Module 3

HW 1

Write the following programs to create graphs and write them to a file in a particular format:

- From an analysis of your code, give the asymptotic running time of your code for the problem excluding the output step.
- 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.

[25 pts] Create a program that accepts a number of vertices "V", creates an undirected complete graph with "V" vertices using an adjacency list data structure of your creation and then output the graph in the format below. All edges should have a weight of 1. Do not include the outputting of the graph in your timing analysis.

Code:

```
def create_complete_graph(V):
  adjacency_list = {}
  for i in range(1, V + 1):
    adjacency_list[i] = [j for j in range(1, V + 1) if j != i]
  return adjacency_list


def output_complete_graph(V):
  graph = create_complete_graph(V)
  pointers = []
  edges = []
  edge_count = 0
  for i in range(1, V + 1):
    pointers.append(edge_count + 1)
    for j in graph[i]:
```

```
            edges.append((j, 1)) # Guarantees that all edges will have a weight of 1.
            edge_count += 1
    pointers.append(edge_count + 1)  # For the endpoint
    return V, pointers, edges


def time_complete_graph(V):
    start = time.time()
    create_complete_graph(V)
    end = time.time()
    return end - start


V_values = [1000, 2000, 3000, 4000, 5000]
complete_times = [round(time_complete_graph(V), 3)for V in V_values]
print(complete_times)
```

<u>output:</u>

```
def time_complete_graph(V):
    start = time.time()
    create_complete_graph(V)
    end = time.time()
    return end - start

V_values = [1000, 2000, 3000, 4000, 5000]
complete_times = [round(time_complete_graph(V), 3)for V in V_values]
print(complete_times)
```

```
35]   ✓  5.2s
```

```
[0.091, 0.381, 0.817, 1.452, 2.485]
```

<u>Output a life that represents a graph:</u>

```
filename = "complete_graph.txt"
V_values = [10, 20, 30, 40, 50]
with open(filename, 'w') as file:
    for V in V_values:
```

```
    n, p, e = output_complete_graph(V)  # Corrected this line
    file.write(f"{n} # 0th value = Number of vertices\n")

    # Correcting the ordinal indicators and writing the pointers
    for index, pointer in enumerate(p):
        ordinal_indicator = 'th' if 11 <= (
            index + 1) % 100 <= 13 else {1: 'st', 2: 'nd', 3: 'rd'}.get((index + 1) % 10, 'th')
        file.write(
            f"{pointer} # {index + 1}{ordinal_indicator} value = starting location for vertex {index}'s edges\n")

    # Writing the edges with correct descriptions
    for index, edge in enumerate(e):
        description_index = index + len(p)
        # Correctly identifies the from-vertex
        vertex_from = ((index // 2) % V) + 1
        file.write(
            f"{edge[0]} {edge[1]} # {description_index}th value = Vertex {vertex_from} is adjacent to Vertex {edge[0]} and has a weight of {edge[1]}\n")
```

Part of the output file content:

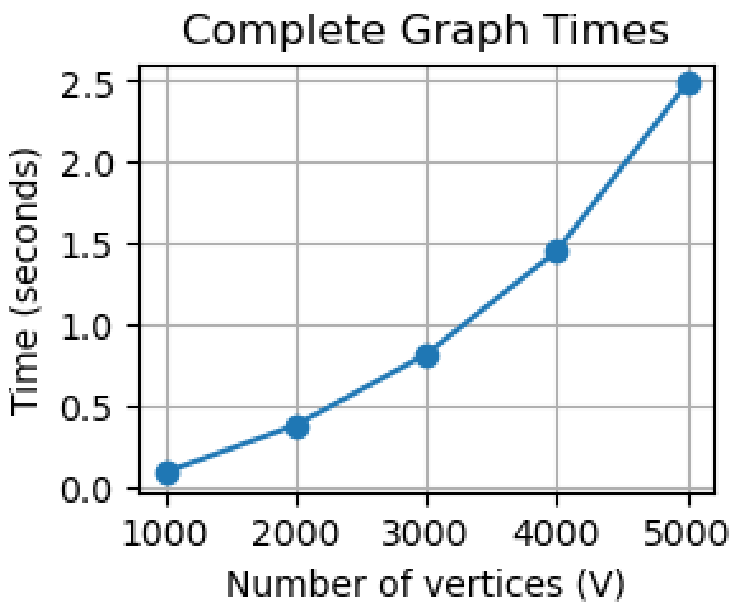
ignments > Module 3 > complete_graph.txt

```
10 # 0th value = Number of vertices
1 # 1st value = starting location for vertex 0's edges
10 # 2nd value = starting location for vertex 1's edges
19 # 3rd value = starting location for vertex 2's edges
28 # 4th value = starting location for vertex 3's edges
37 # 5th value = starting location for vertex 4's edges
46 # 6th value = starting location for vertex 5's edges
55 # 7th value = starting location for vertex 6's edges
64 # 8th value = starting location for vertex 7's edges
73 # 9th value = starting location for vertex 8's edges
82 # 10th value = starting location for vertex 9's edges
91 # 11th value = starting location for vertex 10's edges
2 1 # 11th value = Vertex 1 is adjacent to Vertex 2 and has a weig
3 1 # 12th value = Vertex 1 is adjacent to Vertex 3 and has a weig
4 1 # 13th value = Vertex 2 is adjacent to Vertex 4 and has a weig
5 1 # 14th value = Vertex 2 is adjacent to Vertex 5 and has a weig
6 1 # 15th value = Vertex 3 is adjacent to Vertex 6 and has a weig
```

The chat of running times:

Number of Vertices (V)	Complete Graph Time (seconds)
1000	0.091
2000	0.381
3000	0.817
4000	1.452
5000	2.485

A created graph of running times:



Analysis of the asymptotic running times:

Complete Graphs: The running time increases significantly with the number of vertices. This is expected as the number of edges in a complete graph is proportional to v^2, thus reflecting the O (v^2) time complexity.

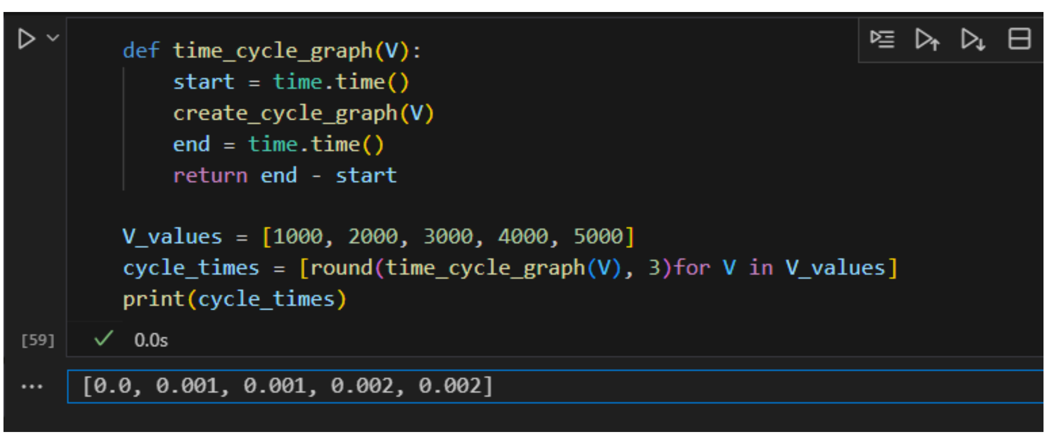
[25 pts] Create a program that accepts a number of vertices “V”, creates a cycle with “V” vertices using an adjacency list data structure of your creation and then output the graph in the format below. All edges should have a weight of 1.   Do not include the outputting of the graph in your timing analysis.

Code:

```
def create_cycle_graph(V):

  adjacency_list = {}

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

    adjacency_list[i] = [(i % V) + 1, (i - 2) % V + 1]

  return adjacency_list


def output_cycle_graph(V):

  graph = create_cycle_graph(V)
```

```
    pointers = []
    edges = []
    edge_count = 0
    for i in range(1, V + 1):
        pointers.append(edge_count + 1)
        for j in graph[i]:
            edges.append((j, 1))
            edge_count += 1
    pointers.append(edge_count + 1)  # For the endpoint
    return V, pointers, edges


def time_cycle_graph(V):
    start = time.time()
    create_cycle_graph(V)
    end = time.time()
    return end - start
V_values = [1000, 2000, 3000, 4000, 5000]
cycle_times = [round(time_cycle_graph(V), 3)for V in V_values]
print(cycle_times)
```

output:



```
def time_cycle_graph(V):
    start = time.time()
    create_cycle_graph(V)
    end = time.time()
    return end - start

V_values = [1000, 2000, 3000, 4000, 5000]
cycle_times = [round(time_cycle_graph(V), 3)for V in V_values]
print(cycle_times)
```

[59] 0.0s

```
[0.0, 0.001, 0.001, 0.002, 0.002]
```

output a file that represents a graph

```
filename = "cycle_graph.txt"
   V_values = [10, 20, 30, 40, 50]
with open(filename, 'w') as file:
   for V in V_values:
       n, p, e = output_cycle_graph(V)  # Corrected this line
       file.write(f"{n} # 0th value = Number of vertices\n")

       # Correcting the ordinal indicators and writing the pointers
       for index, pointer in enumerate(p):
          ordinal_indicator = 'th' if 11 <= (
             index + 1) % 100 <= 13 else {1: 'st', 2: 'nd', 3: 'rd'}.get((index + 1) % 10, 'th')
          file.write(
             f"{pointer} # {index + 1}{ordinal_indicator} value = starting location for vertex {index}'s
edges\n")

       # Writing the edges with correct descriptions
       for index, edge in enumerate(e):
          description_index = index + len(p)
          # Correctly identifies the from-vertex
          vertex_from = ((index // 2) % V) + 1
          file.write(
             f"{edge[0]} {edge[1]} # {description_index}th value = Vertex {vertex_from} is adjacent
to Vertex {edge[0]} and has a weight of {edge[1]}\n")
```

Part of the output file content:

module3.hw.ipynb ● cycle_graph.txt ×

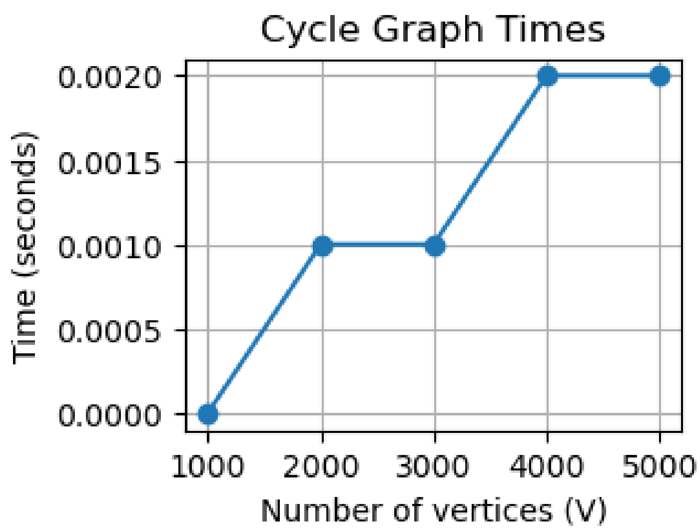
Assignments > Module 3 > cycle_graph.txt

```
10 # 0th value = Number of vertices
1 # 1st value = starting location for vertex 0's edges
3 # 2nd value = starting location for vertex 1's edges
5 # 3rd value = starting location for vertex 2's edges
7 # 4th value = starting location for vertex 3's edges
9 # 5th value = starting location for vertex 4's edges
11 # 6th value = starting location for vertex 5's edges
13 # 7th value = starting location for vertex 6's edges
15 # 8th value = starting location for vertex 7's edges
17 # 9th value = starting location for vertex 8's edges
19 # 10th value = starting location for vertex 9's edges
21 # 11th value = starting location for vertex 10's edges
2 1 # 11th value = Vertex 1 is adjacent to Vertex 2 and has a weight of 1
10 1 # 12th value = Vertex 1 is adjacent to Vertex 10 and has a weight of
3 1 # 13th value = Vertex 2 is adjacent to Vertex 3 and has a weight of 1
1 1 # 14th value = Vertex 2 is adjacent to Vertex 1 and has a weight of 1
4 1 # 15th value = Vertex 3 is adjacent to Vertex 4 and has a weight of 1
2 1 # 16th value = Vertex 3 is adjacent to Vertex 2 and has a weight of 1
5 1 # 17th value = Vertex 4 is adjacent to Vertex 5 and has a weight of 1
3 1 # 18th value = Vertex 4 is adjacent to Vertex 3 and has a weight of 1
6 1 # 19th value = Vertex 5 is adjacent to Vertex 6 and has a weight of 1
4 1 # 20th value = Vertex 5 is adjacent to Vertex 4 and has a weight of 1
7 1 # 21th value = Vertex 6 is adjacent to Vertex 7 and has a weight of 1
5 1 # 22th value = Vertex 6 is adjacent to Vertex 5 and has a weight of 1
```

The chat of running times:

Number of Vertices (V)	Complete Graph Time (seconds)
1000	0.0
2000	0.001
3000	0.001
4000	0.002
5000	0.002

A created graph of running times:



Analysis of the asymptotic running times:

Cycle Graphs: The running time also increases with the number of vertices but at a much lower rate than the complete graphs. Its time increases linearly reflecting the O(V) complexity since each vertex is connected to exactly two other vertices.

[50 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. All edges should have a weight of 1. Store your graph using an adjacency list data structure of your creation and then output the graph in the format below. THETA(V^2) is fine. 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.

Code:

```
import random

import time

def create_random_graph(V, E):

  adjacency_list = {i: [] for i in range(1, V + 1)}

  possible_edges = set()


  # Avoiding the creation of multiple identical edges

  while E > 0 and len(possible_edges) < V * (V - 1) / 2:
```

```
        v1, v2 = random.sample(range(1, V + 1), 2)
        edge = tuple(sorted((v1, v2)))
        if edge not in possible_edges:
            possible_edges.add(edge)
            adjacency_list[v1].append(v2)
            adjacency_list[v2].append(v1)
            E -= 1
    return adjacency_list

def output_random_graph(V, E):
    graph = create_random_graph(V, E)
    pointers = []
    edges = []
    edge_count = 0
    for i in range(1, V + 1):
        pointers.append(edge_count + 1)  # Start of the edges for vertex i
        for j in graph[i]:
            edges.append((j, 1)) # Guarantees that all edges will have a weight of 1.
            edge_count += 1
    pointers.append(edge_count + 1)  # For the endpoint
    return V, pointers, edges

def time_random_graph(V, E):
    start = time.time()
    create_random_graph(V, E)
    end = time.time()
    return end - start

V_values = [1000, 2000, 3000, 4000, 5000]
```

```
# Generate random E values for each V
E_values = [random.randint(0, V*(V-1)//2) for V in V_values]
random_times = [round(time_random_graph(V, E), 3)
                for V, E in zip(V_values, E_values)]
print(random_times)
```

<u>Output a file that represents a graph:</u>

```
filename = "random_graph.txt"
V_values = [10, 20, 30, 40, 50]
E_values = [random.randint(0, V*(V-1)//2) for V in V_values]
with open(filename, 'w') as file:
  for V, E in zip(V_values, E_values):
    n, p, e = output_random_graph(V, E)  # Corrected this line
    file.write(f"{n} # 0th value = Number of vertices\n")

    # Correcting the ordinal indicators and writing the pointers
    for index, pointer in enumerate(p):
      ordinal_indicator = 'th' if 11 <= (
        index + 1) % 100 <= 13 else {1: 'st', 2: 'nd', 3: 'rd'}.get((index + 1) % 10, 'th')
      file.write(
        f"{pointer} # {index + 1}{ordinal_indicator} value = starting location for vertex
{index}'s edges\n")

    # Writing the edges with correct descriptions
    for index, edge in enumerate(e):
      description_index = index + len(p)
      # Correctly identifies the from-vertex
      vertex_from = ((index // 2) % V) + 1
      file.write(
```

f"{edge[0]} {edge[1]} # {description_index}th value = Vertex {vertex_from} is adjacent to Vertex {edge[0]} and has a weight of {edge[1]}\n")

Part of the output file content:

module3.hw.ipynb ● random_graph.txt ×

Assignments > Module 3 > random_graph.txt

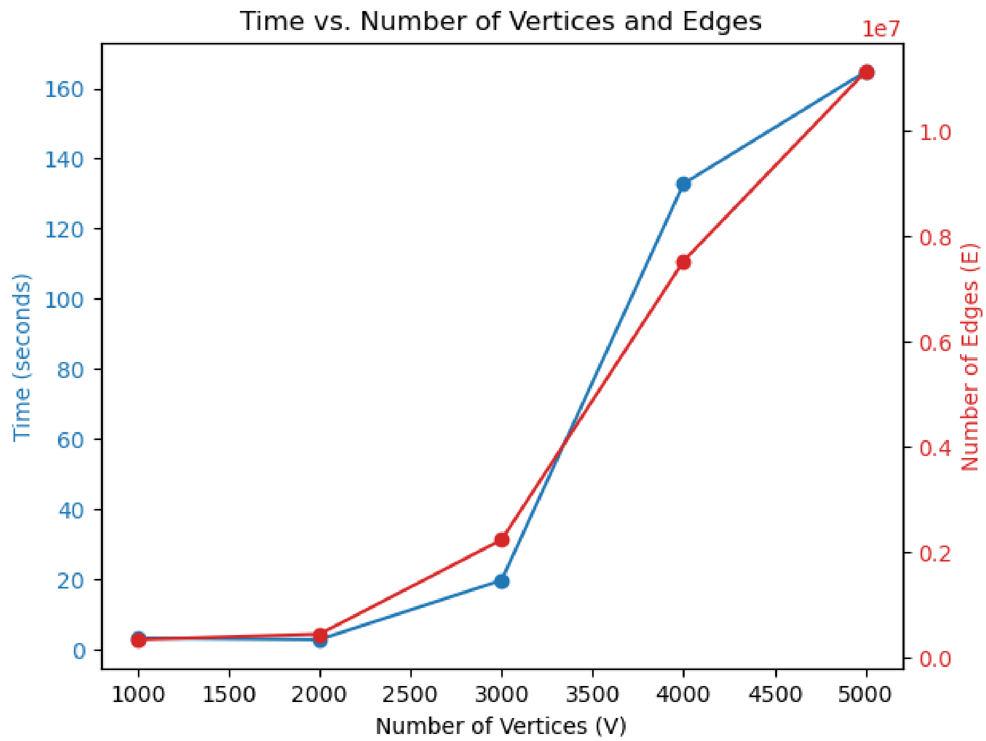
```
10 # 0th value = Number of vertices
1 # 1st value = starting location for vertex 0's edges
9 # 2nd value = starting location for vertex 1's edges
14 # 3rd value = starting location for vertex 2's edges
18 # 4th value = starting location for vertex 3's edges
23 # 5th value = starting location for vertex 4's edges
30 # 6th value = starting location for vertex 5's edges
32 # 7th value = starting location for vertex 6's edges
37 # 8th value = starting location for vertex 7's edges
43 # 9th value = starting location for vertex 8's edges
48 # 10th value = starting location for vertex 9's edges
53 # 11th value = starting location for vertex 10's edges
4 1 # 11th value = Vertex 1 is adjacent to Vertex 4 and has a weight of
6 1 # 12th value = Vertex 1 is adjacent to Vertex 6 and has a weight of
2 1 # 13th value = Vertex 2 is adjacent to Vertex 2 and has a weight of
9 1 # 14th value = Vertex 2 is adjacent to Vertex 9 and has a weight of
8 1 # 15th value = Vertex 3 is adjacent to Vertex 8 and has a weight of
10 1 # 16th value = Vertex 3 is adjacent to Vertex 10 and has a weight o
3 1 # 17th value = Vertex 4 is adjacent to Vertex 3 and has a weight of
7 1 # 18th value = Vertex 4 is adjacent to Vertex 7 and has a weight of
10 1 # 19th value = Vertex 5 is adjacent to Vertex 10 and has a weight o
1 1 # 20th value = Vertex 5 is adjacent to Vertex 1 and has a weight of
5 1 # 21th value = Vertex 6 is adjacent to Vertex 5 and has a weight of
9 1 # 22th value = Vertex 6 is adjacent to Vertex 9 and has a weight of
8 1 # 23th value = Vertex 7 is adjacent to Vertex 8 and has a weight of
4 1 # 24th value = Vertex 7 is adjacent to Vertex 4 and has a weight of
5 1 # 25th value = Vertex 8 is adjacent to Vertex 5 and has a weight of
8 1 # 26th value = Vertex 8 is adjacent to Vertex 8 and has a weight of
1 1 # 27th value = Vertex 9 is adjacent to Vertex 1 and has a weight of
5 1 # 28th value = Vertex 9 is adjacent to Vertex 5 and has a weight of
```

The chat of running times:

Number of Vertices (V)	Complete Graph Time (seconds)
1000	3.141
2000	2.707,
3000	19.641
4000	132.849
5000	164.829

A created graph of running times:

```
fig, ax1 = plt.subplots()


color = 'tab:blue'
ax1.set_xlabel('Number of Vertices (V)')
ax1.set_ylabel('Time (seconds)', color=color)
ax1.plot(V_values, random_times, marker='o', color=color)
ax1.tick_params(axis='y', labelcolor=color)


ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis
color = 'tab:red'
ax2.set_ylabel('Number of Edges (E)', color=color)
ax2.plot(V_values, E_values, marker='o', color=color)
ax2.tick_params(axis='y', labelcolor=color)


fig.tight_layout()
plt.title('Time vs. Number of Vertices and Edges')
plt.show()
```



Analysis of the asymptotic running times:

Random Graphs: The running time is generally lower and increases as the number of vertices increases. However, it does not show as steep an increase as the complete graph, which is due to the limited number of edges created in the random graph compared to the complete graph. The time complexity can vary depending on how the random edges are generated but typically would expect an average case closer to O(V+E).