Jordan Francis – lab 1 – discussion questions
Based on your experimental results, answer the following:
Photos added below

1. MEMORY SCALING
Memory usage scales quadratically as seen from the results. The matrix becomes impractical when it is >= 50,000 nodes for the standard 32 GB system. A 200,000-node system would require 40 GB of RAM.

2. EDGE QUERY PERFORMANCE
When checking edge existence, it was faster for adjacency list when then graphs were larger and sparse, we saw same times when the graphs were small. The sparsity of the graphs allowed the adjacency to be cheap and only need to scan 4 neighbors. A fast edge existence matters when sifting through a dense graph like a well-connected social network.

3. NEIGHBOR ITERATION
When iterating over neighbors it is faster to use Adjacency list because adjacency matrix will trace the entire row and list will only trace through 4 neighbors. The matrix gets slower as V increases because the get_neighbors () call must scan all of V in the row, so the larger V gets the more expensive we see for the program.

4. ALGORITHM IMPLICATIONS
Dijkstra works better with adjacency list, at the large scale tested we see list is done in 0.0551s and matrix takes 20.9509s and explores less nodes than list. If we are only wanting to check edge existence it would make the most sense to use adjacency matrix.

5. REAL-WORLD TRADEOFFS
Road networks with 4-6 edges would be considered sparce and would want to use adjacency list. A similar idea is used with social networks if there were millions of nodes we would need to use adjacency list for best memory usage. If we are looking at a clique of 1000 nodes and 500,000 edges the ration shows that it will be producing a very dense graph which means that list is no longer optimal, and it would be faster to use adjacency matrix.

6. THE CROSSOVER POINT
Matrix becomes worthwhile when the ratio of edge and nodes for the density equal to 1 or higher. For the experimental data it is best to plot memory ratio as the graphs are very sparse and not well connected.

Implement your own graph class

```
class HybridGraph:
    def __init__(self, num_nodes, density_threshold=0.1, block_size=100):
        self.num_nodes = num_nodes
        self.density_threshold = density_threshold
        self.block_size = block_size
        # Global adjacency list (default storage)
        self.adj_list = {i: set() for i in range(num_nodes)}
        # Block storage
        self.blocks = {}  # block_id -> {"type": "list" or "matrix", "data": ...}

    def _get_block_id(self, node):
        return node // self.block_size

    def add_edge(self, u, v):
        self.adj_list[u].add(v)
        self.adj_list[v].add(u)

    def _compute_block_density(self, block_id):
        start = block_id * self.block_size
        end = min(start + self.block_size, self.num_nodes)
        nodes = range(start, end)
        possible_edges = (end - start) ** 2
        actual_edges = 0
        for u in nodes:
            for v in self.adj_list[u]:
                if start <= v < end:
                    actual_edges += 1

        return actual_edges / possible_edges if possible_edges > 0 else 0

    def optimize_blocks(self):
        total_blocks = (self.num_nodes + self.block_size - 1) // self.block_size

        for block_id in range(total_blocks):
            density = self._compute_block_density(block_id)

            start = block_id * self.block_size
            end = min(start + self.block_size, self.num_nodes)

            if density >= self.density_threshold:
                # Convert to matrix
                size = end - start
                matrix = [[0] * size for _ in range(size)]
```

```
            for u in range(start, end):
                for v in self.adj_list[u]:
                    if start <= v < end:
                        matrix[u - start][v - start] = 1

            self.blocks[block_id] = {
                "type": "matrix",
                "data": matrix
            }
        else:
            # Keep as adjacency list
            block_list = {
                u: self.adj_list[u]
                for u in range(start, end)
            }
            self.blocks[block_id] = {
                "type": "list",
                "data": block_list
            }
    def has_edge(self, u, v):
        block_id = self._get_block_id(u)
        if block_id in self.blocks and self.blocks[block_id]["type"] == "matrix":
            start = block_id * self.block_size
            matrix = self.blocks[block_id]["data"]
            return matrix[u - start][v - start] == 1
        else:
            return v in self.adj_list[u]
    def get_neighbors(self, u):
        block_id = self._get_block_id(u)

        if block_id in self.blocks and self.blocks[block_id]["type"] == "matrix":
            start = block_id * self.block_size
            matrix = self.blocks[block_id]["data"]
            row = matrix[u - start]
            return [start + i for i, val in enumerate(row) if val == 1]
        else:
    return list(self.adj_list[u])
```

```
============================================================
 EXPERIMENT: TINY NETWORK
============================================================
Network has 100 nodes
Adjacency matrix would require: 0.00 GB

--- Adjacency List Representation ---

Loading graph (list representation)...
  Loaded 100 nodes in 0.001s
  Loaded 200 edges in 0.002s
  Total load time: 0.003s
  Estimated memory: 0.02 MB
Load time: 0.003 seconds
Peak memory: 0.21 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 5740118 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 0.0008 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.0001s (73 nodes explored)
  A* avg: 0.0001s (13 nodes explored)

--- Adjacency Matrix Representation ---

Loading graph (matrix representation)...
  Detected 100 nodes
  Allocating 100x100 matrix (0.00 GB)...
  Matrix allocated.
  Loaded 100 nodes in 0.001s
  Loaded 200 edges in 0.001s
  Total load time: 0.002s
  Estimated memory: 0.08 MB
Load time: 0.002 seconds
Peak memory: 0.25 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 14453150 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 0.0284 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.0002s (49 nodes explored)
  A* avg: 0.0001s (11 nodes explored)

Press Enter to run experiment on SMALL (961 nodes)...
```

```
============================================================
 EXPERIMENT: SMALL NETWORK
============================================================
Network has 961 nodes
Adjacency matrix would require: 0.01 GB

--- Adjacency List Representation ---

Loading graph (list representation)...
  Loaded 961 nodes in 0.010s
  Loaded 2402 edges in 0.019s
  Total load time: 0.029s
  Estimated memory: 0.16 MB
Load time: 0.030 seconds
Peak memory: 0.80 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 4171776 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 0.0008 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.0012s (622 nodes explored)
  A* avg: 0.0002s (39 nodes explored)

--- Adjacency Matrix Representation ---

Loading graph (matrix representation)...
  Detected 961 nodes
  Allocating 961x961 matrix (0.01 GB)...
  Matrix allocated.
  Loaded 961 nodes in 0.004s
  Loaded 2402 edges in 0.010s
  Total load time: 0.014s
  Estimated memory: 7.13 MB
Load time: 0.017 seconds
Peak memory: 7.49 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 15797755 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 0.2148 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.0123s (542 nodes explored)
  A* avg: 0.0017s (69 nodes explored)

Press Enter to run experiment on MEDIUM (10000 nodes)...
```

```
============================================================
 EXPERIMENT: MEDIUM NETWORK
============================================================
Network has 10000 nodes
Adjacency matrix would require: 0.75 GB

--- Adjacency List Representation ---

Loading graph (list representation)...
  Loaded 10000 nodes in 0.061s
  Loaded 30000 edges in 0.137s
  Total load time: 0.198s
  Estimated memory: 1.83 MB
Load time: 0.198 seconds
Peak memory: 9.42 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 3077936 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 0.0006 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.0103s (5822 nodes explored)
  A* avg: 0.0018s (508 nodes explored)

--- Adjacency Matrix Representation ---

Loading graph (matrix representation)...
  Detected 10000 nodes
  Allocating 10000x10000 matrix (0.75 GB)...
  Matrix allocated.
  Loaded 10000 nodes in 0.240s
  Loaded 30000 edges in 0.101s
  Total load time: 0.342s
  Estimated memory: 763.78 MB
Load time: 0.620 seconds
Peak memory: 766.34 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 5630694 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 2.2798 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.8400s (3653 nodes explored)
  A* avg: 0.0734s (320 nodes explored)

Press Enter to run experiment on LARGE (49729 nodes)...
```

```
============================================================
 COMPARISON SUMMARY
============================================================

Size       Nodes      Edges      List Mem     Matrix Mem   Ratio
--------------------------------------------------------------------------------
tiny       100        200        0.2 MB       0.3 MB       1.2x
small      961        2402       0.8 MB       7.5 MB       9.4x
medium     10000      30000      9.4 MB       766.3 MB     81.3x
large      49729      149187     49.1 MB       18884.8 MB   384.3x

Size       List Edge Q/s    Matrix Edge Q/s Speedup
------------------------------------------------------------
tiny       5740118            14453150        2.5x
small      4171776            15797755        3.8x
medium     3077936            5630694         1.8x
large      1211106            181814          0.2x

Size       List Dijkstra    Matrix Dijkstra Slowdown
------------------------------------------------------------
tiny       0.0001s          0.0002s          1.5x
small      0.0012s          0.0123s          10.0x
medium     0.0103s          0.8400s          81.3x
large      0.0551s          20.9509s         380.0x

Results saved to: /Users/jordanfrancis/4050-Lab-1/data/experiment_results.csv
```

```
============================================================
 EXPERIMENT: LARGE NETWORK
============================================================
Network has 49729 nodes
Adjacency matrix would require: 18.43 GB

--- Adjacency List Representation ---

Loading graph (list representation)...
  Loaded 49729 nodes in 0.195s
  Loaded 149187 edges in 0.702s
  Total load time: 0.897s
  Estimated memory: 9.11 MB
Load time: 0.897 seconds
Peak memory: 49.14 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 1211106 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 0.0017 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 0.0551s (25133 nodes explored)
  A* avg: 0.0172s (4291 nodes explored)

--- Adjacency Matrix Representation ---

Loading graph (matrix representation)...
  Detected 49729 nodes
  Allocating 49729x49729 matrix (18.43 GB)...
  Matrix allocated.
  Loaded 49729 nodes in 7.732s
  Loaded 149187 edges in 0.921s
  Total load time: 8.654s
  Estimated memory: 18871.46 MB
Load time: 19.634 seconds
Peak memory: 18884.77 MB

Running edge queries benchmark (10,000 queries)...
  Speed: 181814 queries/second
Running neighbor iteration benchmark (10,000 nodes)...
  Time: 12.5063 seconds
Running shortest path benchmarks (5 random pairs)...
  Dijkstra avg: 20.9509s (16165 nodes explored)
  A* avg: 4.5370s (3498 nodes explored)
```