SC2001 Lab 2

Group 3:

Karthik Raj U2320917J

Ishita Jain U2321668B

Jayakumar Thirunithiyan U2321424H

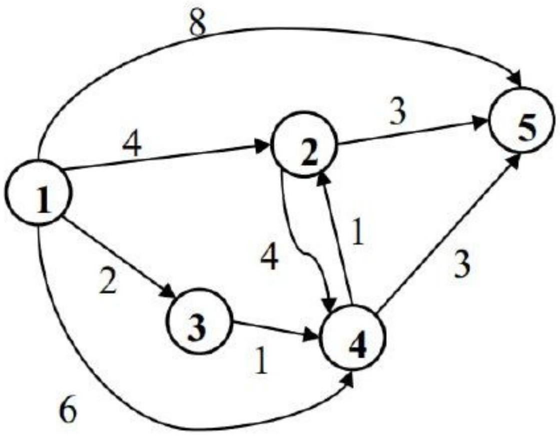
Table of contents

01

Adjacency Matrix implementation

Adjacency Matrix formation

$$
\begin{array}{c c}
\text{vertex} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
0 & 4 & 2 & 6 & 8 \\
\infty & 0 & \infty & 4 & 3 \\
\infty & \infty & 0 & 1 & \infty \\
\infty & 1 & \infty & 0 & 3 \\
\infty & \infty & \infty & \infty & 0
\end{bmatrix}
\end{array}
$$



V² Slots in Adjacency Matrix

Therefore, time complexity is **V²** for generation

```python
# Function to generate a sparse or dense directed graph
def generate_directed_graph(vertices, edges):
    graph = [[0 for _ in range(vertices)] for _ in range(vertices)]
    edge_count = 0

    while edge_count < edges:
        u = random.randint(0, vertices - 1)
        v = random.randint(0, vertices - 1)
        if u != v and graph[u][v] == 0:
            weight = random.randint(1, 10)
            graph[u][v] = weight  # Directed graph, only u -> v
            edge_count += 1

    return graph
```

Our code for generating the graph

```python
# Dijkstra's algorithm using adjacency matrix for directed graph
def dijkstra_matrix_directed(graph, src):
    vertices = len(graph)
    dist = [sys.maxsize] * vertices
    dist[src] = 0
    visited = [False] * vertices

    for _ in range(vertices):
        min_distance = sys.maxsize
        min_index = -1

        for v in range(vertices):
            if not visited[v] and dist[v] < min_distance:
                min_distance = dist[v]
                min_index = v

        u = min_index
        visited[u] = True

        for v in range(vertices):
            if graph[u][v] > 0 and not visited[v] and dist[v] > dist[u] + graph[u][v]:
                dist[v] = dist[u] + graph[u][v]

    return dist
```

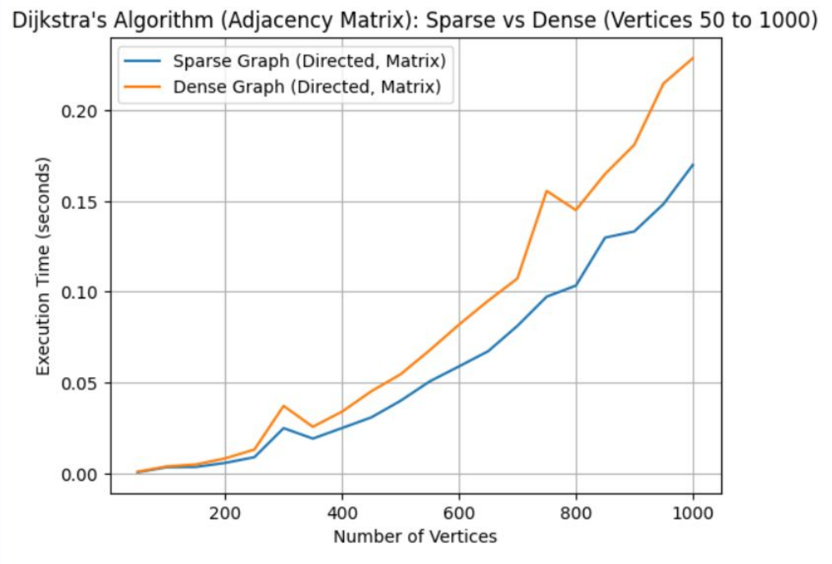
Our code that is implementing the algorithm

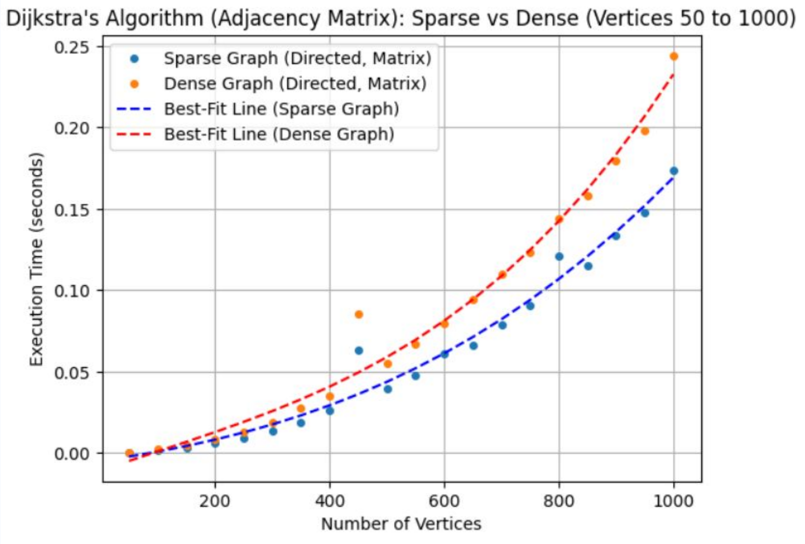
```python
# Measure the execution time for varying vertices for sparse and dense directed graphs
def measure_time_matrix_directed(vertex_range):
    time_results_sparse = []
    time_results_dense = []

    for v in vertex_range:
        sparse_edges = v - 1  # Sparse graph with v-1 edges
        dense_edges = v * (v - 1)  # Dense graph with max edges for directed graph

        sparse_graph = generate_directed_graph(v, sparse_edges)
        dense_graph = generate_directed_graph(v, dense_edges)

        # Measure time for sparse graph
        start_time = time.time()
        dijkstra_matrix_directed(sparse_graph, 0)
        time_results_sparse.append(time.time() - start_time)

        # Measure time for dense graph
        start_time = time.time()
        dijkstra_matrix_directed(dense_graph, 0)
        time_results_dense.append(time.time() - start_time)

    return time_results_sparse, time_results_dense
```

Our code to compute the runtime and return that value

```python
ertices from 50 to 1000 in increments of 50 with best-fit line
with_fit():
range(50, 1001, 50))  # Vertices from 50 to 1000 in increments of 50
nse = measure_time_matrix_directed(vertex_range)

    # Fit best-fit lines (polynomial regression) with degree 3
    sparse_fit = polynomial_fit(vertex_range, time_sparse, 3)
    dense_fit = polynomial_fit(vertex_range, time_dense, 3)

    # Plot original data
    plt.plot(vertex_range, time_sparse, 'o', label="Sparse Graph (Directed, Matrix)", markersize=4)
    plt.plot(vertex_range, time_dense, 'o', label="Dense Graph (Directed, Matrix)", markersize=4)

    # Plot best-fit lines
    plt.plot(vertex_range, sparse_fit, label="Best-Fit Line (Sparse Graph)", linestyle='--', color='blue')
    plt.plot(vertex_range, dense_fit, label="Best-Fit Line (Dense Graph)", linestyle='--', color='red')

    plt.xlabel('Number of Vertices')
    plt.ylabel('Execution Time (seconds)')
    plt.title('Dijkstra\'s Algorithm (Adjacency Matrix): Sparse vs Dense (Vertices 50 to 1000)')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_results_part_a_with_fit()
```

Our code to plot the graph






```python
# Function to add polynomial best-fit line
def polynomial_fit(x, y, degree):
    coefficients = np.polyfit(x, y, degree)
    polynomial = np.poly1d(coefficients)
    return polynomial(x)
```

02

Time complexity Adjacency Matrix and Array Queue

Theoretical Analysis

Overall Time Complexity

$= (V-1)(V+V) + \mathbf{V^2} = 3\mathbf{V^2} - 2V$

$= O(\mathbf{V^2})$

(V-1)

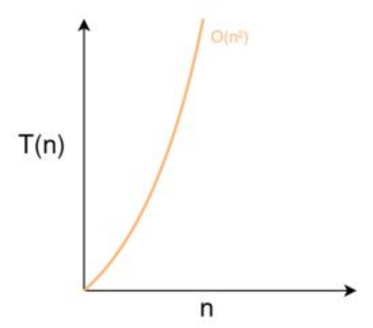
When checking all the potential vertices, pick the minimum distance vertex from the set of vertices not yet processed

(V+V)

First V -> finding the minimum distance vertex.
Second V -> updating the distances for the adjacent vertices of the chosen vertex

V^2

Derived the adjacency matrix



03

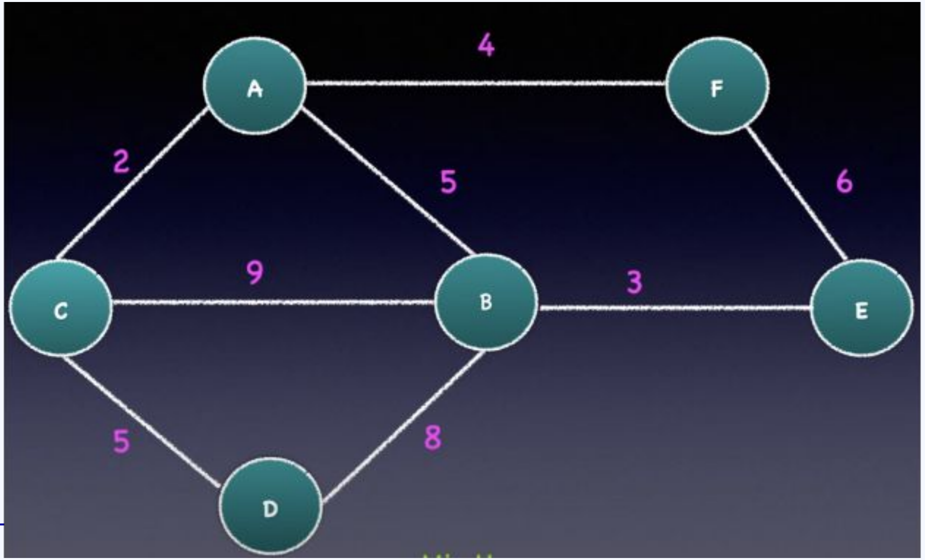
MinHeap Implementation

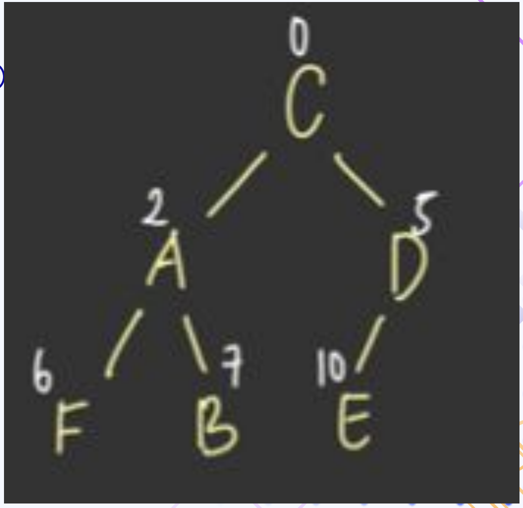
Insert: Add a new element to the heap, maintaining the min-heap property.

Extract Min: Remove the minimum element (the root) and adjust the heap to maintain the min-heap property.

Heapify: A utility function to ensure the min-heap property is maintained.

Complete Binary Tree -> Root Note Smallest Element





Our code for min heap for priority Q

```python
# Function to generate a sparse or dense directed graph as adjacency list
def generate_directed_graph_list(vertices, edges):
    graph = {i: [] for i in range(vertices)}
    edge_count = 0

    while edge_count < edges:
        u = random.randint(0, vertices - 1)
        v = random.randint(0, vertices - 1)
        if u != v and all(v != dest for dest, _ in graph[u]):
            weight = random.randint(1, 10)
            graph[u].append((v, weight))
            edge_count += 1

    return graph
```

```python
# Dijkstra's algorithm using adjacency list for directed graph
def dijkstra_list_directed(graph, src):
    vertices = len(graph)
    dist = [sys.maxsize] * vertices
    dist[src] = 0
    priority_queue = [(0, src)]

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

        if current_dist > dist[u]:
            continue

        for v, weight in graph[u]:
            distance = current_dist + weight
            if distance < dist[v]:
                dist[v] = distance
                heapq.heappush(priority_queue, (distance, v))

    return dist
```

Our code for plotting graphs against runtime

Added best fit line to combat against the fluctuations of python

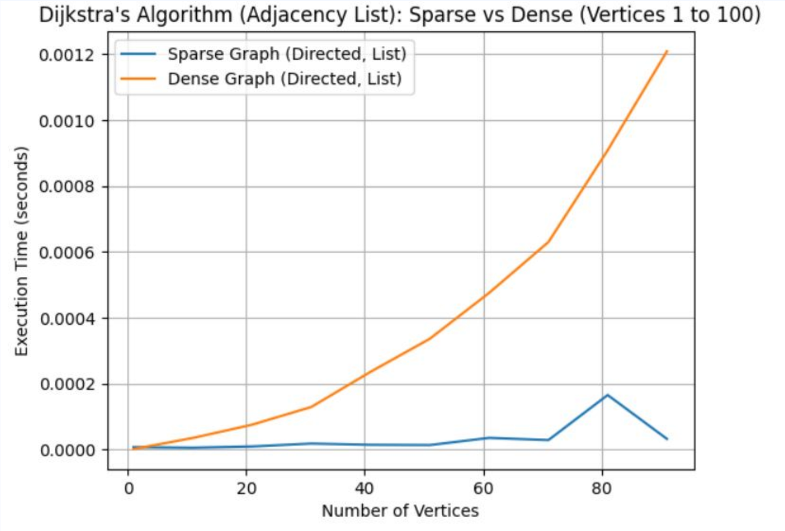
```python
# Measure the execution time for varying vertices for sparse and dense directed graphs using adjacency list
def measure_time_list_directed(vertex_range):
    time_results_sparse = []
    time_results_dense = []

    for v in vertex_range:
        sparse_edges = v - 1  # Sparse graph with v-1 edges
        dense_edges = v * (v - 1)  # Dense graph with max edges for directed graph

        sparse_graph = generate_directed_graph_list(v, sparse_edges)
        dense_graph = generate_directed_graph_list(v, dense_edges)

        # Measure time for sparse graph
        start_time = time.time()
        dijkstra_list_directed(sparse_graph, 0)
        time_results_sparse.append(time.time() - start_time)

        # Measure time for dense graph
        start_time = time.time()
        dijkstra_list_directed(dense_graph, 0)
        time_results_dense.append(time.time() - start_time)

    return time_results_sparse, time_results_dense
```

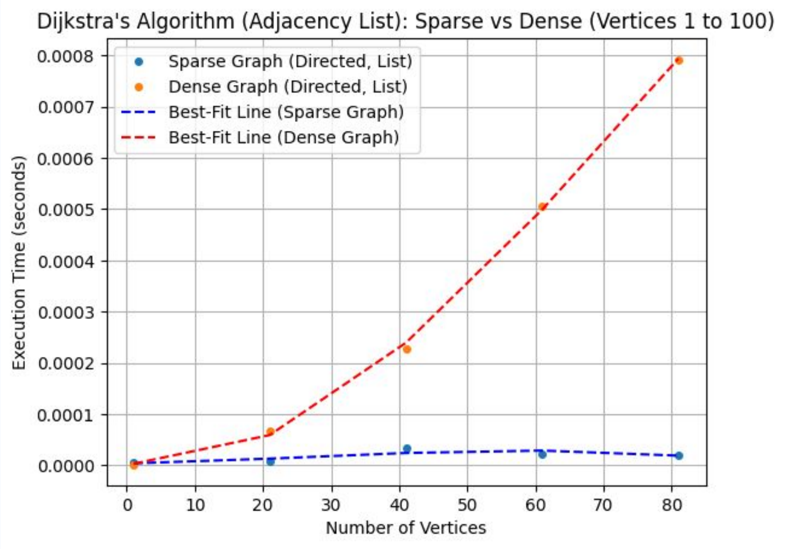
```python
# Function to add polynomial best-fit line
def polynomial_fit(x, y, degree):
    coefficients = np.polyfit(x, y, degree)
    polynomial = np.poly1d(coefficients)
    return polynomial(x)
```

```python
# Plotting results for vertices from 1 to 100 in increments of 100 with best-fit line
def plot_results_part_b_with_fit():
    vertex_range = list(range(1, 101, 10))  # Vertices from 1 to 100 in increments of 10
    time_sparse, time_dense = measure_time_list_directed(vertex_range)

    # Fit best-fit lines (polynomial regression) with degree 3
    sparse_fit = polynomial_fit(vertex_range, time_sparse, 3)
    dense_fit = polynomial_fit(vertex_range, time_dense, 3)

    # Plot original data
    plt.plot(vertex_range, time_sparse, 'o', label="Sparse Graph (Directed, List)", markersize=4)
    plt.plot(vertex_range, time_dense, 'o', label="Dense Graph (Directed, List)", markersize=4)

    # Plot best-fit lines
    plt.plot(vertex_range, sparse_fit, label="Best-Fit Line (Sparse Graph)", linestyle='--', color='blue')
    plt.plot(vertex_range, dense_fit, label="Best-Fit Line (Dense Graph)", linestyle='--', color='red')

    plt.xlabel('Number of Vertices')
    plt.ylabel('Execution Time (seconds)')
    plt.title('Dijkstra\'s Algorithm (Adjacency List): Sparse vs Dense (Vertices 1 to 100)')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_results_part_b_with_fit()
```

For Empirical Data for List with Min Heap

Our graph plotted for varying V in respect to Sparse and Dense Graph





Dense = (n*n-1) - Directed
Sparse = n-1 - Directed

Time complexity with adjacency list and MinHeap queue

1. Create a list for every vertex, so there are V vertices so it takes O(V)
2. After creating the lists, fill up the lists with every edge that exists in the graph -> O(E) for E edges.
3. In total, to create the adjacency list, it takes O(V+E)

Extracting the minimum vertex:

Its an operation that takes log(V) time and you do it for V vertices so in total it takes Vlog(V)

Edge relaxation:

Its an operation that takes log(V) time and you do it for E edges so you have Elog(V)

Theoretical Analysis

Thus we have the total complexity of:

V [Initialization of adjacency list]

+E [Adding edges to the list]

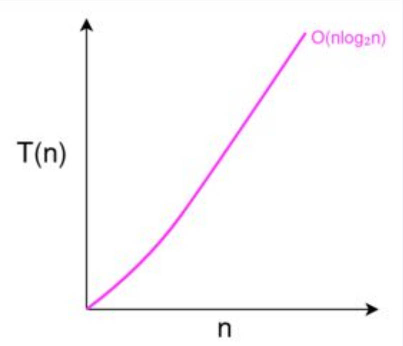
+VlogV [Extraction of minimum element]

+ElogV [Processing Adjacent Vertices - Edge Relaxation

The V+E from the list construction gets dominated because its time complexity becomes insignificant, so we can ignore it when calculating the total time taken:

Simplifying, we get

VlogV + ElogV = (V+E) logV

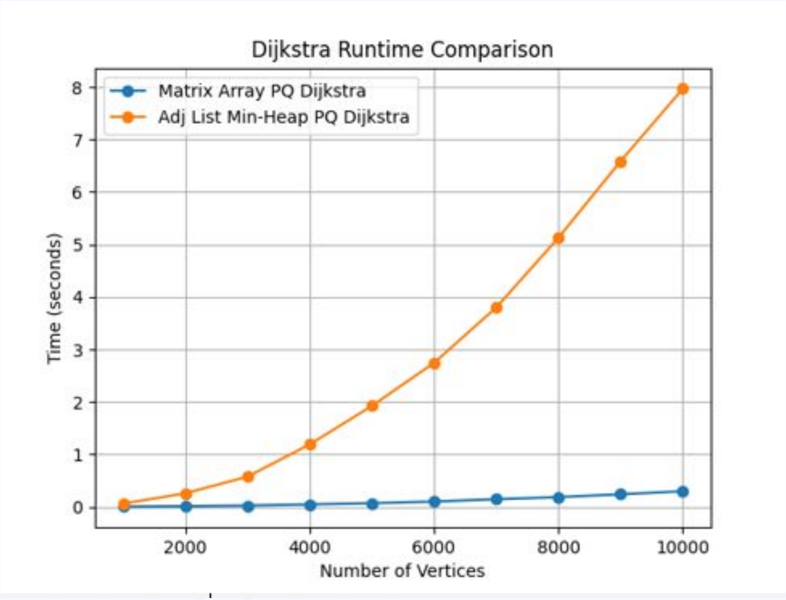


04

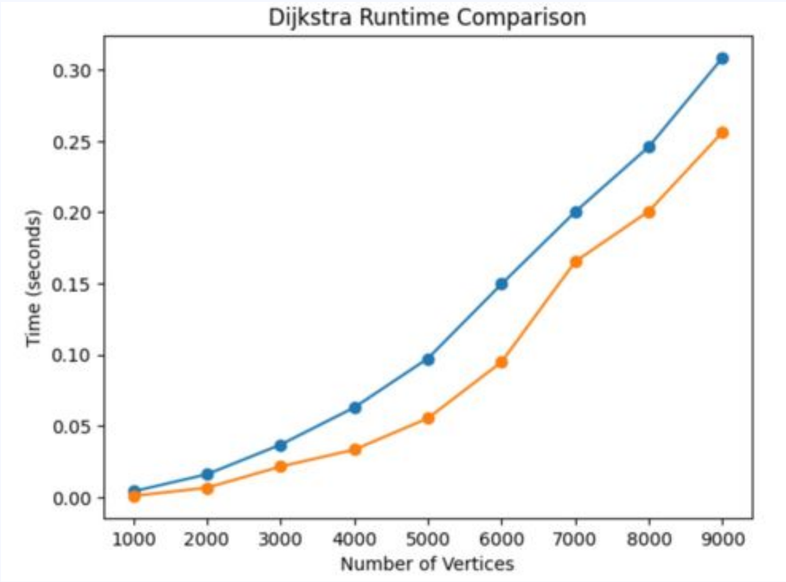
Comparison Analaysis

Comparison Analysis

Dense Graph



Sparse Graph



Adjacency Matrix:

- -Suitable for **dense** graphs (|E|>>|V|), to reduce the effect of the value of |E| on time.
- -Edge operations/lookup take Constant time: O(1)
- -Traversal of neighbors takes O(V) (Linear) time for each Vertice

Adjacency List:

- -Suitable for **sparse** graphs, where |V|>>|E|, saving time and space.
- -Edge operations/lookup take Linear time: O(deg V)
- -Traversal of neighbors takes O(deg V) (Linear) time for each vertice, which is better