



Texas A&M University

COURSE PROJECT REPORT

FALL 2022 CSCE 629-601: ANALYSIS OF ALGORITHMS

Submitted by: Ajinkya Bharat Malhotra (UIN#633002196)



Contents

INTRODUCTION

- Overview
- Background and Motivation
- Objective

IMPLEMENTATION

- Tools Used
- Graph Structure
- Random Graph Generation
 - Sparse Graph
 - Dense Graph
- Heap Structure
- Max-Bandwidth-Path Routing Algorithms
 - Dijkstra's without heap structure
 - Dijkstra's with heap structure
 - Kruskal's
 - Time Complexity
 - UML diagrams

TESTING

- How to run
- Sample Console Output

PERFORMANCE COMPARISON and ANALYSIS

- Sparse Graph
- Dense Graph

FURTHER IMPROVEMENTS

Introduction

Overview

This report discusses the result of the work done in CSCE 629 Analysis of Algorithm course project. It involves generating randomly connected sparse and dense graphs, along with running 3 different kinds of routing algorithms on the randomly generated graphs.

Background and Motivation

Network optimization has been an important area in the current research in computer science and computer engineering. In this course project, I have implemented network routing protocols using the data structures and algorithms studied in class. This provided me an opportunity to translate the learned theoretical understanding into a real-world practical computer program. This course project helped me understand that translating algorithmic ideas at a “higher level” of abstraction into real implementations in a particular programming language is not at all always trivial. The implementations forced me to work on more details of the algorithms, which led to much better in-depth understanding of the algorithms.

Objective

The final goal of this project was to implement all the graph functionalities along with the routing algorithms, to compare and analyze the performance difference between all 3 routing algorithms on different types of randomly generated graphs.

Implementation

Tools Used

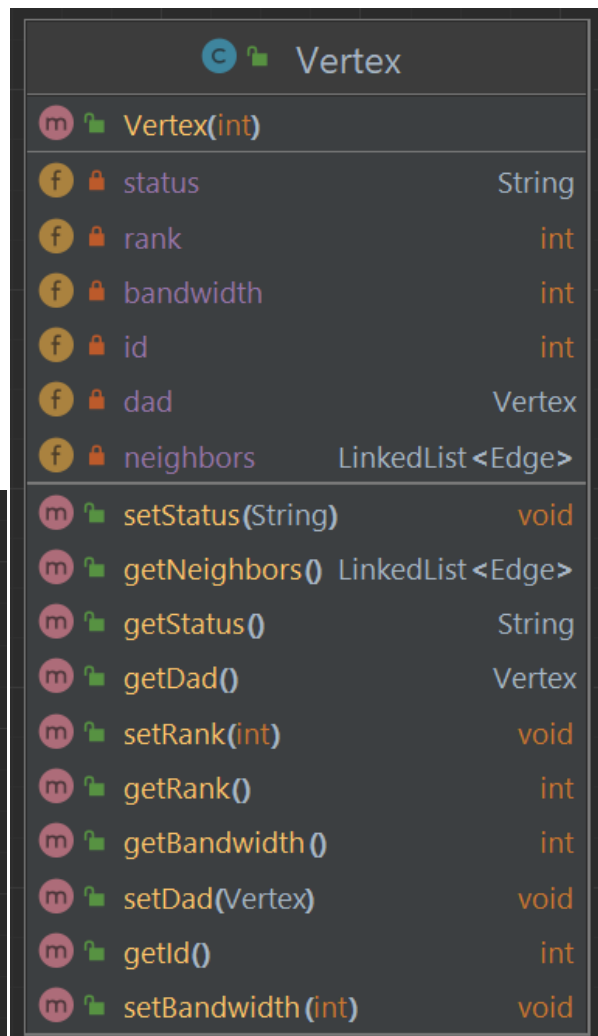
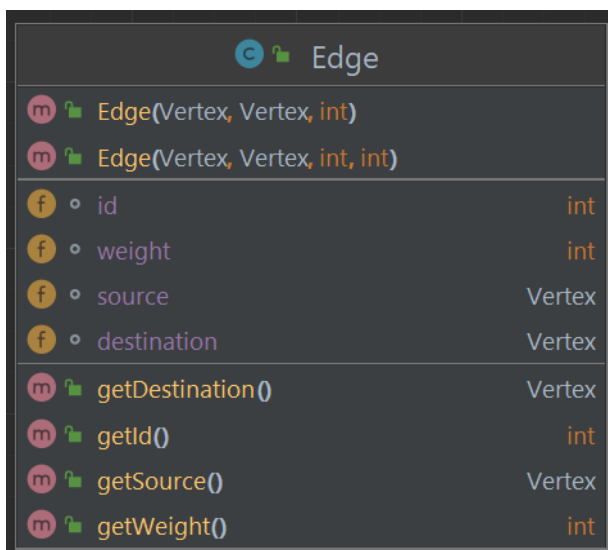
This project was implemented using **java** programming language with the minimal use of default libraries/classes to keep the performance streamlined across all algorithms. The only two default libraries/classes used in the java code were **Random** library/class to generate random graphs and the default **LinkedList** library/class from **java.util** package to store the neighbors/edges of a given vertex to form a graph. This project was developed and tested using Java 18 SDK but has a backward and forward compatibility with other java SDKs.

Graph Structure (*Vertex.java* and *Edge.java*)

The Graph Structure consists of the following two classes. The UML diagram represents constructor, methods and fields used in each class respectively:

Vertex.java stores vertex related data such as dad, rank, status, neighbors, bandwidth, and vertex id.

Edge.java stores edge related data such as source, destination, edge id and weight.



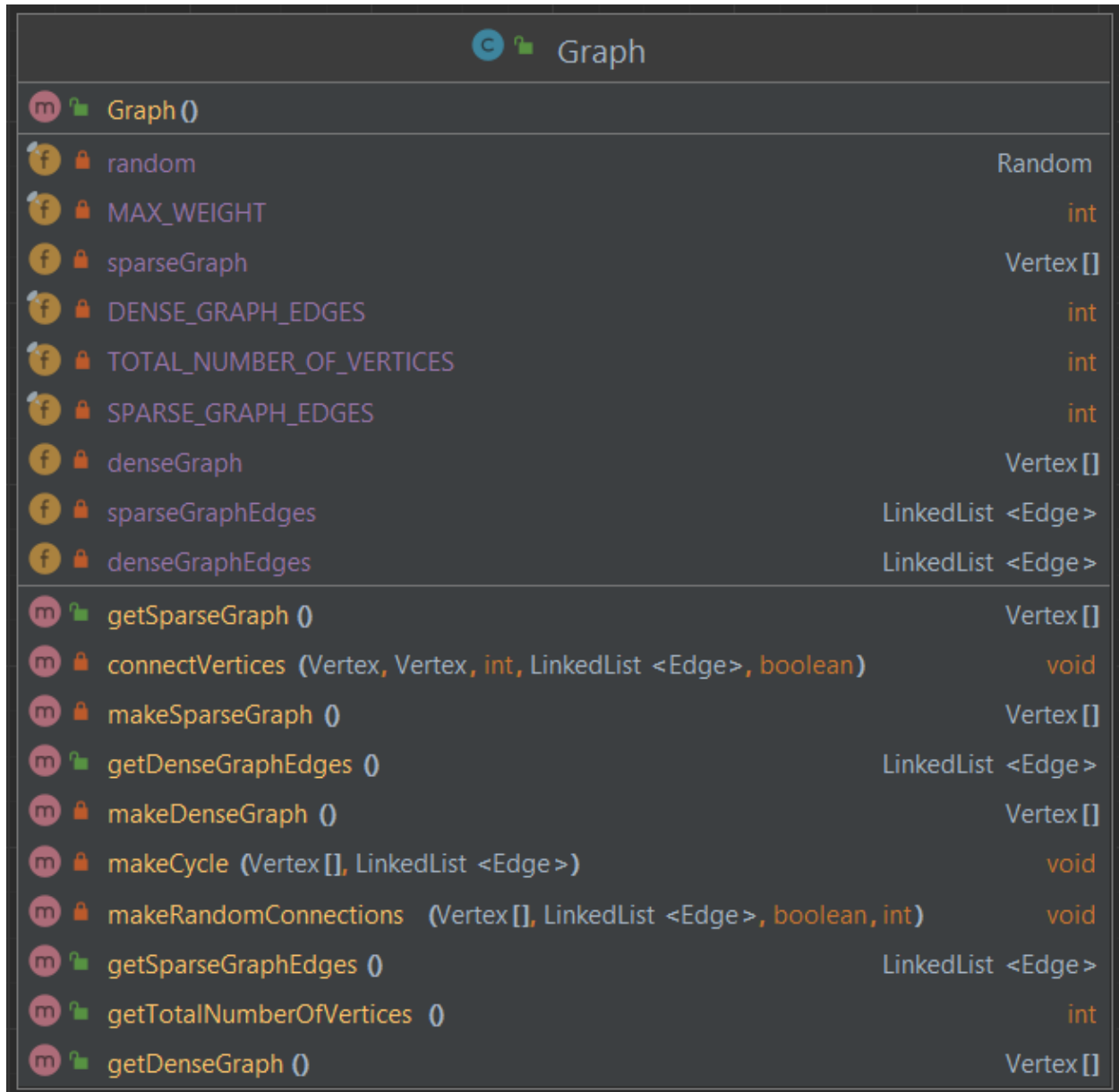
Random Graph Generation (*Graph.java*)

The entire code of graph generation can be found in **Graph.java** class. ***Graph.java* is responsible for generating Vertex[] array which represents the graph.** To generate a random graph where all the vertices are always connected, I start by creating all 5000 vertices and manually connecting all the vertices to their adjacent vertex. This creates a cycle which ensures that all 5000 vertices are connected in the following manner:

$$V0 \rightarrow V1 \rightarrow V2 \rightarrow V3 \rightarrow V4 \rightarrow V5 \rightarrow \dots \rightarrow V4999 \rightarrow V0$$

The motivation behind this idea is that since edges are being generated randomly between vertices, there can be scenarios where a path doesn't exist between vertex X and vertex Y. If the above-mentioned scenario occurs, then all 3 algorithms will not be able to find the path between vertex X and Vertex Y as it doesn't exist. The cycle-creation step ensures that there exists at least one path which will reach any vertex from any vertex. After the cycle is created, I move on to randomly generate N number of edges for each vertex, N will vary depending on the type of graph we are generating. The weight of an edge is selected randomly for both sparse and dense graph. Once the source vertex, randomly selected destination vertex and randomly generated weight have been computed, I call the ***connectVertices()*** method/subroutine to create an edge between two vertices. I follow the exact steps in a for loop until all vertices in the graph have N neighbors.

The following UML diagram represents constructor, methods, and fields in **Graph.java** class.



makeCycle() subroutine/method creates a cycle as explained earlier to ensure that there exists at least one path from any vertex to any vertex.

```
/**
 * This method connects all the vertices to its adjacent vertex and make a cycle for the entire graph.
 *
 * @param graph - Vertex object array which represents the entire graph.
 * @param edges - LinkedList of all the edges object of the above-mentioned graph.
 */
2 usages
private void makeCycle(Vertex[] graph, LinkedList<Edge> edges){
    for(int i=0; i<this.TOTAL_NUMBER_OF_VERTICES; ++i) {
        int nextVertexIndex = (i==this.TOTAL_NUMBER_OF_VERTICES-1) ? 0 : i+1;
        Vertex currVertex = graph[i];
        Vertex nextVertex = graph[nextVertexIndex];
        int weight = random.nextInt(MAX_WEIGHT) + 1;
        connectVertices(currVertex, nextVertex, weight, edges, backEdge: false);
        connectVertices(nextVertex, currVertex, weight, edges, backEdge: true);
    }
}
```

connectVertices() subroutine/method creates an object of type **Edge.java** and connects two vertices with the randomly generated weight.

```
/**
 * This method connects two vertices by adding a new edge into the adjacency list of the source.
 *
 * @param source - Source Vertex.
 * @param destination - Destination Vertex.
 * @param weight - Weight of the edge.
 * @param edges - LinkedList of all the edges object of the above-mentioned graph.
 * @param backEdge - Boolean flag to determine whether the edge being added is a backEdge or not.
 */
4 usages
private void connectVertices(Vertex source, Vertex destination,
                             int weight, LinkedList<Edge> edges, boolean backEdge){
    for(Edge e : source.getNeighbors())
        if(e.destination == destination)
            return;

    Edge edge = new Edge(source, destination, weight);
    source.getNeighbors().add(edge);

    if(!backEdge)
        edges.add(edge);
}
```

makeSparseGraph() subroutine/method starts the generation of sparse graphs.

```
/**
 * This method starts sparse graph generation by initializing all vertices, calling makeCycle method and
 * lastly calling makeRandomConnections method.
 *
 * @return - Vertex object array which represents the entire graph.
 */
1 usage
private Vertex[] makeSparseGraph(){
    System.out.print("Sparse graph generation in progress..... ");
    Vertex[] sparseGraph = new Vertex[TOTAL_NUMBER_OF_VERTICES];
    for(int i=0; i<this.TOTAL_NUMBER_OF_VERTICES; ++i){
        Vertex v = new Vertex(i);
        sparseGraph[i] = v;
    }
    makeCycle(sparseGraph, this.sparseGraphEdges);
    makeRandomConnections(sparseGraph, this.sparseGraphEdges, sparseGraph: true, retryRate: 50000);
    System.out.println("Done!!");
    return sparseGraph;
}
```

makeDenseGraph() subroutine/method starts the generation of dense graphs.

```
/**
 * This method starts dense graph generation by initializing all vertices, calling makeCycle method and
 * lastly calling makeRandomConnections method.
 *
 * @return - Vertex object array which represents the entire graph.
 */
1 usage
private Vertex[] makeDenseGraph(){
    System.out.print("Dense graph generation in progress..... ");
    Vertex[] denseGraph = new Vertex[TOTAL_NUMBER_OF_VERTICES];
    for(int i=0; i<this.TOTAL_NUMBER_OF_VERTICES; ++i) {
        Vertex v = new Vertex(i);
        denseGraph[i] = v;
    }

    makeCycle(denseGraph, this.denseGraphEdges);
    makeRandomConnections(denseGraph, this.denseGraphEdges, sparseGraph: false, retryRate: 10);
    System.out.println("Done!!");

    return denseGraph;
}
```


makeRandomConnection() subroutine/method creates random connections between all vertices in the graph. The number of random connections per vertex will depend on what type of graph is being generated. To ensure that we get the desired number of connections per vertex, I am computing a random variable while looping through the array of vertices created before in ***makeSparseGraph()/makeDenseGraph()*** subroutines.

```
/**
 * This method makes random connection until the numberOfConnectionsPerVertex requirement is met for the graph.
 *
 * @param graph - Vertex object array which represents the entire graph.
 * @param edges - LinkedList of all the edges object of the above-mentioned graph.
 * @param sparseGraph - Boolean flag to determine whether we are generating connections for sparse or dense graph.
 * @param retryRate - Retry rate value.
 */
2 usages
private void makeRandomConnections(Vertex[] graph, LinkedList<Edge> edges, boolean sparseGraph, int retryRate){
    for(int i=0; i<this.TOTAL_NUMBER_OF_VERTICES; ++i) {
        int numberOfConnectionsPerVertex =
            (sparseGraph) ? this.SPARSE_GRAPH_EDGES : random.nextInt( bound: 150) + this.DENSE_GRAPH_EDGES;
        Vertex currVertex = graph[i];
        while(currVertex.getNeighbors().size() < numberOfConnectionsPerVertex){
            int retry = 0;
            Vertex randomVertex = graph[random.nextInt(TOTAL_NUMBER_OF_VERTICES)];
            while(retry <= retryRate && (randomVertex == currVertex ||
                randomVertex.getNeighbors().size() >= numberOfConnectionsPerVertex)) {
                randomVertex = graph[random.nextInt(TOTAL_NUMBER_OF_VERTICES)];
                retry++;
            }

            if(retry > retryRate)
                break;

            int weight = random.nextInt(MAX_WEIGHT) + 1;
            connectVertices(currVertex, randomVertex, weight, edges, backEdge: false);
            connectVertices(randomVertex, currVertex, weight, edges, backEdge: true);
        }
    }
}
```

The difference between each graph type generation is explained below.

Sparse Graph (*Graph.java*)

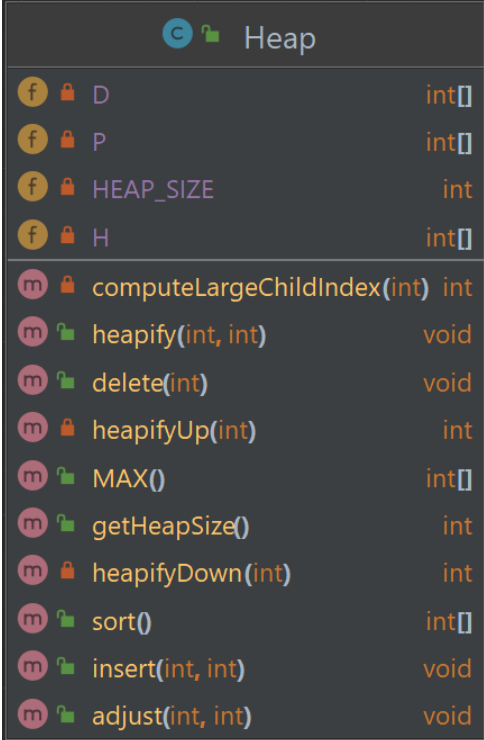
Sparse Graph generation starts by calling the *makeSparseGraph()* subroutine/method. In this subroutine/method, I create all 5000 vertex objects and store them in an object `Vertex[]` array. After this I call the *makeCycle()* method to generate a cycle between those 5000 vertices. After this I move on to generate random edges between vertices using the *makeRandomConnection()* method. For sparse graphs, the number of connections per vertex is set to exactly 6 to ensure that the average vertex degree of the entire graph is always 6.

Dense Graph (*Graph.java*)

Dense Graph generation starts by calling the *makeDenseGraph()* subroutine/method. In this subroutine/method, I create all 5000 vertex objects and store them in an object `Vertex[]` array. After this I call the *makeCycle()* method to generate a cycle between those 5000 vertices. After this I move on to generate random edges between vertices using the *makeRandomConnection()* method. For dense graphs the number of connections per vertex will be randomly select for each vertex ranging from 950 to 1100 to ensure that we get about 20% as our average vertex degree of the entire graph. In our case as we have 5000 vertices in the entire graph, this implies that each vertex will have about 950-1100 edges/neighbors and thus the average vertex degree of the entire graph will be approximately 1000 or 20% to the total number of vertices in graph.

Heap Structure (*Heap.java*)

The heap is given by an array $H[5000]$, where each element $H[i]$ gives the id of a vertex in the graph. The element(vertex/edge) value(bandwidths/weights) is given in array $D[5000]$. Thus, to find the value of an element $H[i]$ in the heap, we can use $D[H[i]]$. In the operation $\text{delete}(v)$ that deletes the element from the heap H , I need to find the position of the vertex in the heap. For this, I have used another array $P[5000]$ so that $P[v]$ is the position/index of vertex v in the heap H . The heap implementation can be found in *heap.java* class along with all the methods/subroutines.



Heap		
f	D	int[]
f	P	int[]
f	HEAP_SIZE	int
f	H	int[]
m	computeLargeChildIndex(int)	int
m	heapify(int, int)	void
m	delete(int)	void
m	heapifyUp(int)	int
m	MAX()	int[]
m	getHeapSize()	int
m	heapifyDown(int)	int
m	sort()	int[]
m	insert(int, int)	void
m	adjust(int, int)	void

computeLargeChildIndex() method/subroutine computes the index of the largest child of a given element in the heap.

heapify() method/subroutine adjusts H , D , P arrays in a non-decreasing order.

delete() method/subroutine deletes an element from H , D , P arrays.

heapifyUp() method/subroutine moves an element up in H , D , P arrays if its data value in D is greater than its parent.

MAX() method/subroutine returns the 0th element in H and the 0th element data in D .

heapifyDown() method/subroutine moves an element down in H , D , P arrays if its bandwidth/weight value is smaller than its either children.

sort() method/subroutine starts the sorting of H , D , P arrays in non-decreasing order.

insert() method/subroutine inserts an element in H , D , P arrays.

adjust() method/subroutine updates an element H value in D and calls $\text{heapify}()$ to adjust the updated element to the correct position in H , D , P arrays.

Max-Bandwidth-Path Routing Algorithms

Dijkstra's *without* heap structure (*Dijkstras.java, withoutHeap() subroutine*)

```
/**
 * This method runs the Dijkstras WITHOUT heap algorithm.
 * Time Complexity:  $O(n^2)$ 
 *
 * @param graph - Vertex object array which represents the entire graph.
 * @param sourceIndex - Index of the selected source vertex.
 * @param typeOfGraph - String to specify whether the algorithm is being run on sparse or dense graph.
 */
1 usage
public void withoutHeap(Vertex[] graph, int sourceIndex, String typeOfGraph){
    long start = System.currentTimeMillis();

    Vertex source = graph[sourceIndex];
    source.setStatus(IN_TREE); source.setBandwidth(Integer.MAX_VALUE); source.setDad(null);

    LinkedList<Vertex> fringers = new LinkedList<>(); int numberOfFringers = 0;
    for(Edge edge : source.getNeighbors()){
        Vertex w = edge.getDestination();
        w.setStatus(FRINGER); w.setDad(source); w.setBandwidth(edge.getWeight());
        fringers.add(w); numberOfFringers++;
    }

    while(numberOfFringers != 0){
        Vertex y = new Vertex( id: -1);
        y.setBandwidth(Integer.MIN_VALUE);
        for(Vertex fringe : fringers){ //  $O(n)$  <-- will be decreased to  $O(\log n)$  with heap
            if(fringe.getBandwidth() > y.getBandwidth())
                y = fringe;
        }
        y.setStatus(IN_TREE); fringers.remove(y); numberOfFringers--;

        for(Edge edge : y.getNeighbors()){
            int wMinBandWidth = Math.min(y.getBandwidth(), edge.getWeight());
            Vertex w = edge.getDestination();
            if(w.getStatus().equals(UNSEEN)){
                w.setStatus(FRINGER); w.setDad(y); w.setBandwidth(wMinBandWidth);
                fringers.add(w); numberOfFringers++;
            }
            else if(w.getStatus().equals(FRINGER) && w.getBandwidth() < wMinBandWidth){
                w.setDad(y); w.setBandwidth(wMinBandWidth);
            }
        }
    }

    Helper.endTimerAndPrintStats(start, algoName: "Dijkstras Without Heap", typeOfGraph);
}
```

Dijkstra's with heap structure (*Dijkstras.java*, with *Heap()* subroutine)

```
/**
 * This method runs the Dijkstras WITH heap algorithm.
 * Time Complexity:  $O((n+m) * \log n) = O(m * \log n)$ 
 *
 * @param graph - Vertex object array which represents the entire graph.
 * @param sourceIndex - Index of the selected source vertex.
 * @param typeOfGraph - String to specify whether the algorithm is being run on sparse or dense graph.
 */
1 usage
public void withHeap(Vertex[] graph, int sourceIndex, String typeOfGraph){
    long start = System.currentTimeMillis();

    Vertex source = graph[sourceIndex];
    source.setStatus(IN_TREE); source.setBandwidth(Integer.MAX_VALUE); source.setDad(null);

    for(Edge edge : source.getNeighbors()){
        Vertex w = edge.getDestination();
        w.setStatus(FRINGER); w.setDad(source); w.setBandwidth(edge.getWeight());
        heap.insert(w.getId(), w.getBandwidth());
    }

    while(heap.getHeapSize() > 0){
        int[] max = heap.MAX();
        Vertex v = graph[max[0]];
        v.setStatus(IN_TREE);
        heap.delete(v.getId());

        for(Edge edge : v.getNeighbors()){
            int wMinBandWidth = Math.min(v.getBandwidth(), edge.getWeight());
            Vertex w = edge.getDestination();
            if(w.getStatus().equals(UNSEEN)){
                w.setStatus(FRINGER); w.setDad(v); w.setBandwidth(wMinBandWidth);
                heap.insert(w.getId(), w.getBandwidth());
            }
            else if(w.getStatus().equals(FRINGER) && w.getBandwidth() < wMinBandWidth){
                w.setDad(v); w.setBandwidth(wMinBandWidth);
                heap.adjust(w.getId(), w.getBandwidth());
            }
        }
    }

    Helper.endTimerAndPrintStats(start, algoName: "Dijkstras With Heap", typeOfGraph);
}
```

Kruskal's (*Kruskals.java*, *kruskals()* subroutine)

```
/**
 * This method runs the Kruskal's algorithm.
 * Time Complexity:  $O(m * \log m)$ 
 *
 * @param graph - Vertex object array which represents the entire graph.
 * @param edges - LinkedList of all the edges object of the above-mentioned graph.
 * @param totalNumberOfVertices - Total number of vertices in the graph.
 * @param typeOfGraph - String to specify whether the algorithm is being run on sparse or dense graph.
 */
1 usage
public Vertex[] kruskals(Vertex[] graph, LinkedList<Edge> edges,
    int totalNumberOfVertices, String typeOfGraph){
    long start = System.currentTimeMillis();

    for(int i=0; i<totalNumberOfVertices; ++i)
        T[i] = makeSet(graph[i]);

    generateHeapAndEdges(edges);
    int[] sortedEdges = heap.sort();

    for(int i=0; i< heap.getHeapSize(); ++i){
        Edge maxEdge = allEdges[sortedEdges[i]];
        Vertex u = maxEdge.getSource();
        Vertex v = maxEdge.getDestination();

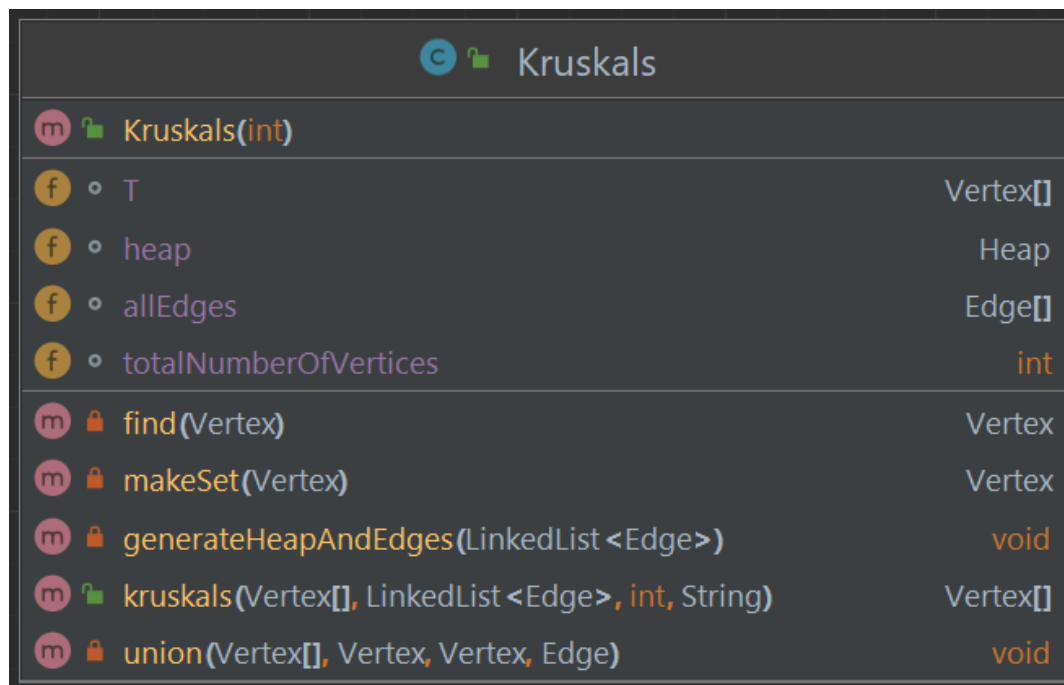
        Vertex rootU = find(u);
        Vertex rootV = find(v);
        if(rootV != rootU)
            union(T, rootU, rootV, maxEdge);
    }

    Helper.endTimerAndPrintStats(start, algoName: "Krushkals on ", typeOfGraph);
    return T;
}
```

Time Complexity

- **Dijkstra's:** $O(n^2)$ where n is the number of vertices in graph.
- **Dijkstra's WITH heap:** $O((m+n) \log n) = O(m \log n)$ where n is the number of vertices and m is the number of edges in graph.
- **Kruskal's:** $O(m \log m)$ where m is the number of edges in graph.

UML Diagrams



Testing

The *Driver.java* code is responsible for generating graphs and running all 3 algorithms on the generated graphs. To test the routing algorithms, I have randomly generated 5 pairs of graphs G1(Sparse) and G2(Dense) graphs using the subroutines implemented in *Graph.java* class. For each graph, I pick 5 pairs of randomly selected source-destination vertices. For each source-destination pair (s, t) on graph G, I am running all the three algorithms on the pair (s, t) and the graph G, and recording their maximum bandwidth, along with running time in milliseconds and the path from s to t in G.

To validate the correctness of all the algorithms, the following piece of code in *Helper.java* class checks the bandwidth of all 3 algorithms and throws a custom exception if the bandwidth is different for any of the algorithms. Under *NO* circumstances, the maximum bandwidth should be different for any of the algorithms. *The maximum Bandwidth of all 3 algorithms should always be the same.*

```
/**
 * Method to make sure that the bandwidths for all 3 algorithms is exactly the same.
 * If it is not then a custom exception is raised.
 *
 * @param maxBwFromDijk - Max bandwidth value from Dijkstras without heap algo.
 * @param maxBwFromDijkWithHeap - Max bandwidth value from Dijkstras with heap algo.
 * @param maxBwFromKrushkals - Max bandwidth value from Krushkals algo.
 * @throws Exception - Exception to report the invalid result as all algos should have the same bandwidth value.
 */
2 usages
public void checkBW(int maxBwFromDijk, int maxBwFromDijkWithHeap, int maxBwFromKrushkals) throws Exception {
    if (maxBwFromDijk != maxBwFromKrushkals || maxBwFromDijkWithHeap != maxBwFromKrushkals) {
        throw new Exception("Invalid result as Maximum Bandwidth value should be the same for all algos");
    }
    System.out.println("=====");
}
```

How to run

By running the *Driver.java* file you can run all three algorithms on Dense and Sparse graphs. To adjust the number of times sparse and dense graph are generated and to adjust the number of times random source and destination are generated on each graph, adjust the following two variable's value in *Driver.java* class:

NUMBER_OF_GRAPHS | NUMBER_OF_PAIRS_PER_GRAPH

Sample Console Output

The following is the sample console output after running all three algorithms on a single sparse and dense graph for only one randomly selected (s, t) pair:

Sparse graph generation in progress..... Done!!

Dense graph generation in progress..... Done!!

Degree of SpareGraph is :6

Degree of DenseGraph is :1004

Running Algorithms:

=====
Printing stats for Dijkstras Without Heap on SparseGraph:

Execution time: 47 milliseconds

Maximum Bandwidth from source vertex (1284) to destination vertex (2102) is: 6228

Path from (1284) to (2102) is (V:2102, W:6228) <= (V:2101, W:7639) <= (V:2100, W:7639) <= (V:2099, W:7756) <= (V:4700, W:7756) <= (V:3840, W:7756) <= (V:4089, W:7756) <= (V:3619, W:7756) <= (V:4365, W:7756) <= (V:1361, W:7756) <= (V:1362, W:7756) <= (V:1322, W:7756) <= (V:1323, W:7756) <= (V:1324, W:7756) <= (V:3017, W:7756) <= (V:3018, W:7756) <= (V:4322, W:7756) <= (V:4805, W:7756) <= (V:4806, W:7756) <= (V:4474, W:7756) <= (V:4473, W:7756) <= (V:2816, W:7756) <= (V:737, W:7756) <= (V:736, W:7756) <= (V:735, W:7756) <= (V:734, W:7756) <= (V:733, W:7756) <= (V:1166, W:7756) <= (V:4181, W:7756) <= (V:4180, W:7756) <= (V:4301, W:7756) <= (V:570, W:7756) <= (V:571, W:7756) <= (V:3762, W:7756) <= (V:3526, W:7756) <= (V:3525, W:7756) <= (V:2642, W:7756) <= (V:2203, W:7756) <= (V:2204, W:7756) <= (V:1227, W:7756) <= (V:1228, W:7756) <= (V:2089, W:7756) <= (V:206, W:7756) <= (V:1038, W:7756) <= (V:1037, W:7756) <= (V:3431, W:7756) <= (V:2202, W:7756) <= (V:3005, W:7756) <= (V:3006, W:7756) <= (V:541, W:7756) <= (V:540, W:7756) <= (V:920, W:7756) <= (V:4233, W:7756) <= (V:4234, W:7756) <= (V:299, W:7756) <= (V:3534, W:7756) <= (V:3417, W:7756) <= (V:3418, W:7756) <= (V:3419, W:7756) <= (V:3969, W:7756) <= (V:41, W:7756) <= (V:40, W:7756) <= (V:39, W:8306) <= (V:4370, W:8306) <= (V:818, W:9404) <= (V:1284, W:2147483647)

=====

Printing stats for Dijkstras With Heap on SparseGraph:

Execution time: 1 milliseconds

Maximum Bandwidth from source vertex (1284) to destination vertex (2102) is: 6228

Path from (1284) to (2102) is (V:2102, W:6228) <= (V:2101, W:7639) <= (V:2100, W:7639) <= (V:2099, W:7756) <= (V:4700, W:7756) <= (V:3840, W:7756) <= (V:4089, W:7756) <= (V:3619, W:7756) <= (V:4365, W:7756) <= (V:1361, W:7756) <= (V:1362, W:7756) <= (V:1322, W:7756) <= (V:1323, W:7756) <= (V:1324, W:7756) <= (V:3017, W:7756) <= (V:3018, W:7756) <= (V:4322, W:7756) <= (V:4805, W:7756) <= (V:4806, W:7756) <= (V:4474, W:7756) <= (V:4473, W:7756) <= (V:2816, W:7756) <= (V:737, W:7756) <= (V:736, W:7756) <= (V:735, W:7756) <= (V:734, W:7756) <= (V:733, W:7756) <= (V:1166, W:7756) <= (V:4181, W:7756) <= (V:4180, W:7756) <= (V:4301, W:7756) <= (V:570, W:7756) <= (V:571, W:7756) <= (V:3762, W:7756) <= (V:3526, W:7756) <= (V:3525, W:7756) <= (V:2642, W:7756) <= (V:2203, W:7756) <= (V:2204, W:7756) <= (V:1227, W:7756) <= (V:1228, W:7756) <= (V:2089, W:7756) <= (V:206, W:7756) <= (V:1038, W:7756) <= (V:1037, W:7756) <= (V:2384, W:7756) <= (V:2162, W:7756) <= (V:914, W:7756) <= (V:1241, W:7756) <= (V:62, W:7756) <= (V:1708, W:7756) <= (V:306, W:7756) <= (V:3687, W:7756) <= (V:3242, W:7756) <= (V:3243, W:7756) <= (V:4964, W:7756) <= (V:826, W:7756) <= (V:3704, W:7756) <= (V:3639, W:7756) <= (V:3638, W:7756) <= (V:4182, W:7756) <= (V:2765, W:7756) <= (V:299, W:7756) <= (V:3534, W:7756) <= (V:3417, W:7756) <= (V:3418, W:7756) <= (V:3419, W:7756) <= (V:3969, W:7756) <= (V:41, W:7756) <= (V:40, W:7756) <= (V:39, W:8306) <= (V:4370, W:8306) <= (V:818, W:9404) <= (V:1284, W:2147483647)

=====

Execution time generating edges, heap and performing sorting is: 4 milliseconds

Printing stats for Krushkals on SparseGraph:

Execution time: 5 milliseconds

Maximum Bandwidth from source vertex (1284) to destination vertex (2102) is: 6228

Path from (1284) to (2102) is (V:2102, W:6228) <= (V:441, W:7756) <= (V:330, W:7756) <= (V:2417, W:7756) <= (V:2415, W:7756) <= (V:2416, W:7756) <= (V:2488, W:7756) <= (V:101, W:7756) <= (V:3519, W:7756) <= (V:4672, W:7756) <= (V:4673, W:7756) <= (V:2883, W:7756) <= (V:2884, W:7756) <= (V:3241, W:7756) <= (V:3242, W:7756) <= (V:1135, W:7756) <= (V:3243, W:7756) <= (V:4964, W:7756) <= (V:3638, W:7756) <= (V:4182, W:7756) <= (V:299, W:7756) <= (V:891, W:7756) <= (V:2334, W:7756) <= (V:2455, W:7756) <= (V:2456, W:7756) <= (V:3417, W:7756) <= (V:3418, W:7756) <= (V:41, W:7756) <= (V:40, W:7756) <= (V:818, W:9404) <= (V:1284, W:2147483647)

=====

Printing stats for Dijkstras Without Heap on DenseGraph:

Execution time: 114 milliseconds

Maximum Bandwidth from source vertex (1284) to destination vertex (2102) is: 9977

Path from (1284) to (2102) is (V:2102, W:9977) <= (V:1942, W:9977) <= (V:2770, W:9977) <= (V:2785, W:9981) <= (V:2220, W:9981) <= (V:178, W:9981) <= (V:1524, W:9981) <= (V:2437, W:9981) <= (V:2873, W:9981) <= (V:1373, W:9981) <= (V:1425, W:9981) <= (V:2803, W:9981) <= (V:3371, W:9981) <= (V:285, W:9981) <= (V:2323, W:9981) <= (V:4759, W:9981) <= (V:2906, W:9981) <= (V:452, W:9981) <= (V:1581, W:9981) <= (V:3103, W:9981) <= (V:28, W:9981) <= (V:51, W:9981) <= (V:2453, W:9981) <= (V:260, W:9981) <= (V:434, W:9981) <= (V:745, W:9981) <= (V:1472, W:9981) <= (V:2476, W:9981) <= (V:262, W:9981) <= (V:313, W:9981) <= (V:11, W:9981) <= (V:1453, W:9981) <= (V:164, W:9981) <= (V:3161, W:9981) <= (V:2954, W:9981) <= (V:1407, W:9981) <= (V:1898, W:9981) <= (V:4678, W:9981) <= (V:1473, W:9981) <= (V:2664, W:9981) <= (V:2216, W:9981) <= (V:2721, W:9981) <= (V:4835, W:9981) <= (V:1478, W:9981) <= (V:3769, W:9981) <= (V:2246, W:9981) <= (V:790, W:9981) <= (V:784, W:9981) <= (V:1387, W:9981) <= (V:2644, W:9981) <= (V:3310, W:9981) <= (V:2769, W:9981) <= (V:3316, W:9981) <= (V:3880, W:9981) <= (V:1316, W:9981) <= (V:537, W:9981) <= (V:1210, W:9981) <= (V:3266, W:9981) <= (V:2292, W:9981) <= (V:2067, W:9981) <= (V:341, W:9981) <= (V:2151, W:9981) <= (V:2945, W:9981) <= (V:1100, W:9981) <= (V:2212, W:9981) <= (V:617, W:9981) <= (V:2122, W:9981) <= (V:1794, W:9981) <= (V:824, W:9981) <= (V:933, W:9981) <= (V:504, W:9981) <= (V:2772, W:9981) <= (V:1838, W:9981) <= (V:111, W:9981) <= (V:1377, W:9981) <= (V:2671, W:9981) <= (V:4041, W:9981) <= (V:1799, W:9981) <= (V:4209, W:9981) <= (V:1861, W:9981) <= (V:407, W:9981) <= (V:2376, W:9981) <= (V:3203, W:9981) <= (V:99, W:9981) <= (V:724, W:9981) <= (V:4722, W:9981) <= (V:4406, W:9981) <= (V:1396, W:9981) <= (V:3223, W:9981) <= (V:3758, W:9998) <= (V:1284, W:2147483647)

=====

Printing stats for Dijkstras With Heap on DenseGraph:

Execution time: 84 milliseconds

Maximum Bandwidth from source vertex (1284) to destination vertex (2102) is: 9977

Path from (1284) to (2102) is (V:2102, W:9977) <= (V:1942, W:9977) <= (V:2770, W:9977) <= (V:2785, W:9981) <= (V:2220, W:9981) <= (V:178, W:9981) <= (V:1524, W:9981) <= (V:2437, W:9981) <= (V:2873, W:9981) <= (V:2097, W:9981) <= (V:3557, W:9981) <= (V:4672, W:9981) <= (V:4406, W:9981) <= (V:1396, W:9981) <= (V:3223, W:9981) <= (V:3758, W:9998) <= (V:1284, W:2147483647)

=====

Execution time generating edges, heap and performing sorting is: 678 milliseconds

Printing stats for Krushkals on DenseGraph:

Execution time: 967 milliseconds

Maximum Bandwidth from source vertex (1284) to destination vertex (2102) is: 9977

Path from (1284) to (2102) is (V:2102, W:9977) <= (V:2512, W:9981) <= (V:720, W:9981) <= (V:1559, W:9981) <= (V:35, W:9981) <= (V:849, W:9981) <= (V:3114, W:9981) <= (V:4406, W:9981) <= (V:1396, W:9981) <= (V:3223, W:9981) <= (V:1284, W:2147483647)

=====

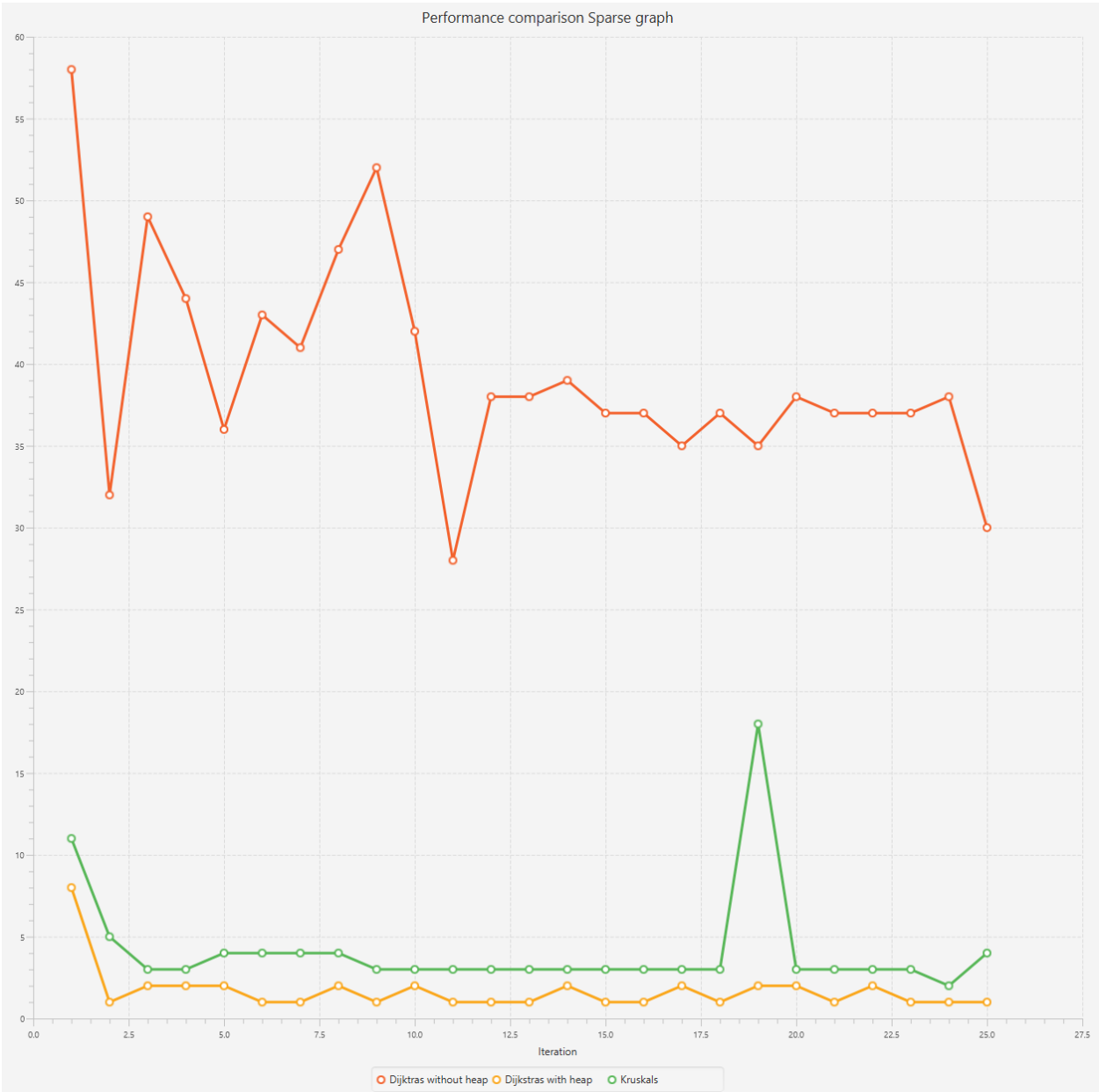
Performance Comparison and Analysis

Sparse Graph

Runtime of all three algorithms in *milliseconds*:

No.	Source to Destination	Dijkstra's	Dijkstra's WITH heap	Kruskal's
1	(1129) to (3406)	58	8	11
2	(1452) to (2206)	32	1	5
3	(2918) to (1493)	49	2	3
4	(605) to (3967)	44	2	3
5	(3082) to (1821)	36	2	4
6	(3160) to (84)	43	1	4
7	(3187) to (578)	41	1	4
8	(1055) to (2184)	47	2	4
9	(2319) to (2956)	52	1	3
10	(1858) to (2394)	42	2	3
11	(2642) to (2123)	28	1	3
12	(3353) to (3243)	38	1	3
13	(4441) to (1465)	38	1	3
14	(1466) to (3865)	39	2	3
15	(2135) to (2137)	37	1	3
16	(3276) to (3181)	37	1	3
17	(4384) to (2570)	35	2	3
18	(2362) to (3103)	37	1	3
19	(4160) to (3509)	35	2	18
20	(3161) to (3424)	38	2	3
21	(4880) to (4040)	37	1	3
22	(701) to (3038)	37	2	3
23	(2269) to (4122)	37	1	3
24	(2946) to (4735)	38	1	2
25	(1313) to (4004)	30	1	4

Performance Comparison in *milliseconds*:



Dijkstra's without heap | **Dijkstra's with heap** | **Kruskal's**

Analysis

For all graphs, including sparse graphs, we have 5000 vertices which makes up the graph and the average vertex degree is 6 for sparse graphs. This implies that every vertex has exactly 6 neighbors or in other words 6 edges which leads to 6 neighboring vertices. **Hence, number of total vertices(n) is 5000 and number of total edges(m) is $5000 \times 6 = 30000/2 = 15000$ for all sparse graphs.** The reason we divide the total number of edges by 2 is because the graph is an undirected graph, every edge gets counted as 2 by default.

As we observe in the above performance table, the following is the pattern in which algorithms complete their execution from quickest to slowest:

Dijkstra's (WITH heap) < Kruskal's < Dijkstra's (WITHOUT heap)

Dijkstra's (with heap) being the fastest, followed by Kruskal's and lastly followed by Dijkstra's (without heap).

This is expected result for sparse graph as Dijkstra's (with heap) has a time complexity of $O((m+n) \log n) = O(m \log n)$ as m is greater than n , whereas Kruskal's has $O(m \log m)$ and Dijkstra's (without heap) has the time complexity of $O(n^2)$. n is the total number of vertices which in our case is 5000 and m is the total number of edges which is 15000. Let us simply plug in the value of n and m to the actual time complexity of the algorithms and see if the numbers match our observations.

Dijkstra's (with heap) = $m \times \log n = 15000 \times \log(5000) = 55,484.55$ (fastest)

Kruskal's = $m \times \log m = 15000 \times \log(15000) = 62,641.37$

Dijkstra's (without heap) = $n \times n = 5000 \times 5000 = 25,000,000.00$ (slowest)

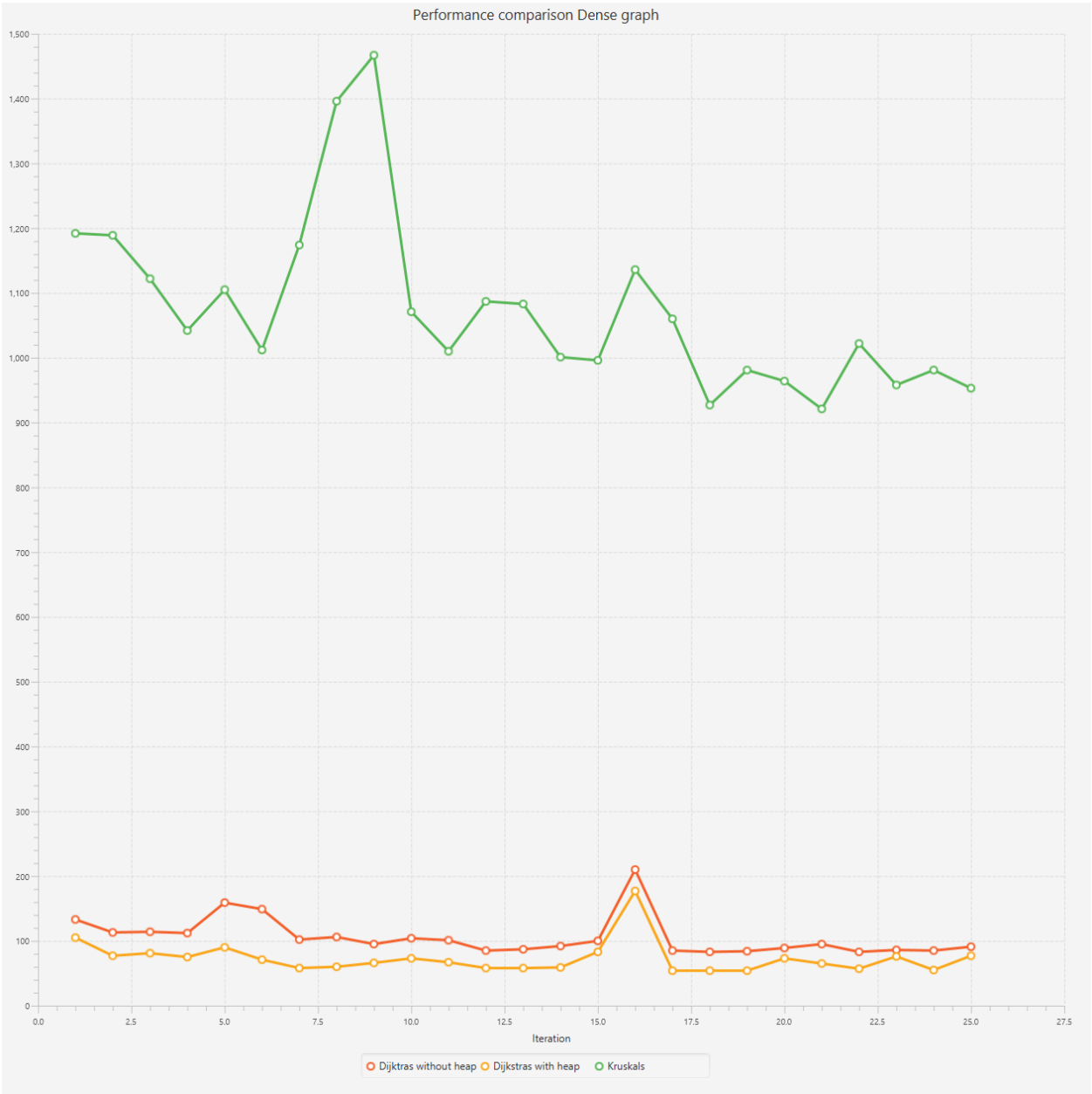
As we can observe, the order of algorithms performance matches with our observations and moreover we can see that the difference between ***Dijkstra's*** and ***Kruskal's*** is minimal which is also observed in our testing/performance data. Lastly, we see a vast difference in the performance of ***Dijkstra's (without heap)*** and as observed in the calculations the vast difference should also be expected due to a much higher time complexity.

Dense Graph

Runtime of all three algorithms in *milliseconds*:

No.	Source to Destination	Dijkstra's	Dijkstra's WITH heap	Kruskal's
1	(1129) to (3406)	133	105	1192
2	(1452) to (2206)	113	77	1189
3	(2918) to (1493)	114	81	1122
4	(605) to (3967)	112	75	1042
5	(3082) to (1821)	159	90	1105
6	(3160) to (84)	149	71	1012
7	(3187) to (578)	102	58	1174
8	(1055) to (2184)	106	60	1396
9	(2319) to (2956)	95	66	1467
10	(1858) to (2394)	104	73	1071
11	(2642) to (2123)	101	67	1010
12	(3353) to (3243)	85	58	1087
13	(4441) to (1465)	87	58	1083
14	(1466) to (3865)	92	59	1001
15	(2135) to (2137)	100	83	996
16	(3276) to (3181)	210	177	1136
17	(4384) to (2570)	86	54	1060
18	(2362) to (3103)	83	54	927
19	(4160) to (3509)	84	54	981
20	(3161) to (3424)	89	73	964
21	(4880) to (4040)	95	65	921
22	(701) to (3038)	83	57	1022
23	(2269) to (4122)	86	76	958
24	(2946) to (4735)	85	551	981
25	(1313) to (4004)	91	77	953

Performance Comparison in *milliseconds*:



Dijkstra's without heap | Dijkstra's with heap | Kruskal's

Analysis

For all graphs, including dense graphs, we have 5000 vertices which makes up the graph and the average vertex degree is about 20% for dense graphs. This implies that every vertex has about 1000 neighbors or in other words approx. 1000 edges which leads to approx. 1000 neighboring vertices. **Hence, number of total vertices(n) is 5000 and approximate number of total edges(m) is $5000 \times 1000 = 5000000/2 = 2500000$ for all dense graphs.** The reason we divide the total number of edges by 2 is because the graph is an undirected graph, every edge gets counted as 2 by default.

As we observe in the above table, the following is the pattern in which algorithms complete their execution from quickest to slowest:

$$\text{Dijkstra's (WITH heap)} < \text{Dijkstra's (WITHOUT heap)} < \text{Kruskal's}$$

Dijkstra's (with heap) being the fastest, followed by Dijkstra's (without heap) and lastly followed by Kruskal's.

This was not the expected result for dense graph as Dijkstra's with heap has a time complexity of $O((m+n) \log n) = O(m \log n)$ as m is greater than n, whereas Dijkstra's (without heap) has the time complexity of $O(n^2)$ and Kruskal's has $O(m \log m)$. As discussed above, n is the total number of vertices which in our case is 5000 and m is the total number of edges which is 2500000. Let us simply plug in the numbers (n and m) to the actual time complexity of the algorithms and see if the numbers match our observations.

$$\text{Dijkstra's (with heap)} = m \times \log n = 2500000 \times \log(5000) = 9,247,425.01$$

$$\text{Dijkstra's (without heap)} = n \times n = 5000 \times 5000 = 25,000,000.00$$

$$\text{Kruskal's} = m \times \log m = 2500000 \times \log(2500000) = 15,994,850.00$$

As we can observe, the order of algorithms performance doesn't match with our observations as after plugging values of n and m it looks like *Dijkstra's (without heap) still* should have been the slowest. But according to our testing/performance data, Kruskal's is the slowest when compared to the other two algorithms. After performing some further analysis in the code by adding timers in various steps of Kruskal's algorithm, I narrowed down that the heap construction and sorting of **2500000** edges is too expensive when speaking from time complexity perspective. Generating

edges, heap and then performing heap sort on heap of size **2500000 edges** results in Kruskal's being the slowest path finding algorithm. Out of the total execution time of Kruskal's approximately 70% of the execution time is taken by the generating edges, heap, and heapsort step. If Kruskal's entire execution time is about 1000 milliseconds, then the generating edges, heap, and heap sort step takes 700 milliseconds. To show the same, I am printing the exact amount of time taken by the generating edges, heap, and heapsort step in the console output while running/executing Kruskal's.

Conclusion: Asymptotically speaking or in terms of Big O, Dijkstra's (without heap) should be the slowest but due to such a high number of edges(m) to insert in the heap followed by heapsort, Kruskal's results in the highest execution time or in other words slowest performance. The above observation also makes sense as the only thing changed between sparse graph and dense graph is the total number of edges in graph. By the increase in number of edges from 15000 to 2500000 in dense graph, this led to an increased execution time for Kruskal's when compared to Dijkstra's.

Further Improvements

As we have identified the bottleneck for the slowest algorithm Kruskal's being the edges/heap generation and heapsort. We can generate the heap while we are randomly generating edges for our graph by directly inserting those edges into our heap and perform heapsort. This way we won't have to generate/sort the heap which running the Kruskal's algorithm. This should decrease the overall execution time of Kruskal's considerably because as we observed in our analysis the heap generation and heapsort make up 70% of execution time out of the total execution time for Kruskal's algorithm. If we eliminate edges/heap generation and heapsort step while running Kruskal's algorithm, we should see performance improvement in Kruskal's algorithm.