Project 2

The Dijkstra's Algorithm

Presented by:

BRANDON JANG JIN TIAN
KEITH LIM EN KAI
GOH ANNA, NINETTE
TEE QIN TONG BETTINA

Table Of Contents

1 Adjacency Matrix Dijkstra’s Algorithm

1. Adjacency Matrix Generation
2. Dijkstra’s Algorithm Implementation (AM)
3. Empirical Analysis of Time Complexity
4. Theoretical Analysis of Time Complexity
5. Comparison Between Empirical and Theoretical Analysis

1

Adjacency Matrix Generation

```python
def generate_directed_adjacency_matrix(V, E, weight_range=(1, 9)):
    if V <= 0 or E < 0 or E > V * (V - 1):
        raise ValueError("Invalid input values")

    # Initialize an empty adjacency matrix filled with zeros
    adjacency_matrix = [[0] * V for _ in range(V)]

    # Generate E random directed edges with random weights within the specified range
    for _ in range(E):
        while True:
            u = random.randint(0, V - 1)
            v = random.randint(0, V - 1)
            if u != v and adjacency_matrix[u][v] == 0:
                break
        weight = random.randint(weight_range[0], weight_range[1])
        adjacency_matrix[u][v] = weight

    return adjacency_matrix
```

1

Adjacency Matrix Generation

```python
# Create an array of matrix with varying numbers of vertices (V)
matrix_vary_V = []
V_values = list(range(5, 50))
constant_edges = 20

for V in V_values:
    adjacency_matrix = generate_directed_adjacency_matrix(V, constant_edges)
    matrix_vary_V.append(adjacency_matrix)
```

1

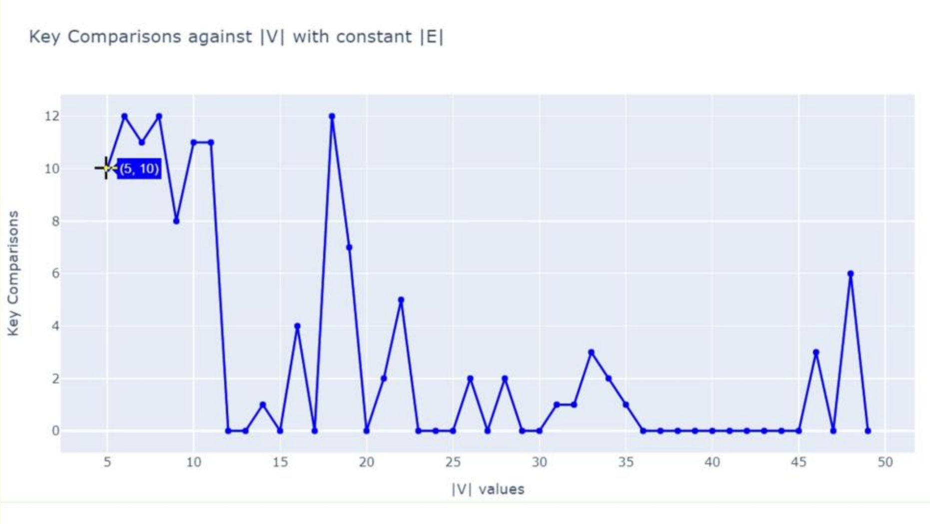
Adjacency Matrix Generation

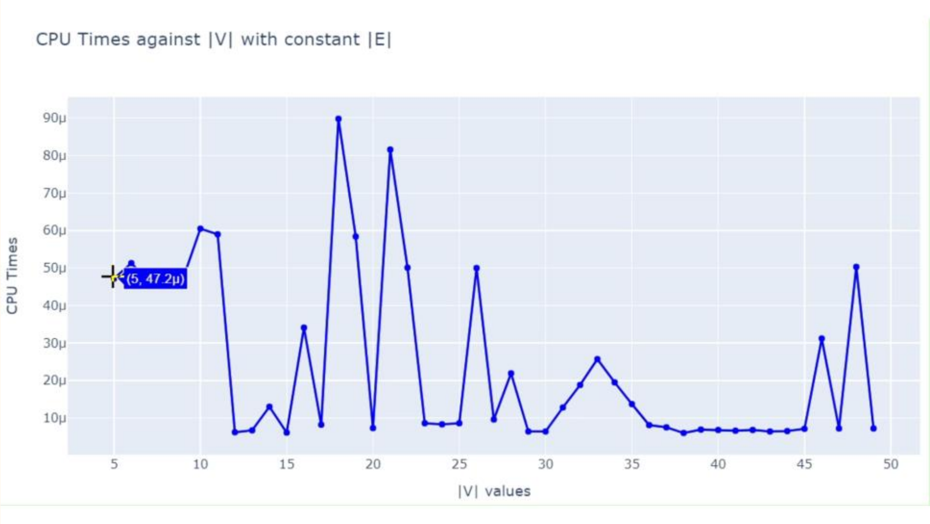
```
# Create an array of matrix with varying numbers of edges (E)
matrix_vary_E_128 = []
constant_vertices = 128
max_edges = constant_vertices * (constant_vertices - 1)
E_values = list(range(1, max_edges+1))

for E in E_values:
    adjacency_matrix = generate_directed_adjacency_matrix(constant_vertices, E)
    matrix_vary_E_128.append(adjacency_matrix)
```

2 Dijkstra’s Algorithm Implementation (AM)

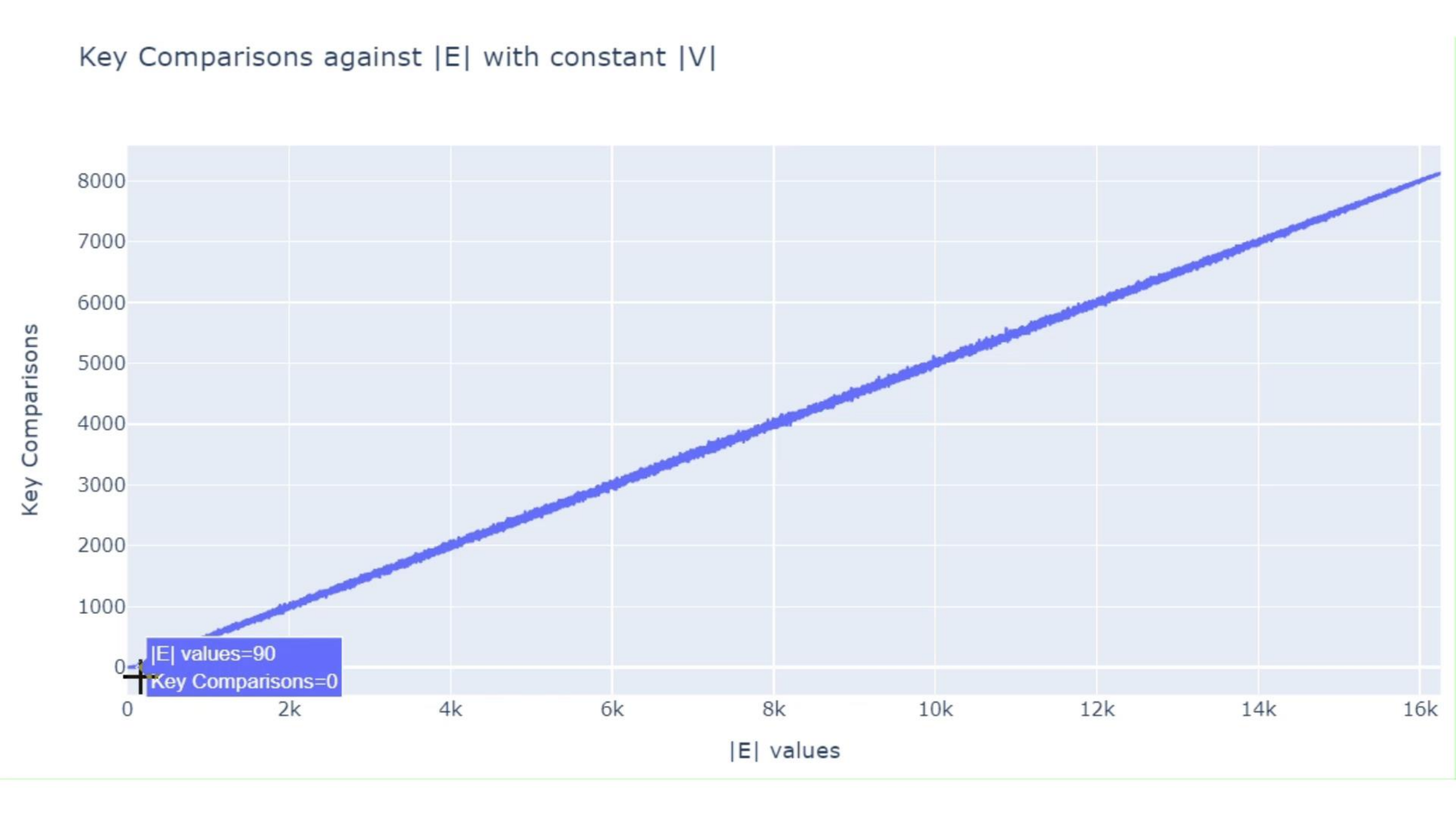
```
def dijkstra_matrix(adjacency_matrix, source):
    num_vertices = len(adjacency_matrix)
    distances = [float('inf')] * num_vertices
    distances[source] = 0
    visited = [False] * num_vertices
    key_comparisons = 0  # Count the number of key comparisons

    start_time = time.perf_counter()  # Use perf_counter for precise CPU time

    # Priority queue implemented as a list of tuples (vertex, distance)
    priority_queue = [(source, 0)]

    while priority_queue:
        # Find the vertex with the minimum distance in the priority queue
        u, min_distance = min(priority_queue, key=lambda x: x[1])
        priority_queue.remove((u, min_distance))  # Remove the vertex from the queue

        # If this vertex has already been visited, skip it
        if visited[u]:
            continue

        visited[u] = True

        for v in range(num_vertices):
            if not visited[v] and adjacency_matrix[u][v] != 0:
                new_distance = distances[u] + adjacency_matrix[u][v]
                key_comparisons += 1  # Increment key comparison count

                if new_distance < distances[v]:
                    distances[v] = new_distance
                    priority_queue.append((v, distances[v]))

    end_time = time.perf_counter()  # Use perf_counter for precise CPU time
    cpu_time = end_time - start_time  # Calculate CPU time

    return distances, key_comparisons, cpu_time
```

3 Empirical Analysis of Time Complexity

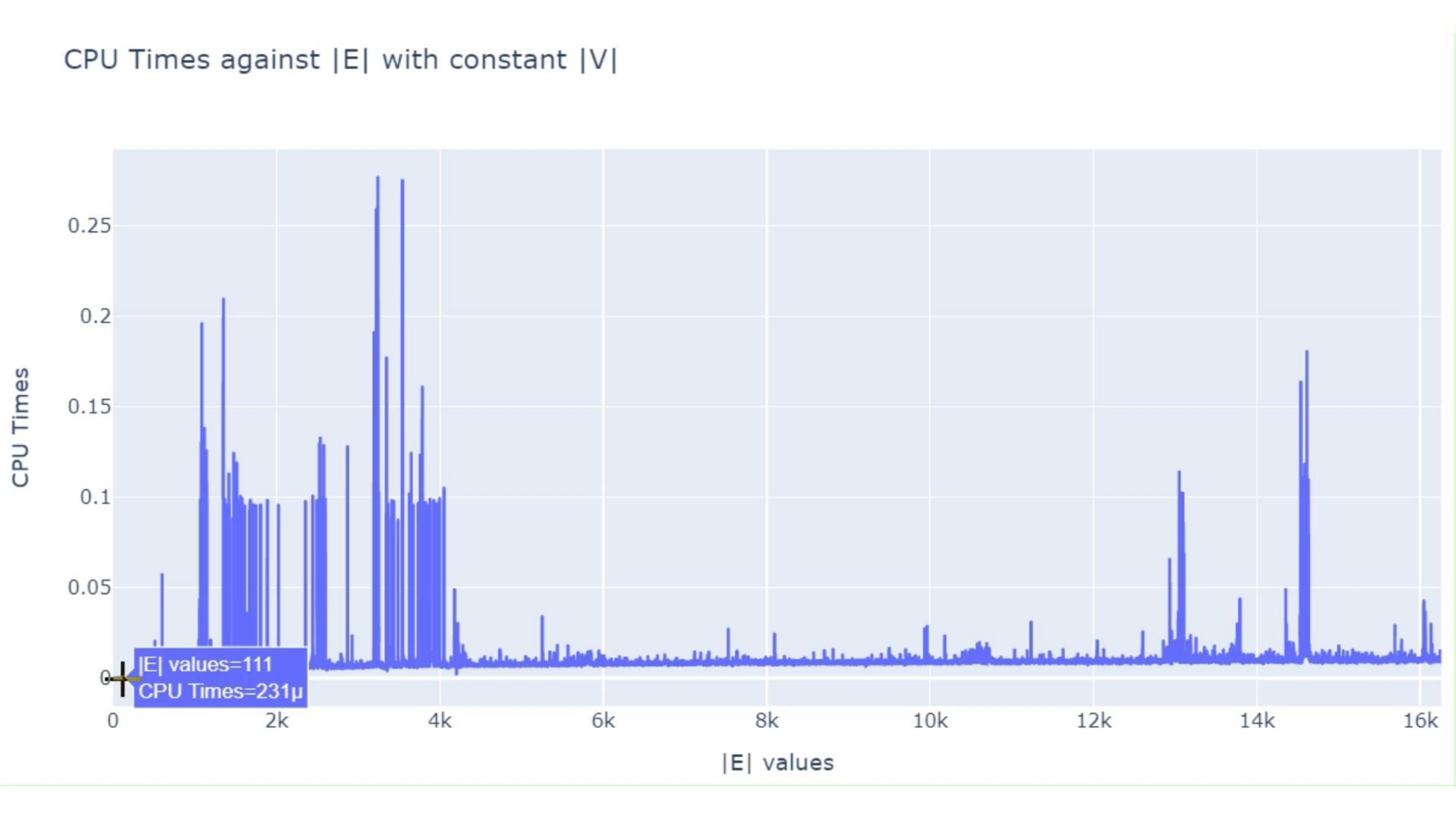




3 Empirical Analysis of Time Complexity



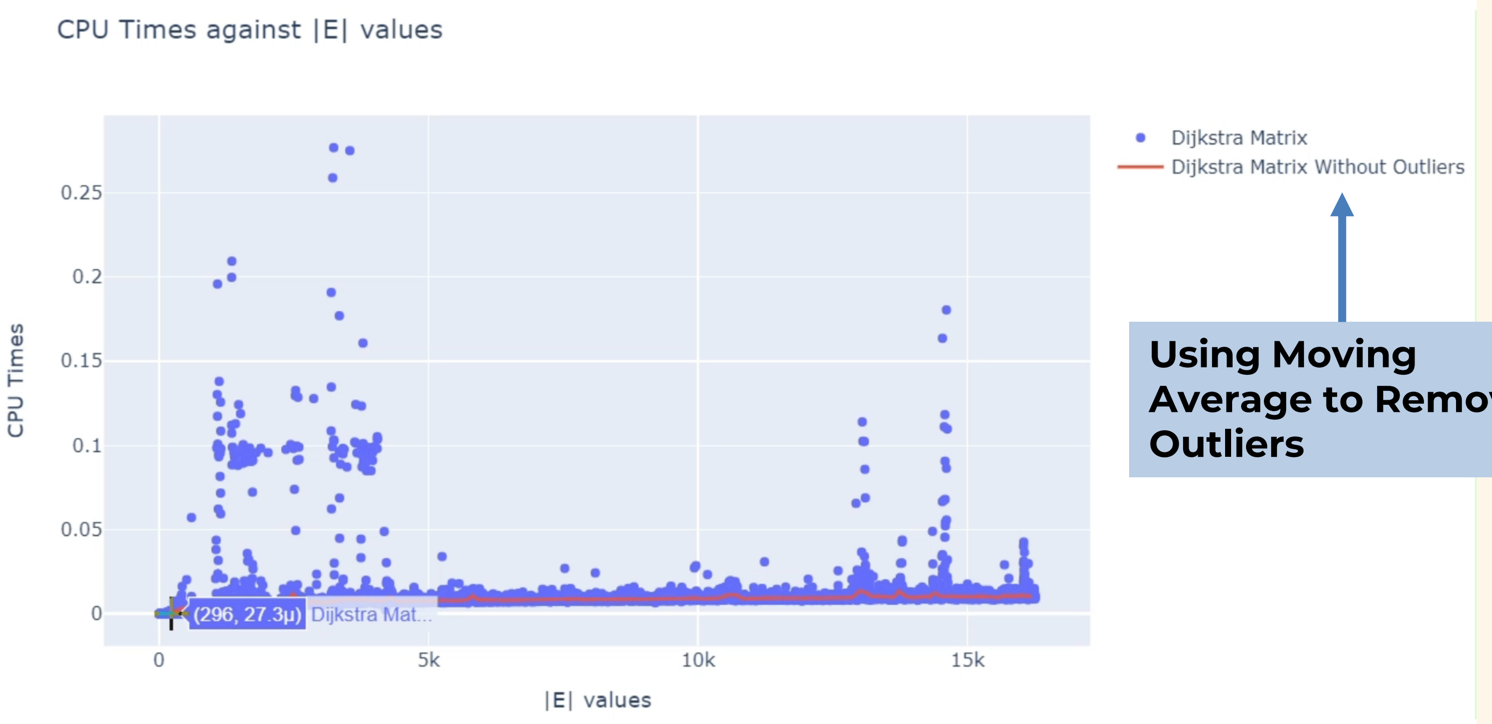
3 Empirical Analysis of Time Complexity



3 Empirical Analysis of Time Complexity

```python
def remove_outliers(cpu_time, v, window_size=5, threshold=2.0):
    # Calculate the moving average of CPU time
    moving_avg = np.convolve(cpu_time, np.ones(window_size) / window_size, mode='same')

    # Calculate the absolute deviation from the moving average
    deviation = np.abs(cpu_time - moving_avg)

    # Calculate the modified Z-score
    median_deviation = np.median(deviation)
    modified_z_scores = 0.6745 * deviation / median_deviation

    # Find the indices of outliers
    outlier_indices = np.where(modified_z_scores > threshold)[0]

    # Remove outliers from both CPU time and v
    cleaned_cpu_time = [cpu_time[i] for i in range(len(cpu_time)) if i not in outlier_indices]
    cleaned_v = [v[i] for i in range(len(v)) if i not in outlier_indices]

    return cleaned_cpu_time, cleaned_v
```

3 Empirical Analysis of Time Complexity



Using Moving Average to Remove Outliers

4 Theoretical Analysis of Time Complexity

```python
# Priority queue implemented as a list of tuples (vertex, distance)
priority_queue = [(source, 0)]

while priority_queue:
    # Find the vertex with the minimum distance in the priority queue
    u, min_distance = min(priority_queue, key=lambda x: x[1])
    priority_queue.remove((u, min_distance))  # Remove the vertex from the queue

    # If this vertex has already been visited, skip it
    if visited[u]:
        continue

    visited[u] = True

    for v in range(num_vertices):
        if not visited[v] and adjacency_matrix[u][v] != 0:
            new_distance = distances[u] + adjacency_matrix[u][v]
            key_comparisons += 1  # Increment key comparison count

            if new_distance < distances[v]:
                distances[v] = new_distance
                priority_queue.append((v, distances[v]))

end_time = time.perf_counter()  # Use perf_counter for precise CPU time
cpu_time = end_time - start_time  # Calculate CPU time
```

O(1)

O(|V|)

O(|V|)

4

Theoretical Analysis of Time Complexity

Complexity of Dijkstra’s Algorithm (Adjacency Matrix)

Initialisation of PQArray + V x Dequeue() + UpdateWeight for All Edges (Worst Case)

$= O(1 + V*V + V*V)$
$= O(2V^2+1)$
$= O(V^2)$, where V is no. of Vertices

4

Theoretical Analysis of Time Complexity

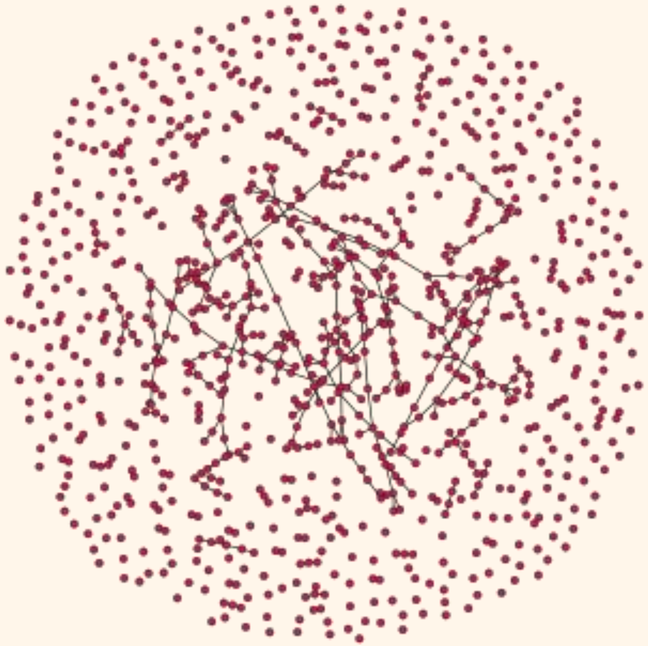
Complexity of Dijkstra's Algorithm (Adjacency Matrix)

Worst Case Scenario: Complete Directed Graph with (|V| x |V-1|) Edges

Best Case Scenario: Directed Weakly Graph which have at least |V-1| Edges

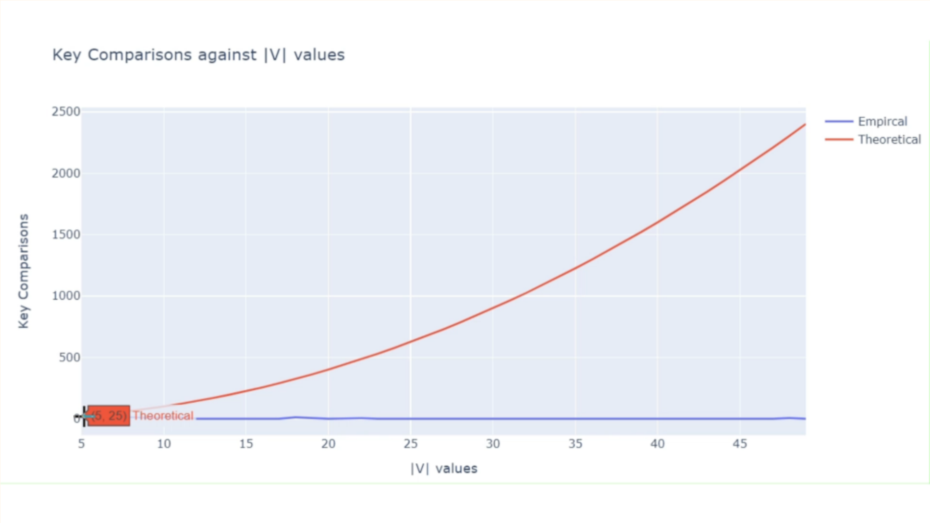
Average Case Scenario: Utilise Erdos-Renyi graphs with randomised weights and probability of edge creation varying from (1/V to V) where V is the no. of vertices

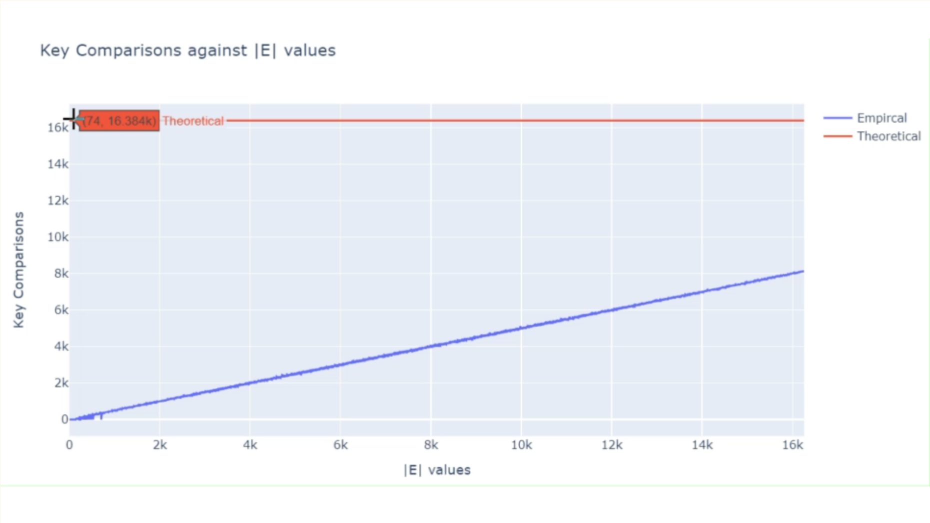
4 Theoretical Analysis of Time Complexity



Average Case Scenario: Utilise Erdos-Renyi graphs with randomised weights and probability of edge creation varying from (1/V to V) where V is the no. of vertices

5 Comparison Between Empirical and Theoretical Analysis





- Empirical Graph Lies Much Lower Than Theoretical Graph
- Corroborates with Our Findings Making It Fair and Justified Outcome

2 Adjacency List Dijkstra’s Algorithm

1. Adjacency List Generation
2. Dijkstra’s Algorithm Implementation (AL)
3. Empirical Analysis of Time Complexity
4. Theoretical Analysis of Time Complexity
5. Comparison Between Empirical and Theoretical Analysis

1 Adjacency List Generation

```python
def adjacency_matrix_to_list(adjacency_matrix):
    num_vertices = len(adjacency_matrix)
    adjacency_list = [[] for _ in range(num_vertices)]

    for u in range(num_vertices):
        for v in range(num_vertices):
            weight = adjacency_matrix[u][v]
            if weight != 0:
                adjacency_list[u].append((v, weight))

    return adjacency_list
```

```python
# Create an array of list converted from matrix
list_vary_V = []

for idx, matrix in enumerate(matrix_vary_V):
    adjacency_list = adjacency_matrix_to_list(matrix)
    list_vary_V.append(adjacency_list)
```

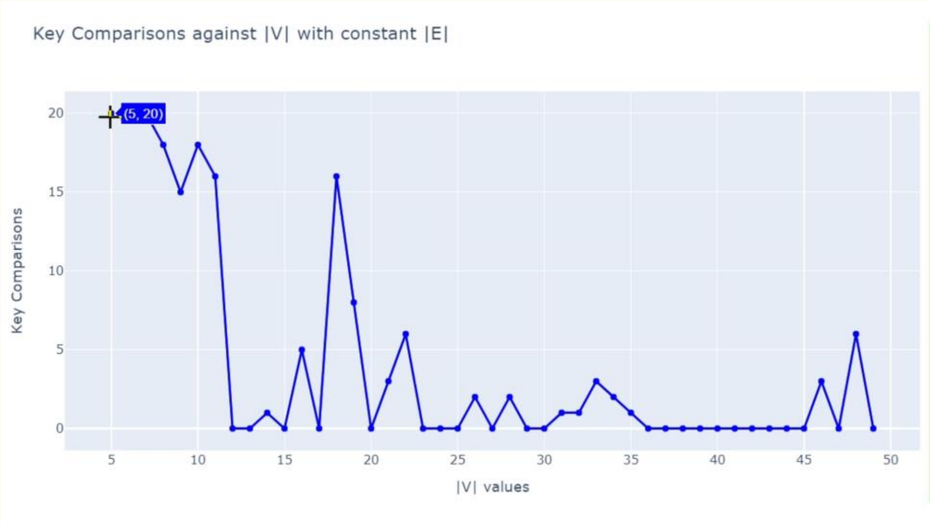
```python
# Create an array of list converted from matrix
list_vary_E = []

for idx, matrix in enumerate(matrix_vary_E):
    adjacency_list = adjacency_matrix_to_list(matrix)
    list_vary_E.append(adjacency_list)
```

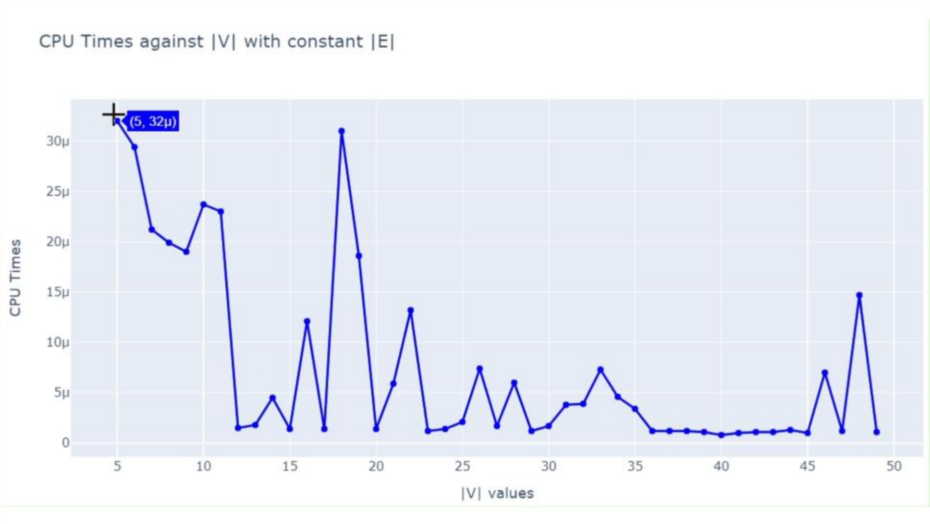
2 Dijkstra’s Algorithm Implementation (AL)

Adjacency List: List for keeping track of the connections between edges E and vertices V in a Graph G

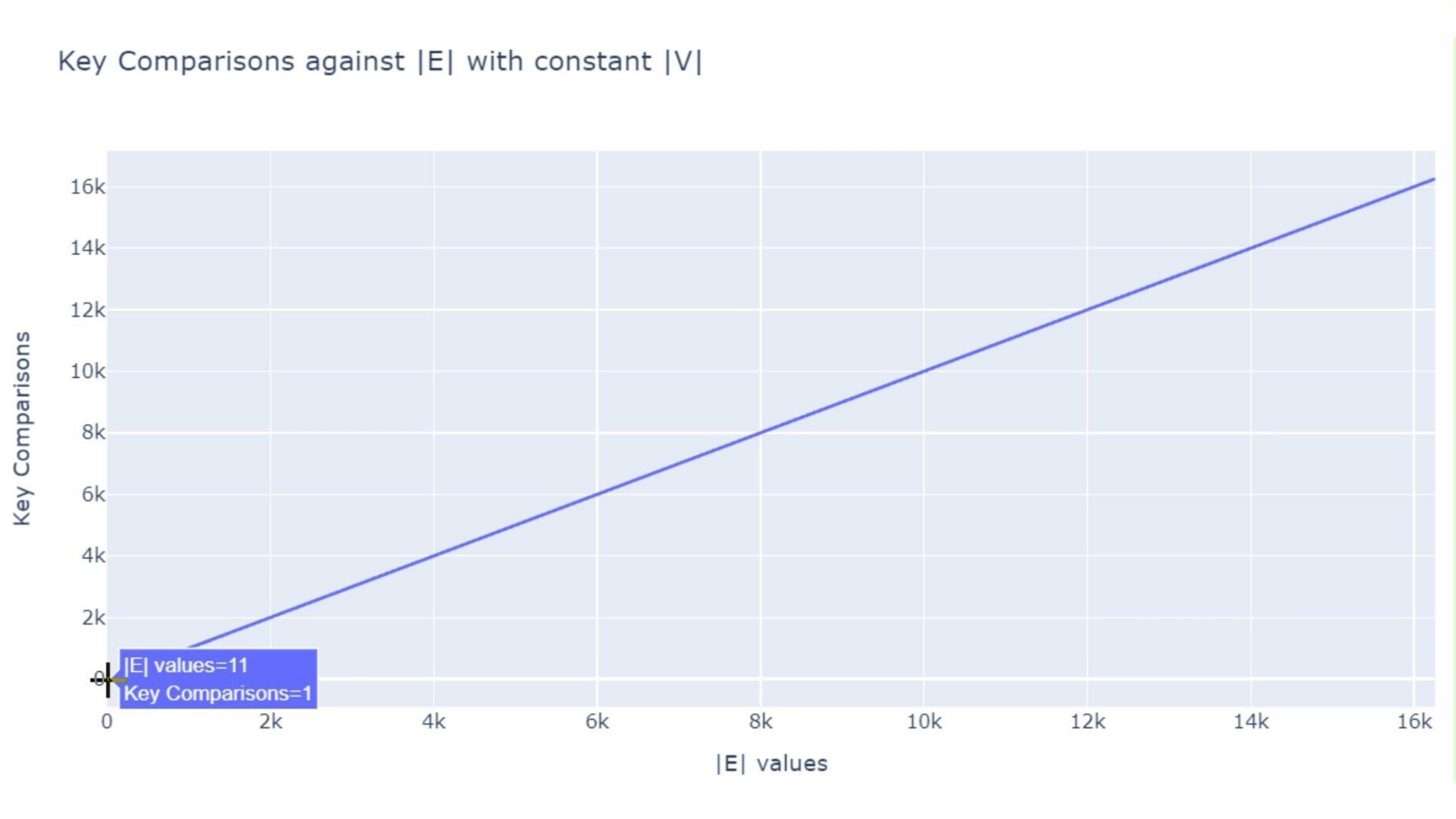
```python
def dijkstra_list(adjacency_list, source):
    num_vertices = len(adjacency_list)
    distances = [float('inf')] * num_vertices
    distances[source] = 0
    visited = [False] * num_vertices
    priority_queue = [(0, source)]
    key_comparisons = 0  # Count the number of key comparisons

    start_time = time.perf_counter()  # Record the start time

    while priority_queue:
        dist_u, u = heapq.heappop(priority_queue)

        # If we've already processed this vertex, skip it.
        if visited[u]:
            continue

        visited[u] = True

        for v, weight in adjacency_list[u]:
            key_comparisons += 1  # Increment key comparison count
            new_distance = distances[u] + weight

            if new_distance < distances[v]:
                distances[v] = new_distance
                heapq.heappush(priority_queue, (distances[v], v))

    end_time = time.perf_counter()  # Record the end time
    cpu_time = end_time - start_time  # Calculate CPU time

    return distances, key_comparisons, cpu_time
```

3 Empirical Analysis of Time Complexity



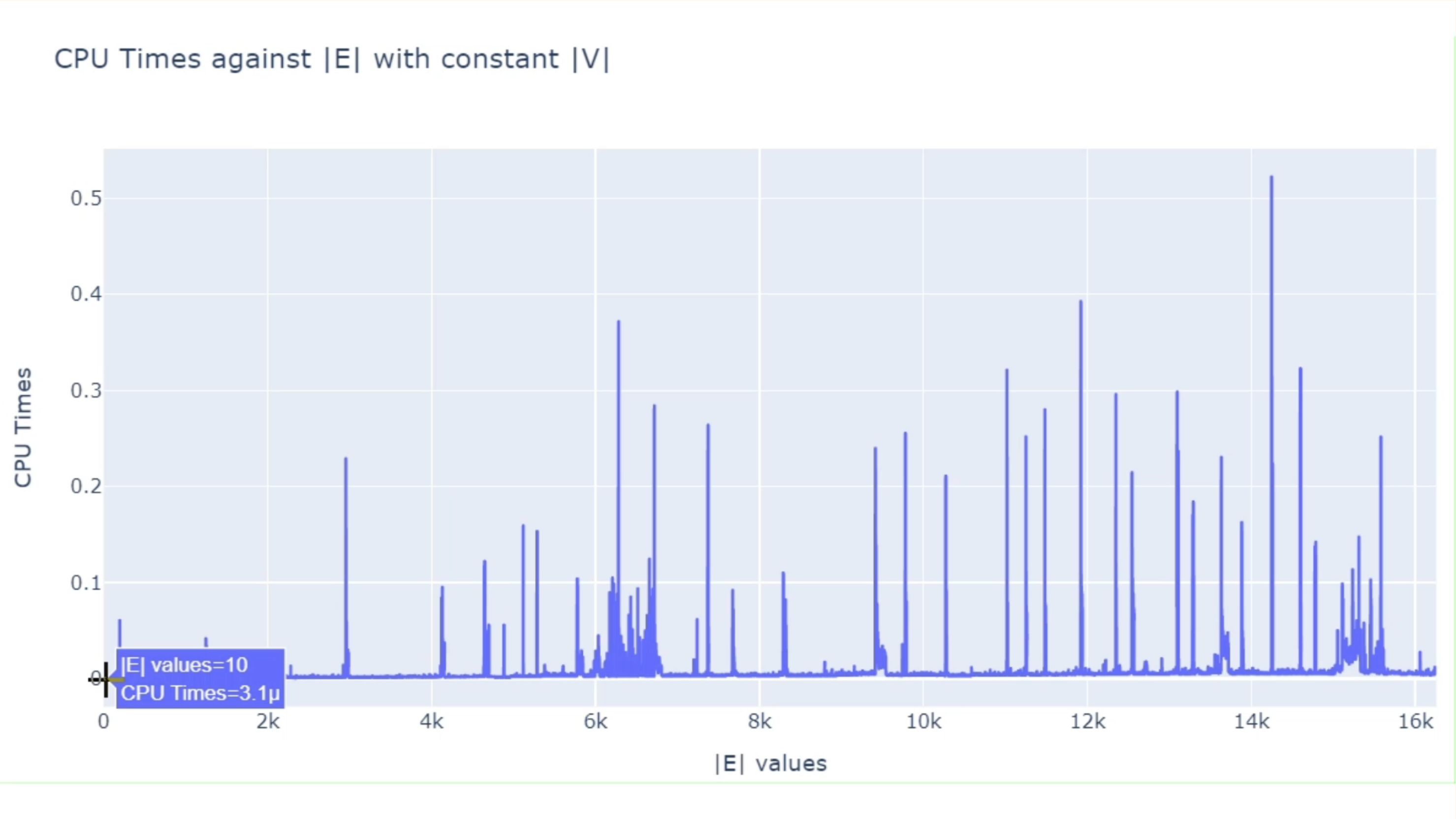


3 Empirical Analysis of Time Complexity



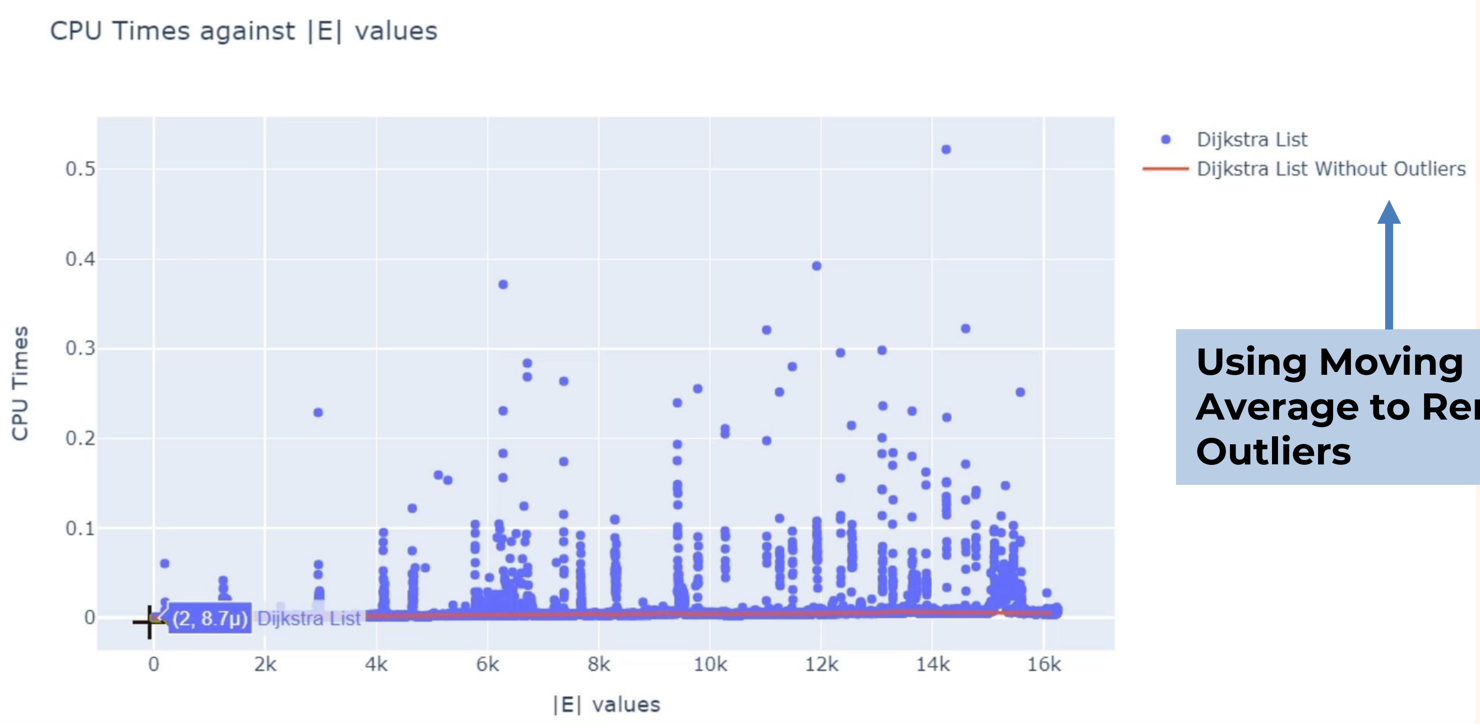
3

Empirical Analysis of Time Complexity



3

Empirical Analysis of Time Complexity



Using Moving Average to Remove Outliers

4 Theoretical Analysis of Time Complexity

Complexity of Dijkstra's Algorithm (Adjacency List)

Initialisation of PQHeap + V * Dequeue() + UpdateWeight of all Edges (Worst case)

$= O(|V| + V * \text{Log } |V| + E * \text{Log } |V|)$

$= O(|V| + ((V + E) * \text{Log } |V|)$

$= O((V + E) * \text{Log } |V|)$

Assuming a weakly connected graph, must have at least |V| - 1 edges. This implies that |V| is in O(|E|), simplifies to:

$= O(E \text{ Log } |V|)$, where V is no. of Vertices and E is no. of Edges.

4

Theoretical Analysis of Time Complexity

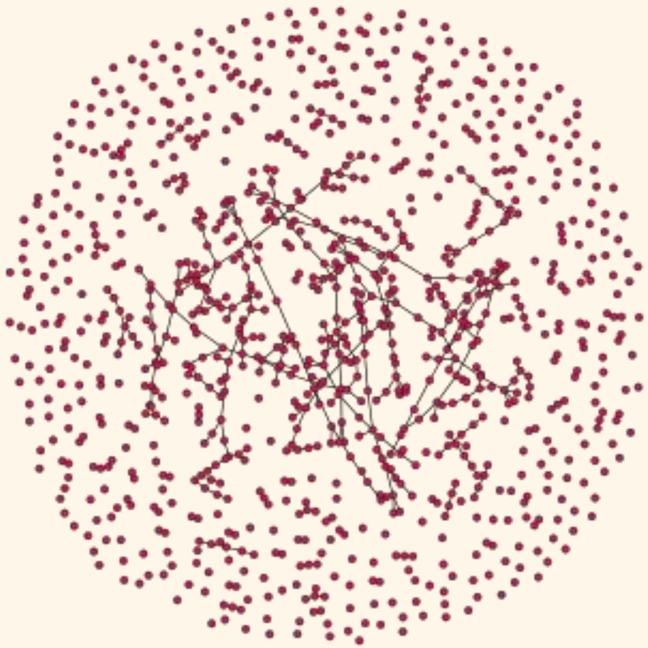
Complexity of Dijkstra’s Algorithm (Adjacency List)

Worst Case Scenario: Complete Directed Graph with (|V| x |V-1|) Edges

Best Case Scenario: Directed Weakly Graph which have at least |V-1| Edges

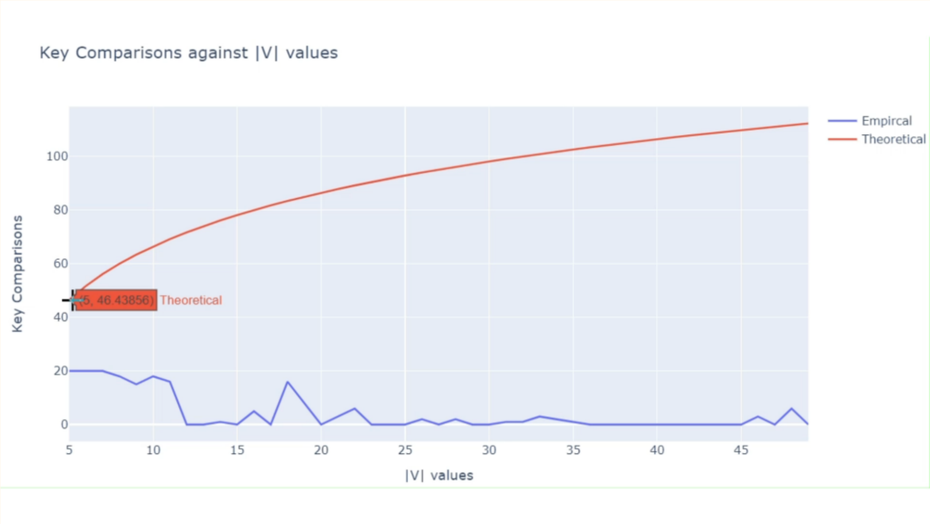
Average Case Scenario: Utilise Erdos-Renyi graphs with randomised weights and probability of edge creation varying from (1/V to V) where V is the no. of vertices

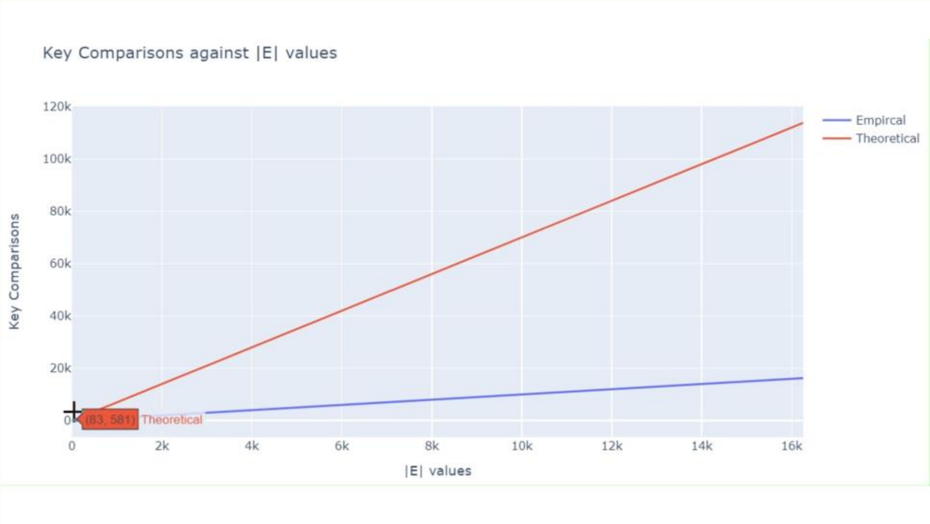
4 Theoretical Analysis of Time Complexity



Average Case Scenario: Utilise Erdos-Renyi graphs with randomised weights and probability of edge creation varying from (1/V to V) where V is the no. of vertices

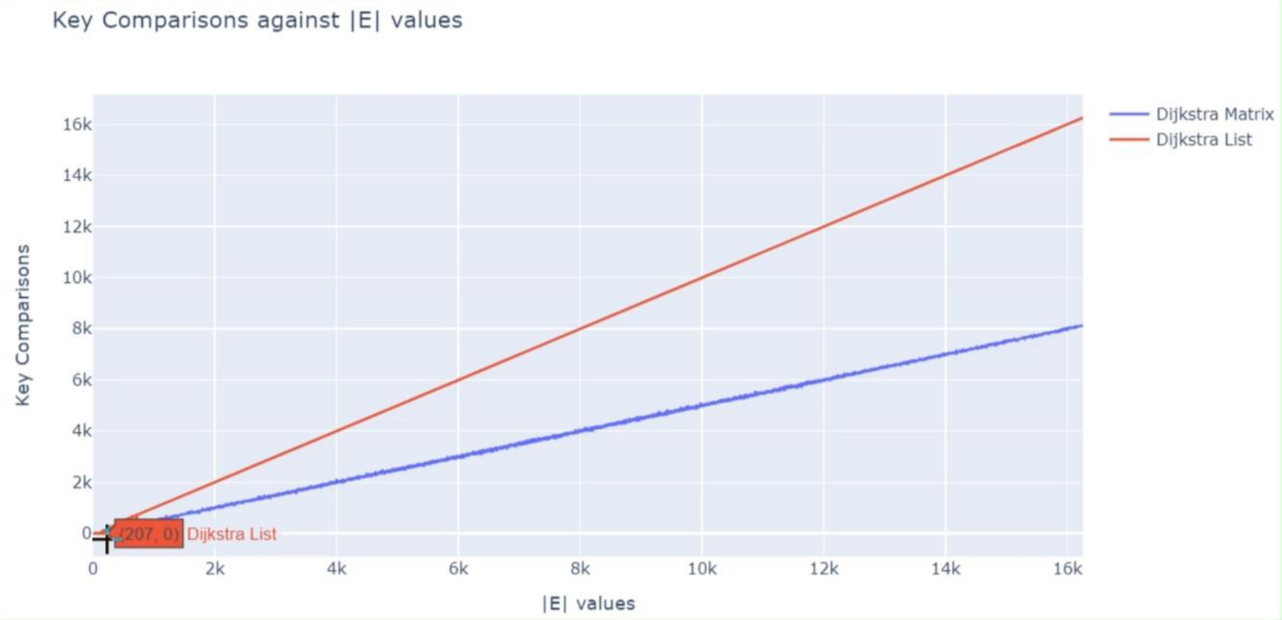
5 Comparison Between Empirical and Theoretical Analysis





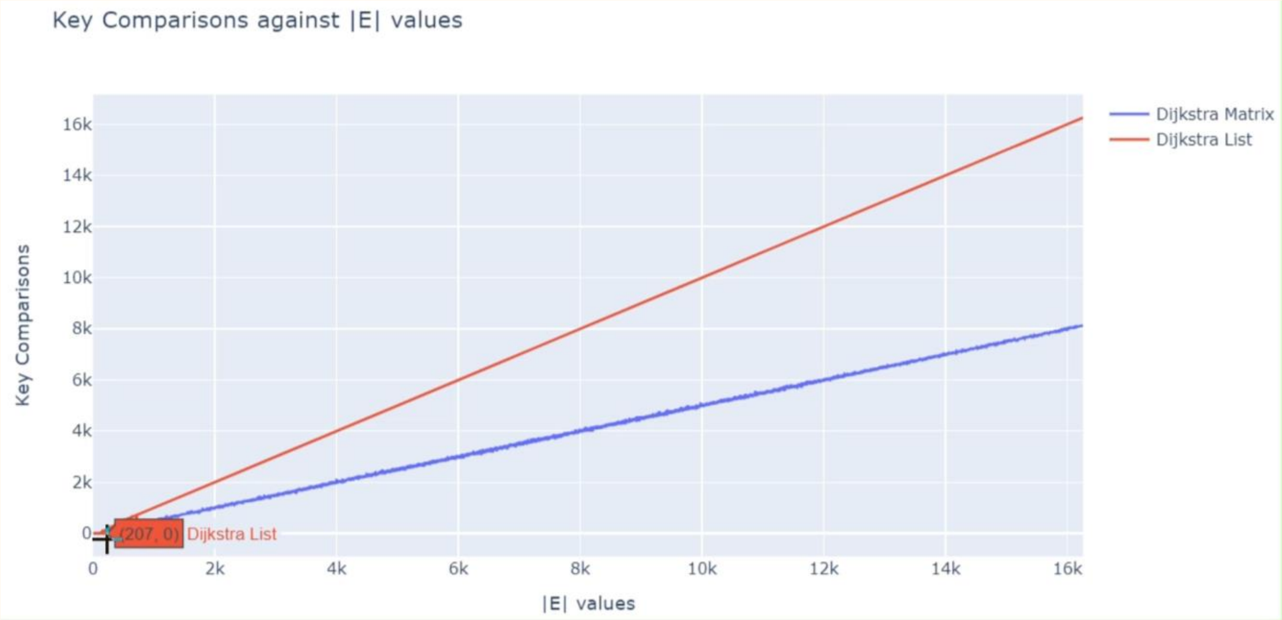
- Large Difference between Empirical and Theoretical
- Empirical Graph Lies Much Lower Than Theoretical Graph

3 Implementation Comparison



- No. Of Comparisons for List is Lesser When Graph is Sparse
- No. Of Comparisons for Matrix is Lesser When Graph is Dense

3 Implementation Comparison



- For Sparse Graphs, Adjacency Lists with a Min-Heap are Generally More Efficient in Terms of Both Time and Space
- For Dense Graphs, Adjacency Matrices with an Array are Generally More Efficient in Terms of Constant-Time Edge Lookup [O(1)], but Space Inefficient