Contents

Introduction and Problem

This project motivates us to solve the Maximum Bandwidth Problem using three different algorithms. The problem is defined as follows:

Let the given graph be defined as G(V, E), where V is the set of vertices and E is the set of edges. For every edge e ∈E we define w_e as the weight of that edge. Using this we can now define the widest path problem as finding the path P in the graph G(V, E) between vertex s and vertex t (where s, t ∈V) such that weight W: min(w_x : x ∈E & x ∈P) is maximum. The value W is also referred sometimes as the bandwidth of the path. Hence the path will also have the maximum bandwidth in the graph.

For example, consider Figure 1. The figure shows an undirected weighted graph that contains 5 vertices connected by 7 edges. The edges have weights ranging from 1 to 8. Let us assume we are interested in finding a path from vertex 0 to vertex 1. Even though there is a path 0 -> 1 with bandwidth 3, we would take the path 0 -> 3 -> 1 according to our definition from before. The bandwidth of the path is the minimum weighted edge along the path.

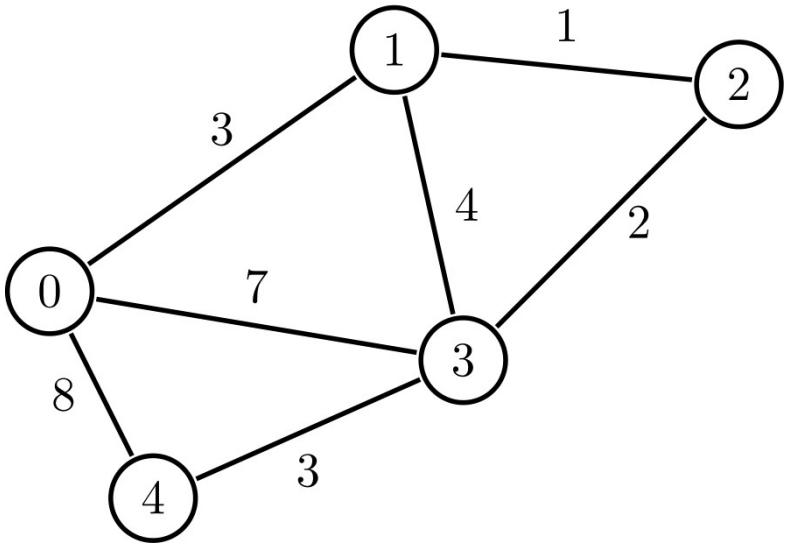

Figure 1

Along with its many applications in network routing, the max bandwidth problem is used to determine the winner of a multiway election using Schulze method. It also finds applications in digital compositing, metabolic pathway analysis and maximum flows.

Algorithms

We approach this problem with two algorithms; the modified Dijkstra's algorithm and the modified Kruskal's algorithm. The Dijkstra's algorithm is implemented in two variations; one that makes no use of a data structure to find the best fringe, and another that uses a max heap to find the best fringe which improves the time complexity of the algorithm.

DIJKSTRA'S ALGORITHM

One common approach to solving the max bandwidth problem is to make use of the modified Dijkstra's algorithm. The Dijkstra's algorithm is used to find the shortest path between vertices by computing a route of edges that have the smallest sum of edge weights in the graph. Instead of finding the least possible sum of the edge weights, we can modify the algorithm to solve the problem stated at the beginning of this document. This is done by picking the fringe with the maximum bandwidth and marking it at every iteration. The maximum bandwidth along any path is calculated as the lesser of the maximum bandwidth of the previous node along the path and the edge connecting the previous node to the node in question.

The two variations of the Dijkstra's algorithm differ in the way the best fringe is chosen. In the first variation we traverse through all the fringes and pick the fringe with the largest bandwidth. A more optimized way of getting the best fringe is to make use of a max heap. The max heap always stores the data element with the highest value at the first index. Therefore, the operation of retrieving the best fringe can be completed in constant time whereas the same operation would take linear time if we traverse through all the fringes looking for the best fringe.

Dijkstra's algorithm without Heap

1) for (v=1; v<=n; v++) status[v]=unseen
2) status[s] = intree
3) for (each edge [s,v])
    a. status[v] = fringe
    b. wt[v] = edge-weight(s,v)
    c. dad[v] = s
4) while (there are fringes)
    a. v = //pick the best fringe
    b. status[v] = intree
    c. for (each edge[v,w])
        i. if(status[w]==unseen)
            1. status[w]=fringe
            2. dad[w] = v
            3. wt[w] = min(wt[v], edge-weight(v,w))
        ii. else if(status[w]==fringe and wt[w] < min(wt[v],edge-weight(v,w))
            1. dad[w] = v
            2. wt[w] = min(wt[v], edge-weight(v,w)
5) output(dad[],wt[])

Dijkstra's algorithm with Heap

1) for (v=1; v<=n; v++) status[v]=unseen
2) status[s] = intree
3) for (each edge [s,v])
    a. status[v] = fringe
    b. wt[v] = edge-weight(s,v)
    c. dad[v] = s
    d. Heap.Insert(v,wt[v])
4) while (there are fringes)
    a. v = Heap.Max()
    b. status[v] = intree
    c. Heap.Delete(v)
    d. for (each edge[v,w])
        i. if(status[w]==unseen)
            1. status[w]=fringe
            2. dad[w] = v
            3. wt[w] = min(wt[v], edge-weight(v,w))
            4. Heap.Insert(w,wt[w])
        ii. else if(status[w]==fringe and wt[w] < min(wt[v],edge-weight(v,w))
            1. dad[w] = v
            2. wt[w] = min(wt[v], edge-weight(v,w)
            3. Heap.Delete(w)
            4. Heap.Insert(w,wt[w])
5) output(dad[],wt[])

KRUSKAL'S ALGORITHM

Kruskal's algorithm is popularly associated to build spanning trees by finding an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. It makes sure to get the subset of edges that build a tree including all vertices. The total weight of the tree is either minimized or maximized depending on the type of spanning tree being built.

To solve the max bandwidth problem, we use the Kruskal's algorithm to build a max spanning tree from the graph in question. The path connecting any two vertices in the max spanning tree is the max bandwidth path between those vertices in the original graph. We also modify the algorithm to make use of the Makeset and Find-Union Disjoint Structures to improve the time complexity of the algorithm.

Find(v)

1) w=v
2) s <- None //flush stack s
3) while(dad[w]!=None)
    a. s <- w
    b. w=dad[w]
4) while(s!=0)
    a. v <- s
    b. dad[v] = w
5) return w

Union(r1,r2)

1) if(rank[r1]>rank[r2])
    a. dad[r2]=r1
2) else if(rank[r2]>rank[r1])
    a. dad[r1]=r2
3) else
    a. dad[r2]=r1
    b. rank[r1]++

Makeset() //make a single node tree

1) dad[v] = 0
2) rank[v] =0

Kruskal's algorithm

1) HeapSort(edges) //increasing order of edge weights
2) T=None
3) for (v=0; v<n; v++) Makeset(v)
4) for (i=1; i<=m; i++)
    a. $e_i=[u_i,v_i]$
    b. r1=Find(u_i); r2 =Find(v_i)
    c. if (r1!=r2)
        i. Union(r1,r2)
        ii. T=T.append(e_i)
5) return T

Complexity Analysis

In the first variation of Dijkstra's algorithm where we don't use the heap, finding the best fringe can take at most V time. This means that the running time of the algorithm is bound by the running time of the while loop which is; **$O(V^2)$**. The more optimized variation, which makes use of the heap structure, takes lesser time. Retrieval of the best fringe takes O(1) time and the Insert() and Delete() operations take O(logV) time. Since the while loop iterates at most 2*E times, the time complexity of the algorithm is **O(E*logV)**.

Kruskal's algorithm depends on the sorting technique used to sort the edges. In this project, I've sorted the edges using HeapSort which has a complexity of O(nlogn). Even though the loop in the algorithm runs E times, the Find-Union operations reduce the running time, the algorithm is bound by the sorting

algorithm. Therefore, the complexity of the whole algorithm is **O(ElogE)**. It must be noted that once the tree is constructed, to get the max bandwidth path from a source to destination, we must perform a DFS on the tree. This DFS only takes O(E) time.

Implementation

The project was implemented on python 3 and run on a system with a 7th Gen Intel CORE i5 processor with 8GB RAM.

GRAPH GENERATION

Two types of graphs, each with 5000 vertices, were generated for this project; a dense graph and a sparse graph. The sparse graph has vertices where each vertex has a degree of 6 and the dense graph has one vertex connected to 20% of the other vertices in a graph. To generate the graphs, we first start creating a cycle by connecting a vertex to it the vertex that follows it in ascending order i.e. vertex 1 is connected to vertex 2, vertex 2 is connected to vertex 3 and so on. Finally, the last vertex (vertex 4999) is connected to the first vertex again creating a cycle. This ensures that all the nodes are connected.

Now, the remaining edges are filled in by randomly picking two vertices in the graph. We also assign edge weights randomly that range between 1 and 1000. We maintain a list to keep track of the degrees of the vertices. Whenever an edge is added in between two vertices the corresponding degree in the list is incremented for both the vertices. At the end we use this degree list to check if the average degree meets our requirement.

Even though these graphs are stored as adjacency lists, an adjacency matrix is generated at the time of graph generation. This adjacency matrix is used to keep track of edges between vertices. Since we are assigning edges randomly, there may be cases where the random generation of vertices produce a vertex numbers that already have an edge connecting them. To prevent another edge from connecting the vertices and creating parallel edges, we can look up the adjacency matrix in constant time to see if there is an edge already connecting the vertices.

HEAP

To implement the algorithms, we maintain a max heap in the form of a list, H. We also maintain two other lists; D and index. The list D stores the weights of the fringes in H such that D[H[i]] gives the weight of the fringe at H[i]. The index list stores the index of a vertex in the heap such that index[vertex] gives the position of the vertex in the heap. The heap supports three operations; Max(), Insert() and Delete(). The Max() operation returns the fringe at index 1 which is also the fringe with the maximum bandwidth. The Insert() operation appends a vertex to the heap and percolates the vertex up the heap to its correct position. The Delete() operation makes use of the index list to a find a vertex in the heap and replaces it with the last element in the heap. This element is then either percolated up or down to its correct position. While percolating the elements up or down we use the edge weights in D to compare one fringe to another.

The Max() operation takes constant time whereas Insert() and Delete() take O(logn) time. The reason for Delete() taking O(logn) instead of linear time is because we are able to get the vertex position in constant time with the help of the index list.

TESTING

The algorithms are tested on the both the dense and the sparse graph. In each iteration on the graphs a random source-destination vertex pair is picked, and the maximum bandwidth of the max bandwidth path is computed by all three algorithms and checked to see if it is the same (as it should be). 5 such iterations are performed on each graph and five pairs of graphs are generated.

To measure the performance, we calculate the time immediately before and after applying the algorithms. The multiple iterations make sure that we have a more accurate average time compared to individual results which may be skewed due to multiprocessing of other processes.

Results and Observations

		Sparse					Dense		
		No Heap (s)	Heap (s)	Kruskal's (s)			No Heap (s)	Heap (s)	Kruskal's (s)
Graph0	Iteration0	2.51781	0.25318	0.389979	Graph0	Iteration0	7.373052	5.539077	68.44319
	Iteration1	2.711865	0.233754	0.38381		Iteration1	7.202375	2.998936	60.84813
	Iteration2	2.540949	0.260474	0.258222		Iteration2	7.482151	5.021971	60.6465
	Iteration3	2.796548	0.23695	0.278401		Iteration3	7.254838	4.925462	60.67748
	Iteration4	2.661421	0.199561	0.288576		Iteration4	7.489006	5.131309	61.04619
Graph1	Iteration0	2.525176	0.165784	0.358995	Graph1	Iteration0	8.363057	5.243369	73.84622
	Iteration1	2.716782	0.259711	0.27281		Iteration1	7.692697	5.008878	59.6138
	Iteration2	2.431287	0.179585	0.249718		Iteration2	7.336115	4.751375	55.4566
	Iteration3	2.498971	0.213574	0.226718		Iteration3	6.961862	4.647384	55.79803
	Iteration4	2.477461	0.249425	0.339237		Iteration4	7.068766	4.970826	56.00555
Graph2	Iteration0	2.22471	0.226699	0.430081	Graph2	Iteration0	6.991057	5.379132	68.55565
	Iteration1	2.514999	0.243922	0.240998		Iteration1	6.904569	4.673657	69.81699
	Iteration2	2.642977	0.267158	0.404198		Iteration2	7.83421	5.040926	55.31672
	Iteration3	2.148319	0.194886	0.217035		Iteration3	6.820886	4.755653	55.44844
	Iteration4	2.419418	0.192132	0.223983		Iteration4	7.260251	4.713734	55.44752
Graph3	Iteration0	2.336337	0.23904	0.317139	Graph3	Iteration0	6.61326	2.813148	67.42443
	Iteration1	2.420362	0.230679	0.305537		Iteration1	6.813439	4.777506	54.22542
	Iteration2	2.413615	0.229667	0.348324		Iteration2	6.606355	4.794106	54.86473
	Iteration3	2.270817	0.241988	0.365414		Iteration3	6.797893	4.647988	55.55499
	Iteration4	2.274747	0.227675	0.217752		Iteration4	6.63854	4.642721	55.31108
Graph4	Iteration0	2.357989	0.235299	0.349178	Graph4	Iteration0	6.933986	4.829741	67.69279
	Iteration1	2.232303	0.249308	0.267922		Iteration1	6.921036	4.80792	55.29302
	Iteration2	2.445624	0.237474	0.241017		Iteration2	6.757086	4.669101	54.28844
	Iteration3	2.34482	0.220079	0.2953		Iteration3	6.871955	4.894745	57.64256
	Iteration4	2.572491	0.183344	0.298588		Iteration4	6.792528	4.648925	55.542
Average		2.459912	0.226854	0.302757			7.111239	4.733104	59.79226

As we can see from the table above the running times vary based on the graph the algorithm is running on. At one extreme, the Kruskal's algorithms takes 59.79s on average to compute the max bandwidth path on a dense graph. This can be explained by the number of edges present in the graph. If each of the 5000 vertices is connected to 20% of the other vertices (i.e. one vertex is connected to 1000 vertices) then the number of edges to be sorted in the first step of the algorithm is $5 * 10^6$. On the other extreme we have Dijkstra's algorithm with heap implementation on the sparse graph. This takes 0.227s on average to compute the path.

It can be seen from the results that for a dense graph the and a pair of source-destination vertex the computation time for:

Djkistra's with Max Heap << Djikistra's without Max Heap << Kruskal's algorithm

Similarly, for a sparse graph:

Djkistra's with Max Heap << Kruskal's algorithm << Djikistra's without Max Heap

Even though Kruskal's takes considerable time to build a spanning tree from a dense graph, it should be noted that once the tree is built, it takes very little time to compute the path between two nodes. In fact, this operation is of the order O(E). Thus, if we want to calculate multiple bandwidth paths between multiple source vertex paths, in a static graph, it makes sense to build the Kruskal's tree.

However, if there are no constraints on the graph, the safest approach to tackle the problem would be the Dijkstra's algorithm implemented with a heap.