RAJALAKSHMI ENGINEERING COLLEGE (Autonomous)

RAJALAKSHMI NAGAR, THANDALAM, CHENNAI-602105

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AI19341

Principles of Artificial Intelligence Lab

THIRD YEAR

FIFTH SEMESTER

220701205

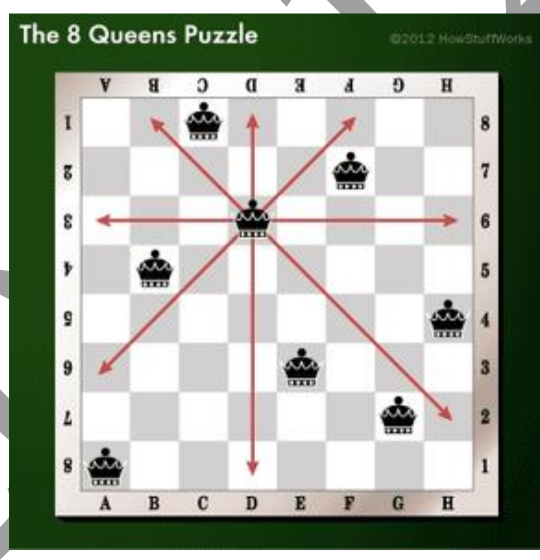
EX.NO:1

8- QUEENS PROBLEM

AIM :

To implement an 8-Queesns problem using Python.

You are given an 8x8 board; find a way to place 8 queens such that no queen can attack any other queen on the chessboard. A queen can only be attacked if it lies on the same row, same column, or the same diagonal as any other queen. Print all the possible configurations. To solve this problem, we will make use of the Backtracking algorithm. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For the given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8.



SOURCE CODE :

```python
print ("Enter the number of queens")
N = int(input())

board = [[0]*N for _ in range(N)]

def attack(i, j):

    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True

    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queens(N)
for i in board:
    print (i)
```

OUTPUT:

```
Enter the number of queens
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

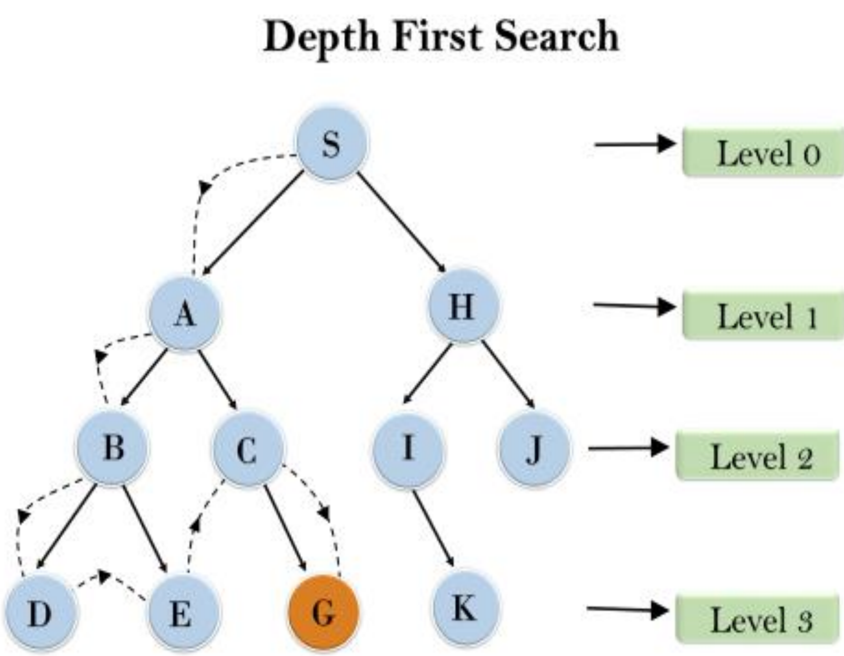

RESULT:

Thus the python code is implement successfully and the output is verified.

EX.NO:2

DEPTH-FIRST SEARCH

- Depth-first search (DFS) algorithm or searching technique starts with the root node of graph G, and then travel deeper and deeper until we find the goal node or the node which has no children by visiting different node of the tree.
- The algorithm, then backtracks or returns back from the dead end or last node towards the most recent node that is yet to be completely unexplored.
- The data structure (DS) which is being used in DFS Depth-first search is stack. The process is quite similar to the BFS algorithm.
- In DFS, the edges that go to an unvisited node are called discovery edges while the edges that go to an already visited node are called block edges.



AIM :
To implement a depth-first search problem using Python.

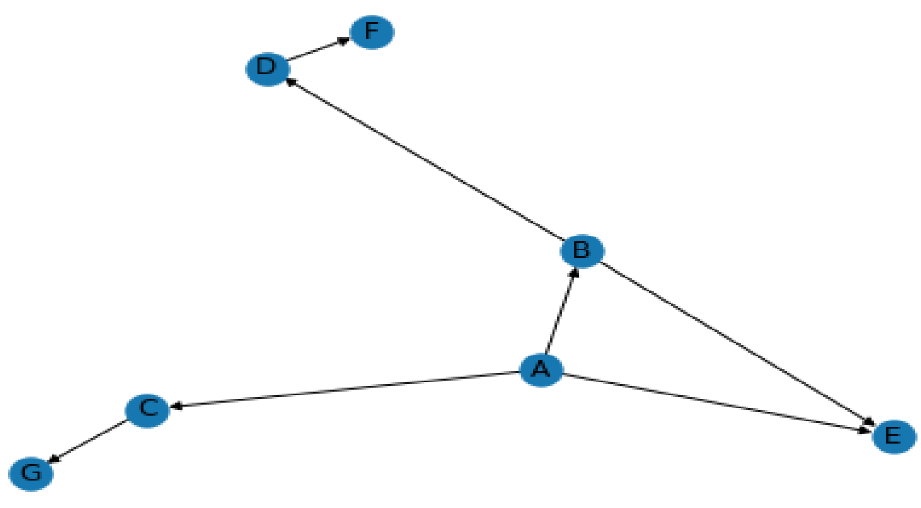
SOURCE CODE :

```
import networkx as nx

#FUNCTION TO SOLVE DFS
```

```python
def solveDFS(graph, v, visited) :
  visited.add(v)
  print(v, end=' ')
  for neighbour in graph[v] :
    if neighbour not in visited :
      solveDFS(graph, neighbour, visited)

g = nx.DiGraph()

#CREATE A GRAPH USING NETWORKX
g.add_edges_from([('A','B'),('A','C'),('C','G'),('B','D'),('B','E'),('D','F'),('A','E')])
# Add edges for that graph
nx.draw(g, with_labels=True) # Graph Visualization

#SOLVE DFS FOR THAT GRAPH
print("Following is DFS from (starting from vertex A)")
visited = set()
solveDFS(g, 'A', visited)
```

OUTPUT :

```
Following is DFS from (starting from vertex A)
A B D F E C G
```



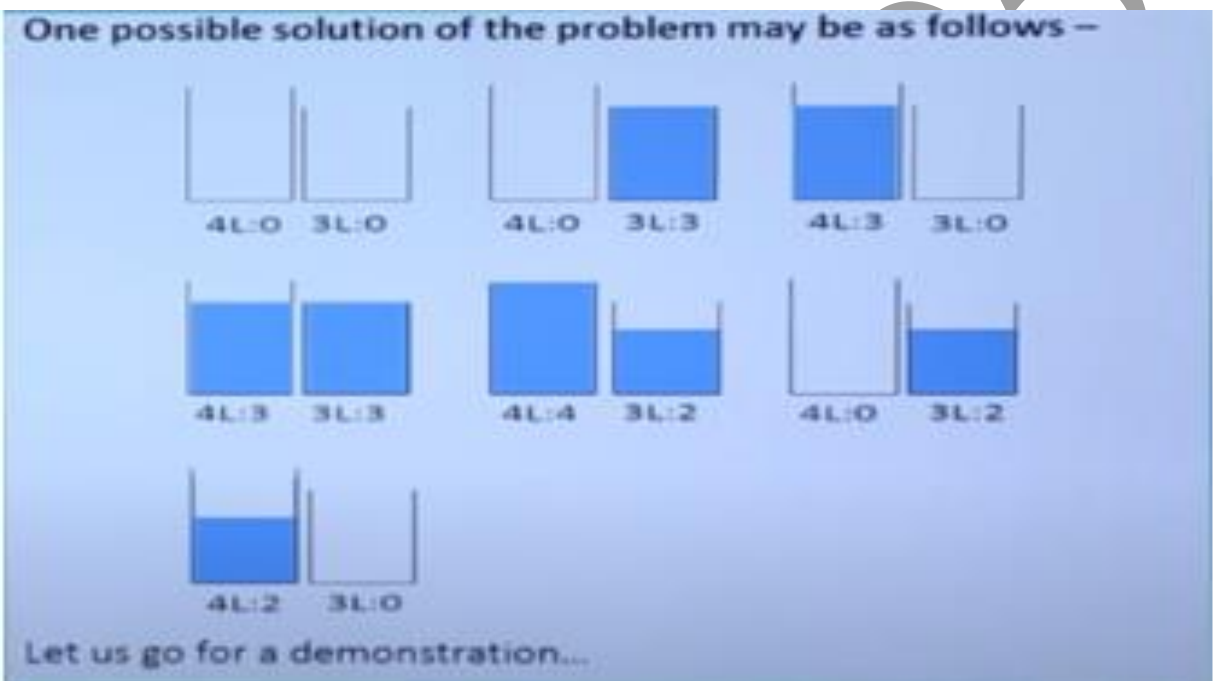
RESULT :

Thus the python code is implement successfully and the output is verified.

EX.NO:3

DEPTH FIRST SEARCH – WATER JUG PROBLEM

In the **water jug problem in Artificial Intelligence**, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.



AIM :

To implement a python program for Water Jug problem using depth first search problem

SOURCE CODE :

```
from collections import deque

def DFS(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], ")")
            break
        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 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])
        q.append([0, b])
    if (not isSolvable):
        print ("No solution")

Jug1, Jug2, target = 4, 3, 2
print("Path from initial state ""to solution state ::")
DFS(Jug1, Jug2, target)
```


OUTPUT :

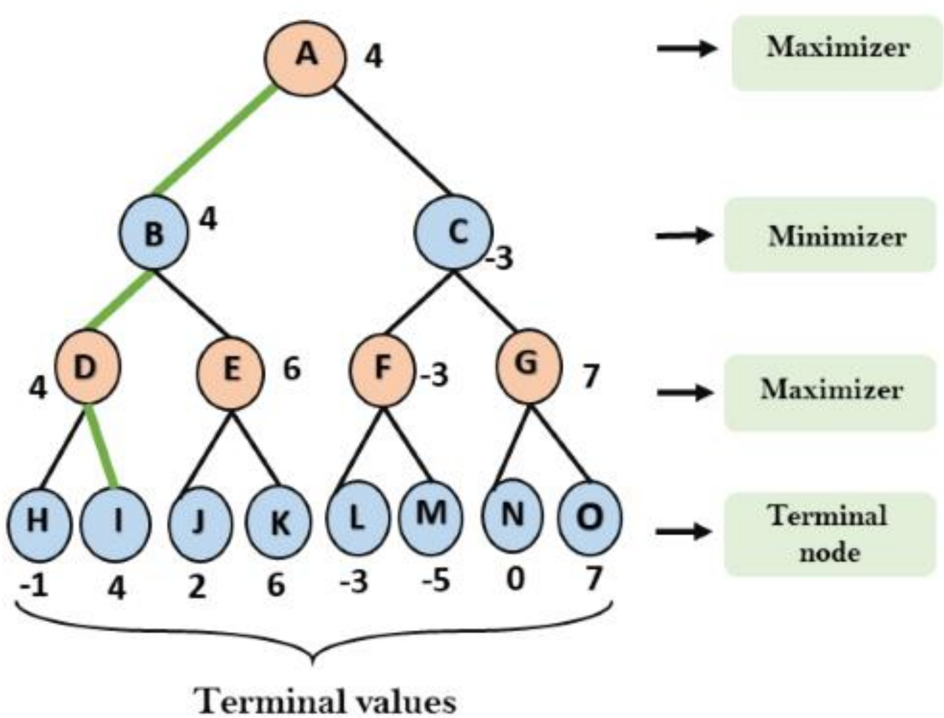
```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

RESULT :
Thus the python code is implement successfully and the output is verified.

EX.NO:4

MINIMAX ALGORITHM

- A simple example can be used to explain how the minimax algorithm works. We've included an example of a game-tree below, which represents a two-player game.
- There are two players in this scenario, one named Maximizer and the other named Minimizer.
- Maximizer will strive for the highest possible score, while Minimizer will strive for the lowest possible score.
- Because this algorithm uses DFS, we must go all the way through the leaves to reach the terminal nodes in this game-tree.
- The terminal values are given at the terminal node, so we'll compare them and retrace the tree till we reach the original state.



AIM :
To implement MINIMAX Algorithm problem using Python.

SOURCE CODE :

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system
HUMAN = -1
COMP = +1
board = [
   [0, 0, 0],
   [0, 0, 0],
   [0, 0, 0],
]
def evaluate(state):
  if wins(state, COMP):
      score = +1
   elif wins(state, HUMAN):
      score = -1
   else:
      score = 0
return score

def wins(state, player):

   win_state = [
      [state[0][0], state[0][1], state[0][2]],
      [state[1][0], state[1][1], state[1][2]],
      [state[2][0], state[2][1], state[2][2]],
      [state[0][0], state[1][0], state[2][0]],
      [state[0][1], state[1][1], state[2][1]],
      [state[0][2], state[1][2], state[2][2]],
      [state[0][0], state[1][1], state[2][2]],
      [state[2][0], state[1][1], state[0][2]],
   ]
   if [player, player, player] in win_state:
      return True
   else:
      return False
def game_over(state):
        return wins(state, HUMAN) or wins(state, COMP)
def empty_cells(state):
   cells = []
   for x, row in enumerate(state):
      for y, cell in enumerate(row):
```

```
            if cell == 0:
                cells.append([x, y])

    return cells
def valid_move(x, y):

    if [x, y] in empty_cells(board):
        return True
    else:
        return False
def set_move(x, y, player):

    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False
def minimax(state, depth, player):
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]
    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score  # max value
        else:
            if score[2] < best[2]:
                best = score  # min value

    return best

def clean():
```

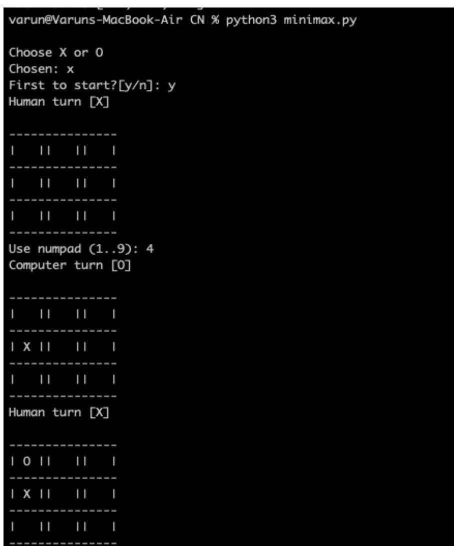
```
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')

def render(state, c_choice, h_choice):

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)

def ai_turn(c_choice, h_choice):

    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)

def human_turn(c_choice, h_choice):
```

```
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

def main():

    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
```

```
        h_choice = input('Choose X or O\nChosen: ').upper()
    except (EOFError, KeyboardInterrupt):
        print('Bye')
        exit()
    except (KeyError, ValueError):
        print('Bad choice')

# Setting computer's choice
if h_choice == 'X':
    c_choice = 'O'
else:
    c_choice = 'X'

# Human may starts first
clean()
while first != 'Y' and first != 'N':
    try:
        first = input('First to start?[y/n]: ').upper()
    except (EOFError, KeyboardInterrupt):
        print('Bye')
        exit()
    except (KeyError, ValueError):
        print('Bad choice')

# Main loop of this game
while len(empty_cells(board)) > 0 and not game_over(board):
    if first == 'N':
        ai_turn(c_choice, h_choice)
        first = ''

    human_turn(c_choice, h_choice)
    ai_turn(c_choice, h_choice)

if wins(board, HUMAN):
    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)
    print('YOU WIN!')
elif wins(board, COMP):
    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)
    print('YOU LOSE!')
```

```
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')

    exit()
if __name__ == '__main__':
    main()
```

OUTPUT:



```
varun@Varuns-MacBook-Air CN % python3 minimax.py

Choose X or O
Chosen: x
First to start?[y/n]: y
Human turn [X]

---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 4
Computer turn [O]

---------------
|   ||   ||   |
---------------
| X ||   ||   |
---------------
|   ||   ||   |
---------------
Human turn [X]

---------------
| O ||   ||   |
---------------
| X ||   ||   |
---------------
|   ||   ||   |
---------------
```

```
Use numpad (1..9): 2
Computer turn [O]

---------------
| O || X ||   |
---------------
| X ||   ||   |
---------------
|   ||   ||   |
---------------
Human turn [X]

---------------
| O || X ||   |
---------------
| X || O ||   |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 3
Computer turn [O]

---------------
| O || X || X |
---------------
| X || O ||   |
---------------
|   ||   ||   |
---------------
Computer turn [O]

---------------
| O || X || X |
---------------
| X || O ||   |
---------------
|   ||   || O |
---------------
YOU LOSE!
varun@Varuns-MacBook-Air CN %
```

RESULT :

Thus the python code is implement successfully and the output is verified.

EX.NO :5

A* SEARCH ALGORITHM

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

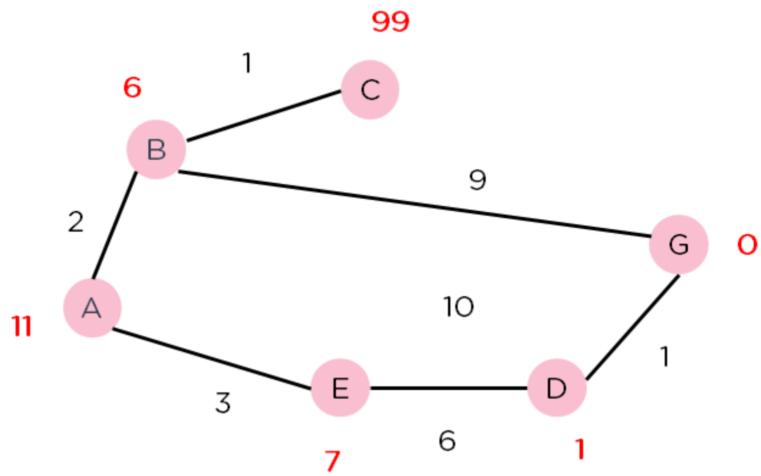
All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost.



AIM :

To implement an A* search algorithm using Python.

SOURCE CODE:

```
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
```

```
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]

    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])
        poo = {}
        poo[start] = 0
        par = {}
        par[start] = start

        while len(open_lst) > 0:
            n = None
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop:
                reconst_path = []

                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()
```

```
                print('Path found: {}'.format(reconst_path))
                return reconst_path


            for (m, weight) in self.get_neighbors(n):
              # if the current node is not present in both open_lst and closed_lst
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)
            open_lst.remove(n)
            closed_lst.add(n)

        print('Path does not exist!')
        return None

adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

OUTPUT:

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

RESULT :

Thus the python code is implement successfully and the output is verified.

EX.NO :6

PROLOG

AIM :

To develop a family tree program using PROLOG with all possible facts, rules, and queries.

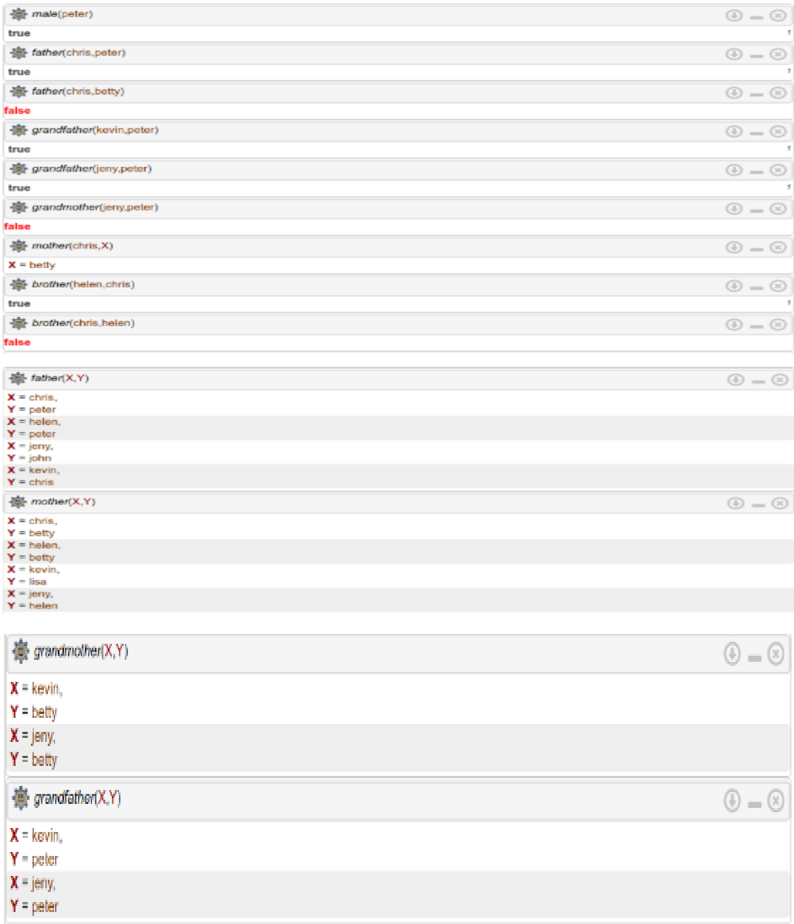
SOURCE CODE:

KNOWLEDGE BASE:

```
/*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 the python code is implement successfully and the output is verified.

EX.NO:7

-

INTRODUCTION TO PROLOG

AIM

To learn PROLOG terminologies and write basic programs.

TERMINOLOGIES

1. Atomic Terms: -

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

   Ex:

   dog
   ab_c_321

2. Variables: -

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

   Ex:

   Dog
   Apple_420

3. Compound Terms: -

   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 (a predicate) and a body (a sequence of predicates separated by commas).

   Ex:

   is_smaller(X,Y):-is_bigger(Y,X).
   aunt(Aunt,Child):-sister(Aunt,Parent),parent(Parent,Child).

SOURCE CODE:

KB1:

```
woman(mia).
```

woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
Query 1: ?-woman(mia).
Query 2: ?-playsAirGuitar(mia).
Query 3: ?-party.
Query 4: ?-concert.

OUTPUT: -

```
?- woman(mia).
true.

?- playsAirGuitar(mia).
false.

?- party.
true.

?- concert.
ERROR: Unknown procedure: concert/0 (DWIM could not correct goal)
?-
```

KB2:

happy(yolanda).
listens2music(mia).
Listens2music(yolanda):-happy(yolanda).
playsAirGuitar(mia):-listens2music(mia).
playsAirGuitar(Yolanda):-listens2music(yolanda).

OUTPUT: -

```
?- playsAirGuitar(mia).
true .

?- playsAirGuitar(yolanda).
true.

?-
```

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: -

```
?- likes(dan,X).
X = sally.

?- married(dan,sally).
true.

?- married(john,brittney).
false.
```

KB4:

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

OUTPUT:

```
?-
|    food(pizza).
true.

?- meal(X),lunch(X).
X = sandwich .

?- dinner(sandwich).
false.

?-
```

KB5:

owns(jack,car(bmw)).
owns(john,car(chevy)).
owns(olivia,car(civic)).
owns(jane,car(chevy)).
sedan(car(bmw)).
sedan(car(civic)).
truck(car(chevy)).

OUTPUT:

```
?-
|    owns(john,X).
X = car(chevy).

?- owns(john,_).
true.

?- owns(Who,car(chevy)).
Who = john ,

?- owns(jane,X),sedan(X).
false.

?- owns(jane,X),truck(X).
X = car(chevy).
```


RESULT:

Thus the python code is implement successfully and the output is verified.

EX.NO :8

UNIFICATION AND RESOLUTION

AIM:

To execute programs based on Unification and Resolution.

Deduction in prolog is based on the Unification and Instantiation. Let's understand these
terminologies by examples rather than by definitions. Remember one thing, matching terms are unified and variables get instantiated. In other words, "Unification leads to Instantiation".

Example 1: Let's see for below prolog program - how 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 'Facts' in top-down
manner for 'likes' predicate with two arguments and it can match likes(john, ...) i.e.
Unification. Then it looks for the value of X asked in query and it returns answer X = jane i.e.
Instantiation - X is instantiated to 'jane'.

Example 2 : 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 (i) predicate
names 'owns' are same on both side (ii) number of arguments for that predicate, i.e. 2, are
equal both side. (iii) 2nd argument with 'car' 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 - (i) select two clauses that contain conflicting terms (ii) combine those two clauses and (iii) 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 can be written in propositional logic like this -

(1) strawberry_picking ← pleasant
(2) happy ← strawberry_picking

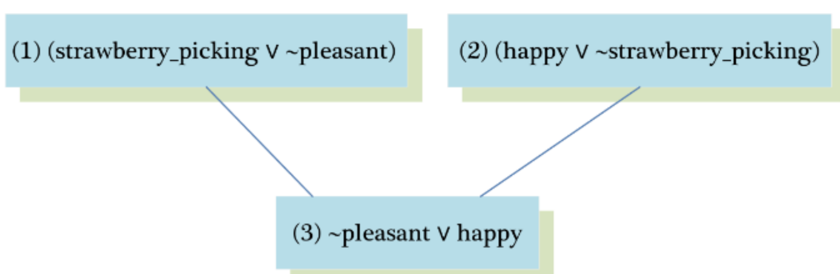
And again these statements can be written in CNF like this -

(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,

(3) ~pleasant ∨ happy

How ? See the figure on right.
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 some 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 Conjunctive Normal Form (CNF) of Part(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 warm day

(6) It is raining

(7) It is sunny

Part(II) : Propositional Statements

(1) enjoy ← sunny ∧ warm

(2) strawberry_picking ← warm ∧ pleasant

(3) ~strawberry_picking ← raining

(4) wet ← raining

(5) warm

(6) raining

(7) sunny

Part(III) : CNF of Part(II)

(1) (enjoy ∨~sunny∨~warm) ∧

(2) (strawberry_picking ∨~warm∨~pleasant) ∧

(3) (~strawberry_picking ∨~raining) ∧

(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.

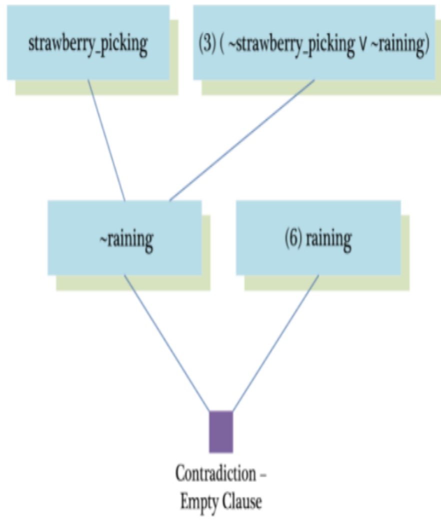
(Goal 2) You will enjoy.

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

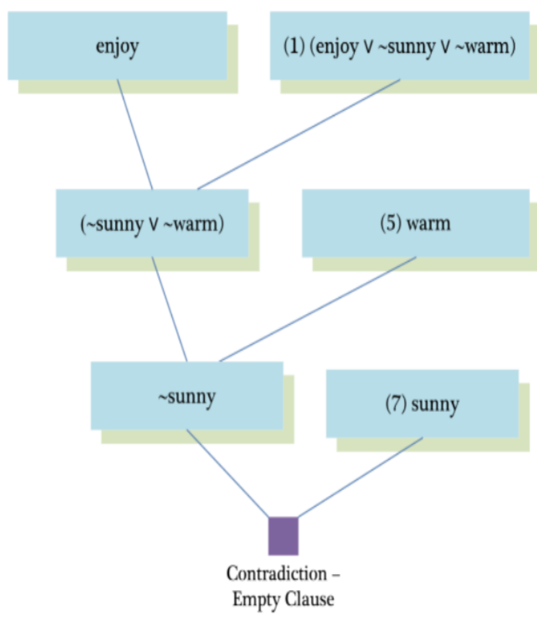
Goal 1 : You are not doing strawberry picking.

Prove : ~strawberry_picking

Assume : strawberry_picking (negate the goal and add it to given clauses).



Goal 2 : You will enjoy.
Prove : enjoy
Assume : ~enjoy (negate the goal and add it to given clauses)



SOURCE CODE:

```
enjoy:-sunny,warm.
strawberrry_picking:-warm,plesant.
```

notstrawberry_picking:-raining.
wet:-raining.
warm.
raining.
sunny.

OUTPUT:

```
?- notstrawberry_picking.
true.

?- enjoy.
true.

?- wet.
true.
```


RESULT:

Thus the python code is implement successfully and the output is verified.

EX.NO :9

FUZZY LOGIC – 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.
Import RGB Image and Convert to Grayscale

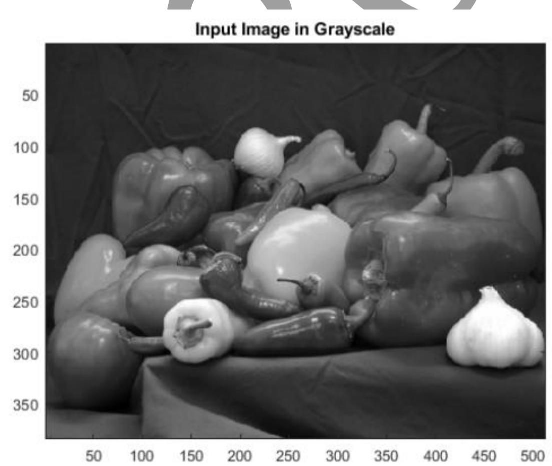
Import the image.

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 Irgb to grayscale so that you can work with a 2-D array instead of a 3-D array. 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 evalfis function for evaluating fuzzy inference systems supports only single-precision and double-precision data.
Therefore, convert Igray to a double array using the im2double function.

```
I = im2double(Igray);
```

Obtain Image Gradient

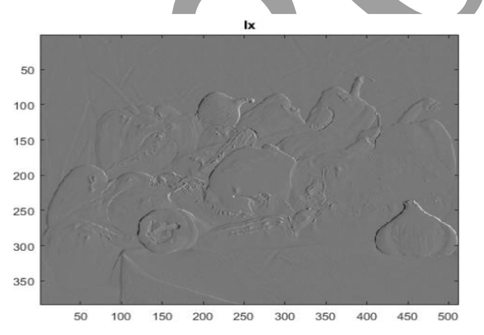
The fuzzy logic edge-detection algorithm for this example relies on the image gradient to locate breaks in uniform regions. Calculate the image gradient along the x-axis and y-axis.

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.

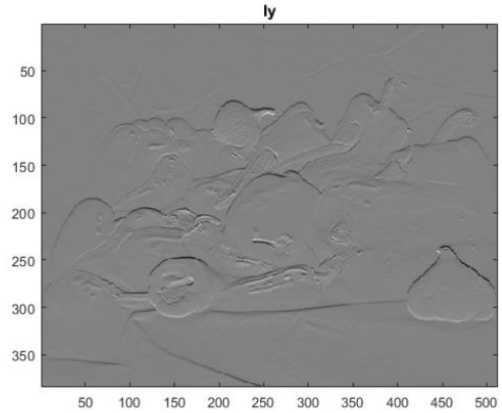
```
Gx = [-1 1];
Gy = Gx';
Ix = conv2(I,Gx,'same');
Iy = conv2(I,Gy,'same');
```

Plot the image gradients.

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




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



Define Fuzzy Inference System (FIS) for Edge Detection
Create a fuzzy inference system (FIS) for edge detection, edgeFIS.

edgeFIS = mamfis('Name','edgeDetection');
Specify the image gradients, Ix and Iy, as the inputs of edgeFIS.
edgeFIS = addInput(edgeFIS,[-1 1],'Name','Ix');
edgeFIS = addInput(edgeFIS,[-1 1],'Name','Iy');

Specify a zero-mean Gaussian membership function for each input. If the gradient value for a pixel is 0, then it belongs to
the zero membership function with a degree of 1.
sx = 0.1;
sy = 0.1;
edgeFIS = addMF(edgeFIS,'Ix','gaussmf',[sx 0],'Name','zero');
edgeFIS = addMF(edgeFIS,'Iy','gaussmf',[sy 0],'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 edgeFIS.
edgeFIS = addOutput(edgeFIS,[0 1],'Name','Iout');

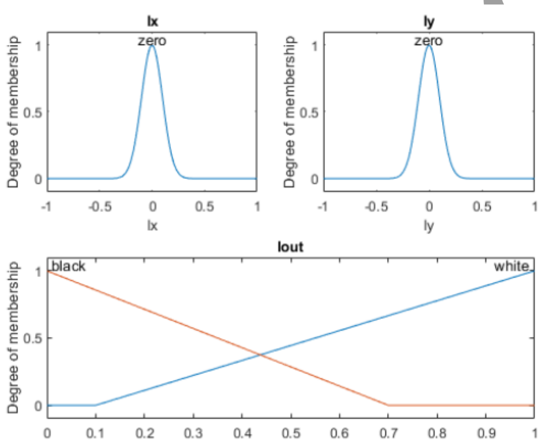
Specify the triangular membership functions, white and black, for Iout.
wa = 0.1;
wb = 1;
wc = 1;
ba = 0;
bb = 0;
bc = 0.7;

```
edgeFIS = addMF(edgeFIS,'Iout','trimf',[wa wb wc],'Name','white');
edgeFIS = addMF(edgeFIS,'Iout','trimf',[ba bb bc],'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 functions of the inputs and outputs of edgeFIS.

```
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')
```



Specify FIS 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 = "If Ix 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.Rules
ans =
1x2 fisrule array with properties:
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

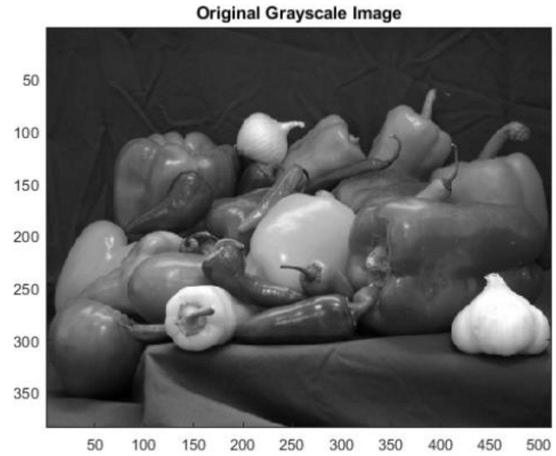
Evaluate the output of the edge detector for each row of pixels in I using corresponding rows of Ix and Iy as inputs.

```
Ieval = zeros(size(I));
for ii = 1:size(I,1)
Ieval(ii,:) = evalfis(edgeFIS,[(Ix(ii,:));(Iy(ii,:))]');
end
```

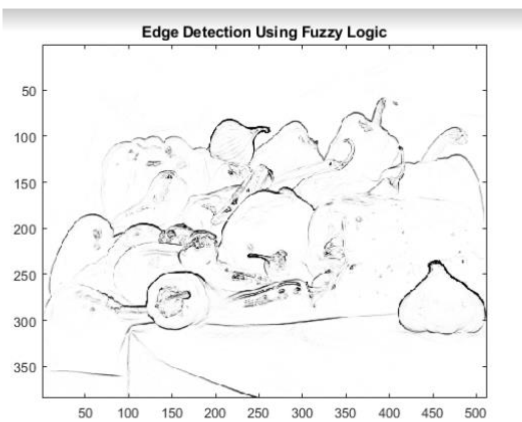
Plot Results

Plot the original grayscale image.

```
figure
image(I,'CDataMapping','scaled')
colormap('gray')
title('Original Grayscale Image')
```



Plot the detected edges.
figure
image(Ieval,'CDataMapping','scaled')
colormap('gray')
title('Edge Detection Using Fuzzy Logic')



RESULT:

Thus the python code is implement successfully and the output is verified.

EX.NO :10

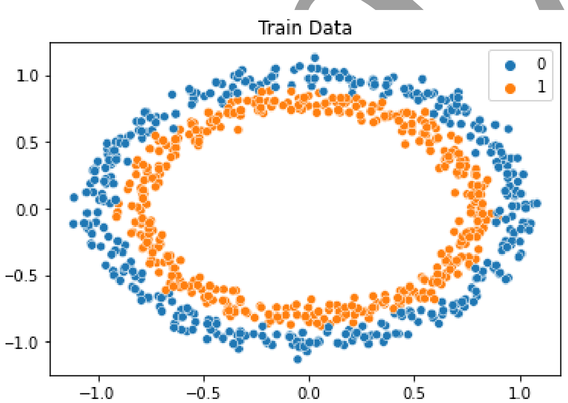
IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON - CLASSIFICATION

AIM :

To implementing artificial neural networks for an application in classification using python.

Source Code :

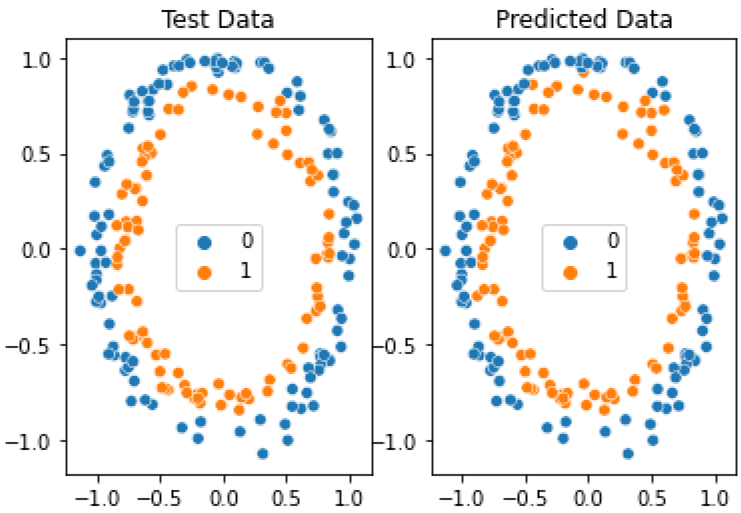
```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_circles
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

X, y = make_circles(n_samples=1000, noise=0.05)

ns.scatterplot(X_train[:,0], X_train[:,1], hue=y_train)
plt.title("Train Data")
plt.show()
```




```
clf = MLPClassifier(max_iter=1000)
clf.fit(X_train, y_train)
print(f"R2 Score for Training Data = {clf.score(X_train, y_train)}")

print(f"R2 Score for Test Data = {clf.score(X_test, y_test)}")

y_pred = clf.predict(X_test)
```

```
fig, ax =plt.subplots(1,2)
sns.scatterplot(X_test[:,0], X_test[:,1], hue=y_pred, ax=ax[0])
ax[1].title.set_text("Predicted Data")
sns.scatterplot(X_test[:,0], X_test[:,1], hue=y_test, ax=ax[1])
ax[0].title.set_text("Test Data")
plt.show()
```

OUTPUT :



RESULT :

Thus the python code is implement successfully and the output is verified.

EX.NO :11

IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON - REGRESSION

AIM :

To implementing artificial neural networks for an application in Regression using python.

SOURCE CODE :

```
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline


X, y = make_regression(n_samples=1000, noise=0.05, n_features=100)

X.shape, y.shape // ((1000, 100), (1000,))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True, random_state=42)

clf = MLPRegressor(max_iter=1000)
clf.fit(X_train, y_train)
print(f"R2 Score for Training Data = {clf.score(X_train, y_train)}")

print(f"R2 Score for Test Data = {clf.score(X_test, y_test)}")
```

OUTPUT :

R2 Score for Test Data = 0.9686558466621529

RESULT :

Thus the python code is implement successfully and the output is verified.

EX.NO :12

DECISION TREE CLASSIFICATION

AIM:
To classify the Social Network dataset using Decision tree analysis

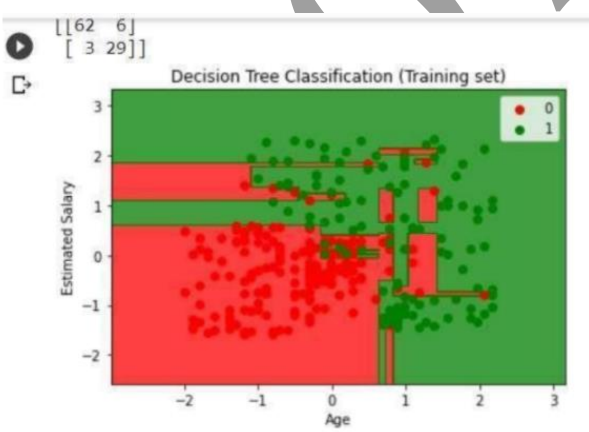
Source Code :

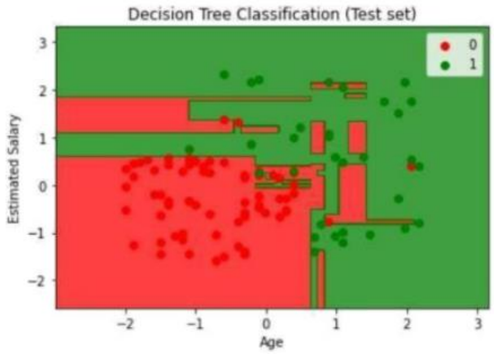
```
# Importing
the libraries import
numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Social_Network_Ads.csv')
X = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, -1].values

# Splitting the dataset into the Training set
and Test set from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Training the Decision Tree Classification model
on the Training set from sklearn.tree import
DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)

classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)
```

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() +
1, step =
0.01), np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('red',
'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j,
1],
c = ListedColormap(('red', 'green'))(i), label =
j) plt.title('Decision Tree Classification
(Training set)') plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() +
1, step =
0.01), np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('red',
'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Decision Tree Classification (Test set)')
plt.xlabel('Age')
```

```
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```


OUTPUT :





RESULT :

Thus the python code is implement successfully and the output is verified.

EX.NO : 13

IMPLEMENTATION OF DECISION TREE CLASSIFICATION TECHNIQUES

AIM:

To implement a decision tree classification technique for gender classification using python.

EXPLANATION:

- Import tree from sklearn.
- Call the function DecisionTreeClassifier() from tree
- Assign values for X and Y.
- Call the function predict for Predicting on the basis of given random values for each given feature.
- Display the output.

SOURCE CODE:

```
from sklearn import tree
#Using DecisionTree classifier for prediction
clf = tree.DecisionTreeClassifier()

#Here the array contains three values which are height,weight and shoe size
X = [[181, 80, 91], [182, 90, 92], [183, 100, 92], [184, 200, 93], [185, 300, 94], [186,
400, 95], [187, 500, 96], [189, 600, 97], [190, 700, 98], [191, 800, 99], [192, 900, 100],
[193, 1000, 101]]
Y = ['male', 'male', 'female', 'male' , 'female', 'male', 'female' , 'male' , 'female', 'male' ,
'female' , 'male' ]
clf = clf.fit(X, Y)

#Predicting on basis of given random values for each given feature
predictionf = clf.predict([[181, 80, 91]])
predictionm = clf.predict([[183, 100, 92]])

#Printing final prediction
print(predictionf)
print(predictionm)
```

OUTPUT:

['male']
['female']

RESULT:

Thus the python code is implement successfully and the output is verified.

EX NO :14

VIMPLEMENTATION OF CLUSTERING TECHNIQUES
K - MEANS

AIM:

To implement a K - Means clustering technique using python language.

EXPLANATION:

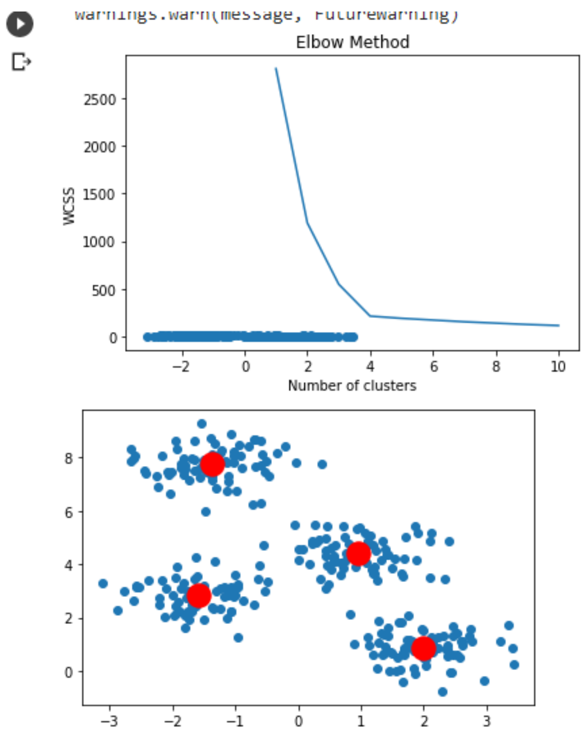
- Import KMeans from sklearn.cluster
- Assign X and Y.
- Call the function KMeans().
- Perform scatter operation and display the output.

SOURCE CODE:

```
Import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
plt.scatter(X[:,0], X[:,1])
wcss = []
for i in range(1, 11):
        kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,
random_state=0)
        kmeans.fit(X)
        wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
kmeans = KMeans(n_clusters=4, init='k-means++', max_iter=300, n_init=10,
random_state=0)
pred_y = kmeans.fit_predict(X)
```

```
plt.scatter(X[:,0], X[:,1])
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='red')
plt.show()
```

OUTPUT :



RESULT:

Thus the python code is implement successfully and the output is verified.