**B.TECH. (2020-24)**

**Artificial Intelligence**

**Lab File**

on

**Artificial Intelligence**

**[CSE401]**

****

Submitted To

**Mr Soumya Ranjan Nayak**

Submitted By

**Hitesh**

**A023119820027**

**4CSE11 (AI)**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY

AMITY UNIVERSITY UTTAR PRADESH

NOIDA (U.P)

**INDEX**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S.No.** | **Category of Assignment** | **Code** | **Exp.**  **No.** | **Name of Experiment** | **Date of Allotment**  **of experiment** | **Date**  **of Evaluation** | **Max.**  **Marks** | **Marks obtained** | **Signature of Faculty** |
| 1. | **Mandatory Experiment\*** | **LR (10)** | **1** | A\* Algorithm | 13/01/2022 | 20/01/2022 |  |  |  |
| 2. | **Mandatory Experiment\*** | **2** | Single Player Game | 20/01/2022 | 27/01/2022 |  |  |  |
| 3. | **Mandatory Experiment\*** | **3** | Tic-Tac-Toe game problem | 27/01/2022 | 03/02/2022 |  |  |  |
| 4. | **Mandatory Experiment\*** | **4** | Knapsack problem | 03/02/2022 | 17/02/2022 |  |  |  |
| 5. | **Mandatory Experiment\*** | **5** | Graph coloring problem | 17/02/2022 | 03/03/2022 |  |  |  |
| 6. | **Mandatory Experiment\*** | **6** | BFS for water jug problem | 03/03/2022 | 10/03/2022 |  |  |  |
| 7 | **Mandatory Experiment\*** | **7** | DFS | 10/03/2022 | 24/03/2022 |  |  |  |
| 8. | **Mandatory Experiment\*** | **8** | Tokenization of word and Sentences | 24/03/2022 | 31/03/2022 |  |  |  |
| 9. | **Mandatory Experiment\*** | **9** | XOR truth table | 31/03/2022 | 07/04/2022 |  |  |  |
| 10. | **Mandatory Experiment\*** | **10** | SCIKIT fuzzy | 13/01/2022 | 20/01/2022 |  |  |  |

**Experiment – 1**

**AIM**

Write a program to implement A\* algorithm in python.

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

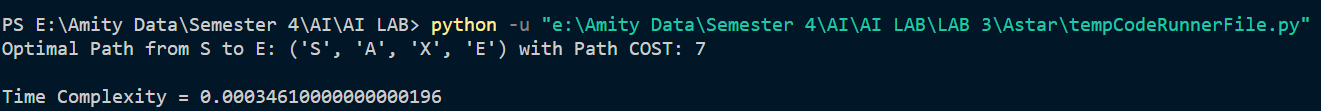
**Libraries USed**

* timeit

**CODE**

|  |
| --- |
| from timeit import default\_timer  class Graph:  f = []  def \_\_init\_\_(self,adj,h = {}):  self.adj\_list = adj    if h == {}:  for i in adj:  for j in adj[i]:  h[j[0]] = 1    self.h = h    def fninsert(self,cur):  k = 0  while k != len(Graph.f):  if Graph.f[k][1][-1] == cur:  temp = Graph.f.pop(k)  for i in self.adj\_list[cur]:  if i[0] not in temp[1]:  Graph.f.insert(k,[ (temp[0] - self.h[cur] + i[1] + self.h[i[0]]), temp[1] + (i[0],) ])    k = k + 1  def isExpandable(self,start,goal):  # if min(Graph.f)[1][-1] == goal:  # return 2  for path in Graph.f:  if path[1][-1] != goal and path[1][-1] in self.adj\_list:  if len(self.adj\_list[path[1][-1]]) != 1:  return 1  elif self.adj\_list[path[1][-1][0]][0] != start:  return 1  else:  return 0  def isPath(self,goal):  for i in Graph.f:  if i[1][-1] == goal:  return 1  return 0  def astar(self,start,goal):  stTime = default\_timer()  Graph.f.clear()    flag = 1  if start not in self.adj\_list:  flag = 0    self.h[goal] = 0    while flag == 1:  if Graph.f == []:  for i in self.adj\_list[start]:  Graph.f.append([i[1] + self.h[i[0]], (start,i[0])])  else:  for i in Graph.f:  # minfn = min(Graph.f)  # if i[1][-1] != goal and minfn == i:  if i[1][-1] != goal:  self.fninsert(i[1][-1])    flag = self.isExpandable(start,goal)    if self.isPath(goal) == 1:  print(f'Optimal Path from {start} to {goal}: {min(Graph.f)[1]} with Path COST: {min(Graph.f)[0]}')  else:  print('\n!!Path not possible!!')  print('\nTime Complexity =',default\_timer()-stTime)      adj\_list = {'S':[('A',1),('B',2)],  'A':[('Y',7),('X',4)],  'B':[('C',7),(('D',1))],  'Y':[('E',3)],  'X':[('E',2)],  'C':[('E',5)],  'D':[('E',12)]  }  h = {'A':5,  'B':6,  'C':4,  'D':15,  'X':5,  'Y':8  }  g = Graph(adj\_list,h)  g.astar('S','E') |

**OUTPUT**



**Conclusion**

A\* algorithm has been implemented successfully.

**Experiment – 2**

**AIM**

Write a program to implement Single Player Game.

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

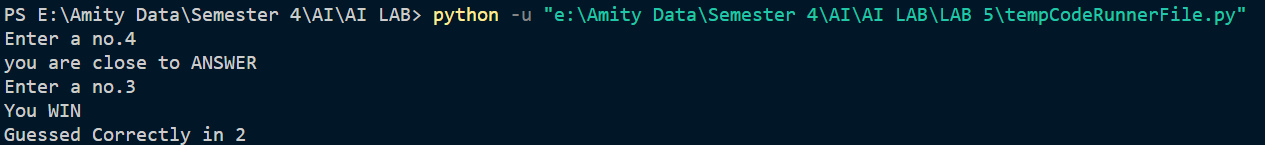
**Libraries USed**

* math
* random

**CODE**

|  |
| --- |
| import math  import random  x = random.randint(1,10)  con = 2  count = 0  while(1):  count = count + 1  test = int(input('Enter a no.'))  if test == x:    print('You WIN\nGuessed Correctly in',count)  break  elif math.fabs(test-x) <= con:  print('you are close to ANSWER')  else:  print('You are far from ANSWER') |

**OUTPUT**



**Conclusion**

Single Player Game has been implemented successfully.

**Experiment – 3**

**AIM**

Write a program to implement Tic-Tac-Toe game problem.

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**Libraries USed**

* os
* tabnanny

**CODE**

|  |
| --- |
| import os  from tabnanny import check  # 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] = opponent  # 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] = player  # 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] = opponent  # 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  def checkWin(board):  for r in board:  if r == ['o','o','o']:  return 1  if r == ['x','x','x']:  return 0  for i in range(len(board)):  c = []  for j in range(len(board)):  c.append(board[j][i])    if c == ['o','o','o']:  return 1  if c == ['x','x','x']:  return 0    if [board[0][0], board[1][1], board[2][2]] == ['o','o','o']:  return 1  if [board[0][0], board[1][1], board[2][2]] == ['x','x','x']:  return 0  if [board[0][2], board[1][1], board[2][0]] == ['o','o','o']:  return 1  if [board[0][2], board[1][1], board[2][0]] == ['x','x','x']:  return 0  def printBoard(board):  for i in board:  for j in i:  print(j, end=" ")  print("\n")  # Driver code  board = [  [ '\_', '\_', '\_' ],  [ '\_', '\_', '\_' ],  [ '\_', '\_', '\_' ]  ]  # bestMove = findBestMove(board)  # print("The Optimal Move is :")  # print("ROW:", bestMove[0], " COL:", bestMove[1])  win = False  draw = False  while not (win or draw):  printBoard(board)  print()  bestMove = findBestMove(board)  board[bestMove[0]][bestMove[1]] = 'o'  if checkWin(board) == 1:  print('cpu wins!')  win = True  break    r = int(input("enter your move (r) : "))  c = int(input("enter your move (c) : "))  board[r-1][c-1] = 'x'  if checkWin(board) == 0:  print('player wins!')  win = True  break    if checkWin(board) != 1 or checkWin(board) != 0:  print("drawwww!!!") |

**OUTPUT**

Text

Description automatically generatedText

Description automatically generated

**Conclusion**

Tic-Tac-Toe game problem has been implemented successfully.

**Experiment – 4**

**AIM**

Implement Brute force solution to the Knapsack problem in Python.

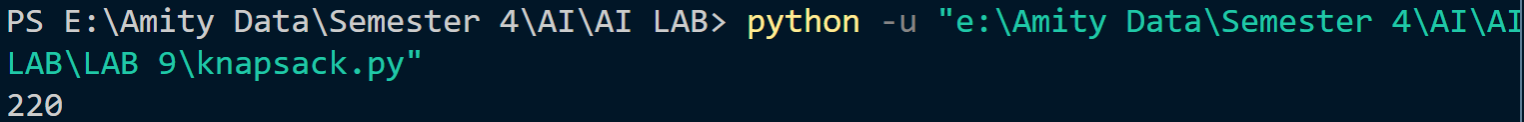
**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**CODE**

|  |
| --- |
| def knapSack(W, wt, val, n):  # initial conditions  if n == 0 or W == 0 :  return 0  # If weight is higher than capacity then it is not included  if (wt[n-1] > W):  return knapSack(W, wt, val, n-1)  # return either nth item being included or not  else:  return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),  knapSack(W, wt, val, n-1))  val = [60, 100, 120]  wt = [10, 20, 30]  W = 50  n = len(val)  print (knapSack(W, wt, val, n)) |

**OUTPUT**

****

**Conclusion**

Brute force implementation of Knapsack problem has been derived successfully.

**Experiment – 5**

**AIM**

Implement Graph colouring problem using python

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**Libraries USed**

* networkx

**CODE**

|  |
| --- |
| import networkx as nx  n = int(input("Enter no. of nodes "))  l = [i for i in range(1,n+1)]  print(l)  network = nx.Graph()  network.add\_nodes\_from(l)  for i in range(1,n):  network.add\_edge(i,i+1,weight=i)  network.add\_edge(1,n,weight = n)  print(network.edges())  print(network.get\_edge\_data(1, 2)['weight'])  color\_list = []  for i in network.edges():  if network.get\_edge\_data(i[0],i[1])['weight'] < 3:  color\_list.append("red")  else:  color\_list.append("blue")    nx.draw(network,node\_color = color\_list,with\_labels = True) |

**OUTPUT**

Chart, line chart

Description automatically generated

**Conclusion**

Graph Colouring problem has been implemented successfully.

**Experiment – 6**

**AIM**

Write a program to implement BFS for water jug problem using Python.

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**Libraries USed**

* collections

**CODE**

|  |
| --- |
| from collections import deque  def BFS(a, b, target):    # Map is used to store the states, every  # state is hashed to binary value to  # indicate either that state is visited  # before or not  m = {}  isSolvable = False  path = []    # Queue to maintain states  q = deque()    # Initialing with initial state  q.append((0, 0))  while (len(q) > 0):    # Current state  u = q.popleft()  #q.pop() #pop off used state  # If this state is already visited  if ((u[0], u[1]) in m):  continue  # Doesn't met jug constraints  if ((u[0] > a or u[1] > b or  u[0] < 0 or u[1] < 0)):  continue  # Filling the vector for constructing  # the solution path  path.append([u[0], u[1]])  # Marking current state as visited  m[(u[0], u[1])] = 1  # If we reach solution state, put ans=1  if (u[0] == target or u[1] == target):  isSolvable = True    if (u[0] == target):  if (u[1] != 0):    # Fill final state  path.append([u[0], 0])  else:  if (u[0] != 0):  # Fill final state  path.append([0, u[1]])  # Print the solution path  sz = len(path)  for i in range(sz):  print("(", path[i][0], ",",  path[i][1], ")")  break  # If we have not reached final state  # then, start developing intermediate  # states to reach solution state  q.append([u[0], b]) # Fill Jug2  q.append([a, u[1]]) # Fill Jug1  for ap in range(max(a, b) + 1):  # Pour amount ap from Jug2 to Jug1  c = u[0] + ap  d = u[1] - ap  # Check if this state is possible or not  if (c == a or (d == 0 and d >= 0)):  q.append([c, d])  # Pour amount ap from Jug 1 to Jug2  c = u[0] - ap  d = u[1] + ap  # Check if this state is possible or not  if ((c == 0 and c >= 0) or d == b):  q.append([c, d])    # Empty Jug2  q.append([a, 0])    # Empty Jug1  q.append([0, b])  # No, solution exists if ans=0  if (not isSolvable):  print ("No solution")  # Driver code    Jug1, Jug2, target = 4, 3, 2  print("Path from initial state to solution state ::")    BFS(Jug1, Jug2, target) |

**OUTPUT**

Text

Description automatically generated with medium confidence

**Conclusion**

Water Jug problem using BFS has been implemented successfully.

**Experiment – 7**

**AIM**

Write a program to implement DFS using Python.

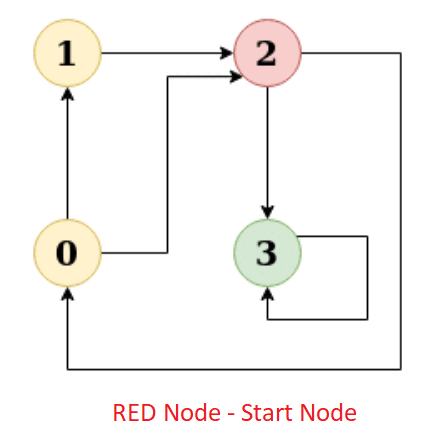
**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**Libraries USed**

* collections

**Graph used**

****

**CODE**

|  |
| --- |
| # Python3 program to print DFS traversal  # from a given given graph  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 code  # Create a graph given  # in the above diagram  g = Graph()  g.addEdge(0, 1)  g.addEdge(0, 2)  g.addEdge(1, 2)  g.addEdge(2, 0)  g.addEdge(2, 3)  g.addEdge(3, 3)  print("Following is DFS from (starting from vertex 2)")  g.DFS(2) |

**OUTPUT**

Text

Description automatically generated

**Conclusion**

DFS has been implemented successfully.

**Experiment – 8**

**AIM**

Tokenization of word and Sentences with the help of NLTK package.

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**Libraries USed**

* nltk

**CODE**

|  |
| --- |
| from nltk.tokenize import \*  text = "God is Great! I won a lottery."  print(word\_tokenize(text))  text = '''God is Great! I won a lottery.  i lost a lotter. help me! i died'''  print(sent\_tokenize(text)) |

**OUTPUT**





**Conclusion**

Tokenization of word and Sentences with the help of NLTK package has been implemented successfully.

**Experiment – 9**

**AIM**

Design an XOR truth table using Python.

**PROGRAMMING LANGUAGE / TOOL USED**

Python Programming Language

**CODE**

|  |
| --- |
| def XOR (a, b):  if a != b:  return 1  else:  return 0  print(f'A B OUTPUT')  print('------------------')  for i in [0,1]:  for j in [0,1]:  print(f'{i} {j} {XOR(i,j)}') |

**OUTPUT**

A picture containing text, device, meter

Description automatically generated

**Conclusion**

Truth table for XOR gate using Python has been constructed successfully.

**Experiment – 10**

**AIM**

Study of SCIKIT fuzzy.

**THEORY**

Scikit-Fuzzy is a collection of fuzzy logic algorithms intended for use in the SciPy Stack, written in the Python computing language.

This package implements many useful tools for projects involving fuzzy logic, also known as grey logic.

scikit-fuzzy (a.k.a. skfuzzy): Fuzzy logic toolbox for Python.

Scikit-fuzzy is a robust set of foundational tools for problems involving fuzzy logic and fuzzy systems. This area has been a challenge for the scientific Python community, largely because the common first exposure to this topic is through the MATLAB® Fuzzy Logic Toolbox™.

Within scikit-fuzzy, universe variables and fuzzy membership functions are represented by numpy arrays. Generation of membership functions is as simple as:

A picture containing scatter chart

Description automatically generated

While most functions are available in the base namespace, the package is factored with a logical grouping of functions in submodules. The current capabilities of scikit-fuzzy include:

* fuzz.membership: Fuzzy membership function generation
* fuzz.defuzzify: Defuzzification algorithms to return crisp results from fuzzy sets
* fuzz.fuzzymath: The core of scikit-fuzzy, containing the majority of the most common fuzzy logic operations.
* fuzz.intervals: Interval mathematics. The restricted Dong, Shah, & Wong (DSW) methods for fuzzy set math live here.
* fuzz.image: Limited fuzzy logic image processing operations.
* fuzz.cluster: Fuzzy c-means clustering.
* fuzz.filters: Fuzzy Inference Ruled by Else-action (FIRE) filters in 1D and 2D.

The goals of scikit-fuzzy are to provide the community with a robust toolkit of independently developed and implemented fuzzy logic algorithms, filling a void in the capabilities of scientific and numerical Python, and to increase the attractiveness of scientific Python as a valid alternative to closed-source options. Scikit-fuzzy is structured similarly to scikit-learn and scikit-image, current source code is available on GitHub, and pull requests are welcome.

**Conclusion**

The fuzzy logic algorithms offered by the SCIKIT package have been studied.