CSE 629 Analysis of Algorithms Course project Maximum Bandwidth Path Problem

Sankalp Chapalgaonkar SANKALPCHAP1@TAMU.EDU
Department of Computer Science & Engineering
Texas A&M University
College Station, Texas

Tutors: Dr. Jianer Chen, Vaibhav Bajaj

Abstract

Network optimization is a significant topic of current computer science and computer engineering study. In this course project, I have created a network routing protocol utilizing the data structures and algorithms learned in class CSE629 Analysis of Algorithms. This allowed me to transform my academic understanding into a real-world practical computer application. This module features random sparse and dense graph generation, and implementation of Dijkstra's and Kruskal's algorithms to solve the maximum bandwidth problem. In the end, I have presented a thorough analysis of the performances of these algorithms on dense and sparse graphs.

Keywords: Maximum Bandwidth, Dijkstra, Kruskal, Heap, Graph Generation

1. Introduction

The goal of this project is to help us realize that translating algorithmic notions at a higher degree of abstraction into tangible implementations in a specific programming language is not always straightforward. Implementations usually urge you to focus on extra details of the techniques, which can result in significantly more knowledge. We concentrate on the Maximum Bandwidth Path problem, which is considered a significant network optimization challenge. Various techniques may be used to find the path with the best potential bandwidth between a given source and destination, but the performance is mainly dependent on the actual implementation of an algorithm and the data structures utilized beneath. First, we build two random weighted undirected graphs, one sparse and one dense with a large degree of vertices. We maintain this generation unpredictable enough to adequately assess the performance of methods. Furthermore, we employ two well-known algorithms - Dijkstra's and Kruskal's - to determine the maximum bandwidth path between the source and destination. However, we conduct a comprehensive examination of the running time of these algorithms after modifying them to use the heap sort rather than not utilizing one. In the later parts of this report, I go over the specifics of random graph generation, implementation of Dijkstra's without heap algorithm, Dijkstra's with heap algorithm, and Kruskal's algorithm. Finally, I present the results in tabular format as well as a thorough performance comparison of these three routing algorithms.

2. Random Graph Generation

We are expected to construct two graphs G_1 and G_2 (each with 5000 vertices) that are sparse and dense in terms of vertex degree. G_1 must have an average vertex degree of 6, and each vertex in G_2 must be adjacent to approximately 20% of the other randomly picked vertices. I created the following models to represent various graph attributes.

1. Vertex - Used to store the attributes of a vertex in the graph
   (a) vertexId (int) - denotes the current vertex number
   (b) edgeWeight (int) - the edge weight between this vertex and the connected vertex
   (c) degree (int) - used to keep track of the degree of each vertex
   (d) next (Vertex) - denotes the next vertex in the linked list
2. Edge - Used to store the attributes of edges in the graph and has vertex1, vertex2, and edgeWeight as its parameters
3. Graph - This is a container to store the attributes of the graph such as graph type (Sparse or Dense), number of vertices, and Vertex[ ] which stores the head of the adjacency list (implemented in the form of LinkedList) of i^{th} vertex at i^{th} index in the array.

I build these random graphs in two parts. Firstly, each vertex is connected to its next vertex thus forming a cycle to ensure that graph has only one strongly connected component. For example, V_0 is connected to V_1 and V_{4999}, V_2 is connected to V_1 and V_3 and so on. After this step, each vertex will have a degree of two which results in a total edge count of 5000. In the second part, we randomly pick two vertices and add an edge if there isn't already existing one, and increase the degree of both vertices. We keep repeating this process until the average degree of the vertex is reached the target degree which is 6 in the case of the sparse graph and 1000 for the dense one. This results in the generation of random graphs having approximately 15000 edges for the sparse graph and 2.5 million for the dense graph. I record the graph generation time and print the average vertex degree (as calculated below) for the sparse graph and the average percentage of adjacent vertices connected in for the dense graph.

$$Average\ Degree = 2 * \frac{Number\ of\ edges}{Number\ of\ Vertices}$$

Because both graphs are weighted, when I create an edge between any two vertices, I apply a random edge weight (generated by the *Random* library from the *java.util* package) with a weight limit of 20000. I tried numerous settings for this weight limit option, ranging from 10 to 50,000. Because the *Random* library was producing a random integer (not a float), the number of buckets for the dense network was relatively smaller for the lower weight limit, resulting in saturating the maximum bandwidth to the weight limit (a possible solution to this was to generate the weights as floats, ensuring enough buckets in order to prevent saturation). When I upped the weight limit to 20000, the maximum bandwidth obtained from the algorithms for the dense graphs was about 19000.

3. Heap Structure

We are required to implement the maximum heap subroutine, which performs fundamental heap operations such as inserting, deleting, and fetching the maximum element. We are supposed to use this subroutine in Dijkstra's algorithm to efficiently discover the fringe with the greatest bandwidth. As we are supposed to find the vertex information for which the edge has the maximum bandwidth, the actual implementation becomes a bit complex. To keep track of vertexId, actual bandwidth of fringes, and indices throughout the heap functionalities, I create three arrays as follows

1. heapArray - To store the actual bandwidth of the fringe (D array as per problem statement)
2. vertexArray - Retrieve the vertex Id from the heap-referenced index (H array as per problem description)
3. positionArray - Maps vertexId's to the index. This array is used to get the index referenced within the heap from Vertex Number. Basically opposite of the vertexArray (which maps from index to Vertex Id). This is the P array as per the project description.

1. **init** - Constructor for the VertexHeap class, which takes in the maxSize parameter and initializes all three arrays.
2. **swapNodes** - This method is used to swap the position of two vertices, which is used during the heapify process. As we are referencing the vertices are linked to all the arrays, and they need to be updated. The following code implements the functionality explained above.

```
private static void swapNodes(int pos1, int pos2) {
    // first modify the position array
    positionArray[vertexArray[pos1]] = pos2;
    positionArray[vertexArray[pos2]] = pos1;

    // then we actually switch the vertex Array
    int tempVertices = vertexArray[pos1];
    vertexArray[pos1] = vertexArray[pos2];
    vertexArray[pos2] = tempVertices;

    // Finally switch the bandwidth values in the Heap array
    int dummy = heapArray[pos1];
    heapArray[pos1] = heapArray[pos2];
    heapArray[pos2] = dummy;
}
```

Figure 1: Swapping in Vertex Heap

3. **maxHeapify** - This is a recursive function to adjust the position of the vertices if it does not follow the heap rules. For a heap of n elements, heapify takes $O(logn)$ time

4. **insert** - We pass in the vertex Id and the corresponding bandwidth as arguments, insert this bandwidth in the heap array at the last position and then perform heapify on this element.
5. **popMax** - This function takes the index as an input, deletes that element from the heap, then returns. I made it general enough to test exhaustively, despite the fact that in Dijkstra's approach, we are always required to fetch the element at index 0. In order to maintain the right connection, we must additionally alter all three arrays. In the end, we reduce the size of the heap by one.
6. **isLeaf** - returns if a vertex is a leaf node or not according to the following logic

$$index \in (\frac{size}{2}, size]$$

4. Problem Statement

Given a source node s and a destination node t in a network G, in which each edge E is associated with a certain bandwidth $bw(E)$, construct a path from s to t in G whose bandwidth is maximized (the bandwidth of a path is equal to the minimum edge-bandwidth over all edges in the path).

$$BW(s,t) = \forall\ e \in E_{st}\ min\{bw(e)\}$$

where E_{st} are the edges in the path from s to t.

5. Routing Algorithms

This is the most important aspect of the project. After generating the random graphs, we now will apply various routing algorithms to discover the maximum bandwidth path between a given source node s and destination node t. We must implement three routing algorithms: Dijkstra's without the heap, Dijkstra's with the heap structure (which was explained in the previous section) to efficiently sort the fringes, and Kruskal's algorithm, which involves generating a maximum spanning tree (MST) and then finding the path by applying Depth First Search on the source vertex. We then evaluate their performance based on the running time of these algorithms. The next subsections will go into the implementation details of these algorithms. I have maintained a separate folder in the repository for creating three different classes for each algorithm, which implements the apply method which takes the Graph, source, and destination as algorithms.

5.1 Dijkstra's Algorithm Without Heap

Dijkstra's algorithm is usually used to solve the shortest path problem in a graph, which can be modified to solve the maximum bandwidth problem as well. It is a greedy approach to finding the fringe with maximum bandwidth and is guaranteed to find the optimal maximum bandwidth path for the graphs having all positive edges. The implementation is highly derived from the pseudo-code provided in the class. To keep track of the multiple stages during algorithm processing, we employ three arrays: parent, status, and bw. I built the *Status* enum, which comprises three vertex states: UNSEEN, FRINGE, and INTREE. We

begin by setting the status array to UNSEEN and filling the fringes from the source vertex. We iterate till the fringes exists and then choose the fringe with the most bandwidth, change its status to INTREE, and update its bandwidth in the bw array. In the end, the bandwidth array will contain the maximum bandwidth for all the vertices, and following the parent array, we can find the path which gives us the maximum bandwidth. Then we just print the node which eventually results in the reverse route from source to destination. Because we are keeping track of the fringes in an array, we will need to cycle over the whole array to determine the largest fringe, forcing the total Dijkstra algorithm to execute in $O(n^2)$ time. This can be improved by using the heap sort which is explained in the next section

5.2 Dijkstra's Algorithm With Heap

In this modified version of Dijkstra's algorithm, instead of traversing throughout the fringes, we use VertexHeap (defined above) to maintain the fringe with the highest bandwidth. We first initialize the VertexHeap with a size equal to the number of nodes in the graph by calling the $init()$ method. Then we start from the source node and insert its neighbors to the heap, thus turning their status from UNSEEN to FRINGE. We then fetch the fringe with maximum vertex (in constant time) by calling the $popMax()$ method at index 0 (remember the heap will store the fringe with maximum bandwidth at the top). Once retrieved, we remove it from the fringes array and update its status from FRINGE to INTREE. Then we repeat this process from the maximum fringe, looking at its neighbors and so on. The actual implementation is as follows

```
// Initialize the vertex heap
VertexHeap.init(n);
// add nodes from src
Vertex[] vertices = graph.getVertices();
Vertex temp = vertices[source];
while (temp != null) {
    status[temp.getVertexId()] = FRINGE;
    dad[temp.getVertexId()] = source;
    bw[temp.getVertexId()] = temp.getEdgeWeight();
    VertexHeap.insert(temp.getVertexId(), bw[temp.getVertexId()]);
    temp = temp.getNext();
}

// add nodes for unseen in heap and update status, dad and bw for unseen and fringe
while (VertexHeap.getSize() != 0) {
    int maxIndex = VertexHeap.popMax( index: 0);
    status[maxIndex] = INTREE;
    Vertex node = vertices[maxIndex];
    while (node != null) {
        if (status[node.getVertexId()] == UNSEEN) {
            status[node.getVertexId()] = FRINGE;
            dad[node.getVertexId()] = maxIndex;
            bw[node.getVertexId()] = Math.min(bw[maxIndex], node.getEdgeWeight());
            VertexHeap.insert(node.getVertexId(), bw[node.getVertexId()]);
        } else if (status[node.getVertexId()] == FRINGE
                && bw[node.getVertexId()] < Math.min(bw[maxIndex], node.getEdgeWeight())) {
            VertexHeap.delete(node.getVertexId());
            dad[node.getVertexId()] = maxIndex;
            bw[node.getVertexId()] = Math.min(bw[maxIndex], node.getEdgeWeight());
            VertexHeap.insert(node.getVertexId(), bw[node.getVertexId()]);
        }
        node = node.getNext();
    }
}
```

Figure 2: Dijkstra's Algorithm using Heap

The insertion and deletion in the Max-heap will take $O(logn)$ time as we need to perform the heapify to adjust the heap after updating the heap array. It always maintains the vertex with maximum bandwidth at the top hence the retrieval is done in constant time. Hence

total time complexity after using heap reduces to $O((m+n)logn)$ which can be written as $O(mlogn)$ as $n << m$. This is a major improvement in the performance of Dijkstra's algorithm and the results of this simulation (which will be presented later in this report) confirm the same.

5.3 Kruskal's Algorithm

A maximum spanning tree is a spanning tree of a weighted graph having maximum weight. Kruskal's algorithm takes a bit different approach to solving the maximum bandwidth problem. Irrespective of the source and destination nodes, we first generate the maximum spanning tree and then apply the DFS from the source to find the maximum bandwidth path. As proved in the class, there exists only one path between any two vertices in the maximum spanning tree, and that path will be the maximum bandwidth path.

I have implemented Kruskal's algorithm using the pseudo-code presented in class with a few modifications in order to take advantage of the heap sort function. We first create a list of all the edges of the graph and use heapSort to sort the edges in non-increasing order of their edge weights (This would take $O(mlogm)$ time considering the graph has m edges). Additionally, I have implemented the UnionFind class in order to have the ability to use the operations MakeSet, Find, Union, maxHeapify, and sortEdges. This class keeps track of the parent and ranking for each vertex allowing us to use an iterative approach for the Find operation. We use the path compression as shown in Fig. 3 as well which benefits the future searches for the vertices which fall under the path of a vertex (to root) on which the find operation is called.

```
public int find(int node) {
    if (parent[node] != node) {
        // implementing the path compression to optimize the future searches
        parent[node] = find(parent[node]);
    }
    return parent[node];
}
```

Figure 3: Find operation with path compression

Once the random graph is generated, we can generate the MST only once and use it for all 5 source-destination pairs. Hence I have implemented the $generateMST()$ function to build the MST while the graph is being created, instead of creating it while the Kruskal algorithm is called on the 5 different source-destination pairs. This provides Kruskal's algorithm an advantage since we only need to compute the MST once and then execute the DFS (which takes much less time than calculating the MST) on any number of s-t pairings. However, with Dijkstra's algorithm, when the source node changes, we must recalculate the path, increasing the total complexity when we consider finding maximum bandwidth paths between multiple vertex pairs.

6. Testing and Results

Sparse Graph Results (average degree of 6)							
Graph attributes			**Running Time (in ms)**				
Graph No.	**Source**	**Destination**	**Graph Generation**	**Dijkstra's without heap**	**Dijkstra's using heap**	**MST Creation**	**Kruskal**
G1	911	2388	14.006833	209.082625	25.931584	42.689958	41.441959
	112	2811		103.130125	10.205167		2.730584
	459	126		119.442	13.164958		4.306792
	3823	3523		84.995542	10.295417		9.359459
	2461	1932		79.390167	8.516		2.831791
G2	2131	882	5.475458	149.247041	1.394416	17.489416	2.12025
	1177	4001		64.285208	1.027291		0.620125
	2169	334		74.307042	1.376291		0.224208
	4694	4913		59.853042	0.8575		0.359125
	3881	1655		69.236917	1.582666		0.993708
G3	2048	4647	4.796541	83.870958	1.685417	7.99225	1.271042
	1768	1455		70.111416	2.308417		0.684125
	1874	2653		69.938125	1.528875		0.555959
	4145	891		65.587833	1.989042		0.111666
	3762	3079		68.974917	1.967958		0.577708
G4	4318	165	4.504583	76.377583	1.245625	5.718625	1.191
	2665	3859		72.854084	2.920083		0.662958
	868	4414		71.274125	1.128416		0.386
	2231	1571		70.428417	1.673166		0.897666
	1684	3308		64.301916	1.301125		0.35525
G5	1182	851	4.852333	94.133375	2.229292	6.532083	1.180834
	4562	119		70.341667	1.669042		0.19875
	4145	3923		58.609084	0.723875		0.345958
	2622	18		70.477291	1.188583		0.190125
	3266	3908		77.764625	1.456958		0.600958
Average Time	**-**	**-**	**6.7271496**	**83.920605**	**3.97468656**	**16.0844664**	**2.96792**

Dense Graph Results (each vertex connected to 20% of other vertices)							
Graph attributes			**Running Time (in ms)**				
Graph No.	**Source**	**Destination**	**Graph Generation**	**Dijkstra's without heap**	**Dijkstra's using heap**	**MST Creation**	**Kruskal**
G1	181	2465	3381.519208	313.799375	39.119625	3058.344667	2.414041
	684	1567		278.314916	59.214167		0.870125
	773	647		256.152084	32.678		1.351166
	518	212		287.914917	33.539125		3.363708
	206	4839		239.420333	39.593334		0.597666
G2	2449	1058	4331.674125	302.156958	29.833042	3175.565542	1.217584
	4919	2902		259.181417	32.856		0.874958
	4215	1626		262.931208	30.112833		1.614542
	4803	1007		280.183	32.420458		2.472583
	1151	2834		206.11225	32.981667		1.350125
G3	997	433	5149.657416	381.704625	41.381708	3182.002542	1.120583
	112	1490		234.315958	40.062792		0.490667
	1287	724		325.974375	40.357959		1.348042
	1103	3202		216.923959	38.518958		0.351334
	4307	2741		249.30775	46.124083		0.494125
G4	4130	3724	5147.068291	387.871167	47.939625	3060.019333	1.886334
	1025	32		278.079875	43.63675		1.436542
	467	2424		243.071542	42.352209		1.347959
	4372	3232		295.298791	39.8735		1.204375
	1552	3342		244.6405	37.957458		0.507417
G5	856	3267	5031.204709	384.300542	40.429125	2894.01675	1.344375
	1359	3068		247.873625	44.324792		0.62825
	962	1700		251.779167	41.0895		0.468125
	4726	4204		227.524166	46.939458		0.218625
	1600	972		252.73775	43.645		0.531625
Average Time	**-**	**-**	**4608.22475**	**276.30281**	**43.285575**	**3073.989767**	**1.18019504**

I build five pairs of sparse and dense graphs, for which I randomly select 5 source-destination pairs and use all three algorithms to generate the maximum bandwidth path, to evaluate the randomness and performance of the graphs. Above are the detailed findings for both dense and sparse graphs. As previously stated, I isolate the MST-generating element of Kruskal's algorithm from the real process and integrate it with the graph creation. The time taken to generate this MST is shown in the table above. As a result, while comparing Kruskal's performance, we must include the MST-generation time as well as the actual algorithm running time (Depth First Search on the generated MST). In the folder submitted, I have attached the console_log file which contains the actual results from one of the iterations of my testing, a snapshot of which can be seen in Fig. 4. In the above table, I have also mentioned the time required to generate these random graphs (as explained in the 2^{nd} section).

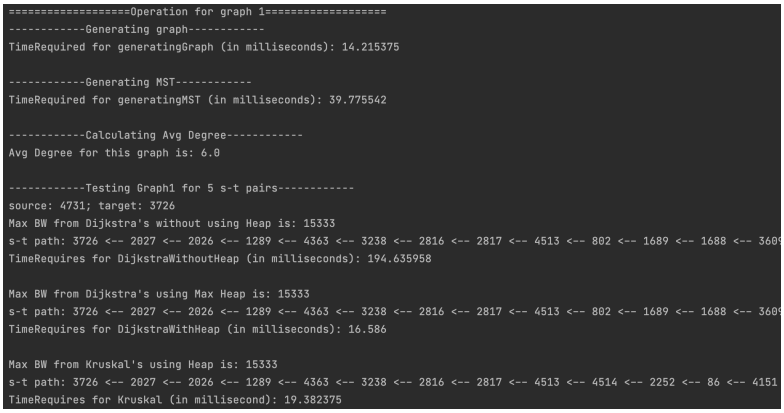

Figure 4: Sample Console Logs

7. Performance Analysis and Conclusion

We know that the running time complexities for three algorithms are as follows

Algorithm	**Time Complexity**
Dijkstra's Without Heap	$O(n^2)$
Dijkstra's with Heap	$O(mlogn)$
Maximum Spanning Tree Build Time	$O(mlogn)$
DFS	$O(m+n)$

Based on the results in the preceding section, it is clear that for the sparse graph, Dijkstra's Algorithm with Maximum Heap outperforms Kruskal's algorithm, while Dijkstra's variant without heap performs the slowest among all three. Dijkstra's sans heap is the slowest in the case of a sparse graph, with a time complexity of $O(n^2)$, since we must traverse the whole fringes array to identify the fringe with the greatest bandwidth. As demonstrated in class, Dijkstra's method with a heap structure and Kruskal's approach with Union-Find operations both have an overall complexity time of $O(mlogn)$. This is due to the fact that

in the Union-Find procedure, the find and heapify operations require $O(logn)$ time whereas insertion and deletion take constant time (as we always insert at the last position or delete the index and swap with the last element in the heap). In addition to Union-Find, we have introduced the Maximum Heap to our version of Kruskal's algorithm for sorting the edges while creating the maximum spanning tree. After generating the MST, we run the DFS from the source to determine the maximum bandwidth path to a particular destination. As a result, even though the time complexity is theoretically $O(mlogn)$, the actual running time for Kruskal and Dijkstra's method will differ due to the constants in the Big-O notation. Although there may be a number of insertion and deletion operations, the time it takes to sort all edges in the graph and then execute DFS on all edges increases the constant value of total time complexity, which explains why it performs slower than Dijkstra's heap implementation.

In terms of algorithm performance, we notice a somewhat different pattern for the dense graph. Dijkstra's heap method, like for the sparse graph, is the quickest of the three. This highlights the algorithm's utility in both types of graphs and its ability to maintain an overall complexity time of $O(mlogn)$ despite the fact that the number of edges (about 2.5 million) is far more than the number of vertices. A fascinating result is obtained in the case of Kruskal's running time complexity. For dense graphs, Kruskal's algorithm runs 12 times slower (average time 3.074 s) as compared to Dijkstra's without a heap (average time 276 ms). This was to be expected given the dense graphs' enormous number of edges. We had to utilize HeapSort in our implementation to sort through the edges, which is expensive on a dense graph and depends on the number of edges. As a result, sorting over two million edges becomes time-consuming, which explains Kruskal's algorithm's poor performance and a probable fault in the implementation that may be corrected.

8. Future Improvements

According to our performance investigation, Kruskal's algorithm performs the poorest in dense graphs, and HeapSort accounts for the majority of the time consumption. This component of the implementation might be enhanced using a different sorting mechanism. One of the biggest concerns is Kruskal's reliance on the number of edges, which means that the more edges there are, the slower this implementation runs. If another sorting approach could sort over the edges faster, the implementation could perform better. The amount of improvement that might be made is debatable, but even decreasing it to be equivalent to Dijkstra's different implementations would be advantageous.

Overall, we were able to successfully design three alternative strategies for addressing the network optimization problem of calculating the maximum bandwidth path. As a result, we were able to assess the performance of each of these algorithms on sparse and dense networks. According to the findings, Dijkstra's method with a heap took the least amount of time to execute and hence performed best for both types of graphs. The next section goes into great detail about the code structure and how to run it.

9. Codebase and Submission

The code base is self-explanatory and divided into sub-directories such as models, algorithms, and graphs in order to have a clear understanding. I have submitted the zip folder which contains the code packages and the project report. In order to test, you can run the main function in Main.java, but one must have the setup for JDK 11 (I used the Intellij v2021.1.3 for development purposes). The implementation of this network was created using Java 11 and has the following critical files

1. **Main.java:** Calls the main functions such as graph generation, applying various algorithms, and running time calculations.
2. **GraphGenerator.java:** Contains graph creation functions such as $checkAdjList$ (to verify if a vertex exists in another vertex's adjacency list), $generateConnectedGraph$, $generateMST$, and $getRandomSourceAndTarget$ which generates the random source-destination pairs.
3. **VertexHeap.java:** Implementation of maximum heap used in the Dijkstra's algorithm
4. **DijkstraWithoutHeap.java:** Implementation of Dijkstra's Algorithms without using MaxHeap
5. **DijkstraWithHeap.java:** Implementation of Dijkstra's Algorithms with using MaxHeap
6. **Kruskal.java:** Implementation of Kruskal's algorithm, especially the DFS function
7. **UnionFind.java:** Data Structure to implement Union-Find used in Kruskal's Algo

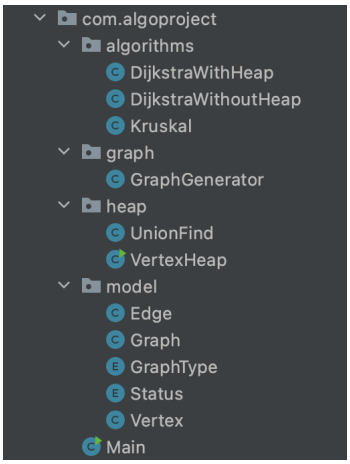

Figure 5: Code Structure