# SC2001 Algorithm Design and Analysis Project 2: The Dijkstra’s Algorithm

Presented by: Zheng Wei, Felicia and Qi Yang

# a)Dijkstra Algorithm: Adjacency Matrix and Array Priority Queue: Code

```java
//Varying V from 10 to 1000,produce graph with fixed edges
public static int[][] generateSparseMatrix(int V) {
    Random rand = new Random();
    int [][]arr = new int[V][V];    //default initialized to 0

    int sparseEdges =(int) (((float)3/(float)8)*(V*V-V)); //fixed E

    //create 3/8(V**2-V) Edges with Random Source and Dest Vertices and Random Weights
    for (int i = 0; i < sparseEdges; i++) {
        int src = rand.nextInt(V);
        int dest = rand.nextInt(V);
        if(src!=dest && arr[src][dest]==0){
            int weight = rand.nextInt(bound:100);     //max weight is assumed 100,distance values
            arr[src][dest] = weight;
        }
        else{
            i--;
        }
    }
    return arr;
}
```

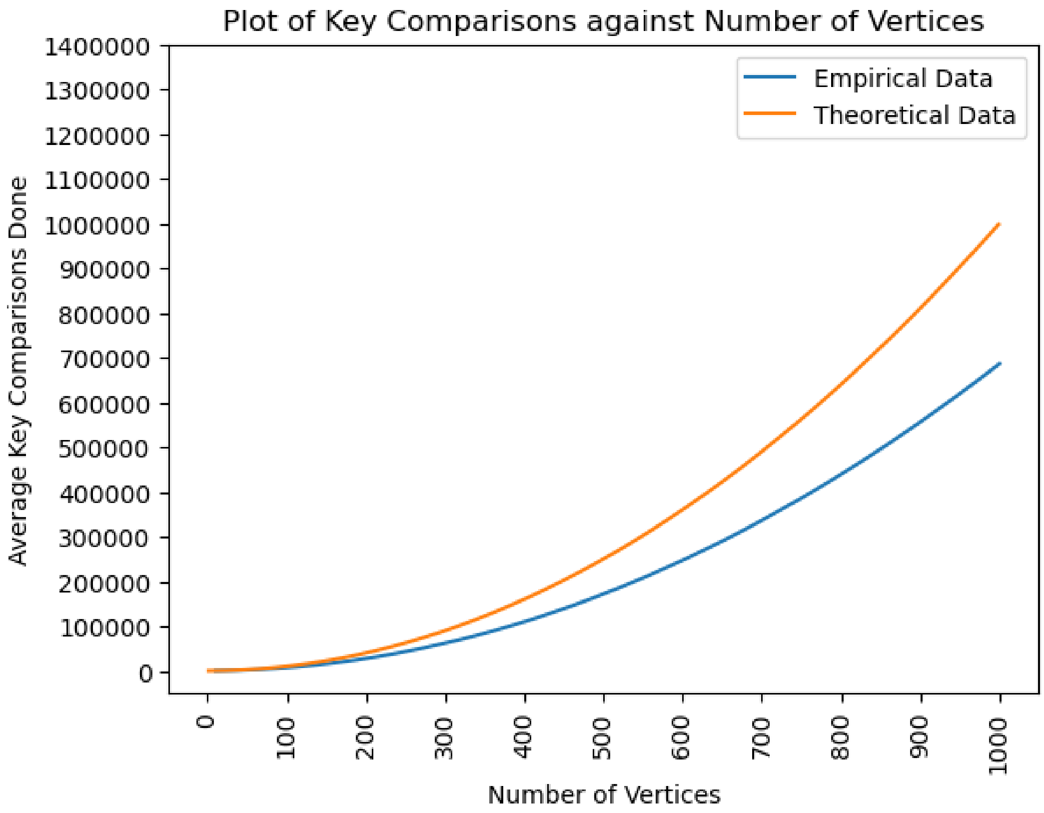
# a)Dijkstra Algorithm: Adjacency Matrix and Array Priority Queue: Code

```
//ARRAY for PRIORITY QUEUE
public static final int NUMBER_OF_ITERATIONS=100;

int[] minDistance(int[]dist, Boolean[]sptSet,int V){      //O(V) time complexity
    int min = Integer.MAX_VALUE,min_index = -1;
    int[] data = new int[2];
    int comparisons = 0;

    for (int v = 0; v < V; v++) {
        if(sptSet[v]==false){
            if(dist[v]<=min){
                min = dist[v];
                min_index = v;
            }
            comparisons++;
        }
    }
    data[0] = min_index;
    data[1] = comparisons;

    return data;    //return data with min_index as first value and comparisons as second value
}
```

# a)Time Complexity:Theoretical

```
int dijkstra(int[][]graph,int src, int V){
    int[]data = new int[2];     //for number of comparisons
    int comparison = 0;

    int[]dist = new int[V];
    Boolean[]sptSet = new Boolean[V];

    for (int i = 0; i < V; i++) {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }

    dist[src] = 0;

    //find shortest distance for remaining, non source V-1 vetices
    for (int count = 0; count < V-1; count++) {     //O(V-1) time = O(V) time
        data = minDistance(dist,sptSet,V);         //O(V) time from minDistance

        int u = data[0];
        sptSet[u] = true;

        comparison+=data[1];
        for (int v = 0; v < V; v++) {      //go through all connections to u,O(V) time
            if(!sptSet[v] && graph[u][v] !=0    //not in shortest path tree,path exist
                && dist[u] != Integer.MAX_VALUE){     //error check
                if(dist[u] + graph[u][v] < dist[v]){    //new shortest path
                    dist[v] = dist[u] + graph[u][v];
                }
                comparison++;
            }
        }

    }//total O(V^2) time

    //printSolution(dist,V);
    return comparison;
}
```

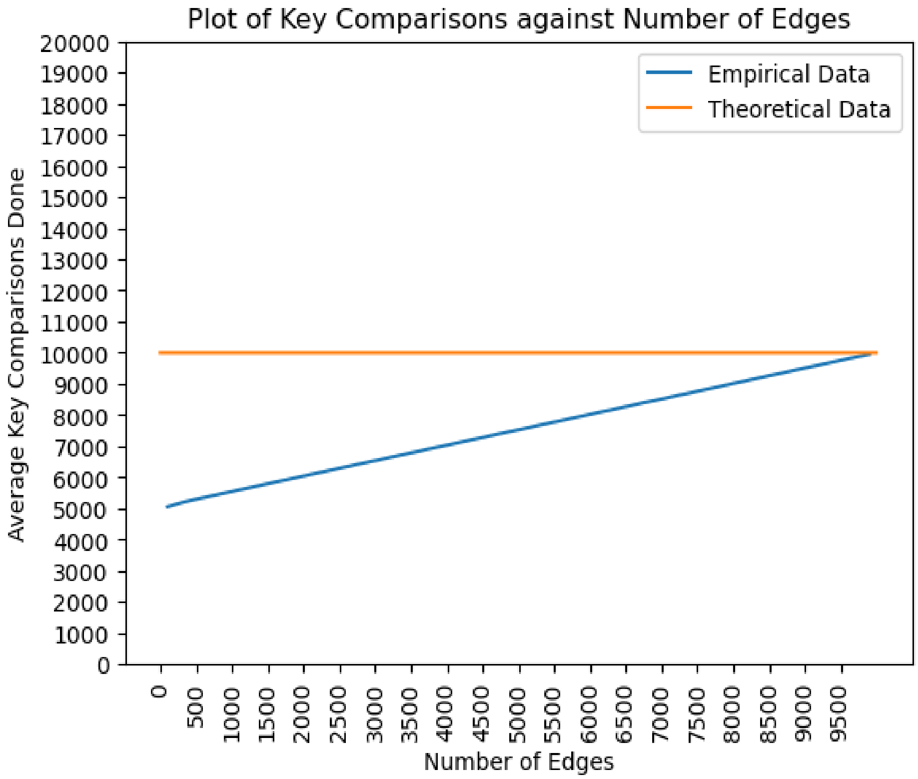
# a)Time Complexity: Empirical VS Theoretical: Varying Vertices




Vertices(V) varied from 10 to 1000

Edges fixed at ⅜(V^2-V) for graphs to be equally dense at every vertice size of 0.75 denseness.

# a)Time Complexity: Empirical VS Theoretical: Varying Edges




Vertices(V) fixed at 100

Edges(E) varied from 100 to maxEdges of 9900 (100*99 OR V*(V-1))

# b) Dijkstra Algorithm: Adjacency Lists and Heap Priority Queue

- deleteMin(List<Node> minHeap)

```
Node deleteMin(List<Node> minHeap) {

    //get rightmost leaf to replace with root
    int last = minHeap.size()-1;
    Node minNode = minHeap.get(index:0);
    minHeap.set(index:0, minHeap.get(last));
    minHeap.remove(last);

    fixHeap(minHeap);

    return minNode;
}
```

# b) Dijkstra Algorithm: Adjacency Lists and Heap Priority Queue

- fixHeap(List<Node> minHeap)

```java
void fixHeap(List<Node> minHeap) {

    int index = 0;
    int size = minHeap.size();

    while(true) {

        int leftChild = 2*index+1;
        int rightChild = 2*index+2;
        int smallest = index;

        if (leftChild < size && minHeap.get(leftChild).weight < minHeap.get(smallest).weight) {
            keyComparisons++;
            smallest = leftChild;
        }

        if (rightChild < size && minHeap.get(rightChild).weight < minHeap.get(smallest).weight) {
            keyComparisons++;
            smallest = rightChild;
        }

        if (smallest != index) {
            swap(minHeap, index, smallest);
            index = smallest;
        } else {
            break;
        }
    }
}
```

## b) Dijkstra Algorithm: Adjacency Lists and Heap Priority Queue

- fixHeap(List<Node> minHeap)

```java
void fixHeap(List<Node> minHeap) {

    int index = 0;
    int size = minHeap.size();

    while(true) {

        int leftChild = 2*index+1;
        int rightChild = 2*index+2;
        int smallest = index;


        if (leftChild < size && minHeap.get(leftChild).weight < minHeap.get(smallest).weight) {
            keyComparisons++;
            smallest = leftChild;
        }

        if (rightChild < size && minHeap.get(rightChild).weight < minHeap.get(smallest).weight) {
            keyComparisons++;
            smallest = rightChild;
        }

        if (smallest != index) {
            swap(minHeap, index, smallest);
            index = smallest;
        } else {
            break;
        }
    }
}
```

# b) Dijkstra Algorithm: Adjacency Lists and Heap Priority Queue

- void swap(List<Node> minHeap, int i, int j)

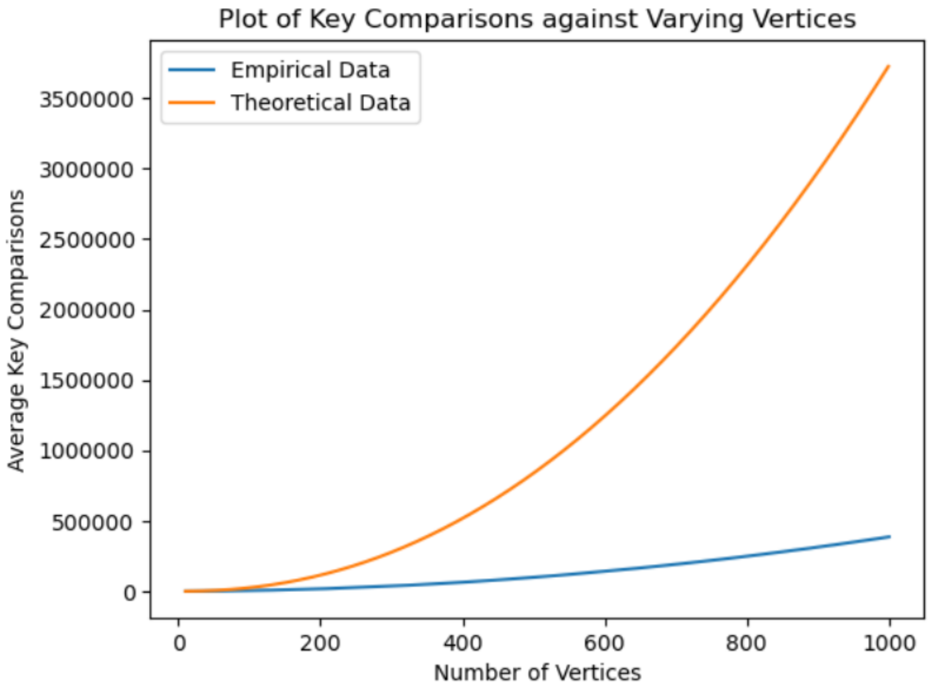
```java
void swap(List<Node> minHeap, int i, int j) {

    Node temp = minHeap.get(i);
    minHeap.set(i, minHeap.get(j));
    minHeap.set(j, temp);
}
```

# b) Dijkstra Algorithm: Adjacency Lists and Heap Priority Queue

```java
int dijkstra(List<List<Node>> adjList, int src, int V) {

        int [] dist = new int [V];
        for(int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
        }
        dist[src] = 0;

        List<Node> minHeap = new ArrayList<Node>();
        minHeap.add(index:0,new Node(src, w:0));

        int[] S=new int[V]
        for(int i=0;i<V;i++)
        {
            S[i]=0;
        }
```

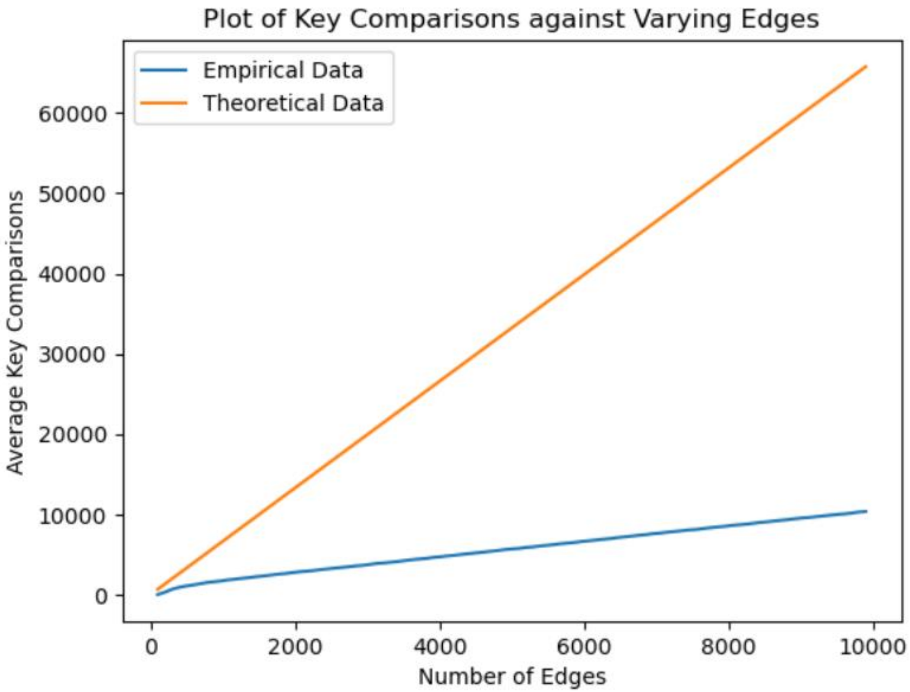
```java
while (!minHeap.isEmpty()) { //Iterate V times

    Node u = deleteMin(minHeap); //O(log V)
    int uVertex = u.vertex;

    if(S[uVertex]==1)continue;
    else S[uVertex]=1;

    for (Node neighbour : adjList.get(uVertex)) { //Iterate nE times (Number of edges per vertex)

        int v = neighbour.vertex;
        int UV = neighbour.weight;

        if (dist[uVertex] != Integer.MAX_VALUE && dist[uVertex] + UV < dist[v] && S[v]!=1) {
            dist[v] = dist[uVertex] + UV;
            minHeap.add(new Node(v, dist[v]));
            fixHeap(minHeap); //O(logv)
        }

        adjListMinHeap.keyComparisons++;
    }
}
return adjListMinHeap.keyComparisons;

//total time complexity: O(V * logV + V*nE*logV)) = O(VlogV + ElogV)= O(ElogV)
```

## b) Theoretical Time Complexity




- Each data point is the average of 100 iterations
- Fixed sparsity of 75%
- Vary vertices from 10 - 1000

# b) Theoretical Time Complexity




- Each data point is the average of 100 iterations
- Fixed 100 vertices
- Varying edges from 100 to 9900 [ V*(V-1) ]

# c)Comparison of Implementations

Part (b)
Adj List + Minimizing Heap

Time complexity: ElogV

As E $\longrightarrow$ V(V-1), E $\longrightarrow$ V^2.

As a graph becomes more dense

Number of Edges Increase

Time complexity approaches $V^2 \log V$

Part (a)
Adj Matrix + Array

Time complexity: V^2

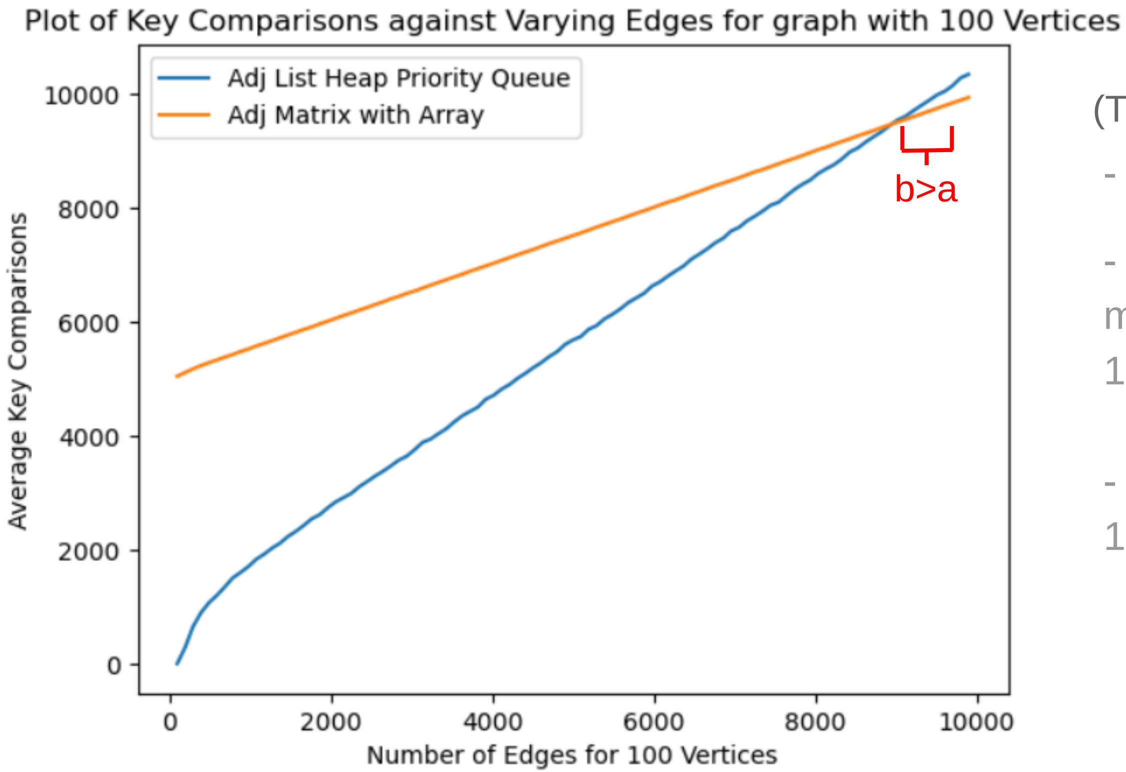
As a graph becomes denser, will partB become slower than partA?

# Test the performance of each implementation against varying **Graph Sparsity**

Measures of Performance:

1. Number of Key Comparisons
2. Time taken

# Increase graph density, observe Key Comparisons



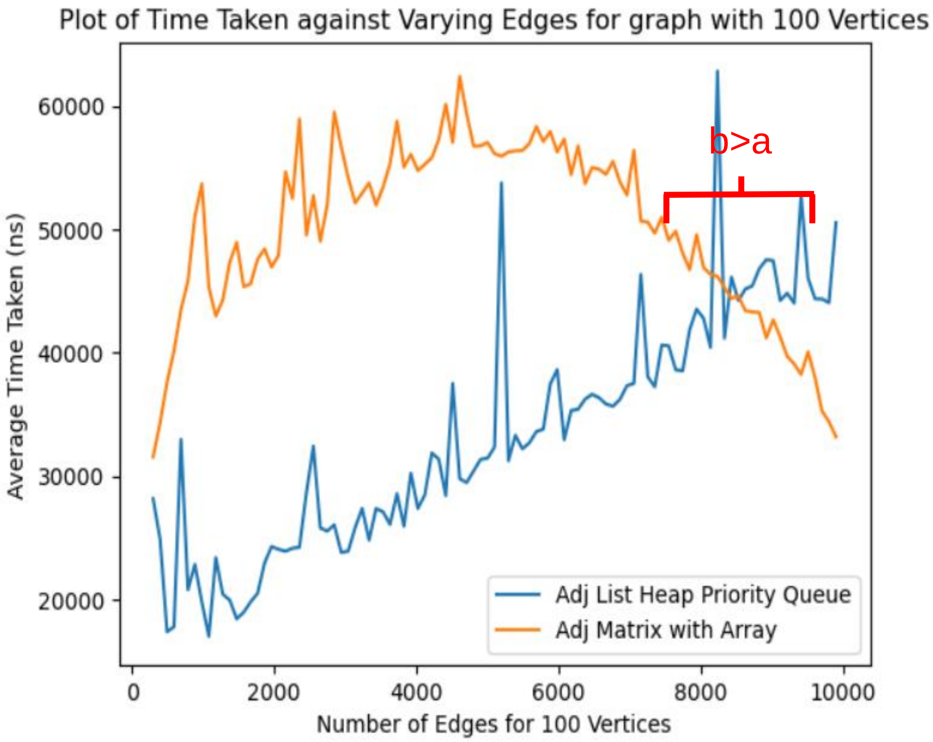

(Test Conditions Same as partA and partB)

- Vertices(V) fixed at 100

- Edges(E) varied from 100 to maximum edges of 9900 (V*(V-1))

- Each data point an average of 100 tests

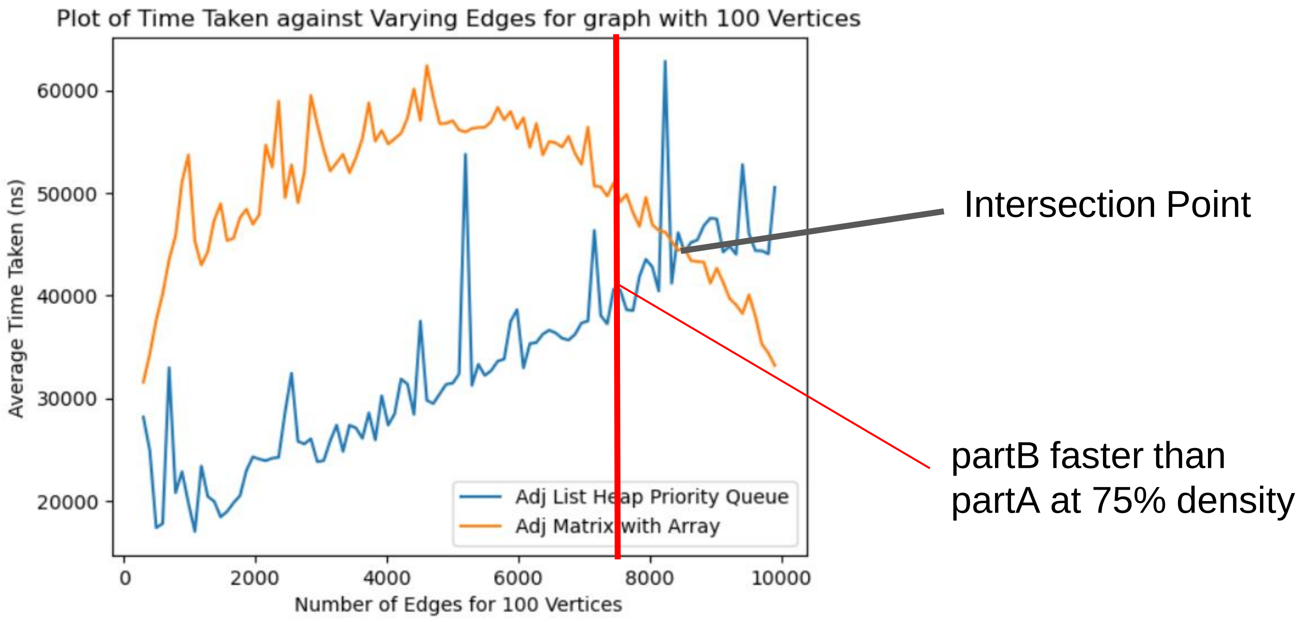
# Increase graph density, observe Time Taken




Test conditions same as previous slide

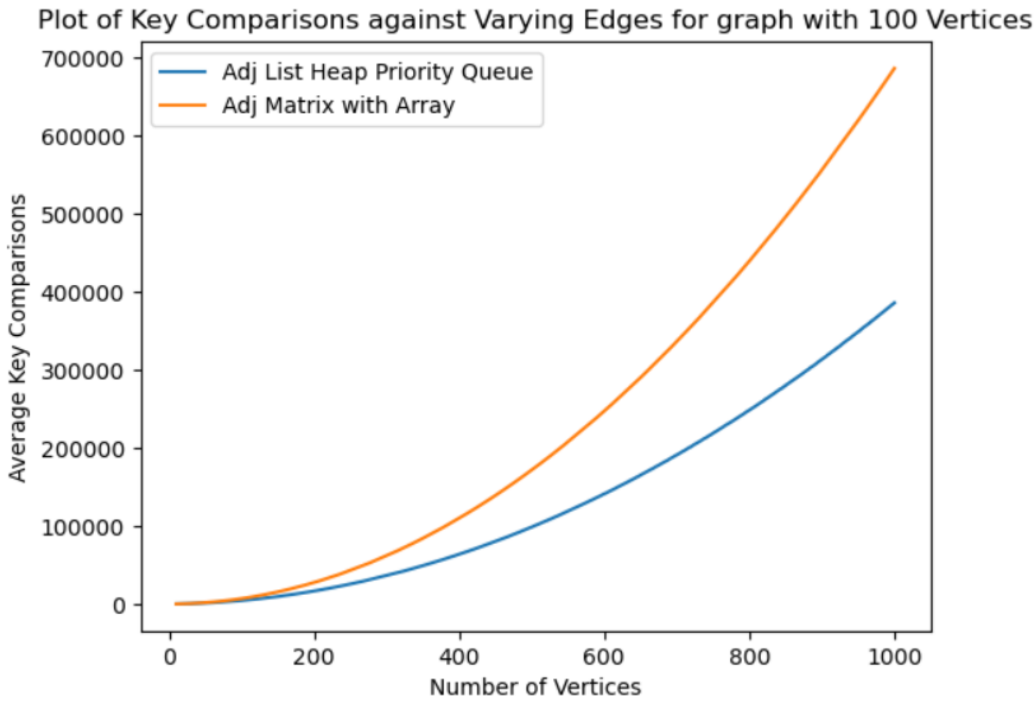
## Findings:

Given a graph, the better implementation is **Indeed** determined by the graph density.




Question to investigate:
Does this mean as long as density is 75%, the optimal algorithm is partB, even as the graph vertices V is varied?

# Increase Vertices, Fix density=0.75, observe Key Comparisons




(Test Conditions Same as partA and partB)

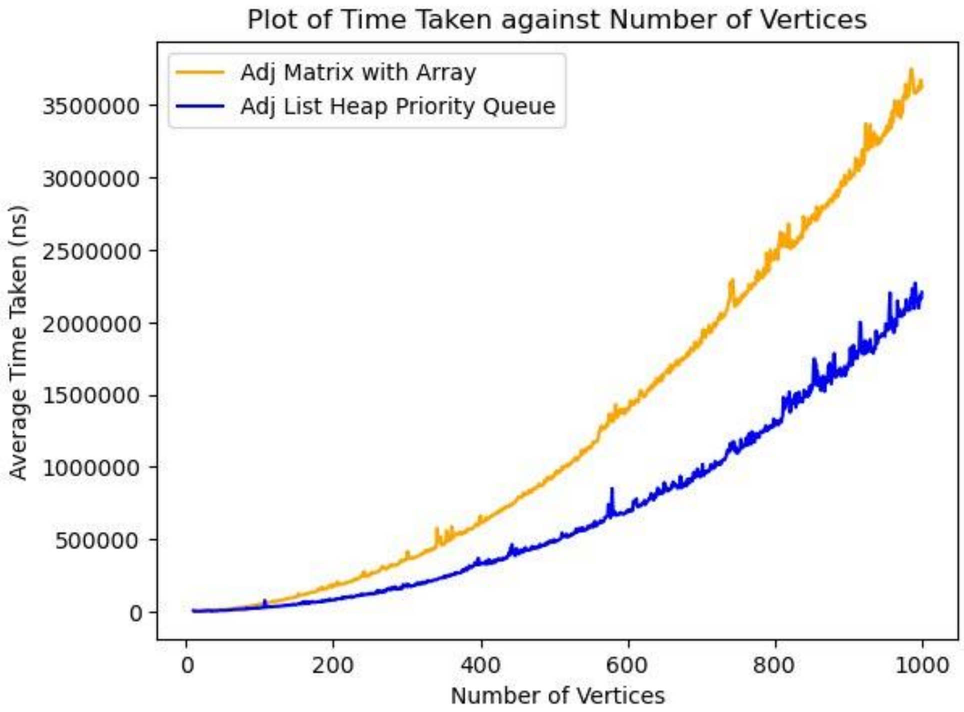
- Vertices(V) varied from 10 to 1000

- Edges fixed at ⅜(V^2-V) for graphs to be equally dense at every vertice size of 0.75 denseness.

- Each data point an average of 100 tests

**Findings:**
For V>10, partB's implementation has lower key comparisons for density=0.75 regardless of graph size

# Increase Vertices, Fix density=0.75, observe Time Taken




Test conditions same as previous slide

**Conclusion:**

- For V>10, Number of Vertices in a graph should not affect which implementation is better. (Provided density is constant)
    - There is an increased disparity in time taken between the optimal and suboptimal implementation, as V increases.

- The key factor that determines which implementation is better, is the density of a graph (the number of Edges it has)
    - At lower density, partB’s implementation (Adj List + Minimizing Heap) is better
    - At higher density, partA’s implementation (Adj Matrix + Array) is better

Thank you! :D