Selected files

9 printable files

```
practical 10-fuzzylogic.py
practical1 - bfs.py
PRACTICAL2-dfs.py
practical3 - itterative dfs.py
practical4-dls.py
practical5-A-star.py
practical6-minmax.py
practical8-alpha-beta pruning.py
practical9-warerjug.py
practical 10-fuzzylogic.py
 1 # Triangular code
 2
 3
   import numpy as np
   import matplotlib.pyplot as plt
 4
 5
 6
   def triangular(x, a, b, c):
 7
        return np.maximum(np.minimum((x-a)/(b-a), (c-x)/(c-b)), \theta)
 8
9
   x = np.linspace(0, 10, 1000)
   a, b, c = 1, 8, 10
10
11
   y = triangular(x, a, b, c)
12
   plt.plot(x, y, label=f'Triangular(a={a}, b={b}, c={c})')
13
14
   plt.xlabel('x')
15 plt.ylabel('Membership degree')
16 plt.title('Triangular Membership Function')
17
   plt.legend()
18 plt.grid(True)
19
   plt.show()
20
21
   #Gaussian Function
22
23
   import numpy as np
24
   import matplotlib.pyplot as plt
25
26
   def gaussian(x, mu, sigma):
27
        return np.exp(-0.5 * ((x - mu) / sigma) ** 2)
28
29
   x = np.linspace(0, 10, 1000)
30
   mu, sigma = 5, 1.5
31
   y = gaussian(x, mu, sigma)
32
   plt.plot(x, y, label=f'Gaussian(mu={mu}, sigma={sigma})')
33
34
   plt.xlabel('x')
35
   plt.ylabel('Membership degree')
   plt.title('Gaussian Membership Function')
36
37
   plt.legend()
   plt.grid(True)
39
   plt.show()
40
41
   #Trapezoid Function
42
43
   import numpy as np
   import matplotlib.pyplot as plt
```

```
45
46
   def trapezoid(x, a, b, c, d):
47
        return np.maximum(np.minimum(np.minimum((x-a)/(b-a), 1), (d-x)/(d-c)), 0)
48
49
   x = np.linspace(0, 10, 1000)
   a, b, c, d = 2, 4, 6, 8
50
51
   y = trapezoid(x, a, b, c, d)
52
53
   plt.plot(x, y, label=f'Trapezoid(a={a}, b={b}, c={c}, d={d})')
54
   plt.xlabel('x')
55 plt.ylabel('Membership degree')
   plt.title('Trapezoid Membership Function')
56
57
   plt.legend()
58
   plt.grid(True)
59
   plt.show()
60
practical1 - bfs.py
   from collections import defaultdict
 1
 2
 3
   class Graph:
 4
        def _ init (self):
 5
            self.graph = defaultdict(list)
 6
 7
        def add_edge(self, u, v):
 8
            self.graph[u].append(v)
9
        def bfs(self, start):
10
11
            visited = set()
            queue = [start]
12
13
            visited.add(start)
14
            while queue:
15
                vertex = queue.pop(0)
16
                print(vertex, end=' ')
17
18
19
                for neighbor in self.graph[vertex]:
20
                    if neighbor not in visited:
21
                        queue.append(neighbor)
22
                        visited.add(neighbor)
23
24
   # Example usage
   if name == " main ":
25
        graph = Graph()
26
27
        graph.add_edge(0, 1)
28
        graph.add_edge(0, 2)
29
        graph.add_edge(1, 2)
30
        graph.add_edge(2, 0)
31
        graph.add_edge(2, 3)
32
        graph.add_edge(3, 3)
33
34
        print("BFS Traversal starting from vertex 2:")
35
        graph.bfs(2)
36
PRACTICAL2-dfs.py
   from collections import defaultdict
 1
 2
 3 class Graph:
```

```
4
        def __init__(self):
 5
            self.graph = defaultdict(list)
 6
 7
        def add edge(self, u, v):
 8
            self.graph[u].append(v)
 9
10
        def dfs_util(self, vertex, visited):
11
            visited.add(vertex)
            print(vertex, end=' ')
12
13
14
            for neighbor in self.graph[vertex]:
                if neighbor not in visited:
15
                     self.dfs_util(neighbor, visited)
16
17
        def dfs(self, start):
18
            visited = set()
19
            self.dfs_util(start, visited)
20
21
22
    # Example usage
23
    if __name__ == "__main__":
24
        graph = Graph()
25
        graph.add_edge(0, 1)
26
        graph.add_edge(0, 2)
27
        graph.add_edge(1, 2)
28
        graph.add_edge(2, 0)
29
        graph.add edge(2, 3)
30
        graph.add_edge(3, 3)
31
32
        print("DFS Traversal starting from vertex 2:")
33
        graph.dfs(2)
34
practical3 - itterative dfs.py
    from collections import defaultdict
 1
 2
 3
    class Graph:
 4
        def __init__(self):
 5
            self.graph = defaultdict(list)
 6
 7
        def add edge(self, u, v):
 8
            self.graph[u].append(v)
 9
        def dfs(self, start):
10
            visited = set()
11
            stack = [start]
12
13
            visited.add(start)
14
15
            while stack:
                vertex = stack.pop()
16
17
                print(vertex, end=' ')
18
19
                for neighbor in reversed(self.graph[vertex]):
20
                     if neighbor not in visited:
21
                         stack.append(neighbor)
22
                         visited.add(neighbor)
23
24
   # Example usage
25
    if __name__ == "__main__":
26
        graph = Graph()
```

```
27
        graph.add_edge(0, 1)
28
        graph.add_edge(0, 2)
29
        graph.add_edge(1, 2)
30
        graph.add_edge(2, 0)
31
        graph.add_edge(2, 3)
32
        graph.add_edge(3, 3)
33
34
        print("DFS Traversal starting from vertex 2 (Iterative):")
35
        graph.dfs(2)
36
practical4-dls.py
   from collections import defaultdict
 2
 3
    graph = {
         'A': ['C', 'D'],
 4
 5
        'C': ['F', 'G'],
        'D': ['I', 'E'],
 6
        'E': ['J', 'K'],
 7
 8
        'F': ['L', 'M'],
9
        'G': ['N', 'O']
10
11
   }
12
13
   def DLS(start, goal, max_depth, depth=0, path=None):
14
        if path is None:
15
            path = []
16
17
        path.append(start)
18
        if start == goal:
19
20
            return path
21
22
        if depth == max depth:
            return False
23
24
25
        for neighbor in graph[start]:
            result = DLS(neighbor, goal, max_depth, depth + 1, path)
26
27
            if result:
28
                return result
29
30
        path.pop()
        return False
31
32
33
   start node = 'A'
   goal_node = input('Enter the goal node: ')
34
   max_depth = int(input("Enter the maximum depth limit: "))
35
36
   print()
37
38
   path_to_goal = DLS(start_node, goal_node, max_depth)
39
40
   if path_to_goal:
41
        print("Path to goal node available:", path_to_goal)
42
   else:
43
        print("No path available for the goal node within the given depth limit.")
practical5-A-star.py
 1
   import heapq
 2
```

```
3
    def a_star(start, goal, graph, heuristic):
 4
 5
        open_list = []
 6
        heapq.heappush(open_list, (0 + heuristic(start), 0, start, [start]))
 7
        closed_set = set()
 8
 9
        while open_list:
10
11
            _, g, current, path = heapq.heappop(open_list)
12
13
            if current == goal:
14
                return path
15
16
            closed_set.add(current)
17
18
19
            for neighbor, cost in graph[current].items():
20
                 if neighbor in closed_set:
21
                     continue
22
23
24
                g_{temp} = g + cost
25
                f_temp = g_temp + heuristic(neighbor)
26
27
                if neighbor not in [i[2] for i in open list]:
28
29
                     heapq.heappush(open_list, (f_temp, g_temp, neighbor, path + [neighbor]))
30
                else:
31
32
                     for open_node in open_list:
33
                         if open_node[2] == neighbor and open_node[1] > g_temp:
34
                             open_list.remove(open_node)
35
                             heapq.heappush(open_list, (f_temp, g_temp, neighbor, path +
    [neighbor]))
                             break
36
37
38
        return None
39
40
41
    graph = {
42
        'AA': {'BB': 1, 'CC': 3},
        'BB': {'AA': 1, 'DD': 1, 'EE': 5},
43
        'CC': {'AA': 3, 'FF': 5},
44
        'DD': {'BB': 1, 'FF': 1},
45
        'EE': {'BB': 5, 'FF': 2},
46
        'FF': {'CC': 5, 'DD': 1, 'EE': 2}
47
48
    }
49
50
51
    def heuristic(node):
52
        heuristics = {
53
            'AA': 10,
             'BB': 8,
54
55
             'CC': 5,
             'DD': 7,
56
             'EE': 3,
57
             'FF': 0
58
59
        }
60
61
        return heuristics[node]
```

```
62
63
    start_node = 'AA'
64
    goal_node = 'FF'
65
    path = a star(start node, goal node, graph, heuristic)
66
    print(f"Path from {start_node} to {goal_node}: {path}")
67
practical6-minmax.py
    terminal_nodes = {
 1
 2
        'H': -2,
 3
        'I': 3,
        'J': 1,
 4
        'K': 5,
 5
 6
        'L': -4,
 7
        'M': -6,
        'N': 2,
 8
        '0': 8
 9
10
    }
11
12
13
   tree = {
        'A': ['C', 'D'],
14
        'C': ['F', 'G'],
15
        'D': ['I', 'E'],
16
        'E': ['J', 'K'],
17
18
        'F': ['L', 'M'],
19
        'G': ['N', 'O']
20
    }
21
    def minimax(node, depth, is_maximizing_player):
22
23
        if node in terminal_nodes:
24
            return terminal nodes[node]
25
        if is maximizing player:
26
            best_value = float('-inf')
27
            for child in tree[node]:
28
29
                value = minimax(child, depth + 1, False)
30
                best_value = max(best_value, value)
31
            return best_value
32
        else:
            best_value = float('inf')
33
            for child in tree[node]:
34
                value = minimax(child, depth + 1, True)
35
36
                best_value = min(best_value, value)
37
            return best value
38
    def find_optimal_path(node, is_maximizing_player):
39
40
        optimal_value = minimax(node, 0, is_maximizing_player)
        path = [node]
41
42
43
        while node in tree:
44
            if is_maximizing_player:
45
                best_value = float('-inf')
                best node = None
46
47
                for child in tree[node]:
48
                    value = minimax(child, 0, not is_maximizing_player)
49
                     if value > best_value:
50
                         best_value = value
51
                         best_node = child
```

```
52
            else:
53
                best_value = float('inf')
54
                best_node = None
55
                for child in tree[node]:
56
                    value = minimax(child, 0, not is_maximizing_player)
57
                    if value < best_value:</pre>
58
                         best value = value
59
                         best_node = child
60
            path.append(best node)
            node = best node
61
            is_maximizing_player = not is_maximizing_player
62
63
        return optimal_value, path
64
65
    optimal value, optimal path = find optimal path('A', True)
66
67
    print(f"The optimal value is {optimal_value} and the optimal path is {' -> '
68
    .join(optimal_path)}")
69
practical8-alpha-beta pruning.py
 1
    import sys
 2
 3
   MAX = sys.maxsize
 4
   MIN = -sys.maxsize - 1
 5
    def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
 6
 7
        if depth == 3:
 8
            return values[nodeIndex]
 9
10
        if maximizingPlayer:
            best = MIN
11
12
            for i in range(2):
                val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
13
14
                best = max(best, val)
15
                alpha = max(alpha, best)
16
                if beta <= alpha:</pre>
                    break
17
18
            return best
19
        else:
20
            best = MAX
21
            for i in range(2):
                val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
22
23
                best = min(best, val)
                beta = min(beta, best)
24
                if beta <= alpha:</pre>
25
                     break
26
27
            return best
28
    if name == " main ":
29
30
31
        values = list(map(int, input("Enter values separated by space: ").split()))
        print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
32
33
34
35
```

36

```
from collections import deque
 1
 2
 3
    def pour(state, jug1, jug2):
 4
 5
        amt = min(state[jug1], (jug_caps[jug2] - state[jug2]))
 6
        new_state = list(state)
 7
        new_state[jug1] -= amt
 8
        new_state[jug2] += amt
 9
        return tuple(new state)
10
11
    def get_successors(state):
12
13
        successors = []
14
        for jug1, jug2 in [(0, 1), (1, 0)]:
            new_state = pour(state, jug1, jug2)
15
            if new_state != state:
16
17
                successors.append(new_state)
18
19
        for jug in [0, 1]:
20
            new state = list(state)
21
            new state[jug] = jug caps[jug]
22
            successors.append(tuple(new_state))
23
24
        for jug in [0, 1]:
25
            new_state = list(state)
26
            new_state[jug] = 0
27
            successors.append(tuple(new_state))
28
29
        return successors
30
31
    def heuristic(state, goal):
32
33
        return sum(abs(state[i] - goal[i]) for i in range(len(state)))
34
35
    def a_star(start, goal):
36
37
        open list = [(heuristic(start, goal), start)]
38
        print(open_list)
39
        closed_list = set()
        parent = {start: None}
40
41
        while open list:
            _, curr_state = open_list.pop(0)
42
43
            if curr_state == goal:
44
                path = deque()
45
                state = curr_state
                while state is not None:
46
                     path.appendleft(state)
47
                     state = parent[state]
48
49
                return list(path)
50
            closed_list.add(curr_state)
            for succ_state in get_successors(curr_state):
51
52
                if succ_state not in closed_list:
53
                     succ cost = heuristic(succ state, goal)
                     open_list.append((succ_cost, succ_state))
54
55
                    open list.sort()
56
                     parent[succ_state] = curr_state
57
        return None
58
59
    jug_{caps} = (4, 3)
60 start_state = (0, 0)
```

```
61  goal_state = (2, 0)
62
63  solution = a_star(start_state, goal_state)
64  if solution:
65     print("Solution:")
66     for state in solution:
67         print(state)
68  else:
69     print("No solution exists.")
70
```