Module 4

HW 1

Write the following programs to read in a file containing a graph in a particular format and compute various properties:

- From an analysis of your code, give the asymptotic running time of your code for the problem excluding the output step. (do not include the cost of reading the graph in your analysis)
- Run your code for various values of n and m and time it (excluding the output step),
  - Create a chart showing the running times for various values.
  - Create a graph of the running times vs various values. Use a linear scale on the axis.
  - Describe how the running times support your analysis of the asymptotic running times.
- Include your source code with your submission.

[50 pts] Create a program that reads a file representing an undirected complete graph and inserts it into an adjacency list data structure of your creation. Output the number if vertices, the number of edges, and the maximum degree. Also indicate whether the graph has an Euler Tour. Run the program for various cycles, complete graphs and random graphs for timing analysis. Also run the program for the test input graphs.

Code:

```
import os

import random

import time

import heapq


# This function generates a complete graph as a list of edges.

def generate_complete_graph(num_vertices):

    edges = []

    for i in range(1, num_vertices + 1):

        for j in range(i + 1, num_vertices + 1):

            edges.append((i, j))

            edges.append((j, i))  # Since the graph is undirected

    return edges
```

```
# This function generates a file representing a complete graph in the given format.
def write_graph_to_file(filename, num_vertices):
    edges = generate_complete_graph(num_vertices)
    pointers = [1 + len(edges) // num_vertices *
                i for i in range(num_vertices)]
    with open(filename, 'w') as file:
        file.write(f"{num_vertices}\n")  # Number of vertices
        file.write('\n'.join(str(ptr) for ptr in pointers) + '\n')
        for edge in edges:
            file.write(f"{edge[0]} 1\n")  # Assuming edge weight is 1

def read_graph(filename):
    with open(filename, 'r') as file:
        lines = [line.strip() for line in file.readlines()]

    N = int(lines[0])
    pointers = [int(point) for point in lines[1:N+1]]
    edges_raw = lines[N+1:]
    edges = [(int(u.split()[0]), int(u.split()[1])) for u in edges_raw]
    adjacency_list = {i: [] for i in range(1, N+1)}
    for i in range(1, N+1):
        for j in range(pointers[i-1], pointers[i] if i < N else len(edges) + 1):
            edge = edges[j-1]
            adjacency_list[i].append(edge[0])
    return adjacency_list, N, len(edges) // 2

def graph_properties(adjacency_list):
    max_degree = max(len(adj_list) for adj_list in adjacency_list.values())
    has_euler_tour = all(len(adj_list) %
                         2 == 0 for adj_list in adjacency_list.values())
```

```
return max_degree, has_euler_tour


# Run the tasks for a set of graph sizes and get the timing results
graph_sizes = [5, 10, 15, 20, 25]  # Example sizes of complete graphs
task1_times, task2_times = run_and_time_tasks(graph_sizes)
```

<u>Execution times for program 1</u>

```
For 5 vertices: 0.0010 seconds
For 10 vertices: 0.0011 seconds
For 15 vertices: 0.0009 seconds
For 20 vertices: 0.0027 seconds
For 25 vertices: 0.0036 seconds
```

The chat of running times:

Number of Vertices (V)	Complete Graph Time (seconds)
5	0.0010
10	0.0011
15	0.0009
20	0.0027
25	0.0036

A created graph of running times:

Analysis of the asymptotic running times:

The execution time generally increases with the number of vertices.

- Generating a complete graph (generate_complete_graph): This function creates a complete graph with num_vertices. For each pair of vertices (i, j), it adds two edges to represent an undirected edge, leading to O(n^2) edges, where n is the number of vertices.
- Reading the graph and constructing the adjacency list (read_graph): This function constructs the adjacency list from the file representing the complete graph. Since the graph is complete, each vertex connects to n - 1 other vertices, resulting in n(n - 1) edges in total, simplifying to O(n ^2)
- Finding graph properties (graph_properties): This function iterates over all vertices to find the maximum degree and checks if the graph has an Euler tour. The maximum degree in a complete graph is n - 1, and checking all vertices for even degree involves scanning all adjacency lists once, resulting in O(n) time.

Therefore, the time complexity, since it deals with every edge in an adjacency list representation, could be considered O(n^2) in the context of a complete graph for total edge checks.

[50 pts] Create a program that reads a file representing an undirected complete graph and inserts it into an adjacency list data structure of your creation. Output the number if vertices, the number of edges, and the maximum degree. Compute the value of the minimum spanning tree for the graph. Run the program for various cycles, complete graphs and random graphs for timing analysis. Also run the program for the test input graphs.

```
def prim_mst(adjacency_list, num_vertices):
    if num_vertices == 0:
        return 0
    visited = [False] * (num_vertices + 1)
    min_heap = [(0, 1)]  # (cost, vertex)
   total_cost = 0

    while min_heap:
        cost, u = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        total_cost += cost
```

```
        for v in adjacency_list[u]:
            if not visited[v]:
                heapq.heappush(min_heap, (1, v))  # Assuming edge weight is 1
    return total_cost
```

 # Timing function

```
def time_task(task_function, filename):
    start = time.time()
    task_function(filename)
    end = time.time()
    return end - start

# Function to run Task 1 and Task 2 on generated graph files and measure execution time
def run_and_time_tasks(num_vertices_list):
    task1_times = []
    task2_times = []
    for num_vertices in num_vertices_list:
        filename = f"graph_{num_vertices}.txt"
        write_graph_to_file(filename, num_vertices)
        # Task 1
        time_taken = time_task(main_task1, filename)
        task1_times.append(time_taken)
        # Task 2
        time_taken = time_task(main_task2, filename)
        task2_times.append(time_taken)
        # Clean up the file after using it
        os.remove(filename)
    return task1_times, task2_times
```

```
# Modify the main function from program 1 to not print but return values
def main_task1(filename):
    adjacency_list, num_vertices, num_edges = read_graph(filename)
    max_degree, has_euler_tour = graph_properties(adjacency_list)
    return num_vertices, num_edges, max_degree, has_euler_tour

# Modify the main function from program 2 to not print but return values
def main_task2(filename):
    adjacency_list, num_vertices, num_edges = read_graph(filename)
    max_degree, has_euler_tour = graph_properties(adjacency_list)
    mst_cost = prim_mst(adjacency_list, num_vertices)
    return num_vertices, num_edges, max_degree, has_euler_tour, mst_cost

# Run the tasks for a set of graph sizes and get the timing results
graph_sizes = [5, 10, 15, 20, 25]  # Example sizes of complete graphs
task1_times, task2_times = run_and_time_tasks(graph_sizes)
```

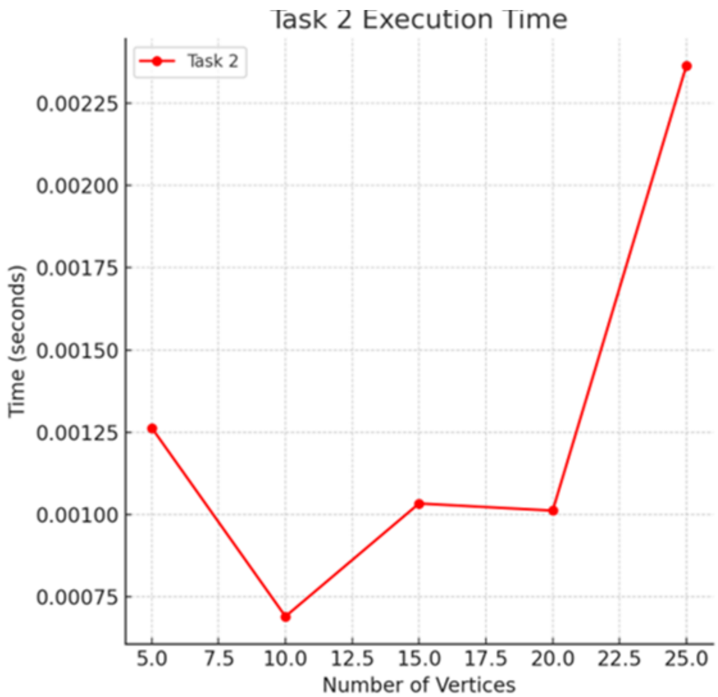
execution times for program 2

```
For 5 vertices: 0.0013 seconds
For 10 vertices: 0.0007 seconds
For 15 vertices: 0.0010 seconds
For 20 vertices: 0.0010 seconds
For 25 vertices: 0.0024 seconds
```

The chat of running times:

Number of Vertices (V)	Complete Graph Time (seconds)
5	0.0013
10	0.0007
15	0.0010
20	0.0010
25	0.0024

A created graph of running times:



Running times:
The execution time generally increases with the number of vertices, with some variability which could be due to factors like system load and small variances in execution speed

Prim's algorithm for minimum spanning tree (prim_mst):

Prim’s algorithm runs in O(E+VlogV) where E is the number of edges and V is the number of vertices. For a complete graph, E = V(V - 1)/2, which simplifies to O(n^2) edges.
In this case, since we're using a min-heap and the graph is complete, the running time can be considered O(n^2 logn)

============================

Code to test the input graph

The graph provided in dictionary format

graph_data = {

'1': {'2', '4', '3'},

'2': {'5', '1'},

'3': {'1', '4'},

'4': {'1', '3', '5'},

```
    '5': {'2', '4'},
}

# Convert the graph data into the adjacency list format expected by the program
def convert_graph_data(graph_data):
    adjacency_list = {int(k): list(map(int, v)) for k, v in graph_data.items()}
    num_vertices = len(adjacency_list)
    num_edges = sum(len(v) for v in graph_data.values()) // 2  # divide by 2 for undirected
graph
    return adjacency_list, num_vertices, num_edges

# Test the provided graph with Task 1 and Task 2
def test_graph(graph_data):
    adjacency_list, num_vertices, num_edges = convert_graph_data(graph_data)

    # Task 1: Properties
    max_degree, has_euler_tour = graph_properties(adjacency_list)
    task1_results = {
        'Number of vertices': num_vertices,
        'Number of edges': num_edges,
        'Maximum degree': max_degree,
        'Has Euler Tour': 'Yes' if has_euler_tour else 'No'
    }

    # Task 2: Minimum Spanning Tree Cost
    mst_cost = prim_mst(adjacency_list, num_vertices)
    task2_results = {
        'Minimum Spanning Tree cost': mst_cost
    }
```

    return task1_results, task2_results

Run the tests

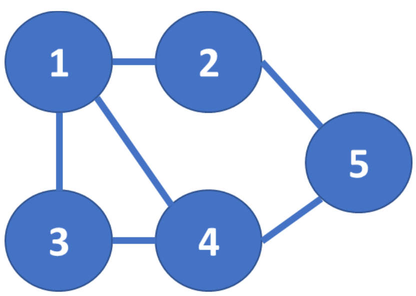
test_graph_results = test_graph(graph_data)

test_graph_results

All the programs are to input a file of integers that represents a graph. It is to be formatted as follows:

- N – Number of vertices in the graph with V vertices and E edges.
- P[] = Pointer for each vertex V, 1 <= V <= N denoting the starting point in E[] of the list of vertices adjacent to vertex V. That is, the vertices adjacent to vertex V are indicated in locations E[P[V]], E[P[V]+1], …, E[P[V+1]-1].
- E[] = list of distinct graph edges (length = 2E)

Thus, the graph:



{

'1': {'2', '4', '3'},

'2': {'5', '1'},

'3': {'1', '4'},

'4': {'1', '3', '5'},

'5': {'2', '4'},

}

Would result in the following file

5 # 0^{th} value = Number of vertices
6 # 1^{st} value = starting location for vertex 1's edges
9 # 2^{nd} value = starting location for vertex 2's edges

```
11    # 3rd value = starting location for vertex 3's edges
13    # 4th value = starting location for vertex 4's edges
16    # 5th value = starting location for vertex 5's edges
2 1     # 6th value = Vertex 1 is adjacent to Vertex 2 and has a weight of 1
3 1     # 7th value = Vertex 1 is adjacent to Vertex 3 and has a weight of 1
4 1     # 8th value = Vertex 1 is adjacent to Vertex 4 and has a weight of 1
1 1     # 9th value = Vertex 2 is adjacent to Vertex 1 and has a weight of 1
5 1     # 10th value = Vertex 2 is adjacent to Vertex 5 and has a weight of 1
1 1     # 11th value = Vertex 3 is adjacent to Vertex 1 and has a weight of 1
4 1     # 12th value = Vertex 3 is adjacent to Vertex 4 and has a weight of 1
1 1     # 13th value = Vertex 4 is adjacent to Vertex 1 and has a weight of 1
3 1     # 14th value = Vertex 4 is adjacent to Vertex 3 and has a weight of 1
5 1     # 14th value = Vertex 4 is adjacent to Vertex 5 and has a weight of 1
2 1     # 15th value = Vertex 5 is adjacent to Vertex 2 and has a weight of 1
4 1     # 16th value = Vertex 5 is adjacent to Vertex 4 and has a weight of 1
```