INDEX

2220701033

NAME : A. Ashwin [redacted] STD : III year SEC. : A ROLL NO. : [redacted]

S.No.	Date	Title	~~Page No.~~ marks	Teacher's Sign/ Remarks
1.	4/9	N queens	6	
2.	4/9	Depth First Search	6	
3.	11/9	A* Algorithm	6	
4.	18/9	A o* algorithm	6	
5.	25/9	Desision tree	8	
6.	9/10	Artificial Nausal Network.	8	
7.	16/10	K-Means.	9	
8	23/10	Minmax	9	
9	30/10	Introduction to prolog	9	
10	6/11	psolog - Family tree.	9	

completed

N queen problem:

Aim:

To implement Nqueen problem

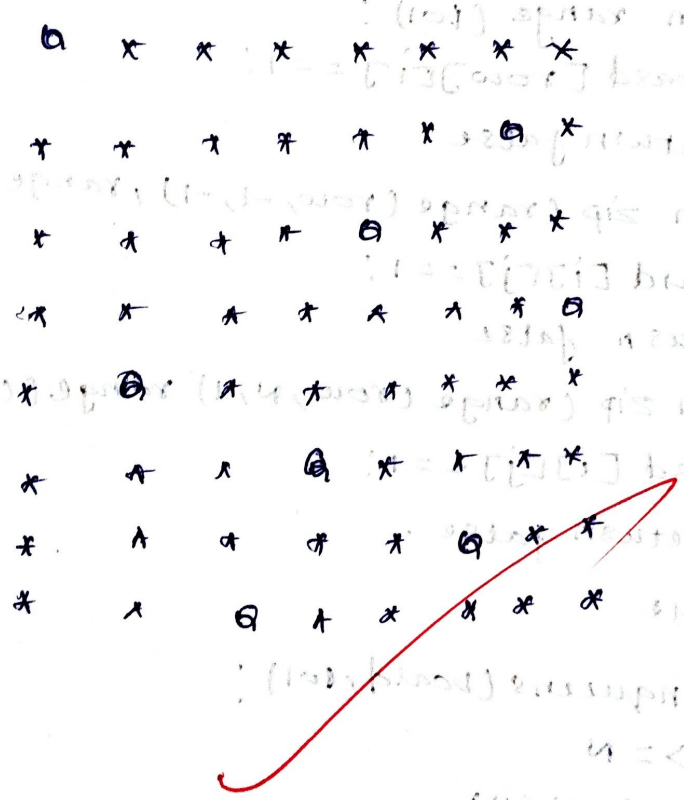
Program:

```
N = int(input("Enter the number of queens:"))
board = [[0]*N for _ in range(N)]
def is_safe(board, row, col):
    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 solve_nqueens(board, col):
    if col >= N:
        return True
    for i in range(N):
        if
            board[i][col] = 1
            if solve_nqueens(board, col+1):
                return True
            board[i][col] = 0
            return False
    if solve_nqueen(board, 0)
        for row in board
```

```
        print(" ".join('Q' if x == i else '*' for x in row))
    else:
        print("No solution exists")
```

output:

Enter the number of queens : 8



Result:

Thus the N queen program has been executed successfully.

Depth first Search.

Aim:

To implement Depth first Search

Program:

```
def dfs - recursive (graph, start, visited = None):
    if visited is none:
        visited = set()
    visited.add(start)
    print(start)
    for neighbor in graph [start]:
        if neighbor not in visited:
            dfs - recursive (graph, neighbours, visited)

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

print ("DFS recursive:")
dfs - recursive (graph, 'A')
def dfs - iterative (graph, start)
        visited = set()
        stack = [start]
    while stack
    vertex = stack.pop()
    if vertex not in visited:
        print (vertex)
        visited.add (vertex)
        stack.extend (neighbor for neighbours in
            graph [vertex] if neighbor not in visited)
```

```
graph = {
  'A' = ['B', 'C'],
  'B' = ['A', 'D', 'E'],
  'C' = ['A', 'F']
  'D' = ['B'],
  'E' = ['B', 'F']
  'F' = ['C', 'E']
Print ("DFS Iterative :")
dfs_Iterative (graph, 'A')
```

output:

```
DFS Iterative:
A
C
F
E
B
D
```

Result:

Thus the depth first search has been implement successfully.

A* Algorithm

Aim:

To Implement A* algorithm.

Program:

```
from heapq import heappop, heappush
class node:
    def __init__(self, position, parent = none):
    self.position = position
    self.parent = parent
    self.g = 0
    self.h = 0
    self.f = 0
    def __eq__(self, other):
        return self.position == other.position
    def __it__(self, other):
        return self.f < other.f
    def a_star(start, goal, grid):
        start_node = Node(start)
        goal_node  Node(goal)
        open_list = set()
        heappush(open_list, start_node)
    while open_list:
        current_node = heappop(open_list)
        closed_list.add(current_node.position)
    if current_node == goal_node:
    path = []
    while current_node:
        path.append(current_node.position)
        current_node = current_node.parent
    return path[::-1]
neighbours = [(0,-1), (0,1), (-1,0), (1,0)]
```

```
for n in neighbors
    neighbor_position = (current_node.position[0]+
                         n[0], current_node.position[1]+n[1])
    if 0<= neighbor_position[0] < len(grid) and 0 <= neighbor
       .position[1] < len(grid[0]) and grid[neighbor_
       position[0]][neighbor_position[1]] == 0:
        neighbor_node = node(neighbor_position, current
                             _node)
    if neighbor_node in position closed_list:
        continue.
    neighbor_node.g = current_node.g + 1
    neighbor_node.h = abs(neighbor_node.position[0] -
                          goal_node.position[0]) + abs(neighbor.
                          node.position[1] - goal_node.position[1])
    neighbor_node.f = neighbor_node.g + neigbor_node.h
    if.all(neighbor_node != open_node for open_node in
                                        open_list)
        heappush(open_list, neighbor_node)
.return none.

grid = [
        [0,1,0,0,0]
        [0,1,0,1,0]
        [0,0,0,1,0]
        [0,1,1,1,0]
        [0,0,0,0,0]]

start = (0,0)
goal = (4,4)
path = a.start[start, goal, grid)
print("path found:", path)
```

output:

path found = [(0,0), (1,0), (2,0), (3,0), (4,0), (4,1)
(4,2), (4,3), (4,4)]

Result:

Thus the A* Algorithm has been explemented and Executed successfully.

AO* Algorithm

Aim:

To implement AO* Algorithm

Program:

```
class graph:
def __init__(self, graph, heuristic)
self.graph = graph
self.heuristic = heuristic
self.solution = {}

def ao_star(self, node)
    print("Expanding: {node}")
if node not in self.graph or not self.graph
                                [node]:
  return.
children = self.graph[node]
best = path = none
min_cost = float('inf')
for group in children
  cost = sum(self.heuristic[child] for child
                                    in group)
  if cost < min_cost
     min_cost = cost
     best_path = group
  self.solution[node] = best_path
print("Best path for {node} = {best_path}
                        with lost {min_cost}")
for child in best_path
   self.ao_star(child)
def get_solution(self):
    return self.solution
graph = {
'A' = [['B','C'], ['A']]
'B' = [['E']]
```

```
'E' = [[G']]
'D' = [['H']]
'F' = [],
G' = [],
'H' = []
}
heuristic = {
'A' = 0, 'B' = 1, 'C' = 2, 'D' = 4, 'E' = 1, 'G' = 3, 'H' = 5
}
graph_obj = Graph(graph, heuristic)
graph_obj = no_stars('A')
Solution = graph_obj.get_solution()
Print("Solution :", solution)
```

Output :

```
Expanding : A
Best path for A = ['B', 'C'] with Best 3
Expanding : B
Best path for B = ['E'] with cost 1
Expanding = E
Expanding = C
Best path = ['G']
Expanding : G
Solution : {'A' = ['B', 'C'], 'B' = ['E'], 'C' = ['G']}
```

Result :

Thus A0* algorithm implemented successfully.

Implementation of Decision tree Classification

Aim:

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

Program:

```
import pandas as pal
from sk
```

Explanation:

* Import the tree
* Call the function from tree
* Assign value for x & y
* call the function on basis of given random values each give feature
* Display the output

Progsam:

```
import pandas as pal
from sklear.tree import decision tree
data = {'Height': [152, 155, 172, 185, 167, 180,
                   164, 177],
 'weight': [45,57,72,85,68,78,22,88]
 'Gender': ['Female', 'female', 'male', 'male',
  'female', 'male', 'female', 'male',
  'female']

  df = pataframe (data)
  X = df ['Height, 'weight']
  Y = df ['Gender'].

classifier = Decisiontree classifier ()
 classifier = fit (X,y)
 height = float (input ("enter height
          (in kg) for prediction"))
```

```
weight=float(input("Enter weight (in kg) for
                    prediction:"))
random_value = pd.DataFrame([[height,
                              weight]],
                            columns=['Height','Weight'])
print(f"Predicted gender for height {height}
      cm and weight {weight} kg:
      {predicted_gender[0]}")
```

output:

Predicted Gender: Female

result:

Thus the python program to implement a decision tree classification technique for gender classification is executed successfully.

Implementation of ANN for an Application using python - Regression.

Aim:

To implement artificial neural networks for an application in regression using python

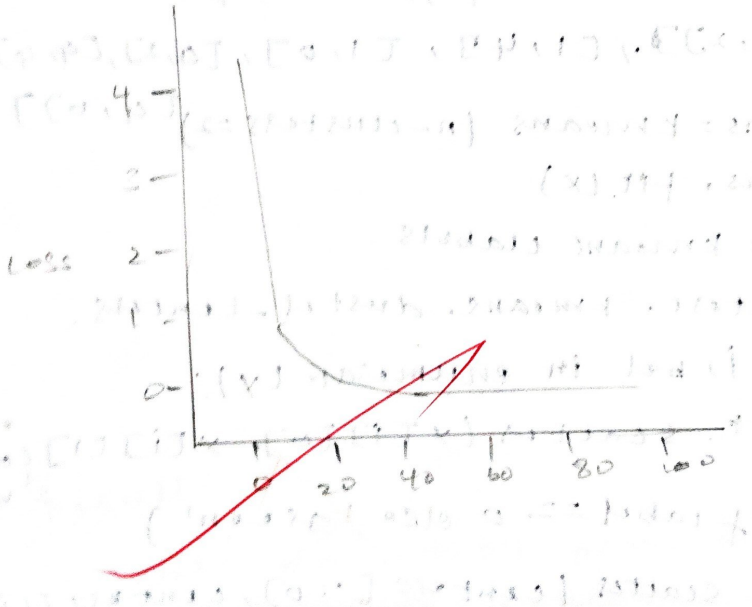
Program:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import standard scaler
from keras_layers import admin
import keras.optimizers import admin
import matplotlib.pyplot as plt.
# Generating Synthetic dataset for demo
x = np.random.rand (1000, 3)
y = 3*x[:,0] + 2^x[:,1] + 1.5 * np.sin(x[:,2]*np.pi+np.random.normal(0,0.1,1000)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
model = sequential()
model.add(Dense(1, input_dim=x_train
          slope[1] activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(optimizer=Adam(learning
                             rule=0.01)
y_pred = model.predict(x_test)
plt.figure(figsize=(12,6))
plt.plot(history.history['loss'], label
         'training loss')
```

```
plt.plot(history.history['val_loss'], label =
                'validation loss')
plt.title('Training & validation loss')
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

output:

Predicted : 0.06, Actual : 25000



Result:

Thus the python program to implement artificial neural networks for an application in regression is executed successfully.

Implementation of k-Means clustering technique.

Aim:

To implement a k-means clustering technique using python language.

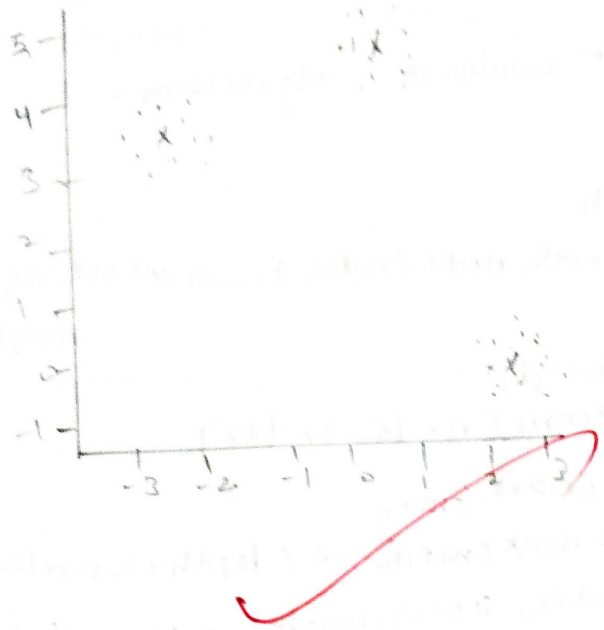
Explanation:

- Import k-means from sklearn.cluster
- Assign x and y
- call the function kmeans()
- perform scatter operation and display the output

Program:

```
from sklearn.cluster import k-means
import matplotlib.pyplot as plt
X = [[1,2]],[1,4],[1,0],[4,2],[4,4],
kmeans = kmeans (n_clusters=2)[4,0]]
kmeans.fit(x)
    y = kmeans.labels_
    centers = kmeans.cluster_centers_
    for i, val in enumerate(y):
      plt.scatter(x[i][0] x[i][1] color
      if label == 0 else 'green')
                                    = 'blue'
    plt.scatter(centers[:,0], centers[1],
      color='red', mask='x', s=100,
      label='centroids']
plt.xlabel('Feature1')
plt.ylabel('Feature2')
plt.title('k-means clustering')
plt.legend()
plt.show()
```

output.



Result:

Thus the python program to implement k-means clustering technique is executed successfully.

MINIMAX

Aim:

To Implement minimax algorithm.

Program.

```
import math
def minimax(depth, node_index, is_maximizes, scores,
                                                height):
    if depth == height:
        return scores[node_index]
    if is_maximizes:
        return max(minimax(depth+1, node_index*2,
            False, scores, height), minimax(depth+1,
                node_index*2+1, false scores, height
    else
        return max(minimax(depth+1, node_index*
            true, scores, height), minmax, minimax
            (depth+1, node_index*2+1)
                true, scores, height))
def calculate_tree_height(num_leaves):
    return math.ceil(math.log2(num_leaves))
scores = [3,5,6,9,1,2,0,-1]
tree_height = calculate_tree_height(len(scores
optimal_score = minimax(0,0, True, scores, tree
print("the optimal                    height)
                    score is :", optimal_score
```

output:

The optimal score is = 5.

result

Thus the decision tree program has been executed successfully.

Introduction to prolog

Aim:

To learn PROLOG terminology and write basic programs.

TERMINOLOGIES.

1) Atomic terms

They are usually strings made up of lowercase and uppercase letters, digits and the underscore, starting with a lowercase letter.

eg. dog. ab_C_321

2) Variables

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

Eg: Dog, Apple_420

3) Compond terms.

Compond terms are made up of a prolog atom and a number of arguments enclosed in parenthesis and seperated by comas

eg: is_bigger(elephant, X). fig(X, _), 7)

4) Fact

A fact is a prediction followed by a dot

eg: bigger_animal(whale). life_a_beautiful

5) Rules

A rules consists of a head and body

eg: h_smaller(Y, X. Y1 = bigger X, X) aunt(Aunt, child) sister(Aunt, parent) parent(parent, child)

source code:

KB 1

```
woman(mia).
woman(jody).
woman(yolanda)
playsAirGuitar(jody)

party.
query1 = ?- woman(mia)
query2 = ?- playsAlgorithm(mia)
query3 = ?- party
query4 = ?- concert.
```

output:

```
?- woman(mia)
  true
?- playsairguitar(mia)
  false
?- concert
Error = unknow procedure. concert/0
```

KB2

```
happy(yolanda)
listens2music(mia).
listen2music(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, britney)
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
```

KB5.

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

output.

```
?- owns(john,X)
X = car(chevy)
?- own(john,_)
  true.
```

Result.

Thus the basic prolog Programs has been executed successfully.

Prolog- Family tree

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)
parent of (chris, peter)
parent of (chris, betty).
parent of (helen, peter)
parent of (kevin, chris).
parent of (kevis, lisa)
parent of (jeny, john)
parent of (jeny, helen)
/* RULES ==*/
/* son, parent
* son, grandparent */

father (x, y) :- male (x), parent of (x, y).
mother (x, y) :- female (y), parent of (x, y)
grandfather (x, y) :- male (y), parent of (x, z), Parent
                                                of (z, y
grandmother (x, y) : female (y). parent of (x, z),
                     parent of (z, y)
brother (x, y) :- male (x), father (x, z) father (y, w), z
                                                == w

sister (x, y) - female (y). father (x, z), father (y, w),
                                                z == w.
```

output

male (peter)

true

father (chris - peter)

true

father (chris, betty)

false

mother (chris, X)

X = betty

brother (chris, nelson)

false

Result.

Thus prolog for family tree program has been executed successfully.