CSCE 629 Analysis of Algorithms

Course Project Report

- Harmanpreet Singh

Introduction

The aim of this project is to implement a network routing protocol using data structures and algorithms learned in the course to solve the MAX-BANDWIDTH-PATH problem. The problem is to find a path between a source and a destination with the max bandwidth in a network. The network can be considered as a weighted undirected graph with two of its vertices as the source and the destination. This problem can be solved using various algorithms, but the performance of each algorithm varies according to their implementation. The project's problem statement asked us to use modified Dijkstra's and Kruskal's algorithm to solve the MAX-BANDWIDTH-PATH problem.

Implementation

The implementation includes two types of random graph generations, heap structures, and three types of routing protocol algorithms. The first aspect of the problem statement is to randomly generate a sparse and a dense graph that will serve as our network to solve the maximum bandwidth path problem. The second aspect is to implement heap structures with helper subroutines like insert, maximum and delete. These heap structures are being used in algorithms to solve the path problem. Lastly, the routing algorithms modified to either use a heap structure (Dijkstra's), heap sort for edges (Kruskal), or without using a heap (Dijkstra's).

1. *Random Graph Generation*

The project statement includes the generation of two types of random undirected graphs with 5000 vertices. The first type of graph(G1) should have an average vertex degree of 6. While for the second type(G2) each vertex should be adjacent to about 20% of other vertices which are randomly chosen. If we calculate the density of these graphs based on the equation:

$$D_U(V,E) = \frac{|E|}{\text{Max}_U(V)} = \frac{|E|}{\frac{|V|\cdot(|V|-1)}{2}} = \frac{2\cdot|E|}{|V|\cdot(|V|-1)}$$

For G1, the average degree is 6 and we take the sum of all the degrees it should be 6*5000 = 30000 which is equal to 2*|E|. Thus, the density comes out to be less than 0.5 for G1 and making it a sparse graph. And for G2, if a vertex is adjacent to about 20% of other vertices, the average degree of each vertex will be 1000. Thus, the density will be greater than 0.5 and making G2 a dense graph.

The graphs are represented using the adjacency list in the Graph class which takes

total number of vertices as the input. The index of the adjacency list will give the vertex's name while the value stored at the index gives the list of edges with other vertices. The edges are represented with an Edge class that stores the start and the end node along with the edge weight which is chosen randomly between 1 to 1000.

There is a GraphGenerator class that provides methods for generating a sparse and a dense graph. For the sparse graph, firstly all the adjacent vertices are connected with a random weight. Further the edges are added abiding by the Handshake Theorem (2E = Sum of deg of vertices) to make the average degree of the vertices as 6. There is a special check to see if an edge already exists between two nodes. Similarly for the dense graph, firstly a cycle is created by connecting all the adjacent nodes. Then, each vertex is connected to other nodes with a random probability less than 20 that makes average degree for each edge to 1000.

2. *Heap Structures*

A Heap is a special complete binary tree that satisfies the heap property. For solving the max bandwidth problem we need to use the max-heap. In a Max-Heap, the key at the root node must be the greatest among the keys present at all its children. The same property must be recursively true for all sub-trees. Two different heap structures are being used in the solution. One (Maxheap) supports Dijkstra's algorithm while the other (MaxHeapKrus) supports Kruskal's. Both are implemented with support for the following operations: MAXIMUM, INSERT and DELETE and align with requirements in the problem statement.

MaxHeap class supports heap data structure to extract the fringe with maximum bandwidth in O(1) time as the maximum value will always be placed at H[0]. Heap also gives the advantage of removing the max item in O(log n). The delete method supports index-based deletion from the heap for a corresponding vertex (Discussed in Dijkstra's implementation). Another array P[5000] is being used to store position of vertex v in the heap H[5000]. Each time the positions of the vertices are swapped, P[v] is changed accordingly. Otherwise the delete operation is costly(O(n)) because we would have to traverse H[] to find the index of a particular vertex.

3. *Routing Algorithms*

The problem statement asks us to use three algorithms to solve the max-bandwidth-path problem. The input to the algorithm is a sparse/dense random graph with a source and a destination.

a. Dijkstra's Algorithm without Heap Structure

Dijkstra's algorithm was modified to find max bandwidth path based on the code taught in class. In MaxBandwidthDijks class, there are three arrays that store the parent, bandwidth, and status of each node. Constants class consists of all the

constant values that are being used repeatedly in the project to promote reusability. The status value of 0 indicates a node is unseen, 1 indicates a node is a fringe, and 2 indices a node is intree. To find the fringe with the maximum bandwidth, all vertices need to be checked with status as fringe causing the overall Dijkstra algorithm to run in O(n^2). There is tracePath method that can give us the path and the path length. The method call is commented to prevent the traceback time being added in the overall time. The implementation of Dijkstra's algorithm with to find max bandwidth is shown below:

```
    //Update vertices adjacent to source
    ArrayList<Edge> adjacentToSource = graph.getAdjacencyList()[source];
    for (Edge edge : adjacentToSource) {
        int w = edge.getAdjacentVertex(source);
        status[w] = Constants.FRINGE;
        parent[w] = source;
        bandwidth[w] = edge.getWeight();
    }

    //Dijsktra algorithm to find destination
    while (status[target] != Constants.INTREE) {
        //pick a fringe v with maximum bandwidth
        int maxBandwidth = Integer.MIN_VALUE;
        int v = -1;
        for (int i = 0; i < graph.getTotalVertices(); i++) {
            if (status[i] == Constants.FRINGE && bandwidth[i] > maxBandwidth) {
                    maxBandwidth = bandwidth[i];
                    v = i;
            }
        }
        status[v] = Constants.INTREE;

        ArrayList<Edge> vAdjacentToMax = graph.getAdjacencyList()[v];
        for (Edge edge : vAdjacentToMax) {
            int w = edge.getAdjacentVertex(v);
            if (status[w] == Constants.UNSEEN) {
                parent[w] = v;
                status[w] = Constants.FRINGE;
                bandwidth[w] = Math.min(bandwidth[v], edge.getWeight());
            } else if(status[w] == Constants.FRINGE && bandwidth[w] < Math.min(bandwidth[v], edge.getWeight())) {
                parent[w] = v;
                bandwidth[w] = Math.min(bandwidth[v], edge.getWeight());
            }
        }
    }
    // Can use this if need to trace Path
    // tracePath(parent, target);
    return bandwidth[target];
}
```

```
public static void tracePath(int[] parent, int target) {
    int i = target;
    int pathLength = 0;
    Stack<Integer> path = new Stack<>();
    while(i != -1) {
        path.add(i);
        i = parent[i];
    }
    while(!path.isEmpty()) {
        System.out.print(path.pop() + " -> ");
        pathLength++;
    }
    System.out.println();
    System.out.println("Path Length: " + pathLength);
}
```

b. Dijkstra's Algorithm without Heap Structure

In this algorithm, we modified the previous Dijkstra's algorithm to keep track of the fringes. Instead of storing the fringes in the array, we use max-heap structure to store them. All the nodes with their bandwidths that are neighbors to source are inserted into the heap. Heap gives the advantage of extracting the maximum element in O(1). Thus, to find the fringe with max bandwidth and delete it from the heap, this extraction takes O(log n) time. If an adjacent vertex is still unseen, then we insert the fringe into the heap and update the information accordingly. However, if the adjacent vertex is a fringe and its bandwidth is lesser than both the minimum bandwidth of the source and the edge weight, that vertex is deleted from the heap. The bandwidth and parent arrays for the vertex are updated and it is inserted back into the heap with the updated bandwidth. The operations for deleting and inserting into a heap take O(log n) as mentioned previously and finding the maximum takes O(1), which results in overall complexity of this algorithm to be O(m*log(n)), where m are the edges of the graph and n are the total number of vertices.

```
for (int i = 0; i < graph.getTotalVertices(); i++) {
    status[i] = Constants.UNSEEN;
}

//Starting with source
status[source] = Constants.INTREE;
parent[source] = -1;

//Update vertices adjacent to source
ArrayList<Edge> adjacentToSource = graph.getAdjacencyList()[source];
for (Edge edge : adjacentToSource) {
    int w = edge.getAdjacentVertex(source);
    status[w] = Constants.FRINGE;
    parent[w] = source;
    bandwidth[w] = edge.getWeight();
    maxHeap.insert(w, bandwidth[w]);
}

//Dijsktra algorithm using Heap
while (status[target] != Constants.INTREE) {
    //pick a fringe v with max bandwidth from the heap
    int v = maxHeap.extractMax();
    status[v] = Constants.INTREE;

    ArrayList<Edge> vAdjacentToMax = graph.getAdjacencyList()[v];
    for (Edge edge : vAdjacentToMax) {
        int w = edge.getAdjacentVertex(v);
        if (status[w] == Constants.UNSEEN) {
            parent[w] = v;
            status[w] = Constants.FRINGE;
            bandwidth[w] = Math.min(bandwidth[v], edge.getWeight());
            maxHeap.insert(w, bandwidth[w]);
        } else if(status[w] == Constants.FRINGE && bandwidth[w] < Math.min(bandwidth[v], edge.getWeight())) {
            maxHeap.delete(w);
            parent[w] = v;
            bandwidth[w] = Math.min(bandwidth[v], edge.getWeight());
            maxHeap.insert(w, bandwidth[w]);
        }
    }
}
// Can use this if need to trace Path
  tracePath(parent, target);
return bandwidth[target];
```

c. Kruskal's Algorithm with Heap Sort

We can use Kruskal's algorithm to find the maximum bandwidth path by building a maximum spanning tree and then using Breadth First Search (BFS) to find the path between the source and the target. For Kruskal's algorithm, the edges need to be sorted in non-increasing order of their bandwidth. For this sorting, the project problem asks us to use heap sort. For this, all the vertices are traversed and the edges in their adjacency list are added to heap using sortEdges method for sorting them using Heap Sort. Further, Union-Find operations are being used to determine the edges of the maximum spanning tree. These methods allow us to build a new graph that contains only the edges that pertain to the maximum spanning tree. Once the maximum spanning tree is generated, the path from source to destination is found using BFS. The returned path will be the maximum bandwidth path from source to destination. As discussed in the class, the algorithm takes O(mlogn) time to generate MST as heap sort takes O(mlogn). Further BFS takes O(m+n) time to find the path between the source and target. Thus total time taken by this algorithm is O(mlogn)

```
public class MaxBandwidthKrus {

    private static MaxHeapKrus heap;
    private static int[] rank;
    private static int[] parent;
    private static Graph mst;

    private static int[] color;
    private static int[] parentBFS;
    private static int[] bandwidth;


    private static void sortEdges(Graph graph) {
        heap = new MaxHeapKrus( maxSize: graph.getTotalDegree() + 1);
        for (int v = 0; v < graph.getTotalVertices(); v++) {
            ArrayList<Edge> edgeList = graph.getAdjacencyList()[v];
            for (Edge edge : edgeList) {
                heap.insert(edge);
            }
        }
    }

    public static int find(int vertex) {
        int v = vertex;
        while (parent[v] != v) {
            v = parent[v];
        }
        return v;
    }

    public static void union(int r1, int r2) {
        if (rank[r1] > rank[r2]) {
            parent[r2] = r1;
        } else if (rank[r1] < rank[r2]) {
            parent[r1] = r2;
        } else {
            parent[r1] = r2;
            rank[r2]++;
        }
    }

    public static int maxBandwidthPath(Graph graph, int source, int destination) {
```

```java
public static int maxBandwidthPath(Graph graph, int source, int destination) {
    // Kruskal Algorithm
    // Sort all edges in non-increasing order first

    sortEdges(graph);
    parent = new int[graph.getTotalVertices()];
    rank = new int[graph.getTotalVertices()];
    for (int i = 0; i < graph.getTotalVertices(); i++) {
        parent[i] = i;
        rank[i] = 1;
    }

    mst = new Graph(graph.getTotalVertices());
    for (int e = 0; e < graph.getTotalEdges(); e++) {
        Edge edge = heap.extractMax();
        int u = edge.getStart();
        int v = edge.getEnd();
        int r1 = find(u);
        int r2 = find(v);
        if (r1 != r2) {
            mst.addEdge(edge.getStart(), edge.getEnd(), edge.getWeight());
            union(r1, r2);
        }
    }

    // Path search using BFS
    color = new int[mst.getTotalVertices()];
    parentBFS = new int[mst.getTotalVertices()];
    bandwidth = new int[mst.getTotalVertices()];
    for (int v = 0; v < mst.getTotalVertices(); v++) {
        color[v] = Constants.WHITE;
        parentBFS[v] = -1;
        bandwidth[v] = Integer.MAX_VALUE;
    }
    color[source] = Constants.GREY;
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(source);

    while (!queue.isEmpty()) {
        int u = queue.poll();
        for (Edge edge : mst.getAdjacencyList()[u]) {
            int v = edge.getAdjacentVertex(u);
```

Test Results and Performance Analysis

The algorithms were tested on 5 sets of sparse and dense graphs with 5 different sources and destinations. The output file is zipped along with the code.

Based on the overall average running time, for sparse graph:

(*Faster) Dijkstra with Heap > Kruskal with Heap > Dijkstra without Heap (Slower)*

The results align to the time analysis discussed in the class. Dijkstra's algorithm without heap takes the most time O(n^2) because of iterating through the entire array of vertices every time to find the maximum fringe. As discussed in class, Kruskal's algorithm with the use of Union-Find operations and Dijkstra's algorithm with heap structure takes time of O(m*log(n)). This is based on the notion that the Find operation takes O(log n) and the heap insert and delete operations also takes O(log n) time. The modification in Kruskal's algorithm to sort the edges in non-increasing order to generate the maximum spanning tree and finding the path using BFS may be the reason for the little performance difference.

Sparse Graph						Dense Graph					
	Src	Dest	Dijks(s)	DijksHeap(s)	KruskHeap(s)		Src	Dest	Dijks(s)	DijksHeap(s)	KruskHeap(s)
G1(15K Edges)	2297	2903	0.06056	0.0058	0.0192	G2(2498472 Edges)	4312	2147	0.0172	0.01737	1.16782
	2777	783	0.04196	0.00503	0.01712		3367	1011	0.00796	0.00552	1.18079
	2596	855	0.00311	0.00541	0.015		3008	38	0.01226	0.01734	1.1794
	3399	2584	0.00831	0.00581	0.00983		1386	905	0.0075	0.04995	1.15229
	3839	2888	0.01235	0.00225	0.00883		1447	3219	0.02298	0.02522	1.15863
G2(15K Edges)	4994	2814	0.01599	0.00131	0.00376	G2(2499925 Edges)	4285	3863	0.0255	0.02948	1.09754
	3298	452	0.01343	0.00045	0.00368		776	2761	0.02074	0.04446	1.08487
	717	3095	0.01093	0.00104	0.00373		4668	1332	0.00943	0.03772	1.08372
	3892	3205	0.01666	0.00119	0.00367		2662	158	0.00771	0.04566	1.06871
	180	1249	0.01287	0.00092	0.0044		3214	3034	0.02214	0.0049	1.12337
G3(15K Edges)	3577	2822	0.00816	0.00127	0.00373	G3(2498755 Edges)	618	3327	0.02493	0.05894	1.22909
	2404	2035	0.01741	0.00129	0.00386		4850	3475	0.03171	0.03938	1.20385
	872	2485	0.00831	0.00069	0.00392		2170	1829	0.01393	0.04371	1.07661
	2270	2191	0.01751	0.00122	0.00381		3435	1589	0.0125	0.03466	1.07409
	4818	834	0.00258	0.0009	0.00377		1481	1481	0.02957	0.00177	1.07048
G4(15K Edges)	4026	4596	0.00876	0.00036	0.0035	G4(2498489 Edges)	2433	3985	0.02964	0.05911	1.15813
	2783	2421	0.00333	0.00061	0.00362		2715	2524	0.01652	0.0233	1.16698
	4708	1771	0.0171	0.0013	0.00377		452	1998	0.01757	0.02214	1.15088
	4301	3910	0.01157	0.00099	0.00358		4456	1222	0.00887	0.04294	1.05362
	3038	4154	0.01282	0.00035	0.00367		1249	795	0.00546	0.01608	1.03449
G5(15K Edges)	1459	1389	0.01172	0.0012	0.00379	G5(2500804 Edges)	422	2416	0.01635	0.02872	1.07396
	3013	4867	0.01592	0.0002	0.00373		2068	3138	0.01995	0.03566	1.07245
	2615	1377	0.00566	0.00053	0.0035		2608	4943	0.03009	0.00444	1.07789
	1940	3496	0.00167	0.00001	0.00371		4947	4932	0.03035	0.00229	1.07714
	230	1782	0.00612	0.00018	0.00352		2180	4718	0.03228	0.002	1.08742
Average			0.0137785	0.00208033	0.00594662				0.0189256	0.02737252	1.1161688

For dense graphs, the performance relation looks like:

(Faster) Dijkstra with Heap > Dijkstra without Heap > Kruskal with Heap (Slower)

Like the performance in sparse graphs, Dijkstra's algorithm with heap performs the best to find the maximum bandwidth path. This proves the usefulness of this algorithm in both types of graphs. On the other hand, Kruskal's algorithm obtains the worst performance in dense graphs. In our implementation, we used HeapSort to sort the edges and in dense graphs the number of edges can increase manifolds. There were around 2.5M edges for each dense graph generated and sorting these many values can take a long time thus not being able to abide by the theoretical complexity of O(m*log(n)).

Conclusion and Improvement Scope

Implementation and analysis show that different algorithms perform differently based on the input and the data structures being used in the implementation. Three different algorithms were successfully implemented and tested to solve the network optimization problem to find the maximum bandwidth path. The results demonstrated that Dijkstra's with heap algorithm had the least execution time and the best performance for both sparse and dense graphs.

Kruskal's algorithm performs worst with dense graphs and HeapSort is the bottleneck. HeapSort may not be the best technique to sort the edges and can be replaced with a better sorting algorithm such as merge sort. Merge sort is slightly faster than heap sort for larger sets, but it requires twice the memory of heap sort because of the second array. There are many efficient sorting algorithms for streaming input data that can be used to store and sort the fringes. As we are aware that for dense graphs adjacency matrix gives more advantage than adjacency list, it may help to improve the performance. Java supports priority queues which are based on the priority heap that may help to improve the HeapSort efficiency. Lastly, as discussed in the class to use 2-3 trees to store the fringes can be used with some modifications to efficiently perform the delete operation.