***The functions from the class Graph***

def topological\_sort(self):  
 *"""  
 A function that performs a topological sort on the graph, determining in this way if it is a DAG.  
 :return: A list containing a topological order for the graph or None if the graph is not a DAG  
 """* in\_degree = {i: 0 for i in range(self.\_\_number\_of\_vertices)}  
 for v in self.parse\_vertices():  
 for neighbor in self.parse\_outbound\_edges(v):  
 in\_degree[neighbor] += 1  
  
 # Queue for vertices with no incoming edges  
 queue = deque([v for v in in\_degree if in\_degree[v] == 0])  
 top\_order = []  
  
 while queue:  
 vertex = queue.popleft()  
 top\_order.append(vertex)  
 for neighbor in self.parse\_outbound\_edges(vertex):  
 in\_degree[neighbor] -= 1  
 if in\_degree[neighbor] == 0:  
 queue.append(neighbor)  
  
 # This means there was a cycle, which shouldn't happen in a DAG  
 if len(top\_order) != self.\_\_number\_of\_vertices:  
 return None  
 return top\_order  
  
def is\_dag(self):  
 *"""  
 A function that checks if the graph is a DAG.  
 :return: True if the graph is a DAG, False otherwise  
 """* return self.topological\_sort() is not None  
  
def find\_highest\_cost\_path(self, start, end):  
 *"""  
 Time Complexity: O(number\_of\_vertices + number\_of\_edges)  
 A function that finds a highest cost path between two given vertices in a DAG.  
 :param start: The starting vertex  
 :param end: The vertex we want to reach  
 :return: The highest cost of a path and a path having that cost  
 """* top\_order = self.topological\_sort()  
 if top\_order is None:  
 return 0, []  
  
 distances = {vertex: float('-inf') for vertex in self.parse\_vertices()}  
 distances[start] = 0  
 predecessor = {vertex: None for vertex in self.parse\_vertices()}  
  
 for vertex in top\_order:  
 if distances[vertex] != float('-inf'):  
 for neighbor in self.parse\_outbound\_edges(vertex):  
 edge\_cost = self.\_\_costs[(vertex, neighbor)]  
 new\_cost = distances[vertex] + edge\_cost  
 if new\_cost > distances[neighbor]:  
 distances[neighbor] = new\_cost  
 predecessor[neighbor] = vertex  
  
 if distances[end] == float('-inf'):  
 return 0, []  
  
 path = []  
 current\_vertex = end  
 while current\_vertex is not None:  
 path.append(current\_vertex)  
 current\_vertex = predecessor[current\_vertex]  
 path.reverse()  
 return distances[end], path  
  
def count\_paths(self, start, end):  
 *"""  
 Time Complexity: O(number\_of\_vertices + number\_of\_edges)  
 A function that counts the number of paths between two given vertices in a DAG.  
 :param start: The starting vertex  
 :param end: The vertex we want to reach  
 :return: The number of paths  
 """* top\_order = self.topological\_sort()  
 if top\_order is None:  
 return 0  
  
 path\_count = [0] \* self.\_\_number\_of\_vertices  
 path\_count[start] = 1  
  
 for vertex in top\_order:  
 if path\_count[vertex] > 0:  
 for neighbor in self.parse\_outbound\_edges(vertex):  
 path\_count[neighbor] += path\_count[vertex]  
  
 return path\_count[end]  
  
def find\_lowest\_cost\_paths(self, start, end):  
 *"""  
 Time Complexity: O(number\_of\_vertices + number\_of\_edges)  
 A function that finds the number of distinct lowest cost paths between two given vertices in a DAG  
 :param start: The starting vertex  
 :param end: The vertex we want to reach  
 :return: The lowest cost of a path and the number of distinct paths having that cost  
 """* top\_order = self.topological\_sort()  
 if top\_order is None:  
 return 0, 0  
  
 distances = {vertex: float('inf') for vertex in self.parse\_vertices()}  
 path\_count = {vertex: 0 for vertex in self.parse\_vertices()}  
 distances[start] = 0  
 path\_count[start] = 1  
  
 for vertex in top\_order:  
 for neighbor in self.parse\_outbound\_edges(vertex):  
 edge\_cost = self.\_\_costs[(vertex, neighbor)]  
 new\_cost = distances[vertex] + edge\_cost  
 if new\_cost < distances[neighbor]:  
 distances[neighbor] = new\_cost  
 path\_count[neighbor] = path\_count[vertex]  
 elif new\_cost == distances[neighbor]:  
 path\_count[neighbor] += path\_count[vertex]  
  
 return distances[end], path\_count[end]

***The functions from the UI***

def check\_dag\_ui(self):  
 *"""  
 A function that checks if the current graph is a DAG and prints a topological sorting if it is a DAG.  
 :return: None, but it prints a message if the graph is not a DAG, or a topological order otherwise  
 """* if self.\_\_current is None:  
 print("No graph selected.")  
 return  
 graph = self.\_\_graphs[self.\_\_current]  
 if graph.is\_dag():  
 topological\_order = graph.topological\_sort()  
 print("The graph is a DAG. Topological order is:")  
 print(topological\_order)  
 else:  
 print("The graph is not a DAG.")  
  
def find\_highest\_cost\_path\_ui\_DAG(self):  
 *"""  
 A function that finds a highest cost path between two given vertices in a DAG and that cost.  
 :return: None, but it prints a message if the graph is not a DAG, or a highest cost path and that cost otherwise  
 """* if self.\_\_current is None:  
 print("No graph selected.")  
 return  
  
 graph = self.\_\_graphs[self.\_\_current]  
 if not graph.is\_dag():  
 print("The graph is not a DAG. Cannot find paths.")  
 return  
  
 start = int(input("Enter the start vertex: "))  
 end = int(input("Enter the end vertex: "))  
 max\_cost, path = graph.find\_highest\_cost\_path(start, end)  
 if len(path) == 0:  
 print(f"No paths available from {start} to {end}.")  
 else:  
 print(f"Maximum cost from {start} to {end} is {max\_cost} and one of the paths having this cost is:\n{path}.")  
  
def count\_paths\_ui(self):  
 *"""  
 A function that finds the total number of paths between two vertices in a DAG.  
 :return: None, but it prints a message if the graph is not a DAG, or the number of paths otherwise  
 """* if self.\_\_current is None:  
 print("No graph selected.")  
 return  
  
 graph = self.\_\_graphs[self.\_\_current]  
 if not graph.is\_dag():  
 print("The graph is not a DAG. Cannot count paths.")  
 return  
  
 start = int(input("Enter the start vertex: "))  
 end = int(input("Enter the end vertex: "))  
 num\_paths = graph.count\_paths(start, end)  
 print(f"The number of distinct paths from vertex {start} to vertex {end} is {num\_paths}.")  
  
def find\_lowest\_cost\_paths\_ui\_DAG(self):  
 *"""  
 A function that finds the number of lowest cost paths between two given vertices in a DAG and that cost.  
 :return: None, but it prints a message if the graph is not a DAG, or the number of lowest cost paths and that cost otherwise  
 """* if self.\_\_current is None:  
 print("No graph selected.")  
 return  
  
 graph = self.\_\_graphs[self.\_\_current]  
 if not graph.is\_dag():  
 print("The graph is not a DAG. Cannot find paths.")  
 return  
  
 start = int(input("Enter the start vertex: "))  
 end = int(input("Enter the end vertex: "))  
 min\_cost, count\_paths = graph.find\_lowest\_cost\_paths(start, end)  
 if count\_paths == 0:  
 print(f"No paths available from {start} to {end}.")  
 else:  
 print(f"Minimum cost from {start} to {end} is {min\_cost} with {count\_paths} distinct path(s).")