**Lab File**

**Artificial Intelligence**

**[CSE401]**

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING



**Submitted To: Submitted By:**

 Dr. S. K. Dubey Krishna Sahu Associate Professor A2305219080 CSE Department, ASET B. Tech (CSE)

6CSE2X

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY

AMITY UNIVERSITY UTTAR PRADESH

NOIDA-201301

**INDEX**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Experiment No.** | **Category**  **of Assignment** | **Code** | **Name of Experiment** | **Date of Allotment of experiment** | **Date of Evaluation** | **Max.**  **Marks** | **Marks obtained** | **Sign.**  **of Faculty** |
|  | **Mandatory Experiment** | **LR (0)** | Write a program to implement A\* algorithm in python |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Write a program to implement Single Player Game |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Write a program to implement Tic-Tac-Toe game problem |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Implement Brute force solution to the Knapsack problem in Python |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Implement Graph coloring problem using python |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Write a program to implement BFS for water jug problem using Python |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Write a program to implement DFS using Python |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Tokenization of word and Sentences with the help of NLTK package |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Design an XOR truth table using Python |  |  | **1** |  |  |
|  | **Mandatory Experiment** |  | Study of SCIKIT fuzzy |  |  | **1** |  |  |

# **EXPERIMENT 1**

**Date:**

**Objective:** Write a program to implement A\* algorithm in python

**Software Used:** Python

**Theory:**

A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

The path found by A\* is made of graph nodes and edges. The edges are abstract mathematical concepts. A\* will tell you to move from one location to another but it won’t tell you how. Remember that it doesn’t know anything about rooms or doors; all it sees is the graph. You’ll have to decide whether a graph edge returned by A\* means moving from tile to tile or walking in a straight line or opening a door or swimming or running along a curved path.

Tradeoffs: For any given game map, there are many different ways of making a pathfinding graph to give to A\*. The above map makes most doorways into nodes;

Diagram

Description automatically generated

**Code:**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] is None:

pass

else:

for (m, weight) in get\_neighbors(n):

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

# update g(m)

g[m] = g[n] + weight

# change parent of m to n

parents[m] = n

# if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n is None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

c = 0

while parents[n] != n:

c += g[n]

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

print('Cost: {}'.format(c))

return path

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

H\_dist = {

'A': 14,

'B': 12,

'C': 11,

'D': 6,

'E': 4,

'F': 11,

'G': 0,

}

return H\_dist[n]

# Describe your graph here

Graph\_nodes = {

'A': [('B', 4), ('C', 3)],

'B': [('E', 12), ('F', 5)],

'C': [('E', 10), ('D', 7)],

'D': [('E', 2)],

'E': [('G', 5)],

'F': [('G', 16)],

}

aStarAlgo('A', 'G')

**INPUT:**

{

'A': [('B', 7), ('C', 4)],

'B': [('E', 10), ('F', 5)],

'C': [('E', 12), ('D', 7)],

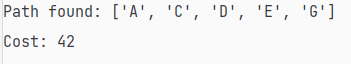
'D': [('E', 4)],

'E': [('G', 9)],

'F': [('G', 11)],

}

**OUTPUT**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Artificial Intelligence |
| Course Code | [CSE401] | | Semester | 6 |
| Student Name | Krishna Sahu | | Enrollment No. | A2305219080 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# **EXPERIMENT 2**

**Date:**

**Objective:** Write a program to implement Single Player Game

**Software Used:** Python

**Theory:**

Definition:

“It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state”.

A picture containing text, shoji, crossword puzzle, clipart

Description automatically generated

The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle i s a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it.

A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration such as two situations given below.

The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration such as two situations given below.

**Code:**

# Python3 program to print the path from root

# node to destination node for N\*N-1 puzzle

# algorithm using Branch and Bound

# The solution assumes that instance of

# puzzle is solvable

# Importing copy for deepcopy function

import copy

# Importing the heap functions from python

# library for Priority Queue

from heapq import heappush, heappop

# This variable can be changed to change

# the program from 8 puzzle(n=3) to 15

# puzzle(n=4) to 24 puzzle(n=5)...

n = 3

# bottom, left, top, right

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

# A class for Priority Queue

class priorityQueue:

# Constructor to initialize a

# Priority Queue

def \_\_init\_\_(self):

self.heap = []

# Inserts a new key 'k'

def push(self, k):

heappush(self.heap, k)

# Method to remove minimum element

# from Priority Queue

def pop(self):

return heappop(self.heap)

# Method to know if the Queue is empty

def empty(self):

if not self.heap:

return True

else:

return False

# Node structure

class node:

def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

cost, level):

# Stores the parent node of the

# current node helps in tracing

# path when the answer is found

self.parent = parent

# Stores the matrix

self.mat = mat

# Stores the position at which the

# empty space tile exists in the matrix

self.empty\_tile\_pos = empty\_tile\_pos

# Storesthe number of misplaced tiles

self.cost = cost

# Stores the number of moves so far

self.level = level

# This method is defined so that the

# priority queue is formed based on

# the cost variable of the objects

def \_\_lt\_\_(self, nxt):

return self.cost < nxt.cost

# Function to calculate the number of

# misplaced tiles ie. number of non-blank

# tiles not in their goal position

def calculateCost(mat, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mat[i][j]) and

(mat[i][j] != final[i][j])):

count += 1

return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

level, parent, final) -> node:

# Copy data from parent matrix to current matrix

new\_mat = copy.deepcopy(mat)

# Move tile by 1 position

x1 = empty\_tile\_pos[0]

y1 = empty\_tile\_pos[1]

x2 = new\_empty\_tile\_pos[0]

y2 = new\_empty\_tile\_pos[1]

new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

# Set number of misplaced tiles

cost = calculateCost(new\_mat, final)

new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

cost, level)

return new\_node

# Function to print the N x N matrix

def printMatrix(mat):

for i in range(n):

for j in range(n):

print("%d " % (mat[i][j]), end = " ")

print()

# Function to check if (x, y) is a valid

# matrix coordinate

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatrix(root.mat)

print()

# Function to solve N\*N - 1 puzzle algorithm

# using Branch and Bound. empty\_tile\_pos is

# the blank tile position in the initial state.

def solve(initial, empty\_tile\_pos, final):

# Create a priority queue to store live

# nodes of search tree

pq = priorityQueue()

# Create the root node

cost = calculateCost(initial, final)

root = node(None, initial,

empty\_tile\_pos, cost, 0)

# Add root to list of live nodes

pq.push(root)

# Finds a live node with least cost,

# add its children to list of live

# nodes and finally deletes it from

# the list.

while not pq.empty():

# Find a live node with least estimated

# cost and delete it form the list of

# live nodes

minimum = pq.pop()

# If minimum is the answer node

if minimum.cost == 0:

# Print the path from root to

# destination;

printPath(minimum)

return

# Generate all possible children

for i in range(n):

new\_tile\_pos = [

minimum.empty\_tile\_pos[0] + row[i],

minimum.empty\_tile\_pos[1] + col[i], ]

if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

# Create a child node

child = newNode(minimum.mat,

minimum.empty\_tile\_pos,

new\_tile\_pos,

minimum.level + 1,

minimum, final,)

# Add child to list of live nodes

pq.push(child)

# Driver Code

# Initial configuration

# Value 0 is used for empty space

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

# Solvable Final configuration

# Value 0 is used for empty space

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

# Blank tile coordinates in

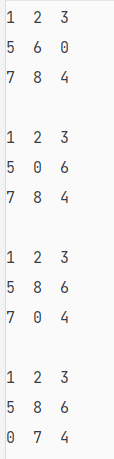
# initial configuration

empty\_tile\_pos = [ 1, 2 ]

# Function call to solve the puzzle

solve(initial, empty\_tile\_pos, final)

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Artificial Intelligence |
| Course Code | [CSE401] | | Semester | 6 |
| Student Name | Krishna Sahu | | Enrollment No. | A2305219080 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# **EXPERIMENT 3**

**Date:**

**Objective:** Write a program to implement Tic-Tac-Toe game problem

**Software Used:** Python

**Theory:**

Tic-Tac-Toe is among the games played between two players played on a 3 x 3 square grid. Each player inhabits a cell in their respective turns, keeping the objective of placing three similar marks in a vertical, horizontal, or diagonal pattern. The first player utilizes the Cross (X) as the marker, whereas the other utilizes the Naught or Zero (O).

Now, let us understand the design of Tic-Tac-Toe.

We will be using the command prompt in order to play Tic-Tac-Toe. Thus, it is the primary objective to construct a design for the Tic-Tac-Toe game.

**Code:**

print("--------Tic Tac Toe----------")

board = ["-", "-", "-",

"-", "-", "-",

"-", "-", "-"]

game\_still\_going = True

winner = None

current\_player = "X"

def play\_game():

# Show the initial game board

display\_board()

# Loop until the game stops (winner or tie)

while game\_still\_going:

# Handle a turn

handle\_turn(current\_player)

# Check if the game is over

check\_if\_game\_over()

# Flip to the other player

flip\_player()

# Since the game is over, print the winner or tie

if winner == "X" or winner == "O":

print(winner + " won.")

elif winner == None:

print("Tie.")

# Display the game board to the screen

def display\_board():

print("\n")

print(board[0] + " | " + board[1] + " | " + board[2] + " 1 | 2 | 3")

print(board[3] + " | " + board[4] + " | " + board[5] + " 4 | 5 | 6")

print(board[6] + " | " + board[7] + " | " + board[8] + " 7 | 8 | 9")

print("\n")

# Handle a turn for an arbitrary player

def handle\_turn(player):

# Get position from player

print(player + "'s turn.")

position = input("Choose a position from 1-9: ")

# Whatever the user inputs, make sure it is a valid input, and the spot is open

valid = False

while not valid:

# Make sure the input is valid

while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:

position = input("Choose a position from 1-9: ")

# Get correct index in our board list

position = int(position) - 1

# Then also make sure the spot is available on the board

if board[position] == "-":

valid = True

else:

print("You can't go there. Go again.")

# Put the game piece on the board

board[position] = player

# Show the game board

display\_board()

# Check if the game is over

def check\_if\_game\_over():

check\_for\_winner()

check\_for\_tie()

# Check to see if somebody has won

def check\_for\_winner():

# Set global variables

global winner

# Check if there was a winner anywhere

row\_winner = check\_rows()

column\_winner = check\_columns()

diagonal\_winner = check\_diagonals()

# Get the winner

if row\_winner:

winner = row\_winner

elif column\_winner:

winner = column\_winner

elif diagonal\_winner:

winner = diagonal\_winner

else:

winner = None

# Check the rows for a win

def check\_rows():

# Set global variables

global game\_still\_going

# Check if any of the rows have all the same value (and is not empty)

row\_1 = board[0] == board[1] == board[2] != "-"

row\_2 = board[3] == board[4] == board[5] != "-"

row\_3 = board[6] == board[7] == board[8] != "-"

# If any row does have a match, flag that there is a win

if row\_1 or row\_2 or row\_3:

game\_still\_going = False

# Return the winner

if row\_1:

return board[0]

elif row\_2:

return board[3]

elif row\_3:

return board[6]

# Or return None if there was no winner

else:

return None

# Check the columns for a win

def check\_columns():

# Set global variables

global game\_still\_going

# Check if any of the columns have all the same value (and is not empty)

column\_1 = board[0] == board[3] == board[6] != "-"

column\_2 = board[1] == board[4] == board[7] != "-"

column\_3 = board[2] == board[5] == board[8] != "-"

# If any row does have a match, flag that there is a win

if column\_1 or column\_2 or column\_3:

game\_still\_going = False

# Return the winner

if column\_1:

return board[0]

elif column\_2:

return board[1]

elif column\_3:

return board[2]

# Or return None if there was no winner

else:

return None

# Check the diagonals for a win

def check\_diagonals():

# Set global variables

global game\_still\_going

# Check if any of the columns have all the same value (and is not empty)

diagonal\_1 = board[0] == board[4] == board[8] != "-"

diagonal\_2 = board[2] == board[4] == board[6] != "-"

# If any row does have a match, flag that there is a win

if diagonal\_1 or diagonal\_2:

game\_still\_going = False

# Return the winner

if diagonal\_1:

return board[0]

elif diagonal\_2:

return board[2]

# Or return None if there was no winner

else:

return None

# Check if there is a tie

def check\_for\_tie():

# Set global variables

global game\_still\_going

# If board is full

if "-" not in board:

game\_still\_going = False

return True

# Else there is no tie

else:

return False

# Flip the current player from X to O, or O to X

def flip\_player():

# Global variables we need

global current\_player

# If the current player was X, make it O

if current\_player == "X":

current\_player = "O"

# Or if the current player was O, make it X

elif current\_player == "O":

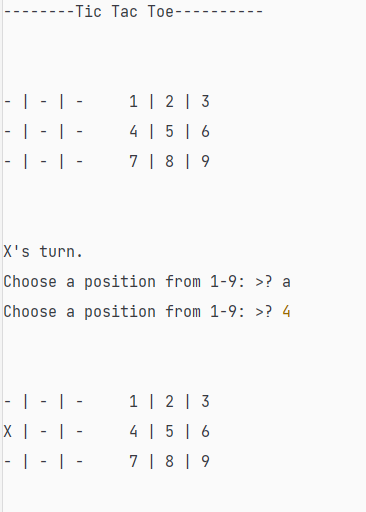
current\_player = "X"

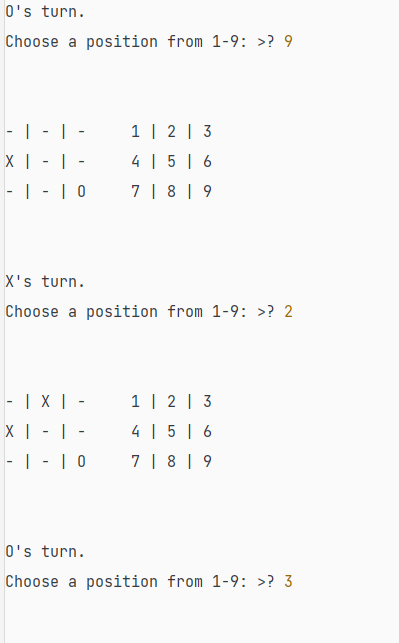
# ------------ Start Execution -------------

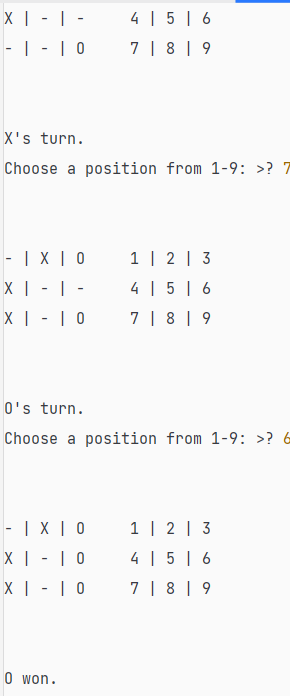
# Play a game of tic tac toe

play\_game()

**Output:**







|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Artificial Intelligence |
| Course Code | [CSE401] | | Semester | 6 |
| Student Name | Krishna Sahu | | Enrollment No. | A2305219080 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# **EXPERIMENT 4**

**Date:**

**Objective:** Implement Brute force solution to the Knapsack problem in Python

**Software Used:** Python

**Theory:**

A knapsack problem algorithm is a constructive approach to combinatorial optimization. The problem is basically about a given set of items, each with a specific weight and a value. Therefore the programmer needs to determine each item’s number to include in a collection so that the total weight is less than or equal to a given limit. And also, the total value is maximum. It derives its name from the fixed-size knapsack that must be filled up to its limit with the most valuable items.

0/1 knapsack problem is a special case knapsack problem that does not fill the knapsack with fractional items.

**Code:**

import itertools

from xml.etree.ElementPath import find

def find\_combination(nums):

result: list = []

for idx in range(len(nums)):

for sub\_set in itertools.combinations(nums, idx):

result.append(list(sub\_set))

result.pop(0)

result.append(nums)

return result

wt = [2, 5, 10,5]

val = [20, 30, 50, 10]

W = 16

n = len(val)

print("Item\tWeight\tValue")

for i in range(n):

print(i+1,"\t",wt[i],"\t",val[i])

print("Knapsack Capacity:",W)

result= find\_combination([1,2,3,4])

tw=[]

tp=[]

w=0

p=0

for item in result:

for i in item:

w+=wt[i-1]

p+=val[i-1]

tw.append(w)

tp.append(p)

w=0

p=0

max=tp[0]

maxindex=0

print("Item Weight Profit \n")

for i in range(len(result)):

if tw[i]>W:

print(result[i],"-->",tw[i],"-->",tp[i],"--->","Not feasible")

else:

print(result[i],"-->",tw[i],"-->",tp[i])

if max<tp[i]:

max=tp[i]

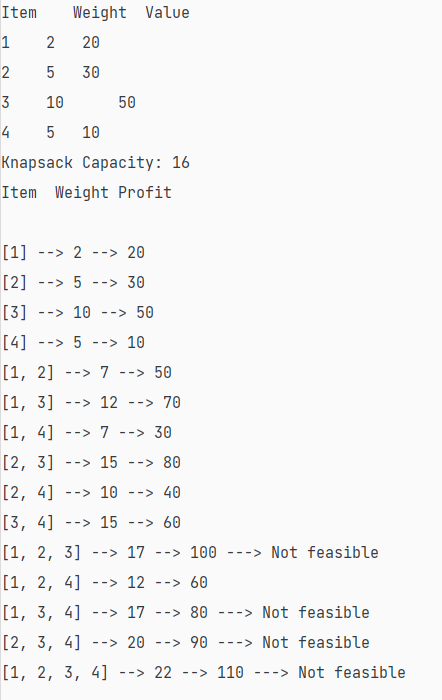
maxindex=i

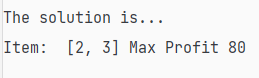
# pos=tp.index(max(tp))

print("\n\nThe solution is...")

print("Item: ",result[maxindex], "Max Profit", tp[maxindex])

**Output:**





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Artificial Intelligence |
| Course Code | [CSE401] | | Semester | 6 |
| Student Name | Krishna Sahu | | Enrollment No. | A2305219080 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# **EXPERIMENT 5**

**Date:**

**Objective:** Implement Graph coloring problem using python

**Software Used:** Python

**Theory:**

Graph coloring problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like Edge Coloring (No vertex is incident to two edges of same color) and Face Coloring (Geographical Map Coloring) can be transformed into vertex coloring.

Diagram

Description automatically generated

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 2 colors.

vertex\_coloring

The problem to find chromatic number of a given graph is NP Complete.

Applications of Graph Coloring:

The graph coloring problem has huge number of applications.

1) Making Schedule or Time Table: Suppose we want to make am exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams? This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

**Code:**

colors = ['Red', 'Blue', 'Green','Yellow', 'Black']

states = ['A','B','C','D','E']

neighbors = {}

neighbors['A']=['B','C']

neighbors['B']=['A','E']

neighbors['E']=['B','D']

neighbors['D']=['E','C']

neighbors['C']=['A','D']

# neighbors['Andhra'] = ['Karnataka', 'TamilNadu']

# neighbors['Karnataka'] = ['Andhra', 'TamilNadu', 'Kerala']

# neighbors['TamilNadu'] = ['Andhra', 'Karnataka', 'Kerala']

# neighbors['Kerala'] = ['Karnataka', 'TamilNadu']

colors\_of\_states = {}

def promising(state, color):

for neighbor in neighbors.get(state):

color\_of\_neighbor = colors\_of\_states.get(neighbor)

if color\_of\_neighbor == color:

return False

return True

def get\_color\_for\_state(state):

for color in colors:

if promising(state, color):

return color

print("\n\n\n")

def main():

for state in states:

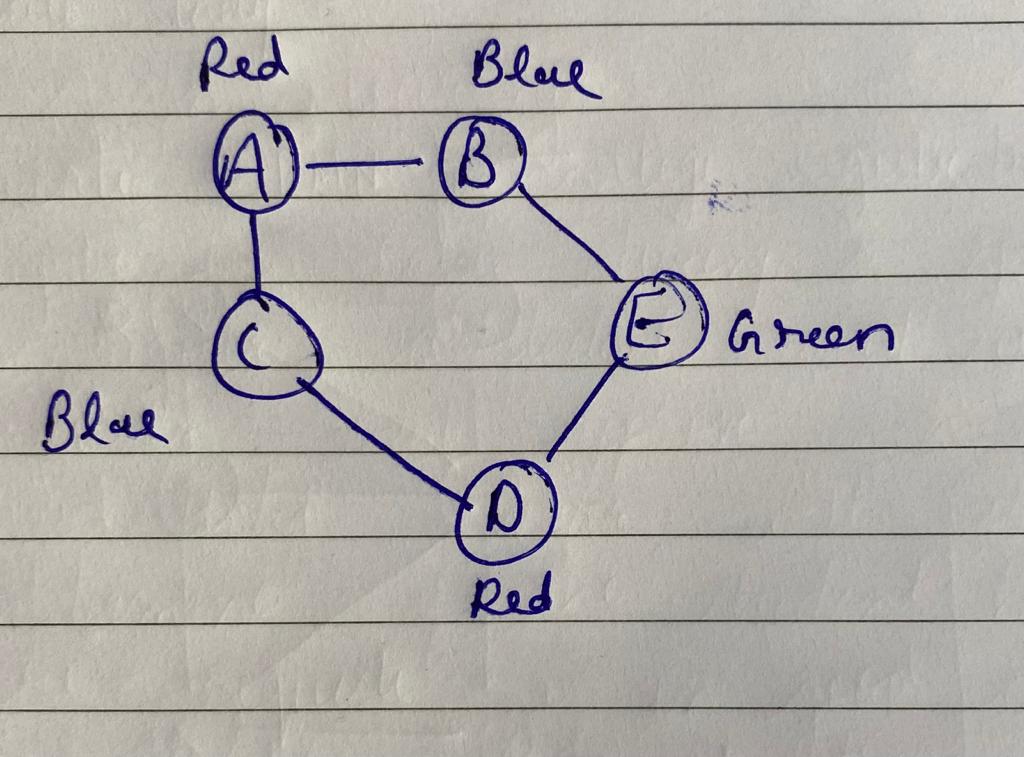
colors\_of\_states[state] = get\_color\_for\_state(state)

print (colors\_of\_states)

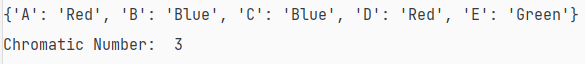
print("Chromatic Number: ",len(set(colors\_of\_states.values())))

main()

**Input:**

****

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Artificial Intelligence |
| Course Code | [CSE401] | | Semester | 6 |
| Student Name | Krishna Sahu | | Enrollment No. | A2305219080 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# **EXPERIMENT 6**

**Date:**

**Objective:** Write a program to implement BFS for water jug problem using Python

**Software Used:** Python

**Theory:**

You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2

Determine the path from the initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

Empty a Jug, (X, Y)->(0, Y) Empty Jug 1

Fill a Jug, (0, 0)->(X, 0) Fill Jug 1

Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

**Code:**

from collections import deque

def BFS(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 = 5,3,1

print("Path from initial state to solution state ::")

print("Capacity of x: ",Jug1)

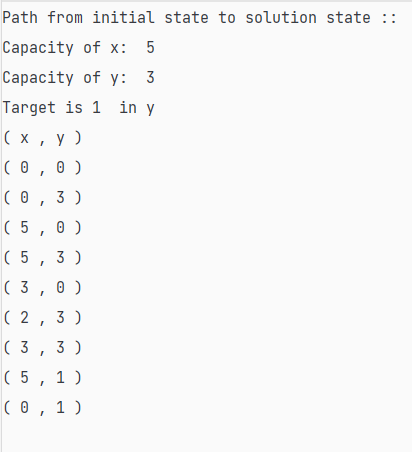
print("Capacity of y: ",Jug2)

print("Target is",target," in y")

print("( x , y )")

BFS(Jug1, Jug2, target)

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Artificial Intelligence |
| Course Code | [CSE401] | | Semester | 6 |
| Student Name | Krishna Sahu | | Enrollment No. | A2305219080 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |