Jordan Francis - Lab 1

```
jordanfrancis@Jordans-MacBook-Pro scripts % python3 generate_network.py --size tiny

=== Generating TINY network ===
Description: Baseline - no visible difference

Expected nodes: 100
Adjacency matrix memory: 0.00 GB
Generating 100 nodes in a 10x10 grid...
Generating ~200 edges (avg degree 4)...
Adding 20 additional connections...
Saving 100 nodes to ../data/tiny_nodes.csv
Saving 200 edges to ../data/tiny_edges.csv
Metadata saved to ../data/tiny_meta.txt

✓ Network generation complete!
  Actual nodes: 100
  Actual edges: 200
  Actual avg degree: 4.00
jordanfrancis@Jordans-MacBook-Pro scripts % python3 generate_network.py --size small

=== Generating SMALL network ===
Description: Small town road network

Expected nodes: 1000
Adjacency matrix memory: 0.01 GB
Generating 961 nodes in a 31x31 grid...
Generating ~2402 edges (avg degree 5)...
Adding 542 additional connections...
Saving 961 nodes to ../data/small_nodes.csv
Saving 2402 edges to ../data/small_edges.csv
Metadata saved to ../data/small_meta.txt

✓ Network generation complete!
  Actual nodes: 961
  Actual edges: 2402
  Actual avg degree: 5.00
```

```
jordanfrancis@Jordans-MacBook-Pro scripts % python3 generate_network.py --size medium

=== Generating MEDIUM network ===
Description: City road network - matrix starts to show strain

Expected nodes: 10000
Adjacency matrix memory: 0.75 GB
Generating 10000 nodes in a 100x100 grid...
Generating ~30000 edges (avg degree 6)...
Adding 10200 additional connections...
Saving 10000 nodes to ../data/medium_nodes.csv
Saving 30000 edges to ../data/medium_edges.csv
Metadata saved to ../data/medium_meta.txt

✓ Network generation complete!
  Actual nodes: 10000
  Actual edges: 30000
  Actual avg degree: 6.00
jordanfrancis@Jordans-MacBook-Pro scripts % python3 generate_network.py --size large

=== Generating LARGE network ===
Description: Metro area - matrix uses ~10GB RAM

Expected nodes: 50000
Adjacency matrix memory: 18.63 GB
Generating 49729 nodes in a 223x223 grid...
Generating ~149187 edges (avg degree 6)...
Adding 50175 additional connections...
Saving 49729 nodes to ../data/large_nodes.csv
Saving 149187 edges to ../data/large_edges.csv
Metadata saved to ../data/large_meta.txt

✓ Network generation complete!
  Actual nodes: 49729
  Actual edges: 149187
  Actual avg degree: 6.00
```

Exercise 1: Memory Scaling Analysis

Network	Nodes	List Memory	Matrix Memory	Ratio
tiny	100	0.2	0.3	1.2x
small	961	0.8	7.5	9.4x
medium	10000	9.4	766.3	81.3x
large	49729	49.1	18884.8	384.3x

Questions:

The relationship between node count and matrix memory is quadratic $O(N^2)$, the matrix must store for every possible pair of nodes. Based on our table we see after the 50,000 mark of nodes the matrix will become impractical. The memory for 200,000 nodes is to be assumed to be ~40 GB of memory which would not fit on a 32 GB of RAM.

```
jordanfrancis@Jordans-MacBook-Pro c % ./graph_benchmark ../data/medium_nodes.csv ../data/medium_edges
.csv


========================================
 CS3050 Graph Representation Benchmark
========================================
Nodes: 10000
Matrix would require: 0.75 GB

--- Adjacency List ---
Load time: 13.5 ms
Memory: 1.68 MB
Nodes: 10000, Edges: 30000

Benchmarks:
  Edge queries: 10000 in 0.6 ms (16181229 q/s), density: 0.0008
  Dijkstra: 5 queries in 3.3 ms (avg 0.65 ms, 2526 nodes explored)

--- Adjacency Matrix ---
  Allocating 10000x10000 matrix (0.75 GB)...
  Matrix allocated.
Load time: 29.6 ms
Memory: 763.24 MB

Benchmarks:
  Edge queries: 10000 in 10.0 ms (1004016 q/s), density: 0.0008
  Dijkstra: 5 queries in 102.7 ms (avg 20.55 ms, 2526 nodes explored)

========================================
 Benchmark complete!
========================================
jordanfrancis@Jordans-MacBook-Pro c % ▮
```

Exercise 2: Edge Query Benchmark

The matrix has O(1) edge lookup. Measure the speedup:

```
# Test code is in benchmark_edge_queries() # In the C program
# Run 10,000 random edge existence checks
```

Questions:

   The matrix edge lookup on the smaller node scale is only slightly faster but as the graphs grow larger the difference grows as well, thus the difference grows with graph size. For the medium 10,000 nodes the adjacency list completes the nodes in 9.4 while the adjacency matrix took 766.3 we do not see that the list is complete the search in a faster time because of the sparsity of the graph. A fast edge look up is best in a dense graph where a system where constant time edge checks are critical.

```
● ● ●   ■ c — -zsh — 83×59
=======================================
 CS3050 Graph Representation Benchmark
=======================================
Nodes: 961
Matrix would require: 0.01 GB

--- Adjacency List ---
Load time: 1.5 ms
Memory: 0.14 MB
Nodes: 961, Edges: 2402

Benchmarks:
  Edge queries: 10000 in 0.4 ms (24691345 q/s), density: 0.0072
  Dijkstra: 5 queries in 0.4 ms (avg 0.08 ms, 504 nodes explored)

--- Adjacency Matrix ---
  Allocating 961x961 matrix (0.01 GB)...
  Matrix allocated.
Load time: 3.5 ms
Memory: 7.08 MB

Benchmarks:
  Edge queries: 10000 in 0.4 ms (25641003 q/s), density: 0.0072
  Dijkstra: 5 queries in 2.7 ms (avg 0.54 ms, 504 nodes explored)

=======================================
 Benchmark complete!
=======================================
jordanfrancis@Jordans-MacBook-Pro c % ./graph_benchmark ../data/medium_nodes.csv ..
/data/medium_edges.csv

=======================================
 CS3050 Graph Representation Benchmark
=======================================
Nodes: 10000
Matrix would require: 0.75 GB

--- Adjacency List ---
Load time: 12.0 ms
Memory: 1.68 MB
Nodes: 10000, Edges: 30000

Benchmarks:
  Edge queries: 10000 in 0.6 ms (15974443 q/s), density: 0.0008
  Dijkstra: 5 queries in 2.9 ms (avg 0.59 ms, 2526 nodes explored)

--- Adjacency Matrix ---
  Allocating 10000x10000 matrix (0.75 GB)...
  Matrix allocated.
Load time: 59.7 ms
Memory: 763.24 MB

Benchmarks:
  Edge queries: 10000 in 15.2 ms (658198 q/s), density: 0.0008
  Dijkstra: 5 queries in 148.1 ms (avg 29.61 ms, 2526 nodes explored)

=======================================
 Benchmark complete!
```

```
make: Nothing to be done for `all`.
jordanfrancis@Jordans-MacBook-Pro c % ./graph_benchmark ../data/ti

=======================================
 CS3050 Graph Representation Benchmark
=======================================
Nodes: 100
Matrix would require: 0.00 GB

--- Adjacency List ---
Load time: 0.4 ms
Memory: 0.01 MB
Nodes: 100, Edges: 200

Benchmarks:
  Edge queries: 10000 in 0.4 ms (24271834 q/s), density: 0.0410
  Dijkstra: 5 queries in 0.0 ms (avg 0.01 ms, 39 nodes explored)

--- Adjacency Matrix ---
  Allocating 100x100 matrix (0.00 GB)...
  Matrix allocated.
Load time: 0.2 ms
Memory: 0.08 MB

Benchmarks:
  Edge queries: 10000 in 0.3 ms (32051303 q/s), density: 0.0410
  Dijkstra: 5 queries in 0.1 ms (avg 0.01 ms, 39 nodes explored)

=======================================
 Benchmark complete!
=======================================
jordanfrancis@Jordans-MacBook-Pro c % ▊
```

Exercise 3: Algorithm Performance

Network	List Dijkstra	Matrix Dijkstra	Slowdown
tiny	0.0001 s	0.0002 s	1.5x
small	0.0012 s	0.0123 s	10.0x
medium	0.0103 s	0.8400 s	81.3x

Questions:

Matrix- based is slower because we are receiving data from a small count. The slowdown occurs because the matrix-based must scan the entirety to find neighbors resulting in O(N) per node no matter the edges that exists. We see it happening during the realization step where every possible neighbor is examined. For the matrix it would be fastest in a high-density graph as there will be the most real edges for the nodes making good use of the matrix time.

```
====================================
 Benchmark complete!
====================================
jordanfrancis@Jordans-MacBook-Pro c % cd -
~/4050-Lab-1
jordanfrancis@Jordans-MacBook-Pro 4050-Lab-1 % python3 scripts/generate_network.py --size huge


=== Generating HUGE network ===
Description: Regional network - matrix WILL crash 32GB machine!
[
Expected nodes: 100000
Adjacency matrix memory: 74.51 GB

⚠ WARNING: Matrix representation will use >20GB RAM!
    Consider using adjacency list only for this size.
Generating 99856 nodes in a 316x316 grid...
Generating ~299568 edges (avg degree 6)...
Adding 100488 additional connections...
Saving 99856 nodes to ../data/huge_nodes.csv
Saving 299568 edges to ../data/huge_edges.csv
Metadata saved to ../data/huge_meta.txt

✓ Network generation complete!
  Actual nodes: 99856
  Actual edges: 299568
  Actual avg degree: 6.00
```

Exercise 4: The Breaking Point (Demonstration)

Before execution memory is low and stable during the memory has a large increase for the reservation for the huge memory set. When memory is exhausted the program will most likely do one of the following fail in allocation, crash, or cause the system to be unresponsive.

Exercise 5: Real-World Decision Making

Given these scenarios, choose the appropriate representation and justify:

1. **Google Maps routing**: ~50 million road intersections, avg degree 4
2. **Social network analysis**: 1 billion users, avg 200 friends
3. **Circuit analysis**: 10,000 components, each connected to 3 others
4. **Dense communication matrix**: 500 servers, all-to-all connections

For a Google Maps routing system, it would be best to us an adjacency list because the graph is extremely large and sparse. As the average degree is 4 using an adjacency list would be near impossible to store. The social network analysis would once again need to use a adjacency list due to the mass amount of users and average friends (connections) that is established in the scenario both of these attributes allude to a very large scale but sparse connectivity in comparison. Circuit analysis has a similar set up for the scenario although it is seemingly not as large the connections between them is only 3 making it another sparse connectivity so we should use an adjacency list. Finally for the dense communication matrix we see that it is in a small manageable bound of only 500 which are all interconnected because of this density we should use an adjacency matrix.