Example Project 2
Group 1

Lim Li Ping Joey, U2321331C
Vinodhithaa, U2322818J
Murugan Rithika U2323063E

Project 2

To investigate how the choice of **Graph Representation** and **Priority Queue** data structures affect the **time complexity** of **Dijkstra's Algorithm**

Adjacency Matrix + Array PQ

vs

Adjacency List + Heap PQ

Graphs

- Dijkstra's Algorithm works on **Weighted** and **Directed** graphs
- Two datasets:
  - Sparse Graphs: $|E| \approx |V|$
  - Dense Graphs: $|E| \approx |V|^2$

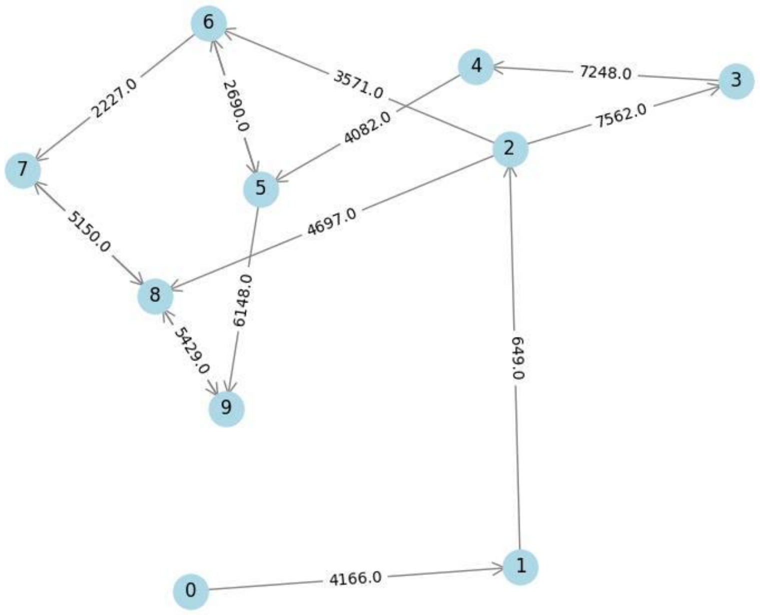
- Graphs are **connected** to make the comparisons meaningful

- There can be **bidirectional edges with different weights**, but there are **no parallel edges** between the same pair of (src, dest) vertices

```python
# (|V|, |E|)
sparse = []
dense = []
for num_vertices in [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000]:
    sparse.append([num_vertices, int(num_vertices*1.5)])
    dense.append([num_vertices, num_vertices * (num_vertices-1)])
```

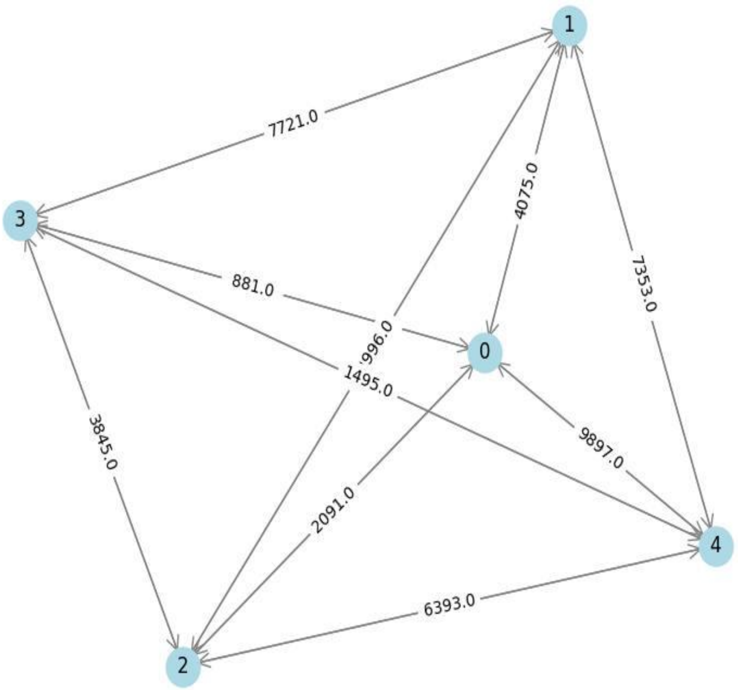
```python
edges = []
edge_set = set()
    # Ensure the graph is connected by adding at least consecutive edges
    for i in range(num_nodes - 1):
        src = i
        dest = i + 1
        weight = random.randint(1, 10000)
        edges.append((src, dest, weight))
        edge_set.add((src, dest))

    # Add additional random edges to meet the required number of edges
    while len(edges) < num_edges:

        src = random.randint(0, num_nodes - 1)
        dest = random.randint(0, num_nodes - 1)
        while src == dest or (src,dest) in edge_set:
            src = random.randint(0, num_nodes - 1)
            dest = random.randint(0, num_nodes - 1)

        weight = random.randint(1, 10000)
        edges.append((src, dest, weight))
        edge_set.add((src, dest))

    # Write the graph to a file in the required format
    with open(file_name, 'w') as f:
        for src, dest, weight in edges:
            f.write(f"{src} {dest} {weight}\n")
```

Graphs



Sparse

$$|E| \approx |V|$$



Dense

$$|E| \approx |V|^2$$

part (a)

For Adjacency Matrix + Array PQ:

1) Code Implementation
2) Theoretical Analysis
3) Empirical Analysis on Sparse vs Dense Graphs
4) Comparison of Big O with Sparse and Dense Graphs

a) Adjacency Matrix as Graph & Array for Priority Queue

```python
def dijkstras_adjacency_matrix_and_array_pq(adj_matrix, src):
    print("Running Dijkstra's with Adjacency Matrix + Array")
    key_comps = 0 #increment when distance is compared
    distances= []
    pi= []
    S= []
    for _ in adj_matrix[0]:
        distances.append(float("inf"))
        pi.append(None)
        S.append(0)

    distances[src]=0
    V= len(distances) #number of vertices/nodes
    unvisited= len(distances)
```

V

```python
while unvisited>0:

    #select node with smallest current distance
    smallest_distance= float("inf")
    index_smallest_distance=-1

    for index_of_current_node in range(V):

        key_comps+=1

        if distances[index_of_current_node]< smallest_distance and S[index_of_current_node]==0:
            smallest_distance= distances[index_of_current_node]
            index_smallest_distance= index_of_current_node

    if index_smallest_distance ==-1:
        return distances, key_comps #graph is unconnected and there are nodes with infinite distance

    #visit adjacent nodes of node with current smallest distance
    for index_of_adj_node in range(V):

        weight= adj_matrix[index_smallest_distance][index_of_adj_node]

        if weight != 0 and S[index_of_adj_node]==0:

            key_comps+=1
            new_distance= weight + distances[index_smallest_distance]
            current_distance= distances[index_of_adj_node]

            if new_distance < current_distance:
                distances[index_of_adj_node]= new_distance
                pi[index_of_adj_node]= index_smallest_distance #update predecessor

    S[index_smallest_distance]=1 #all adjacent nodes have been visited
    unvisited-=1

return distances, key_comps
```

V

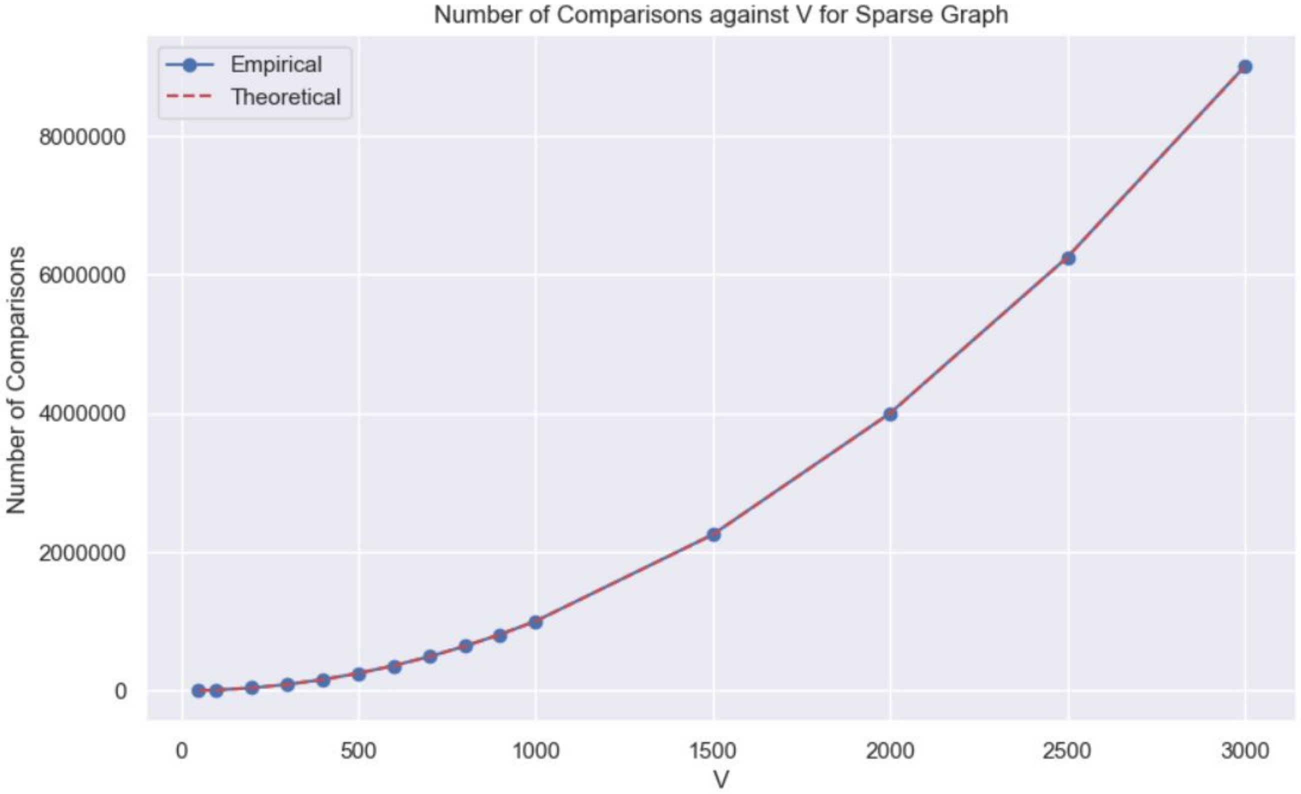
V

O(V)

O(V-1)

Overall Time Complexity: $O(V^2)$

Sparse Graph



Theoretical

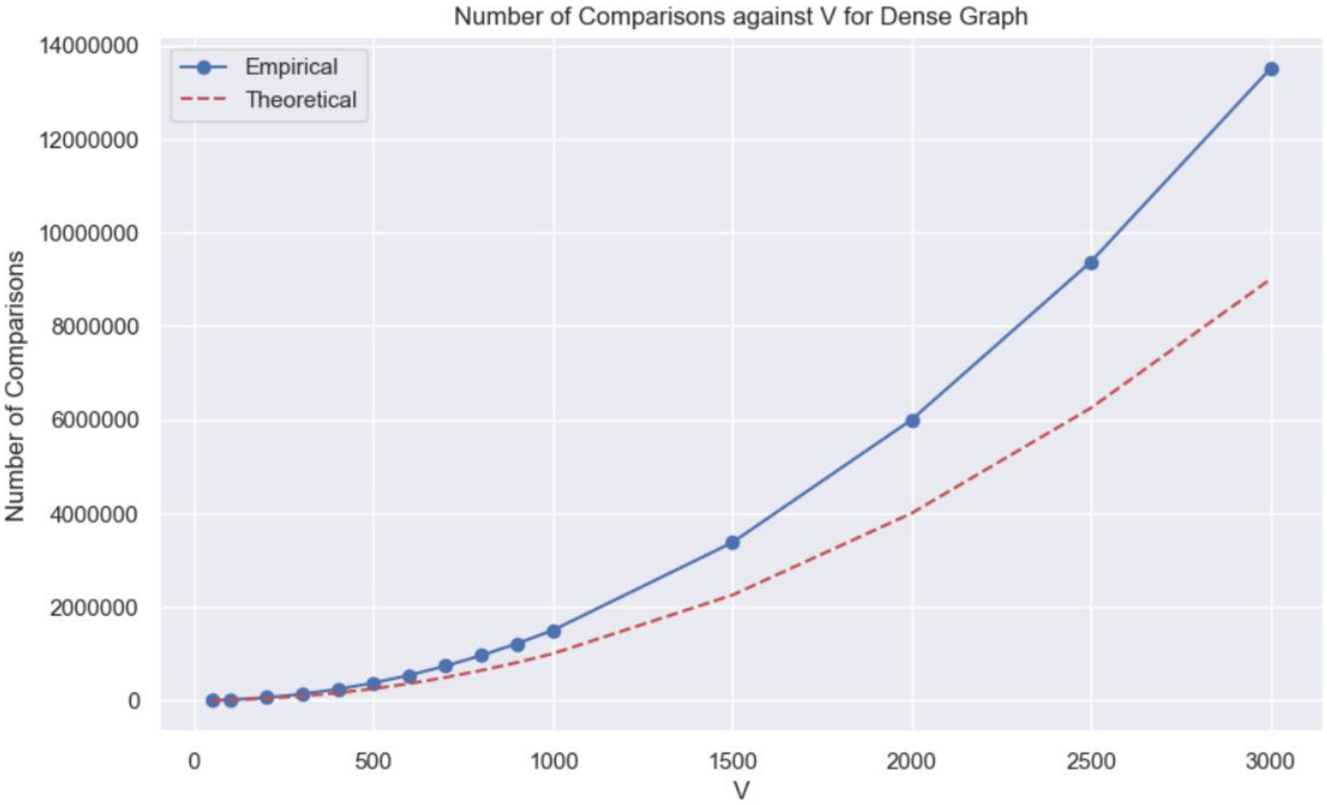
	V	V^2
0	50	2500
1	100	10000
2	200	40000
3	300	90000
4	400	160000

Empirical

	V	Number of Comparisons
0	50	2550
1	100	10106
2	200	40211
3	300	90316
4	400	160433

Theoretical number of comparisons: $O(V^2)$

Dense Graph



Theoretical number of comparisons: $O(V^2)$

```python
#visit adjacent nodes of node with current smallest distance
for index_of_adj_node in range(V):

    weight= adj_matrix[index_smallest_distance][index_of_adj_node]

    if weight != 0 and S[index_of_adj_node]==0:

        key_comps+=1
        new_distance= weight + distances[index_smallest_distance]
        current_distance= distances[index_of_adj_node]

        if new_distance < current_distance:
            distances[index_of_adj_node]= new_distance
            pi[index_of_adj_node]= index_smallest_distance #update predecessor

S[index_smallest_distance]=1 #all adjacent nodes have been visited
unvisited-=1
```

Dense graphs have more edges so weight != 0 and key_comps is incremented more number of times

```python
while unvisited>0:

    #select node with smallest current distance
    smallest_distance= float("inf")
    index_smallest_distance=-1

    for index_of_current_node in range(V):

        key_comps+=1

        if distances[index_of_current_node]< smallest_distance and S[index_of_current_node]==0:
            smallest_distance= distances[index_of_current_node]
            index_smallest_distance= index_of_current_node

    if index_smallest_distance ==-1:
        return distances, key_comps #graph is unconnected and there are nodes with infinite distance

    #visit adjacent nodes of node with current smallest distance
    for index_of_adj_node in range(V):

        weight= adj_matrix[index_smallest_distance][index_of_adj_node]

        if weight != 0 and S[index_of_adj_node]==0:

            key_comps+=1
            new_distance= weight + distances[index_smallest_distance]
            current_distance= distances[index_of_adj_node]

            if new_distance < current_distance:
                distances[index_of_adj_node]= new_distance
                pi[index_of_adj_node]= index_smallest_distance #update predecessor

    S[index_smallest_distance]=1 #all adjacent nodes have been visited
    unvisited-=1

return distances, key_comps
```

V

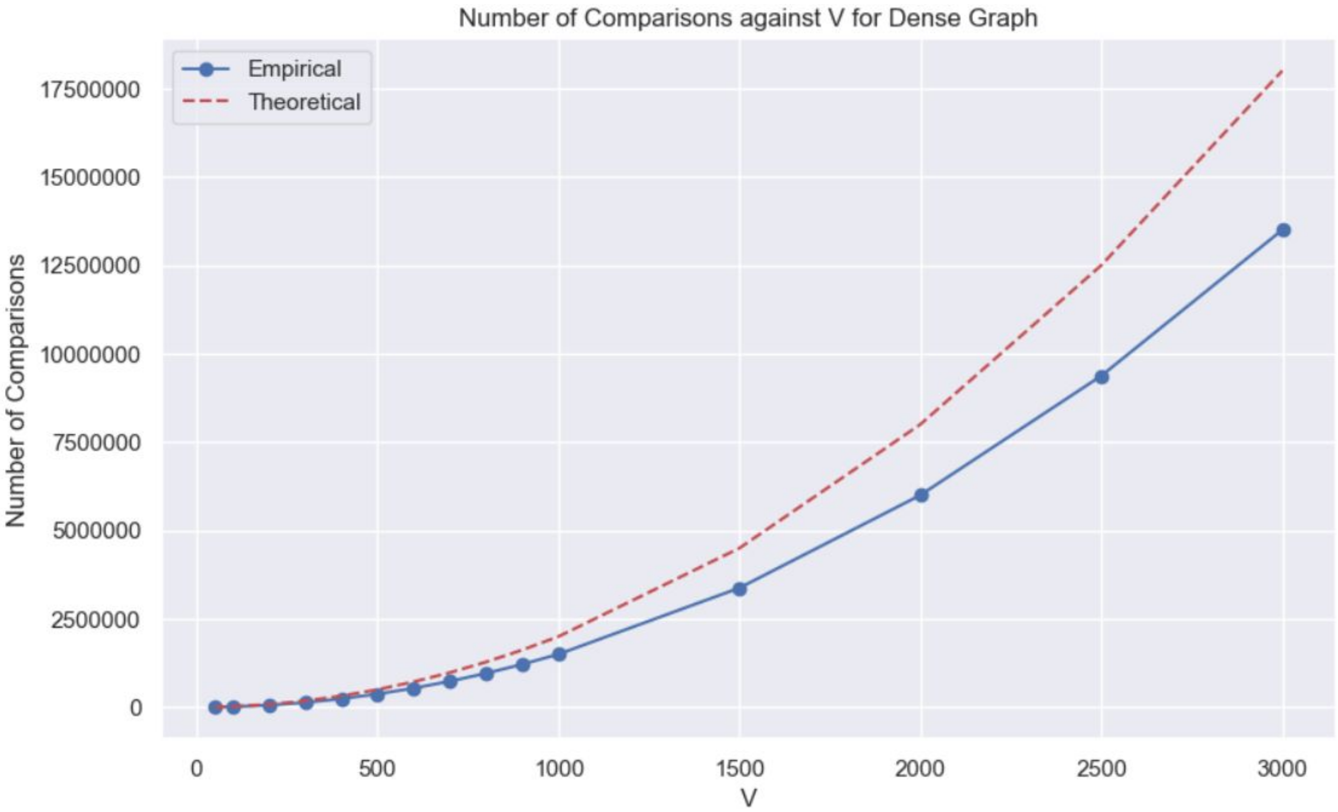
V

$O(2V)$

$O(V-1)$

Upper bound: $O(2V^2)$

Dense Graph



Theoretical number of comparisons: $O(2V^2)$

part (b)

For Adjacency List + Heap PQ:

1) Code Implementation
2) Theoretical Analysis
3) Empirical Analysis on Sparse vs Dense Graphs
4) Comparison of Big O with Sparse and Dense Graphs

b) Code Impl - HeapItem

```
class HeapItem:
    # A class to wrap heap elements and count comparisons.
    comparison_count = 0  # Class-level variable to count comparisons

    def __init__(self, priority, vertex):
        self.priority = priority
        self.vertex = vertex

    def __lt__(self, other):
        """Override < operator for heap comparisons and count
them."""HeapItem.comparison_count += 1
        return self.priority < other.priority

    def __eq__(self, other):
        return self.priority == other.priority
```

b) Code Impl - Initialisation

```
def dijkstras_adjacency_list_and_heap_pq(adj_list, src):
    # initialise variables
    HeapItem.comparison_count = 0
    key_comps = 0
    distances = []
    pi = []
    S = []
    for _ in adj_list.keys():
        distances.append(float("inf"))
        pi.append(None)
        S.append(0)
    distances[src] = 0
    heap_pq = []
    # push all vertices into the heap
    for vertex in adj_list.keys():
        heapq.heappush(heap_pq, HeapItem(distances[vertex],
vertex))
```

b) Theoretical Analysis

Adjacency List as Graph + Heap for Priority Queue

```
def dijkstras_adjacency_list_and_heap_pq(adj_list, src):
    # initialise variables
    HeapItem.comparison_count = 0
    key_comps = 0
    distances = []
    pi = []
    S = []
    for _ in adj_list.keys():
        distances.append(float("inf"))
        pi.append(None)
        S.append(0)
    distances[src] = 0
    heap_pq = []
    # push all vertices into the heap
    for vertex in adj_list.keys():
        heapq.heappush(heap_pq, HeapItem(distances[vertex],
vertex))
```

$O(|V|)$

$O(1)$

$O(|V|\log|V|)$

b) Code Impl - Main Part of Dijkstra’s

```
while heap_pq:
        # Extract Cheapest
        heapItem = heapq.heappop(heap_pq)
        current_dist = heapItem.priority
        u = heapItem.vertex
        if (current_dist > distances[u]):
            continue
        S[u] = 1

        for (v,weight) in adj_list[u]:
            # +1 for comparing distances
            key_comps += 1
            if S[v] == 0 and distances[v] > current_dist +
weight:         # relaxation
                distances[v] = current_dist + weight
                pi[v] = u # predecessor
                heapq.heappush(heap_pq, HeapItem(distances[v],
v))     total_cmps = HeapItem.comparison_count + key_comps
    return distances, total_cmps
```

b) Theoretical Analysis

```
    while heap_pq:
        # Extract Cheapest
        heapItem = heapq.heappop(heap_pq)
        current_dist = heapItem.priority
        u = heapItem.vertex
        if (current_dist > distances[u]):
            continue
        S[u] = 1

        for (v,weight) in adj_list[u]:
            # +1 for comparing distances
            key_comps += 1
            if S[v] == 0 and distances[v] > current_dist +
weight:             # relaxation
                distances[v] = current_dist + weight
                pi[v] = u # predecessor
                heapq.heappush(heap_pq, HeapItem(distances[v],
v))     total_cmps = HeapItem.comparison_count + key_comps
        return distances, total_cmps
```

$O(|V| \log |V|)$

$O(|E| \log |V|)$

Total Time Complexity:

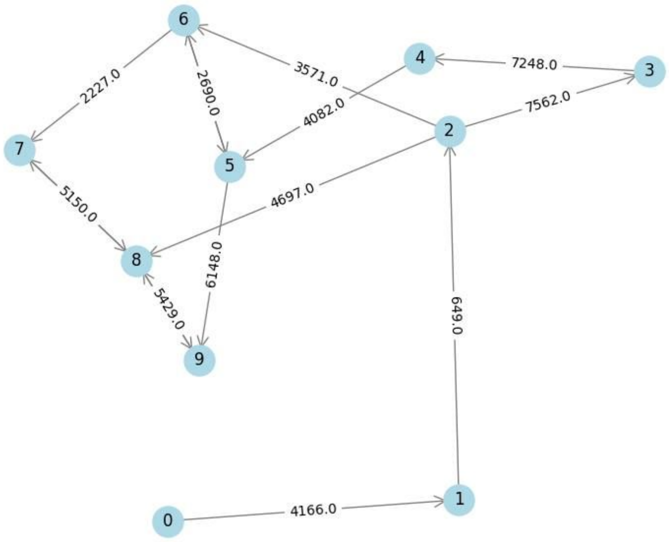
$$O(|V|)+O(|V|\log|V|)+O(|V|\log|V|)+O(|E|\log|V|)=O((|V|+|E|)\log|V|)$$

Hence, the overall time complexity is $O((|V|+|E|)\log|V|)$

Sparse and Dense graphs

Sparse Graph:

- A graph is considered **sparse** when the number of edges $|E|$ is roughly proportional to the number of vertices $|V|$, i.e., $|E| \approx |V|$



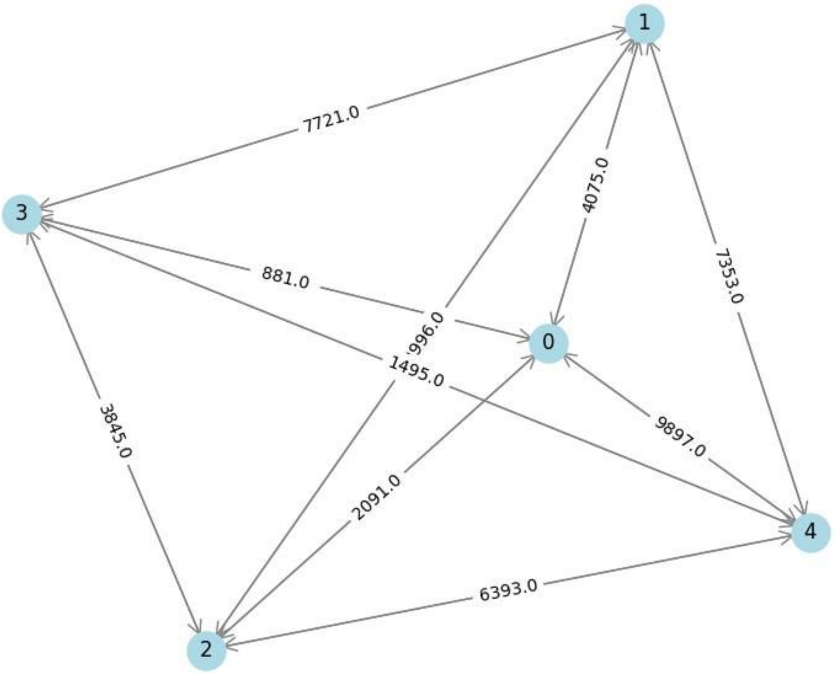
In a sparse graph, since $|E| \approx |V|$, we can substitute $|E|$ with $|V|$ in the time complexity formula.

$O((|V|+|V|)\log|V|)=O(2|V|\log|V|)=O(|V|\log|V|)$

For sparse graphs, the time complexity simplifies to:
$\mathbf{O(|V|\log|V|)}$

Dense Graph:

- A graph is considered **dense** when the number of edges approaches the maximum possible, i.e., $|E| \approx |V|^2$



In a dense graph, the number of edges scales as $|V|^2$, so we substitute $|E|$ with $|V|^2$ in the time complexity formula:

$$O((|V| + |V|^2) \log |V|) = O((|V|^2 \log |V|)$$

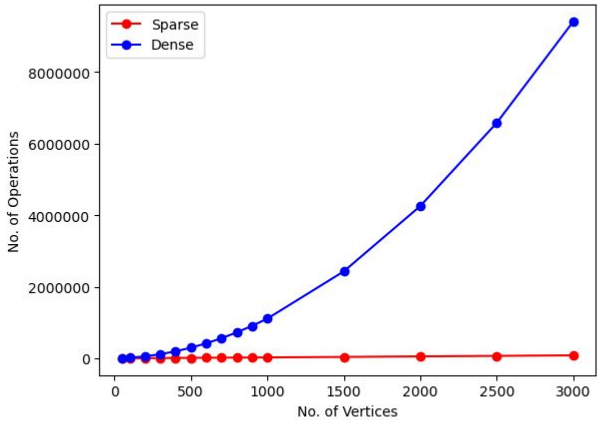
For dense graphs, the time complexity becomes:

$$\mathbf{O(|V|^2 \log |V|)}$$

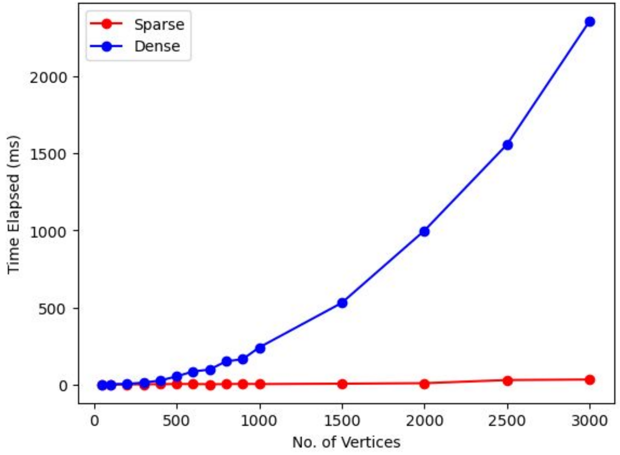
b) Empirical Analysis

Split into two tests: No. of Operations and Time Elapsed (ms)

No. of Operations: Relaxation Checks + Heap Cmps



No. of Operations



Time Elapsed (ms)

V	50	100	200	300	400	500	600	700	800	900	1000	1500	2000	2500	3000

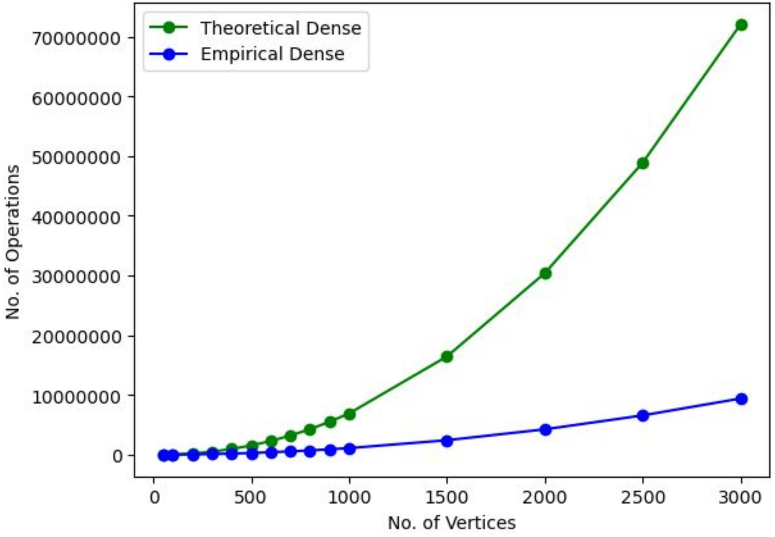
Sparse: |E| = 1.5 * |V|
Dense: |E| = |V| * (|V|-1)

b) Empirical Analysis

Theoretical Dense vs Empirical Dense

No. of Operations: Relaxation Checks + Heap Cmps

Theoretical Upper Bound: $O((|V| + |E|)log|V|)$

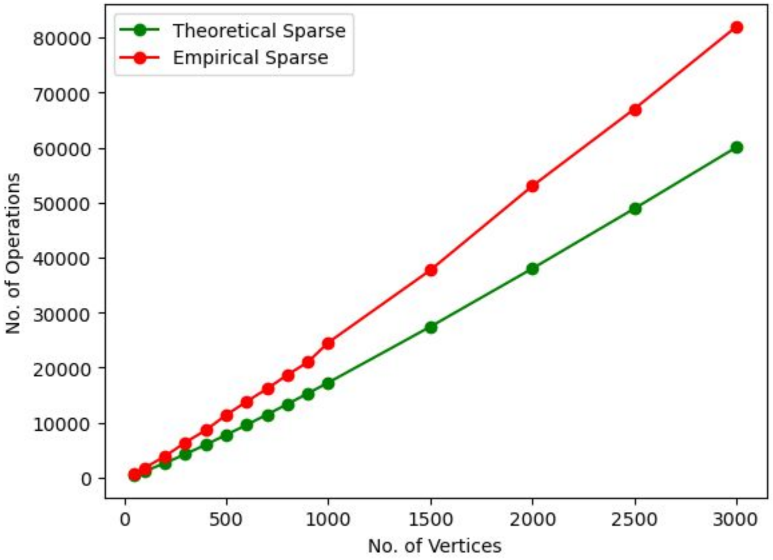


b) Empirical Analysis

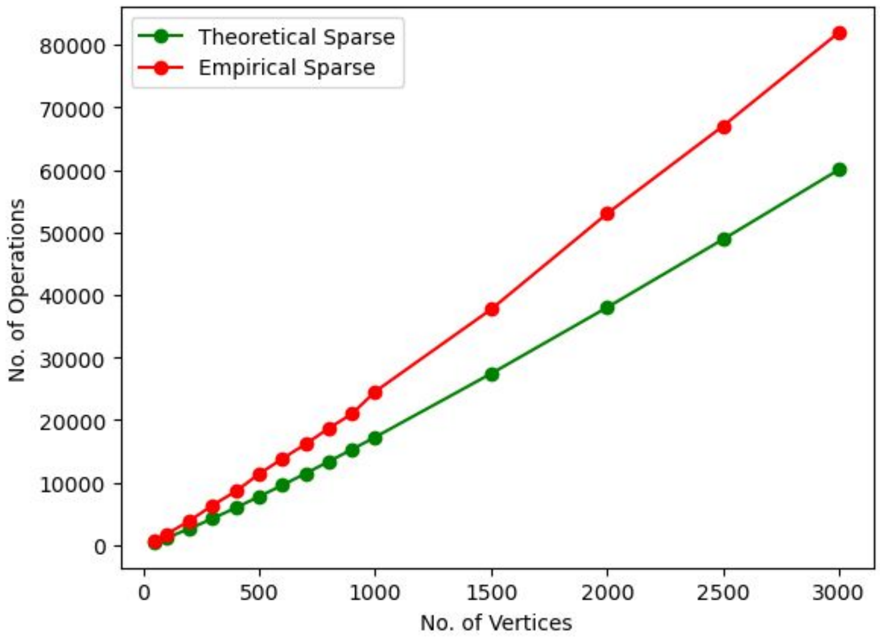
Theoretical Sparse vs Empirical Sparse

No. of Operations: Relaxation Checks + Heap Cmps

Theoretical Upper Bound: $O((|V| + |E|)log|V|)$



b) Empirical Analysis



Why is there a **discrepancy**?

$$O((|V| + |E|)log|V|)$$

b) Theoretically Modelling our Empirical Data

$O((|V| + |E|)log|V|)$ shows the upper bound;

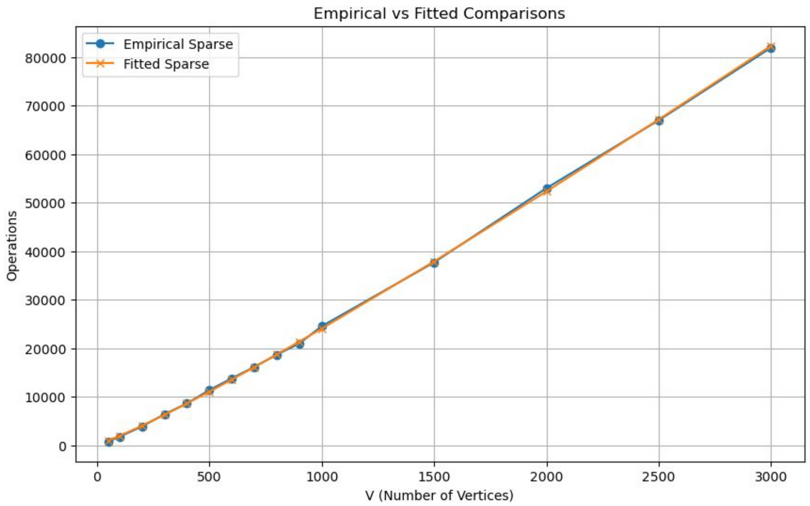
the worst case for Adjacency List + Heap Dijkstra's Implementation. But how do we model our actual Empirical Data?

- Linear Regression to find c_1, c_2 in

$$c_1|V|log|V| + c_2|E|log|V|$$

By fitting the equation to our Empirical Data using Linear Regression, we can model how the implementation scales

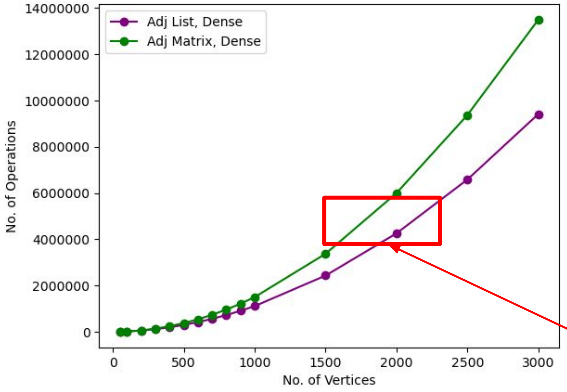
- Potential Problem: Overfitting

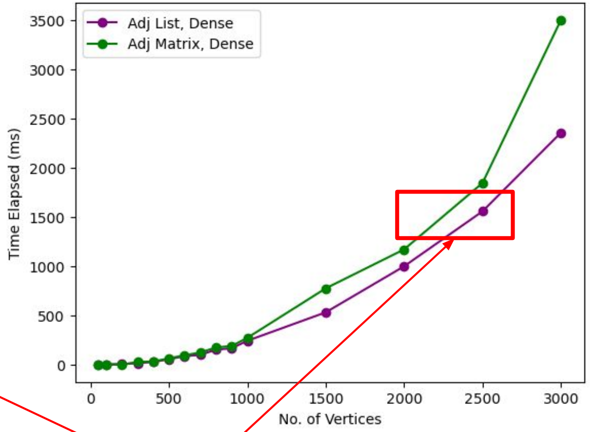



```
c1 (V coefficient): 1.0468023425
c2 (E coefficient): 1.5702035137
```

c) Comparison

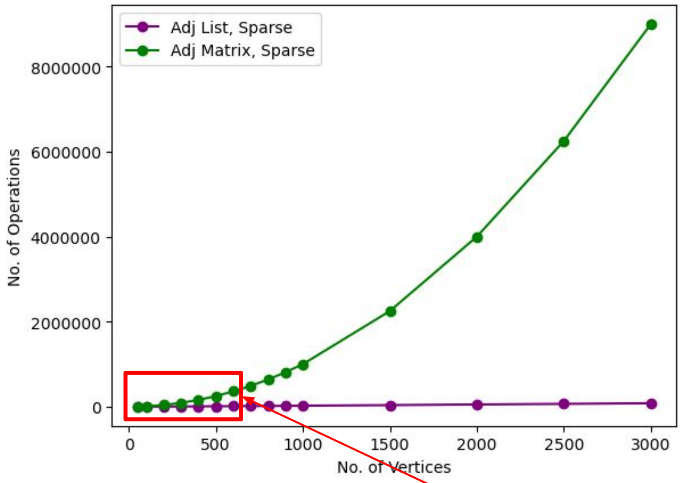
Dense graphs for **Adjacency List + Heap** and the **Adjacency Matrix + Array**

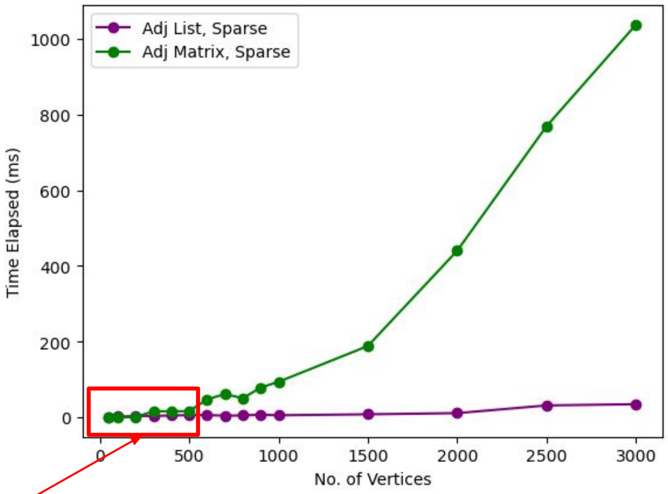




Significant difference

Sparse graphs for **Adjacency List + Heap** and the **Adjacency Matrix + Array**





Negligible difference

Reasons to Still Use an Adjacency Matrix:

- **Space constraints are less critical**, making the larger memory use acceptable in **dense graphs**.
- The **matrix structure simplifies certain algorithms,** especially in dense graphs, where direct edge lookups ($O(1)$) are more useful.
- When $\mathbf{E \approx V^2}$ (very dense graphs), the logarithmic overhead of heap operations in the adjacency list makes the matrix still competitive.
- For small graphs, where the number of vertices is relatively limited, the adjacency matrix can be a simpler, more intuitive structure to use.

Conclusion

Criteria	Adjacency Matrix + Array	Adjacency List + Min-Heap
Graph Size Considerations	Smaller graphs, where heap's logarithmic overhead is less significant	Larger graphs, where efficient memory use and speed are more important
Edge Access	Direct edge access, constant-time lookup for edges	Worst-case edge lookup O(|V|) when a node is connected to all others
Time Complexity	Constant-time edge lookup ($O(1)$) but slower updates for neighbors in the priority queue	Faster overall performance due to lower operational overhead, especially with edge relaxation in larger graphs
Space Efficiency	Uses more space, especially in sparse graphs (stores all possible edges)	More memory-efficient for sparse graphs (stores only existing edges)
Scenarios	When direct edge access is needed and space is not a major concern	When performance (speed) and memory efficiency are crucial for large-scale graphs
Key Takeaway	Competitive in dense graphs or smaller datasets where constant-time edge lookup matters despite space requirements	Ideal for most cases, especially for larger, sparse graphs where speed and memory efficiency are key

Thank you!