INDEX

POAI OBSERVATION

NAME : HARSHINI AKSHAYA STD : SEC. : ROLL NO. :

S.No.	Date	Title	Page No.	Teacher's Sign/ Remarks
1.		N-Queens Problem		
2.		Depth First Search		
3.		Water Jug problem using DFS		
4.		A* Search		
5.		Minimax Algorithm		
6.		Prolog		
7		Introduction to prolog		
8.		Unification & Resolution		
9.		Fuzzy logic - Image processing		
10.		Implementing ANN for an application using Python		
11.		Implementing ANN for an appl using python regression		
12.		Decision tree classification		
13.		Implementation of decision tree classification		
14.		Implementation of clustering technique (K Means)		

completed

Exp No: 1

Date:

N- Queens Problem

AIM:

To solve the N-Queen problem where the goal is to place n-queens on a n*n chessboard such that no two queens attack each other.

Algorithm:

1) Create a n x n chessboard with all cells set to 0, representing no queens placed
2) Ensure no queen is in the same row, upper diagonal, or lower diagonal for a given position
3) Try placing a queen in each row of the current column. If it is safe using safe()
4) Move to the next colum if placing a queen works, else backtrack by removing queen
5) If queens are placed in all columns return success
6) Display the board
7) If no solution exists, print "solution does not exist".

Program:

```
def issafe(board, row, col, n):
    for i in range(col):
        if board[row][i] == 1:
            return false
    for i,j in zip(range(row,-1,-1),
                   range(col,-1,-1)):
        if board[i][j] == 1:
            return false
    for i,j in zip(range(row,n,1),
                   range(col,-1,-1)):
        if board[i][j] == 1:
            return false
    return true

def solveNQUtil(board, col, n):
    if col == n:
        return true
    for i in range(n):
        if isSafe(board, i, col, n):
            board[i][col] = 1
            if solveNQUtil(board, col+1, n)
                == true:
                return true
```

```
        board[i][col] = 0
    return false

def solveNQ(n):
    board = [[0]*n for i in range(n)]
    if solveNQUtil(board, 0, n) == False:
        print("solution does not exist")
        return false
    for i in board:
        print(i)
        return true
n = int(input("enter n value:"))
solveNQ(n)
```

OUTPUT:

```
Enter value: 5
[1, 0, 0, 0, 0]
[0, 0, 0, 1, 0]
[0, 1, 0, 0, 0]
[0, 0, 0, 0, 1]
[0, 0, 1, 0, 0]
```

Exp 9 Depth First Search

AIM:

Aim: To implement depth first search to traverse a graph and explore all vertices by visiting as far along each branch as possible before backtracking

Algorithm:

1) Start
2) Initialize an empty stack and a list to keep track of visited nodes
3) Push the starting node onto stack and mark visited
4) While the stack is not empty, repeat
5) to step 7
6) Pop the top node from the stack
6) Print or process the popped node
7) For each adjacent unvisited neighbour of the popped node
8) Mark the neighbour one visited

a) Push the unvisited neighbours onto the stack.

b) Repeat until all reachable nodes are visited

c) stop

Program:

```
def dfs(graph, start):
    stack = [start]
    visited = set()

    while stack:
        node = stack.pop(),
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
        for neighbour in graph[node]:
            if neighbor not in visited:
                stack.append(neighbour)


graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': []
    'E': ['F'],
    'F': []
}
```

print ("DFS Traversal starting from node 'A': ")
dfs (graph, 'A')

Output:

DFS Traversal starting from node 'A':

A C F B E D

Experiment 3

Water Jug problem using DFS

Aim:

To solve the water jug problem using DFS

Algorithm:

1) create a stack to store the states of the jugs
2) Initialise the stack with the initial state
3) While the stack is not empty, do the following
   ⇒ Pop a state from the stack
   ⇒ If the stack represents the desired quantity, stop and return the solution.
   ⇒ Generate all possible next states from the current state
   ⇒ Push the next state onto the stack
4) If the stack becomes empty and no solution is found, the problem is unsolvable.

Program:

```
def solveWaterJugProblem(capacity_jug1, capacity_jug2, desired_quantity):
    stack = []
    visited = set()
```

```
parent = { }
start_state = (0, 0)
stack.append(start_state)
visited.add(start_state)
parent[start_state] = None
while stack:
    current_state = stack.pop()
    if current_state[0] == desired_quantity
    or current_state[1] == desired_quantity:
        path = []
        while current_state is not None:
            path.append(current_state)
            current_state = parent[current_state]
        return path[::-1]
    next_states = generateNextStates(current_state
        capacity_jug1, capacity_jug2)
    for state in next_states:
        if state not in visited:
            visited.add(state)
            parent[state] = current_state
            stack.append(state)
return "No solution found"
```

```
def generateNextStates(state, capacity_jug1,
        capacity_jug2):
    next_states = []
    next_states.append((capacity_jug1, state[1]))
    next_states.append((state[0], capacity_jug2))
    next_states.append((0, state[1]))
    next_states.append((state[0], 0))
    pour_amount = min(state[0], capacity_jug2
                      - state[1])
    next_states.append((state[0] - pour_amount,
        state[1] + pour_amount))
    pour_amount = min(state[1], capacity_jug_state
                      [0])
    next_states.append((state[0] + pour_amount,
        state[1] - pour_amount))
    return next
solution = solveWaterJug(4, 3, 2)
print("solution: ", solution)
```

OUTPUT:

solution : [(0, 0), (0, 3), (3, 0), (3, 3), (4, 2)]

Experiment 4:

A* Search

AIM:

To find the shortest path from a start node to a goal node using A* search.

Algorithm:

1. Create open and closed sets. Start with the initial node
2. Add the start node to the open set with an initial cost of 0
3. Remove the node with lowest (f) value (cost + heuristic) from open
4. If the current node is goal node reconstruct the path.
5. For each neighbour, calculate 'g' & 'h' & 'f' values
6. If the neighbour is not in open set or Cour cost path is found. update costs.
7. Add the neighbour to the open set, if it is not already in the closed set

8. Repeat: until the open set to empty or the goal is found.

Program:

```
import heapq
def a_star(start, goal, h, neighbors):
    open_set = []
    heapq.heappush(open_set, (0+h(start),
                              0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: h(start)}
    while open_set:
        _, current_g, current = heapq.heappop
                                (open_set)
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]
```

```
for neighbor in neighbor(current):
    tentative_g = g_score[current] + 1
    if neighbor not in g_score or
        tentative_g < g_score[neighbor]
    came_from[neighbor] = current
    g_score[neighbor] = tentative_g
    f_score[neighbor] = tentative_g + h(neighbor)
    if higher not in [i[2] for i in open_set]
        heapq.heappush(open_set,
            f_score[neighbor], tentative_g
                neighbor))
return None
def heuristic(node):
    goal_position = (5, 5)
    return abs(node[0] - goal_position
                [0]) +
        abs(node[1] - goal_position[1])
```

```
def neighbors(node):
    x, y = node
    return [(x+1, y), (x-1, y), (x, y+1),
            (x, y-1)]

start = (0,0)
goal = (5,5)
path = a_star(start, goal, heuristic,
       neighbors(node):
print(path)
```

OUTPUT:

[(0,0) (1,0) (2,0) (3,0) (4,0) (5,0)
(5,1) (5,2) (5,3) (5,4) (5,5)]

Result:

Thus the program is successfully executed and the output is verified

Exp 10

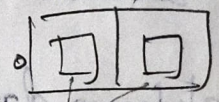
AIM

Implementing Artificial Neural Networks for an application using python classification

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

X_train, y_train = make_circles(n_samples=700,
                                noise=0.05)

X_test, y_test = make_circles(n_samples=300,
                              noise=0.05)

sns.scatterplot(x=X_train[:,0],
                y=X_train[:,1]   # save line
                hue=y_train)
plt.title("Train Data")
plt.show()
```

```
model = MLPClassifier(max_iter=2000)
model.fit(X_train, y_train)
print(f"R2 score for training data =
        {model.score(X_train, y_train)}")
print(f"R2 score for test data =
        {model.score(X_test, y_test)}")
y_pred = model.predict(X_test)
fig, ax = plt.subplots(1,2)
sns.scatterplot(x = X_test[:,0],
                y = X_test[:,1],
                hue = y_pred,
                ax = ax[0])
ax[0].title.set_text("Predicted Data")
sns.scatterplot(x = X_test[:,0],
                y = X_test[:,1],
                hue = y_test,
                ax = ax[1])
ax[1].title.set_text("Test Data")
plt.show()
```



scatterplots

test

	0	1
0		
1		

i/p ↙ o/p (actual o/p)

same

Result:

The code was successfully executed & the output was verified

plt.subplots

↓

Not same

Exp 11

AIM:

Implementing Artificial Neural Network for an application using Python regression.

CODE:

```
import numpy as np
import matplotlib.pyplot as py
import seaborn as sns

from sklearn.datasets import make_regression
from sklearn.model_selection import
                    train_test_split
from sklearn.neural_network import MLPRegressor

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

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

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

```
model=MLPRegressor(max_iter=2000)
model.fit(X_train, y_train)

printf("%f", R2 score for training Data
        =model.score(X_train, y_train))

printf("%f", R2 score for test data =
        model.score(X_test, y_test))
```

```
model = MLPRegressor(max_iter=2000).fit(X_train,
                                        y_train)
```

1. Import
2. Dataset (snf)
3. split (x, y, t, r)
4. Model train
5. print score

Result:

The program was successfully executed and the output was verified

Exp 12 DECISION TREE CLASSIFICATION

~~Aim~~

AIM To implement decision tree classification using Python

CODE:

```
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from google.colab import drive
drive.mount("/content/gdrive")
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

dataset = pd.read_csv('/content/gdrive/My Drive/Social_Network_Ads.csv')

dataset
X = dataset.iloc[:, [2,3]].values
y = dataset.iloc[:, -1].values

split
X_train, X_test, y_train, y_test
= train_test_split(X, y, test_size=0.25, random_state=0)
```

User ID	Gender	Age	Estimated Salary	Purchased
0	1	2	3	4 0,1

```
from sklearn.metrics import confusion_matrix
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
model = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
[[62  6]
 [ 3 29]]
```

```
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,
    model.predict(np.array([X1.ravel(),
                            X2.ravel()]).T)
    .reshape(X1.shape),
    alpha=0.25
    cmap=ListedColormap(['red', 'green']))

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
```

1. Import
2. Dataset
3. Split
4. Scaling
5. Train model & predict

Result:

The program was successfully executed & the output was verified

Exp 13 IMPLEMENTATION OF DECISION TREE CLASSIFICATION TECHNIQUE

AIM

To implement a decision tree classification technique using Python.

CODE:

```
from sklearn import tree
model = tree.DecisionTreeClassifier()
X = [ [181, 80, 91]
      [182, 90, 92]
      [183, 100, 92]
      [184, 200, 93]
      [185, 800, 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
      female
      male
      female
      female
      male ]
mode = model.fit(X, Y)
```

```
prediction_f = model.predict([[181, 80, 11]])
prediction_m = model.predict([[180, 100, 92]])

print(prediction_f)
print(prediction_m)
```

Output

```
['male']
['female']
```

Result:

The program was successfully