SRI SAI RAM ENGINEERING COLLEGE

SAI LEO NAGAR, WEST TAMBARAM, CHENNAI-44

(An Autonomous Institution)

NAME :_____

REGISTER NUMBER :_____

20CBPL603 – ARTIFICIAL INTELLIGENCE LABORATORY

(III YEAR/ VI SEM)

(BATCH: 2022 – 2026)

B. TECH COMPUTER SCIENCE AND BUSINESS SYSTEMS

ACADEMIC YEAR: 2024 – 2025

SAI RAM ENGINEERING COLLEGE

An Autonomous Institution | Affiliated to Anna University & Approved by AICTE, New Delhi
Accredited by NBA and NAAC "A+" | BIS/EOMS ISO 21001 : 2018 Certified and NIRF ranked institution
Sai Leo Nagar, West Tambaram, Chennai - 600 044. www.sairam.edu.in

Certificate

Register No: ……

Certified that this is the Bonafide Record of work done by Mr./Ms._____ in the B. TECH degree Course COMPUTER SCIENCE AND BUSINESS SYSTEMS in the 20CBPL603 – ARTIFICIAL INTELLIGENCE Laboratory during the academic year 2024-2025.

Station: Chennai – 600044

Date :

STAFF IN-CHARGE **HEAD OF THE DEPARTMENT**

Submitted for End Semester Practical Examination held on_____ at Sri Sai Ram Engineering College, Chennai – 600044.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

Lab Requirements

1. Python
2. Along with the python we have install and import the following packages and libraries
    a. numpy
    b. deepcopy
    c. time
    d. sys
    e. re
    f. intertools
    g. matplotlib
    h. pandas
    i. xrange
3. Prolog

Prerequisite knowledge

- Problem solving and logics skills
- Python programming skills
- Logical thinking
- Gaming experience

INDEX

S.NO	DATE	NAME OF THE EXPERIMENT	PAGE NO	SIGN
1.		8 QUEENS PROBLEM		
2.		DEPTH FIRST SEARCH		
3.		IMPLEMENT MINIMAX ALGORITHM		
4.		IMPLEMENT A* ALGORITHM		
5.		IMPLEMENTATION OF UNIFICATION AND RESOLUTION ALGORITHM		
6.		IMPLEMENTATION OF BACKWARD CHAINING		
7.		IMPLEMENTATION OF BLOCKS WORLD PROGRAM		
8.		IMPLEMENTATION OF SVM FOR AN APPLICATION USING PYTHON		
9.		IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON		
10.		IMPLEMENTATION OF DECISION TREE		
11.		IMPLEMENTATION OF KMEAN ALGORITHM		

EXP NO: 1
DATE: **8 QUEENS PROBLEM**

AIM:

To write a program to solve 8 queens problem.

ALGORITHM:

Step 1: Create a function backtrack that simply places the queens on corresponding rows and columns and marks them visited.

Step 2: The working of backtrack is as follows:

a) If the current row is equal to 8, then a solution has been found. Therefore, add this to the answer.

b) Traverse through the columns of the current row. A teach column, try to place the queen in (row,col):

(i) Calculate the diagonal and anti-diagonal which the current square belongs to. If it is unvisited, place the queen in the (row, col).

(ii) Skip this column, if the queen cannot be visited.

Step 3: If the queen has been placed successfully, call the backtrack function of row+1.

Step 4: Since, all paths have now been explored; clear the values of the queens placed so far and the visiting arrays,so that next distinct solution can be found.

PROGRAM:

```
N = 8 # (size of the chessboard)
def solveNQueens(board, col):
        if col == N:
            print(board)
            return True
        for i in range(N):
            if isSafe(board, i, col):
                board[i][col] = 1
                if solveNQueens(board, col + 1):
                    return True
                board[i][col] = 0
        return False
    def isSafe(board, row, col):
        for x in range(col):
            if board[row][x] == 1:
                return False
        for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[x][y] == 1:
                return False
        for x, y in zip(range(row, N, 1), range(col, -1, -1)):
            if board[x][y] == 1:
                return False
        return True
     board = [[0 for x in range(N)] for y in range(N)]
    if not solveNQueens(board, 0):
    print("No solution found")
```

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 program to solve the 8 queens problem was executed successfully

EXP NO: 2
DATE: **DEPTH FIRST SEARCH**

AIM:

To solve a problem using depth first search.

ALGORITHM:

Step 1: Create a word in this game-connect adjacent vertices and thus creating a path on the graph.
Step 2: Create a set of all words in English alphabet and a reasonable length of a word (let's assume that words are not longer than N chars)
Step 3: Start iterating from the vertices and from each vertex invoke DFS that run still the length N. for each of the words (that are bigger than 3 in the rules) check If it is in the dictionary and if it is save them somewhere.

PROGRAM:

```
from collections import defaultdict

        # This class represents a directed graph using
        # adjacency list representation
          class Graph:
        # Constructor
          def __init__(self):
       # Default dictionary to store graph
          self.graph = defaultdict(list)
       # Function to add an edge to graph
          def addEdge(self, u, v):
          self.graph[u].append(v)
       # A function used by DFS
           def DFSUtil(self, v, visited):

       # Mark the current node as visited
      # and print it
        visited.add(v)
        print(v, end=' ')

     # Recur for all the vertices
     # adjacent to this vertex
    for neighbour in self.graph[v]:
    if neighbour not in visited:
    self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
      def DFS(self, v):

    # Create a set to store visited vertices
visited = set()
```

```
        # Call the recursive helper function
        # to print DFS traversal
           self.DFSUtil(v, visited)


      # Driver's code
           if __name__ == "__main__":
          g = Graph()
          g.addEdge(0, 1)
          g.addEdge(0, 2)
          g.addEdge(2, 1)
          g.addEdge(2, 0)
          g.addEdge(1, 4)
          g.addEdge(1, 5)
          g.addEdge(4, 6)

          print("Following is Depth First Traversal (starting from vertex 0)")

          # Function call
          g.DFS(0)
```

OUTPUT:

```
Following is Depth First Traversal (starting from vertex 0)
0 1 4 6 5 2
```

RESULT:

Thus, the program to find the valid words from the given boggle was executed using the depth first search.

EXP.NO:3

DATE: **IMPLEMENTATION OF MINIMAX ALGORITHM**

AIM:

To execute a program to implement minimax algorithm.

ALGORITHM:

Step 1: This new function called findBestMove() evaluates all the available moves using minimax() and then returns the best move the maximizer can make.

Step 2: To check whether or not the current move is better than the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally.

Step 3:The code for the maximizer and minimizer in the minimax() function is similar to findBestMove(), the only difference is, instead of returning a move, it will return a value.

Step 4: To check whether the game is over and to make sure there are no moves left we use isMovesLeft() function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively.

Step 5: Assume that there are 2 possible ways for X to win the game from a give board state.

- Move A : X can win in 2 move
- Move B : X can win in 4 moves

Step 6: Our evaluation function will return a value of +10 for both moves A and B.
Even though the move A is better because it ensures a faster victory, our AI may choose B sometimes.

Step 7: To overc ome this problem, we subtract the depth value from the evaluated score. This means that in case of a victory it will choose the victory which takes least number of moves and in case of a loss it will try to prolong the game and play as many moves as possible. So the new evaluated value will be

- Move A will have a value of +10 – 2 = 8
- Move B will have a value of +10 – 4 = 6

Step 8: Now since move A has a higher score compared to move B our AI will choose move A over move B.

Step 9: The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value as the minimizer always tries to get, as negative a value as possible.

PROGRAM:

```
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False
```

```
# This is the evaluation function as discussed
# in the previous article ( http://goo.gl/sJgv68 )
def evaluate(b) :

    # Checking for Rows for X or O victory.
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            elif (b[row][0] == opponent) :
                return -10

    # Checking for Columns for X or O victory.
    for col in range(3) :

        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

            if (b[0][col] == player) :
                return 10
            elif (b[0][col] == opponent) :
                return -10
     # Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

        if (b[0][0] == player) :
            return 10
        elif (b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

        if (b[0][2] == player) :
            return 10
        elif (b[0][2] == opponent) :
            return -10
    # Else if none of them have won then return 0
    return 0

# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax) :
    score = evaluate(board)

    # If Maximizer has won the game return his/her
    # evaluated score
    if (score == 10) :
        return score
```

```
# If Minimizer has won the game return his/her
# evaluated score
if (score == -10) :
    return score

# If there are no more moves and no winner then
# it is a tie
if (isMovesLeft(board) == False) :
    return 0

# If this maximizer's move
if (isMax) :
    best = -1000

    # Traverse all cells
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j]=='_') :

                # Make the move
                board[i][j] = player

                # Call minimax recursively and choose
                # the maximum value
                best = max( best, minimax(board,
                                  depth + 1,
                                  not isMax) )

                # Undo the move
                board[i][j] = '_'
    return best
# If this minimizer's move
else :
    best = 1000

    # Traverse all cells
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j] == '_') :


                # Make the move
                board[i][j] = opponent

                # Call minimax recursively and choose
                # the minimum value
                best = min(best, minimax(board, depth + 1, not isMax))
```

```
                # Undo the move
                board[i][j] = '_'
        return best

# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
    for  i  in  range(3)  :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j] == '_') :

                # Make the move
                board[i][j] = player

                # compute evaluation function for this
                # move.
                moveVal = minimax(board, 0, False)

                # Undo the move
                board[i][j] = '_'

                # If the value of the current move is
                # more than the best value, then update
                # best
                if (moveVal > bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal

    print("The value of the best Move is :", bestVal)

    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]

 bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

OUTPUT:

The value of the best Move is : 10

The Optimal Move is :

ROW: 2 COL: 2

RESULT:

Thus, the program to find the next optimal move of the player was executed using the MINIMAX algorithm

EXP.NO:4
DATE: **IMPLEMENT A* ALGORITHM**

AIM:

To execute a program to implement A* algorithm

ALGORITHM:

Step 1: Initialize: Initialize the open list with the start node and the closed list as empty. The open list contains nodes to be evaluated, while the closed list contains nodes that have already been evaluated.

Step 2: Select Node: Select the node with the lowest f value (f = g + h) from the open list and move it to the closed list.

Step 3: Check Goal: If the selected node is the goal node, reconstruct the path and return it as the solution.

Step 4: Expand Node: For each neighbour of the selected node, calculate its g, h, and f values. If the neighbour is not in the open list, add it. If it is already in the open list with a higher f value, update its f value and parent node.

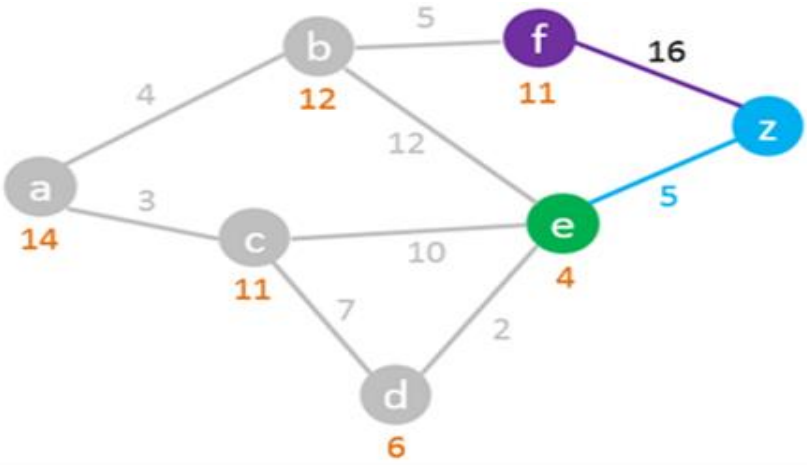
Step 5: Repeat: Repeat steps 2 to 4 until the open list is empty or the goal node is found.

PROGRAM:

```
graph = {
'a': {'b': 4, 'c': 3},
'b': {'f': 5, 'e': 12},
'c': {'e': 10, 'd': 7},
'd': {'e': 2},
'e': {'z': 5},
'f': {'z': 16},
}
# define the heuristic function
heuristic = {
'a': 14,
'b': 12,
'c': 11,
'd': 6,
'e': 4,
'f': 11,
'z': 0,
  }
from queue import PriorityQueue
def a_star_search(graph, start, goal, heuristic):
frontier = PriorityQueue()
frontier.put((0, start))
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0
   while not frontier.empty():
   current = frontier.get()[1]

   if current == goal:
      path = []
      while current is not None:
```

```
            path.append(current)
            current = came_from[current]
        path.reverse()
        return path
    for neighbor in graph[current]:
        new_cost = cost_so_far[current] + graph[current][neighbor]
        if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
            cost_so_far[neighbor] = new_cost
            priority = new_cost + heuristic[neighbor]
            frontier.put((priority, neighbor))
            came_from[neighbor] = current
 return None

 # Example usage
start = 'a'

goal = 'z'

 path = a_star_search(graph, start, goal, heuristic)
if path:
 print("Shortest path from", start, "to", goal, ":", path)
else:
 print("No path found from", start, "to", goal)
```

SAMPLE GRAPH:



OUTPUT:

Shortest path from a to z: ['a', 'c', 'd', 'e', 'z']

RESULT:

Thus, the program to implement the A* algorithm was created and executed successfully.

INTRODUCTION TO PROLOG

OBJECTIVE:

Study of Prolog.

PROLOG-PROGRAMMING IN LOGIC

PROLOG stands for Programming, In Logic — an idea that emerged in the early 1970's to use logic as programm language. The early developers of this idea included Rober Kowaiski at Edinburgh (on the theoretical side), Marrten van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles implementation. David D.H. Warren's efficient implementation at Edinburgh in the mid -1970's greatly contributed to the popularity of PROLOG. PROLOG is a programming language centred around a small set of basic mechanisms, Including pattern matching, tree based data structuring and automatic backtracking. This Small set constitutes a surprisingly powerful and flexible programming framework. PROLOG is especially well suited for problems that involve objects- in particular,structuredobjects- and relations between them.

SYMBOLIC LANGUAGE

PROLOG is a programming language for symbolic, non-numeric computation. It is especially well-suited for solving problems that involve objects and relations between objects. For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y. and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language forArtJIcia1 LanguageA1,) and non- numerical programming.

There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code, when the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

FACTS, RULES AND QUERIES Programming in PROLOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program Responses to user queries are determined through a form of inference control known as resolution.

FOR EXAMPLE:

a) FACTS:

Some facts about family relationships could be written as:sister(sue,bill)

parent(ann.sam)male(jo) female(riya)

b) RULES:

To represent the general rule for grandfather, we write:grandfather(X2)

parent(X,Y)

parent(Y,Z)male(X)

Queries:

Given a database of facts and rules such as above, we may make queries by typing after a query symbol '?' statement such as:

?-parent(X,sam)

?grandfather(X,Y)X=jo, Y=sam

PROLOG IN DISGINING EXPERT SYSTEMS

An expert system is a set of programs that manipulates encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system's knowledge is obtained from expert sources such as texts, journal articles. databases etc and encoded in a form suitable for the system to use in its inference or reasoning process.Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into knowledge base, then tested, and refined continually throughout the life of the system PROLOG serves as a powerful language in designing expert systems because of its following features.

- Use of knowledge rather than data
- Modification of the knowledge base without recompilation of the control programs. Capable of explaining conclusion.
- Symbolic computations resembling manipulations of natural language. Reason with meta-knowledge.

META PROGRAMMING:

A meta-program is a program that takes other programs as data. Interpreters and compilers are examples of mela-programs. Meta-interpreter is a particular kind of meta-program: an interpreter for a language written in that language. So, a PROLOG interpreter is an interpreter for PROLOG, itself written in PROLOG. Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta- programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in Implementing new ideas quickly.

OUTCOME:

Students will get the basic idea of how to program in prolog and its working environment.

EXP.NO:5
DATE:

IMPLEMENTATION OF UNIFICATION AND RESOLUTION ALGORITHM

AIM:

To Unify the process of finding a substitute that makes two separate logical atomic expressions identical. The substitution process is necessary for unification.

ALGORITHM:

Step. 1: If Ψ1 or Ψ2 is a variable or constant, then:

- If Ψ1 or Ψ2 are identical, then return NIL.
- Else if Ψ1 is a variable,

a) Else if Ψ1 is a variable,

- then if Ψ1 occurs in Ψ2, then return FAILURE
- Else return {(Ψ2/ Ψ1)}.

b) Else if Ψ2 is a variable,

- If Ψ2 occurs in Ψ1 then return FAILURE,
- Else return {(Ψ1 / Ψ2)}.

c) Else return FAILURE.

Step 2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.
Step 3: IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.
Step 4: Set Substitution set(SUBST) to NIL.
Step 5: For i=1 to the number of elements in Ψ1.

- Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.
- If S = failure then returns Failure
- If S ≠ NIL then do,
- Apply S to the remainder of both L1 and L2.
- SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

EXAMPLES:

```
?- a = a.                  ** Two identical atoms unify **
 yes
?- a = b.                  ** Atoms don't unify if they aren't identical **
 no
?- X = a.                  ** Unification instantiates a variable to an atom **
 X=a
 yes
?- X = Y.                  ** Unification binds two differently named variables **
 X=_125451                 ** to a single, unique variable name **
 Y=_125451
 yes
?- foo(a,b) = foo(a,b).    ** Two identical complex terms unify **
 yes
?- foo(a,b) = foo(X,Y).    ** Two complex terms unify if they are **
 X=a                       ** of the same arity, have the same principal**
 Y=b                       ** functor and their arguments unify **
 yes
```

```
?- foo(a,Y) = foo(X,b).      ** Instantiation of variables may occur **
 Y=b                         ** in either of the terms to be unified **
 X=a
 yes
?- foo(a,b) = foo(X,X).      ** In this case there is no unification **
 no                          ** because foo(X,X)  must have the same **
                             ** 1st and 2nd arguments **
 ?- 2*3+4 = X+Y.             ** The term 2*3+4 has principal functor + **
 X=2*3                       ** and therefore unifies with X+Y with X instantiated**
 Y=4                         ** to 2*3 and Y instantiated to 4 **
yes
 ?- [a,b,c] = [X,Y,Z].       ** Lists unify just like other terms **
 X=a
 Y=b
 Z=c
yes
?- [a,b,c] = [X|Y].          ** Unification using the '|' symbol  can be used **
 X=a                         ** to find the head element, X, and tail list, Y, **
 Y=[b,c]                     ** of a list **
yes
 ?- [a,b,c] = [X,Y|Z].       ** Unification on lists doesn't have to be **
 X=a                         ** restricted to finding the first head element **
 Y=b                         ** In this case we find the 1st and 2nd elements **
 Z=[c]                       ** (X and Y) and then the tail list (Z) **
yes
?- [a,b,c] = [X,Y,Z|T].      ** This is a similar example but here **
 X=a                         ** the first 3 elements are unified with **
 Y=b                         ** variables X, Y and Z, leaving the **
 Z=c                         ** tail, T, as an empty list [] **
 T=[]
yes
?- [a,b,c] = [a|[b|[c|[]]]].  ** Prolog is quite happy to unify these **
yes                           ** because they are just notational **
                              ** variants of the same Prolog term **
```

RESOLUTION:

- Resolution is a theorem proving technique that proofs by contradictions. It's a single inference rule which cane efficiently operate on conjunctive normal form or clausal form.
- It is used in both propositional as well as first order predicate logic in different ways.
- In this resolution method, we use proof of refutation technique to prove the given statement.

STEPS IN RESOLUTION:

Step 1: conversion of facts into FOL.

Step 2: convert FOL statement into CNF

Step 3: Negate the statement which need to prove

Step 4: If a contradiction (the empty clause) is derived, then the original sentence is proven.

EXAMPLE:

1. The humidity is high or the sky is cloudy
2. If the sky is cloudy, then it will rain
3. If then humidity is high, then it is hot
4. It is not hot

Goal: It will rain
Step 1: conversion to first order logic
A: the humidity is high
B: sky is cloudy
The humidity is high or the sky is cloudy

A v B

C: it will rain
If the sky is cloudy, then it will rain
B->C
D:it is hot
If the humidity is high, then it is hot
A->D
~D it is not hot

RESULT:

Thus, the program was executed to implement the unification and resolution.

EXP.NO:6
DATE: **IMPLEMENTATION OF BACKWARD CHAINING**

AIM:

To execute a program to implement backward chaining.

ALGORITHM:

1. Start the Program.
2. TO-DO return false if certain key is not there in kb.
3. Return false if proper combination does not exist.
4. Return false if query arguments number do not match with consequents.
5. You should not replace predicate values same as of variable occurrence.
6. Writing files to output file.
7. Stop the program.

PROGRAM:

Example:

```
% Knowledge Base
parent(john, mary).
parent(mary, alice).
parent(alice, sophie).

% Backward chaining rule to determine ancestry
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

OUTPUT:

```
% ?- ancestor(john, sophie).
True
```

RESULT:

Thus, the program was executed to implement the backward chaining.

EXP.NO:7
DATE: **IMPLEMENTATION OF BLOCKS WORLD PROGRAM**

AIM:

To implement the blocks world program.

ALGORITHM:

Step 1: Define the Initial State: Define the initial configuration of blocks, including the number of blocks and their positions on the stacks.
Step 2: Define the Goal State: Define the target configuration of blocks, including the number of blocks and their positions on the stacks.
Step 3: Generate Next States: Generate all possible next configurations by moving one block at a time from one stack to another.
Step 4: Check for Goal State: Check if the current configuration is the goal state. If yes, return the solution.
Step 5: Explore Next States: Explore each of the next configurations, generating all possible next configurations, and so on.
Step 6: Repeat the Process: Repeat steps 3-5 until the goal state is reached or all possible configurations have been explored.

PROGRAM:

```
from collections import deque
class BlockWorld:
 def __init__(self, initial_state, goal_state):
    self.initial_state = initial_state
    self.goal_state = goal_state
 def print_state(self, state):
    for i, stack in enumerate(state):
       print(f"Stack {i+1}: {stack}")
 def is_goal_state(self, state):
    return state == self.goal_state
 def get_next_states(self, state):
    next_states = []
    for i, stack in enumerate(state):
       if stack:
          block = stack[-1]
          for j, other_stack in enumerate(state):
             if i != j:
                new_state = [s[:] for s in state]
                new_state[i].pop()
                new_state[j].append(block)
                next_states.append(new_state)
    return next_states


 def solve(self):
    queue = deque([(self.initial_state, [self.initial_state])])
    visited = set()
    visited.add(tuple(tuple(stack) for stack in self.initial_state))
    while queue:
```

```
        state, path = queue.popleft()
        if self.is_goal_state(state):
            return path

        for next_state in self.get_next_states(state):
            next_state_tuple = tuple(tuple(stack) for stack in next_state)
            if next_state_tuple not in visited:
                visited.add(next_state_tuple)
                queue.append((next_state, path + [next_state]))
    return None
# Example usage
initial_state = [["A"], ["B", "C"], []]
goal_state = [["C", "B", "A"], [], []]
block_world = BlockWorld(initial_state, goal_state)
print("Initial State:")
block_world.print_state(initial_state)
print("\nGoal State:")
block_world.print_state(goal_state)

solution = block_world.solve()
if solution:
print("\nSolution:")
for i, state in enumerate(solution):
    print(f"Step {i+1}:")
    block_world.print_state(state)
    print()
else:
print("No solution found")
```

OUTPUT:

```
Initial State:
Stack 1: ['A']
Stack 2: ['B', 'C']
Stack 3: []

Goal State:
Stack 1: ['C', 'B', 'A']
Stack 2: []
Stack 3: []

Solution:
Step 1:
Stack 1: ['A']
Stack 2: ['B', 'C']
Stack 3: []

Step 2:
Stack 1: []
Stack 2: ['B', 'C']
Stack 3: ['A']

Step 3:
Stack 1: ['C']
Stack 2: ['B']
Stack 3: ['A']

Step 4:
Stack 1: ['C', 'B']
Stack 2: []
Stack 3: ['A']

Step 5:
Stack 1: ['C', 'B', 'A']
Stack 2: []
Stack 3: []
```

RESULT:

Thus, the program to implement the blocks world program was executed successfully.

EXP.NO:8
DATE:

IMPLEMENTATION OF SVM FOR AN APPLICATION USING PYTHON

AIM:

To execute a program to implement Support Vector Machine using python.

ALGORITHM:

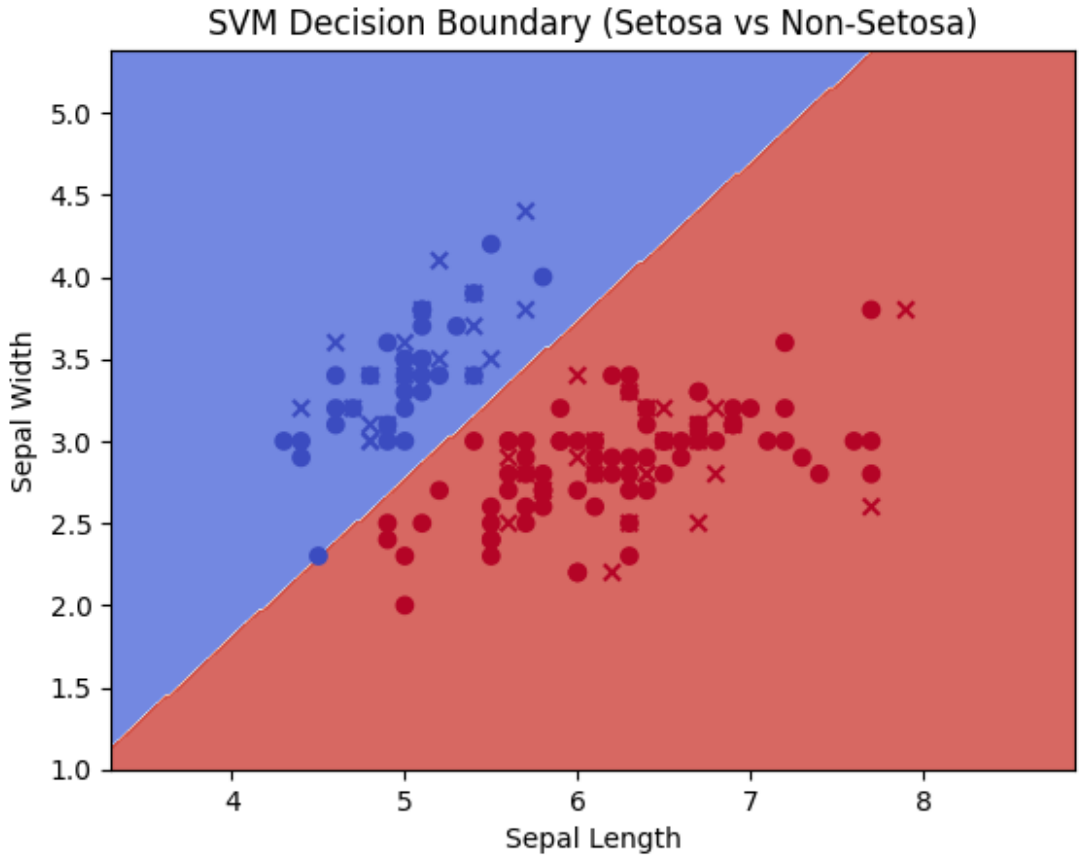
Step 1: Load the Iris dataset.
Step 2: Select the first two features (sepal length and sepal width).
Step 3: Convert labels to a binary classification (Setosa vs Non-Setosa).
Step 4: Split the data into training and testing sets.
Step 5: Initialize the SVM classifier with a linear kernel.
Step 6: Train the model on the training data.
Step 7: Predict the labels for the test data.
Step 8: Calculate and print the accuracy of the model.
Step 9: Create a meshgrid for plotting the decision boundary.
Step 10: Predict the class for each point in the meshgrid.
Step 11: Plot the decision boundary using contourf().
Step 12: Scatter plots the training data with circles.
Step 13: Scatter plot the testing data with crosses.
Step 14: Display the plot with labels and title.

PROGRAM:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
# Load and prepare data (Iris dataset)
data = datasets.load_iris()
X = data.data[:, :2]  # Only the first two features
y = np.where(data.target == 0, 0, 1)  # Setosa vs Non-Setosa
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Train SVM model
svm = SVC(kernel='linear')
svm.fit(X_train, y_train)
# Predict and print accuracy
y_pred = svm.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")
# Plot decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))
Z = svm.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
```

```
# Display plot
plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.coolwarm, marker='o')
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=plt.cm.coolwarm, marker='x')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('SVM Decision Boundary (Setosa vs Non-Setosa)')
plt.show()
```

OUTPUT:



RESULT:

Thus, the program to execute the application of social network ads using support vector machine was executed successfully.

EXP.NO:9
DATE:

IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON

AIM:

To execute a program to implement artificial neural networks for an application using python.

ALGORITHM:

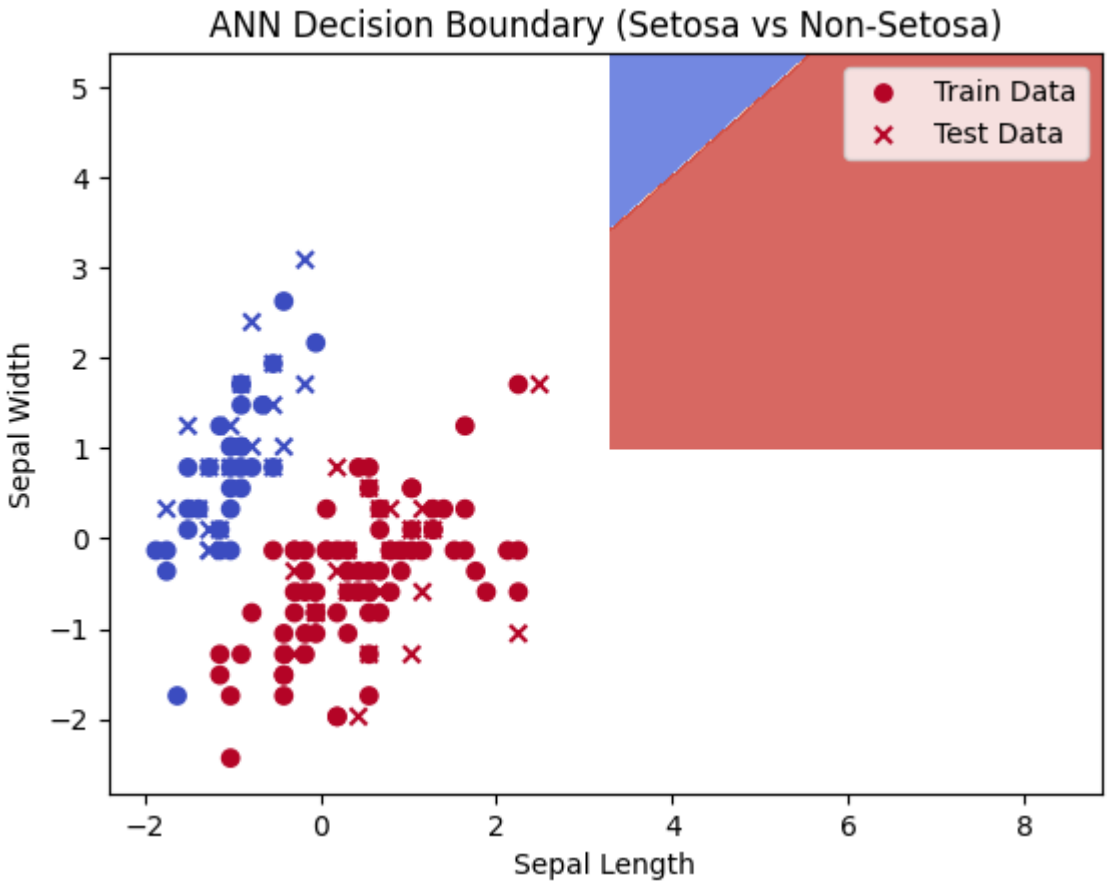
Step 1: Load the Iris dataset, select the first two features (sepal length and sepal width), and convert the target labels into a binary classification ("Setosa vs Non-Setosa").
Step 2: Standardize the features using StandardScaler to ensure proper scaling for neural network training.
Step 3: Split the dataset into training and testing sets.
Step 4: Build an ANN model with a sequential structure, adding a hidden layer with 8 neurons and a ReLU activation function, followed by an output layer with a sigmoid activation for binary classification.
Step 5: Compile the model using binary cross-entropy loss and Adam optimizer.
Step 6: Train the model on the training data for 50 epochs with a batch size of 10.
Step 7: Evaluate the model on the test data, predicting class labels and calculating accuracy.
Step 8: Plot the decision boundary by creating a meshgrid and plotting the predicted class regions along with the training and testing data points.

PROGRAM:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load and prepare data (Iris dataset)
data = datasets.load_iris()
X = data.data[:, :2]  # Only the first two features
y = np.where(data.target == 0, 0, 1)  # Setosa vs Non-Setosa

# Standardize features (important for neural networks)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)
```

```
# Build the ANN model
model = Sequential()
model.add(Dense(8, input_dim=2, activation='relu'))  # First hidden layer with 8 neurons
model.add(Dense(1, activation='sigmoid'))  # Output layer with 1 neuron (binary classification)

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=10, verbose=1)

# Evaluate the model on the test data
y_pred = (model.predict(X_test) > 0.5).astype(int)  # Convert probabilities to 0 or 1

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Plotting decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))
Z = (model.predict(np.c_[xx.ravel(), yy.ravel()]) > 0.5).astype(int)
Z = Z.reshape(xx.shape)


# Display plot
plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.coolwarm, marker='o', label='Train Data')
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=plt.cm.coolwarm, marker='x', label='Test Data')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('ANN Decision Boundary (Setosa vs Non-Setosa)')
plt.legend()
plt.show()
```

OUTPUT:



RESULT:

Thus, the program to implement artificial neural networks for an application using python was executed successfully.

EXP.NO:10

DATE:

IMPLEMENTATION OF DECISION TREE

AIM:

To execute a program to implement decision tree.

ALGORITHM:

Step 1: Import Libraries: Import numpy, matplotlib, and scikit-learn for data handling, modeling, and visualization.

Step 2: Load Dataset: Use load_iris() from sklearn.datasets to load the Iris dataset.

Step 3: Split Data: Use train_test_split() to divide the data into 70% training and 30% testing sets.

Step 4: Create the Decision Tree: Initialize the DecisionTreeClassifier with random_state=42 for reproducibility.

Step 5: Train the Model: Fit the model to the training data using fit() on X_train and y_train.

Step 6: Make Predictions: Predict outcomes on the test set with predict() and evaluate accuracy using accuracy_score().

Step 7: Visualize the Decision Tree: Visualize the tree structure using plot_tree() to understand how decisions are made.

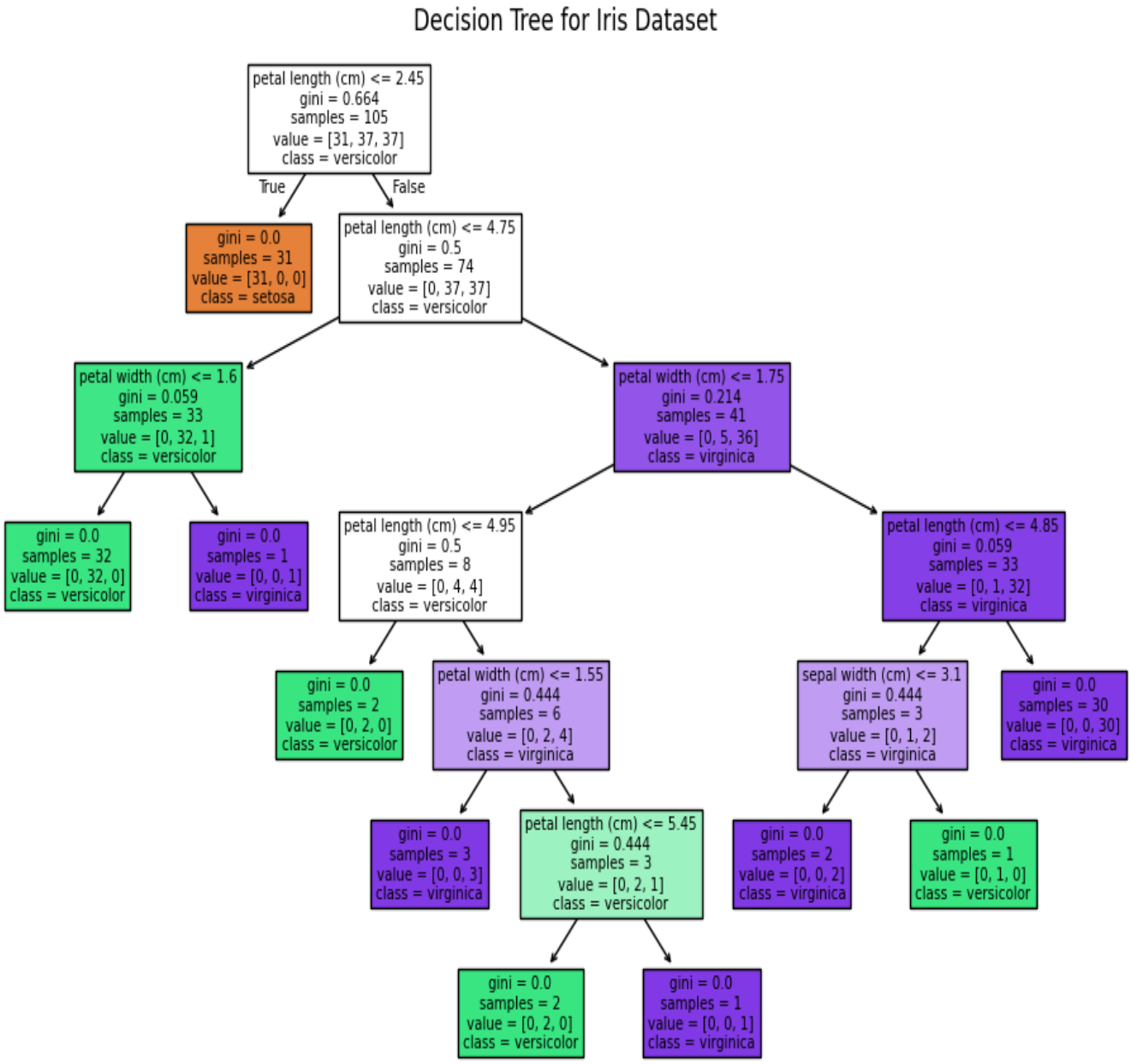
PROGRAM:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn.tree import plot_tree

# Load the Iris dataset
data = load_iris()
X = data.data  # Features (sepal length, sepal width, petal length, petal width)
y = data.target  # Target labels (species)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)

# Train the model
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = metrics.accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
# Visualize the decision tree
plt.figure(figsize=(12, 8))
plot_tree(clf, filled=True, feature_names=data.feature_names, class_names=data.target_names)
plt.title('Decision Tree for Iris Dataset')
plt.show()
```

OUTPUT:

Accuracy 100%



RESULT:

Thus, the program to implement decision tree was executed successfully.

EXP.NO:11

DATE:

IMPLEMENTATION OF K-MEAN ALGORITHM

AIM:

To execute a program to implement K- mean algorithm.

ALGORITHM:

Step 1: Initialize: Randomly select the first centroid, then use the k-means++ method to select the remaining centroids based on distance.
Step 2: Assign points: For each point, calculate its distance to each centroid and assign it to the nearest centroid.
Step 3: Update centroids: Recalculate the centroids by averaging the points assigned to each centroid.
Step 4: Repeat: Iterate steps 2 and 3 until the centroids do not change or the maximum iterations are Reached.
Step 4: Evaluate: For each point, determine the nearest centroid and return the centroid assignments.
Step 5: Visualize the clusters and the centroids on a scatter plot.

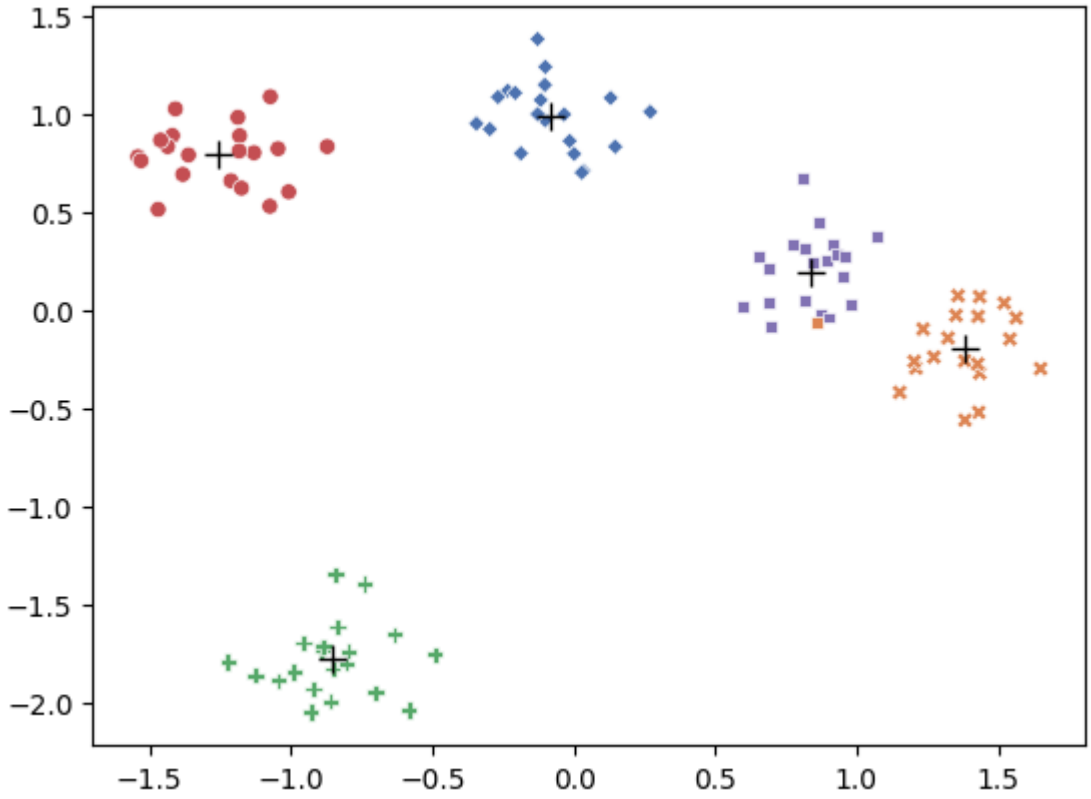
PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from numpy.random import uniform
from sklearn.datasets import make_blobs
import seaborn as sns
import random

    # Euclidean distance between a point and a dataset
    def euclidean(point, data):
        return np.sqrt(np.sum((point - data)**2, axis=1))

class KMeans:
    def __init__(self, n_clusters=8, max_iter=300):
        self.n_clusters = n_clusters
  self.max_iter = max_iter

def fit(self, X_train):
    # Initialize the centroids using the "k-means++" method
    self.centroids = [random.choice(X_train)]  # First centroid is a random point from the data
    for _ in range(self.n_clusters - 1):
        # Calculate distances from points to the centroids
        dists = np.sum([euclidean(centroid, X_train) for centroid in self.centroids], axis=0)
        # Normalize the distances
        dists /= np.sum(dists)
        # Choose the next centroid based on the distances (probabilistic selection)
        new_centroid_idx, = np.random.choice(range(len(X_train)), size=1, p=dists)
        self.centroids.append(X_train[new_centroid_idx])

    # Iterate and update centroids until convergence or max_iter is reached
    iteration = 0
    prev_centroids = None
    while np.not_equal(self.centroids, prev_centroids).any() and iteration < self.max_iter:
        # Sort each datapoint, assigning it to the nearest centroid
        sorted_points = [[] for _ in range(self.n_clusters)]
```

```
        for x in X_train:
            dists = euclidean(x, self.centroids)
            centroid_idx = np.argmin(dists)  # Find the nearest centroid
            sorted_points[centroid_idx].append(x)

        # Push current centroids to previous, reassign centroids as the mean of the points belonging to
 them
        prev_centroids = self.centroids
        self.centroids = [np.mean(cluster, axis=0) for cluster in sorted_points]

        # Catch any NaNs that occur if a centroid has no points assigned
        for i, centroid in enumerate(self.centroids):
            if np.isnan(centroid).any():
                self.centroids[i] = prev_centroids[i]

        iteration += 1

def evaluate(self, X):
    centroids = []
    centroid_idxs = []
    for x in X:
        dists = euclidean(x, self.centroids)
        centroid_idx = np.argmin(dists)  # Find the closest centroid
        centroids.append(self.centroids[centroid_idx])

        centroid_idxs.append(centroid_idx)
    return centroids, centroid_idxs

        # Create a dataset of 2D distributions
        centers = 5
        X_train, true_labels = make_blobs(n_samples=100, centers=centers, random_state=42)

        # Standardize features
        X_train = StandardScaler().fit_transform(X_train)

        # Fit centroids to dataset
        kmeans = KMeans(n_clusters=centers)
        kmeans.fit(X_train)

        # View results
        class_centers, classification = kmeans.evaluate(X_train)

        # Plot the results
        sns.scatterplot(x=[X[0] for X in X_train],
                        y=[X[1] for X in X_train],
                        hue=true_labels,
                        style=classification,
                        palette="deep",
                        legend=None)

        # Plot the centroids
        plt.plot([x for x, _ in kmeans.centroids],
```

```
        [y for _, y in kmeans.centroids],
        'k+', markersize=10)

    plt.show()
```

OUTPUT:



RESULT:

Thus, the program to implement K- mean algorithm was executed successfully.