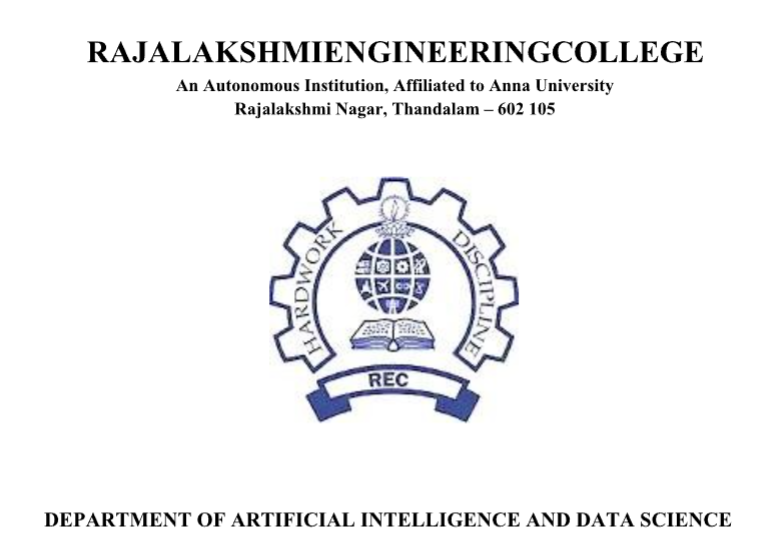
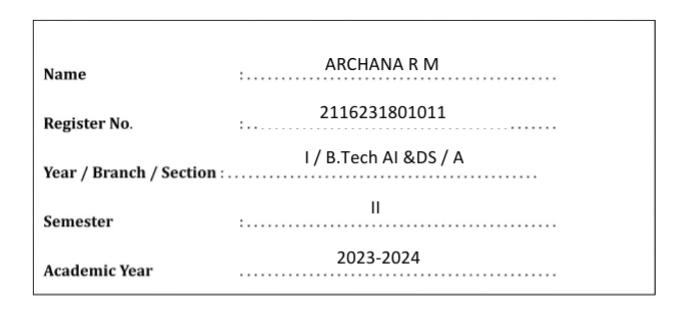
****

**AI23231 – Principle of Artificial Intelligence**

****

**INDEX**

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

|  |  |
| --- | --- |
| **EXP.NO : 1** | **8- QUEENS PROBLEM** |
| **DATE : 20/02/24** |

**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 1:**

def is\_safe(board, row, col):

# Check if there is a queen in the same column

for i in range(row):

if board[i][col] == 'Q':

return False

# Check upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 'Q':

return False

# Check upper diagonal on the right side

for i, j in zip(range(row, -1, -1), range(col, len(board))):

if board[i][j] == 'Q':

return False

return True

def solve\_queen\_util(board, row):

n = len(board)

if row == n:

return True

for col in range(n):

if is\_safe(board, row, col):

board[row][col] = 'Q'

if solve\_queen\_util(board, row + 1):

return True

board[row][col] = '-'

return False

def solve\_queen():

board = [['-' for \_ in range(8)] for \_ in range(8)] # Initialize the chessboard

if solve\_queen\_util(board, 0) == False:

print("Solution does not exist")

return False

print\_board(board)

return True

def print\_board(board):

for row in board:

print(' '.join(row))

solve\_queen()

**SAMPLE OUTPUT:**

Q - - - - - - -

- - - - Q - - -

- - - - - - - Q

- - - - - Q - -

- - Q - - - - -

- - - - - - Q -

- Q - - - - - -

- - - Q - - - -

**OUTPUT:**

Q - - - - - - -

- - - - Q - - -

- - - - - - - Q

- - - - - Q - -

- - Q - - - - -

- - - - - - Q -

- Q - - - - - -

- - - Q - - - -

**RESULT :**

Thus, the experiment to solve the 8-Queen Problem by using Python has been

executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :2** | **DEPTH FIRST SEARCH** |
| **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 2:**

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

component.append(node)

visited[node] = True

for child in graph[node]:

if not visited[child]:

dfs(child, graph, visited, component)

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

graph = {

0: [2],

1: [2, 3],

2: [0, 1, 4],

3: [1, 4],

4: [2, 3]

}

node = 0

visited = [False] \* len(graph)

component = []

dfs(node, graph, visited, component)

print("Following is the Depth-First search:", component)

**SAMPLE OUTPUT:**

Following is the Depth-First search: [0, 2, 1, 3, 4]  
  
  
**OUTPUT:**

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

**RESULT:**

Thus, the experiment to solve Depth First Search by using Python has been executed and verified successfully

|  |  |
| --- | --- |
| **EXP.NO :3** | **WATER JUG PROBLEM USING BFS** |
| **DATE :05/03/2024** |

**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()

if (u[0], u[1]) in m:

continue

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

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)

for i in range(sz):

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

return

q.append([u[0], b])

q.append([a, u[1]])

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

c = u[0] + ap

d = u[1] - ap

if c == a or (d == 0 or 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])

q.append([0, b])

if not isSolvable:

print("No solution")

Jug1, Jug2, target = 4, 3, 2

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

BFS(Jug1, Jug2, target)

**SAMPLE OUTPUT:**

Path from initial state to solution state:

( 0 , 0 )

( 0 , 3 )

( 4 , 0 )

( 4 , 3 )

( 1 , 2 )

( 2 , 1 )

( 2 , 0 )

**OUTPUT:**

Path from initial state to solution state:

( 0 , 0 )

( 0 , 3 )

( 4 , 0 )

( 4 , 3 )

( 1 , 2 )

( 2 , 1 )

( 2 , 0 )

**RESULT:**

Thus, the experiment to solve Water Jug Problem using BFS by using Python has been

executed and verified successfully

|  |  |
| --- | --- |
| **EXP.NO :4** | **WATER JUG PROBLEM USING DFS** |
| **DATE :12/03/2024** |

**AIM:**

To solve the Water jug Problem using Depth First Search.

**ALGORITHM:**

Step1:Start with two empty Jugs: Jug1 &amp; 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:

i. If the current state has been visited before,stop &amp; return“FALSE”.

ii. Mark the Current state as visited.

iii. Try the action: fill jug 1, fill jug 2, empty jug 1, empty jug 2, pour

water from jug 1 to jug2.

iv. 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 &amp; 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

if x <capacity\_x:

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

path.append((x, y))

return True

if y <capacity\_y:

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

path.append((x, y))

return True

if x > 0:

if dfs(0, y, path):

path.append((x, y))

return True

if y > 0:

if dfs(x, 0, path):

path.append((x, y))

return True

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

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"

capacity\_x = 4

capacity\_y = 3

target = 2

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

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 OUTPUT:**

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:**

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

**RESULT:**

Thus, the experiment to solve Water Jug Problem using DFS by using Python has been

executed and verified successfully

|  |  |
| --- | --- |
| **EXP.NO :5** | **A\* SEARCH ALGORITHM** |
| **DATE :19/03/2024** |

**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 :**

from collections import deque

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

n = v

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

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

return reconst\_path

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

if m not in open\_list and m not in closed\_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

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')

**SAMPLE OUTPUT:**

Path found: ['A', 'B', 'D']  
 **OUTPUT:**

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

**RESULT:**

Thus, the experiment to solve the A\*search Algorithm by using Python has been

executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :6** | **BREADTH FIRST SEARCH** |
| **DATE :26/03/2024** |

**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 defaultdict

class Graph:

def \_\_init\_\_(self):

self.graph = defaultdict(list)

def addEdge(self, u, v):

self.graph[u].append(v)

def BFS(self, s):

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

queue = []

queue.append(s)

visited[s] = True

while queue:

s = queue.pop(0)

print(s, end=" ")

for i in self.graph[s]:

if not visited[i]:

queue.append(i)

visited[i] = True

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

g = 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("Following is the breadth First traversal (starting from vertex 2):")

g.BFS(2)

**SAMPLE OUTPUT** :

Following is the breadth First traversal (starting from vertex 2):

2 0 3 1

**OUTPUT :**

Following is the breadth First traversal (starting from vertex 2):

2 0 3 1

**RESULT:**

Thus, the experiment to solve the Breadth First Search by using Python has been

executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :7** | **AO\* SEARCH ALGORITHM** |
| **DATE :02/04/2024** |

**AIM:**

To implement the AO\* Search Algorithm using python

**ALGORITHM:**

Step1:Start

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

Step3: While the open list is not empty:

i. Select the node with the lowest estimated total cost (f-value).

ii. If the selected node is the goal, return the solution.

iii. Generate successor nodes ,calculate their costs, and add them to

the openlist if they are better or not in the closed list.

iv. 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

elifnode.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 = Node.a\_star\_search(initial\_state, is\_goal, successors, heuristic)

if path:

print("Path found:", path)

else:

print("No path found")

**SAMPLE OUTPUT:**

Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]  
  
**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 the AO\* Search Algorithm by using Python has been

executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :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&#39;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, self.heuristic))

return successors

@staticmethod

def calculate\_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) # Fixed line

if result is not None:

return result

initial\_state = 0

goal\_state = 5

initial\_node = Node(initial\_state, None, 0, Node.calculate\_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 is found")  
  
  
**SAMPLE OUTPUT:**

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

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

**RESULT:**

Thus, the experiment to solve the Recursive Best First Search Algorithm by using Python has been executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :9** | **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)

**SAMPLE OUTPUT:**

Solution exists and following are the assigned colours:

1 2 3 2

**OUTPUT:**

Solution exists and following are the assigned colours:

1 2 3 2

**RESULT:**

Thus, the experiment to solve the Map colouring by using Python has been

executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :10** | **MINIMAX** |
| **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 = int(math.log2(len(scores)))

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

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

**SAMPLE OUTPUT :**

The optimal value is:12

**OUTPUT:**

The optimal value is:12

**RESULT:**

Thus, the experiment to solve the Min Max Algorithm by using Python has been

executed and verified successfully.

|  |  |
| --- | --- |
| **EXP.NO :11** | **ALPHA-BETA PURNING** |
| **DATE :30/04/2024** |

**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, 0, -1]

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

**SAMPLE OUTPUT:**

The optimal value is: 5

**OUTPUT:**

The optimal value is: 5

**RESULT:**

Thus, the experiment to solve the Alpha Beta Pruning 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**:

KNOWLEDGE BASE:

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).

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).

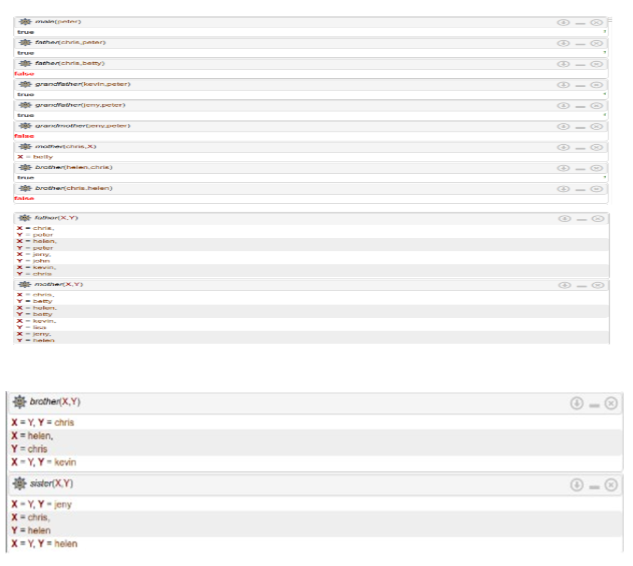
brother(X,Y):-male(Y), father(X,Z),

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** |

**AIM**:

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.

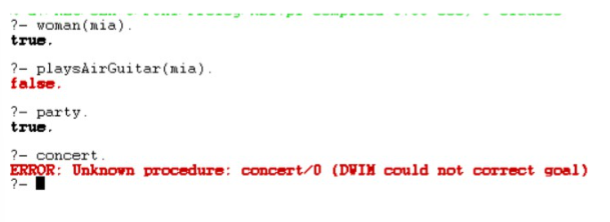
Query1:?-woman(mia).

Query2:?-playsAirGuitar(mia).

Query3:?-party.

Query4:?-concert

**OUTPUT**:



**KB2**:

happy(yolanda).

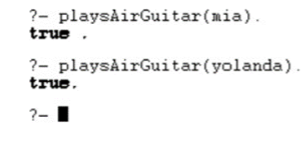
listens2music(mia).

Listens2music(yolanda):-happy(yolanda).

playsAirGuitar(mia):-listens2music(mia).

playsAirGuitar(Yolanda):-listens2music(yolanda).

**OUTPUT**:



**KB3**:

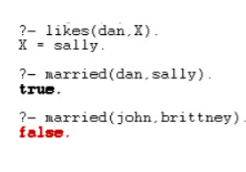
likes(dan,sally). likes(sally,dan).

likes(john,brittney).

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

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

**OUTPUT**:



**KB4**:

food(burger).

food(sandwich).

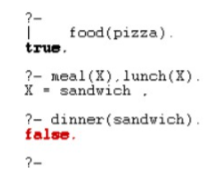
food(pizza).

lunch(sandwich).

dinner(pizza).

meal(X):-food(X).

**OUTPUT**:



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

find\_max(X,Y,X):-X&gt;=Y,!.

find\_max(X,Y,Y):-X&lt;Y.

find\_min(X,Y,X):-X=&lt;Y,!.

find\_min(X,Y,Y):-X&gt;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

|  |  |
| --- | --- |
| **EXP.NO :13** | **Unification And Resolution** |
| **DATE :25/05/2024** |

**AIM**:

To execute programs based on Unification and Resolution.

**PROGRAM**:

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

|  |  |
| --- | --- |
| **EXP.NO :14** | **Fuzzylogic – Image Processing** |
| **DATE :01/06/2024** |

**AIM**:

To develop a fuzzy logic for image processing

**PROGRAM:**

Import RGB Image and Convert to Grayscale Import the

image.

Irgb=imread(&#39;peppers.png&#39;);

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

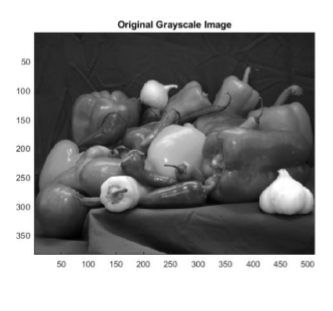
function.

Igray=rgb2gray(Irgb); figure

image(Igray,&#39;CDataMapping&#39;,&#39;scaled&#39;) colormap(&#39;gray&#39;)

title(&#39;Input Image in Grayscale&#39;)

**OUTPUT**:



**RESULT**:

Thus the Fuzzy logic – image processing has been executed and verified successfully