**RAJALAKSHMI ENGINEERING COLLEGE**

**RAJALAKSHMI NAGAR, THANDALAM – 602 105**

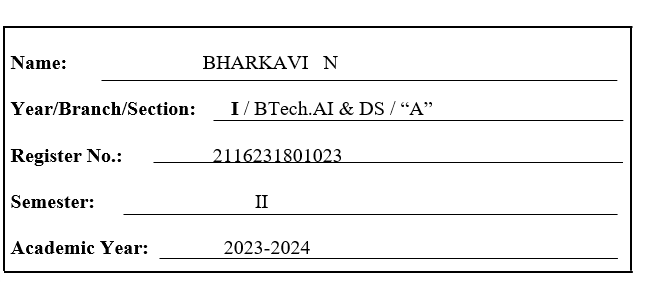


**AI23231 PRINCIPLES OF ARTIFICIAL INTELLIGENCE**

**(Regulation 2023)**

**LAB RECORD**

**lL**

****

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No.** | **Date** | **Name Of The Experiments** | **Page No.** | **Faculty Sign.** |
| 1. | 20/02/2024 | 8-QUEENS PROBLEM | 3 |  |
| 2. | 27/02/2024 | DEPTH FIRST SEARCH(DFS) | 6 |  |
| 3. | 05/03/2024 | WATER JUG PROBLEM  USING BFS | 8 |  |
| 4. | 12/03/2024 | WATER JUG PROBLEM  USING DFS | 12 |  |
| 5. | 19/03/2024 | BREADTH  FIRST  SEARCH | 15 |  |
| 6. | 26/03/2024 | A\* SEARCH ALGORITHM | 18 |  |
| 7. | 02/04/2024 | AO\* SEARCH  ALGORITHM | 21 |  |
| 8. | 09/04/2024 | RECURSIVE  BEST-FIRST SEARCH   ALGORITHM | 24 |  |
| 9. | 16/04/2024 | MAP  COLOURING | 26 |  |
| 10. | 23/04/2024 | MIN-MAX ALGORITHM | 28 |  |
| 11. | 30/04/2024 | ALPHA-BETA PRUNING | 30 |  |
| 12.a. | 04/05/2024 | PROLOG | 32 |  |
| 12.b. | 11/05/2024 | PROLOG  INTRODUCTION | 36 |  |
| 13. | 25//05/2024 | UNIFICATION AND RESOLUTION | 42 |  |
| 14. | 01/06/2024 | FUZZY LOGIC | 48 |  |

|  |  |
| --- | --- |
| **EX.N0:1** | **8- QUEENS PROBLEM** |
| **DATE: 20/02/2024** |

**AIM:**

To solve 8Queen Problem using python

**ALGORITHM:**

Step1: Initiate an empty chess board of size8x8.

Step 2: Start with the left most column and place a queen in the first row of the column.

Step3: Move to the next column and place a queen in the first row of the column.

Step4: Repeat Step3 until either all 8queen has been placed or it is impossible to place a

Queen in the current column without violating the rule of the problem.

Step5: If it not possible to place a queen in the current column

**PROGRAM:**

N = 8  # Assuming N is defined elsewhere

def solveNQueens(board, col):

    # Base case: If all columns are filled, print the board and return True

    if col == N:

        printBoard(board)

        return True

    # Recursive case: Try placing queens in each row of the current column

    for i in range(N):

        # Check if it's safe to place a queen in this row and column

        if isSafe(board, i, col):

            # Place a queen in this position

            board[i][col] = 'Q'

            # Recursively solve the problem for the next column

            if solveNQueens(board, col + 1):

                return True

            # Backtrack: If placing a queen doesn't lead to a solution, reset the position

            board[i][col] = 0

    # If no solution is found for this column, return False

    return False

def isSafe(board, row, col):

    # Check if there's a Queen in the same row

    for x in range(col):

        if board[row][x] == 'Q':

            return False

    # Check upper diagonal on left side

    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[x][y] == 'Q':

            return False

    # Check lower diagonal on left side

    for x, y in zip(range(row, N, 1), range(col, -1, -1)):

        if board[x][y] == 'Q':

            return False

    # If no conflicting queens are found, return True

    return True

def printBoard(board):

    # Print the board configuration

    for i in range(N):

        for j in range(N):

            print(board[i][j], end=" ")

        print()

# Initialize the board with all cells set to 0

board = [[0 for x in range(N)] for y in range(N)]

# Start solving the N-Queens problem from column 0

if not solveNQueens(board, 0):

    print("No solution found")  # If no solution is found, print a message

**Output**

Q 0 0 0 0 0 0 0

0 0 0 0 0 0 Q 0

0 0 0 0 Q 0 0 0

0 0 0 0 0 0 0 Q

0 Q 0 0 0 0 0 0

0 0 0 Q 0 0 0 0

0 0 0 0 0 Q 0 0

**RESULT:**

Thus, the experiment to **solve the 8-Queen Problem** by using Python has been executed and verified successfully.

|  |  |
| --- | --- |
| **EX.N0:2** | **DEPTH FIRST SEARCH (DFS)** |
| **DATE: 27/02/2024** |

**AIM :**

To implement a depth first search problem using python.

**ALGORITHM:**

Step 1: Start.

Step 2: Start with root node G

Step 3: Search deep.if goal node has no children visit different node

Step 4: If state is reach goal node.

**PROGRAM**

def dfs(node, graph, visited, component):

    component.append(node)  # Store answer

    visited[node] = True  # Mark visited

    # Traverse to each adjacent node of a node

    for child in graph[node]:

        if not visited[child]:  # Check whether the node is visited or not

            dfs(child, graph, visited, component)  # Call the dfs recursively

if \_\_name\_\_ == "\_\_main\_\_":

    # Graph of nodes

    graph = {

        0: [2],

        1: [2, 3],

        2: [0, 1, 4],

        3: [1, 4],

        4: [2, 3]

    }

    node = 0  # Starting node

    visited = [False]\*len(graph)  # Make all nodes to False initially

    component = []

    dfs(node, graph, visited, component)  # Traverse to each node of a graph

    print( f" Following is the Depth-first search: {component}")

**Output:**

Following is the Depth-first search: [0, 2, 1, 3, 4]

**RESULT:**

Thus the experiment to do Depth first search by using python has been executed and verified Successfully.

**WATER JUG PROGRAM USING DFS**

|  |  |
| --- | --- |
| **EX.N0:3** | **WRITE PROGRAM TO WATER JUG PROGRAM USING DFS** |
| **DATE: 05/03/24** |

**AIM:**

To solve the Water jug Problem using Depth First Search.

**ALGORITHM:**

Step1:Start with two empty Jugs: Jug1 & Jug 2

Step 2: Define the capacities of jug 1, jug 2 and desired volume.

Step3: Create a set“visited\_states”to avoid repeating the same state

Step4:  Start the function of DFS jug:

1. If the current state has been visited before,stop & return“FALSE”.
2. Mark the Current state as visited.
3. Try the action: fill jug 1, fill jug 2, empty jug 1, empty  jug 2, pour water from jug 1 to jug2.
4. If no action  lead to the goal then return “False”.

Step 5:Create a function solve jug problem that calls “DFS JUG” with the initial state & print an approximate message if the goal can’t be reached

**PROGRAM:**

def water\_jug\_dfs(capacity\_x, capacity\_y, target):

    def dfs(x, y, path):

        if x == target or y == target:

            path.append((x, y))

            return True

        if visited[x][y]:

            return False

        visited[x][y] = True

        # Try all possible operations on jugs

        # Fill jug x

        if x < capacity\_x:

            if dfs(capacity\_x, y, path):

                path.append((x, y))

                return True

        # Fill jug y

        if y < capacity\_y:

            if dfs(x, capacity\_y, path):

                path.append((x, y))

                return True

        # Empty jug x

        if x > 0:

            if dfs(0, y, path):

                path.append((x, y))

                return True

        # Empty jug y

        if y > 0:

            if dfs(x, 0, path):

                path.append((x, y))

                return True

        # Pour from x to y

        if x > 0 and y < capacity\_y:

            pour = min(x, capacity\_y - y)

            if dfs(x - pour, y + pour, path):

                path.append((x, y))

                return True

        # Pour from y to x

        if y > 0 and x < capacity\_x:

            pour = min(y, capacity\_x - x)

            if dfs(x + pour, y - pour, path):

                path.append((x, y))

                return True

        return False

    visited = [[False for \_ in range(capacity\_y + 1)] for \_ in range(capacity\_x + 1)]

    path = []

    if dfs(0, 0, path):

        path.reverse()

        return path

    else:

        return "No solution found."

# Define capacities of jugs and the target amount of water

capacity\_x = 4

capacity\_y = 3

target = 2

# Find the solution path

solution\_path = water\_jug\_dfs(capacity\_x, capacity\_y, target)

# Print the solution steps or indicate no solution found

if solution\_path != "No solution found.":

    for step, (x, y) in enumerate(solution\_path):

        print(f"Step {step}: Jug X: {x}, Jug Y: {y}")

else:

    print("No solution found.")

**SAMPLE I/O AND O/P:**

Step 0: Jug X: 0, Jug Y: 0

Step 1: Jug X: 4, Jug Y: 0

Step 2: Jug X: 4, Jug Y: 3

Step 3: Jug X: 0, Jug Y: 3

Step 4: Jug X: 3, Jug Y: 0

Step 5: Jug X: 3, Jug Y: 3

Step 6: Jug X: 4, Jug Y: 2

**OUTPUT:**

**RESULT:** Thus the experiment to solve water jug problem using DFS by using python has been executed and verified Successfully.

**WATER JUG PROGRAM USING BFS**

|  |  |
| --- | --- |
| **EX.N0:4** | **WRITE PROG PROGRAM TO WATER JUG PROGRAM USING BFS** |
| **DATE: 12/03/2024** |

**AIM:**

To solve the Water jug Problem using Breadth First Search.

**ALGORITHM:**

Step1:Start.

Step2:Get the capacity of A jug and B jug and Target.

Step3: Create water jug problem function.

Step4:In Function A,B, Target are parameter.

Step 5:In Function initialize state = (0, 0).

Step6:  Create parent empty set.

Step7:Create frontier isn’t there is all the possible states to will be stored

Step8:  Using while loop frontier is the condition.

Step 9:Assign state =frontier pop.

Step10:If state is reach the target

**PROGRAM:**

from collections import deque

def BFS(a, b, target):

    m = {}

    isSolvable = False

    path = []

    q = deque()

    q.append((0, 0))

    while len(q) > 0:

        u = q.popleft()

        print("Exploring state:", u)

        if (u[0], u[1]) in m:

            print("Already visited state:", u)

            continue

        if u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0:

            print("Invalid state:", u)

            continue

        path.append([u[0], u[1]])

        m[(u[0], u[1])] = 1

        if u[0] == target or u[1] == target:

            isSolvable = True

            if u[0] == target:

                if u[1] != 0:

                    path.append([u[0], 0])

            else:

                if u[0] != 0:

                    path.append([0, u[1]])

            sz = len(path)

            print("Path:")

            for i in range(sz):

                print("(", path[i][0], ",", path[i][1], ")")

            break

        q.append([u[0], b])  # Fill Jug2

        q.append([a, u[1]])  # Fill Jug1

        for ap in range(max(a, b) + 1):

            c = u[0] + ap

            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):

                q.append([c, d])

            c = u[0] - ap

            d = u[1] + ap

            if ((c == 0 and c >= 0) or d == b):

                q.append([c, d])

        q.append([a, 0])  # Empty Jug1

        q.append([0, b])  # Empty Jug2

    if not isSolvable:

        print("No solution")

if \_\_name\_\_ == '\_\_main\_\_':

    Jug1, Jug2, target = 4, 3, 2

    print("Path from initial state to solution state:")

    BFS(Jug1, Jug2, target)

**OUTPUT:**

Path from initial state to solution state:

( 0 , 0 )

( 0 , 3 )

( 4 , 0 )

( 4 , 3 )

( 3 , 0 )

( 1 , 3 )

( 3 , 3 )

**RESULT:** Thus the experiment to do water jug problem(BFS) by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0:5** | **BREADTH-FIRST SEARCH (BFS)** |
| **DATE: 19/03/2024** |

**AIM:**

Write a Program to Implement Depth First Search using Python

**ALGORITHM:**

STEP 1: Start BFS from a given source vertex (s):

STEP 2: Create a boolean array visited to keep track of visited vertices.

STEP 3: Create an empty queue queue to store vertices to be visited.

STEP 4: Enqueue the source vertex s into the queue and mark it as visited.

STEP 5: While the queue is not empty. Dequeue a vertex s from the queue.

STEP 6: Print the value of s. For each adjacent vertex i of s.

STEP 7: If i is not visited. Enqueue i into the queue. Mark i as visited.

STEP 8: Repeat the process until the queue is empty.

STEP 9: Print the sequence of vertices visited during the BFS traversal.

**PROGRAM:**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        # default dictionary to store the graph

        self.graph = defaultdict(list)

    def addEdge(self, u, v):

        # function to add an edge to the graph

        self.graph[u].append(v)

    def BFS(self, s):

        # Mark all the vertices as not visited

        visited = [False] \* (max(self.graph) + 1)

        # Create a queue for BFS

        queue = []

        # Mark the source node as visited and enqueue it

        queue.append(s)

        visited[s] = True

        while queue:

            # Dequeue a vertex from the queue and print it

            s = queue.pop(0)

            print(s, end=" ")

            # Get all adjacent vertices of the dequeued vertex s.

            # If an adjacent vertex has not been visited,

            # mark it visited and enqueue it

            for i in self.graph[s]:

                if not visited[i]:

                    queue.append(i)

                    visited[i] = True

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    # Create a graph

    g = Graph()

    # Add edges to the graph

    g.addEdge(0, 1)

    g.addEdge(0, 2)

    g.addEdge(1, 2)

    g.addEdge(2, 0)

    g.addEdge(2, 3)

    g.addEdge(3, 3)

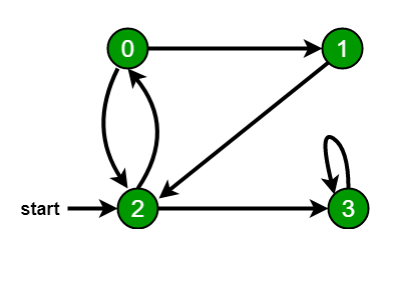
    # Print BFS traversal starting from vertex 2

    print("Following is Breadth First Traversal (starting from vertex 2):")

    g.BFS(2)

**OUTPUT:**

 Following is Breadth First Traversal (starting from vertex 2) 2 0 3 1



**RESULT:** Thus the experiment to do Breadth First Search by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0:6** | **A\*SEARCH ALGORITHM** |
| **DATE:26/03/24** |

**AIM:**

To implement the A\* Search Algorithm using python

**ALGORITHM:**

Step1:Start

Step2:initiate open list and close list

Step3:While the open list is not empty, find the node with least form the openlist, call it “q”

Step 4:Stop working when you find the destination or you can’t find the destination going through all possible points from collections import deque

**PROGRAM:**

class Graph:

    def \_\_init\_\_(self, adjacency\_list):

        self.adjacency\_list = adjacency\_list

    def get\_neighbors(self, v):

        return self.adjacency\_list[v]

    def h(self, n):

        H = {'A': 1, 'B': 1, 'C': 1, 'D': 1}

        return H[n]

    def a\_star\_algorithm(self, start\_node, stop\_node):

        open\_list = set([start\_node])

        closed\_list = set([])

        g = {}

        g[start\_node] = 0

        parents = {}

        parents[start\_node] = start\_node

        while len(open\_list) > 0:

            n = None

            for v in open\_list:

                if n is None or (g[v] + self.h(v) < g[n] + self.h(n)):

                    n = v

            if n is None:

                print('Path does not exist!')

                return None

            if n == stop\_node:

                reconst\_path = [n]

                while parents[n] != n:

                    n = parents[n]

                    reconst\_path.append(n)

                reconst\_path.reverse()

                print('Path found: {}'.format(reconst\_path))

                return reconst\_path

            open\_list.remove(n)

            closed\_list.add(n)

            for m, weight in self.get\_neighbors(n):

                if m in closed\_list:

                    continue

                if m not in open\_list:

                    open\_list.add(m)

                    parents[m] = n

                    g[m] = g[n] + weight

                else:

                    if g[m] > g[n] + weight:

                        g[m] = g[n] + weight

                        parents[m] = n

        print('Path does not exist!')

        return None

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    adjacency\_list = {

        'A': [('B', 1), ('C', 3)],

        'B': [('A', 1), ('D', 5)],

        'C': [('A', 3), ('D', 2)],

        'D': [('B', 5), ('C', 2)]

    }

    graph = Graph(adjacency\_list)

    graph.a\_star\_algorithm('A', 'D')

**OUTPUT:**

Path found: ['A', 'C', 'D']

**RESULT:** Thus the experiment to solve A\* search algorithm by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0:7** | **AO\* SEARCH ALGORITHM** |
| **DATE:09/04/2024** |

**AIM:**

To implement the AO\* Search Algorithm using  python

**ALGORITHM:**

Step1:Start

Step2:Initialize an open list with the start node and an empty closedlist

Step3: While the open list is not empty:

1. Select the node with the lowest estimated total cost (f-value).
2. If the selected node is the goal, return the solution.
3. Generate successor nodes ,calculate their costs, and add them to the openlist if they are better or not in the closed list.
4. Add the selected node to the closed list.

Step 4: If the open list becomes empty, and no solution is found, the problem has no solution.

Step5:Stop

**PROGRAM:**

import heapq

class Node:

    def \_\_init\_\_(self, state, g\_value, h\_value, parent=None):

        self.state = state

        self.g\_value = g\_value

        self.h\_value = h\_value

        self.parent = parent

    def f\_value(self):

        return self.g\_value + self.h\_value

def a\_star\_search(initial\_state, is\_goal, successors, heuristic):

    open\_list = [Node(initial\_state, 0, heuristic(initial\_state), None)]

    closed\_set = set()

    while open\_list:

        open\_list.sort(key=lambda node: node.f\_value())

        current\_node = open\_list.pop(0)

        if is\_goal(current\_node.state):

            path = []

            while current\_node:

                path.append(current\_node.state)

                current\_node = current\_node.parent

            return list(reversed(path))

        closed\_set.add(current\_node.state)

        for child\_state in successors(current\_node.state):

            if child\_state in closed\_set:

                continue

            g\_value = current\_node.g\_value + 1

            h\_value = heuristic(child\_state)

            child\_node = Node(child\_state, g\_value, h\_value, current\_node)

            for i, node in enumerate(open\_list):

                if node.state == child\_state:

                    if node.g\_value > g\_value:

                        open\_list.pop(i)

                        break

                elif node.g\_value > g\_value:

                    open\_list.insert(i, child\_node)

                    break

            else:

                open\_list.append(child\_node)

    return None

if \_\_name\_\_ == "\_\_main\_\_":

    def is\_goal(state):

        return state == (4, 4)

    def successors(state):

        x, y = state

        return [(x+1, y), (x, y+1)]

    def heuristic(state):

        x, y = state

        return abs(4 - x) + abs(4 - y)

    initial\_state = (0, 0)

    path = a\_star\_search(initial\_state, is\_goal, successors, heuristic)

    if path:

        print("Path found:", path)

    else:

        print("No path found")

**OUTPUT:**

Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

**RESULT:** Thus the experiment to solve AO\* search algorithm by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0:8** | **RECURSIVE BEST-FIRST SEARCH ALGORITHM** |
| **DATE:09/04/2024** |

**AIM:**

To implement the Recursive Best-First Search using python

**ALGORITHM:**

Step 1: Start with an initial node and add it to a priority queue.

Step 2: While the priority queue is not empty:

Step 3: Pop the node with the lowest estimated cost.

Step 4: If the popped node is a goal state, you've found the solution.

Step 5: Generate child nodes and estimate their costs.

Step 6: Sort child nodes by their estimated costs.

Step 7: Recursively apply RBFS to the child with the lowest estimated cost.

Step 8: If a node returns a failure, update its cost to be higher than the next best child.

Step 9: Continue until a solution is found or all nodes are explored.

Step 10: RBFS terminates when a solution is found or when all nodes have been explored.

**PROGRAM:**

class Node:

    def \_\_init\_\_(self, state, parent=None, cost=0, heuristic=0):

        self.state = state

        self.parent = parent

        self.cost = cost

        self.heuristic = heuristic

        self.f = cost + heuristic

    def is\_goal(self, goal):

        return self.state == goal

    def generate\_successors(self, goal):

        successors = []

        if self.state < goal:

            successors.append(Node(self.state + 1, self, self.cost + 1, heuristic(self.state + 1, goal)))

        return successors

def heuristic(state, goal):

    return abs(goal - state)

def rbfs(node, f\_limit, goal):

    if node.is\_goal(goal):

        return node

    successors = node.generate\_successors(goal)

    if not successors:

        return None

    while True:

        successors.sort(key=lambda x: x.f)

        best = successors[0]

        if best.f > f\_limit:

            return None

        if len(successors) > 1:

            alternative = successors[1].f

        else:

            alternative = float('inf')

        result = rbfs(best, min(f\_limit, alternative), goal)

        if result is not None:

            return result

initial\_state = 0

goal\_state = 5

initial\_node = Node(initial\_state, None, 0, heuristic(initial\_state, goal\_state))

solution = rbfs(initial\_node, float('inf'), goal\_state)

if solution is not None:

    path = []

    while solution is not None:

        path.append(solution.state)

        solution = solution.parent

    path.reverse()

    print("RBFS Path:", path)

else:

    print("No solution found.")

**OUTPUT:**

RBFS Path: [0, 1, 2, 3, 4, 5]

**RESULT:** Thus the experiment to solve Recursive Best First search algorithm by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0:9** | **CSP-MAP COLOURING** |
| **DATE:16/04/2024** |

**AIM:**

To implement the CSP-Map Colouring Algorithm using python Program.

**ALGORITHM:**

Step 1: Define regions, colors, and constraints.

Step 2: Create an empty assignment.

Step 3: Start with the first uncolored region.

Step 4: Recursively try colors for each uncolored region.

Step 5: Check if the chosen color complies with constraints.

Step 6: If consistent, assign the color and continue to the next region.

Step 7: If not, backtrack and try the next color.

Step 8: Repeat until all regions are assigned colors or determine no valid assignment exists.

**PROGRAM:**

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    def isSafe(self, v, colour, c):

        for i in range(self.V):

            if self.graph[v][i] == 1 and colour[i] == c:

                return False

        return True

    def graphColourUtil(self, m, colour, v):

        if v == self.V:

            return True

        for c in range(1, m + 1):

            if self.isSafe(v, colour, c):

                colour[v] = c

                if self.graphColourUtil(m, colour, v + 1):

                    return True

                colour[v] = 0

        return False

    def graphColouring(self, m):

        colour = [None] \* self.V

        if not self.graphColourUtil(m, colour, 0):

            print("No solution exists")

            return False

        print("Solution exists and Following are the assigned colours:")

        for c in colour:

            print(c, end=' ')

        return True

if \_\_name\_\_ == '\_\_main\_\_':

    g = Graph(4)

    g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]

    m = 3

    g.graphColouring(m)

**Output:**

Solution Exists: Following are the assigned colours

1 2 3 2

**RESULT:** Thus the experiment to CSP map coloring algorithm by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0:10** | **MIN-MAX ALGORITHM** |
| **DATE:23/04/2024** |

**AIM:**

To implement the MIN MAX Algorithm using python Program.

**ALGORITHM:**

Step 1: Start with the current game state, player, and depth.

Step 2: If the game is over, return a utility value (positive for a win, negative for a loss, 0 for a draw).

Step 3: For the maximizing player, choose the move that maximizes the utility value by

recursively exploring possible moves.

Step 4: For the minimizing player, choose the move that minimizes the utility value by doing the

same.

Step 5: Make an initial call with the current player and depth 0 to find the best move.

Step 6: Recursively explore all possible moves and counter-moves, considering rational

opponents, to determine the best move for the current player.

**PROGRAM:**

import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):

    if curDepth == targetDepth:

        return scores[nodeIndex]

    if maxTurn:

        return max(minimax(curDepth + 1, nodeIndex \* 2, False, scores, targetDepth),

                   minimax(curDepth + 1, nodeIndex \* 2 + 1, False, scores, targetDepth))

    else:

        return min(minimax(curDepth + 1, nodeIndex \* 2, True, scores, targetDepth),

                   minimax(curDepth + 1, nodeIndex \* 2 + 1, True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log2(len(scores))

print("The optimal value is : ", end="")

print(minimax(0, 0, True, scores, treeDepth))

**SAMPLE I/O AND O/P:**

The optimal value is: 12

**OUTPUT:**

The optimal value is: 12

**RESULT:** Thus the experiment to solve Minmax algorithm by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EX.N0: 11** | **ALPHA-BETAPRUNING** |
| **DATE: 30/04/24** |

**AIM:**

To implement the Alpha-Beta Pruning using python Program.

**ALGORITHM:**

Step 1: Start with the current game state, player, depth, alpha (initially negative infinity), and

beta (initially positive infinity).

Step 2: If the game is over, return the utility value.

Step 3: For the maximizing player, explore moves and update alpha while pruning when beta is

less than or equal to alpha.

Step 4: For the minimizing player, explore moves and update beta while pruning when alpha is

greater than or equal to beta.

Step 5: Make an initial call with the current player, depth 0, -∞ for alpha, and +∞ for beta to find

the best move.

Step 6: Recursively explore and prune branches based on the alpha and beta values to optimize

the search.

Step 7: Return the utility value of the best move for the current player at the initial call.

**Program:**

MAX, MIN = 1000, -1000

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

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

**SAMPLE I/O AND O/P:**

The optimal value is : 5

**OUTPUT:**

The optimal value is : 5

**RESULT:** Thus the experiment to solve Alpha beta pruning algorithm by using python has been executed and verified Successfully.

|  |  |
| --- | --- |
| **EXP.NO : 12(a)** | **PROLOG** |
| **DATE : 04/05/2024** |

**AIM:**

To develop a family tree program using PROLOG with all possible facts,rulesand queries

**Program:**

**KNOWLEDGEBASE:**

/\*FACTS::\*/ male(peter). male(john). male(chris). male(kevin).

female(betty). female(jeny). female(lisa). female(helen).

parentOf(chris,peter). parentOf(chris,betty). parentOf(helen,peter). parentOf(helen,betty). parentOf(kevin,chris). parentOf(kevin,lisa). parentOf(jeny,john). parentOf(jeny,helen).

/\*RULES::\*/

/\* son,parent

\*son,grandparent\*/

father(X,Y):-male(Y), parentOf(X,Y).

mother(X,Y):-female(Y), parentOf(X,Y).

grandfather(X,Y):-male(Y),

parentOf(X,Z), parentOf(Z,Y).

grandmother(X,Y):-female(Y), parentOf(X,Z),

parentOf(Z,Y).

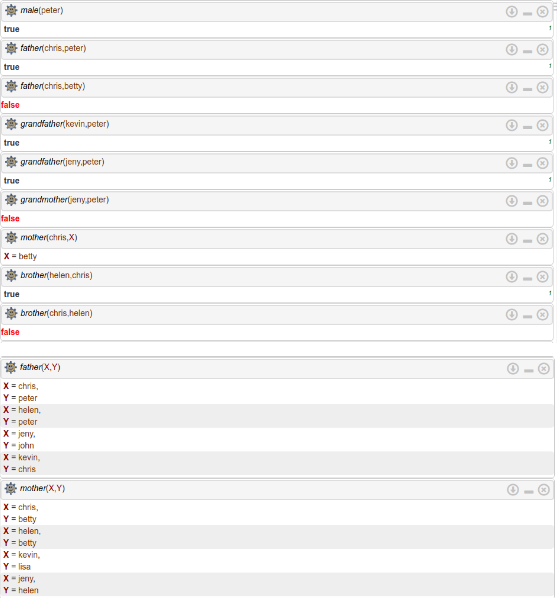
brother(X,Y):-male(Y), father(X,Z),

father(Y,W), Z==W.

sister(X,Y):-female(Y), father(X,Z),

father(Y,W), Z==W.

**OUTPUT:**



**RESULT:**

Thus to implement the family tree program using PROLOG with all possible facts,rulesand queries has been executed successfully and verified 

|  |  |
| --- | --- |
| **EXP.NO : 12(b)** | **PROLOG INTRODUCTION** |
| **DATE :11/05/2024** |

**PROLOG:**

To learn PROLOG  terminologies and write basic programs.

**TERMINOLOGIES**

**1.Atomic Terms:-**

             Atomic terms are usually strings made up of lower- and  upper case letters, digits,  and the underscore, starting with a  lowercase letter.

**Ex:**

dog ab\_c\_321

**2.Variables:-**

          Variables are string , letters, digits, and the underscore, starting with a capital letter or an underscore.

**Ex:**

Dog Apple\_420

**3.CompoundTerms:-**

           Compound terms are made up of a PROLOG atom and a number of arguments (PROLOG terms,

i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.

**Ex**:

is\_bigger(elephant,X) f(g(X,\_),7)

**4.Facts:-**

           A fact is a predicate followed by a dot.

**Ex:**

bigger\_animal(whale).

life\_is\_beautiful.

**5.Rules:-**

      A rule consists of a head (apredicate) and a body(asequence of predicates separated by commas).

**Ex:**

is\_smaller(X,Y):-is\_bigger(Y,X).

aunt(Aunt,Child):-sister(Aunt,Parent),parent(Parent,Child).

**SOURCECODE:**

**KB1:**

woman(mia).

woman(jody).

woman(yolanda).

playsAirGuitar(jody).

party.

Query1:?-woman(mia).

Query2:?-playsAirGuitar(mia).

Query3:?-party.

Query4:?-concert

**OUTPUT:** 

**KB2:**

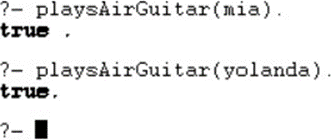
     happy(yolanda).

listens2music(mia).

Listens2music(yolanda):-happy(yolanda).

playsAirGuitar(mia):-listens2music(mia).

playsAirGuitar(Yolanda):-listens2music(yolanda).

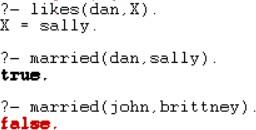
**OUTPUT:**

**KB3:**

likes(dan,sally). likes(sally,dan). likes(john,brittney).

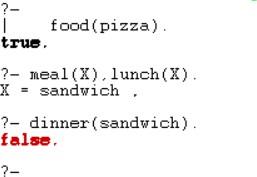
married(X,Y):- likes(X,Y), likes(Y,X).

friends(X,Y):-likes(X,Y) ;likes(Y,X).

**OUT**

**KB4:**

food(burger). food(sandwich). food(pizza). lunch(sandwich). dinner(pizza). meal(X):-food(X).

**OUTPUT:**

**KB5: Find minimum maximum of two numbers**

find\_max(X,Y,X):-X>=Y,!.

find\_max(X,Y,Y):-X<Y.

find\_min(X,Y,X):-X=<Y,!.

find\_min(X,Y,Y):-X>Y.

**Output:**

| ?- find\_max(100,200,Max).

Max = 200

yes

| ?- find\_max(40,10,Max).

Max = 40

yes

| ?- find\_min(40,10,Min).

Min = 10

yes

| ?- find\_min(100,200,Min).

Min = 100

yes

| ?-

**KB6:**

Here are some simple clauses.

likes(mary,food).

likes(mary,wine).

likes(john,wine).

likes(john,mary).

The following queries yield the specified answers.

| ?- likes(mary,food).

yes.

| ?- likes(john,wine).

yes.

| ?- likes(john,food).

no.

How do you add the following facts?

1. John likes anything that Mary likes

2. John likes anyone who likes wine

3. John likes anyone who likes themselve

% Existing facts

likes(mary, food).

likes(mary, wine).

likes(john, wine).

likes(john, mary).

% New facts

likes(john, X) :- likes(mary, X). % John likes anything that Mary likes

likes(john, Y) :- likes(Y, wine). % John likes anyone who likes wine

likes(john, Z) :- likes(Z, Z).    % John likes anyone who likes themselves

% Queries and their answers

% Query: likes(mary, food).

% Answer: yes.

% Explanation: Mary likes food (existing fact).

% Query: likes(john, wine).

% Answer: yes.

% Explanation: John likes wine (existing fact).

% Query: likes(john, food).

% Answer: no.

% Explanation: John does not like food (not explicitly defined).

% Existing facts

likes(mary, food).

likes(mary, wine).

likes(john, wine).

likes(john, mary).

% New facts and rules

likes(john, X) :- likes(mary, X).

% John likes anything that Mary likes.

% This rule means that if Mary likes something X, then John also likes X.

Likes(john, Y) :- likes(Y, wine).

% John likes anyone who likes wine.

% This rule means that if someone Y likes wine, then John also likes Y.

likes(john, Z) :- likes(Z, Z).

% John likes anyone who likes themselves.

% This rule means that if someone Z likes themselves, then John also likes Z.

% Queries and their answers

% Query: likes(mary, food).

% Answer: yes.

% Explanation: Mary likes food (existing fact).

% Query: likes(john, wine).

% Answer: yes.

% Explanation: John likes wine (existing fact).

% Query: likes(john, food).

% Answer: no.

% Explanation: John does not like food (not explicitly defined).

**RESULT:**

Thus to implement the PROLOG has been executed successfully and verified 

|  |  |
| --- | --- |
| **EX.N0 : 13** | **UNIFICATION AND RESOLUTION** |
| **DATE :25/05/2024** |

**AIM:**

To execute programs based on Unification and Resolution.

**Example1:**In the below prolog program, unification and instantiation take place after querying

Facts :

likes(john,jane). likes(jane, john).

Query:

?- likes(john, X). Answer:X=jane.

Here upon asking the query first prolog start to search matching terms in predicate with two arguments and it can match likes(john, ...)i.e.Unification. Then it looks for the value of X asked in queryand it returns answer X = jane i.e.Instantiation - X is instantiated to jane.

**Example2:** At the prolog query prompt,when you write below query,

?-owns(X,car(bmw))=owns(Y,car(C)). You will get Answer : X = Y, C = bmw.

Here owns(X,car(bmw))and owns(Y,car(C))unifies– because

1. Predicate names are same on both side
2. Number of arguments for thatpredicate,i.e.2,are equal both side.
3. 2nd argument with predicate inside the brackets are same both side and even in that predicate again number of arguments are same. So, here terms unify in which X=Y. So, Y is substituted with X--i.e. written as{X|Y}and C is instantiated to bmw, -- written as{bmw | C} and this is called Unification with Instantiation.

But when you write?-owns(X,car(bmw))= likes(Y,car(C)). Then prolog will return False, since it can not match the ;owns; and ;likes; predicates.

Resolution is one kind of proof technique that works this way–

1. Select two clauses
2. Combine those two clauses and cancel out the conflicting terms.

For example we have following statements,

1. If it is a pleasant day you will do strawberry picking
2. If you are doing strawberry picking you are happy.

Above statements canbe written in propositional logic like this-

1. strawberry\_picking←pleasant
2. happy←strawberry\_picking

And again these statement scan be written in CNF like this-

1. (strawberry\_picking∨~pleasant)∧
2. (happy∨~strawberry\_picking)

By resolving these two clauses and cancelling out the conflicting terms

;strawberry\_picking;and;~strawberry\_picking;,we can have one new clause,

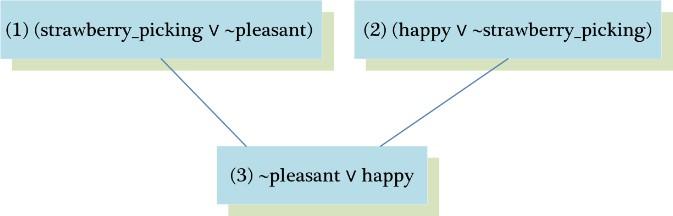
1. ~pleasant ∨happy

How ?Seethefigureonright.

When we write above new clause in infer or implies form,we have

;pleasant→happy;or;happy←pleasant;

i.e.If it is a pleasant day you are happy.



* But sometimes from the collection of the statements we have, we want to know the answer of this question-“Is it possible to prove me other statements from what we actually know?”In order to prove this we need to make some inferences and those other statements can be shown true using Refutation proof method i.e. proof by contradiction using Resolution. So for the asked goal we will negate the goal and will add it to the given statements to prove the contradiction.

* Let’s see an example to understand how Resolution and Refutation work. In below example, Part(I) represents the English meanings for the clauses, Part(II) represents the propositional logic statements for given english sentences, Part(III) represents the ConjunctiveNormalForm(CNF)ofPart(II)and Part(IV)shows some other statements we want to prove using Refutation proof method.

**Part(I):English Sentences**

1. If it is sunny and warm day you will enjoy.
2. If it is warm and pleasant day you will do strawberry picking
3. If it is raining then no strawberry picking.
4. If it is raining you will get wet.
5. It is warmday
6. It is raining
7. It is sunny

**Part(II):PropositionalStatements**

1. enjoy←sunny∧warm
2. strawberry\_picking ←warm∧pleasant
3. ~strawberry\_picking←raining
4. wet←raining
5. warm
6. raining
7. sunny

**Part(III):CNFof Part(II)**

1. (enjoy∨~sunny∨~warm)∧
2. (strawberry\_pickingwarmpleasant)
3. (~strawberry\_pickingraining)
4. (wet∨~raining)
5. (warm)^
6. (raining)∧
7. (sunny)

**Part(IV):Other statements we want to prove by refutation**

(Goal 1) You are not doing strawberry picking.

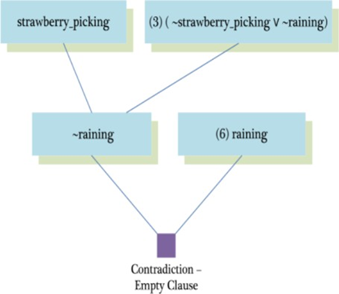
(Goal2)You will enjoy.

(Goal 3) Try it yourself : You will get wet.

Goal1:You are not doing strawberry picking.

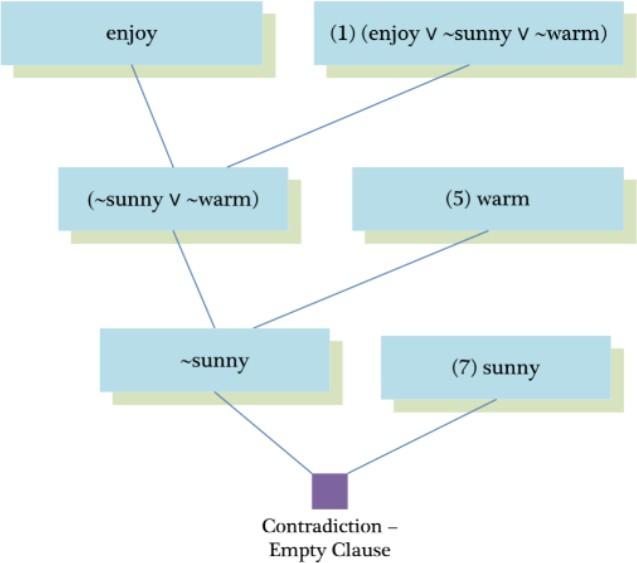
Prove :~strawberry\_picking

Assume: strawberry\_picking (negate the goal and add it to given clauses)



Goal2:You will enjoy.

Prove :enjoy

Assume:~enjoy(negate the goal and add it to given clauses)

**SOURCECODE:**

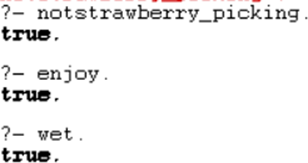
enjoy:-sunny,warm.

strawberrry\_picking:-warm,plesant.

notstrawberry\_picking:-raining. wet:-raining.

warm. raining. sunny.

**OUTPUT:**

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**RESULT:**

Thus to implement programs based on Unification and resolution  
algorithm has been executed successfully

|  |  |
| --- | --- |
| **EX.N0 : 14** | **FUZZYLOGIC – IMAGE PROCESSING** |
| **DATE :01/06/2024** |

**AIM:** To develop a fuzzy logic for image processing

An edge is a boundary between two uniform regions. You can detect an edge by comparing the intensity of neighbouring pixels. However, because uniform regions are not crisply defined,small intensity differences between two neighbouring pixels do not always represent an edge. Instead, the intensity difference might represent a shading effect. The fuzzy logic approach for image processing allows you to use membership functions to define the degree to which a pixel belongs to an edge or a uniform region.

**PROGRAM**

Import RGB Image and Convert to Grayscale Import the image.

Irgb=imread('peppers.png');

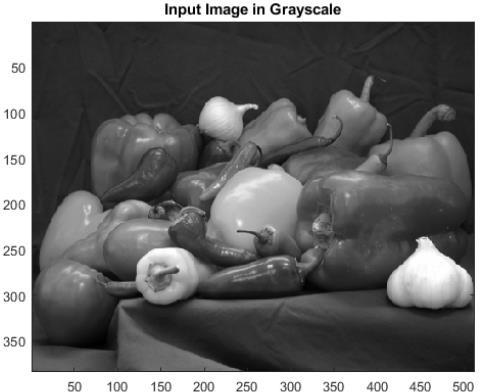
Irgb is a 384 x 512 x 3 uint8 array. The three channels of Irgb (third array dimension) represent the red, green, and blue intensities of the image.

Convert Irgbto grayscale so that you can work witha2-Darrayinstead ofa3-Darray. To do so, use the rgb2gray function.

Igray=rgb2gray(Irgb); figure

image(Igray,'CDataMapping','scaled') colormap('gray')

title('Input Image in Grayscale')



**Convert Image to Double-Precision Data**

The eval fis function for evaluating fuzzy inference systems supports only single-precision and double-precision data.

Therefore, convert I gray to adoublearrayusingtheim2doublefunction.

I = im2double(Igray);

**ObtainImageGradient**

The fuzzy logic edge-detection algorithm for this example relies on the image gradient to locate breaks in uniform regions. Calculate the image gradient along the x-axis and y-axis.

Gx and Gy are simple gradient filters. To obtain a matrix containing the x-axis gradients of I, you convolve I with Gx using the conv2 function. The gradient values are in the [-1 1] range. Similarly, to obtain the y-axis gradients of I, convolve I with

Gy.

Gx=[-11]; Gy = Gx';

Ix=conv2(I,Gx,'same');

Iy= conv2(I,Gy,'same');

**Plot the image gradients.**

figure image(Ix,'CDataMapping','scaled') colormap('gray')

title('Ix')

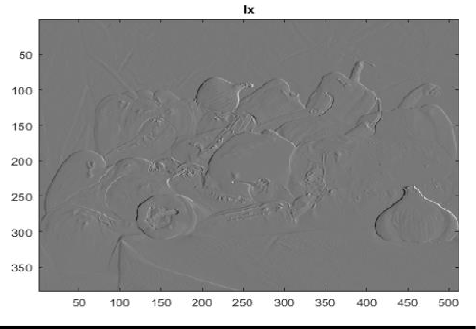
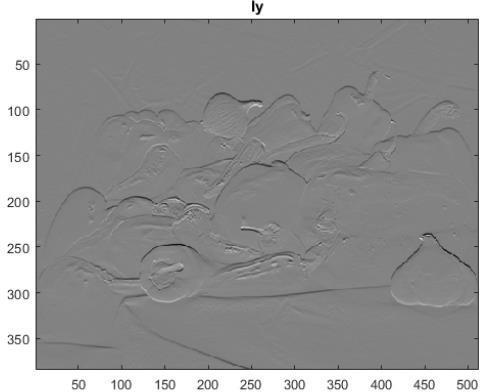


figure image(Iy,'CDataMapping','scaled') colormap('gray')

title('Iy')



Define FuzzyInferenceSystem(FIS)forEdgeDetection

Createafuzzyinferencesystem(FIS)for edgedetection,edgeFIS.

EdgeFIS=mamfis('Name','edgeDetection');

Specifythe imagegradients,IxandIy,asthe inputsofedgeFIS.

edgeFIS = addInput(edgeFIS,[-1 1],'Name','Ix');

edgeFIS=addInput(edgeFIS,[-11],'Name','Iy');

Specify azero-mean Gaussian membership function for each input.If  the gradient value for a pixel is 0, then it belongs to

The zero membership function withadegreeof1. sx = 0.1;

sy=0.1;

edgeFIS=addMF(edgeFIS,'Ix','gaussmf',[sx0],'Name','zero');

edgeFIS=addMF(edgeFIS,'Iy','gaussmf',[sy0],'Name','zero');

sx and sy specify the standard deviation for the zero membership function for the Ix and Iy inputs.

To adjust the edge detector performance, you can change the values of sx and sy. Increasing the values makes the algorithm less sensitive to the edges in the image and decreases the intensity of the detected edges.

Specify the intensity of the edge-detected image as an output of edge FIS. EdgeFIS = addOutput(edgeFIS,[0 1],'Name','Iout');

Specify the triangular membership functions, white and black,for I out.

wa = 0.1;

wb=1;

wc =1;

ba=0;

bb=0;

bc=0.7;

edgeFIS=addMF(edgeFIS,'Iout','trimf',[wawbwc],'Name','white');

edgeFIS=addMF(edgeFIS,'Iout','trimf',[babbbc],'Name','black');

As you can with sx and sy, you can change the values of wa,wb,wc,ba,bb,and bc to adjust the edge detector performance. The triplets specify the start, peak, and end of the triangles of the membership functions. These parameters influence the intensity of the detected edges.

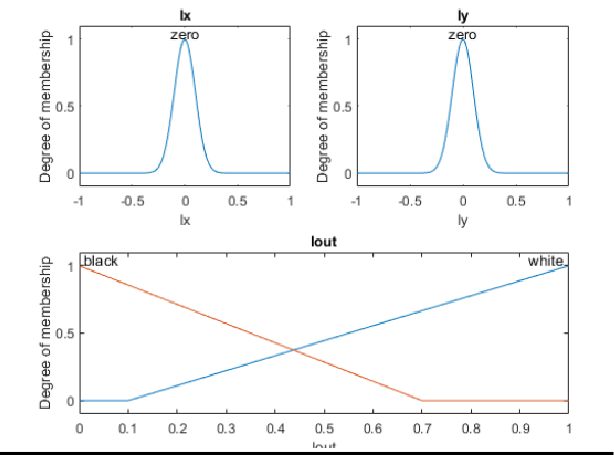
**Plot the membership function so the inputs and outputs of edge FIS.** figure

Subplot(2,2,1) plotmf(edgeFIS,'input',1) title('Ix')

subplot(2,2,2) plotmf(edgeFIS,'input',2) title('Iy')

subplot(2,2,[3 4])

plotmf(edgeFIS,'output',1) title('Iout')



**SpecifyFIS Rules**

Add rules to make a pixel white if it belongs to a uniform region and black otherwise. A pixel is in a uniform region when the image gradient is zero in both directions. If either direction has a nonzero gradient, then the pixel is on an edge.

r1 ="IfIx is zero and Iy is zero then Iout is white";

r2="If Ix is not zero or Iy is not zero then Iout is black";

edgeFIS = addRule(edgeFIS,[r1 r2]);

edgeFIS.Rulesans =

1x2fisrulearraywithproperties: Description

Antecedent Consequent Weight Connection

 Details: Description

1. "Ix==zero&Iy==zero=>Iout=white(1)"
2. "Ix~=zero|Iy~=zero=>Iout=black(1)"

Evaluate FIS

Evaluate the output of the edge detector for each row of pixels in I using corresponding rows of Ix and Iy as inputs.

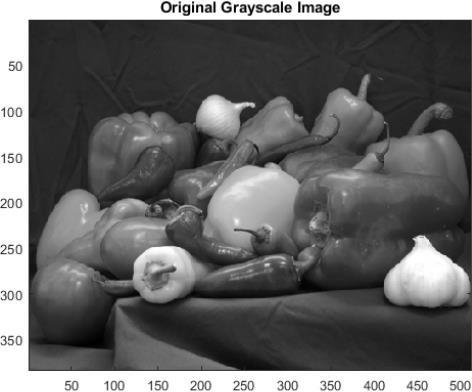
Ieval=zeros(size(I)); for ii = 1:size(I,1)

Ieval(ii,:)=evalfis(edgeFIS,[(Ix(ii,:));(Iy(ii,:))]'); end

**Plot Results**

Plot the original grayscaleimage. figure image(I,'CDataMapping','scaled') colormap('gray')

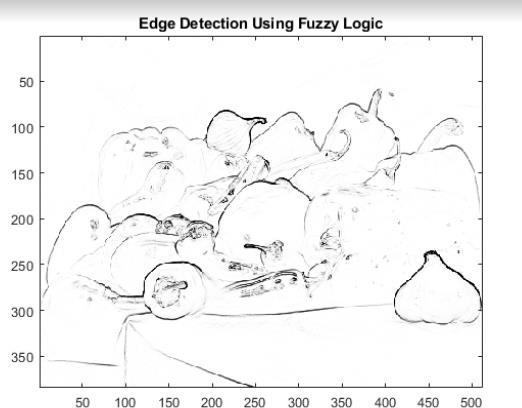
title('Original Grayscale Image')



Plotthedetectededges. figure

image(Ieval,'CDataMapping','scaled') colormap('gray')

title('EdgeDetectionUsingFuzzyLogic')



**RESULT:**

Thus the Fuzzylogic – imageprocessing has been executed and verified successfully