

scores = {'A': 3, 'B': 5, 'C': 2, 'D': 9, 'E': 12, 'F': 5, 'G': 23}
max_depth = 2

def get_children(node):
    if node == 'A':
        return ['B', 'C']
    elif node == 'B':
        return ['D', 'E']
    elif node == 'C':
        return ['F', 'G']
    else:
        return []

print("Minimax Score:", minimax('A', 0, True, scores, max_depth))
```

```
Minimax Score: 9
```

```
def alpha_beta(position, depth, alpha, beta, is_maximizing, scores, max_depth):
    if depth == max_depth:
        return scores[position]

    if is_maximizing:
        max_eval = float('-inf')
        for child in get_children(position):
            eval = alpha_beta(child, depth + 1, alpha, beta, False, scores, max_depth)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in get_children(position):
            eval = alpha_beta(child, depth + 1, alpha, beta, True, scores, max_depth)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

scores = {'A': 3, 'B': 5, 'C': 2, 'D': 9, 'E': 12, 'F': 5, 'G': 23}
max_depth = 2

def get_children(node):
    if node == 'A':
        return ['B', 'C']
    elif node == 'B':
        return ['D', 'E']
    elif node == 'C':
        return ['F', 'G']
    else:
        return []

print("Alpha-Beta Score:", alpha_beta('A', 0, float('-inf'), float('inf'), True, scores, max_depth))
```

```
Alpha-Beta Score: 9
```