8 PUZZLE PROGRAM:

```
import copy

# Importing the heap methods from the python
# library for the Priority Queue
from heapq import heappush, heappop

# This particular var can be changed to transform
# the program from 8 puzzle(n=3) into 15
# puzzle(n=4) and so on ...
n = 3

# bottom, left, top, right
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

# creating a class for the Priority Queue
class priorityQueue:

    # Constructor for initializing a
    # Priority Queue
    def __init__(self):
        self.heap = []

    # Inserting a new key 'key'
    def push(self, key):
        heappush(self.heap, key)
```

```
    # funct to remove the element that is minimum,
    # from the Priority Queue
    def pop(self):
        return heappop(self.heap)

    # funct to check if the Queue is empty or not
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# structure of the node
class nodes:

    def __init__(self, parent, mats, empty_tile_posi,
                costs, levels):

        # This will store the parent node to the
        # current node And helps in tracing the
        # path when the solution is visible
        self.parent = parent

        # Useful for Storing the matrix
        self.mats = mats

        # useful for Storing the position where the
        # empty space tile is already existing in the matrix
        self.empty_tile_posi = empty_tile_posi
```

```
        # Store no. of misplaced tiles
        self.costs = costs

        # Store no. of moves so far
        self.levels = levels

    # This func is used in order to form the
    # priority queue based on
    # the costs var of objects
    def __lt__(self, nxt):
        return self.costs < nxt.costs

# method to calc. the no. of
# misplaced tiles, that is the no. of non-blank
# tiles not in their final posi
def calculateCosts(mats, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1

    return count

def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
            levels, parent, final) -> nodes:
```

```
    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)

    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                costs, levels)
    return new_nodes

# func to print the N by N matrix
def printMatsrix(mats):

    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")

        print()

# func to know if (x, y) is a valid or invalid
# matrix coordinates
```

```
def isSafe(x, y):

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

# Printing the path from the root node to the final node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatsrix(root.mats)
    print()

# method for solving N*N - 1 puzzle algo
# by utilizing the Branch and Bound technique. empty_tile_posi is
# the blank tile position initially.
def solve(initial, empty_tile_posi, final):

    # Creating a priority queue for storing the live
    # nodes of the search tree
    pq = priorityQueue()

    # Creating the root node
    costs = calculateCosts(initial, final)
    root = nodes(None, initial,
                 empty_tile_posi, costs, 0)

    # Adding root to the list of live nodes
```

```
pq.push(root)

# Discovering a live node with min. costs,
# and adding its children to the list of live
# nodes and finally deleting it from
# the list.
while not pq.empty():

    # Finding a live node with min. estimatsed
    # costs and deleting it form the list of the
    # live nodes
    minimum = pq.pop()

    # If the min. is ans node
    if minimum.costs == 0:

        # Printing the path from the root to
        # destination;
        printPath(minimum)
        return

    # Generating all feasible children
    for i in range(n):
        new_tile_posi = [
            minimum.empty_tile_posi[0] + rows[i],
            minimum.empty_tile_posi[1] + cols[i], ]

        if isSafe(new_tile_posi[0], new_tile_posi[1]):
```

```
        # Creating a child node
        child = newNodes(minimum.mats,
                minimum.empty_tile_posi,
                new_tile_posi,
                minimum.levels + 1,
                minimum, final,)

        # Adding the child to the list of live nodes
        pq.push(child)

# Main Code

# Initial configuration
# Value 0 is taken here as an empty space
initial = [ [ 1, 2, 3 ],
      [ 5, 6, 0 ],
      [ 7, 8, 4 ] ]

# Final configuration that can be solved
# Value 0 is taken as an empty space
final = [ [ 1, 2, 3 ],
    [ 5, 8, 6 ],
    [ 0, 7, 4 ] ]

# Blank tile coordinates in the
# initial configuration
empty_tile_posi = [ 1, 2 ]

# Method call for solving the puzzle
```

```
solve(initial, empty_tile_posi, final)
```

8QUEENS:

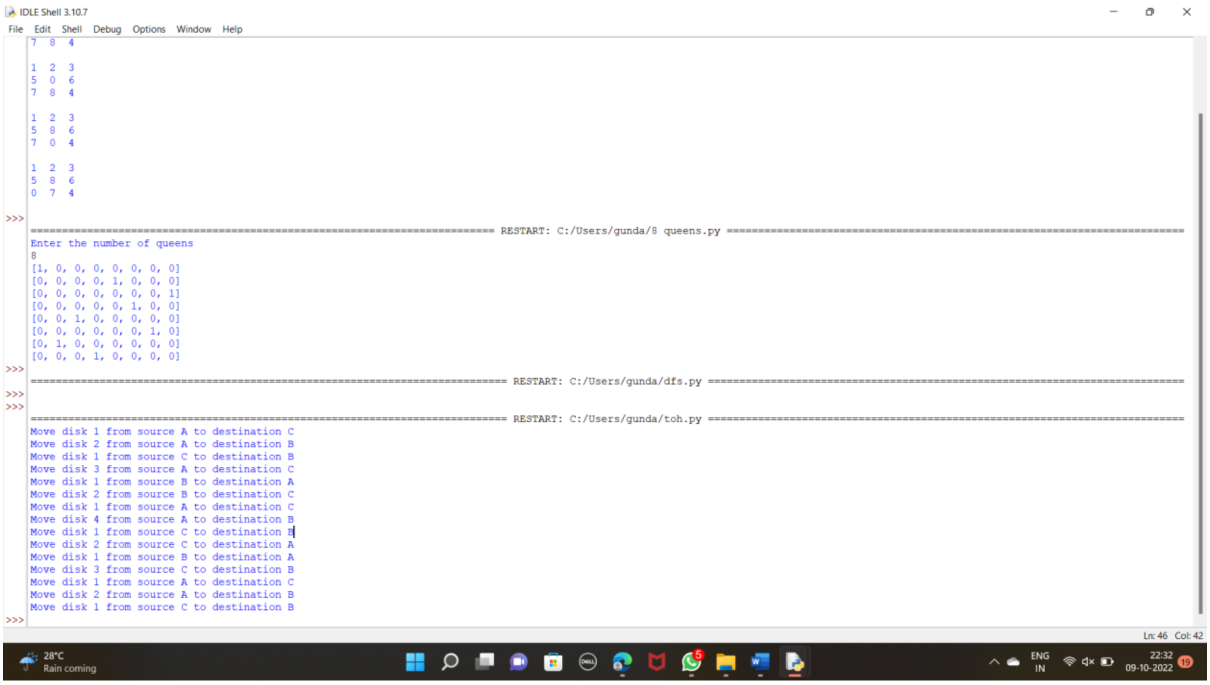
```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
```

```
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
for i in board:
    print (i)
```

Tower of Hanoi:

```
def TowerOfHanoi(n , source, destination, auxiliary):
    if n==1:
```

```
        print ("Move disk 1 from source",source,"to destination",destination)
        return
    TowerOfHanoi(n-1, source, auxiliary, destination)
    print ("Move disk",n,"from source",source,"to destination",destination)
    TowerOfHanoi(n-1, auxiliary, destination, source)

# Driver code
n = 4
TowerOfHanoi(n,'A','B','C')
```

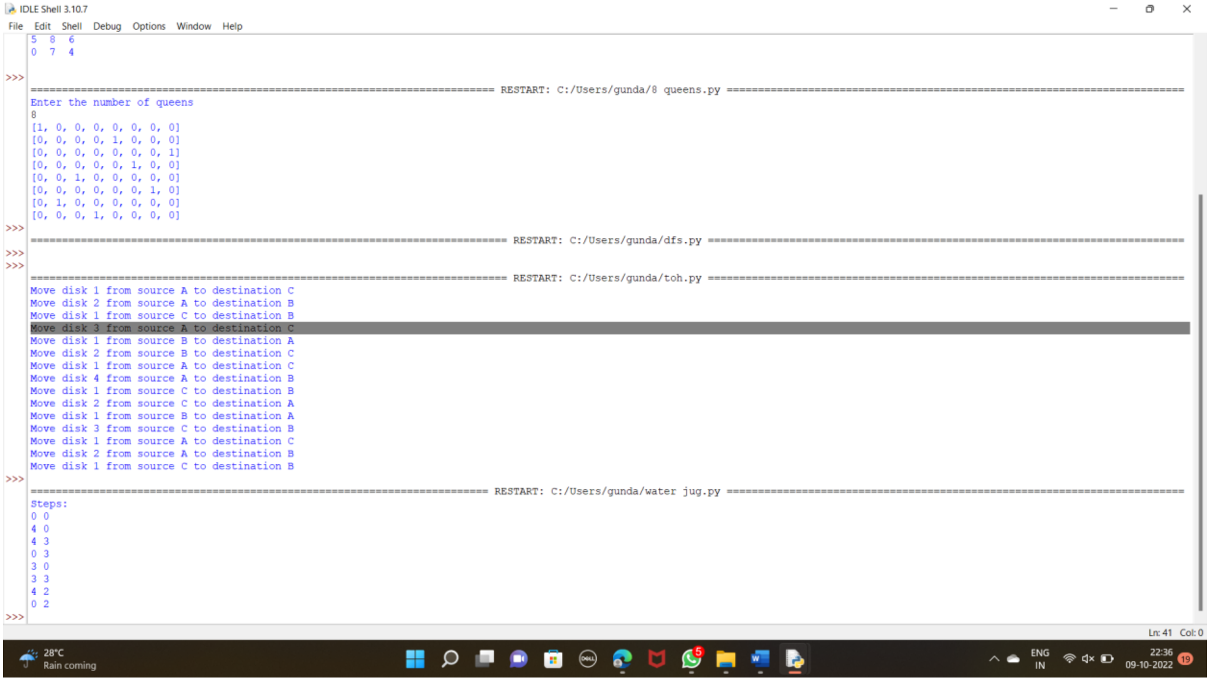


Water jug:

```
# This function is used to initialize the
# dictionary elements with a default value.
from collections import defaultdict

# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
```

```
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)

# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):

    # Checks for our goal and
    # returns true if achieved.
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    # Checks if we have already visited the
    # combination or not. If not, then it proceeds further.
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)

        # Changes the boolean value of
        # the combination as it is visited.
        visited[(amt1, amt2)] = True
```

```
        # Check for all the 6 possibilities and
        # see if a solution is found in any one of them.
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))

    # Return False if the combination is
    # already visited to avoid repetition otherwise
    # recursion will enter an infinite loop.
    else:
        return False

print("Steps: ")

# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
```



```
5  8  6
0  7  4

>>>
==================== RESTART: C:/Users/gunda/8 queens.py ====================
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]
>>>
==================== RESTART: C:/Users/gunda/dfs.py ====================
>>>
>>>
==================== RESTART: C:/Users/gunda/toh.py ====================
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
>>>
==================== RESTART: C:/Users/gunda/water jug.py ====================
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
>>>
```

Greedy first search :

```
from queue import PriorityQueue

v = 14

graph = [[] for i in range(v)]


# Function For Implementing Best First Search

# Gives output path having lowest cost



def best_first_search(actual_Src, target, n):

	visited = [False] * n

	pq = PriorityQueue()

	pq.put((0, actual_Src))

	visited[actual_Src] = True


	while pq.empty() == False:

		u = pq.get()[1]
```

```
		# Displaying the path having lowest cost
		print(u, end=" ")
		if u == target:
			break

		for v, c in graph[u]:
			if visited[v] == False:
				visited[v] = True
				pq.put((c, v))
	print()

# Function for adding edges to graph


def addedge(x, y, cost):
	graph[x].append((y, cost))
	graph[y].append((x, cost))



# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
```

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

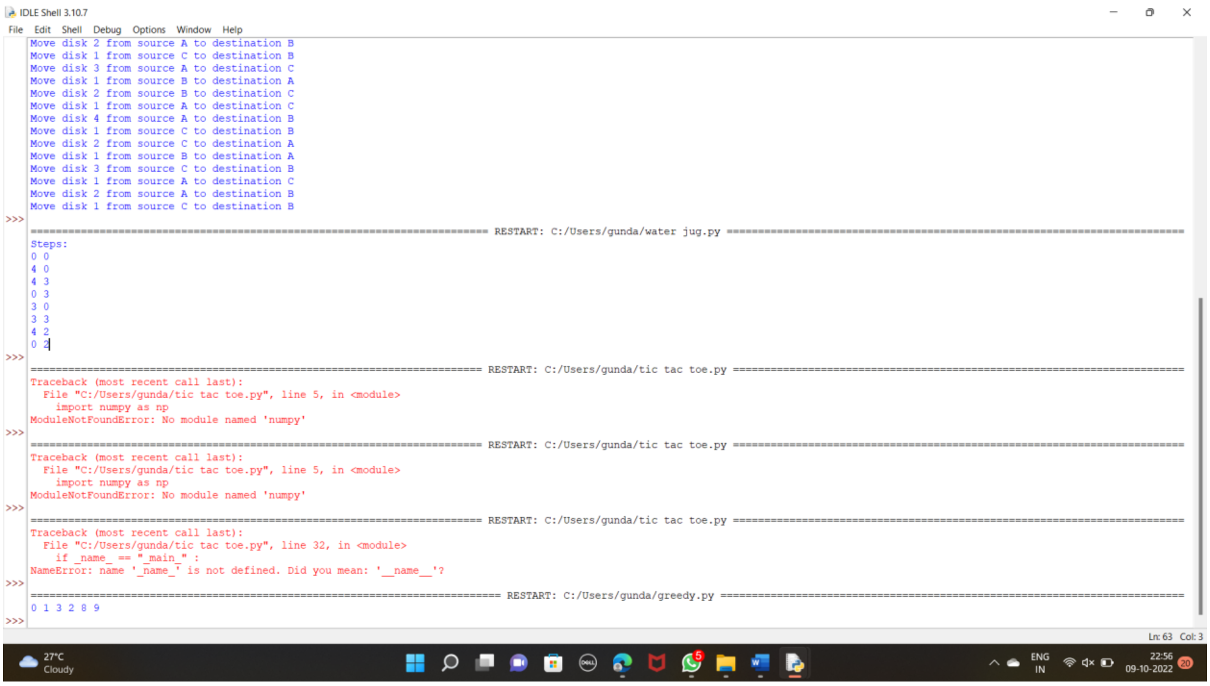
addedge(9, 13, 2)

source = 0

target = 9

best_first_search(source, target, v)

This code is contributed by Jyotheeswar Ganne



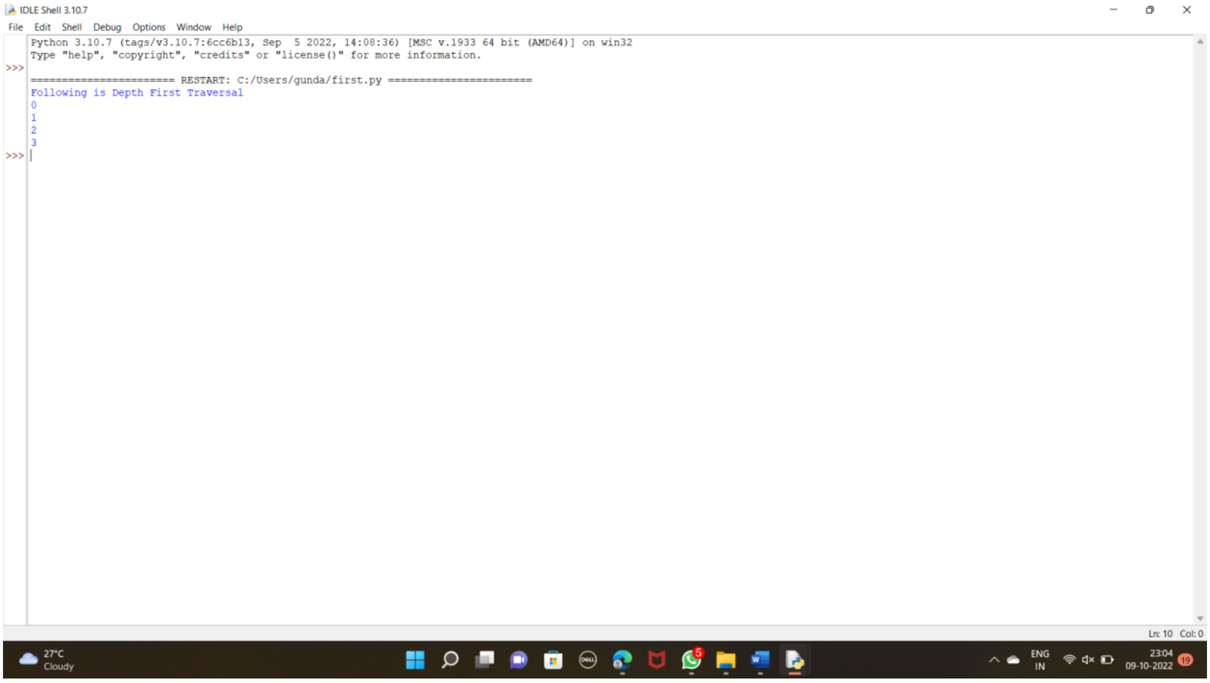
Dfs:

Python program to print DFS traversal for complete 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[v]= True
		print v,

		# Recur for all the vertices adjacent to
		# this vertex
		for i in self.graph[v]:
			if visited[i] == False:
				self.DFSUtil(i, visited)
```

```
    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self):
        V = len(self.graph) #total vertices

        # Mark all the vertices as not visited
        visited =[False]*(V)

        # Call the recursive helper function to print
        # DFS traversal starting from all vertices one
        # by one
        for i in range(V):
            if visited[i] == False:
                self.DFSUtil(i, 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 Depth First Traversal"
g.DFS()
```



Map colouring:

```
# Python3 program for the above approach

# Number of vertices in the graph
# define 4 4

# check if the colored
# graph is safe or not


def isSafe(graph, color):

    # check for every edge
    for i in range(4):
        for j in range(i + 1, 4):
            if (graph[i][j] and color[j] == color[i]):
```

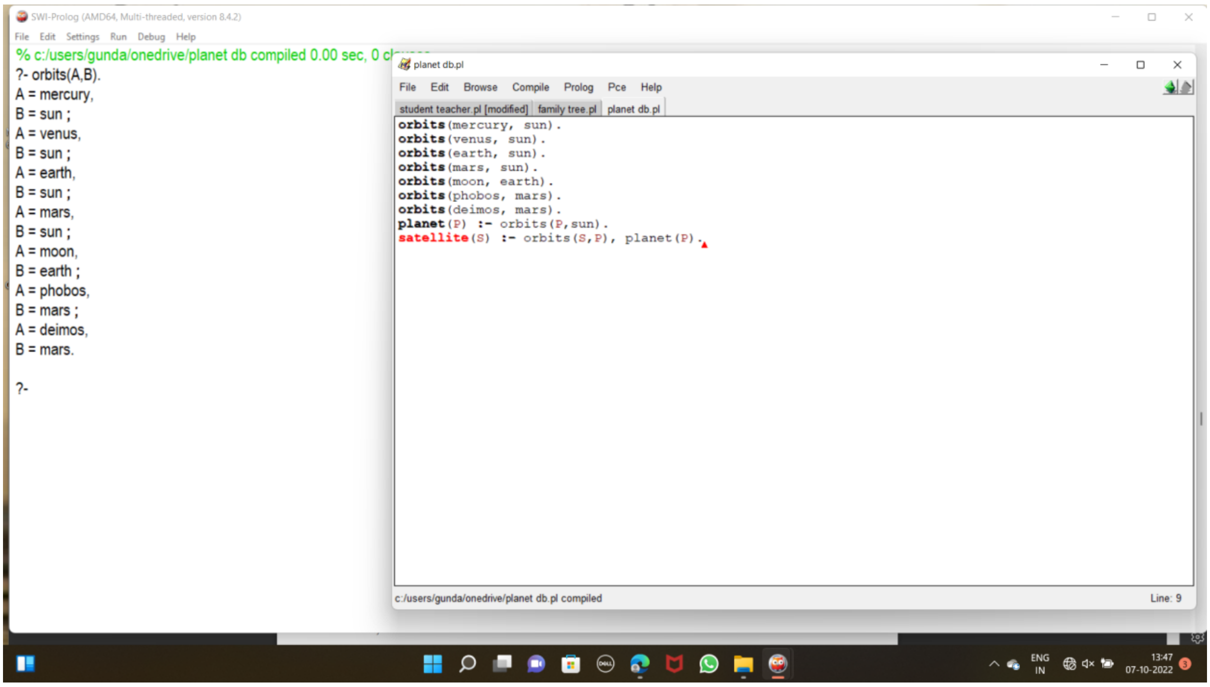
```
                        return False
        return True

# /* This function solves the m Coloring
# problem using recursion. It returns
# false if the m colours cannot be assigned,
# otherwise, return true and prints
# assignments of colours to all vertices.
# Please note that there may be more than
# one solutions, this function prints one
# of the feasible solutions.*/


def graphColoring(graph, m, i, color):

        # if current index reached end
        if (i == 4):

                # if coloring is safe
                if (isSafe(graph, color)):

                        # Print the solution
                        printSolution(color)
                        return True
                return False

        # Assign each color from 1 to m
        for j in range(1, m + 1):
                color[i] = j
```

```
        # Recur of the rest vertices
        if (graphColoring(graph, m, i + 1, color)):
            return True
        color[i] = 0
    return False

# /* A utility function to print solution */


def printSolution(color):
    print("Solution Exists:" " Following are the assigned colors ")
    for i in range(4):
        print(color[i], end=" ")



# Driver code
if __name__ == '__main__':

    # /* Create following graph and
    # test whether it is 3 colorable
    # (3)---(2)
    # | / |
    # | / |
    # | / |
    # (0)---(1)
    # */
    graph = [
        [0, 1, 1, 1],
```

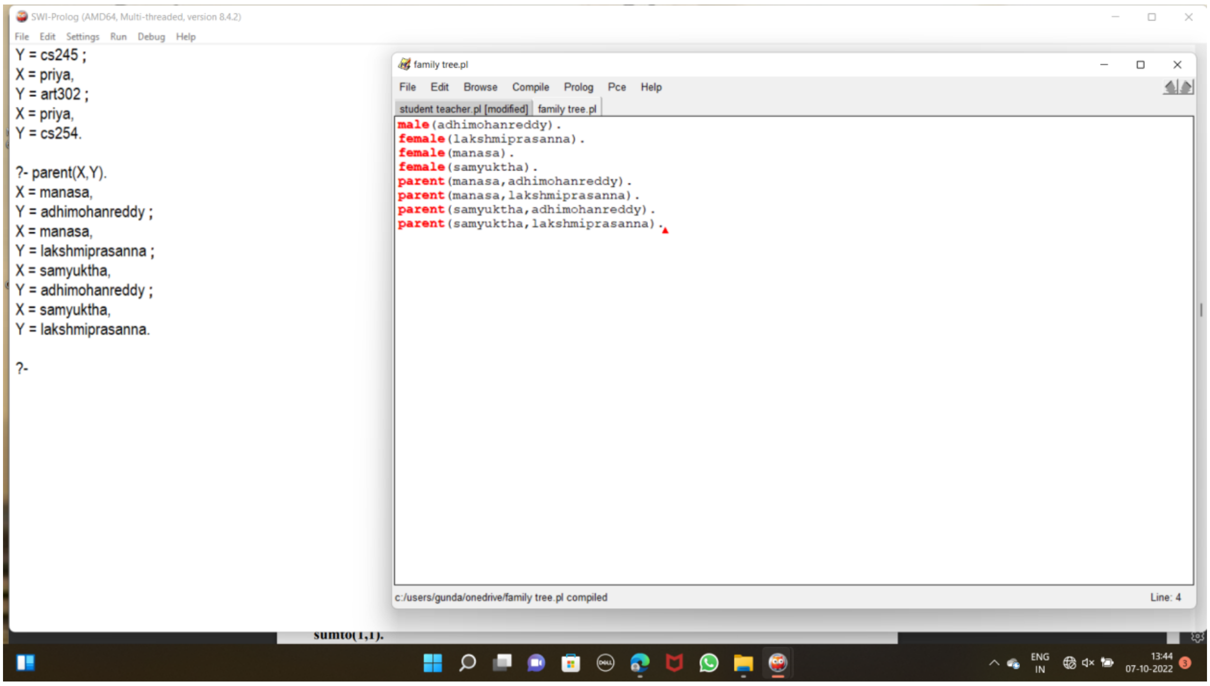
```
        [1, 0, 1, 0],
        [1, 1, 0, 1],
        [1, 0, 1, 0],
]
m = 3 # Number of colors

# Initialize all color values as 0.
# This initialization is needed
# correct functioning of isSafe()
color = [0 for i in range(4)]

# Function call
if (not graphColoring(graph, m, 0, color)):
        print("Solution does not exist")
```

Prolog programms:

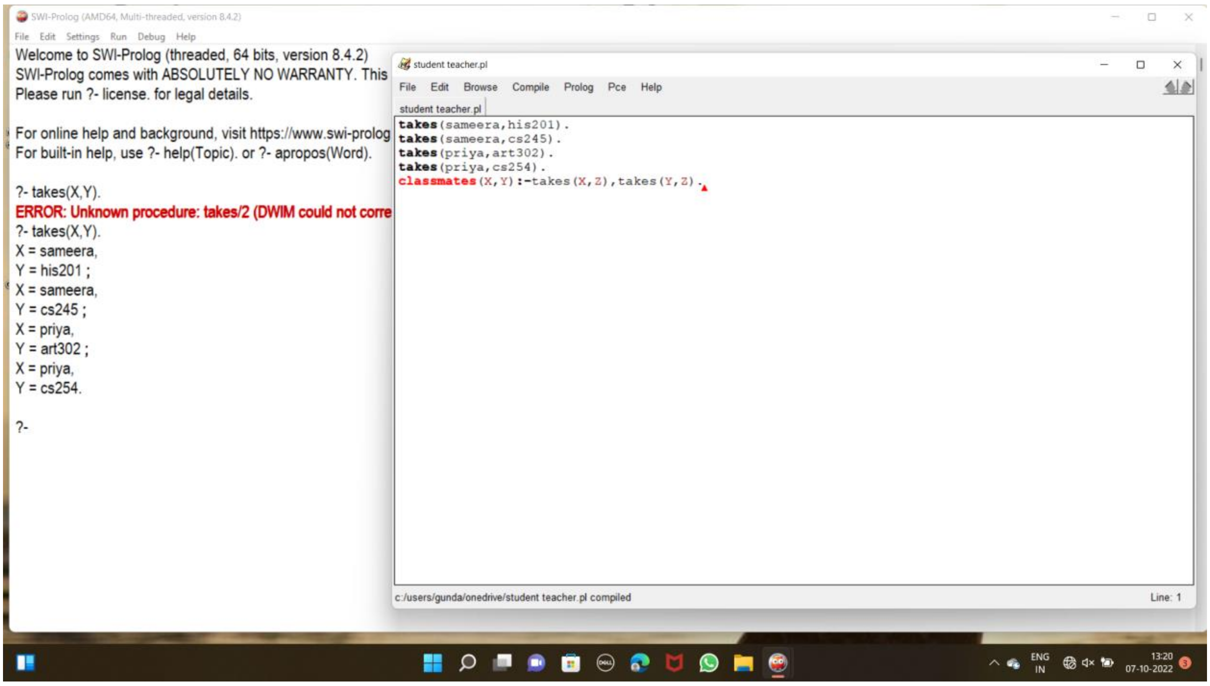
Planet db:



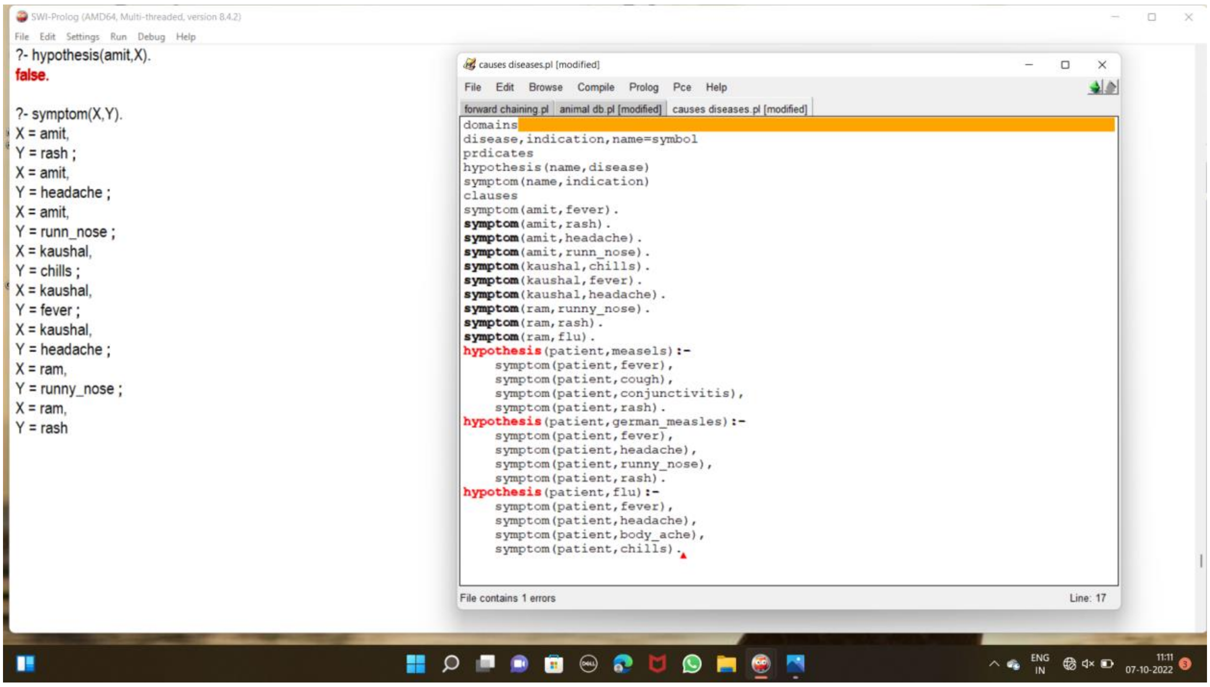
Family tree:



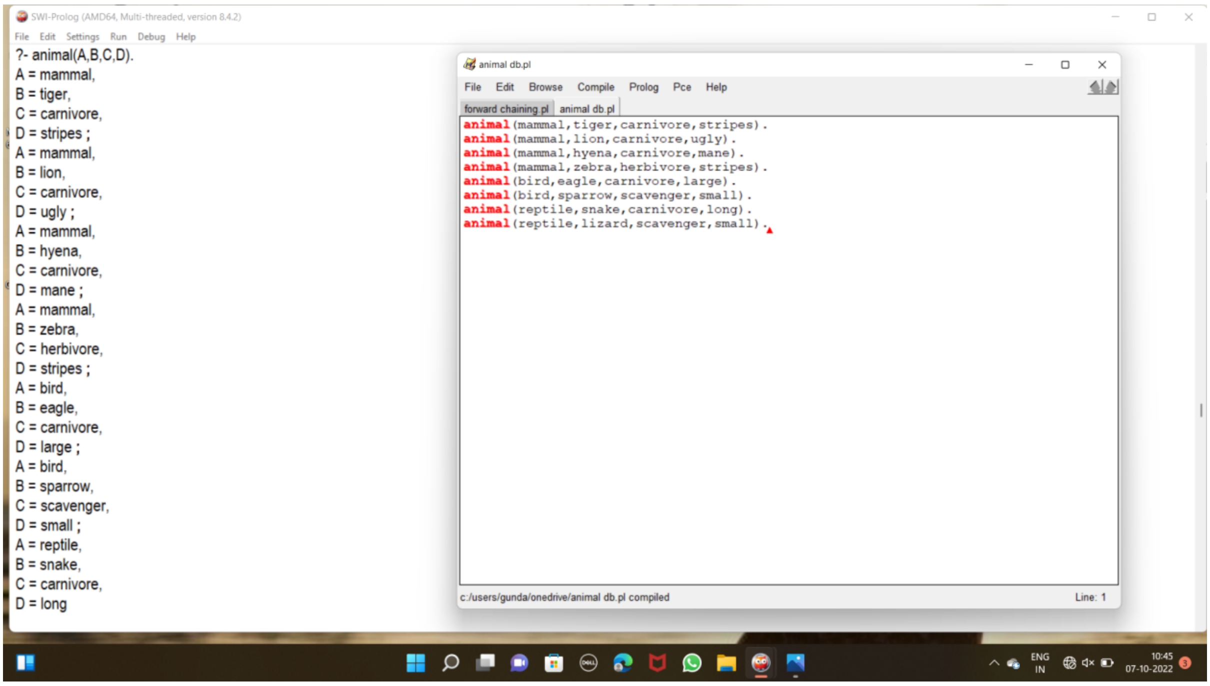
Student teacher :



Causes diseases:



Animal db:



Forward chain: