Module 5

[100 pts] Create a program that accepts a number of vertices “V” and a number of edges “E”.

You program should create a graph with “V” vertices and “E” edges between random pairs of vertices. The vertices should be chosen with a skewed distribution.

You should also provide for a “seed” input which establishes the beginning of your PRNG. All runs with the same seed should produce the same graphs. An option should be provided so that all edges should have a weight of 1 or all edges have a random positive integer weight with a value for the maximum weight.

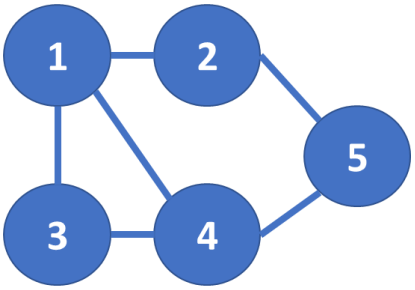
Store your graph using an adjacency list data structure of your creation and then output the graph in the format below.

Do not include the outputting of the graph in your timing analysis. Be sure to include the case where all possible edges are created as well as the case with no edges.

All the programs are to output 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:



Number of vertices (N): 5

Edges list (E[]): [(1, 2), (1, 3), (1, 4), (3, 4), (4, 5), (2,5)]

graph = {'1': set(['2', '4', '3']),

'2': set(['1', '5']),

'3': set(['1', '4']),

'4': set(['3', '1', '5']),

'5': set(['2', '4'])}

Would result in the following file

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

**Answers:**

**Details of the code:**

```
import random


def generate_graph(V, max_weight=False, seed=None, uniform_weight=True):
    if seed is not None:
        random.seed(seed)
    E= random.randint(1, V*(V-1)//2)
    # Generate the graph
    graph = {str(v): set() for v in range(1, V+1)}

    # Populate the graph with edges
    vertices = list(graph.keys())
    for _ in range(E):
        # Skewed distribution: more likely to pick vertices from the start of the
list
        # Choose a vertex with a probability proportional to its degree
        V1, V2 = random.choices(vertices, weights=[i+1 for i in range(V)], k=2)
        weight = random.randint(1, max_weight) if not uniform_weight else 1

        # Add the edge to the graph, avoiding self-loops and duplicate edges
        if V1 != V2 and V2 not in graph[V1]:
            graph[V1].add((V2, weight))
            graph[V2].add((V1, weight))  # undirected graph

    return graph


def format_graph(graph):
    output = []
    vertices = sorted(graph.keys())
    n = len(vertices)
    output.append(f"{n} # 0th value = Number of vertices")

    # Placeholder for edge start pointers/locations
    pointers = [0] * (n + 1)
    edges_list = []
    edge_descriptions = []

    for i, v in enumerate(vertices, 1):
        edges = sorted(graph[v])
        # Adjusting index to fit output format
```

```
        pointers[i] = len(edges_list) + n + 1
        for e in edges:
            # check if edge has a weigh specified
            if isinstance(e, tuple) and len(e) == 2:
                edge_str = f"{e[0]} {e[1]}"
            else:
                edge_str = f"{e} 1"  # Default weight of 1
        edges_list.append(edge_str)
        edge_desc = f"# {len(edges_list) + n}th value = Vertex {v} is adjacent to
Vertex {e[0]} and has a weight of {e[1]}"
        edge_descriptions.append(edge_desc)

    for i, p in enumerate(pointers[1:], 1):
        output.append(
            f"{p} # {i}st value = starting location for vertex {i}’s edges")

    for i, edge in enumerate(edges_list, n + 1):
        output.append(f"{edge} {edge_descriptions[i - n - 1]}")

    return '\n'.join(output)
```

```
graph = {
    '1': {('2', 1), ('4', 1), ('3', 1)},
    '2': {('1', 1), ('5', 1)},
    '3': {('1', 1), ('4', 1)},
    '4': {('3', 1), ('1', 1), ('5', 1)},
    '5': {('2', 1), ('4', 1)}
}

output_grath = format_graph(graph)

print(output_grath)
```

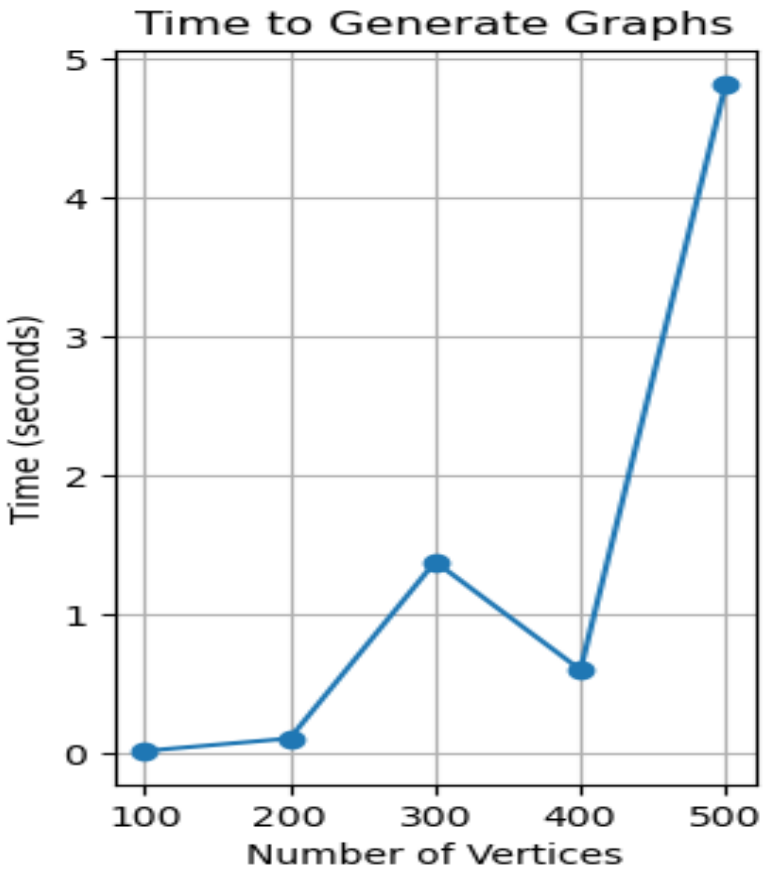
**Output:**

```
5 # 0th value = Number of vertices
6 # 1st value = starting location for vertex 1’s edges
7 # 2st value = starting location for vertex 2’s edges
8 # 3st value = starting location for vertex 3’s edges
9 # 4st value = starting location for vertex 4’s edges
10 # 5st value = starting location for vertex 5’s edges
4 1 # 6th value = Vertex 1 is adjacent to Vertex 4 and has a weight of 1
5 1 # 7th value = Vertex 2 is adjacent to Vertex 5 and has a weight of 1
4 1 # 8th value = Vertex 3 is adjacent to Vertex 4 and has a weight of 1
5 1 # 9th value = Vertex 4 is adjacent to Vertex 5 and has a weight of 1
4 1 # 10th value = Vertex 5 is adjacent to Vertex 4 and has a weight of 1
```

**Chart for Running Time**

| Number of Vertices (V) | Complete Graph Time (seconds) |
|---|---|
| 100 | 0.021 |
| 200 | 0.111 |
| 300 | 1.382 |
| 400 | 0.604 |
| 500 | 4.816 |

**A graph of the running times vs various values. Use a linear scale on the axis:**



**Asymptotic running time analysis:**

Analyzing the asymptotic running time for the program that generates a graph and prepares it in a specific format, without considering the outputting part, involves looking at:

1. Graph generation:
    - Involves initializing the graph structure and populating it with edges.
    - Initialization of V vertices is O(V).
    - Edge insertion is performed E times. In the worst case, where the graph is fully connected and undirected, each vertex can connect to every other vertex, leading to E= $\frac{v(v-1)}{2}$
    - in a fully connected graph. However, edge insertion in a set is typically O(1) assuming a good hash function.
2. Graph processing for adjacency list format (ignoring output generation):
    Sorting the vertices, if required for processing, would take O(VlogV).
    Iterating through vertices to prepare the adjacency list involves going through each edge twice in an undirected graph, but since we're analyzing based on operations inside the main loop, this is more about how many times we are accessing and inserting into data structures. Each edge insertion into the adjacency list is O(1) operation, but we perform it for every edge.
    Considering these points:
        - In the no-edge case (E=0), the time complexity is O(V) because we only deal with vertices and no edge insertions happen.
        - In the fully connected graph case, the time complexity for generating and processing edges becomes significant. Edge generation and processing in this case, because of the nature of iteration over E possible edges, lead to O(E) complexity. Since E can be as large as E= $\frac{v(v-1)}{2}$ in a complete graph, the complexity in terms of V would be O(V^2)

Thus, the time complexity of the program is:

- O(V) for the case with no edges (dominated by vertex initialization).
- O(V^2) for the case with a fully connected graph case, where every vertex is connected every other vertex.