SC2001 Example Class 2

Project 2: The Dijkstra’s Algorithm

Daniel Tan
Daniel Tay
Zhi Heng
Wei Jie

Agenda

01 Algorithm

Implementation + Code

02 Adjacency Matrix + Priority Queue (Array)

Time Complexity + Analysis

03 Adjacency List + Priority Queue (Heap)

Time Complexity + Analysis

04 Comparison

Which is better?

01 Algorithm

Dijkstra's Algorithm

Implementation in Java

```java
public static void Dijkstra(int graph[][], int source) { // source starts from vertex 1
        int i;
        PriorityQueueItem nextVertex;

        for (i = 0; i < numVertices; i++) {
                shortestDistances[i] = Integer.MAX_VALUE; // infinity
                predecessors[i] = -1; // -1 as null pointer
                solutionSet[i] = 0; // 1 is vertex is in S
        }

        shortestDistances[source - 1] = 0;
        solutionSet[source - 1] = 1;

        for (i = 0; i < numVertices; i++) { // loop edges from source
                if (graph[source - 1][i] != -1) { // edge found
                        shortestDistances[i] = shortestDistances[source - 1] + graph[source - 1][i]; // updating shortest distance
                        priorityQueue.enqueue(i + 1, shortestDistances[i]); // add vertex to priority queue
                        predecessors[i] = source; // update predecessors
                }
        }
```

Dijkstra's Algorithm

Implementation in Java

```
while (!priorityQueue.isEmpty()) { // while priorityQueue not empty
        nextVertex = priorityQueue.dequeue();

        int currentVertexID = nextVertex.getVertexID();
        solutionSet[currentVertexID - 1] = 1; // add to solution set

        for (i = 0; i < numVertices; i++) { // loop edges from nextVertex
                                                                                                                        {
                shortestDistances[i] = shortestDistances[currentVertexID - 1] + graph[currentVertexID - 1][i]; // updating shortest distance
                priorityQueue.enqueue(i + 1, shortestDistances[i]); // add vertex to priority queue
                predecessors[i] = currentVertexID;
            }
        }
}
```

02

Adjacency Matrix + Priority Queue (Array)

Dijkstra’s Algorithm

Adjacency Matrix + Priority Queue Array Implementation

```
class GraphMatrix { // directed adjacency matrix to store graph
    private int adjMatrix[][];
    private int numVertices;

    public GraphMatrix(int numVertices) {
        this.numVertices = numVertices;
        adjMatrix = new int[numVertices][numVertices];
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                adjMatrix[i][j] = -1; // we take -1 to be infinity
            }
        }
    }

    public void addEdge(int v1, int v2, int weight) {
        adjMatrix[v1][v2] = weight;
    }

    public void removeEdge(int v1, int v2) {
        adjMatrix[v1][v2] = -1;
    }
```

Dijkstra’s Algorithm

Adjacency Matrix + Priority Queue Array Implementation

```java
public class PriorityQueueItem {
    private int vertexID;
    private int weight;

    public PriorityQueueItem() {}

    public int getVertexID() {
        return vertexID;
    }

    public int getWeight() {
        return weight;
    }

    public void setVertexID(int vertexID) {
        this.vertexID = vertexID;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }
}
```

```java
class PriorityQueueArray extends PriorityQueueItem {
    private PriorityQueueItem priorityQueue[];
    private int size;
    private int tail;

    public PriorityQueueArray(int size) {
        this.size = size;
        this.tail = 0;
        priorityQueue = new PriorityQueueItem[size];
        for (int i = 0; i < size; i++) {
            priorityQueue[i] = new PriorityQueueItem();
        }
```

Dijkstra’s Algorithm

Adjacency Matrix + Priority Queue Array Implementation

```
public void enqueue(int vertexID, int weight) { // add entry to tail of array
    if (tail == size - 1)
        System.out.println("Priority Queue is full!");
    else {
        priorityQueue[tail].setVertexID(vertexID);
        priorityQueue[tail].setWeight(weight);
        tail++;
    }
}

public PriorityQueueItem dequeue() { // remove least weight entry from the array
    int minIndex = 0;
    PriorityQueueItem min = priorityQueue[0];
    for(int i=1;i<tail;i++) {
        if(priorityQueue[i].getWeight() < min.getWeight()) {
            min = priorityQueue[i];
            minIndex = i;
        }
    }
    // shift all elements after the minIndex down 1 position
    for(int i=minIndex;i<tail;i++) {
        priorityQueue[i] = priorityQueue[i+1];
    }
    tail--;
    return min;
}
```

Dijkstra - Adjacency Matrix Time Complexity

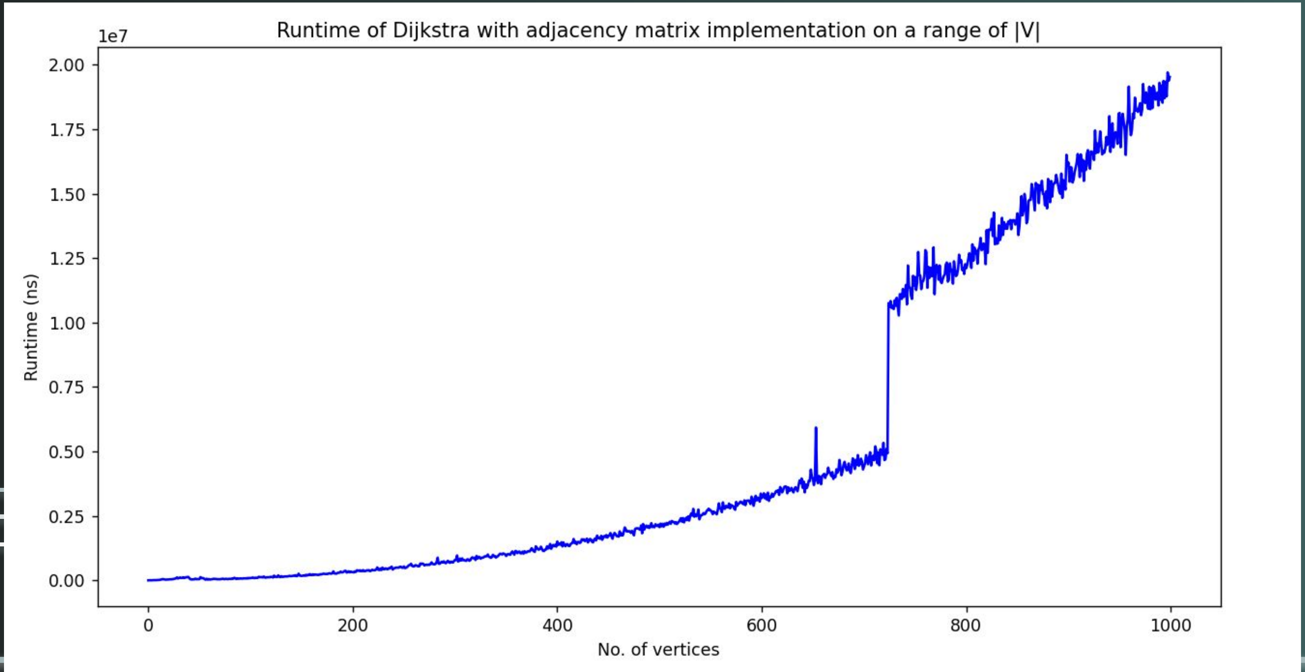
Visit all vertices - $O(|V|)$

Explore at most ($|V|$ - 1) edges for each vertex - $O(|V| - 1)$

Overall Theoretical Worst-case Time Complexity - $O(|V|^2 + |V|)$

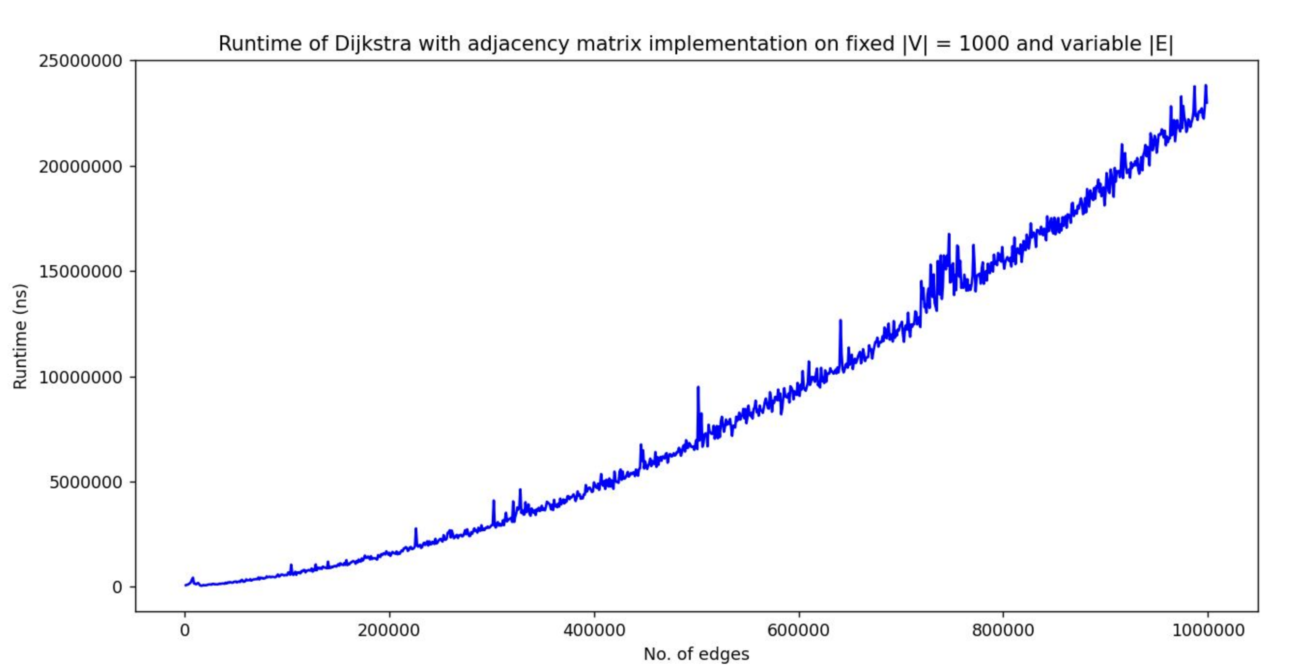
Dijkstra's Algorithm

Adjacency Matrix + Priority Queue Array Implementation
(Runtime graph on complete weighted graphs)



Dijkstra's Algorithm

Adjacency Matrix + Priority Queue Array Implementation
(Runtime graph on non-complete weighted graphs)



03 Adjacency List + Priority Queue (Heap)

001

Dijkstra’s Algorithm

Adjacency List + Min Heap Priority Queue

```
class GraphAdjList { // directed adjacency list to store graph
    private AdjacencyList adjList[];
    private int numVertices;
    private int numEdges;

    public GraphAdjList(int numVertices) {
        this.numVertices = numVertices;
        adjList = new AdjacencyList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjList[i] = new AdjacencyList(); // each index in the adjList array holds an empty linked
        }
    }

    public void addEdge(int v1, int v2, int weight) {
        adjList[v1].addEdge(v2, weight);
        this.numEdges++;
    }

    public void removeEdge(int v1, int v2) {
        adjList[v1].removeEdge(v2);
        this.numEdges--;
    }
```

```
public class ListNode {
    private int vertexID;
    private int weight;
    private ListNode next;

    public ListNode() {}

    public ListNode(int vertexID, int weight) {
        // initialise each weighted edge node
        this.vertexID = vertexID;
        this.weight = weight;
        this.next = null;
    }

    public int getVertexID() {
        return this.vertexID;
    }

    public int getWeight() {
        return this.weight;
    }

    public void setNext(ListNode nextNode) {
        this.next = nextNode;
    }

    public ListNode getNext() {
        return this.next;
    }
```

Dijkstra's Algorithm

Adjacency List + Min Heap Priority Queue

```java
public class AdjacencyList {
    private ListNode head;
    private int size;

    public AdjacencyList() {
        this.head = null;
        this.size = 0;
    }

    public void addEdge(int vertexID, int weight) { // add a new weighted edge to the list
        ListNode curNode;
        ListNode newNode = new ListNode(vertexID, weight);
        // check if the head list node is assigned or not
        if(this.head == null) {
            // add the new list node to the head of the linked list
            this.head = newNode;
        }
        else {
            // add the new list node to the end of the linked list
            curNode = this.head;
            while(curNode.getNext() != null) {
                curNode = curNode.getNext();
            }
            curNode.setNext(newNode);
        }
        this.size++;
    }
```

```java
public void removeEdge(int vertexID) {
    ListNode curNode, preNode, nextNode;
    // check if head of list is assigned
    if(this.head == null) {
        // nothing to remove, return
        return;
    }
    else if(this.head.getVertexID() == vertexID) {
        this.head = null;
        this.size--;
    }
    else {
        curNode = this.head.getNext();
        preNode = this.head;
        // loop until reach the node to be deleted
        while(curNode.getVertexID() != vertexID) {
            preNode = curNode;
            curNode = curNode.getNext();
        }
        // make the previous node point to the node after the node to be deleted
        nextNode = curNode.getNext();
        preNode.setNext(nextNode);
        curNode = null;
    }
}
```

Dijkstra - Adjacency List Time Complexity

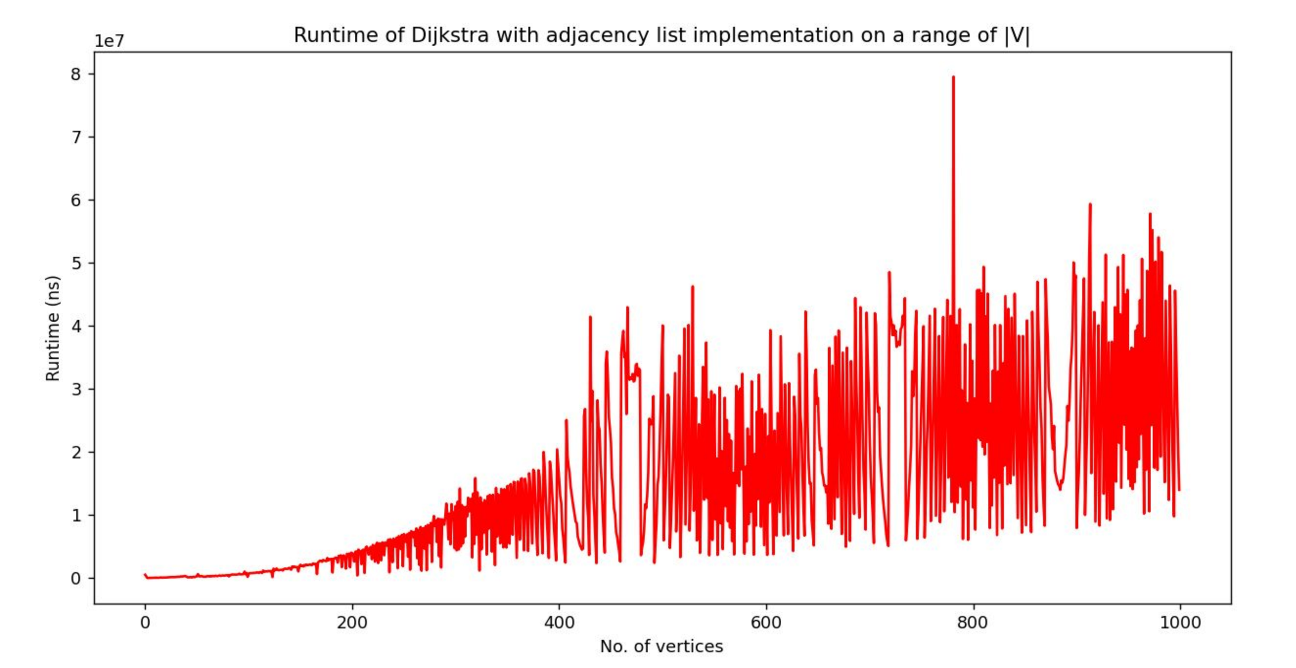
Time complexity for operations in Minimising Heap - $O(\log|V|)$

No. of operation function calls - $O(|V| + |E|)$
No. of $|E|$ for worst case (complete graph) = $|V|^2 - |V|$

Overall Worst-case Theoretical Time Complexity = $O(|V| \log|V| + |E| \log|V|)$
$\approx$ $O(|E|\log|V|)$, $E >> V$

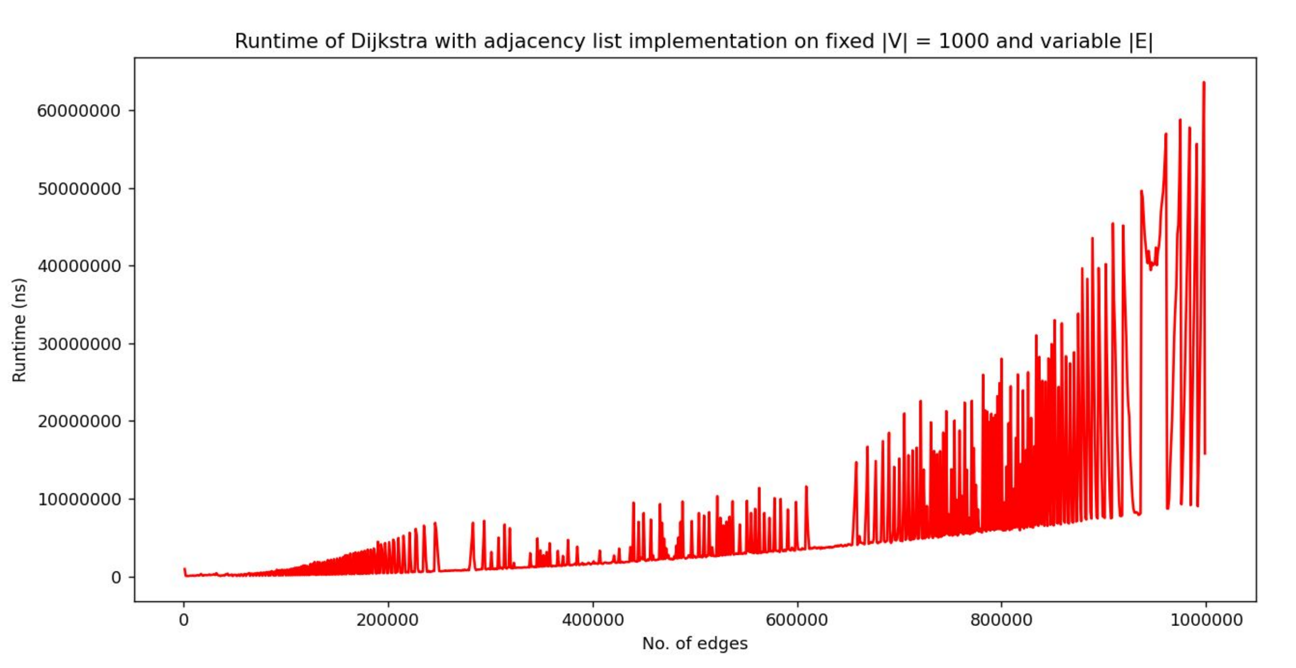
Dijkstra's Algorithm

Adjacency List + Min Heap Priority Queue
(Runtime graph on complete weighted graphs)



Dijkstra's Algorithm

Adjacency List + Min Heap Priority Queue
(Runtime graph on non-complete weighted graphs)



04 Comparison

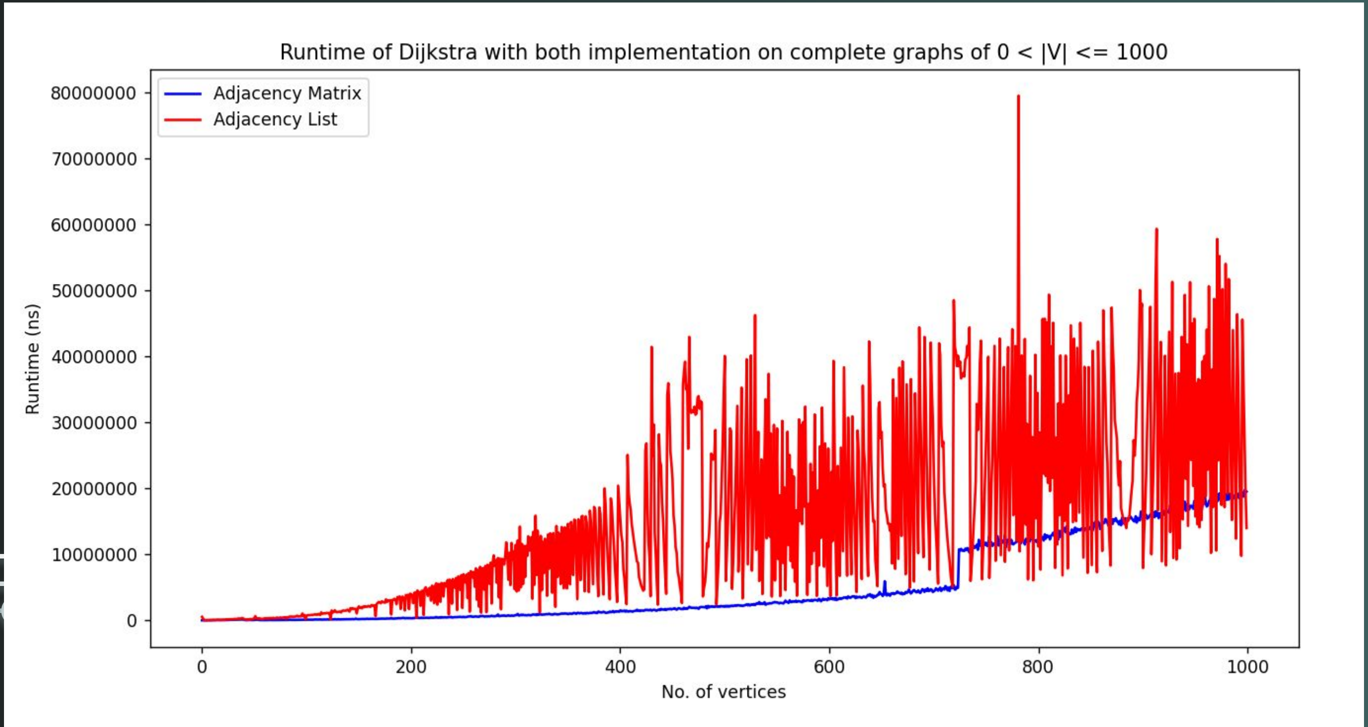
Method of Comparison

Plot complete graphs with |V| ranging from 0 - 1000

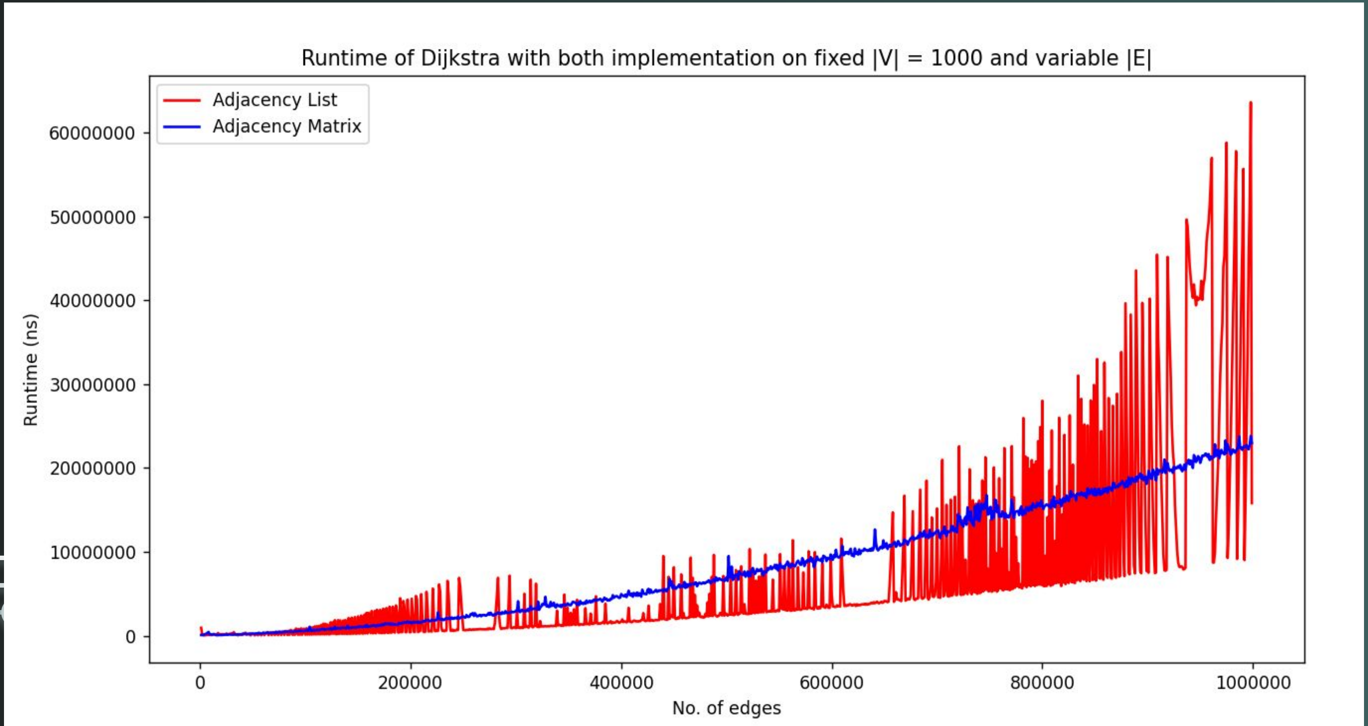
Plot non-complete graphs of fixed |V| = 1000 and variable |E|

Runtime Comparison

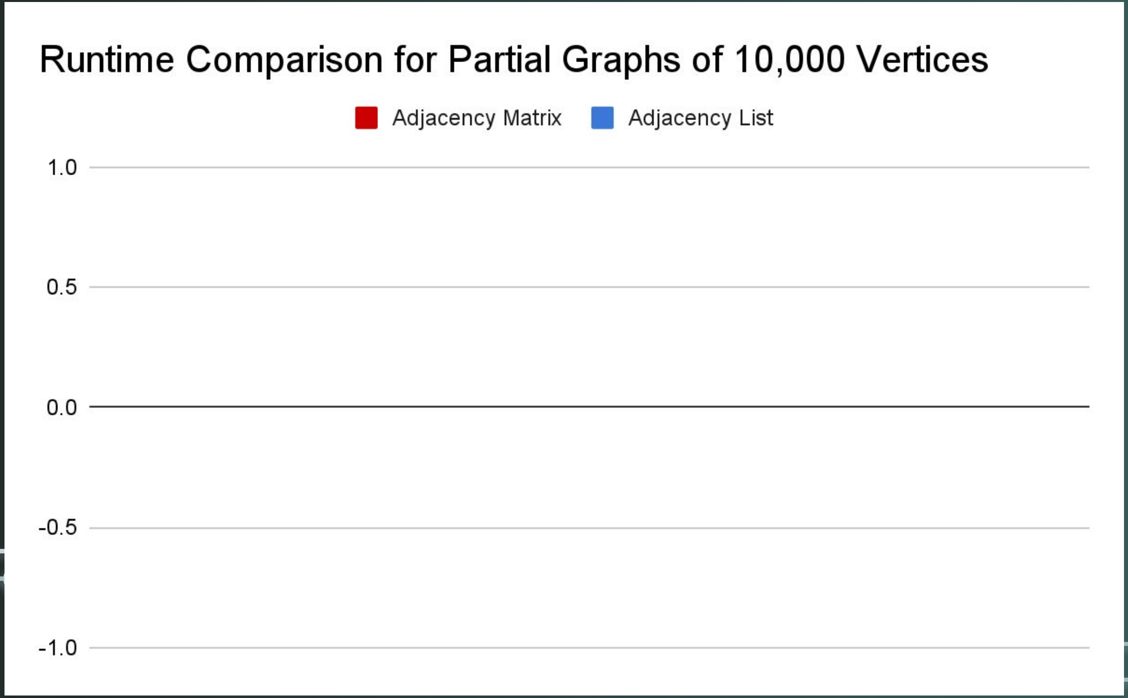
Comparison Between Implementations



Comparison Between Implementations



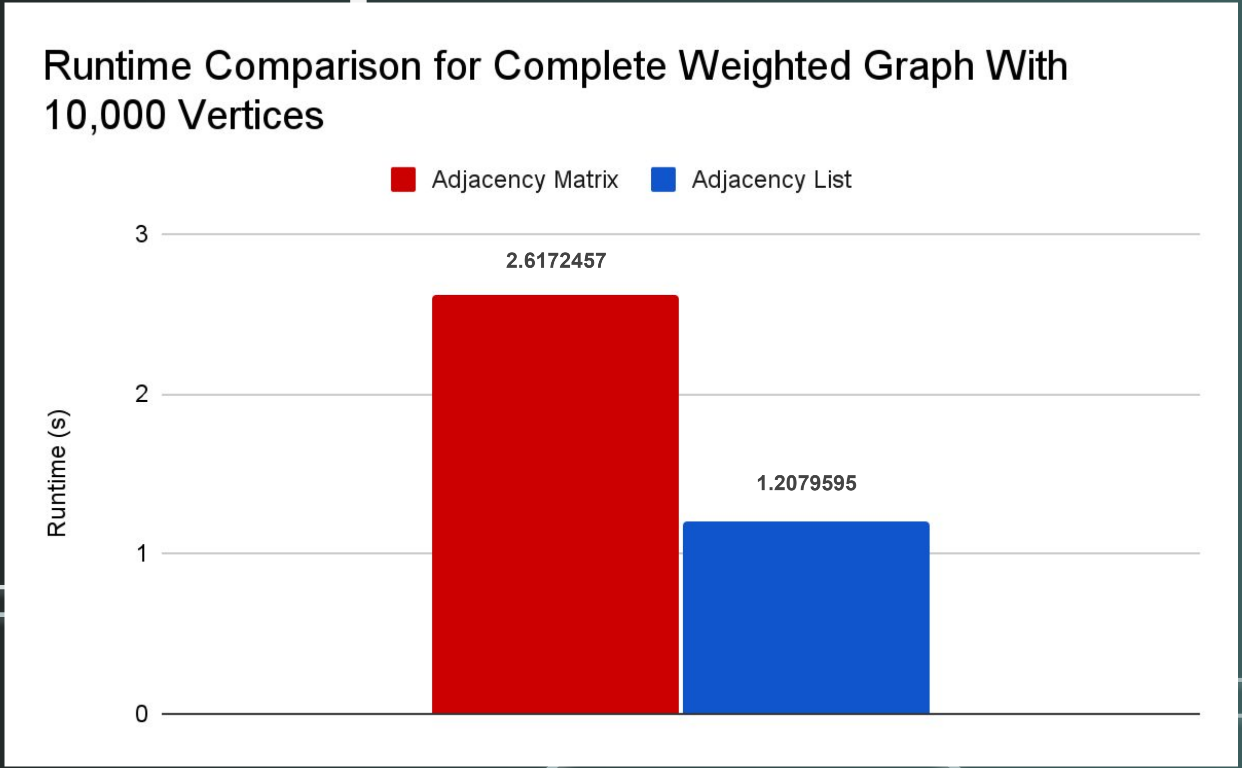
Comparison Between Implementations



Comparison Between Implementations

```
Initialising Graph
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
	at a.GraphMatrix.<init>(GraphMatrix.java:9)
	at a.DijkstraMatrix.initialiseFromFile(DijkstraMatrix.java:118)
	at a.DijkstraMatrix.main(DijkstraMatrix.java:50)
```

Comparison Between Implementations



Conclusion

- For complete graphs, the adjacency matrix implementation of Dijkstra's Algorithm has the better performance.
- For sparse graphs, when |E| is small compared to |V| , the adjacency list implementation has the better performance.
- Overall, the adjacency list implementation has the better performance compared to the adjacency matrix implementation when number of vertices |V| increases

Thank you!