SC2001 PROJECT 2

BY ATHENA, ARUN, AFREEN

Start

CONTENT

ABSTRACT

This project aims to explore the impact of different graph representations and priority queue implementations on the time complexity of Dijkstra's algorithm

(a) Input graph G stored in an adjacency matrix and we use an array for the priority queue

(b) Input graph G is stored in an array of adjacency lists and we use a minimizing heap for the priority queue

(c) Comparing the 2 implementation in (a) and (b) and conclude which implementation is better in what circumstances

DIJKSTRA'S PSEUDOCODE

```
Dijkstra_ShortestPath(Graph G, Node source):
  for each vertex v in G:
    d[v] = infinity          <------  Initialize distances to infinity
    p[v] = null              <------  Predecessor of each vertex is initially null
    S[v] = 0                 <------  S[v] indicates whether v is in the shortest path set (S)

  d[source] = 0              <------  Distance to the source is set to 0

  Q = PriorityQueue()        <------  Initialize priority queue with all vertices,
                                      where the priority is their distance d[v]

  for each vertex v:
    Q.insert(v, d[v])
```

DIJKSTRA'S PSEUDOCODE

```
while Q is not empty:
    u = ExtractCheapest(Q)              <----  Get the vertex u with the minimum distance
    S[u] = 1                            <----  Mark vertex u as part of the shortest path set

    for each vertex v adjacent to u:    <----  Update distances for all adjacent vertices of u
        if S[v] == 0 and d[v] > d[u] + w(u, v):  <----  Only update if vertex v is not in S
            d[v] = d[u] + w(u, v)       <----  Update distance
            p[v] = u                    <----  Update predecessor
            Q.update(v, d[v])           <----  Update the priority queue

//End of while loop
```

DIJKSTRA'S SOURCE CODE

```c
// Dijkstra's algorithm for a graph represented using an adjacency matrix
void Dijkstra_ShortestPath(int graph[V][V], int source) {
    int d[V];      // d[v] will hold the shortest distance from source to vertex v
    int pi[V];     // pi[v] will hold the predecessor of vertex v in the shortest path
    int S[V];      // S[v] is 1 if vertex v is in the shortest path set, otherwise 0

    // Initialize all distances as INFINITE and predecessors as null (-1), S[] as false
    for (int i = 0; i < V; i++) {
        d[i] = INT_MAX;
        pi[i] = -1;  // -1 denotes no predecessor
        S[i] = 0;    // Vertex v is not yet processed
    }

    // Distance from the source to itself is always 0
    d[source] = 0;

    // Put all vertices in the priority queue Q, While its not empty
    for (int count = 0; count < V - 1; count++) {
        // Extract the vertex u with the minimum distance from Q
        int u = ExtractCheapest(d, S);

        // If the minimum distance vertex is -1, then all reachable vertices have been processed
        if (u == -1) break;

        // Mark vertex u as processed (add it to set S)
        S[u] = 1;

        // For each vertex v adjacent to u
        for (int v = 0; v < V; v++) {
            // Check if there is an edge from u to v, v is not processed, and
            // if the distance to v can be minimized through u
            if (!S[v] && graph[u][v] && d[u] != INT_MAX
                && d[u] + graph[u][v] < d[v]) {
                d[v] = d[u] + graph[u][v];  // Update d[v]
                pi[v] = u;                  // Update predecessor of v
            }
        }
    }

    // Print the shortest distances and predecessors
    printSolution(d, pi);
}
```

```c
// Function to find the vertex with the minimum distance value
int ExtractCheapest(int d[], int S[]) {
    int min = INT_MAX, min_index = -1;

    for (int v = 0; v < V; v++) {
        if (!S[v] && d[v] <= min) {
            min = d[v];
            min_index = v;
        }
    }

    return min_index;
}
```

GENERATING GRAPHS - ALGO IMPLEMENTATION

```
FUNCTION generate_random_graph(V, E):

  CREATE graph as a VxV matrix filled with zeros

  edges_added = 0

  WHILE edges_added < E:

    u = RANDOM integer between 0 and V-1
    v = RANDOM integer between 0 and V-1
```

Initialize a V x V matrix with zeros

Initialize edges counter to 0

A while loop runs until the desired number of edges has been added. Each iteration attempts to add a new edge to the graph.

Two vertices u and v are randomly selected

GENERATING GRAPHS - ALGO IMPLEMENTATION

IF u is not equal to v AND graph[u][v] == 0:

1ST test to ensure that the vertices u and v to be different i.e no self loop
2ND test to ensure there isn't a already existing loop between vertices u and v

weight = RANDOM integer between 1 and 9

Randomly generate a weight to the edge (between 1 and 9)

graph[u][v] = weight

Add the weight

edges_added += 1

Edge counter incremented

Return the generated graph
RETURN graph

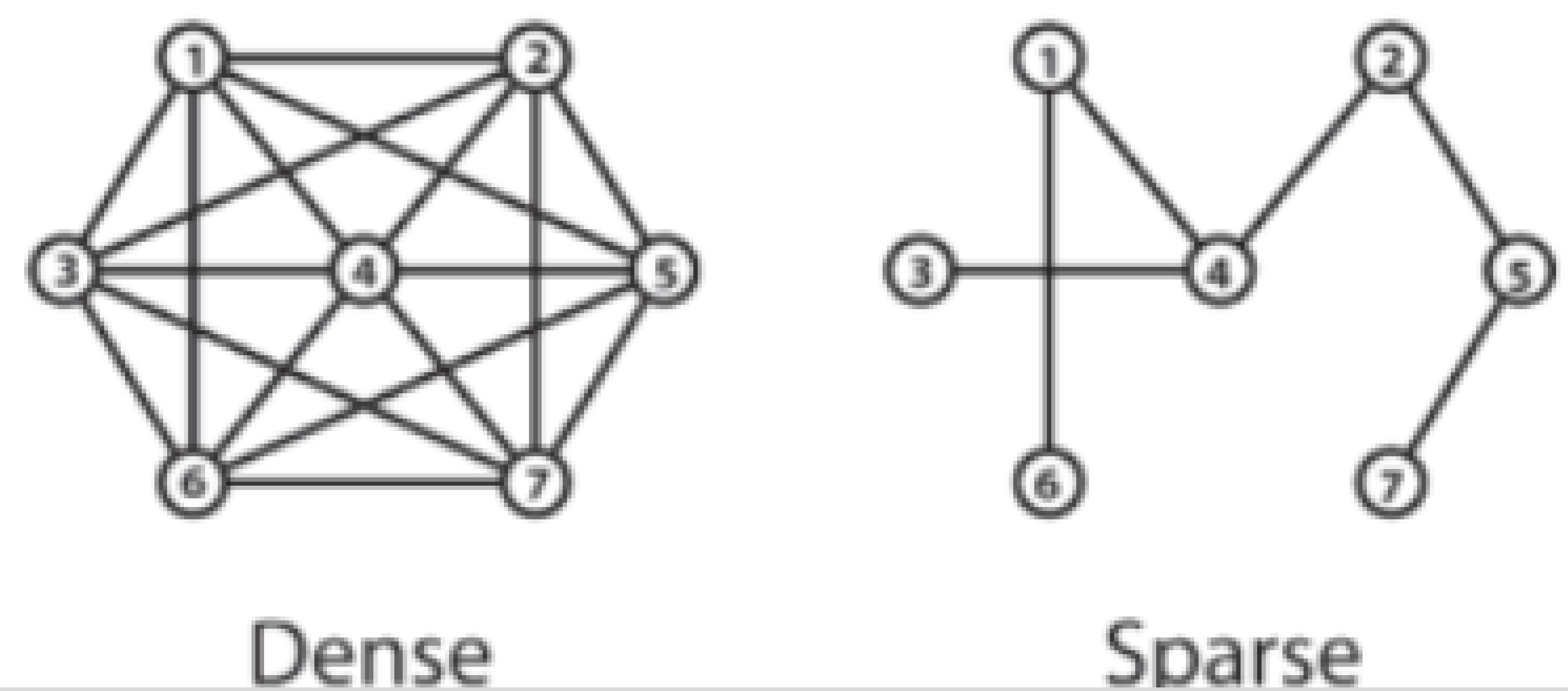
DENSE VS SPARSE

$$E = \frac{v\left(v-1\right)}{2}$$

- For undirected graphs

$$E = v\left(v-1\right)$$

- For directed graphs



Denser graphs has high connectivity compared to sparser graphs making denser graphs have density E/v^2 closer to one

ADJACENCY MATRIX AND ARRAY

1. Adjacency Matrix Representation:
A 2D array where the weight of the edge between any two vertices u and v is stored at matrix[u][v].

2. Array-Based Priority Queue:
A simple array to store the distances. The vertex with the smallest distance is selected by scanning the entire array which takes $O(|V|)$ time.

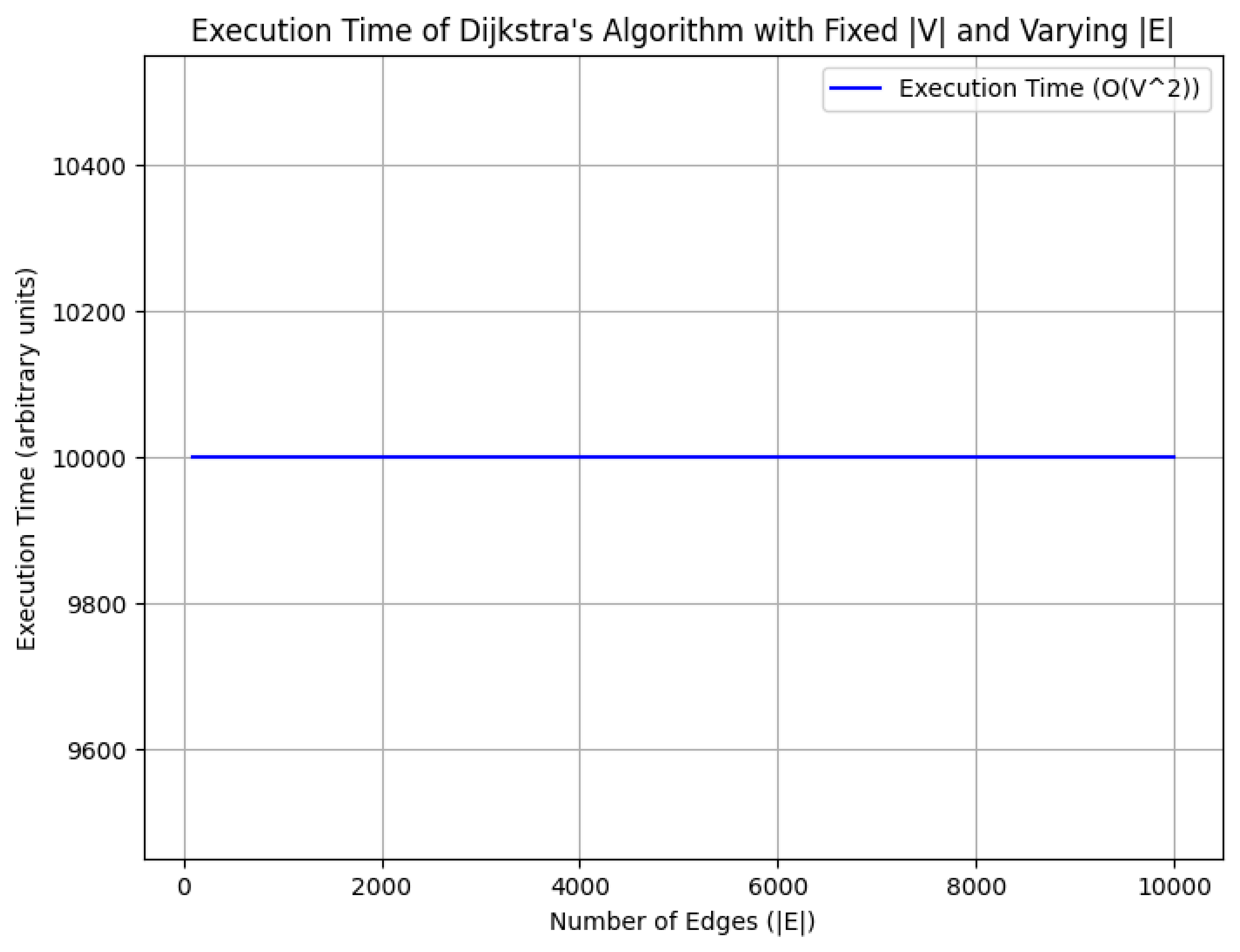
THEORETICAL ANALYSIS WHEN V IS FIXED AND E IS VARIED

1. Main Loop:
    - Iterates over all vertices: **O(|V|)**
    - Finding the minimum distance vertex (using an array-based queue): **O(|V|) per iteration.**
    - Total for all vertices: **O(|V|^2)**
2. Updating Neighbors:
    - Each vertex checks O(|V|) neighbors in the adjacency matrix.
    - Total: **O(|V|^2)**
3. Overall Time Complexity:
    - O(|V|^2): Constant regardless of the number of edges (|E|) as long as the number of vertices (|V|) is fixed.
4. Graph Density:
    - This time complexity indicates that whether the graph is dense (many edges) or sparse (few edges), the runtime remains the same with a fixed |V| since it's dominated by vertex count, not edge count.

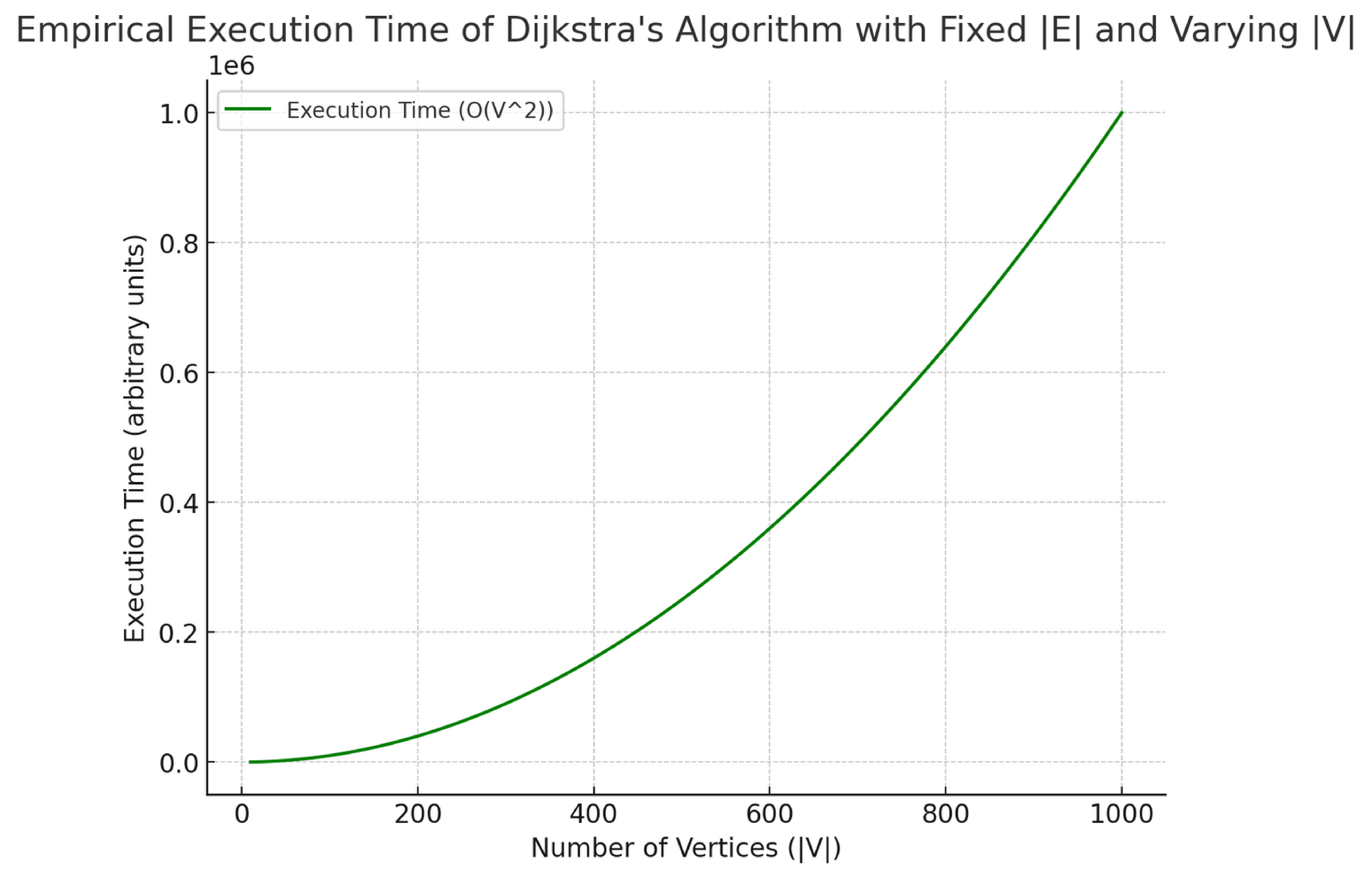
THEORETICAL ANALYSIS WHEN E IS FIXED AND V IS VARIED

1. Total Time Complexity:
   - Overall: **O(|V|^2).**
2. Impact of Fixed |E|:
   - With a fixed |E|, the complexity remains O(|V|^2), indicating it mainly depends on the number of vertices |V|.
   - The time complexity scales quadratically with |V|, not |E|.
3. Graph Density (Sparse vs. Dense):
   - Sparse Graph: Few edges relative to vertices—runtime remains O(|V|^2) since it's dominated by vertex count, not edge count.
   - Dense Graph: Many edges relative to vertices—runtime still O(|V|^2) as the adjacency matrix approach focuses on vertices, making it unaffected by edge density.

EMPIRICAL ANALYSIS WHEN V IS FIXED AND E IS VARIED



EMPIRICAL ANALYSIS WHEN E IS FIXED AND V IS VARIED



ARRAY OF ADJACENCY LISTS AND MINIMISING HEAP

Array of Adjacency Lists

- An array of adjacency lists represents a graph by using an array where each index corresponds to a vertex, and each element contains a list of its neighboring vertices and the edge weights connecting them.
- It allows fast traversal of the neighbors of each vertex, making it ideal for algorithms like Dijkstra's.

ARRAY OF ADJACENCY LISTS AND MINIMISING HEAP

Minimising Heaps

- A special tree-based data structure that is used to implement a priority queue.
- Allows for efficient extraction of the smallest element, which is essential in algorithms like Dijkstra's where you repeatedly select the vertex with the smallest tentative distance.

THEORETICAL ANALYSIS

Graph Representation (Adjacency List): Each edge (u,v) is listed once, resulting in a space complexity of O(V+E).

Min Heap:

- Push — O(logV)
- Pop — O(logV)

Overall Time Complexity:

- Initialization: O(V)
- Vertex Extraction (Pop): Each vertex is extracted once; O(VlogV)
- Relaxation: Each edge is relaxed at most once, and for each relaxation, insert into the heap; O(ElogV).

O((V+E)logV)

THEORETICAL ANALYSIS

Dense

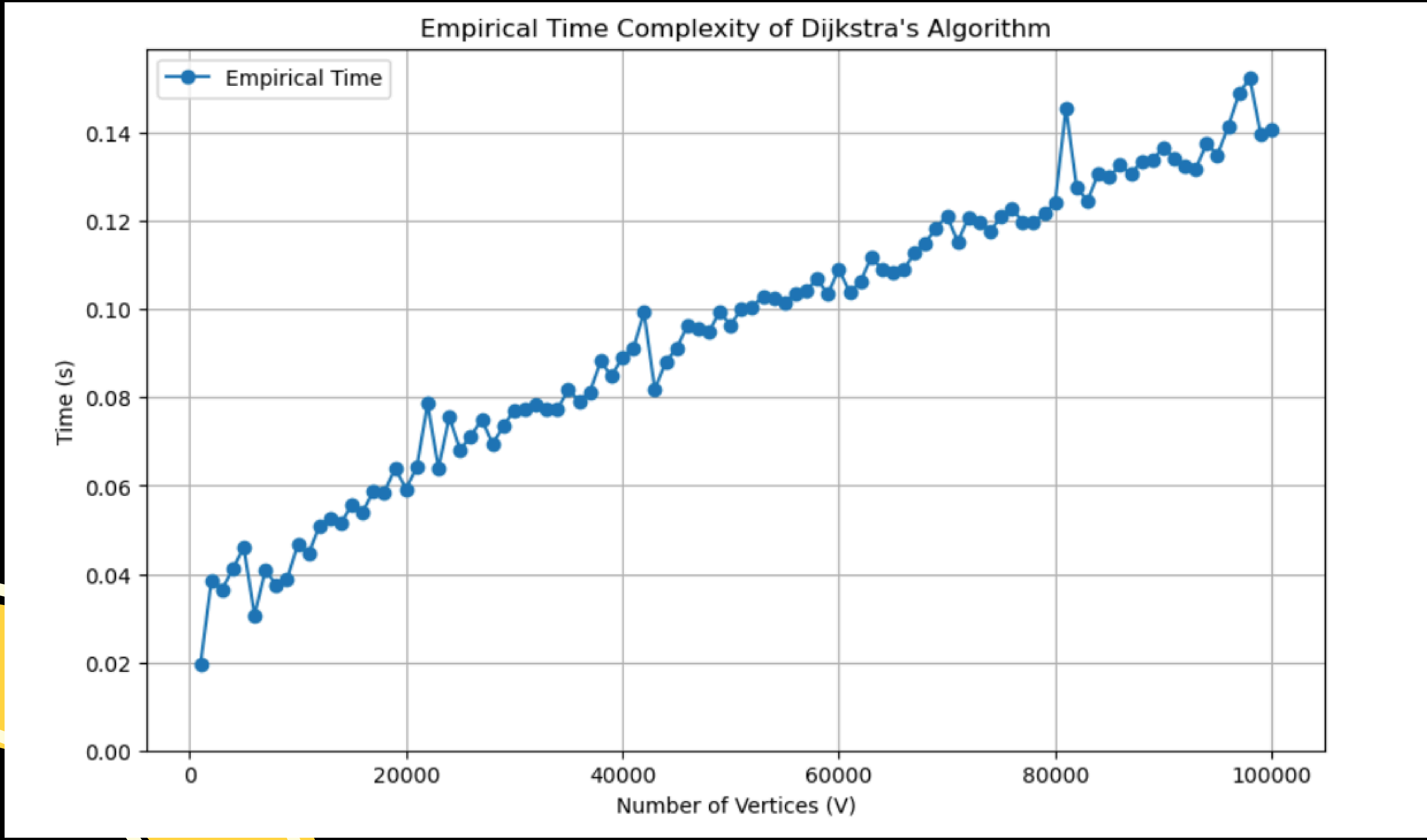
- As the graph gets denser, the number of edges E approaches V^2 (the maximum possible number of edges in a complete graph). In this case, the time complexity approaches

O (V^2 log V)

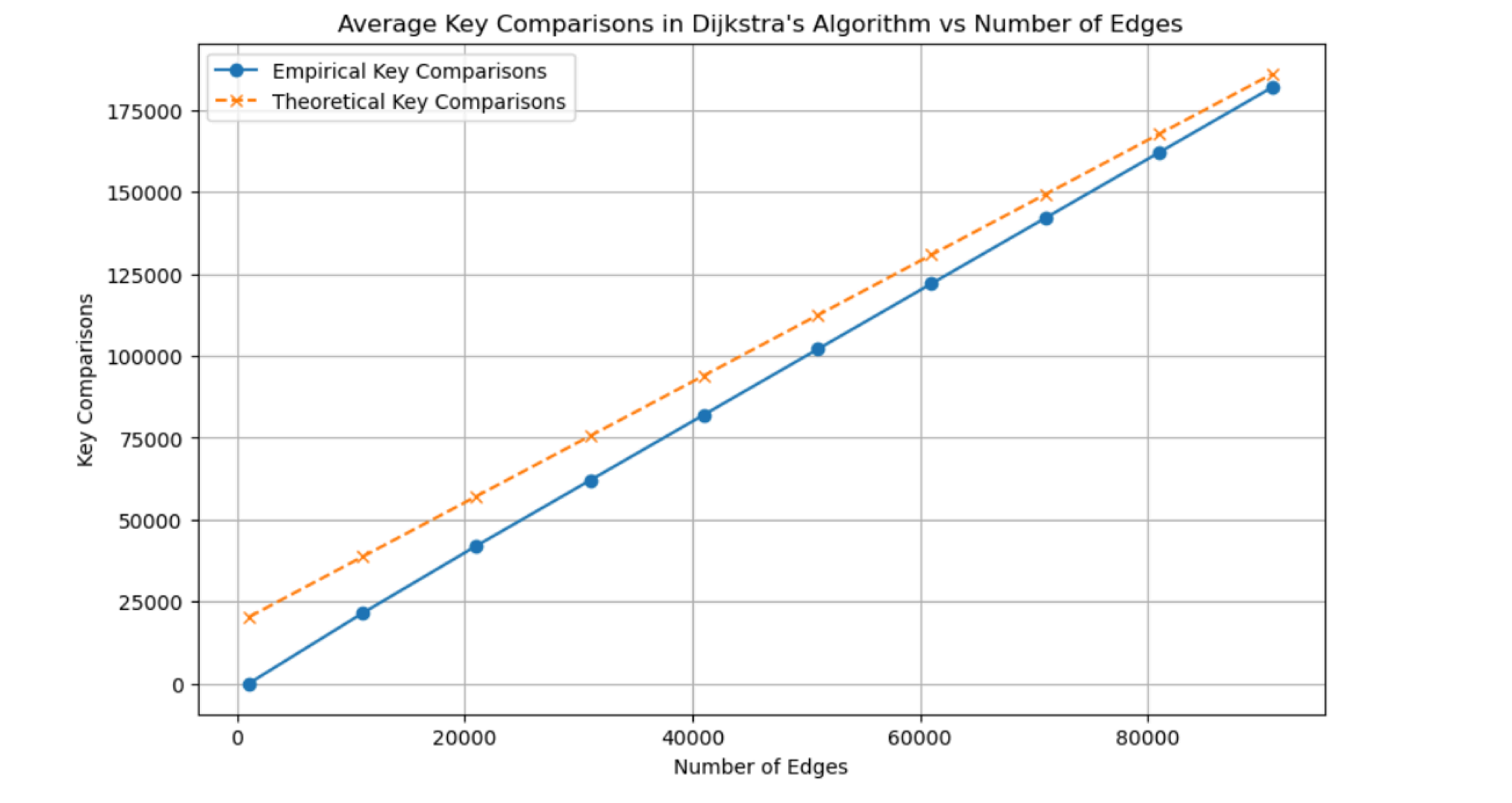
Sparse

- When the graph is sparse, E is much smaller than V^2. The complexity remains close to O(VlogV)).

EMPIRICAL ANALYSIS WHEN E IS FIXED AND V IS CHANGED

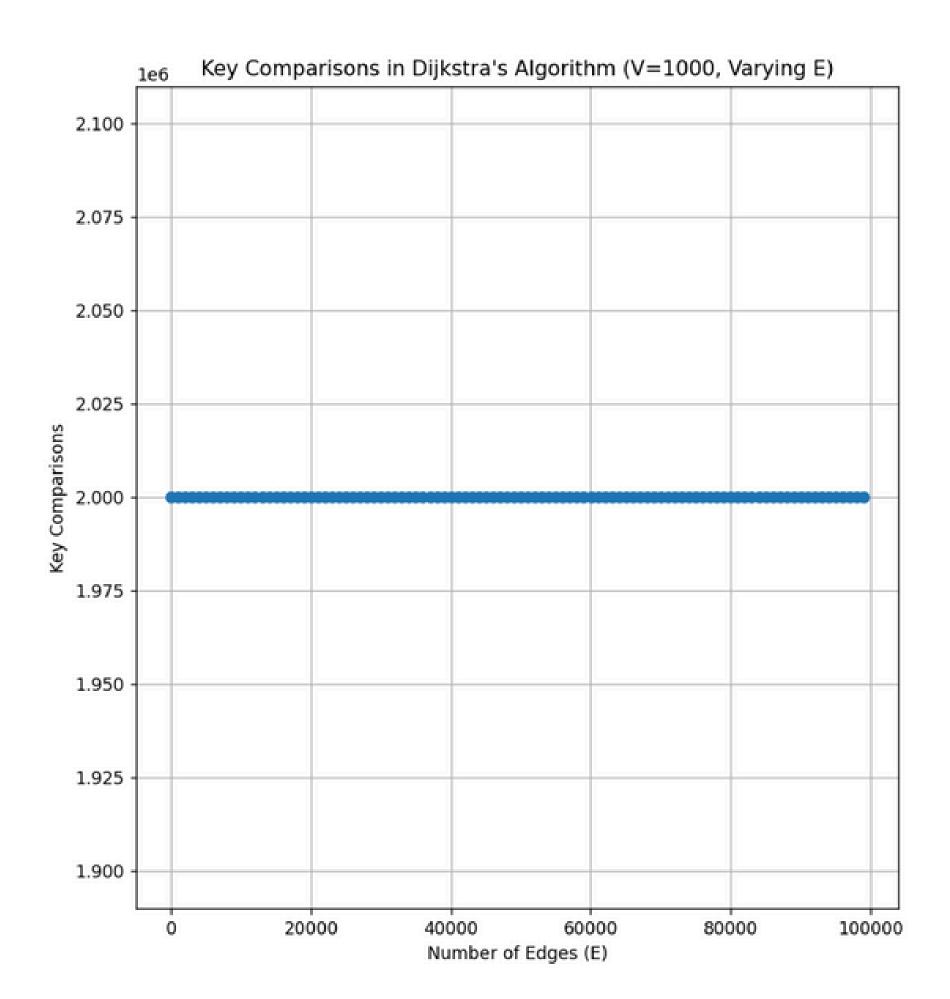


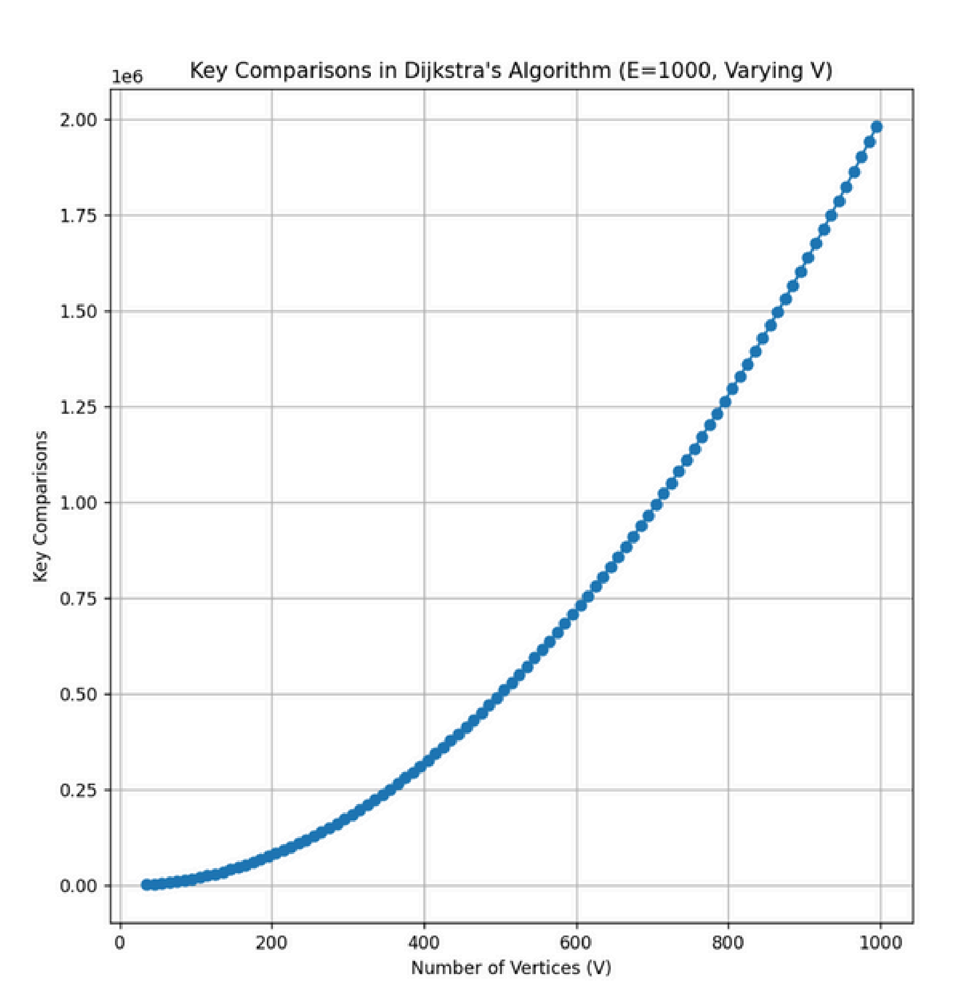
EMPIRICAL ANALYSIS WHEN V IS FIXED AND E IS CHANGED



Implementation(a):

- Just like the execution time, the key comparisons also showed very similar trends
- This implementation is mostly dependent on V rather than on E.





Implementation(b):

- This implementation is mostly dependent on E and a varying V doesn't affect its performance much.

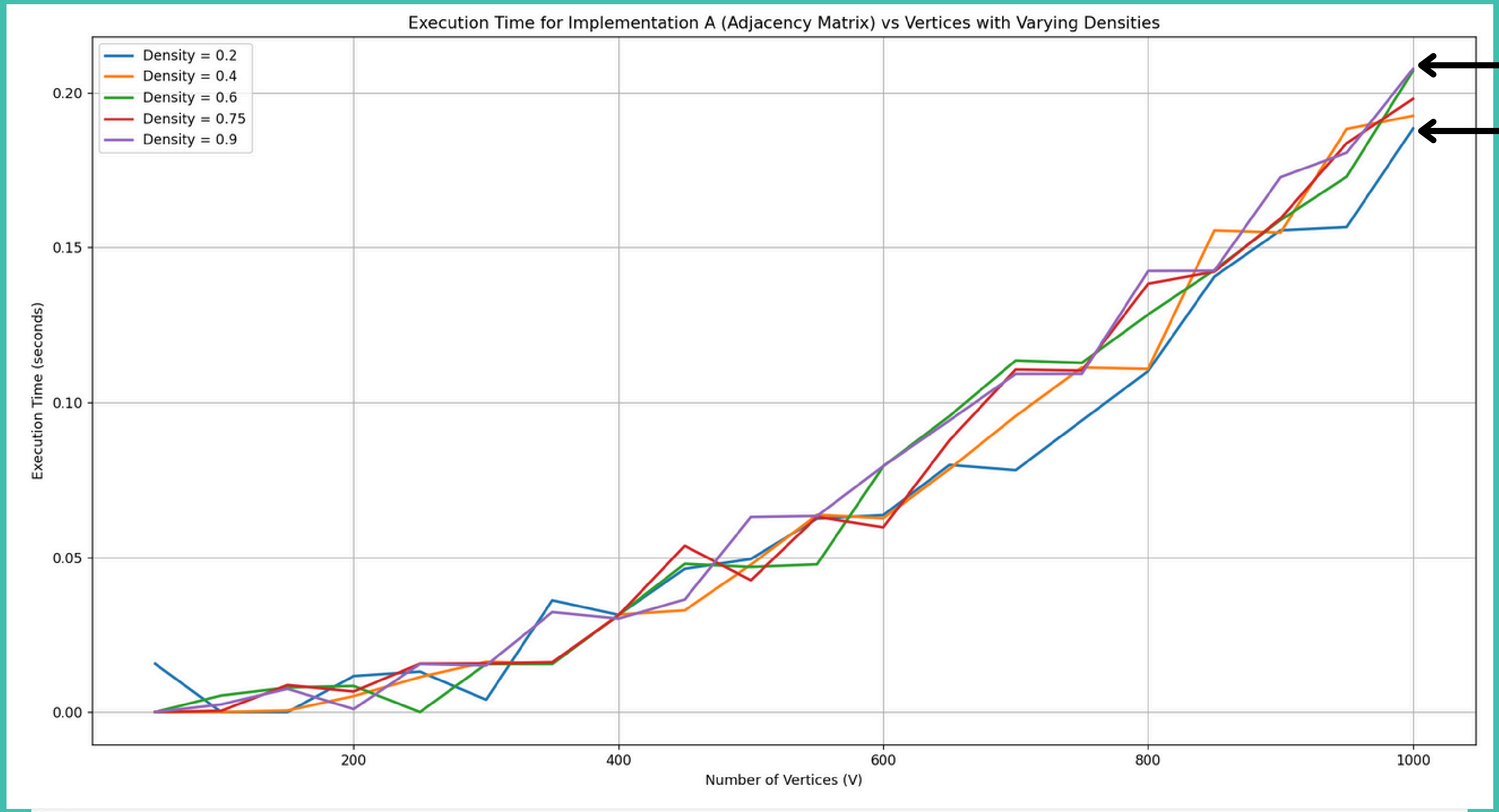
- Even as V increases against a fixed E, the execution time only increases fairly making the overall performance still remain stable.

- Implementation A is more suitable for denser graphs(E is large) where the relationship between vertices and edges is high

$$E = \frac{v\left(v-1\right)}{2}$$ - For undirected graphs

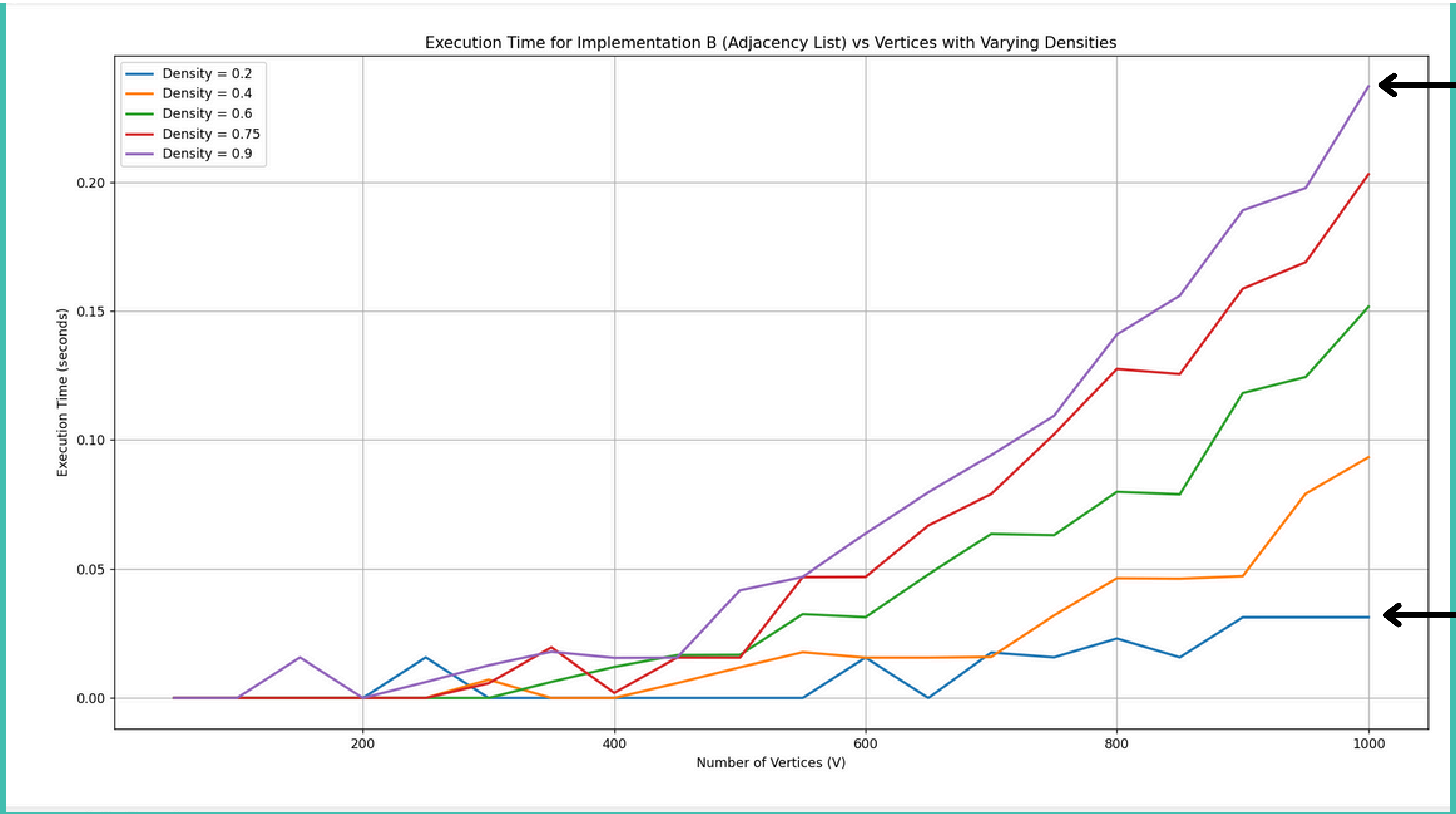
$$E = v\left(v-1\right)$$ - For directed graphs

- Implementation B is more suitable for sparse graphs where the number of edges are limited relative to the number of vertices.



0.9 density : ~0.21

0.2 density : ~0.18



0.9 density : ~0.24

0.2 density : ~0.03

THANK YAAAAA