CSCE 629 Analysis of Algorithms Project Report

Naresh Choudhary UIN: 427008954

Introduction

Today network is an important asset and optimizing it to achieve high bandwidth path is crucial. The purpose of this project is to implement a network routing protocol learnt in the class. Two famous algorithms for finding maximum bandwidth path are Dijkstra and Kruskal. Final goal of this project is to implement both of these algorithms to find maximum bandwidth path in a dense graph and a sparse graph. While implementing various versions of these algorithms using JAVA, I found new details which are shared throughout the report.

Graph Data Structure

I have used adjacency list to represent Graph. It is an array of Linked Lists with each LinkedList holding *Edges*. Every *Edge* object holds 3 variables: u (source), v(destination) and w (weight). So, for a given graph with V vertex, there will be V LinkedList's where LinkedList at index i representing vertex i.

Methods:

1. *Graph(int numberOfVertex):* Default constructor to initialize V LinkedList
2. *addEdge(int u, int v, int w):* This method adds an object *Edge(u,v,w)* to the head of LinkedList at position u. Since we need a undirected graph, another object *Edge(v,u,w)* is added to LinkedList at v.

Graph Representation:

```
0 ==> [(0,6,15), (0,3,13), (0,8,2), (0,2,19)]
1 ==> [(1,6,7), (1,3,19), (1,4,12), (1,2,20)]
2 ==> [(2,7,14), (2,8,8), (2,1,20), (2,0,19)]
3 ==> [(3,9,15), (3,1,19), (3,0,13), (3,7,11)]
4 ==> [(4,6,12), (4,7,12), (4,9,15), (4,1,12)]
5 ==> [(5,8,9), (5,9,15), (5,7,4), (5,6,12)]
6 ==> [(6,4,12), (6,1,7), (6,0,15), (6,5,12), (6,8,6)]
7 ==> [(7,4,12), (7,2,14), (7,3,11), (7,5,4)]
8 ==> [(8,5,9), (8,2,8), (8,0,2), (8,6,6), (8,9,18)]
9 ==> [(9,5,15), (9,3,15), (9,8,18), (9,4,15)]
```

Random Graph Generation

Graph First task is to generate Graph of two kind: Dense and Sparse. It uses Random class provided by java to generate random number in the range of zero to *numberOfVertex*. Sample output of Graph in Appendix 1.4. Below are methods in the class *GraphGenerator.java* :

1. *byDegree(int degree)*: For every vertex u, it generates edge with random vertex and random weight and joins it with u until the number of edges become equal to degree; where

   v ranges [v+1, *numberOfVertex*]
   w ranges (0, 2*numberOfVertex]

   Since it is possible that for u, random vertex v might generate two times resulting in two edges between (u,v) with different weight, I used a hash Set to add only unique edges.

   Code:

```
for (int i = 0; i < numberOfVertex; i++) {

        Clear vertexSet
        Add existing edges in LinkedListAt(i) into vertex Set.

        count = vertexSet. Size();
        loopControl = 0;

        while (count < edgesPerVertex && loopControl < 100000) {
                int randomVertex = generateRandomNumber.nextInt(numberOfVertex - i) + i;
                int randomWeight = generateRandomNumber.nextInt(2 * numberOfVertex) + 1;

                if (i != randomVertex && !vertexSet.contains(randomVertex)) {
                        g.addEdge(i, randomVertex, randomWeight);
                        count++;
                        vertexSet.add(randomVertex);
                        }
                        loopControl++;
        }

}
```

2. *byPercentage(long percentage):* This method calculates degree based on percentage and numberOfVertex and then uses same logic as 1 to generate graph.
3. *intializeBasicConnectedGraph():* Only using above method can create many disjoint graphs. Therefore, this method connects all vertex once by creating an array storing all vertex, shuffling them and connecting vertex at index i to vertex at i+1

```
        ArrayList<Integer> list = new ArrayList<Integer>();

        for(int j=0; j< numberOfVertex; j++)
                        list.add(j);

        Collections.shuffle(list);

        for(int i =0; i< numberOfVertex-1;i++) {
                int tempWeight = generateRandomNumber.nextInt(2 *numberOfVertex) + 1;
                g.addEdge(list.get(i), list.get(i+1), tempWeight);
        }
```

Heap Structure

Maximum Heap structure is used in Dijkstra 2 and Kruskal algorithm to sort vertex and edges respectively. Structure is similar to the algorithm written in Homework #2. For both the algorithms, goal was to sort vertices/edges based on their bandwidth. This is done by using two arrays H[i] and D[i] where value of H[i] gives the name of the vertex and value of D[H[i]] gives the bandwidth based on which vertex is sorted.

1. MaxHeapForDijkstra2.java: It is used as max heap for Dijkstra algorithm. Code can be found at *Appendix 2*. Below are the methods:
   1.1 *MaxHeapForDijkstra2(numberOfVertex):* Constructor to initialized 3 arrays of size V.
   1.2 *add(Vertex, bandwidth):* This method adds the vertex in the last index of H and then *heapifyUp()* the vertex based on its bandwidth. O(n) will be log(n).
   1.3 *pollMax():* This method returns the maximum value in the heap (H[0]), replaces H[0] by *H[lastIndex]* and then performs *heapifyDown()* to maintain heap property. In short *pollMax*() performs *MAXIMUM* and *DELETE* operation. O(n) will be log(n).
   1.4 *update(vertex, newWeight)*: In Dijkstra, there is a requirement that a vertex already present in Heap needs to be deleted (not necessarily at topmost element) and added again with new Bandwidth. So this method 1.Updates D[i] with new value in O(1); 2. Finds the index j of vertex in H[i] and then calls *heapifyUp()* at index j because in maximum bandwidth path Dijkstra will always update a vertex's bandwidth only when it is greater then existing bandwidth. Hence we call *heapifyUp()* to move the vertex up in the Heap.
   Now the difficult part is to find the index j at which vertex is currently stored in H[]. Finding it by looping through H[i] will take O(n) time. Therefore we create another array *vertexLocator[]* which stores current index of vertex v in H[]. In short, value of *vertexLocator[i]* gives current index of vertex i in Heap. Time taken for *update*() will be log(n).
   Value of these vertex must be changed as soon as the vertex changes position in heap. Hence it is done in *swap()* and *heapifyUp(*).
   1.5 heapifyUp(): Standard method to move vertex up with greater bandwidth
   1.6 heapifyDown(): Standard method to move vertex down

2. MaxHeapForKruskal.java: This is used as Heap for Kruskal's Max bandwidth path. This heap uses an array of Edges and sorts them based on their weight. Algorithm remains the same as used in Homework #2. Code can be found at *Appendix 3*. Methods used are:
   2.1 add(Edge e)
   2.2 pollMax()
   2.3 heapifyDown();
   2.4 heapifyUp(index j);
   2.5 swap();
   2.6 Other helper methods

Routing Algorithms

Input to all three algorithms is a Graph G, source s and destination t. Algorithm outputs the maximum bandwidth from source to destination.

Dijkstra's Algorithm without Heap

MaxBandwidthPathDijkstra1.java: It uses 3 arrays to store status, parent and bandwidth of each vertex. Status of a vertex can be White for unseen, Grey for fringe and Black for in-tree. Since we are using array, to find maximum fringe, we need to iterate through entire array resulting in a time complexity of O(n^2). Pseudo code is as below:

```
INPUT: G(source, destination)

1.   Initialize status[], dad[], bandwidth[] ;
2.   for vertex = 1 to V
             status[vertex] = White ;

             bandwidth[vertex] = ∞ ;

3.   status[source] = Black ;
4.   for each edge (source,v)
             status[v] = Grey ;
             bandwidth[v] = weight[source,v] ;
             dad [v] = source ;
5.   while (status[destination] != Black)
             Take u of maximum bandwidth from bandwidth[u] ;
             status[u] = Black ;
             for each edge (u,v)
                     if (status[v] === White)
                             status[v] = Grey ;
                             dad[v] = u ;
                             bandwidth[v] = Min (bandwidth[u], weight (u,v)) ;
                     else if (  status[v] === Grey &&  bandwidth[v] < weight (u,v) )
                             bandwidth[v] = Min (bandwidth[u], weight (u,v)) ;
                             dad[v] = u ;
6.   return bandwidth[destination] ;
```

Dijkstra's Algorithm with Heap

MaxBandwidthPathDijkstra2: In this algorithm we change the way to find maximum fringe using a Max Heap rather than array. We start by adding all the neighbor vertex of source to heap and then polling the maximum bandwidth vertex from heap in *O(logn)* time. Next for the unseen neighbor vertex, we add them to heap. And if the vertex is in fringe, we update its bandwidth if new bandwidth is greater than previous. Time complexity of this algorithm is *O(mlogn),* where m is the number of edges and n is the number of vertex. Pseudo code is as below:

```
INPUT: G(source, destination)

1.  Initialize status[], dad[], bandwidth[], heap F ;
2.  for vertex = 1 to V
            status[vertex] = WHITE ;
            bandwidth[vertex] = ∞ ;
3.  status[source] = BLACK ;
4.  for each edge (source,v)
            status[v] = GREY ;
            bandwidth[v] = weight[source,v] ;
            dad [v] = source ;
            F.add(v) ;
5.  while (F.isNotNull)
            u = F.max(), Delete (F, u) ;
            status[u] = BLACK ;
            for each edge (u,v)
                    if (status[v] === WHITE)
                            status[v] = GREY ;
                            dad[v] = u ;
                            bandwidth[v] = Min (bandwidth[u], weight (u,v)) ;
                            Insert(F, v) ;
                    else if (  status[v] === GREY &&  bandwidth[v] < weight (u,v) )
                            bandwidth[v] = Min (bandwidth[u], weight (u,v)) ;
                            dad[v] = u ;
                            UPDATE(F, v) ;
6.  return bandwidth[destination] ;
```

Kruskal's Algorithm

Kruskal's implementation builds maximum spanning tree and then uses BFS to find the maximum bandwidth between source and destination. This is done by adding all the edges in heap and sorting them in non increasing order based on their weight. Additionally, we used MakeSet, Find and Union taught in the class to keep track of parent, size and rank of each vertex. These methods allow us to make a maximum spanning tree.

```
INPUT: G(source, destination)

    1.  Initialize status[], dad[], bandwidth[], edgeHeap F ;
    2.  Sort the edges using F;
    3.  Initialize newGraph;
    4.  for vertex = 1 to V
                MakeSet(vertex) ;
    5.  for each edge E(u,v)
                rank[u] = FIND(u);
                rank[v] = FIND(v) ;
                if(rank[u] != rank[v])
                        addEdge(u,v) to newGraph;
                        UNION(rank[u], rank[v]);
    6.  Perform BFS on newGraph to find shortest path;
    7.  return bandwidth[destination] ;
```

```
MakeSet(vertex)

        dad[vertex] = 0;
        rank[vertex] = 1 ;
```

```
FIND(vertex)

        while( dad[vertex] != vertex)
                vertex = dad[vertex];
        return vertex ;
```

```
UNION (r1, r2)

        if ( rank[r1] > rank[r2] )
                dad[r2] = r1;
        else if ( rank[r1] < rank[r2] )
                dad[r1] = r2;
        else
                dad[r1] = r2;
                rank[r2] ++ ;
```

Testing

Below are the results after running all three algorithms for 10 pair of graphs (5 dense and 5 sparse) with 5 pair of source and destination for each graph. Below are the results:

****

Graph Type: Sparse

****

Graph	Vertexs	Edges	Source	Destination	Dijkstra_Max_BW|| Time Taken	Dijkstra_Max_BW_WithHeap|| Time Taken	Kruskal_Max_BW|| Time Taken
Graph5	5000	31288	1939	3960	6614||158ms	6614||16ms	6614||33ms
Graph5	5000	31288	4655	1024	5635||170ms	5635||11ms	5635||32ms
Graph5	5000	31288	2778	192	7105||190ms	7105||10ms	7105||120ms
Graph5	5000	31288	1810	418	7572||121ms	7572||1ms	7572||18ms
Graph5	5000	31288	3884	39	6985||106ms	6985||5ms	6985||12ms
Graph4	5000	31342	4238	3226	6834||96ms	6834||1ms	6834||13ms
Graph4	5000	31342	2738	2252	7827||126ms	7827||3ms	7827||18ms
Graph4	5000	31342	4179	2831	7773||101ms	7773||0ms	7773||11ms
Graph4	5000	31342	2980	1597	5359||110ms	5359||5ms	5359||12ms
Graph4	5000	31342	1296	1160	5002||64ms	5002||5ms	5002||12ms
Graph3	5000	31260	2829	2715	7341||95ms	7341||3ms	7341||12ms
Graph3	5000	31260	4842	1367	8007||71ms	8007||2ms	8007||11ms
Graph3	5000	31260	3027	2971	7346||93ms	7346||6ms	7346||20ms
Graph3	5000	31260	1144	3323	7084||95ms	7084||3ms	7084||13ms
Graph3	5000	31260	3413	2838	6068||68ms	6068||2ms	6068||11ms
Graph2	5000	31338	2246	1963	7305||71ms	7305||3ms	7305||16ms
Graph2	5000	31338	2769	2047	5893||75ms	5893||4ms	5893||10ms
Graph2	5000	31338	1420	230	7840||63ms	7840||1ms	7840||10ms
Graph2	5000	31338	705	1772	7198||83ms	7198||2ms	7198||16ms
Graph2	5000	31338	67	3936	5885||107ms	5885||3ms	5885||17ms
Graph1	5000	31310	1137	1798	7085||83ms	7085||3ms	7085||12ms
Graph1	5000	31310	1548	4735	6956||82ms	6956||5ms	6956||18ms
Graph1	5000	31310	744	1379	7602||105ms	7602||3ms	7602||17ms
Graph1	5000	31310	3652	596	6789||106ms	6789||5ms	6789||17ms
Graph1	5000	31310	2129	51	6799||87ms	6799||6ms	6799||18ms

```
**************************************************************************************************************
Graph Type: Dense
**************************************************************************************************************
```

Graph	Vertexs	Edges	Source	Destination	Dijkstra_Max_BW|| Time Taken	Dijkstra_Max_BW_WithHeap|| Time Taken	Kruskal_Max_BW|| Time Taken
Graph5	5000	5001010	3557	1675	9979||289ms	9979||158ms	9979||9714ms
Graph5	5000	5001010	1993	529	9972||266ms	9972||188ms	9972||7426ms
Graph5	5000	5001010	3206	223	9973||240ms	9973||87ms	9973||8200ms
Graph5	5000	5001010	795	2801	9982||205ms	9982||93ms	9982||8026ms
Graph5	5000	5001010	2305	382	9985||270ms	9985||40ms	9985||8167ms
Graph4	5000	5000780	2933	2404	9988||223ms	9988||1057ms	9988||6730ms
Graph4	5000	5000780	595	3160	9981||237ms	9981||123ms	9981||6663ms
Graph4	5000	5000780	4387	1694	9981||205ms	9981||151ms	9981||6803ms
Graph4	5000	5000780	1672	288	9983||279ms	9983||32ms	9983||7084ms
Graph4	5000	5000780	386	4080	9988||210ms	9988||16ms	9988||6712ms
Graph3	5000	5000982	1598	3317	9966||284ms	9966||175ms	9966||9062ms
Graph3	5000	5000982	982	786	9984||279ms	9984||160ms	9984||6798ms
Graph3	5000	5000982	3006	1524	9989||233ms	9989||40ms	9989||6880ms
Graph3	5000	5000982	4455	1142	9978||178ms	9978||54ms	9978||6905ms
Graph3	5000	5000982	634	915	9969||276ms	9969||144ms	9969||6806ms
Graph2	5000	5001296	3019	2524	9973||278ms	9973||77ms	9973||7002ms
Graph2	5000	5001296	4351	4957	9989||281ms	9989||70ms	9989||6929ms
Graph2	5000	5001296	3507	1183	9984||197ms	9984||139ms	9984||6802ms
Graph2	5000	5001296	560	4982	9983||207ms	9983||381ms	9983||9208ms
Graph2	5000	5001296	930	3365	9968||312ms	9968||117ms	9968||7603ms
Graph1	5000	5000988	265	4194	9982||247ms	9982||185ms	9982||6790ms
Graph1	5000	5000988	3379	3780	9971||182ms	9971||140ms	9971||6871ms
Graph1	5000	5000988	4493	2279	9992||272ms	9992||78ms	9992||6566ms
Graph1	5000	5000988	2632	3980	9988||221ms	9988||101ms	9988||6569ms
Graph1	5000	5000988	2465	2557	9985||193ms	9985||102ms	9985||6570ms

Note: For same graph Graph1 and different pair of source and destination, I am calculating spanning tree again. Another way can be for same Graph, I could have generated spanning tree only once and could have found rest of 4 source destination pair using BFS.

Time is calculated by using Java inbuilt function *System.currentTimeMillis()* which gives current time in milliseconds. Time is calculated as below:

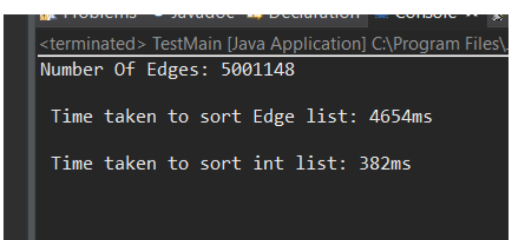
```java
startTime = System.currentTimeMillis();
maxD1 = maxBandwidthPathDijkstra1.maxBandwidthPath(g, source, destination);
endTime = System.currentTimeMillis();
System.out.print("    " + maxD1 + "||" + (endTime - startTime) + "ms ");
```

Observation

1. For sparse graph, Dijkstra without heap takes more time than Dijkstra with heap. As explained earlier, former takes *$O(n^2)$* while latter takes *$O(mlogn)$*.
2. For dense graph, above statement holds true.
3. For sparse graph, overall rank is Dijkstra(with Heap) > Kruskal > Dijkstra(without Heap)
4. For dens graph, overall rank is Dijkstra(with Heap) > Dijkstra(without Heap) > Kruskal
5. For dense graph, Kruskal is expected to take relatively lesser time but in implementation it is taking surprisingly more time. This might be due to the reason that Dijkstra(with heap) stops as soon as destination is black but Kruskal calculates whole spanning tree and then do the BFS. Another reason might be because I created a heap of edges and sorting edges (based on weight) compared to sorting integer takes more time.
   To check this, I made the edge class “comparable” and used Java provided sorting function. Time take by sorting edges was higher than sorting integers.

```
public static void edgeVsIntSorting() {

		GraphGenerator graphGenerator = new GraphGenerator(5000);
		Graph g = graphGenerator.byDegree(1000);
		System.out.println("Number Of Edges: " + g.getNumberOfEdge());

		Edge[] edgelist = new Edge[g.getNumberOfEdge()];
		int[] intList = new int[g.getNumberOfEdge()];

		// adding edges in array
		for (int i = 0, k = 0,x=0; i < g.G.length; i++) {
			ArrayList<Edge> adjList = new ArrayList<>(g.getLinkedListAtPosition(i));
			for (int j = 0; j < adjList.size(); j++)
				edgelist[k++] = adjList.get(j);

			for (int j = 0; j < adjList.size(); j++)
				intList[x++] = adjList.get(j).getW();
		}

		long startTime = 0, endTime = 0;
		startTime = System.currentTimeMillis();
		Arrays.sort(edgelist);
		endTime = System.currentTimeMillis();
		System.out.println("\n Time taken to sort Edge list: " + (endTime - startTime) +
"ms");

		startTime = System.currentTimeMillis();
		Arrays.sort(intList);
		endTime = System.currentTimeMillis();
		System.out.println("\n Time taken to sort int list: " + (endTime - startTime) +
"ms");
	}
```

O/P:



To optimize Kruskal, instead of creating a heap of edges, I could have used the heap used in Dijkstra. Instead of adding vertex based on “vertex & bandwidth” edges could have been added based on “Edge_ID (0 to 4999) & weight”. Other way to optimize was to use compression Kruskal.

Appendix

1. Testing HeapForKruskal:

To test if Heap for Kruskal is polling correct edge, I compared the output of inbuilt function to sort and output of MaxHeap.java. To demo, I generated graph of 5000 vertex with 5000000 edges. After generating, added all the edges into the Heap as well as to an array eList. Then used method *Arrays.sort()* to sort the array list. Then polled each element out of heap and compared it with Array to check if both are same or not.

```
public static void testHeap() {

		GraphGenerator graphGenerator = new GraphGenerator(5000);
		Graph g = graphGenerator.byDegree(1000);
		System.out.println("g.numberofEdge " + g.getNumberOfEdge());
		MaxHeapForKruskal maxHeap = new MaxHeapForKruskal(g.getNumberOfEdge());
		Edge[] elist = new Edge[g.getNumberOfEdge()];

		int k = 0;

		for (int i = 0; i < g.G.length; i++) {
			ArrayList<Edge> adjList = new ArrayList<>(g.getLinkedListAtPosition(i));

			// adding edges in heap
			for (Edge edge : adjList)
				maxHeap.add(edge); // adding edges in array
			for (int j = 0; j < adjList.size(); j++)
				elist[k++] = adjList.get(j);
		}

		Arrays.sort(elist);

		Edge temp;
		Boolean b = false;
		for (int s = 0; s < elist.length && !(maxHeap.isNull()); s++) {
			temp = maxHeap.pollMax();
			if (elist[s].w != temp.w) {
				System.out.println("elist edge: " + elist[s] + "is not same as "
+ temp);

				b = true;
			}
		}
		if (b)
			System.out.println("heap failing");
		else
			System.out.println("Heap working fine");

	}
```

2. MaxHeapForDijkstra

```
public class MaxHeapForDijkstra2 {
	int lastIndex, x, previousW, tempVertices, i, largeChildIndex, maxValue = 0;
	int indexOfVetexInHeap = -1;
	int[] H;
	int[] D;
	int[] vertexLocator;

	// constructor
	public MaxHeapForDijkstra2(int numberOfVertex) {
		H = new int[numberOfVertex];
		D = new int[numberOfVertex];
		vertexLocator = new int[numberOfVertex];
	}
	public void add(int vertex, int bandwidth) {
		H[lastIndex] = vertex;
		D[vertex] = bandwidth;
		heapifyUp(lastIndex);
		lastIndex++;
	}
	public int pollMax() {
		lastIndex--;
		if (lastIndex < 0)
			return -1;
		maxValue = H[0];
		H[0] = H[lastIndex];
		heapifyDown();
		return maxValue;
	}
	public void update(int v, int w) {
		previousW = D[v];
		D[v] = w;
		x = vertexLocator[v];
		if (w > previousW) heapifyUp(x);
	}
	public void heapifyDown() {
		i = 0;
		largeChildIndex = 0;
		while (hasLeftChild(i)) {
			largeChildIndex = getLeftChildIndex(i);
			if (hasRightChild(i) && rightChildBandwidth(i) > leftChildBandwidth(i))
				largeChildIndex = getRightChildIndex(i);

			if (D[H[i]] > D[H[largeChildIndex]])
				break;
			else
				swap(i, largeChildIndex);

			i = largeChildIndex;
		}

	}
	public void heapifyUp(int index) {


		  if (!(hasParent(index) && parentBandwidth(index) < D[H[index]]))
		  vertexLocator[H[index]] = index;


		while (hasParent(index) && parentBandwidth(index) < D[H[index]]) {

			swap(getParentIndex(index), index);
			index = getParentIndex(index);
		}
	}

	public void swap(int pos1, int pos2) {

		vertexLocator[H[pos1]] = pos2;
		vertexLocator[H[pos2]] = pos1;
		tempVertices = H[pos1];
		H[pos1] = H[pos2];
		H[pos2] = tempVertices;
	}

	// other helper methods
```

3. MaxHeapForKruskal

```
public class MaxHeapForKruskal {
	int lastIndex = 0;
	Edge[] EdgeArray;

	public MaxHeapForKruskal(int numberOfEdge) {
		EdgeArray = new Edge[numberOfEdge];
	}
	public void add(Edge e) {
		EdgeArray[lastIndex] = e;
		heapifyUp(lastIndex);
		lastIndex++;
	}
	public Edge pollMax() {

		lastIndex--;
		if (lastIndex < 0)
			System.out.println("gone wrong");
		Edge maxValue = EdgeArray[0];
		//System.out.println(maxValue);
		EdgeArray[0] = EdgeArray[lastIndex];
		heapifyDown();
		return maxValue;
	}
	public void heapifyDown() {
		int i = 0;
		int largeChildIndex = 0;
		while (hasLeftChild(i)) {
			largeChildIndex = getLeftChildIndex(i);
			if (hasRightChild(i) && rightChildBandwidth(i) > leftChildBandwidth(i))
				largeChildIndex = getRightChildIndex(i);

			if( EdgeArray[i].getW() > EdgeArray[largeChildIndex].getW())
				break;
			else
				swap(i, largeChildIndex);

			i = largeChildIndex;
		}
	}
```

4. Generated Graph Structure

```
62 ==> [(62,54,181), (62,42,129), (62,32,112), (62,14,162), (62,13,98), (62,7,97), (62,45,176), (62,60,7)]
63 ==> [(63,98,158), (63,84,7), (63,55,64), (63,5,194), (63,67,110), (63,73,12)]
64 ==> [(64,82,116), (64,50,104), (64,32,4), (64,25,177), (64,40,130), (64,99,168)]
65 ==> [(65,85,198), (65,54,81), (65,49,184), (65,7,110), (65,98,154), (65,14,141)]
66 ==> [(66,73,59), (66,75,4), (66,56,94), (66,44,22), (66,53,25), (66,90,188)]
67 ==> [(67,70,60), (67,76,82), (67,82,178), (67,80,53), (67,48,55), (67,63,110)]
68 ==> [(68,82,76), (68,92,95), (68,48,143), (68,26,44), (68,74,100), (68,25,86)]
69 ==> [(69,92,141), (69,60,159), (69,42,170), (69,33,15), (69,11,163), (69,72,76)]
70 ==> [(70,87,98), (70,90,9), (70,67,60), (70,17,58), (70,31,74), (70,7,12)]
71 ==> [(71,84,33), (71,99,91), (71,90,19), (71,61,123), (71,15,67), (71,16,181)]
72 ==> [(72,84,52), (72,78,12), (72,45,69), (72,33,139), (72,69,76), (72,6,55)]
73 ==> [(73,86,45), (73,66,59), (73,24,45), (73,2,157), (73,63,12), (73,77,151)]
74 ==> [(74,96,62), (74,83,12), (74,26,1), (74,23,158), (74,79,84), (74,68,100)]
75 ==> [(75,84,171), (75,66,4), (75,55,122), (75,45,136), (75,46,119), (75,41,39)]
76 ==> [(76,67,82), (76,59,16), (76,33,77), (76,17,94), (76,12,190), (76,77,78), (76,57,155)]
77 ==> [(77,40,98), (77,35,178), (77,32,181), (77,27,44), (77,73,151), (77,76,78)]
78 ==> [(78,72,12), (78,42,18), (78,5,72), (78,0,183), (78,18,33), (78,93,107)]
79 ==> [(79,94,47), (79,83,58), (79,56,169), (79,47,4), (79,22,186), (79,74,84)]
80 ==> [(80,83,98), (80,84,187), (80,67,53), (80,37,120), (80,38,186), (80,36,138)]
81 ==> [(81,33,30), (81,10,123), (81,8,6), (81,1,31), (81,85,33), (81,3,185)]
82 ==> [(82,85,5), (82,68,76), (82,67,178), (82,64,116), (82,28,7), (82,30,84)]
83 ==> [(83,80,98), (83,79,58), (83,74,12), (83,61,119), (83,53,21), (83,15,132), (83,29,92), (83,95,142)]
84 ==> [(84,80,187), (84,75,171), (84,72,52), (84,71,33), (84,63,7), (84,42,105), (84,39,134), (84,13,198), (84,6,183),
(84,1,200), (84,4,50)]
85 ==> [(85,82,5), (85,65,198), (85,13,142), (85,9,75), (85,7,6), (85,41,119), (85,81,33)]
86 ==> [(86,91,6), (86,73,45), (86,36,196), (86,24,184), (86,39,32), (86,29,83)]
87 ==> [(87,92,188), (87,70,98), (87,46,92), (87,30,128), (87,95,3), (87,89,170)]
88 ==> [(88,98,181), (88,92,110), (88,23,58), (88,20,10), (88,36,2), (88,27,110)]
89 ==> [(89,93,121), (89,99,185), (89,97,16), (89,34,59), (89,87,170), (89,92,142)]
90 ==> [(90,71,19), (90,70,9), (90,53,123), (90,37,69), (90,21,117), (90,20,49), (90,14,56), (90,66,188), (90,11,70)]
91 ==> [(91,94,44), (91,86,6), (91,48,49), (91,2,192), (91,59,79), (91,31,105)]
92 ==> [(92,88,110), (92,87,188), (92,69,141), (92,68,95), (92,59,101), (92,16,195), (92,89,142), (92,10,105)]
93 ==> [(93,96,197), (93,89,121), (93,52,181), (93,35,4), (93,78,107), (93,39,197)]
94 ==> [(94,91,44), (94,79,47), (94,29,159), (94,20,61), (94,14,100), (94,61,147)]
95 ==> [(95,97,21), (95,98,151), (95,55,146), (95,21,177), (95,83,142), (95,87,3)]
96 ==> [(96,93,197), (96,74,62), (96,49,61), (96,43,79), (96,8,87), (96,1,38), (96,19,57), (96,58,198)]
97 ==> [(97,95,21), (97,89,16), (97,48,126), (97,31,136), (97,3,101), (97,37,95)]
98 ==> [(98,95,151), (98,88,181), (98,63,158), (98,41,29), (98,31,43), (98,6,163), (98,33,179), (98,65,154)]
99 ==> [(99,89,185), (99,71,91), (99,45,43), (99,22,50), (99,3,80), (99,64,168), (99,51,147)]
```