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

```python
# Triangular code

import numpy as np
import matplotlib.pyplot as plt

def triangular(x, a, b, c):
    return np.maximum(np.minimum((x-a)/(b-a), (c-x)/(c-b)), 0)

x = np.linspace(0, 10, 1000)
a, b, c = 1, 8, 10
y = triangular(x, a, b, c)

plt.plot(x, y, label=f'Triangular(a={a}, b={b}, c={c})')
plt.xlabel('x')
plt.ylabel('Membership degree')
plt.title('Triangular Membership Function')
plt.legend()
plt.grid(True)
plt.show()

#Gaussian Function

import numpy as np
import matplotlib.pyplot as plt

def gaussian(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu) / sigma) ** 2)

x = np.linspace(0, 10, 1000)
mu, sigma = 5, 1.5
y = gaussian(x, mu, sigma)

plt.plot(x, y, label=f'Gaussian(mu={mu}, sigma={sigma})')
plt.xlabel('x')
plt.ylabel('Membership degree')
plt.title('Gaussian Membership Function')
plt.legend()
plt.grid(True)
plt.show()

#Trapezoid Function

import numpy as np
import matplotlib.pyplot as plt
```

```python

def trapezoid(x, a, b, c, d):
    return np.maximum(np.minimum(np.minimum((x-a)/(b-a), 1), (d-x)/(d-c)), 0)

x = np.linspace(0, 10, 1000)
a, b, c, d = 2, 4, 6, 8
y = trapezoid(x, a, b, c, d)

plt.plot(x, y, label=f'Trapezoid(a={a}, b={b}, c={c}, d={d})')
plt.xlabel('x')
plt.ylabel('Membership degree')
plt.title('Trapezoid Membership Function')
plt.legend()
plt.grid(True)
plt.show()

```

practical1 - bfs.py

```python
from collections import defaultdict

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

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

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

        while queue:
            vertex = queue.pop(0)
            print(vertex, end=' ')

            for neighbor in self.graph[vertex]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)

# Example usage
if __name__ == "__main__":
    graph = Graph()
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(1, 2)
    graph.add_edge(2, 0)
    graph.add_edge(2, 3)
    graph.add_edge(3, 3)

    print("BFS Traversal starting from vertex 2:")
    graph.bfs(2)

```

PRACTICAL2-dfs.py

```python
from collections import defaultdict

class Graph:
```

```python
    def __init__(self):
        self.graph = defaultdict(list)

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

    def dfs_util(self, vertex, visited):
        visited.add(vertex)
        print(vertex, end=' ')

        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)

    def dfs(self, start):
        visited = set()
        self.dfs_util(start, visited)

# Example usage
if __name__ == "__main__":
    graph = Graph()
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(1, 2)
    graph.add_edge(2, 0)
    graph.add_edge(2, 3)
    graph.add_edge(3, 3)

    print("DFS Traversal starting from vertex 2:")
    graph.dfs(2)

```

practical3 - itterative dfs.py

```python
from collections import defaultdict

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

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

    def dfs(self, start):
        visited = set()
        stack = [start]
        visited.add(start)

        while stack:
            vertex = stack.pop()
            print(vertex, end=' ')

            for neighbor in reversed(self.graph[vertex]):
                if neighbor not in visited:
                    stack.append(neighbor)
                    visited.add(neighbor)

# Example usage
if __name__ == "__main__":
    graph = Graph()
```

```python
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(1, 2)
    graph.add_edge(2, 0)
    graph.add_edge(2, 3)
    graph.add_edge(3, 3)

    print("DFS Traversal starting from vertex 2 (Iterative):")
    graph.dfs(2)

```

practical4-dls.py

```python
from collections import defaultdict

graph = {
     'A': ['C', 'D'],
    'C': ['F', 'G'],
    'D': ['I', 'E'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O']

}

def DLS(start, goal, max_depth, depth=0, path=None):
    if path is None:
        path = []

    path.append(start)

    if start == goal:
        return path

    if depth == max_depth:
        return False

    for neighbor in graph[start]:
        result = DLS(neighbor, goal, max_depth, depth + 1, path)
        if result:
            return result

    path.pop()
    return False

start_node = 'A'
goal_node = input('Enter the goal node: ')
max_depth = int(input("Enter the maximum depth limit: "))
print()

path_to_goal = DLS(start_node, goal_node, max_depth)

if path_to_goal:
    print("Path to goal node available:", path_to_goal)
else:
    print("No path available for the goal node within the given depth limit.")
```

practical5-A-star.py

```python
import heapq

```

```python
def a_star(start, goal, graph, heuristic):

    open_list = []
    heapq.heappush(open_list, (0 + heuristic(start), 0, start, [start]))
    closed_set = set()

    while open_list:

        _, g, current, path = heapq.heappop(open_list)

        if current == goal:
            return path

        closed_set.add(current)


        for neighbor, cost in graph[current].items():
            if neighbor in closed_set:
                continue


            g_temp = g + cost
            f_temp = g_temp + heuristic(neighbor)


            if neighbor not in [i[2] for i in open_list]:
                heapq.heappush(open_list, (f_temp, g_temp, neighbor, path + [neighbor]))
            else:

                for open_node in open_list:
                    if open_node[2] == neighbor and open_node[1] > g_temp:
                        open_list.remove(open_node)
                        heapq.heappush(open_list, (f_temp, g_temp, neighbor, path + [neighbor]))
                        break

    return None


graph = {
    'AA': {'BB': 1, 'CC': 3},
    'BB': {'AA': 1, 'DD': 1, 'EE': 5},
    'CC': {'AA': 3, 'FF': 5},
    'DD': {'BB': 1, 'FF': 1},
    'EE': {'BB': 5, 'FF': 2},
    'FF': {'CC': 5, 'DD': 1, 'EE': 2}
}


def heuristic(node):
    heuristics = {
        'AA': 10,
        'BB': 8,
        'CC': 5,
        'DD': 7,
        'EE': 3,
        'FF': 0
    }

    return heuristics[node]
```

```python

start_node = 'AA'
goal_node = 'FF'
path = a_star(start_node, goal_node, graph, heuristic)
print(f"Path from {start_node} to {goal_node}: {path}")

```

practical6-minmax.py

```python
terminal_nodes = {
    'H': -2,
    'I': 3,
    'J': 1,
    'K': 5,
    'L': -4,
    'M': -6,
    'N': 2,
    'O': 8
}


tree = {
    'A': ['C', 'D'],
    'C': ['F', 'G'],
    'D': ['I', 'E'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O']
}

def minimax(node, depth, is_maximizing_player):
    if node in terminal_nodes:
        return terminal_nodes[node]

    if is_maximizing_player:
        best_value = float('-inf')
        for child in tree[node]:
            value = minimax(child, depth + 1, False)
            best_value = max(best_value, value)
        return best_value
    else:
        best_value = float('inf')
        for child in tree[node]:
            value = minimax(child, depth + 1, True)
            best_value = min(best_value, value)
        return best_value

def find_optimal_path(node, is_maximizing_player):
    optimal_value = minimax(node, 0, is_maximizing_player)
    path = [node]

    while node in tree:
        if is_maximizing_player:
            best_value = float('-inf')
            best_node = None
            for child in tree[node]:
                value = minimax(child, 0, not is_maximizing_player)
                if value > best_value:
                    best_value = value
                    best_node = child
```

```python
        else:
            best_value = float('inf')
            best_node = None
            for child in tree[node]:
                value = minimax(child, 0, not is_maximizing_player)
                if value < best_value:
                    best_value = value
                    best_node = child
        path.append(best_node)
        node = best_node
        is_maximizing_player = not is_maximizing_player

    return optimal_value, path

optimal_value, optimal_path = find_optimal_path('A', True)

print(f"The optimal value is {optimal_value} and the optimal path is {' -> '
.join(optimal_path)}")

```

practical8-alpha-beta pruning.py

```python
import sys

MAX = sys.maxsize
MIN = -sys.maxsize - 1

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best

if __name__ == "__main__":

    values = list(map(int, input("Enter values separated by space: ").split()))
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))




```

practical9-warerjug.py

```python
from collections import deque

def pour(state, jug1, jug2):

    amt = min(state[jug1], (jug_caps[jug2] - state[jug2]))
    new_state = list(state)
    new_state[jug1] -= amt
    new_state[jug2] += amt
    return tuple(new_state)

def get_successors(state):

    successors = []
    for jug1, jug2 in [(0, 1), (1, 0)]:
        new_state = pour(state, jug1, jug2)
        if new_state != state:
            successors.append(new_state)

    for jug in [0, 1]:
        new_state = list(state)
        new_state[jug] = jug_caps[jug]
        successors.append(tuple(new_state))

    for jug in [0, 1]:
        new_state = list(state)
        new_state[jug] = 0
        successors.append(tuple(new_state))

    return successors

def heuristic(state, goal):

    return sum(abs(state[i] - goal[i]) for i in range(len(state)))

def a_star(start, goal):

    open_list = [(heuristic(start, goal), start)]
    print(open_list)
    closed_list = set()
    parent = {start: None}
    while open_list:
        _, curr_state = open_list.pop(0)
        if curr_state == goal:
            path = deque()
            state = curr_state
            while state is not None:
                path.appendleft(state)
                state = parent[state]
            return list(path)
        closed_list.add(curr_state)
        for succ_state in get_successors(curr_state):
            if succ_state not in closed_list:
                succ_cost = heuristic(succ_state, goal)
                open_list.append((succ_cost, succ_state))
                open_list.sort()
                parent[succ_state] = curr_state
    return None

jug_caps = (4, 3)
start_state = (0, 0)
```

```python
goal_state = (2, 0)

solution = a_star(start_state, goal_state)
if solution:
    print("Solution:")
    for state in solution:
        print(state)
else:
    print("No solution exists.")
```