**Program-1**

**Aim:** Write a python program for Breadth First Search.

**Algorithm:**

1. Choose a starting node and enqueue it to a queue.
2. Mark the starting node as visited.
3. While the queue is not empty, dequeue a node from the front of the queue.
4. For each of the dequeued node's neighbours that are not visited, mark them as visited and enqueue them to the queue.
5. Repeat steps 3-4 until the queue is empty.

**Theory:**

The Breadth First Search (BFS) algorithm is used to traverse a graph or a tree in a breadth-first manner. BFS starts at the root node and explores all the neighbouring nodes at the current depth before moving on to the nodes at the next depth. This is done by using a queue data structure. The algorithm marks each visited node to avoid revisiting it.

**Program:**

g={1:[2,3],2:[4,5],3:[6,7],4:[],5:[],6:[],7:[]}

goal=7

def bfs(g, start, goal):

visited = {}

for i in g:

visited[i] = False

q = []

if start == goal:

return f"{start},Goal node found"

visited[start] = True

q.append(start)

while q:

curr = q.pop(0)

print(curr, end= " ")

if curr == goal:

return f"Goal node found"

for j in g[curr]:

if not visited[j]: # Check if the neighbor j is visited, not curr

visited[j] = True

q.append(j)

return "Goal node not found"

print(bfs(g,1,goal))

**Output:**

****

**Program-2**

**Aim:** Write a python program for Depth First Search.

**Algorithm:**

1. Initialize an empty list visited to keep track of nodes currently being visited.
2. Choose a starting node and mark it as visited.
3. For each unvisited neighbor of the starting node, recursively call the DFS algorithm starting from that neighbor.
4. Repeat step 2,3 for all unvisited neighbours.

**Theory:**

The Depth First Search (DFS) algorithm is used to traverse a graph or a tree in a depth-first manner, exploring as far as possible along each branch before backtracking. This is done by using a stack data structure.

**Program:**

g={1:[2,3],2:[4,5],3:[6,7],4:[],5:[],6:[],7:[]}

goal=7

visited={}

for i in g:

visited[i]=False

def dfs(visited,g,start,goal):

print(start,end=" ")

if start==goal:

return "goal node found"

visited[start]= True

for j in g[start]:

if not visited[j]:

result=dfs(visited,g,j,goal)

if result== "goal node found":

return result

return "goal node not found"

print(dfs(visited,g,1,goal))

**Output:**

****

**Program-3**

**Aim:** Write a python program for Iterative Deepening Depth First Search.

**Algorithm:**

1. Begin by initializing an empty dictionary called visited to keep track of visited nodes.
2. Start with an initial depth\_limit set to 0.
3. Continue the process in a loop, which includes the following steps until the goal is found or the search possibilities are exhausted:

* Execute a depth-first search (DFS) up to the current depth\_limit using the dfs function.
* Print the current depth and the sequence of nodes visited at that depth.
* If the goal node is located during the search, announce its discovery along with the depth at which it was found and exit the loop.
* If the goal is not found and all nodes have been visited, print that all possible paths have been explored and exit.

1. If the goal is not found within the current depth\_limit, increment the depth\_limit and initiate another round of search.
2. If the goal node remains unfound after all possible depths are explored, indicate that the target node is not reachable.

**Theory:**

Iterative Deepening Depth First Search (IDDFS) merges the methodologies of Breadth-First Search (BFS) and Depth-First Search (DFS) by conducting several DFS iterations, each with an incrementally increased depth cap, until it locates the desired node. IDDFS harnesses the low memory usage of DFS while ensuring the thoroughness of BFS, making it an ideal choice for exploring expansive state spaces when memory is constrained.

**Program:**

def iter\_deepening\_dfs(g, start, goal):

max\_depth = 0

while True:

visited = {}

for i in g:

visited[i] = False

traversed\_values = []

result = dfs\_with\_limit(g, start, goal, max\_depth, visited, traversed\_values)

print(f"Iteration {max\_depth}: Traversed values: {traversed\_values}")

print(result)

if result=="goal node found":

return

max\_depth += 1

def dfs\_with\_limit(g, current\_vertex, goal, max\_depth, visited, traversed\_values, depth=0):

if depth > max\_depth:

return "depth limit reached"

traversed\_values.append(current\_vertex)

if current\_vertex == goal:

return "goal node found"

visited[current\_vertex] = True

for neighbor in g[current\_vertex]:

if not visited[neighbor]:

result = dfs\_with\_limit(g, neighbor, goal, max\_depth, visited, traversed\_values, depth + 1)

if result == "goal node found":

return result

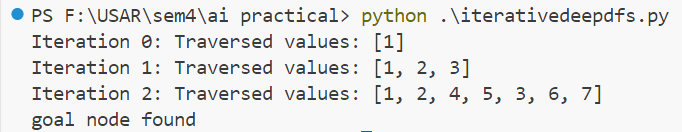
return "goal node not found"

g = {1: [2, 3], 2: [4, 5], 3: [6, 7], 4: [], 5: [], 6: [], 7: []}

goal = 7

iter\_deepening\_dfs(g, 1, goal)

**Output:**

****

**Program-4**

**Aim:** Write a python program for A\* search.

**Algorithm:**

1. First, the initialization step is performed, which involves creating open and closed lists and initializing dictionaries for costs and parents.
2. Next, exploration takes place, where the current node is added to the closed list, neighbours are explored, costs are updated if a better path is found, and neighbours are added to the open list.
3. Subsequently, the goal check is conducted by verifying if the open list is empty.
4. Finally, the return step occurs, where the found path is returned or an indication of no path found is provided.

**Theory:**

A\* search carefully weighs each node's actual distance from the starting point against its estimated distance to the goal. By striking this balance, A\* intelligently navigates through complex graphs. This blend of strategic decision-making and thorough exploration makes A\* a powerful tool for solving large puzzles, particularly in memory-constrained situations.

**Program:**

import heapq

graph = {

'S': [('A', 1), ('B', 4)],

'A': [('B', 2), ('C', 5), ('D', 12)],

'B': [('C', 2)],

'C': [('D', 3)],

'D': [],

}

heuristics = {

'S': 8,

'A': 7,

'B': 3,

'C': 2,

'D': 1

}

start\_node = 'S'

goal\_node = 'D'

def astar\_search(graph, start, goal, heuristics):

open\_list = [(0, start)]

visited = {node: False for node in graph}

g\_costs = {node: float('inf') for node in graph}

g\_costs[start] = 0

parents = {}

while open\_list:

\_, current = heapq.heappop(open\_list)

if current == goal:

path = [current]

while current != start:

current = parents[current]

path.append(current)

path.reverse()

return path

if not visited[current]:

visited[current] = True

for neighbor, cost in graph[current]:

tentative\_g\_cost = g\_costs[current] + cost

if tentative\_g\_cost < g\_costs[neighbor]:

g\_costs[neighbor] = tentative\_g\_cost

f\_cost = tentative\_g\_cost + heuristics[neighbor]

heapq.heappush(open\_list, (f\_cost, neighbor))

parents[neighbor] = current

return None

print(astar\_search(graph,start\_node,goal\_node,heuristics))

**Output:**

****

**Program-5**

**Aim:** Write a python program for Mini-Max algorithm.

**Algorithm:**

1. Initialize a flag is\_maximizing\_player which will have value as true or false
2. If the flag is true, choose the maximum value from the child nodes of the given node and vice versa
3. Recursively repeat until optimal value of root node is obtained
4. Optimal path is obtained from the optimal value by calling the minimax function using a top down approach

**Theory:**

The Mini-max algorithm is like a smart tool used in games where two players compete. It helps players make the best choices by thinking about what their opponent might do. This algorithm works by looking at all the possible moves in a game and figuring out which one gives the best result. It's commonly used in games like Chess, Checkers, and tic-tac-toe to help players plan their moves wisely.

**Program:**

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

# Define a function to find the optimal path

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

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

terminal\_nodes = {

'H': -3,

'I': 5,

'J': 1,

'K': 4,

'L': -1,

'M': -7,

'N': 0,

'O': 6

}

tree = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F', 'G'],

'D': ['H', 'I'],

'E': ['J', 'K'],

'F': ['L', 'M'],

'G': ['N', 'O']

}

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)}")

**Output:**

****

**Program-6(A)**

**Aim:** Write a program to perform given prolog.

**Algorithm:**

1. Initialize Knowledge Base:

* Define the facts about entities and their relationships.
* Add specific facts about who is involved in which activities or roles.

2. Define Inference Rules:

* Create rules that describe how new relationships can be inferred from existing facts.
* Use logical statements to define these inference rules.

3. Querying the Knowledge Base:

* Formulate queries to extract specific information based on the defined rules and facts.
* The system should be able to match queries against the knowledge base and apply inference rules.

4. Execution:

* The system executes the queries by matching them against the facts and applying the inference rules.
* It returns all instances where the rules are satisfied.

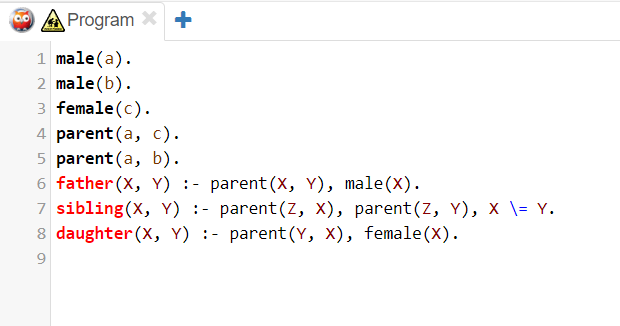
**Theory:**

Prolog, a declarative programming language, operates on a foundation of logical inference and pattern matching. Facts and Rules: In Prolog, you define facts and rules. Facts represent ground truths, while rules establish relationships or conditions.

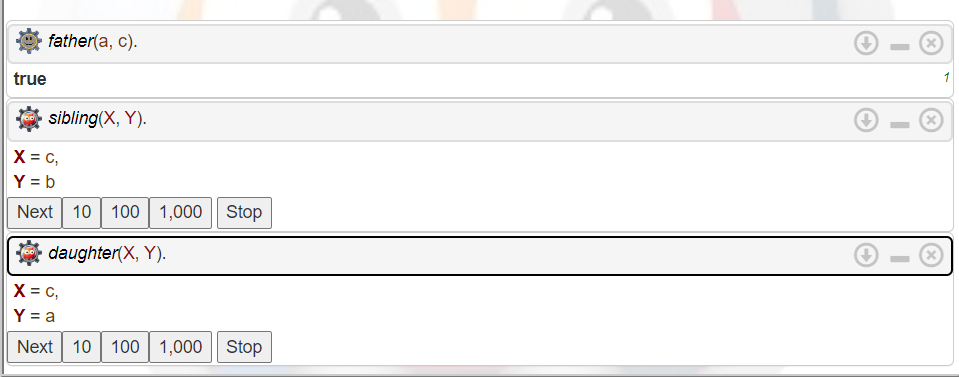
Queries: Queries drive Prolog's execution. You pose queries to the system, which then attempts to find solutions based on the defined facts and rules.

Variables: Prolog uses variables to enable flexible pattern matching and querying. Variables start with an uppercase letter or underscore.

**Program:**

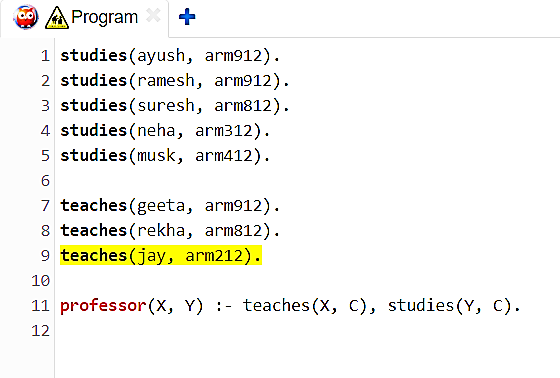


**Output:**

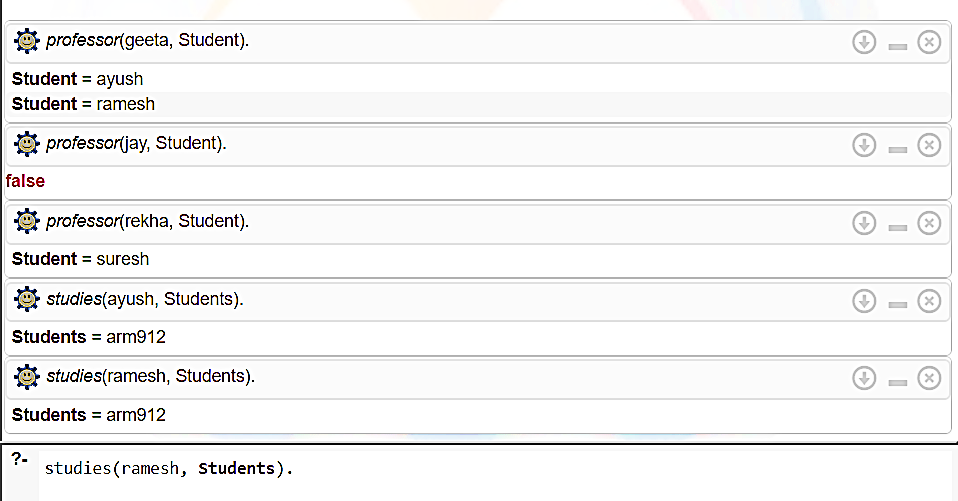


**Program-6(B)**

**Program:**



**Output:**

****

**Program-7**

**Aim:** Write a python program to implement Depth Limited Search.

**Algorithm:**

1. Start with a stack to keep track of nodes to visit
2. Push the starting node onto the stack and mark it as visited.
3. While the stack is not empty and the depth limit has not been reached:
   1. Pop a node from the stack.
   2. Process the node.
   3. Push all unvisited neighbours of the node onto the stack at a depth less than the depth limit and mark them as visited.

4.Repeat until the stack is empty or the depth limit is reached.

**Theory:**

Depth-limited search (DLS) is a variant of Depth-First Search (DFS) where the depth of exploration is limited by a specified depth limit. It starts at a chosen node and explores as far as possible along each branch within the depth limit before backtracking. DLS is useful when DFS needs to be constrained to a certain depth to prevent infinite loops or excessive memory consumption.

**Program:**

def dfs\_recursive(graph, node, visited, depth\_limit, goal):

if depth\_limit == 0:

return

visited[node] = True

print(node, end=' ')

if node == goal:

return "Goal found"

for neighbor in graph[node]:

if not visited[neighbor]:

result = dfs\_recursive(graph, neighbor, visited, depth\_limit - 1, goal)

if result == "Goal found":

return "Goal found"

def depth\_limited\_search(graph, start\_node, depth\_limit, goal):

visited = {node: False for node in graph}

return dfs\_recursive(graph, start\_node, visited, depth\_limit, goal)

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': [],

'F': []

}

start\_node = 'A'

depth\_limit = 2

goal = 'F'

result = depth\_limited\_search(graph, start\_node, depth\_limit, goal)

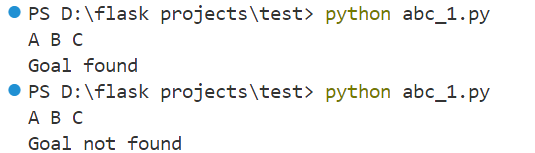
if result == "Goal found":

print("\nGoal found")

else:

print("\nGoal not found")

**Output:**



**Program-8**

**Aim:** Write a python program to implement Alpha Beta pruning.

**Algorithm:**

1. Initialize Alpha and Beta: Set alpha to negative infinity and beta to positive infinity.
2. Recursive Search: Begin recursive search through the game tree, considering each possible move from the current position.
3. Alpha-Beta Updates: At each level of recursion, update alpha and beta values based on the values encountered in the search.
4. Pruning: If at any point the beta value becomes less than or equal to the alpha value, prune the subtree from further consideration.
5. Termination: Terminate the search when all moves have been explored or when a terminal game state is reached.
6. Backtracking: As the recursion unwinds, propagate the best move (maximizing or minimizing) up the tree.

**Theory:**

Alpha-beta pruning is a search algorithm used in game trees to optimize the minimax algorithm by efficiently discarding subtrees that are deemed irrelevant to the final decision. It maintains two values, alpha and beta, representing the best possible outcome for the maximizing and minimizing player, respectively, encountered along the search path. Through recursive depth-first search, it updates these values and prunes subtrees where it is known that the opponent can force a better outcome for themselves. This allows it to explore fewer nodes compared to the standard minimax algorithm, leading to a significant reduction in search time while still guaranteeing the same optimal move. By pruning away branches that cannot affect the final decision, alpha-beta pruning enables more efficient evaluation of game positions in complex games like chess or checkers.

**Program:**

MAX = float('inf')

MIN = -float('inf')

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

values = [3, 5, 6, 9, 1, 2, 0,-1]

print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))

**Output:**



**Program-9**

**Aim:** Write a Python program for the given water jug problem using A\* search algorithm:- You are given 2 Jugs, 4 Gallons, and a 3 Gallon one. Neither of the jugs has any measuring marks on them. A pump can be used to fill the jugs with water. How can we get exactly 2 Gallons of water in 4-gallon jug?

**Algorithm:**

1.Initialization:

* Define resource capacities and start/goal states.
* Create open list for states.
* Add start state with priority and actual cost set to 0.
* Create closed list for visited states.

1. A Search\*:

* While open list not empty:
* Select state with lowest total cost.
* If selected state is goal, return path.
* Add selected state to closed list.
* Generate successor states.
* For each successor:
* Calculate actual cost.
* If successor not in closed list:
* Calculate total cost.
* Add successor to open list.
* If no solution and open list empty, return no solution.

1. Output Solution:

* If solution found, return path.

**Theory:**

The water jug problem, a classic puzzle in computer science and mathematics, involves two jugs of different capacities and a desired quantity of water to be measured precisely using these jugs. The challenge is to determine a sequence of actions—such as filling, emptying, or pouring water between the jugs—that allows achieving the desired quantity of water in one of the jugs while following certain constraints. Common constraints include limiting the number of operations, ensuring no water is wasted, and only using the available capacities of the jugs. This problem showcases concepts like state-space exploration, search algorithms, and constraint satisfaction, making it a fundamental example in algorithmic problem-solving and puzzle-solving strategies.

**Program:**

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), 0, start)] # (f, g, state)

closed\_list = set()

parent = {start: None}

while open\_list:

\_, g, 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):

succ\_g = g + 1 # Assuming the cost of each step is 1

if succ\_state not in closed\_list:

succ\_cost = succ\_g + heuristic(succ\_state, goal)

open\_list.append((succ\_cost, succ\_g, succ\_state))

open\_list.sort()

parent[succ\_state] = curr\_state

return None

jug\_caps = (4, 3)

start\_state = (0, 0)

goal\_state = (2, 0)

solution = a\_star(start\_state, goal\_state)

if solution:

print("Solution:")

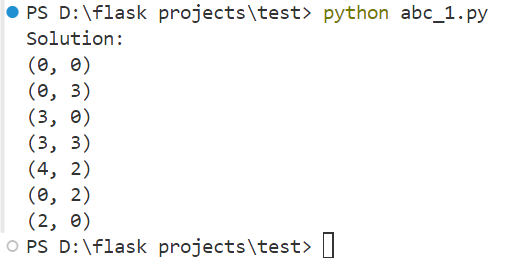
for state in solution:

print(state)

else:

print("No solution exists.")

**Output:**



**Program-10**

**Aim:** WAP in python to draw membership function curve in fuzzy r elations of the following function:

a) Triangular function

        b) Gaussian function

        c) Trapezoid function

**Algorithm:**

a) Triangular Function:

1. Input the parameters for the triangular function: the minimum value, the peak value, and the maximum value.
2. Generate a range of x-values from the minimum to the maximum.
3. Calculate the membership degree for each x-value using the triangular function formula.
4. Plot the x-values against their corresponding membership degrees on a graph.

b) Gaussian Function:

1. Input the parameters for the Gaussian function: the mean and the standard deviation.
2. Generate a range of x-values spanning several standard deviations around the mean.
3. Calculate the membership degree for each x-value using the Gaussian function formula.
4. Plot the x-values against their corresponding membership degrees on a graph.

c) Trapezoid Function:

1. Input the parameters for the trapezoid function: the left support, the left peak, the right peak, and the right support.
2. Generate a range of x-values covering the entire span of the trapezoid.
3. Calculate the membership degree for each x-value using the trapezoid function formula.
4. Plot the x-values against their corresponding membership degrees on a graph.

**Theory:**

Triangular Function

The triangular membership function is one of the simplest forms of representing fuzzy sets. It is defined by three parameters 𝑎*a*, 𝑏*b*, and 𝑐*c* which correspond to the lower limit, the peak, and the upper limit respectively. The function increases linearly from 𝑎*a* to 𝑏*b* and then decreases linearly from 𝑏*b* to 𝑐*c*. This function is useful in scenarios where a gradual transition between degrees of membership is desired, and it is particularly easy to implement and interpret.

Gaussian Function

The Gaussian membership function is defined using the Gaussian distribution, characterized by a mean 𝑚*m* and a standard deviation 𝜎*σ*. This function forms a bell-shaped curve, which is smooth and continuous, making it suitable for modelling situations where the degree of membership changes gradually and symmetrically around the mean. The Gaussian function is often used in control systems and pattern recognition due to its smoothness and differentiability.

Trapezoid Function

The trapezoid membership function is defined by four parameters 𝑎*a*, 𝑏*b*, 𝑐*c*, and 𝑑*d* which form a trapezoid shape. The function remains constant between 𝑏*b* and 𝑐*c*, increases linearly from 𝑎*a* to 𝑏*b*, and decreases linearly from 𝑐*c* to 𝑑*d*. This function is more flexible than the triangular function and can model situations with a plateau or where the membership degree is constant over a range of values.

**Program:**

**Triangular**

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')

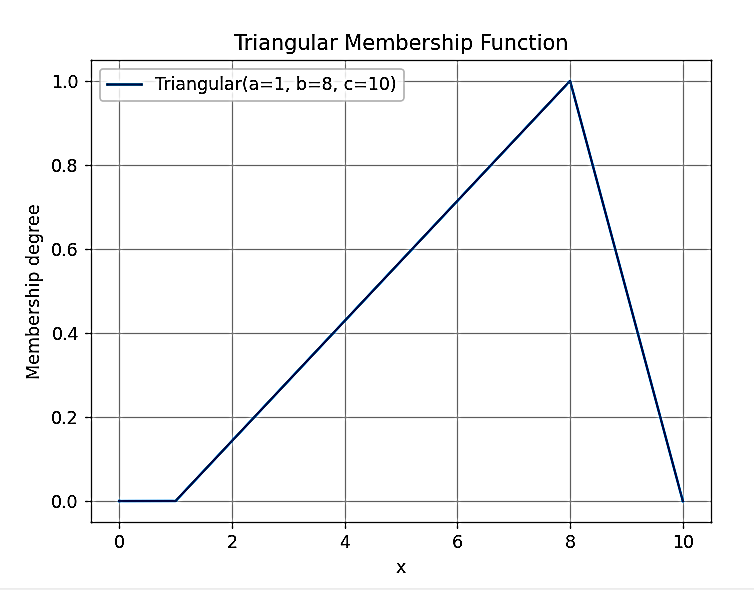
plt.ylabel('Membership degree')

plt.title('Triangular Membership Function')

plt.legend()

plt.grid(True)

plt.show()

****

**Gaussian**

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')

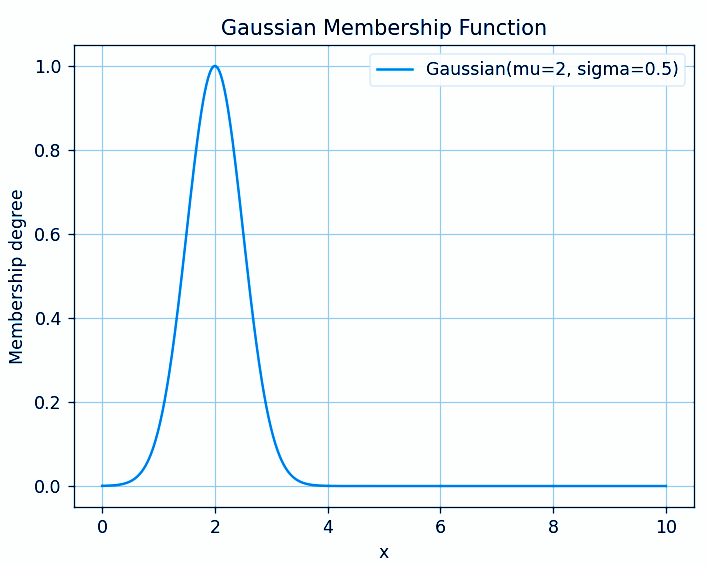
plt.ylabel('Membership degree')

plt.title('Gaussian Membership Function')

plt.legend()

plt.grid(True)

plt.show()

****

**Trapezoid**

import numpy as np

import matplotlib.pyplot as plt

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')

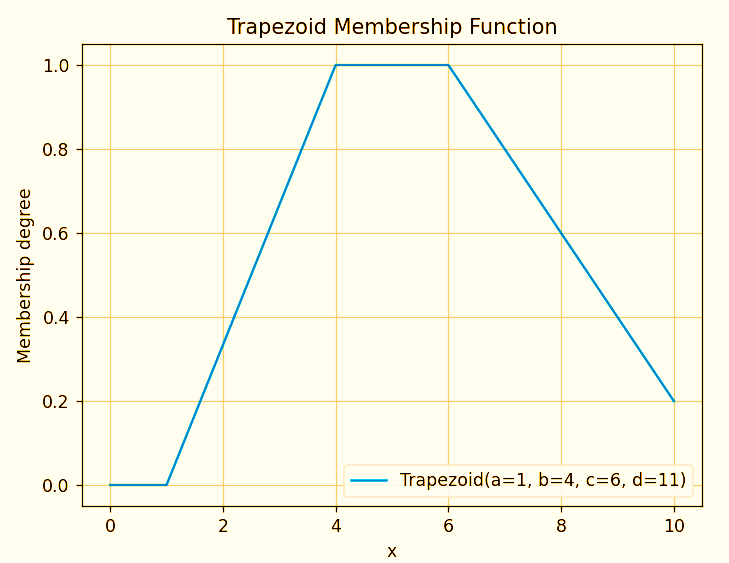
plt.ylabel('Membership degree')

plt.title('Trapezoid Membership Function')

plt.legend()

plt.grid(True)

plt.show()

****