**Practical Slip AI**

**Slip 1**

**1)Python program that demonstrates the hill climbing algorithm to find the maximum of a mathematical function.(For example f(x) = -x^2 + 4x)**

import random

# Define the function to maximize

def f(x):

return -x\*\*2 + 4\*x

# Hill Climbing Algorithm

def hill\_climbing(start\_x, step\_size, max\_iterations):

current\_x = start\_x

current\_value = f(current\_x)

for \_ in range(max\_iterations):

# Generate new candidate points

new\_x = current\_x + random.uniform(-step\_size, step\_size)

new\_value = f(new\_x)

# If the new candidate is better, move to that point

if new\_value > current\_value:

current\_x, current\_value = new\_x, new\_value

return current\_x, current\_value

# Parameters

start\_x = random.uniform(0, 4) # Start between 0 and 4

step\_size = 0.1 # Step size for climbing

max\_iterations = 1000 # Number of iterations

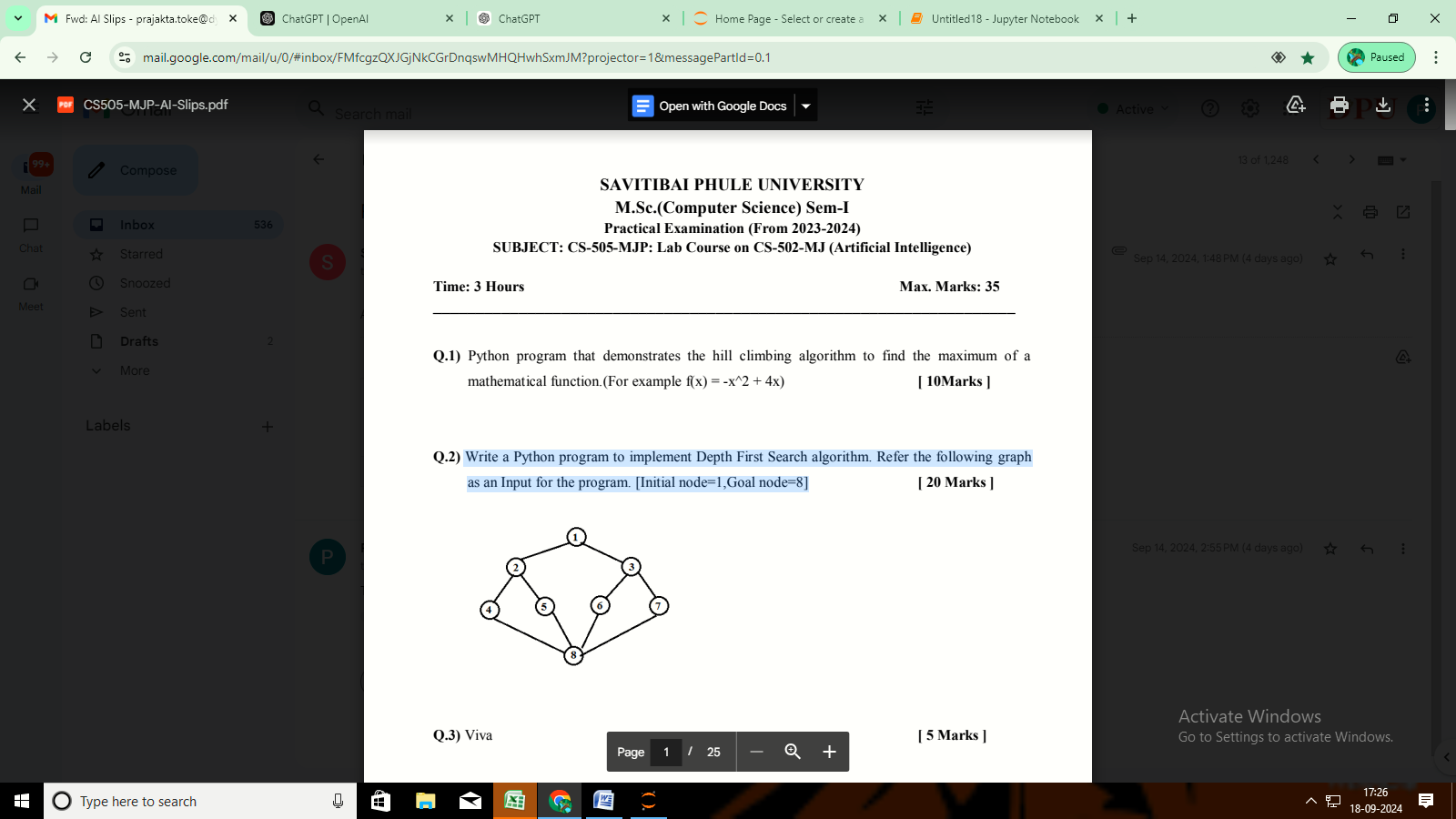
# Execute hill climbing

max\_x, max\_value = hill\_climbing(start\_x, step\_size, max\_iterations)

# Print the results

print(f"Maximum found at x = {max\_x:.2f} with value f(x) = {max\_value:.2f}")

**2)Write a Python program to implement Depth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=1,Goal node=8]**



# Graph represented as an adjacency list

graph = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6, 7],

4: [2, 8],

5: [2, 8],

6: [3, 8],

7: [3, 8],

8: [4, 5, 6, 7]

}

def dfs(graph, start, goal, visited=None):

if visited is None:

visited = set()

# Mark the current node as visited

visited.add(start)

print(f"Visiting node: {start}")

# Check if the current node is the goal

if start == goal:

print(f"Goal node {goal} found!")

return True

# Explore the neighbors

for neighbor in graph[start]:

if neighbor not in visited:

if dfs(graph, neighbor, goal, visited):

return True

return False

# Initial node and goal node

initial\_node = 1

goal\_node = 8

# Running the DFS algorithm

if not dfs(graph, initial\_node, goal\_node):

print(f"Goal node {goal\_node} not found.")

**slip2**

**1)Write a python program to generate Calendar for thegiven month and year?**

import calendar

# Function to display the calendar

def generate\_calendar(year, month):

# Create a TextCalendar instance

cal = calendar.TextCalendar()

# Generate the calendar for the specified month and year

month\_calendar = cal.formatmonth(year, month)

return month\_calendar

# Get user input

try:

year = int(input("Enter the year (e.g., 2023): "))

month = int(input("Enter the month (1-12): "))

if 1 <= month <= 12:

# Generate and print the calendar

print(generate\_calendar(year, month))

else:

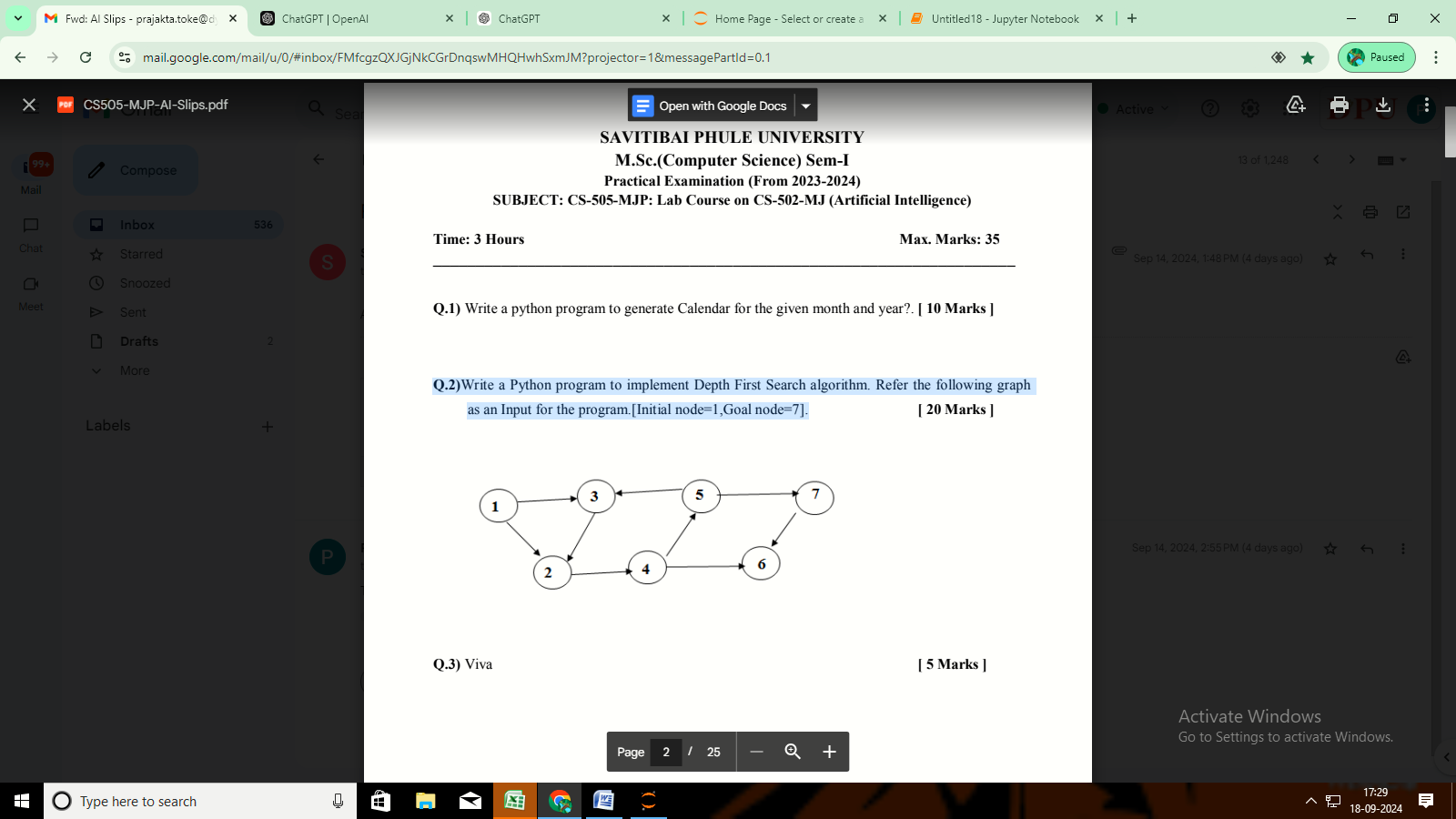
print("Please enter a valid month (1-12).")

except ValueError:

print("Invalid input. Please enter numeric values for year and month.")

**2)Write a Python program to implement Depth First Search algorithm. Refer the following graph**

**as an Input for the program.[Initial node=1,Goal node=7].**



# Graph represented as an adjacency list

graph = {

1: [2, 3],

2: [4],

3: [2],

4: [5, 6],

5: [7],

6: [],

7: [6]

}

def dfs(graph, start, goal, visited=None):

if visited is None:

visited = set()

# Mark the current node as visited

visited.add(start)

print(f"Visiting node: {start}")

# Check if the current node is the goal

if start == goal:

print(f"Goal node {goal} found!")

return True

# Explore the neighbors

for neighbor in graph[start]:

if neighbor not in visited:

if dfs(graph, neighbor, goal, visited):

return True

return False

# Initial node and goal node

initial\_node = 1

goal\_node = 7

# Running the DFS algorithm

if not dfs(graph, initial\_node, goal\_node):

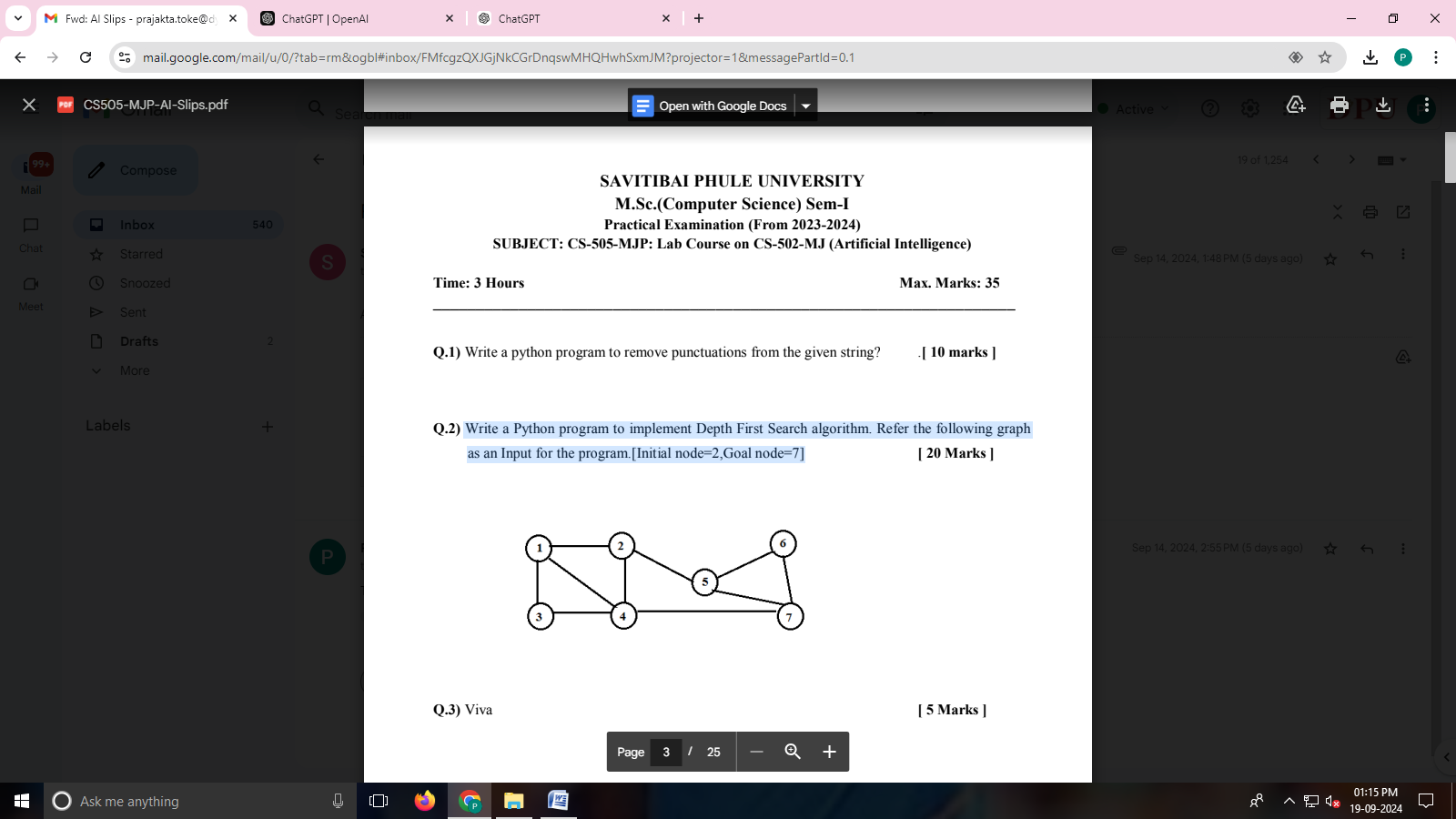
print(f"Goal node {goal\_node} not found.")

**slip3**

**1)Write a python program to remove punctuations from the given string?**

**2)Write a Python program to implement Depth First Search algorithm. Refer the following graph**

**as an Input for the program.[Initial node=2,Goal node=7]**



# Graph represented as an adjacency list

graph = {

1: [2, 3, 4],

2: [1, 4, 5],

3: [1, 4],

4: [1, 2, 3, 7],

5: [2, 6, 7],

6: [5, 7],

7: [4, 5, 6]

}

def dfs(graph, start, goal, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(f"Visiting node: {start}")

# Check if the current node is the goal

if start == goal:

print(f"Goal node {goal} found!")

return True

# Explore the neighbors

for neighbor in graph[start]:

if neighbor not in visited:

if dfs(graph, neighbor, goal, visited):

return True

return False

# Initial node and goal node

initial\_node = 2

goal\_node = 7

# Running the DFS algorithm

if not dfs(graph, initial\_node, goal\_node):

print(f"Goal node {goal\_node} not found.")

**slip4**

**1)Write a program to implement Hangman game using python.**

**Description:**

**Hangman is a classic word-guessing game. The user should guess the word correctly by**

**entering alphabets of the user choice. The Program will get input as single alphabet from the**

**user and it will matchmaking with the alphabets in the original**

import random

def choose\_word():

return random.choice(["python", "hangman", "code"])

def hangman():

word = choose\_word()

guessed\_letters = ""

tries = 6

print("Welcome to Hangman!")

while tries > 0:

display = "".join(letter if letter in guessed\_letters else "\_" for letter in word)

print("Word: ", display)

print("Guessed letters: ", guessed\_letters)

print(f"Tries left: {tries}")

guess = input("Guess a letter: ").lower()

if len(guess) != 1 or not guess.isalpha() or guess in guessed\_letters:

print("Invalid guess. Try again.")

continue

guessed\_letters += guess

if guess not in word:

tries -= 1

print(f"'{guess}' is not in the word.")

else:

print(f"Good job! '{guess}' is in the word.")

if all(letter in guessed\_letters for letter in word):

print(f"Congratulations! You guessed the word '{word}'.")

break

else:

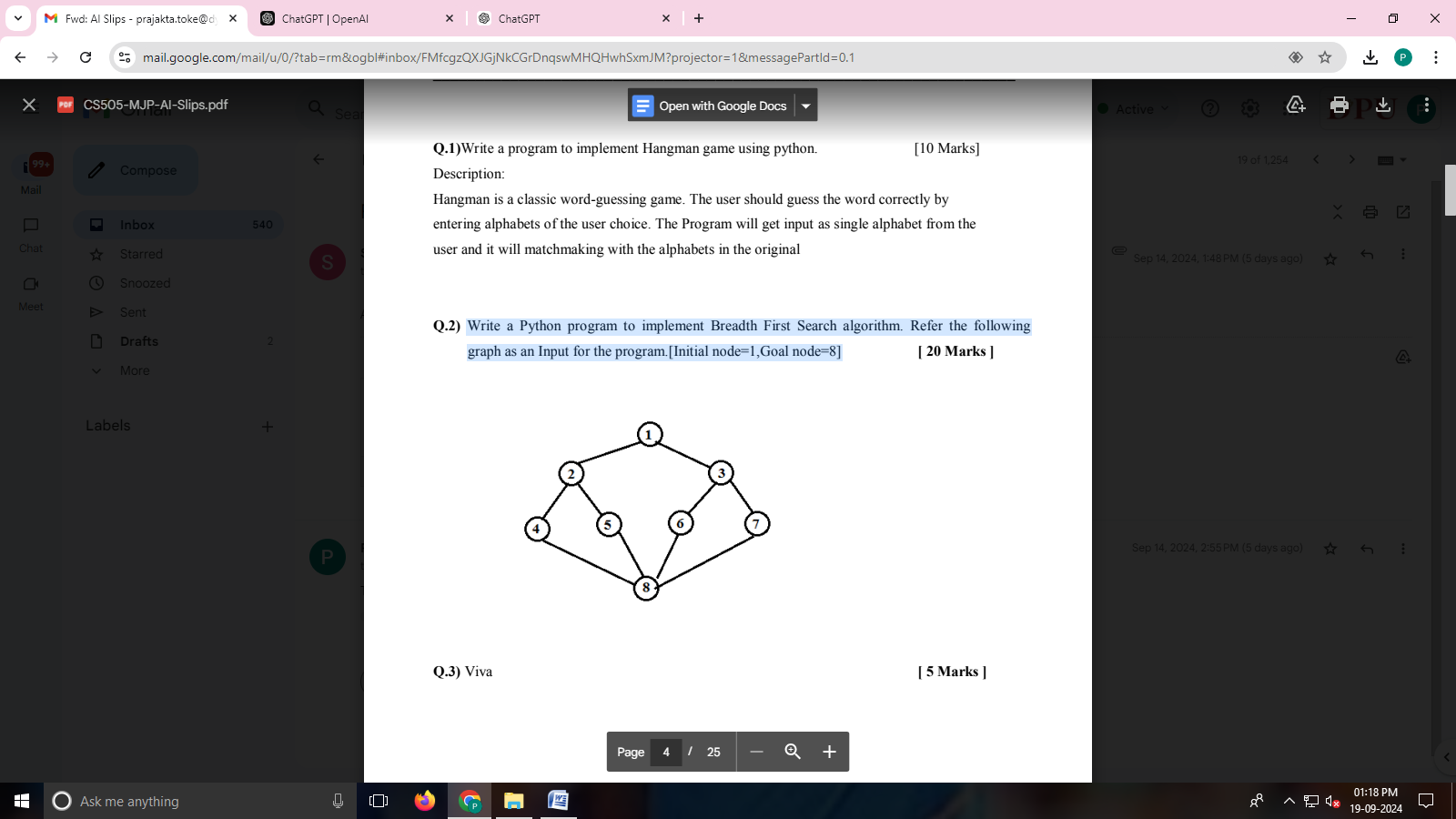
print(f"You lost! The word was '{word}'.")

# Run the Hangman game

hangman()

**2)Write a Python program to implement Breadth First Search algorithm. Refer the following**

**graph as an Input for the program.[Initial node=1,Goal node=8]**



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*graph={1:[2,3,4],2:[1,4,5],3:[1,4],4:[3,1,2,7],5:[2,6,7],6:[5,7],7:[6,5]}**

**visited=set()**

**def dfs(visited,graph,root):**

**if root not in visited:**

**print(root)**

**visited.add(root)**

**for neighbour in graph[root]:**

**dfs(visited,graph,neighbour)**

**dfs(visited,graph,2) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

from collections import deque

# Graph represented as an adjacency list

graph = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6, 7],

4: [2, 8],

5: [2, 8],

6: [3, 8],

7: [3, 8],

8: [4, 5, 6, 7]

}

def bfs(graph, start, goal):

# Queue for BFS and a set for visited nodes

queue = deque([start])

visited = set()

visited.add(start)

while queue:

current = queue.popleft()

print(f"Visiting node: {current}")

# Check if the current node is the goal

if current == goal:

print(f"Goal node {goal} found!")

return True

# Add unvisited neighbors to the queue

for neighbor in graph[current]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

print(f"Goal node {goal} not found.")

return False

# Initial node and goal node

initial\_node = 1

goal\_node = 8

# Running the BFS algorithm

bfs(graph, initial\_node, goal\_node)

**slip5**

**1)Write a python program to implement Lemmatization using NLTK**

#pip install nltk

import nltk

from nltk.stem import WordNetLemmatizer

# Download WordNet data if not already downloaded

nltk.download('wordnet')

nltk.download('punkt')

# Initialize the lemmatizer

lemmatizer = WordNetLemmatizer()

def lemmatize\_words(words):

lemmatized\_words = [lemmatizer.lemmatize(word) for word in words]

return lemmatized\_words

# Example input

input\_text = "The cats are running faster than the dog and they ran quickly."

# Tokenize the input text into words

words = nltk.word\_tokenize(input\_text)

# Perform lemmatization

lemmatized = lemmatize\_words(words)

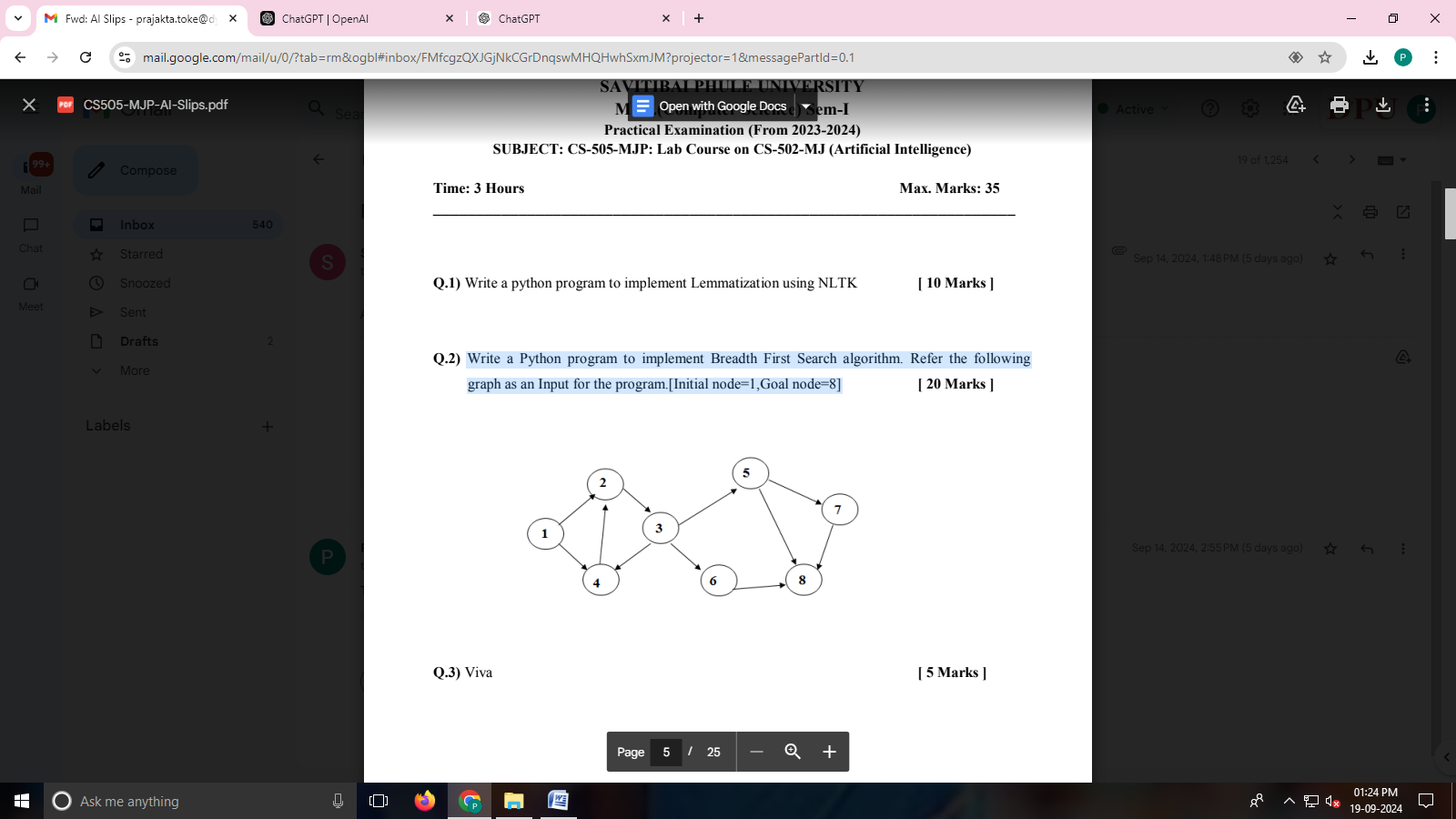
# Display the results

print("Original Words:", words)

print("Lemmatized Words:", lemmatized)

**2)Write a Python program to implement Breadth First Search algorithm. Refer the following**

**graph as an Input for the program.[Initial node=1,Goal node=8]**



from collections import deque

# Graph represented as an adjacency list

graph = {

1: [2, 4],

2: [3],

3: [4, 5, 6],

4: [2],

5: [7, 8],

6: [8],

7: [8],

8: []

}

def bfs(graph, start, goal):

# Queue for BFS and a set for visited nodes

queue = deque([start])

visited = set()

visited.add(start)

while queue:

current = queue.popleft()

print(f"Visiting node: {current}")

# Check if the current node is the goal

if current == goal:

print(f"Goal node {goal} found!")

return True

# Add unvisited neighbors to the queue

for neighbor in graph[current]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

print(f"Goal node {goal} not found.")

return False

# Initial node and goal node

initial\_node = 1

goal\_node = 8

# Running the BFS algorithm

bfs(graph, initial\_node, goal\_node)

**slip6 same que slip 5 2nd que**

**1)Write a python program to remove stop words for a given passage from a text file using NLTK?.**

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

# Download NLTK resources (uncomment if running for the first time)

nltk.download('stopwords')

nltk.download('punkt')

# Load the stop words

stop\_words = set(stopwords.words('english'))

# Function to remove stop words

def remove\_stop\_words(text):

# Tokenize the text

words = word\_tokenize(text)

# Remove stop words

filtered\_words = [word for word in words if word.lower() not in stop\_words]

return filtered\_words

# Read passage from text file

with open('passage.txt', 'r') as file:

passage = file.read()

# Remove stop words from the passage

filtered\_words = remove\_stop\_words(passage)

# Print the results

#print("Original Passage:")

#print(passage)

print("\nFiltered Words (Stop Words Removed):")

print(filtered\_words)

slip7

**1)Write a python program implement tic-tac-toe using alpha beeta pruning**

import math

# Constants for players

PLAYER\_X = 'X'

PLAYER\_O = 'O'

EMPTY = ' '

def print\_board(board):

for row in board:

print('|'.join(row))

print('-' \* 5)

def check\_winner(board):

# Check rows, columns, and diagonals for a win

for i in range(3):

if board[i][0] == board[i][1] == board[i][2] != EMPTY:

return board[i][0]

if board[0][i] == board[1][i] == board[2][i] != EMPTY:

return board[0][i]

if board[0][0] == board[1][1] == board[2][2] != EMPTY:

return board[0][0]

if board[0][2] == board[1][1] == board[2][0] != EMPTY:

return board[0][2]

return None

def is\_full(board):

return all(cell != EMPTY for row in board for cell in row)

def minimax(board, depth, alpha, beta, maximizing\_player):

winner = check\_winner(board)

if winner == PLAYER\_X:

return -1

elif winner == PLAYER\_O:

return 1

elif is\_full(board):

return 0

if maximizing\_player:

max\_eval = -math.inf

for i in range(3):

for j in range(3):

if board[i][j] == EMPTY:

board[i][j] = PLAYER\_O

eval = minimax(board, depth + 1, alpha, beta, False)

board[i][j] = EMPTY

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = math.inf

for i in range(3):

for j in range(3):

if board[i][j] == EMPTY:

board[i][j] = PLAYER\_X

eval = minimax(board, depth + 1, alpha, beta, True)

board[i][j] = EMPTY

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

def best\_move(board):

best\_eval = -math.inf

move = (-1, -1)

for i in range(3):

for j in range(3):

if board[i][j] == EMPTY:

board[i][j] = PLAYER\_O

eval = minimax(board, 0, -math.inf, math.inf, False)

board[i][j] = EMPTY

if eval > best\_eval:

best\_eval = eval

move = (i, j)

return move

def main():

board = [[EMPTY for \_ in range(3)] for \_ in range(3)]

print("Tic-Tac-Toe!")

print\_board(board)

while True:

# Player X (human)

while True:

try:

x, y = map(int, input("Enter your move (row and column): ").split())

if board[x][y] == EMPTY:

board[x][y] = PLAYER\_X

break

else:

print("Cell already occupied. Try again.")

except (ValueError, IndexError):

print("Invalid move. Please enter row and column numbers (0, 1, or 2).")

print\_board(board)

if check\_winner(board):

print("Player X wins!")

break

if is\_full(board):

print("It's a draw!")

break

# Player O (AI)

print("AI is making a move...")

move = best\_move(board)

if move != (-1, -1):

board[move[0]][move[1]] = PLAYER\_O

print\_board(board)

if check\_winner(board):

print("Player O wins!")

break

if is\_full(board):

print("It's a draw!")

break

if \_\_name\_\_ == "\_\_main\_\_":

main()

**2)Write a Python program to implement Simple Chatbot.**

class SimpleChatbot:

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

self.responses = {

"hello": "Hello! How can I assist you today?",

"how are you": "I'm just a computer program, but thanks for asking! How can I help you?",

"what is your name": "I am a simple chatbot created to assist you.",

"bye": "Goodbye! Have a great day!",

"help": "You can ask me about my name, how I'm doing, or just say hello!",

}

def get\_response(self, user\_input):

# Normalize the user input to lowercase

user\_input = user\_input.lower()

# Return the corresponding response or a default message

return self.responses.get(user\_input, "I'm sorry, I don't understand that.")

def main():

print("Welcome to the Simple Chatbot! (type 'bye' to exit)")

chatbot = SimpleChatbot()

while True:

user\_input = input("You: ")

if user\_input.lower() == "bye":

print(chatbot.get\_response(user\_input))

break

response = chatbot.get\_response(user\_input)

print("Chatbot:", response)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**slip8**

**1)Write a Python program to accept a string. Find and print the number of upper case alphabets**

**and lower case alphabets.**

def count\_case\_characters(input\_string):

upper\_case\_count = 0

lower\_case\_count = 0

for char in input\_string:

if char.isupper():

upper\_case\_count += 1

elif char.islower():

lower\_case\_count += 1

return upper\_case\_count, lower\_case\_count

def main():

# Accept input from the user

user\_input = input("Enter a string: ")

# Count uppercase and lowercase letters

upper\_count, lower\_count = count\_case\_characters(user\_input)

# Print the results

print(f"Number of uppercase letters: {upper\_count}")

print(f"Number of lowercase letters: {lower\_count}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**slip9**

**1)Write python program to solve 8 puzzle problem using A\* algorithm**

import heapq

class PuzzleState:

def \_\_init\_\_(self, board, g=0, h=0):

self.board = board

self.g = g # Cost from start to node

self.h = h # Heuristic cost to goal

self.f = g + h # Total cost

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

return self.f < other.f

def get\_blank\_position(self):

return self.board.index(0)

def is\_goal(self):

return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

def get\_possible\_moves(self):

blank\_index = self.get\_blank\_position()

moves = []

row, col = divmod(blank\_index, 3)

# Possible directions: up, down, left, right

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

for dr, dc in directions:

new\_row, new\_col = row + dr, col + dc

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_blank\_index = new\_row \* 3 + new\_col

new\_board = self.board[:]

new\_board[blank\_index], new\_board[new\_blank\_index] = new\_board[new\_blank\_index], new\_board[blank\_index]

moves.append(PuzzleState(new\_board, self.g + 1))

return moves

def heuristic(self):

# Using Manhattan distance as the heuristic. Manhattan Distance is the sum of absolute # differences between points across all the dimensions.(it is a concept of ML)

distance = 0

for index, value in enumerate(self.board):

if value != 0:

target\_index = value - 1

target\_row, target\_col = divmod(target\_index, 3)

current\_row, current\_col = divmod(index, 3)

distance += abs(target\_row - current\_row) + abs(target\_col - current\_col)

return distance

def a\_star(start\_state):

open\_set = []

closed\_set = set()

start\_node = PuzzleState(start\_state, 0, 0)

start\_node.h = start\_node.heuristic()

heapq.heappush(open\_set, start\_node)

while open\_set:

current\_node = heapq.heappop(open\_set)

if current\_node.is\_goal():

return current\_node.g # Return the number of moves

closed\_set.add(tuple(current\_node.board))

for neighbor in current\_node.get\_possible\_moves():

neighbor.h = neighbor.heuristic()

if tuple(neighbor.board) in closed\_set:

continue

# Add to open set if not already present

heapq.heappush(open\_set, neighbor)

return None # No solution found

if \_\_name\_\_ == "\_\_main\_\_":

# Initial state of the puzzle

initial\_state = [1, 2, 3, 4, 0, 5, 6, 7, 8] # Example puzzle state

moves = a\_star(initial\_state)

if moves is not None:

print(f"Solution found in {moves} moves.")

else:

print("No solution found.")

**2)Write a Python program to solve water jug problem. 2 jugs with capacity 5 gallon and 7 gallon**

**are given with unlimited water supply respectively. The target to achieve is 4 gallon of water in**

**second jug.**

from collections import deque

def water\_jug\_problem(capacity1, capacity2, target):

# Queue for BFS

queue = deque()

# Set to keep track of visited states

visited = set()

# Initial state: (jug1, jug2)

initial\_state = (0, 0)

queue.append((initial\_state, [])) # (state, path)

visited.add(initial\_state)

while queue:

(jug1, jug2), path = queue.popleft()

# Add the current state to the path

path.append((jug1, jug2))

# Check if we reached the goal

if jug2 == target:

return path

# Possible actions:

possible\_states = [

(capacity1, jug2), # Fill jug1

(jug1, capacity2), # Fill jug2

(0, jug2), # Empty jug1

(jug1, 0), # Empty jug2

(min(jug1 + jug2, capacity1), jug2 - (capacity1 - jug1) if jug1 + jug2 > capacity1 else 0), # Pour jug2 to jug1

(jug1 - (capacity2 - jug2) if jug1 + jug2 > capacity2 else 0, min(jug1 + jug2, capacity2)) # Pour jug1 to jug2

]

for state in possible\_states:

if state not in visited:

visited.add(state)

queue.append((state, path.copy()))

return None # No solution found

def print\_solution(solution):

if solution is None:

print("No solution found.")

else:

print("Steps to achieve target:")

for step in solution:

print(f"Jug 1: {step[0]} gallons, Jug 2: {step[1]} gallons")

if \_\_name\_\_ == "\_\_main\_\_":

capacity1 = 5 # Capacity of jug1

capacity2 = 7 # Capacity of jug2

target = 4 # Target amount in jug2

solution = water\_jug\_problem(capacity1, capacity2, target)

print\_solution(solution)

**slip10)**

**1)Write Python program to implement crypt arithmetic problem**

**TWO+TWO=FOUR**

from itertools import permutations

def is\_valid\_assignment(T, W, O, F, U, R):

"""Check if the assignment is valid for the equation TWO + TWO = FOUR."""

two = T \* 100 + W \* 10 + O

four = F \* 1000 + O \* 100 + U \* 10 + R

return two + two == four

def solve\_cryptarithmetic():

letters = 'TWOFRU'

unique\_letters = set(letters)

for perm in permutations(range(10), len(unique\_letters)):

# Create a mapping from letters to digits

mapping = dict(zip(unique\_letters, perm))

T = mapping['T']

W = mapping['W']

O = mapping['O']

F = mapping['F']

U = mapping['U']

R = mapping['R']

# Ensure leading digits are not zero

if T != 0 and F != 0:

if is\_valid\_assignment(T, W, O, F, U, R):

print(f"T={T}, W={W}, O={O}, F={F}, U={U}, R={R}")

print(f"{T}{W}{O} + {T}{W}{O} = {F}{O}{U}{R}")

# Run the solver

if \_\_name\_\_ == "\_\_main\_\_":

solve\_cryptarithmetic()

**Write a python program using mean end analysis algorithm problem of transforming a string of**

**lowercase letters into another string.**

def means\_end\_analysis(start: str, goal: str):

"""Transform start string into goal string using means-end analysis."""

if len(start) != len(goal):

raise ValueError("Strings must be of the same length")

current = list(start)

goal = list(goal)

steps = []

while current != goal:

print(f"Current State: {''.join(current)}")

print(f"Goal State: {''.join(goal)}")

# Find the first character that differs

for i in range(len(current)):

if current[i] != goal[i]:

# Substitute the differing character

print(f"Substituting {current[i]} with {goal[i]} at position {i}")

current[i] = goal[i]

steps.append((current[i], i)) # Keep track of steps

break # Move to the next state

print(f"Transformation complete in {len(steps)} steps.")

for step in steps:

print(f"Changed character '{step[0]}' at position {step[1]}")

if \_\_name\_\_ == "\_\_main\_\_":

start\_string = "abcde"

goal\_string = "axcye"

means\_end\_analysis(start\_string, goal\_string)

**Write a Python program to solve water jug problem. Two jugs with capacity 4 gallon and 3**

**gallon are given with unlimited water supply respectively. The target is to achieve 2 gallon of**

**water in second jug.**

from collections import deque

def water\_jug\_problem(capacity\_a, capacity\_b, target):

# Initial state: (0 gallons in jug A, 0 gallons in jug B)

initial\_state = (0, 0)

queue = deque([initial\_state])

visited = set()

visited.add(initial\_state)

while queue:

current\_a, current\_b = queue.popleft()

# Check if we've reached the goal

if current\_b == target:

print(f"Achieved {target} gallons in jug B.")

return

# Possible states to explore

possible\_states = [

(capacity\_a, current\_b), # Fill jug A

(current\_a, capacity\_b), # Fill jug B

(0, current\_b), # Empty jug A

(current\_a, 0), # Empty jug B

(current\_a - min(current\_a, capacity\_b - current\_b), current\_b + min(current\_a, capacity\_b - current\_b)), # Pour A to B

(current\_a + min(current\_b, capacity\_a - current\_a), current\_b - min(current\_b, capacity\_a - current\_a)), # Pour B to A

]

for state in possible\_states:

if state not in visited:

visited.add(state)

queue.append(state)

print("It's not possible to measure the target amount.")

# Run the water jug problem

if \_\_name\_\_ == "\_\_main\_\_":

capacity\_a = 4 # Capacity of jug A

capacity\_b = 3 # Capacity of jug B

target = 2 # Target amount in jug B

water\_jug\_problem(capacity\_a, capacity\_b, target)

**Write a Python program to simulate 4-Queens problem.**

def print\_solution(board):

"""Print the chessboard configuration."""

for row in board:

print(" ".join("Q" if col else "." for col in row))

print()

def is\_safe(board, row, col):

"""Check if a queen can be placed at board[row][col]."""

# Check this row on left side

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on left side

for i, j in zip(range(row, len(board)), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, col):

"""Utilize backtracking to solve the N-Queens problem."""

if col >= len(board):

print\_solution(board)

return True

for i in range(len(board)):

if is\_safe(board, i, col):

board[i][col] = 1 # Place queen

# Recur to place the rest of the queens

solve\_n\_queens\_util(board, col + 1)

# Backtrack

board[i][col] = 0 # Remove queen

return False

def solve\_n\_queens(n):

"""Solve the N-Queens problem and print all solutions."""

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_n\_queens\_util(board, 0):

print("No solution exists.")

if \_\_name\_\_ == "\_\_main\_\_":

n = 4 # Size of the board (4x4 for the 4-Queens problem)

solve\_n\_queens(n)

**Write a Python program to implement Mini-Max Algorithm. (Same program as tic tac toe)**

**Write a Python program to simulate 8-Queens problem.**

def print\_solution(board):

for row in board:

print(" ".join("Q" if cell else "." for cell in row))

print()

def is\_safe(board, row, col):

# Check this column on the upper side

for i in range(row):

if board[i][col]:

return False

# Check the upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if j >= 0 and board[i][j]:

return False

# Check the upper diagonal on the right side

for i, j in zip(range(row, -1, -1), range(col, len(board))):

if j < len(board) and board[i][j]:

return False

return True

def solve\_n\_queens(board, row):

if row >= len(board):

print\_solution(board)

return True # Return True to stop after finding one solution

found\_solution = False

for col in range(len(board)):

if is\_safe(board, row, col):

board[row][col] = True # Place the queen

found\_solution = solve\_n\_queens(board, row + 1) or found\_solution

board[row][col] = False # Backtrack

return found\_solution

def solve\_8\_queens():

board = [[False] \* 8 for \_ in range(8)]

solve\_n\_queens(board, 0)

if \_\_name\_\_ == "\_\_main\_\_":

solve\_8\_queens()

**Write a python program to sort the sentence in alphabetical order?**

def sort\_sentence(sentence):

# Split the sentence into words

words = sentence.split()

# Sort the words in alphabetical order

sorted\_words = sorted(words, key=str.lower) # Using key=str.lower to sort case-insensitively

# Join the sorted words back into a sentence

sorted\_sentence = ' '.join(sorted\_words)

return sorted\_sentence

if \_\_name\_\_ == "\_\_main\_\_":

input\_sentence = input("Enter a sentence: ")

sorted\_sentence = sort\_sentence(input\_sentence)

print("Sorted sentence:", sorted\_sentence)

**Write a Python program to simulate n-Queens problem.**

def print\_solution(board):

for row in board:

print(" ".join("Q" if cell else "." for cell in row))

print()

def is\_safe(board, row, col):

# Check this column on the upper side

for i in range(row):

if board[i][col]:

return False

# Check the upper diagonal on the left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if j >= 0 and board[i][j]:

return False

# Check the upper diagonal on the right side

for i, j in zip(range(row, -1, -1), range(col, len(board))):

if j < len(board) and board[i][j]:

return False

return True

def solve\_n\_queens(board, row):

if row >= len(board):

print\_solution(board)

return True # Return True to stop after finding one solution

found\_solution = False

for col in range(len(board)):

if is\_safe(board, row, col):

board[row][col] = True # Place the queen

found\_solution = solve\_n\_queens(board, row + 1) or found\_solution

board[row][col] = False # Backtrack

return found\_solution

def solve\_n\_queens\_problem(n):

board = [[False] \* n for \_ in range(n)]

solve\_n\_queens(board, 0)

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter the number of queens (n): "))

solve\_n\_queens\_problem(n)

**Write a Program to Implement Monkey Banana Problem using Python**

class State:

def \_\_init\_\_(self, monkey\_position, has\_banana):

self.monkey\_position = monkey\_position # 'ground' or 'table'

self.has\_banana = has\_banana # True or False

def \_\_str\_\_(self):

return f"Monkey is at: {self.monkey\_position}, Has banana: {self.has\_banana}"

def move\_to\_table(state):

if state.monkey\_position == 'ground':

return State('table', state.has\_banana)

return state

def move\_to\_ground(state):

if state.monkey\_position == 'table':

return State('ground', state.has\_banana)

return state

def climb(state):

if state.monkey\_position == 'ground':

return State('table', state.has\_banana)

return state

def pick\_banana(state):

if state.monkey\_position == 'table' and not state.has\_banana:

return State('table', True)

return state

def plan(state):

print("Initial State:", state)

if state.has\_banana:

print("The monkey has already picked the banana!")

return

# Plan steps to get the banana

if state.monkey\_position == 'ground':

print("Step 1: Climb to the table.")

state = climb(state)

print(state)

if state.monkey\_position == 'table':

print("Step 2: Pick up the banana.")

state = pick\_banana(state)

print(state)

if state.has\_banana:

print("The monkey has successfully picked the banana!")

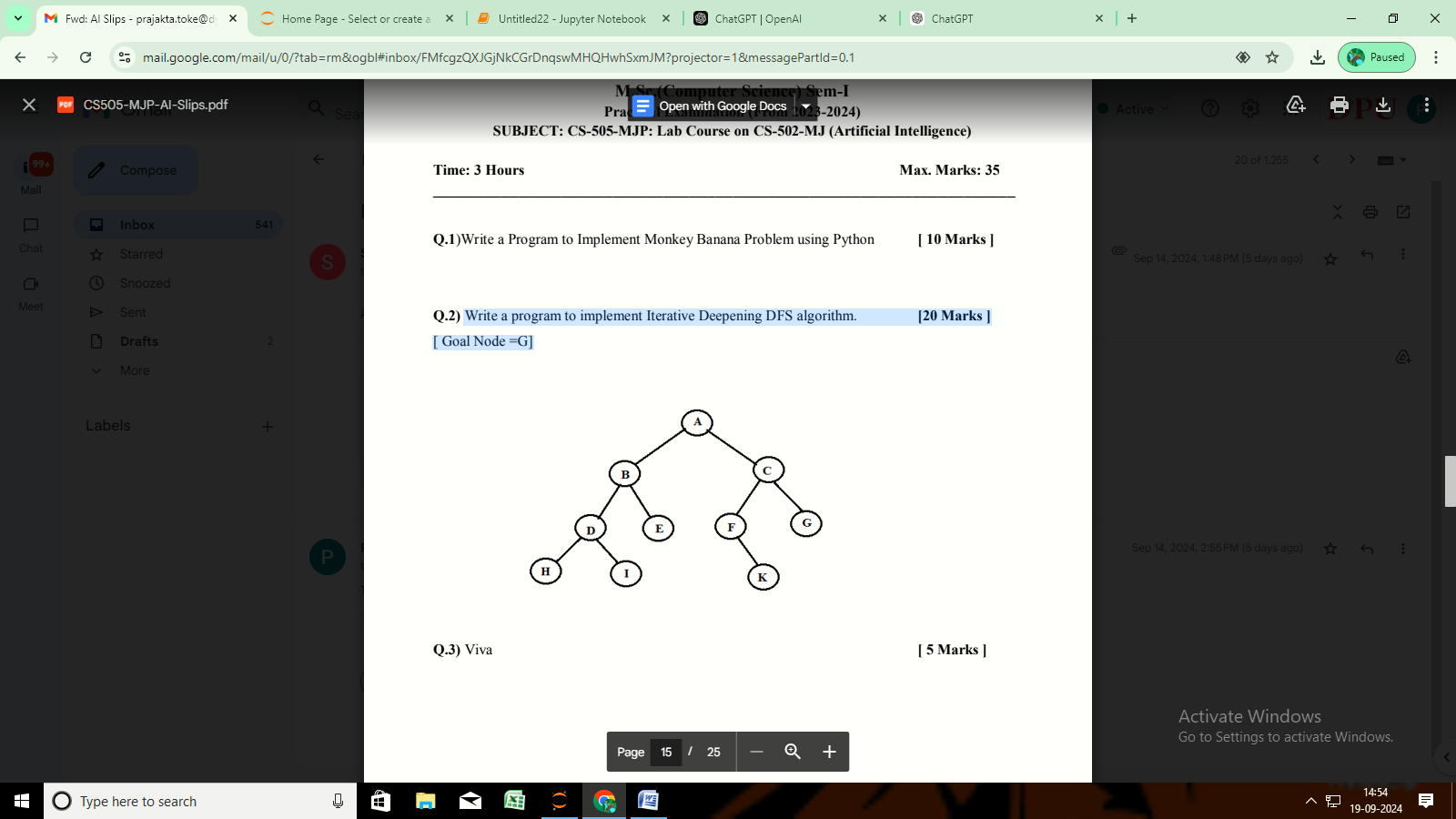
if \_\_name\_\_ == "\_\_main\_\_":

initial\_state = State('ground', False)

plan(initial\_state)

**Write a program to implement Iterative Deepening DFS algorithm.**

**[ Goal Node =G]**



def depth\_limited\_dfs(graph, node, goal, depth):

"""Performs depth-limited DFS."""

if depth == 0:

return node == goal

elif depth > 0:

for neighbor in graph.get(node, []):

if depth\_limited\_dfs(graph, neighbor, goal, depth - 1):

return True

return False

def iterative\_deepening\_dfs(graph, start, goal):

"""Iterative Deepening DFS algorithm."""

depth = 0

while True:

print(f"Searching at depth: {depth}")

if depth\_limited\_dfs(graph, start, goal, depth):

return True

depth += 1

# Example graph

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F', 'G'],

'D': ['H', 'I'],

'E': [],

'F': ['K'],

'G': [],

'H': [],

'I': [],

'K': []

}

# Starting the search

start\_node = 'A'

goal\_node = 'G'

if iterative\_deepening\_dfs(graph, start\_node, goal\_node):

print(f"Goal node '{goal\_node}' found.")

else:

print(f"Goal node '{goal\_node}' not found.")

**Write a Program to Implement Tower of Hanoi using Python**

def tower\_of\_hanoi(n, source, target, auxiliary):

"""Recursive function to solve the Tower of Hanoi problem."""

if n == 1:

print(f"Move disk 1 from {source} to {target}")

return

tower\_of\_hanoi(n - 1, source, auxiliary, target)

print(f"Move disk {n} from {source} to {target}")

tower\_of\_hanoi(n - 1, auxiliary, target, source)

if \_\_name\_\_ == "\_\_main\_\_":

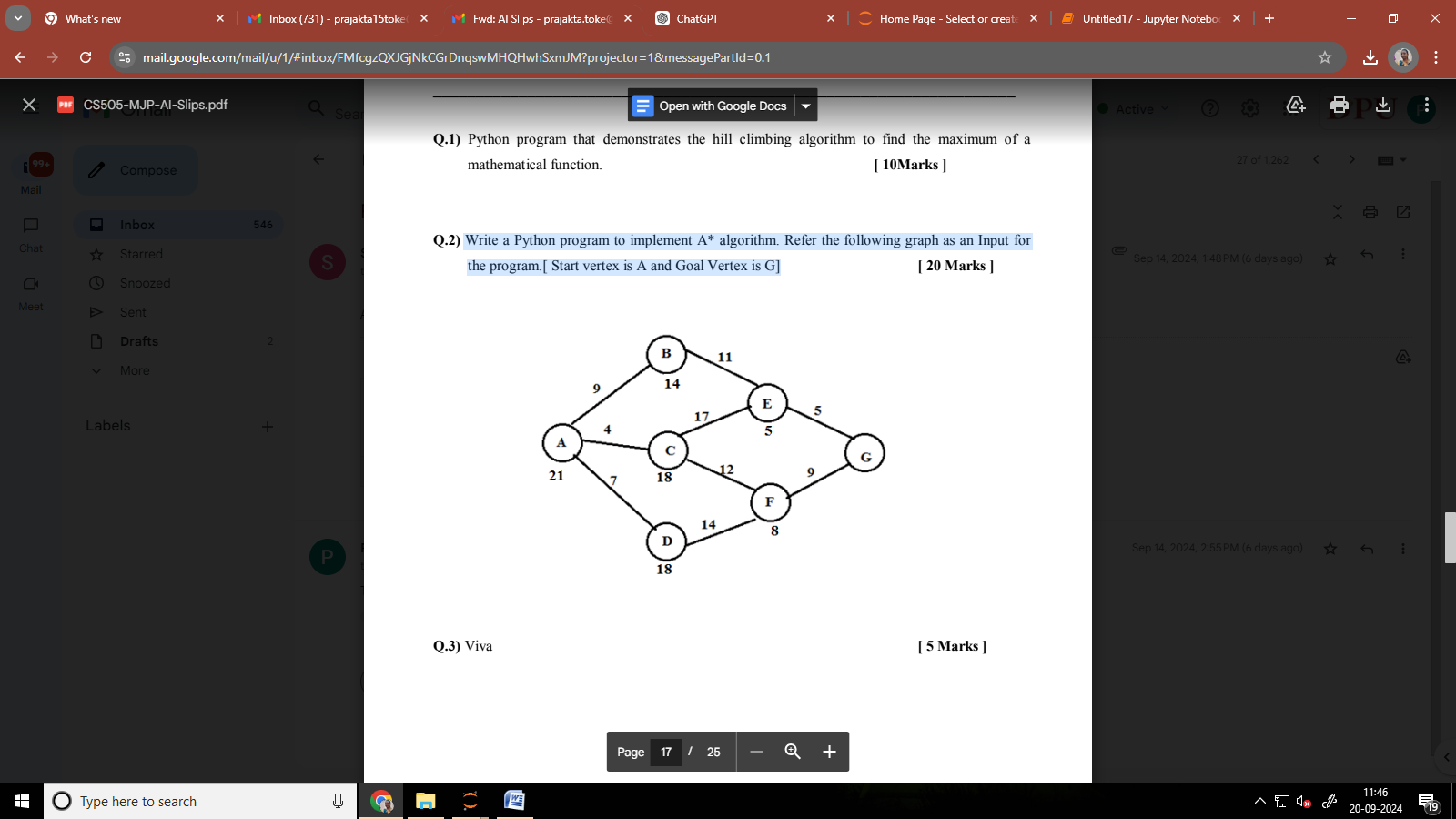
# Number of disks

n = 3 # You can change this number to solve for more disks

tower\_of\_hanoi(n, 'A', 'C', 'B') # A, B, C are the names of the rods

**Write a Python program to implement A\* algorithm. Refer the following graph as an Input for**

**the program.[ Start vertex is A and Goal Vertex is G]**



import heapq

class Node:

def \_\_init\_\_(self, name, g=0, h=0):

self.name = name

self.g = g # Cost from start to node

self.h = h # Heuristic cost to goal

self.f = g + h # Total cost

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

return self.f < other.f

def a\_star(graph, heuristics, start, goal):

open\_set = []

closed\_set = set()

start\_node = Node(start, 0, heuristics[start])

heapq.heappush(open\_set, start\_node)

while open\_set:

current\_node = heapq.heappop(open\_set)

if current\_node.name == goal:

return current\_node.g # Return the total cost to reach the goal

closed\_set.add(current\_node.name)

for neighbor, cost in graph[current\_node.name]:

if neighbor in closed\_set:

continue

g\_cost = current\_node.g + cost

h\_cost = heuristics[neighbor]

neighbor\_node = Node(neighbor, g\_cost, h\_cost)

# Add to open set if not already present or if the new path is cheaper

if all(neighbor\_node.f < existing\_node.f for existing\_node in open\_set if existing\_node.name == neighbor):

heapq.heappush(open\_set, neighbor\_node)

return None # No path found

if \_\_name\_\_ == "\_\_main\_\_":

# Graph representation

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('D', 2), ('E', 5)],

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

'D': [('B', 2), ('G', 1)],

'E': [('B', 5), ('G', 3)],

'F': [('C', 3), ('G', 5)],

'G': [('D', 1), ('E', 3), ('F', 5)]

}

# Heuristic values (straight-line distance or any other estimate)

heuristics = {

'A': 7, # Estimated cost from A to G

'B': 6, # Estimated cost from B to G

'C': 2, # Estimated cost from C to G

'D': 1, # Estimated cost from D to G

'E': 4, # Estimated cost from E to G

'F': 3, # Estimated cost from F to G

'G': 0 # Cost from G to G

}

start\_vertex = 'A'

goal\_vertex = 'G'

total\_cost = a\_star(graph, heuristics, start\_vertex, goal\_vertex)

if total\_cost is not None:

print(f"The total cost from {start\_vertex} to {goal\_vertex} is: {total\_cost}")

else:

print(f"No path found from {start\_vertex} to {goal\_vertex}.")

**Implement a system that performs arrangement of some set of objects in a room. Assume that**

**you have only 5 rectangular, 4 square-shaped objects. Use A\* approach for the placement of**

**the objects in room for efficient space utilisation. Assume suitable heuristic, and dimensions of**

**objects and rooms. (Informed Search)**

import heapq

class Object:

def \_\_init\_\_(self, width, height, shape):

self.width = width

self.height = height

self.shape = shape # 'rectangular' or 'square'

def \_\_repr\_\_(self):

return f"{self.shape}({self.width}x{self.height})"

class Room:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

self.layout = [[0] \* width for \_ in range(height)] # 0 indicates empty space

def can\_place(self, obj, x, y):

"""Check if the object can be placed at (x, y)"""

if x + obj.width > self.width or y + obj.height > self.height:

return False

for i in range(y, y + obj.height):

for j in range(x, x + obj.width):

if self.layout[i][j] != 0: # Occupied space

return False

return True

def place\_object(self, obj, x, y):

"""Place the object in the room"""

for i in range(y, y + obj.height):

for j in range(x, x + obj.width):

self.layout[i][j] = 1 # Mark space as occupied

def remove\_object(self, obj, x, y):

"""Remove the object from the room"""

for i in range(y, y + obj.height):

for j in range(x, x + obj.width):

self.layout[i][j] = 0 # Mark space as empty

def heuristic(self):

"""Calculate a simple heuristic based on remaining space"""

remaining\_space = sum(row.count(0) for row in self.layout)

return remaining\_space

class State:

def \_\_init\_\_(self, room, objects, cost=0):

self.room = room

self.objects = objects # List of remaining objects

self.cost = cost

self.heuristic\_value = room.heuristic()

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

return (self.cost + self.heuristic\_value) < (other.cost + other.heuristic\_value)

def a\_star\_arrangement(objects, room):

initial\_state = State(room, objects)

open\_set = []

heapq.heappush(open\_set, initial\_state)

while open\_set:

current\_state = heapq.heappop(open\_set)

if not current\_state.objects: # All objects placed

print("Arrangement found:")

for row in current\_state.room.layout:

print(' '.join(['X' if cell else '.' for cell in row]))

return

for i, obj in enumerate(current\_state.objects):

for y in range(current\_state.room.height):

for x in range(current\_state.room.width):

if current\_state.room.can\_place(obj, x, y):

new\_room = Room(current\_state.room.width, current\_state.room.height)

new\_room.layout = [row[:] for row in current\_state.room.layout]

new\_room.place\_object(obj, x, y)

new\_objects = current\_state.objects[:i] + current\_state.objects[i+1:] # Remaining objects

new\_state = State(new\_room, new\_objects, current\_state.cost + 1)

heapq.heappush(open\_set, new\_state)

print("No arrangement found.")

if \_\_name\_\_ == "\_\_main\_\_":

# Define objects

rectangular\_objects = [Object(2, 1, 'rectangular') for \_ in range(5)]

square\_objects = [Object(1, 1, 'square') for \_ in range(4)]

all\_objects = rectangular\_objects + square\_objects

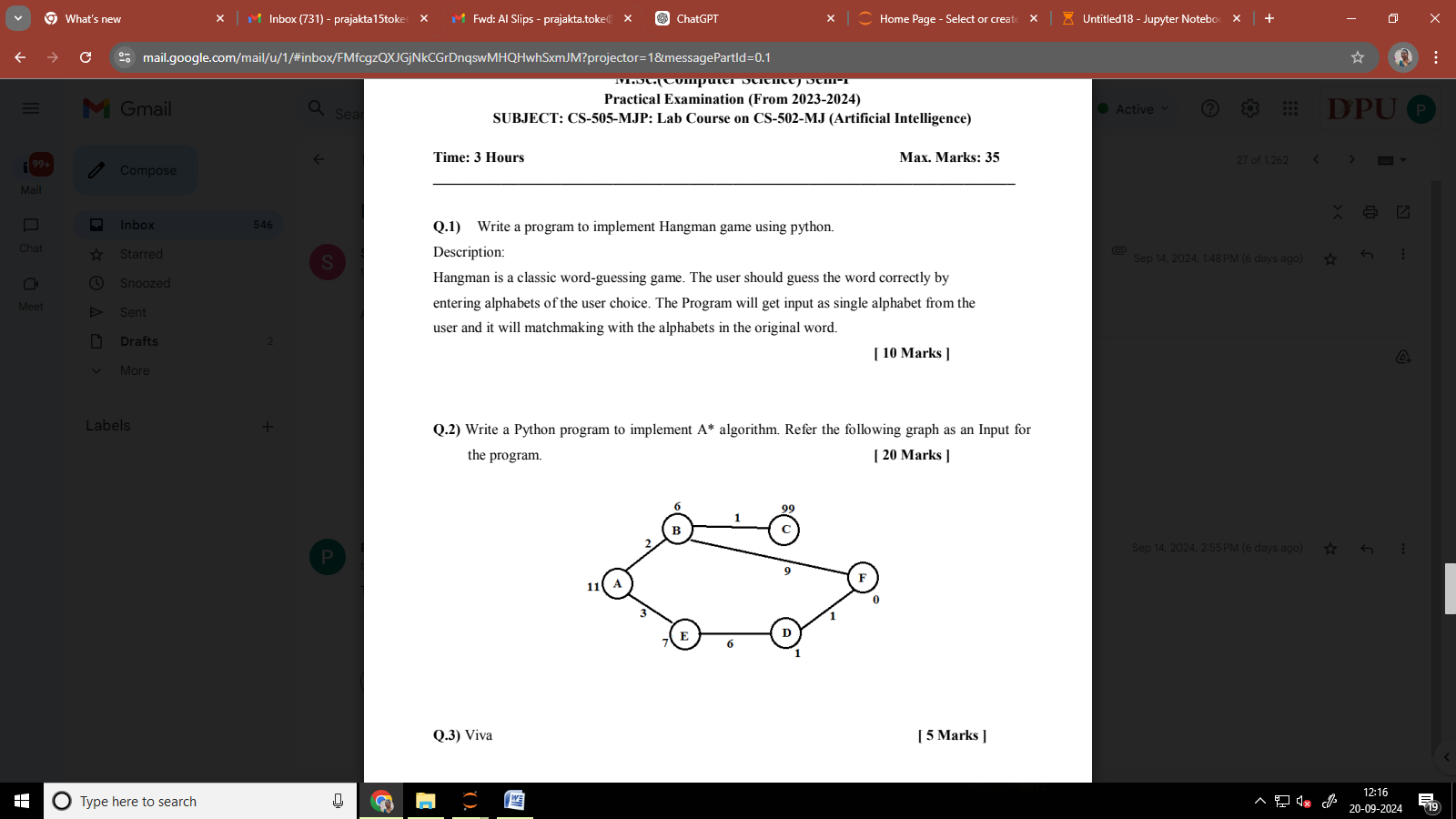
# Room dimensions (e.g., 4x4)

room = Room(4, 4)

# Run the arrangement algorithm

a\_star\_arrangement(all\_objects, room)

**Write a Python program to implement A\* algorithm. Refer the following graph as an Input for the program.**



import heapq

class Node:

def \_\_init\_\_(self, name, g=0, h=0):

self.name = name

self.g = g # Cost from start to node

self.h = h # Heuristic cost to goal

self.f = g + h # Total cost

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

return self.f < other.f

def a\_star(graph, heuristics, start, goal):

open\_set = []

closed\_set = set()

start\_node = Node(start, 0, heuristics[start])

heapq.heappush(open\_set, start\_node)

while open\_set:

current\_node = heapq.heappop(open\_set)

if current\_node.name == goal:

return current\_node.g # Return the total cost to reach the goal

closed\_set.add(current\_node.name)

for neighbor, cost in graph[current\_node.name]:

if neighbor in closed\_set:

continue

g\_cost = current\_node.g + cost

h\_cost = heuristics[neighbor]

neighbor\_node = Node(neighbor, g\_cost, h\_cost)

# Add to open set if not already present or if the new path is cheaper

if all(neighbor\_node.f < existing\_node.f for existing\_node in open\_set if existing\_node.name == neighbor):

heapq.heappush(open\_set, neighbor\_node)

return None # No path found

if \_\_name\_\_ == "\_\_main\_\_":

# Graph representation

graph = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1), ('F', 9)],

'C': [('B', 1)],

'D': [('E', 6), ('F', 1)],

'E': [('D', 6), ('A', 3)],

'F': [('D', 1), ('B', 9)]

}

# Heuristic values (estimates of the cost to reach the goal)

heuristics = {

'A': 6, # Estimated cost from A to F

'B': 5, # Estimated cost from B to F

'C': 4, # Estimated cost from C to F

'D': 2, # Estimated cost from D to F

'E': 3, # Estimated cost from E to F

'F': 0 # Cost from F to F (goal)

}

start\_vertex = 'A'

goal\_vertex = 'F'

total\_cost = a\_star(graph, heuristics, start\_vertex, goal\_vertex)

if total\_cost is not None:

print(f"The total cost from {start\_vertex} to {goal\_vertex} is: {total\_cost}")

else:

print(f"No path found from {start\_vertex} to {goal\_vertex}.")

**Build a bot which provides all the information related to you in college**

class CollegeBot:

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

self.college\_info = {

"courses": {

"BCS": "Bachelor in Computer Science",

"BCA": "Bachelor of Cmputer Application",

"MSc-CA": "Master Of Science in Computer Application",

"MSc-CS": "Master Of Science in Computer Science"

},

"faculty": {

"Dr. Sujata Patil": "Head of Computer Science",

"Prof. Shubhangi Ghule": "MSc-CS Class teacher"

},

"events": [

"Induction Day - August 1",

"Career Fair - November 15",

"Homecoming - November 20"

],

"facilities": [

"Library - Open from 8 AM to 6 PM",

"Cafeteria - Open from 7 AM to 8 PM"

]

}

def get\_courses(self):

return self.college\_info["courses"]

def get\_faculty(self):

return self.college\_info["faculty"]

def get\_events(self):

return self.college\_info["events"]

def get\_facilities(self):

return self.college\_info["facilities"]

def run(self):

print("Welcome to the College Information Bot!")

print("You can ask about courses, faculty, events, or facilities.")

while True:

query = input("\nWhat would you like to know? (type 'exit' to quit) ").strip().lower()

if query == 'exit':

print("Goodbye!")

break

elif 'courses' in query:

courses = self.get\_courses()

print("Available Courses:")

for code, name in courses.items():

print(f"{code}: {name}")

elif 'faculty' in query:

faculty = self.get\_faculty()

print("Faculty Members:")

for name, role in faculty.items():

print(f"{name}: {role}")

elif 'events' in query:

events = self.get\_events()

print("Upcoming Events:")

for event in events:

print(f"- {event}")

elif 'facilities' in query:

facilities = self.get\_facilities()

print("Campus Facilities:")

for facility in facilities:

print(f"- {facility}")

else:

print("Sorry, I didn't understand that. Please ask about courses, faculty, events, or facilities.")

if \_\_name\_\_ == "\_\_main\_\_":

bot = CollegeBot()

bot.run()

**Write a Python program for the following Cryptarithmetic problems.**

**GO + TO = OUT**

from itertools import permutations

def is\_valid\_assignment(G, O, T, U):

"""Check if the assignment is valid for the equation GO + TO = OUT."""

go = G \* 10 + O

to = T \* 10 + O

out = U \* 100 + O \* 10 + T

return go + to == out

def solve\_cryptarithmetic():

letters = 'GOTU'

for perm in permutations(range(10), len(letters)):

G, O, T, U = perm

# Ensure G and U are not zero (since they are leading digits)

if G != 0 and U != 0:

if is\_valid\_assignment(G, O, T, U):

print(f"G={G}, O={O}, T={T}, U={U}")

print(f"{G}{O} + {T}{O} = {U}{O}{T}")

# Run the solver

if \_\_name\_\_ == "\_\_main\_\_":

solve\_cryptarithmetic()

**Write a Python program for the following Cryptarithmetic problems**

**SEND + MORE = MONEY**

from itertools import permutations

def is\_valid\_assignment(S, E, N, D, M, O, R, Y):

"""Check if the assignment is valid for the equation SEND + MORE = MONEY."""

send = S \* 1000 + E \* 100 + N \* 10 + D

more = M \* 1000 + O \* 100 + R \* 10 + E

money = M \* 10000 + O \* 1000 + N \* 100 + E \* 10 + Y

return send + more == money

def solve\_cryptarithmetic():

letters = 'SENDMORY'

for perm in permutations(range(10), len(letters)):

S, E, N, D, M, O, R, Y = perm

# Ensure S and M are not zero (since they are leading digits)

if S != 0 and M != 0:

if is\_valid\_assignment(S, E, N, D, M, O, R, Y):

print(f"S={S}, E={E}, N={N}, D={D}, M={M}, O={O}, R={R}, Y={Y}")

print(f"{S}{E}{N}{D} + {M}{O}{R}{E} = {M}{O}{N}{E}{Y}")

# Run the solver

if \_\_name\_\_ == "\_\_main\_\_":

solve\_cryptarithmetic()

**Write a Python program for the following Cryptorithmetic problems**

**CROSS+ROADS = DANGER**

from itertools import permutations

def is\_valid\_assignment(C, R, O, S, A, D, N, G, E):

"""Check if the assignment is valid for the equation CROSS + ROADS = DANGER."""

cross = C \* 10000 + R \* 1000 + O \* 100 + S \* 10 + S

roads = R \* 10000 + O \* 1000 + A \* 100 + D \* 10 + S

danger = D \* 100000 + A \* 10000 + N \* 1000 + G \* 100 + E \* 10 + R

return cross + roads == danger

def solve\_cryptarithmetic():

letters = 'CROSSROADSDANGER'

unique\_letters = set(letters)

# Ensure we have at most 10 unique letters

if len(unique\_letters) > 10:

print("Too many unique letters for digits!")

return

for perm in permutations(range(10), len(unique\_letters)):

# Create a mapping from letters to digits

mapping = dict(zip(unique\_letters, perm))

C = mapping['C']

R = mapping['R']

O = mapping['O']

S = mapping['S']

A = mapping['A']

D = mapping['D']

N = mapping['N']

G = mapping['G']

E = mapping['E']

# Ensure leading digits are not zero

if C != 0 and R != 0 and D != 0:

if is\_valid\_assignment(C, R, O, S, A, D, N, G, E):

print(f"C={C}, R={R}, O={O}, S={S}, A={A}, D={D}, N={N}, G={G}, E={E}")

print(f"{C}{R}{O}{S}{S} + {R}{O}{A}{D}{S} = {D}{A}{N}{G}{E}{R}")

# Run the solver

if \_\_name\_\_ == "\_\_main\_\_":

solve\_cryptarithmetic()