First 7 slip solutions

Slip 1:

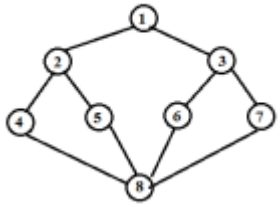
Q.1) Python program that demonstrates the hill climbing algorithm to find the maximum of a mathematical function.(For example f(x) = -x^2 + 4x)

```
import numpy as np

def objective_function(x):
    """
    The objective function to maximize.
    """
    return -x**2 + 4*x

def hill_climbing(initial_x, step_size, num_steps):
    """
    Hill climbing algorithm to find the maximum of a function.
    """
    current_x = initial_x

    for step in range(num_steps):
        current_value = objective_function(current_x)
        next_x = current_x + step_size
        next_value = objective_function(next_x)

        if next_value > current_value:
            current_x = next_x

    return current_x, objective_function(current_x)

# Set initial parameters
initial_x = 0.0
step_size = 0.1
num_steps = 100

# Run the hill climbing algorithm
result_x, result_value = hill_climbing(initial_x, step_size, num_steps)

# Print the results
print(f"Maximum found at x = {result_x}")
print(f"Maximum value is {result_value}")
```

Output:

```
Maximum found at x = 2.0000000000000004
Maximum value is 4.0
```

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]




```
# DFS algorithm in Python
# The visited variable keeps a record of the nodes explored
def dfs(graph,start,goal,stack,visited):
   stack.append(start)
   visited.append(start)
   print('The path traversed is:')
   while stack:
      # Pop from stack to set current element
      element=stack.pop()
      print(element,end=" ")
      if(element==goal):
         break
      for neighbor in graph[element]:
         if neighbor not in visited:
            stack.append(neighbor)
            visited.append(neighbor)
# A dictionary representing the illustrated graph
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']
}
start='1'
goal='8'
visited=[]
stack=[]
```

Output:

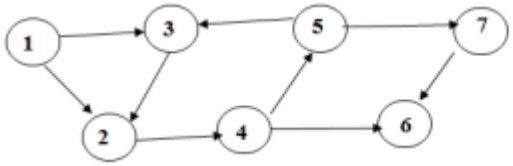
```
The path traversed is:
1 3 7 8
```

Slip 2:

Q1: Q.1) Write a python program to generate Calendar for the given month and year?. [10 Marks]

```
C:\Users\User>python
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import calendar
>>> yy=2023
>>> mm=12
>>> print(calendar.month(yy, mm))
   December 2023
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Q.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]. **[20 Marks]**




```
# DFS algorithm in Python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, start, end):
        if start not in self.graph:
            self.graph[start] = []
        self.graph[start].append(end)

def dfs(graph, current, goal, path=[]):
    path = path + [current]

    if current == goal:
        return path

    if current not in graph:
        return None

    for neighbor in graph[current]:
        if neighbor not in path:
            new_path = dfs(graph, neighbor, goal, path)
            if new_path:
                return new_path

    return None

# Example Usage:
# Create a directed graph and add edges
```

```
g = Graph()
g.add_edge('1', '2')
g.add_edge('1', '3')
g.add_edge('2', '4')
g.add_edge('3', '2')
g.add_edge('4', '5')
g.add_edge('4', '6')
g.add_edge('5', '3')
g.add_edge('5', '7')
g.add_edge('7', '6')

# Perform DFS from node '1' to '7'
initial_node = '1'
goal_node = '7'
result_path = dfs(g.graph, initial_node, goal_node)

# Display the result
if result_path:
    print(f"Path from {initial_node} to {goal_node}: {result_path}")
else:
    print(f"No path found from {initial_node} to {goal_node}")
```

Output:
Path from 1 to 7: ['1', '2', '4', '5', '7']

Slip 3:

Q.1) Write a python program to remove punctuations from the given string? .[10 marks]

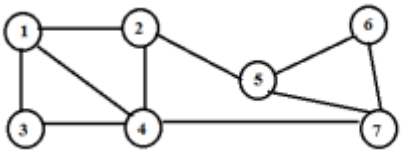
```
import string

def remove_punctuation(input_string):
    # Create a translation table with None for all punctuation characters
    translator = str.maketrans("", "", string.punctuation)

    # Use the translation table to remove punctuations from the input string
    result_string = input_string.translate(translator)

    return result_string

# Example Usage:
input_string = "Hello, world! This is an example string with punctuations!!!"
result = remove_punctuation(input_string)

print("Original String:")
print(input_string)

print("\nString after removing punctuations:")
print(result)
```

Output:

Original String:
Hello, world! This is an example string with punctuations!!!

String after removing punctuations:
Hello world This is an example string with punctuations

Q.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] **[20 Marks]**




```
# DFS algorithm in Python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node1, node2):
        # Add edges for an undirected graph
        if node1 not in self.graph:
            self.graph[node1] = []
        if node2 not in self.graph:
            self.graph[node2] = []
        self.graph[node1].append(node2)
        self.graph[node2].append(node1)

def dfs(graph, current, goal, path=[]):
    path = path + [current]

    if current == goal:
        return path

    if current not in graph:
        return None

    for neighbor in graph[current]:
        if neighbor not in path:
            new_path = dfs(graph, neighbor, goal, path)
            if new_path:
                return new_path

    return None

# Example Usage:
# Create an undirected graph and add edges
g = Graph()
g.add_edge('1', '2')
g.add_edge('1', '3')
g.add_edge('1', '4')
```

```
g.add_edge('2', '1')
g.add_edge('2', '4')
g.add_edge('2', '5')
g.add_edge('3', '1')
g.add_edge('3', '4')
g.add_edge('4', '2')
g.add_edge('4', '7')
g.add_edge('5', '2')
g.add_edge('5', '6')
g.add_edge('5', '7')
g.add_edge('6', '5')
g.add_edge('6', '7')
g.add_edge('7', '4')
g.add_edge('7', '5')
g.add_edge('7', '6')

# Perform DFS from node '2' to '7'
initial_node = '2'
goal_node = '7'
result_path = dfs(g.graph, initial_node, goal_node)

# Display the result
if result_path:
    print(f"Path from {initial_node} to {goal_node}: {result_path}")
else:
    print(f"No path found from {initial_node} to {goal_node}")
```

Output:
Path from 2 to 7: ['2', '1', '3', '4', '7']

Slip 4:

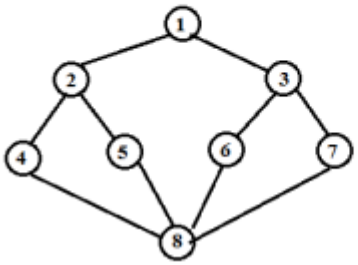
Q.1) Write a program to implement Hangman game using python. [10 Marks]

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():
    word_list = ["python", "hangman", "programming", "computer", "algorithm", "code"]
    return random.choice(word_list)

def display_word(word, guessed_letters):
    display = ""
    for letter in word:
        if letter in guessed_letters:
            display += letter
        else:
            display += "_"
```

```
    return display

def hangman():
    word_to_guess = choose_word()
    guessed_letters = []
    max_attempts = 6
    attempts = 0

    print("Welcome to Hangman!")

    while True:
        current_display = display_word(word_to_guess, guessed_letters)
        print("\nCurrent Word:", current_display)

        if current_display == word_to_guess:
            print("Congratulations! You guessed the word:", word_to_guess)
            break

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

        if guess in guessed_letters:
            print("You already guessed that letter. Try again.")
            continue

        guessed_letters.append(guess)

        if guess not in word_to_guess:
            attempts += 1
            print(f"Wrong guess! Attempts left: {max_attempts - attempts}")

            if attempts == max_attempts:
                print("Sorry, you ran out of attempts. The correct word was:", word_to_guess)
                break

        if "_" not in display_word(word_to_guess, guessed_letters):
            print("Congratulations! You guessed the word:", word_to_guess)
            break

if __name__ == "__main__":
    hangman()
```

Q.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] **[20 Marks]**




```
#BFS using python
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node1, node2):
        # Add edges for an undirected graph
        if node1 not in self.graph:
            self.graph[node1] = []
        if node2 not in self.graph:
            self.graph[node2] = []
        self.graph[node1].append(node2)
        self.graph[node2].append(node1)

def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current, path = queue.popleft()

        if current == goal:
            return path

        if current not in visited:
            visited.add(current)

            for neighbor in graph[current]:
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))

    return None

# Example Usage:
# Create an undirected graph and add edges
g = Graph()
g.add_edge('1', '2')
```

```
g.add_edge('1', '3')
g.add_edge('2', '4')
g.add_edge('2', '5')
g.add_edge('3', '6')
g.add_edge('3', '7')
g.add_edge('4', '8')
g.add_edge('5', '8')
g.add_edge('6', '8')
g.add_edge('7', '8')

# Perform BFS from node '1' to '8'
initial_node = '1'
goal_node = '8'
result_path = bfs(g.graph, initial_node, goal_node)

# Display the result
if result_path:
    print(f"Path from {initial_node} to {goal_node}: {result_path}")
else:
    print(f"No path found from {initial_node} to {goal_node}")
```

Output:

```
Path from 1 to 8: ['1', '2', '4', '8']
```

Slip 5:

Q.1) Write a python program to implement Lemmatization using NLTK [10 Marks]

To perform lemmatization using NLTK (Natural Language Toolkit) in Python, you'll need to install the NLTK library first. You can install it using:

```
pip install nltk
```

After installing NLTK, you can use the following Python program to implement lemmatization:

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus import wordnet

nltk.download('punkt')
nltk.download('wordnet')

def get_wordnet_pos(tag):
    if tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('R'):
        return wordnet.ADV
    elif tag.startswith('J'):
```

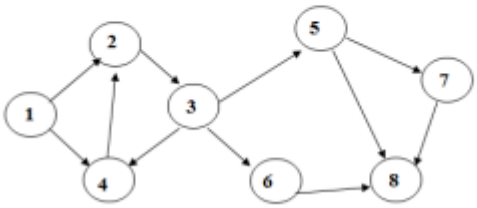
```
        return wordnet.ADJ
    else:
        return wordnet.NOUN  # Default to noun if the part of speech is not recognized

def lemmatize_text(text):
    lemmatizer = WordNetLemmatizer()
    tokens = word_tokenize(text)
    pos_tags = nltk.pos_tag(tokens)

    lemmatized_tokens = []
    for token, tag in pos_tags:
        pos = get_wordnet_pos(tag)
        lemmatized_token = lemmatizer.lemmatize(token, pos=pos)
        lemmatized_tokens.append(lemmatized_token)

    lemmatized_text = ' '.join(lemmatized_tokens)
    return lemmatized_text

# Example Usage:
input_text = "The dogs are barking loudly in the garden."
lemmatized_result = lemmatize_text(input_text)

print("Original Text:")
print(input_text)

print("\nLemmatized Text:")
print(lemmatized_result)
```

Output:

Original Text:
The dogs are barking loudly in the garden.

Lemmatized Text:
The dog be bark loudly in the garden .

Q.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] **[20 Marks]**




```
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}
```

```
    def add_edge(self, start, end):
        if start not in self.graph:
            self.graph[start] = []
        self.graph[start].append(end)

def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current, path = queue.popleft()

        if current == goal:
            return path

        if current not in visited:
            visited.add(current)

            for neighbor in graph.get(current, []):
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))

    return None

# Example Usage:
# Create a directed graph and add edges
g = Graph()
g.add_edge('1', '2')
g.add_edge('1', '4')
g.add_edge('2', '3')
g.add_edge('3', '4')
g.add_edge('3', '5')
g.add_edge('3', '6')
g.add_edge('4', '2')
g.add_edge('5', '7')
g.add_edge('5', '8')
g.add_edge('6', '8')
g.add_edge('7', '8')

# Perform BFS from node '1' to '8'
initial_node = '1'
goal_node = '8'
result_path = bfs(g.graph, initial_node, goal_node)

# Display the result
if result_path:
    print(f"Path from {initial_node} to {goal_node}: {result_path}")
else:
    print(f"No path found from {initial_node} to {goal_node}")
```

Output:

Path from 1 to 8: ['1', '2', '3', '5', '8']

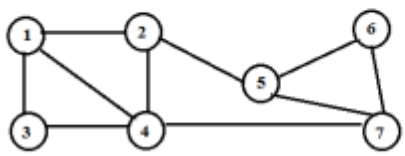
Slip 6:

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

To remove stop words from a given passage in a text file using NLTK (Natural Language Toolkit), you'll need to install the NLTK library first. You can install it using:
pip install nltk
After installing NLTK, you can use the following Python program to remove stop words from a passage in a text file:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

nltk.download('stopwords')
nltk.download('punkt')

def remove_stop_words(input_text):
    stop_words = set(stopwords.words('english'))
    words = word_tokenize(input_text)
    filtered_words = [word for word in words if word.lower() not in stop_words]
    return ' '.join(filtered_words)

def process_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        passage = file.read()
        return remove_stop_words(passage)

# Example Usage:
file_path = 'your_text_file.txt'  # Replace with the path to your text file
processed_passage = process_file(file_path)

print("Original Passage:")
print(passage)

print("\nPassage after removing stop words:")
print(processed_passage)
```

Q.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]. **[20Marks]**




```
from collections import deque
```

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node1, node2):
        # Add edges for an undirected graph
        if node1 not in self.graph:
            self.graph[node1] = []
        if node2 not in self.graph:
            self.graph[node2] = []
        self.graph[node1].append(node2)
        self.graph[node2].append(node1)

def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current, path = queue.popleft()

        if current == goal:
            return path

        if current not in visited:
            visited.add(current)

            for neighbor in graph[current]:
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))

    return None

# Example Usage:
# Create an undirected graph and add edges
g = Graph()
g.add_edge('1', '2')
g.add_edge('1', '3')
g.add_edge('1', '4')
g.add_edge('2', '1')
g.add_edge('2', '4')
g.add_edge('3', '1')
g.add_edge('3', '4')
g.add_edge('4', '1')
g.add_edge('4', '2')
g.add_edge('4', '3')
g.add_edge('5', '2')
g.add_edge('5', '6')
g.add_edge('5', '7')
g.add_edge('6', '5')
g.add_edge('6', '7')
```

```
g.add_edge('7', '5')
g.add_edge('7', '6')
g.add_edge('7', '4')

# Perform BFS from node '1' to '8'
initial_node = '1'
goal_node = '8'
result_path = bfs(g.graph, initial_node, goal_node)

# Display the result
if result_path:
   print(f"Path from {initial_node} to {goal_node}: {result_path}")
else:
   print(f"No path found from {initial_node} to {goal_node}")
```

Output:

```
No path found from 1 to 8
```

Slip 7

Q.1) Write a python program implement tic-tac-toe using alpha beeta pruning [10 Marks]

```
          import math

def print_board(board):
   for row in board:
      print(" ".join(row))
   print()

def evaluate(board):
   # Check rows, columns, and diagonals for a win
   for row in board:
      if all(cell == row[0] and cell != ' ' for cell in row):
         return 1 if row[0] == 'X' else -1

   for col in range(3):
      if all(board[row][col] == board[0][col] and board[row][col] != ' ' for row in range(3)):
         return 1 if board[0][col] == 'X' else -1

   if all(board[i][i] == board[0][0] and board[i][i] != ' ' for i in range(3)):
      return 1 if board[0][0] == 'X' else -1

   if all(board[i][2 - i] == board[0][2] and board[i][2 - i] != ' ' for i in range(3)):
      return 1 if board[0][2] == 'X' else -1

   return 0  # No winner

def is_full(board):
   return all(cell != ' ' for row in board for cell in row)

def get_empty_cells(board):
```

```
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def minimax(board, depth, alpha, beta, is_maximizing):
    score = evaluate(board)

    if score == 1:
        return score - depth

    if score == -1:
        return score + depth

    if is_full(board):
        return 0

    if is_maximizing:
        max_eval = -math.inf
        for i, j in get_empty_cells(board):
            board[i][j] = 'X'
            eval = minimax(board, depth + 1, alpha, beta, False)
            board[i][j] = ' '
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = math.inf
        for i, j in get_empty_cells(board):
            board[i][j] = 'O'
            eval = minimax(board, depth + 1, alpha, beta, True)
            board[i][j] = ' '
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)

    for i, j in get_empty_cells(board):
        board[i][j] = 'X'
        move_val = minimax(board, 0, -math.inf, math.inf, False)
        board[i][j] = ' '

        if move_val > best_val:
            best_move = (i, j)
            best_val = move_val

    return best_move
```

```
def main():
    board = [[' ' for _ in range(3)] for _ in range(3)]

    while True:
        print_board(board)

        # Player's move
        row = int(input("Enter row (0, 1, or 2): "))
        col = int(input("Enter column (0, 1, or 2): "))
        if board[row][col] != ' ':
            print("Cell already taken. Try again.")
            continue
        board[row][col] = 'O'

        # Check for player win
        if evaluate(board) == -1:
            print_board(board)
            print("You win!")
            break

        # Check for a draw
        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        # AI's move
        print("AI's move:")
        ai_row, ai_col = find_best_move(board)
        board[ai_row][ai_col] = 'X'

        # Check for AI win
        if evaluate(board) == 1:
            print_board(board)
            print("AI wins!")
            break

        # Check for a draw
        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

if __name__ == "__main__":
    main()
```

output:

```
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 2
AI's move:
X
    O
```

Q.2) Write a Python program to implement Simple Chatbot. [20 Marks]

```
import random

def simple_chatbot(user_input):
    greetings = ["hello", "hi", "hey", "greetings", "howdy"]
    goodbyes = ["bye", "goodbye", "see you", "farewell"]
    questions = ["how are you", "what's your name", "how's the weather", "tell me a joke"]

    user_input = user_input.lower()

    if any(greeting in user_input for greeting in greetings):
        return "Hello! How can I help you?"

    elif any(goodbye in user_input for goodbye in goodbyes):
        return "Goodbye! Have a great day."

    elif any(question in user_input for question in questions):
        return respond_to_question(user_input)

    else:
        return "I'm sorry, I didn't understand that. Can you please rephrase?"

def respond_to_question(question):
    if "how are you" in question:
        return "I'm just a chatbot, but I'm doing well. Thanks for asking!"

    elif "what's your name" in question:
        return "I'm a simple chatbot. You can call me ChatBot."

    elif "how's the weather" in question:
        return "I'm sorry, I don't have real-time weather information. You can check a weather
website."

    elif "tell me a joke" in question:
        jokes = ["Why did the computer go to therapy? It had too many bytes of emotional baggage!",
                 "Why don't scientists trust atoms? Because they make up everything!"]
        return random.choice(jokes)

    else:
        return "I'm not sure how to respond to that. Ask me something else!"
```

```
def main():
    print("Simple Chatbot: Hi there! Ask me anything or just say hello. Type 'exit' to end the
conversation.")

    while True:
        user_input = input("You: ")
        if user_input.lower() == 'exit':
            print("Simple Chatbot: Goodbye!")
            break

        response = simple_chatbot(user_input)
        print("Simple Chatbot:", response)

if __name__ == "__main__":
    main()
```

Output:

```
Simple Chatbot: Hi there! Ask me anything or just say hello. Type 'exit' to end the conversation.
You: How are you?
Simple Chatbot: I'm just a chatbot, but I'm doing well. Thanks for asking!
You: What is your name?
Simple Chatbot: I'm sorry, I didn't understand that. Can you please rephrase?
You:
```