CCP6214 – Algorithm Analysis & Design Report

TT6L: Group – 4

2410

Student ID	Student Name	Task Description	Percentage %
1221301874	Sadman Zulfiquer	Question 1	25
1211101398	Poh Ern Qi	Question 3	25
1211102289	Tan Teng Hui	Question 2	25
1211104274	Tan Xin Thong	Question 4	25

Table of Contents

Question 1: Dataset 1

Algorithm Time Complexity: *O (n)*

Algorithm results: As the per the requirements the dataset can only contain digits of group leader Id (0123478), in our experiment after generating lots of datasets it leads to lots of repeated values as the dataset size grows, therefore ***increase in dataset size is directly proportional to increase in repeated values in the dataset (more repeated values for less digit values e.g. 2-digit values).*** The algorithm can create a random number containing digits from 2-5 e.g. 10s, 100s, 1000s, 10,000s.

picture of dataset size 100 & 1 million, repeated values count for a random value

Only 4 time's value 21 is repeated, dataset size 100

Value 21 repeated 14943 times, dataset size 1 million

Average time taken to generate each dataset:
Time taken to Generate Set 1: 12 ms

Time taken to Generate Set 2: 14 ms

Time taken to Generate Set 3: 21 ms

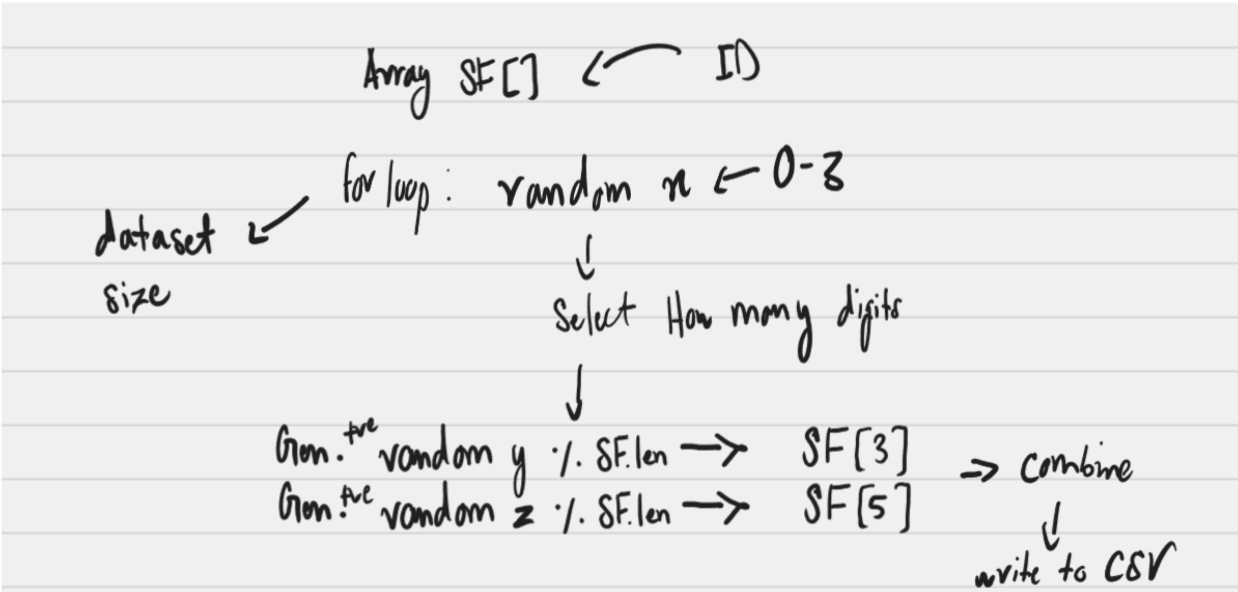
Time taken to Generate Set 4: 83 ms

Time taken to Generate Set 5: 170 ms

Time taken to Generate Set 6: 274 ms

Suitability: The dataset can be useful to experiment with sorting algorithms especially between stable & unstable sorting algorithm.

Improvements: There are few possible improvements can be made to the algorithm such as generating random values with more digits, decreasing the probability of 2-3 digits numbers been generated depending on the dataset size to keep have less repeated values and shuffling the array after each random value is generated, that is used to store group leaders Id.



Simple illustration on how this algorithm Works

Question 1: Dataset 2

Algorithm Time Complexity: *O* (n^2)

Algorithm results: As per the requirements we connected each to at least 3 other stars with majority of stars connected to 4 other stars to meet the requirement of total 54 edges. The random values x, y, z and profit are 3-digit random values while, the weight values are limited to 2-digit random values, after experimenting the sum of weight values for 20 stars is approximately 1000-1200 kg on average. (which is enough to test Question 4 Algorithm)

E.g. for a few stars

Star name = 1, x = 137, y = 16, z = 160, weight = 61, profit = 601, connectedStarsName = [2, 20, 3, 4,]

Star name = 2, x = 331, y = 776, z = 63, weight = 31, profit = 7, connectedStarsName = [3, 1, 4, 5,]

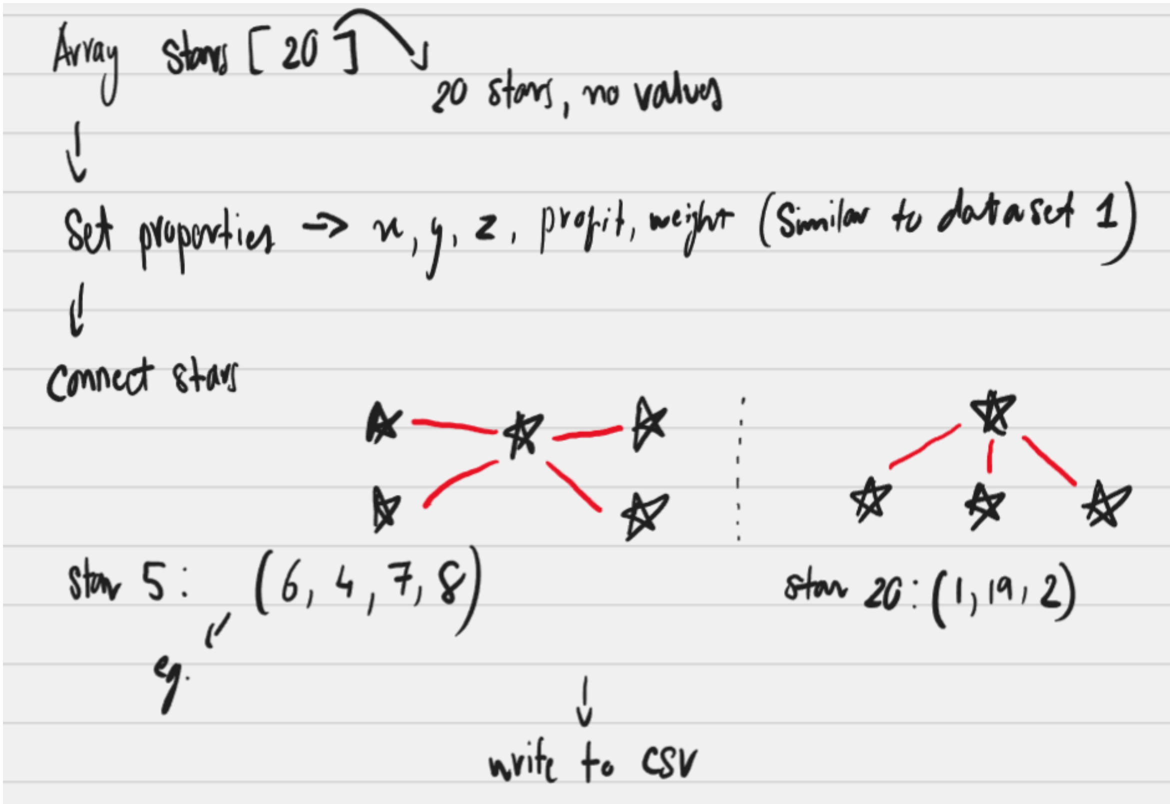
Star name = 3, x = 9, y = 673, z = 997, weight = 96, profit = 13, connectedStarsName = [4, 2, 5, 6,]

Star name = 4, x = 79, y = 31, z = 100, weight = 71, profit = 67, connectedStarsName = [5, 3, 6, 7,]

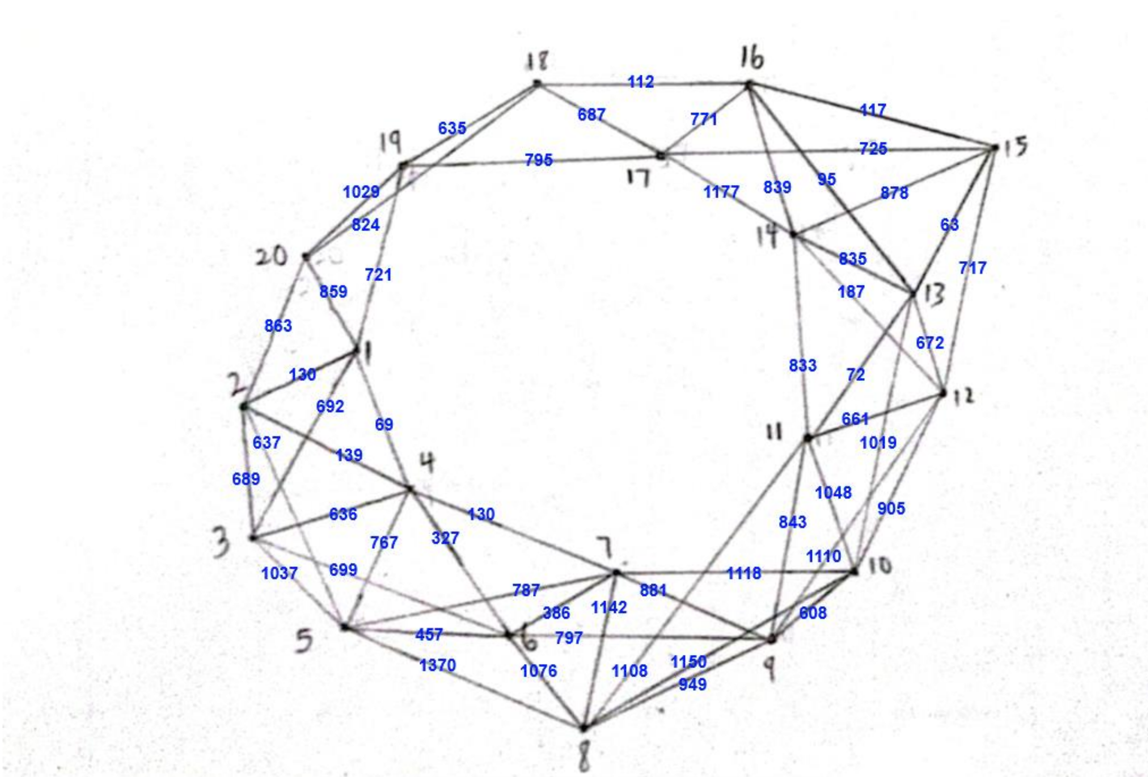
Average time taken to generate dataset: 85 ms

Suitability: The dataset can be useful to experiment with graph algorithms or as well as some greedy algorithm such as Dijkstra’s, Prim-Jarnik’s & Kruskal's.

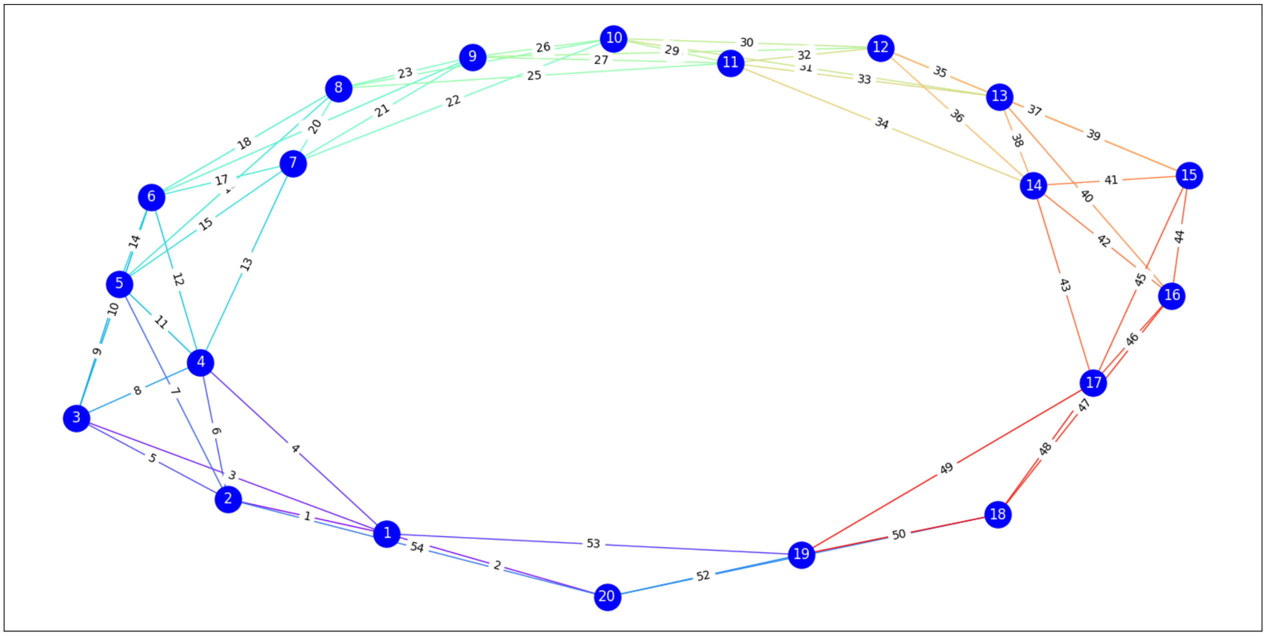
Improvements: Few improvements that made is better CSV file output & possibly adding additional code for better star visualization.



Simple illustration on how the algorithm works



e.g. Stars with distance, from a generated dataset



Stars with edge count

Question 2: Heap Sort

1. Initialize the Array in ‘processFile’ method

Reading the file

```java
ublic static void processFile(String inputFile, String outputFile) {
    try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {
        String line;
        List<Integer> list = new ArrayList<>();// create a list to store the integers
        while ((line = reader.readLine()) != null) { // read all lines
            String[] stringValues = line.split(regex:","); // split the line by comma

            // convert the string values to integers and add them to the list
            for (String stringValue : stringValues) {
                list.add(Integer.parseInt(stringValue));
            }
        }
```

- The code uses a `BufferedReader` to read the input file line by line.
- Each line is split by commas to extract the individual integer values.
- These values are converted from `String` to `int` and added to a `List<Integer>`.

Sorting the integers.

```java
// convert the list to an array
int[] A = list.stream().mapToInt(i -> i).toArray();

heapSort(A);
```

- The list of integers is converted to an array of integers, `A`.
- The `heapSort` method is called to sort the array.

Store Output

```java
// write the sorted array to a CSV file
try (PrintWriter writer = new PrintWriter(new FileWriter(outputFile))) {
    for (int value : A) {
        writer.println(value);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

- The sorted array is written to the output file using a `PrintWriter`.
- Each integer in the array is written on a new line in the output file.

2. Heapify Process

```java
private static void heapify(int[] A, int arraySize, int j) {
    int max;
    int left = 2 * j + 1;
    int right = 2 * j + 2;

    if (left < arraySize && A[left] > A[j])
        max = left;
    else
        max = j;

    if (right < arraySize && A[right] > A[max])
        max = right;

    if (max != j) {
        int temp = A[j];
        A[j] = A[max];
        A[max] = temp;
        heapify(A, arraySize, max);
    }
}
```

The **`heapify`** method ensures that the subtree rooted at index `j` is a max-heap. The method works as follows:

- Determine the left and right child indices of `j`.
- Identify the largest value among the current node (`j`) and its children.
- If the largest value is not the current node, swap the current node with the largest value and recursively heapify the affected subtree.

3. Heapsort Process

```java
private static void heapSort(int[] A) {

    int arraySize = A.length;

    // Record the start time before building the heap
    long startTime = System.currentTimeMillis();

    // Build heap (rearrange array)
    for (int j = arraySize / 2 - 1; j >= 0; j--)
        heapify(A, arraySize, j);
    // Record the end time after building the heap
    long endTime = System.currentTimeMillis();

    // Calculate the time taken to build the heap
    System.out.println("Time taken to insert all data into the priority queue: " + (endTime - startTime) + " ms");

    // Record the start time before dequeuing the data
    long startTime1 = System.currentTimeMillis();

    // One by one extract an element from heap
    for (int i = arraySize - 1; i >= 0; i--) {
        // Move current root to end
        int temp = A[0];
        A[0] = A[i];
        A[i] = temp;

        // call max heapify on the reduced heap
        heapify(A, i, j:0);
    }
    // Record the end time after dequeuing the data
    long endTime1 = System.currentTimeMillis();

    // Calculate the time taken to dequeue the data
    System.out.println("Time taken to dequeue the data: " + (endTime1 - startTime1) + " ms");
}
```

Building the Max-Heap:

- Start from the last non-leaf node and call heapify to ensure all subtrees are max-heaps.
- This transforms the entire array into a max-heap.
- The time taken to build the heap is recorded.

Extracting Elements from the Heap:

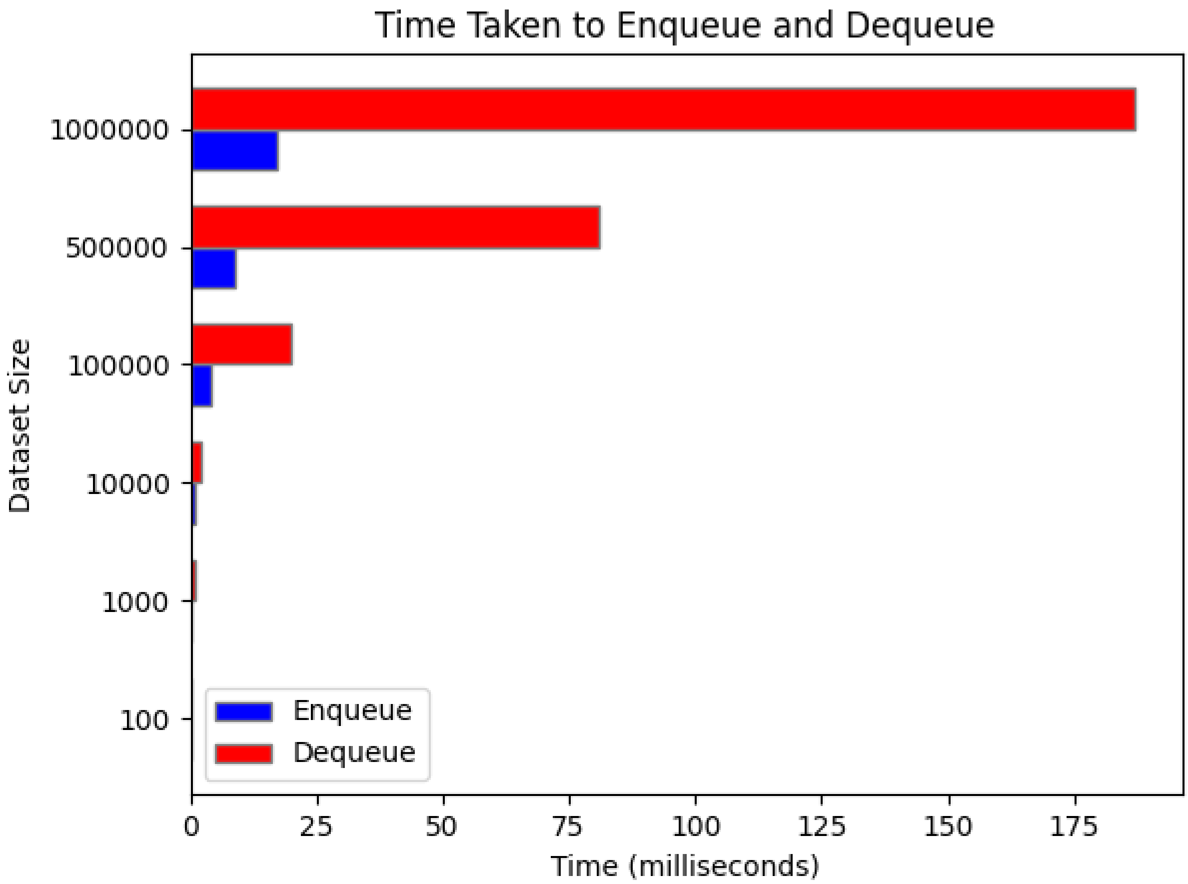
- Swap the root (maximum element) with the last element of the heap.

- Reduce the heap size by one and call heapify on the root to restore the max-heap property.
- Repeat this process until the heap size is reduced to one.
- The time taken to extract the elements is recorded.

Output Result

```
Time taken to insert all data into the priority queue: 0 ms
Time taken to dequeue the data: 0 ms

Time taken to insert all data into the priority queue: 0 ms
Time taken to dequeue the data: 1 ms

Time taken to insert all data into the priority queue: 1 ms
Time taken to dequeue the data: 2 ms

Time taken to insert all data into the priority queue: 4 ms
Time taken to dequeue the data: 20 ms

Time taken to insert all data into the priority queue: 9 ms
Time taken to dequeue the data: 81 ms

Time taken to insert all data into the priority queue: 17 ms
Time taken to dequeue the data: 187 ms
```

Graph



1. **Time taken to Enqueue**:
   - The time taken to enqueue elements represents the time complexity of building the max-heap.
   - As the dataset size increases, the time taken to enqueue also increases. This aligns with the O(logn) time complexity for building the heap.
2. **Time taken to Dequeue**:
   - The time taken to dequeue elements represents the time complexity of extracting elements from the heap.
   - As the dataset size increases, the time taken to dequeue also increases, but at a faster rate. This aligns with the O(nlogn) time complexity for extracting elements.

Conclusion:

- Heapsort offers an optimal time complexity of O(nlogn) for sorting.
- It achieves this efficiency by leveraging the properties of binary heaps.
- Despite its relatively high time complexity, heapsort is advantageous due to its in-place sorting nature, making it suitable for scenarios where memory usage is a concern.

Question 2: Selection Sort

1. Initialize the Array in ‘processFile’ method

```
try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {

    String line;
    List<Integer> list = new ArrayList<>();// create a list to store the integers
    while ((line = reader.readLine()) != null) { // read all lines
        String[] stringValues = line.split(regex:","); // split the line by comma

        // convert the string values to integers and add them to the list
        for (String stringValue : stringValues) {
            list.add(Integer.parseInt(stringValue));
        }
    }
```

- Uses `BufferedReader` to read the input file line by line.
- Each line is split by commas to extract individual integer values.
- These values are converted from `String` to `int` and added to a `List<Integer>`.

Sorting the integers

```
// Record the start time before inserting data into the priority queue
long startTime = System.currentTimeMillis();
selectionSort(numbers);
;
// Record the end time after inserting data into the priority queue
long endTime = System.currentTimeMillis();

// Calculate the time taken to insert all data into the priority queue
System.out.println("Time taken to sort all data: " + (endTime - startTime) + " ms");
```

To measure the performance of the sorting process, the code records the start time just before the sorting begins and the end time immediately after the sorting completes. The difference between these two timestamps gives the total time taken for the sorting operation, which is then printed to the console in milliseconds.

Store Output

```java
// write the sorted array to a CSV file
try (PrintWriter writer = new PrintWriter(new FileWriter(outputFile))) {
    for (int value : numbers) {
        writer.println(value);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

- The sorted array is written to the output file using a `PrintWriter`.
- Each integer in the array is written on a new line in the output file.

2.Selection Sort Process

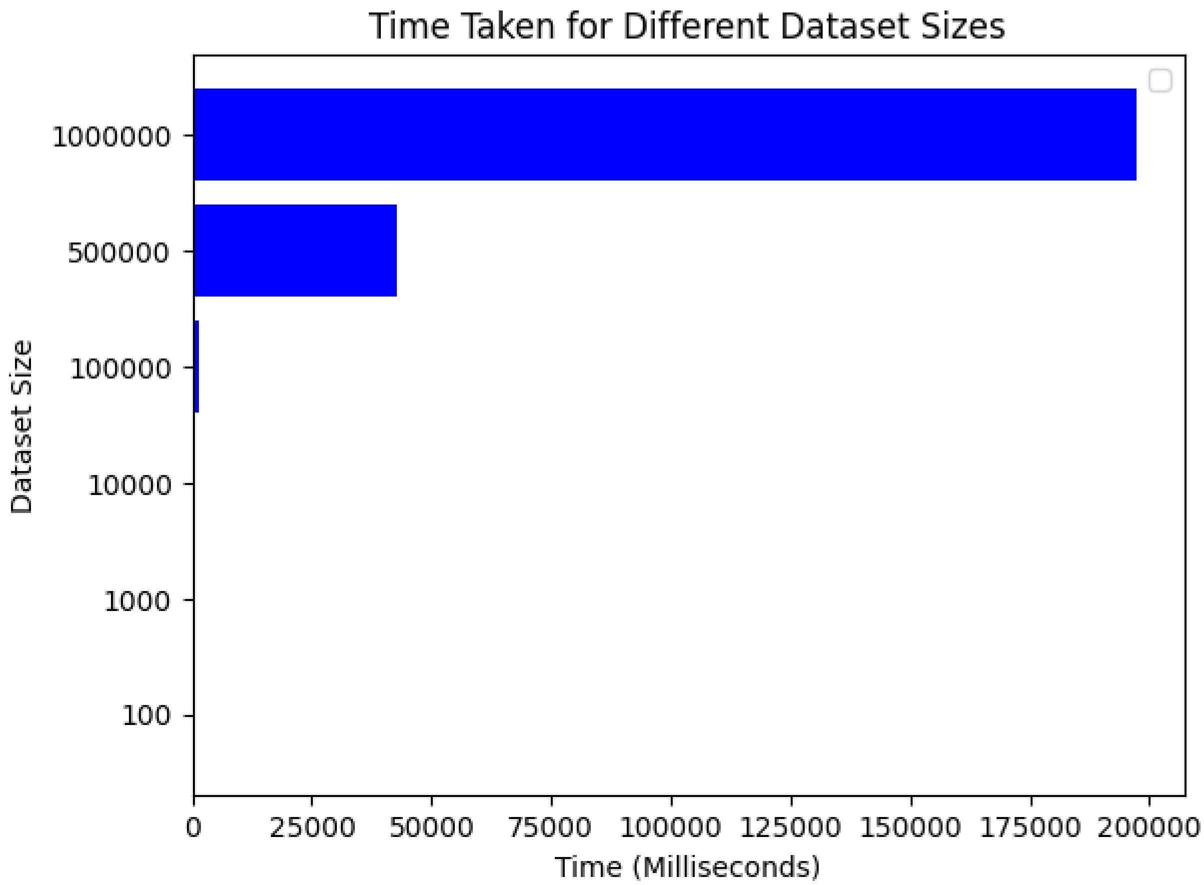
```java
private static void selectionSort(int[] numbers) {
    int n = numbers.length;
    for (int i = 0; i < n - 1; i++) {
        int min = numbers[i];
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (numbers[j] < min) {
                min = numbers[j];
                minIndex = j;
            }
        }
        swap(numbers, i, minIndex);
    }
}

private static void swap(int[] numbers, int a, int b) {
    int temp = numbers[a];
    numbers[a] = numbers[b];
    numbers[b] = temp;
}
```

The sorting is accomplished using the selection sort algorithm, which sorts the array in ascending order. The algorithm operates by iterating through the array, assuming the current element is the smallest in the remaining unsorted portion. It then searches for the smallest element in this unsorted segment, and if a smaller element is found, it swaps the current element with this smaller element. This process repeats for each element in the array until the entire array is sorted.

Output

```
Time taken to sort all data: 0 ms

Time taken to sort all data: 7 ms

Time taken to sort all data: 22 ms

Time taken to sort all data: 1450 ms

Time taken to sort all data: 42634 ms

Time taken to sort all data: 197356 ms
```

Graph



Selection sort is a straightforward and easy-to-implement sorting algorithm with predictable performance characteristics. However, due to its O(n^2) time complexity, it is inefficient for large datasets compared to more advanced algorithms like quicksort, mergesort, or heapsort, which have better average and worst-case time complexities O(nlogn). The space complexity of O(1) is a favorable aspect of selection sort, but it does not compensate for the poor time performance on larger datasets. Thus, while selection sort can be useful for educational purposes and small datasets, it is generally not recommended for sorting large collections of data.

Question 3: Shortest Path (Dijkstra's)

Main :

```java
    public static void main(String[] args) {
        Dijkstra graph = new Dijkstra(vertices:20); // graph with 20 vertices

        processFile(graph);

        try {
        PrintWriter writer = new PrintWriter(new File(pathname:"Q3/D_results.txt"));

        dijkstraStart(graph, sourceVertex:1, writer);

        writer.close();

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    }
```

The **main()** method in class Dijkstra performs functions by initializing the graph with 20 vertices, process the file and run the Dijkstra algorithm, **dijkstraStart().**

Dijkstra class:

```java
class Dijkstra {
    int vertices;
    Map<Integer, Star> stars;
    List<List<Vertex>> adjacencyList;
    Star star;

    public Dijkstra(int vertices) {
        this.vertices = vertices;
        stars = new HashMap<>();
        adjacencyList = new ArrayList<>();
        for (int i = 0; i <= vertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }
```

List<List<Vertex>> adjacencyList is an adjacency list, a common way to represent a graph. It is implemented as a list of lists, where each list represents a vertex and the inner **List<Vertex>** contains list of vertices connected to it.

Function to read the file:

```java
    //function to read file
    public static void processFile(Dijkstra graph) {
        HashMap<Integer, double[]> starPositions = new HashMap<>();

        //read dataset2
        try (BufferedReader br = new BufferedReader(new FileReader(fileName:"dataSet2.csv"))) {
            String line = br.readLine(); // Read and ignore the header
            while ((line = br.readLine()) != null) {
                String[] values = line.split(regex:",");
                int starName = Integer.parseInt(values[0]); // Parse the 1st column, star name
                double x = Double.parseDouble(values[1]); // Parse the 2nd column, x-coordinate of star
                double y = Double.parseDouble(values[2]); // Parse y-coordinate
                double z = Double.parseDouble(values[3]); // Parse z-coordinate
                starPositions.put(starName, new double[] { x, y, z }); // Store coordinates in the map
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Read connected_stars.csv and compute distances
        try (BufferedReader br = new BufferedReader(new FileReader(fileName:"connected_stars.csv"))) {
            String line = br.readLine(); // Skip header
            while ((line = br.readLine()) != null) {
                String[] values = line.split(regex:",");
                int star1 = Integer.parseInt(values[0]); // Parse the star name
                int star2 = Integer.parseInt(values[1]); // Parse the star that is connected to
                double[] pos1 = starPositions.get(star1);  // Retrieve the first star's coordinates
                double[] pos2 = starPositions.get(star2); // Retrieve the connected star's coordinates
                int distance = Star.calculateDistance(pos1, pos2); // Calculate distance between stars
                graph.addEdge(star1, star2, distance);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

The **processFile()** function mainly performs 2 functions, reading dataset2.csv and connected_stars.csv. When reading dataset2.csv, the buffered reader parses the star name, the x, y and z coordinates of the stars and stores the coordinates in a map. In this case, a HashMap is used to store the star name and its respective coordinates. When reading connected_stars.csv, it reads the star name, the star that it is connected to and proceeds to calculate the distance between the stars. The **addEdge()** method adds the edges to a adjacency list.

Function to add edge:

```java
    public void addEdge(int source, int destination, int distance) {
        // Add edge from source to destination if it doesn't exist
        if (!edgeExists(source, destination, distance)) {
            adjacencyList.get(source).add(new Vertex(destination, distance));
        }
        // Since this is an undirected graph, add edge from destination to source if it doesn't exist
        if (!edgeExists(destination, source, distance)) {
            adjacencyList.get(destination).add(new Vertex(source, distance));
        }
    }
```

Function to check whether edge exists:

```
    public boolean edgeExists(int from, int to, double distance) {
        // Iterate over all edges emanating from the vertex 'from'
        for (Vertex vertex : adjacencyList.get(from)) {
            // Check if there is an edge going to vertex 'to' with the exact distance specified
            if (vertex.getVertex() == to && Double.compare(vertex.distance, distance) == 0) {
                return true;
            }
        }
        return false;
    }
```

The **edgeExists()** method mainly checks if the edges already exists in both the source and destination in an adjacency list (since it is undirected graph).

Dijkstra algorithm:

```
    public static void dijkstraStart(Dijkstra graph, int sourceVertex, PrintWriter writer) {
        // Keeps track of whether a vertex has been fully processed.
        boolean[] visited = new boolean[graph.vertices + 1];

        // Stores the shortest distance from the source vertex to every other vertex.
        int[] distances = new int[graph.vertices + 1];

        // Holds the previous vertex from the source for each vertex.
        int[] predecessors = new int[graph.vertices + 1];

        // Fill predecessors array with -1 indicating no predecessors initially.
        Arrays.fill(predecessors, -1);

        // Initialize distances array with Integer.MAX_VALUE to represent infinity.
        Arrays.fill(distances, Integer.MAX_VALUE);

        // The distance of the source vertex is set to 0.
        distances[sourceVertex] = 0;

        // Priority queue to select the next vertex with the shortest distance.
        PriorityQueue<Vertex> pq = new PriorityQueue<>(new Comparator<Vertex>() {
            @Override
            public int compare(Vertex e1, Vertex e2) {
                return Integer.compare(e1.distance, e2.distance);
            }
        });

        //add source vertex to priority queue
        pq.offer(new Vertex(sourceVertex, distance:0));

```

dijkstraStart() is where the algorithm begins. An array of visited, distances and predecessors are used to keep track of the vertex that has already been visited, storing the shortest distance from source vertex to every other vertex and holding the predecessors from the sources for each vertex. Arrays of predecessors are populated with -1 indicating no predecessors initially. The distance array is prefilled with a very large value to ensure that any actual distance calculated during the execution of the algorithm (which is usually much smaller) will replace this initial placeholder. A priority queue is used to make sure it is always sorted by distance in ascending order.

```
        while (!pq.isEmpty()) {
            // Extract the vertex with the minimum distance.
            Vertex current = pq.poll();
            int currentVertex = current.vertex;

            // Skip this vertex if it has already been processed.
            if (visited[currentVertex]) {
                continue;
            }

            // Mark the current vertex as processed.
            visited[currentVertex] = true;

            // Explore all adjacent vertices of the current vertex.
            List<Vertex> vertices = graph.adjacencyList.get(currentVertex);
            for (Vertex vertex : vertices) {
                // Proceed only if the adjacent vertex has not been visited.
                if (!visited[vertex.vertex]) {
                    // Calculate the distance to the adjacent vertex.
                    int newDist = distances[currentVertex] + vertex.distance;
                    // Update the distance if it is shorter than the previously known distance.
                    if (newDist < distances[vertex.vertex]) {
                        distances[vertex.vertex] = newDist;
                        predecessors[vertex.vertex] = currentVertex;
                        // Add the adjacent vertex to the queue with the new distance.
                        pq.offer(new Vertex(vertex.vertex, newDist));
                    }
                }
            }
```

The for loop iterates until the priority queue becomes empty. It firsts extract the vertex with its minimum distance and mark it as visited. Then, it explores all adjacent vertices of the current vertex and by iterating the connected vertices, it calculates the distance of the current vertex to the adjacent vertex and updates the distance if it is shorter than the previously known distance, as well as its predecessors. Then, it adds the vertex to the queue with the new distance, and the process continues again.

```java
        // Print the shortest distances from the source vertex to all other vertices.
        printDistances(distances, sourceVertex, writer);
        // Print the shortest paths from the source vertex to all other vertices.
        printPaths(sourceVertex, predecessors, graph.vertices, writer);
    }

    //function to print distances from star 1 to other stars
    private static void printDistances(int[] distances, int sourceVertex, PrintWriter writer) {
        writer.println("Shortest paths from star " + sourceVertex);
        for (int i = 1; i < distances.length; i++) {
            writer.println("To star " + i + ", " + (distances[i] == Integer.MAX_VALUE ? "No path" : Integer.toString(distances[i])));
        }
        writer.println();
    }

    //function to print the predecessors of the path of star
    public static void printPaths(int sourceVertex, int[] predecessors, int vertices, PrintWriter writer) {
        writer.println("Graph representing shortest Paths from star " + sourceVertex);
        for (int i = 0; i <= vertices; i++) {
            if (i != sourceVertex && predecessors[i] != -1) {
                writer.print("Path to star " + i + ", ");
                printPath(i, predecessors, writer);
                writer.println();
            }
        }
        writer.println();
```

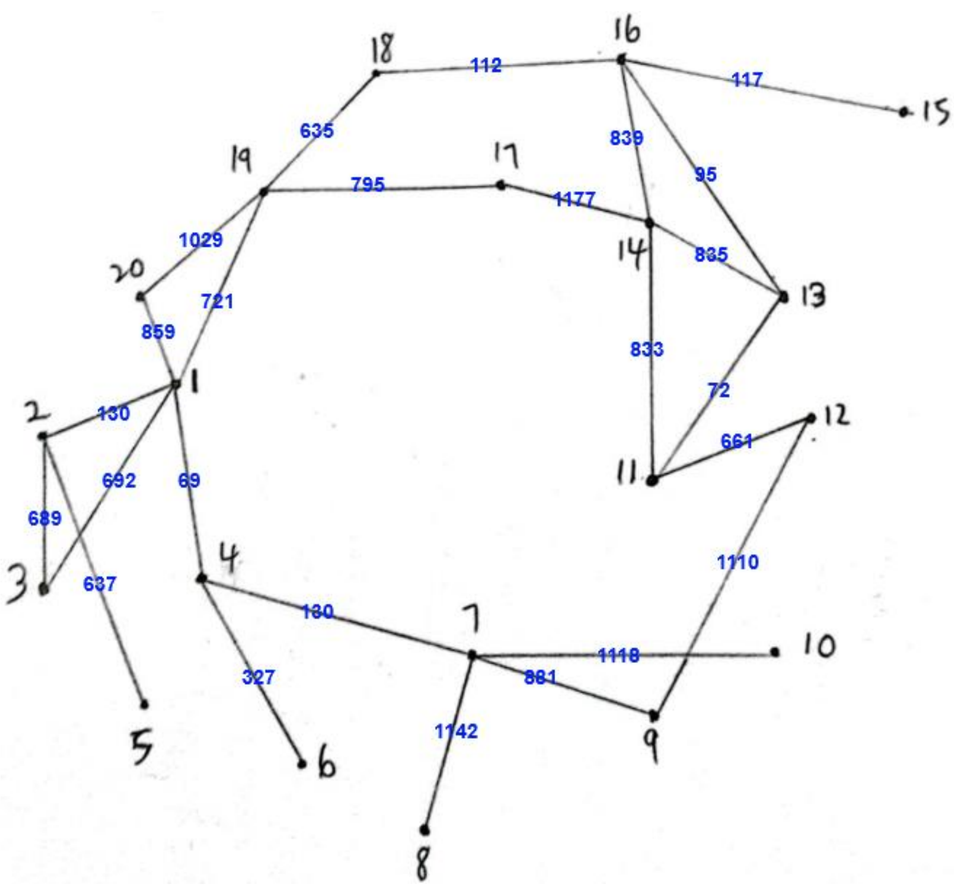
This part of code then prints out all shortest paths, which is stored in **D_results.txt**.

Output in D_results.txt :

```
Shortest paths from star 1
To star 1, 0
To star 2, 130
To star 3, 692
To star 4, 69
To star 5, 767
To star 6, 396
To star 7, 199
To star 8, 1341
To star 9, 1080
To star 10, 1317
To star 11, 1635
To star 12, 2190
To star 13, 1563
To star 14, 2307
To star 15, 1585
To star 16, 1468
To star 17, 1516
To star 18, 1356
To star 19, 721
To star 20, 859
```

```
Graph representing shortest Paths from star 1
Path to star 2, 1 -> 2
Path to star 3, 1 -> 3
Path to star 4, 1 -> 4
Path to star 5, 1 -> 2 -> 5
Path to star 6, 1 -> 4 -> 6
Path to star 7, 1 -> 4 -> 7
Path to star 8, 1 -> 4 -> 7 -> 8
Path to star 9, 1 -> 4 -> 7 -> 9
Path to star 10, 1 -> 4 -> 7 -> 10
Path to star 11, 1 -> 19 -> 18 -> 16 -> 13 -> 11
Path to star 12, 1 -> 4 -> 7 -> 9 -> 12
Path to star 13, 1 -> 19 -> 18 -> 16 -> 13
Path to star 14, 1 -> 19 -> 18 -> 16 -> 14
Path to star 15, 1 -> 19 -> 18 -> 16 -> 15
Path to star 16, 1 -> 19 -> 18 -> 16
Path to star 17, 1 -> 19 -> 17
Path to star 18, 1 -> 19 -> 18
Path to star 19, 1 -> 19
Path to star 20, 1 -> 20
```

Graph representing the path:



Discussion:

For the time complexity, the insertion into the priority queue is O(log V), where n is the number of vertices currently in the priority queue. The **poll()** operation, which retrieves and removes the vertex with the minimum distance from the priority queue, also operates in O(log V). As there are V vertices, the total complexity for all polls is O(V log V). For each vertex, the algorithm potentially relaxes the edges leading out of it, which means in the worst case, each edge is considered once when its originating vertex is processed. Each relaxation involves a comparison and possibly an update of the distance array, followed by an insertion into the priority queue. This contributes an additional O(log V) due to the re-insertion into the priority queue and since each edge can cause such an operation, the total time for all edge relaxations accumulates to O(E log V). Hence, the overall time complexity becomes O(V log V+E log V) =O((V + E) log V).

For the space complexity, The visited array**, boolean[] visited = new boolean[graph.vertices + 1];** is used to keep track of whether each vertex has been fully processed. It is a boolean array with an entry for each vertex in the graph, so it occupies O(V) space, where V is the number of vertices. Same goes to the distance array, predecessors' array, and priority queue. The predecessors array has an entry for each vertex, consuming O(V) and for priority queues, the worst case could contain all vertices at some point during the algorithm's execution, hence it also takes O(V). As we are using an adjacency list for our algorithm, the total space taken by an adjacency list is O(V + E), where E is the number of edges. Hence, by combining the space complexity denotes O(V + E).

Question 3: Minimum Spanning Tree (Kruskal's)

The **main** method in class kruskal's algorithm performs the functions by mainly reading data from 2 files, **dataset2.csv** and **connected_stars.csv**, then outputs the result in **K_results.txt**, which is the same method and concept used in Dijkstra. (more detail will be found in code)

Constructor of edge class:

```java
class EdgeK {
    int src, dest;
    int distance;

    //src is the source vertex from which the edge originates.
    //dest is destination vertex at which the edge terminates.
    //distance represent the weight.
    EdgeK(int src, int dest, int distance) {
        this.src = src;
        this.dest = dest;
        this.distance = distance;
    }

```

This is the constructor of edge class. The src represents the source vertex, dest is destination vertex and distance represents the weight.

```java
ArrayList<EdgeK> edges = new ArrayList<>();
```

```java
edges.add(new EdgeK(star1, star2, distance));
```

When reading the file, edges are added to the arraylist, which will be used in Kruskal's algorithm.

Kruskal Algorithm:

```java
    public void Kruskal(ArrayList<EdgeK> edges, int V, PrintWriter writer) {

        // sorts the edges of the graph based on their weights in ascending order.
        Collections.sort(edges, new Comparator<EdgeK>() {
            @Override
            public int compare(EdgeK edge1, EdgeK edge2) {
                return Integer.compare(edge1.distance, edge2.distance);
            }
        });
        // Parent array to track the root of each vertex.
        parent = new int[V + 1];
        for (int i = 0; i <= V; i++) {
        // Initialize each vertex as its own parent.
            parent[i] = i;
        }

        // Variable to store the weight of the MST.
        int mst_weight = 0;
        // List to store the edges included in the MST.
        List<EdgeK> mst = new ArrayList<>();
```

In Kruskal’s algorithm, the algorithm starts by sorting the edges of the graph based on their distance in ascending order. An array, **parent= new int[V + 1]** stores the parent array to track the root of each vertex (star). V+1 is used in this case since the star number starts from 1 up to 20, an extra space is allocated to make indexing straightforward and to avoid off-by-one errors. Then, the for loop is used to initialize each vertex as its own parent, which is crucial for the union-find operation to be implemented later.

```java
        for (EdgeK edge : edges) {
            int nextSource = find(edge.src); // Find root of the source vertex.
            int nextDest = find(edge.dest); // Find root of the destination vertex.

            // If nextsource and nextdest are different, it indicates that edge.src and edge.dest are in
            // different subsets, /the edge can be safely added without forming a cycle
            if (nextSource != nextDest) {
                mst.add(edge); // Add the edge to the MST
                parent[nextDest] = nextSource; // Union the two subsets.
                mst_weight += edge.distance; // Add the weight of the edge to the MST weight
            }
            //If MST contains enough edges to span all vertices, stop the process.
            if (mst.size() == V - 1) {
                break;
            }

        }
        //display the output
        writer.println(x:"Edges in MST");
        for (EdgeK edge : mst) {
            writer.println(edge.src + " -- " + edge.dest + " distance:  " + edge.distance);
        }
        writer.println("\nTotal MST weight: " + mst_weight);
    }
```

Union find:

```java
    public int find(int i) {
        // This condition checks whether the current element i is its own parent.
        // If i is its own parent, it means i is the root of its subset
        if (parent[i] != i)
            parent[i] = find(parent[i]);
        return parent[i];
    }
```

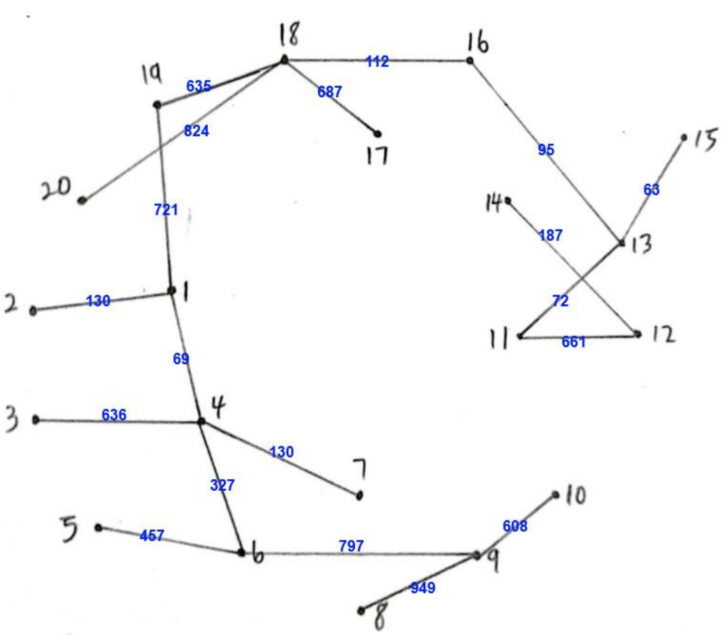
Every edges in the arraylist is loop through to find out whether adding the edge in the MST would create a cycle. Firstly, it finds the root of the source vertex and destination vertex. **if (parent[i] != i)** checks whether the current vertex i is its own parent. If i is its own parent, it indicates that i is the root of its subset. If i is not its own parent, it means i is part of a larger subset, and we need to find the root of this subset. If i is not the root, the function makes a recursive call to find the root of i's parent and until it finds the root of i's parent, it returns it.

Next, it compares the source and destination and if nextSource and nextDest is different, it indicates that the source and destination are in different subsets, which can be safely added in the MST. However, if nextSource and nextDest are the same value, it means that the root of both source and destination are the same (belonging to same parent), in this case adding this edge would create a cycle, hence it will not be added to the MST. The process continues until enough edges have been added to span all vertices in the graph, and the output is saved in a file.

Output in K_results.txt

```
Edges in MST
13 -- 15 distance:  63
1 -- 4 distance:  69
11 -- 13 distance:  72
13 -- 16 distance:  95
16 -- 18 distance:  112
1 -- 2 distance:  130
4 -- 7 distance:  130
12 -- 14 distance:  187
4 -- 6 distance:  327
5 -- 6 distance:  457
9 -- 10 distance:  608
18 -- 19 distance:  635
3 -- 4 distance:  636
11 -- 12 distance:  661
17 -- 18 distance:  687
19 -- 1 distance:  721
6 -- 9 distance:  797
18 -- 20 distance:  824
8 -- 9 distance:  949

Total MST weight: 8160
```

Graph of minimum spanning tree



Discussion:

To discuss about the time complexity of the algorithm, the algorithm firsts uses the sorting operation **collections.sort()**, which internally uses a modified version of the merge sort algorithm to sort the list efficiently, hence taking O (E * log E) time to sort edges, where E is the number of edges. The **find()** operation compresses the path by making every node along the path point directly to the root, which has an amortized time complexity of O(α(V)), where α(alpha) is the inverse Ackermann function, which grows very slowly. Given that the checking for each edge and performing the find operation twice per edge (once for each vertex), the cost for all find operations over all edges is O(Eα(V)) and each union operation itself is O(1). Hence, O(E log E) + O(Eα(V)) ≈ O(E log E) is the time complexity.

For the space complexity, **int [] parent**, this array keeps track of the root of each subset to which a vertex belongs. The size of this array is directly proportional to the number of vertices, V, as it includes an entry for each vertex. Hence, the space complexity contribution from the parent array is O(V). For edge list, **ArrayList<EdgeK> edges = new ArrayList<>()**, storing each edge to sort and process them, the space complexity contribution from the edge list is O(E). The MST list, **List<EdgeK> mst = new ArrayList<>()** stores the edges that are included in the Minimum Spanning Tree (MST). In the worst case, an MST for a connected graph with V vertices has V−1 edges. Therefore, the space complexity for the MST list is O(V). Besides, other variables and temporary storage used for sorting and looping through the edges contribute a minor amount, generally considered O(1) in space complexity analysis. By combining all, the space complexity would be O(V+E).

Question 4: Dynamic Programming

Knapsack Algorithm Initialization:

```java
        int maxCapacity = 800;  // Maximum capacity
        int[][] knapsackTable = new int[num_of_stars.size() + 1][maxCapacity + 1];  // Table row and column 21 x 801
        ArrayList<Integer> selectedStars = new ArrayList<>();
        int[] result = knapsack(stars_weight, stars_profit, num_of_stars, maxCapacity, knapsackTable, selectedStars);

        System.out.println("Total profit: " + result[0]);
        System.out.println("Total weight: " + result[1]);

        saveResults(filename:"Knapsack.csv", knapsackTable, selectedStars, result[0], result[1], num_of_stars, stars_weight, stars_profit);
    }
```

This part first defines the maximum capacity of the bag as 800, following is the table named **`knapsackTable`**. **`ArrayList<Integer> selectedStars = new ArrayList<>(),`** initializes the list to store selected items. **`int[] result = knapsack(...)`**: Calls the knapsack method and stores the result. Total profit and total weight are then printed. **`saveResults(...)`**: Calls the saveResults method to write the results to a file.

Knapsack Method:

```java
public static int[] knapsack(ArrayList<Integer> weights, ArrayList<Integer> profits, ArrayList<Integer> stars, int maxCapacity, int[][] knapsackTable, ArrayList<Integer> selectedStars) {
        int n = weights.size(); // Number of items

        // Create table
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= maxCapacity; w++) {
// Condition 1, if bag capacity =0 or no items, profit is 0
                if (i == 0 || w == 0) {
                    knapsackTable[i][w] = 0;
                }
                // Condition 2, items weight <= bag weight
                else if (weights.get(i - 1) <= w) {
                    knapsackTable[i][w] = Math.max(profits.get(i - 1) + knapsackTable[i - 1][w - weights.get(i - 1)], knapsackTable[i - 1][w]);
                }
                // Condition 3, items weight > bag weight
                else {
                    knapsackTable[i][w] = knapsackTable[i - 1][w];
                }
            }
        }
```

This part shows the knapsack algorithm, the table is filled with the profits and weights. Within the nested loops, the algorithm follows three conditions to fill the table:

1. Condition 1 (if (i == 0 || w == 0)): If there are no items (i == 0) or the knapsack capacity is zero (w == 0), the profit is zero. This initializes the first row and column of the table to zero.
2. Condition 2 (else if (weights.get(i - 1) <= w)): If the weight of the current item (weights.get(i - 1)) is less than or equal to the current capacity (w), the algorithm considers including the item in the knapsack. It calculates the maximum profit by comparing the profit of including the item (profits.get(i - 1) + knapsackTable[i - 1][w - weights.get(i - 1)]) with the profit of excluding the item (knapsackTable[i - 1][w]). The result is stored in knapsackTable[i][w].
3. Condition 3 (else): If the weight of the current item exceeds the current capacity, the item cannot be included. Therefore, the value is the same as the value without the current item (knapsackTable[i - 1][w]).

Checking for selected stars:

```
// To find out which items are included, we need to backtrack
        int profit_result = knapsackTable[n][maxCapacity];
        int w = maxCapacity;
        int totalWeight = 0;

        for (int i = n; i > 0 && profit_result > 0; i--) {
            // If the profit is different, then this item was included
            if (profit_result != knapsackTable[i - 1][w]) {
                selectedStars.add(i - 1);

                // Since this item is included, its weight is subtracted
                profit_result -= profits.get(i - 1);
                w -= weights.get(i - 1);

                // Add weight of the selected item
                totalWeight += weights.get(i - 1);
            }
        }

        System.out.println("Selected stars: " + selectedStars);

        return new int[]{knapsackTable[n][maxCapacity], totalWeight};
    }
```

This part shows the backtrack method so that we can find which stars are selected. By iterating backward through the items and comparing the profit values, it determines and records the selected items, updates the remaining profit and capacity, and calculates the total weight of the selected items. The result including the maximum profit and total weight is returned.

Output:

Knapsack.csv

Discussion:

Time Complexity: For the first part of the algorithm is about filling in the knapsack table with the star's profit. The table has n +1 rows and W +1 columns, where n is the number of items and W is the maximum capacity of the knapsack. The algorithm fills each cell of the table, and there are $(n + 1)$ x$(W + 1)$ cells. Each cell operation (filling the cell) takes constant time $O(1)$. Therefore, the overall time complexity for filling the table is $O(n \times W)$. Then for the second part of the algorithm is about backtracking so that we can find the selected stars. This backtracking step iterates through most n items, with each iteration taking constant time $O(1)$. Therefore, the time complexity for backtracking is $O(n)$. Overall, the time complexity of the knapsack algorithm is $O(n \times W)$.

Space complexity: For the knapsack table, the table has $(n+1) \times (W+1)$ elements. Thus, the space required for the table is $O(n \times W)$. The algorithm uses several auxiliary data structures, including the **`num_of_stars`**, **`stars_weight`**, **`stars_profit`**, and **`selectedStars`** lists. Each of these lists has at most n elements. Therefore, the space required for the auxiliary data structures is $O(n)$. Overall, the space complexity of the knapsack algorithm is $O(n \times W)$.

References

1. GeeksforGeeks. (n.d.). Dijkstra's Algorithm for Adjacency List Representation (Greedy Algo-8). Retrieved from https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/
2. GeeksforGeeks. (2024, May 30). 0/1 Knapsack problem. Retrieved from https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
3. ChatGPT on [how to write a Kruskal algorithm in java]
4. ChatGPT on [Read and Write CSV file java]
5. Heap Sort and Selection Sort from Algorithm Design and Analysis lab 07.