**Algorithmic Aspects of Telecommunication Networks**

**Project 1 – Aparna Pavithran (axp161730)**

1. **How the program works**

Program has 7 Java files. Driver.java will drive the entire program. From the Driver file, it will go to the Graph.java file for reading the inputs of graph. Graph file will read all the inputs as such number of vertices in the graph, k value and the utd id. After reading all the inputs it will create vertices and edges with the help of Vertex.java and Edge.java files. If the number of vertices is 20 then utd id has to be 10-digit number. If the number of vertices is given as 10 then utd id has to be given as 5-digit number. Number of vertices cannot be odd due to this. After reading all the inputs it will print the input Graph.

After this Graph file will call fastsoln function in the FastSolution.java file to do the algorithmic part of it. This will do all the calculations needed basically the algorithmic part of the project. Fastsoln function will create an adjacency matrix for the cost of the edges, that is aij. It is picking k random values for each edge and then based on that it will give edge weight or cost as 1 or 200. If the aij=1 if j =k.

The traffic demand or bij value is also calculated here. It is calculated from the input utd id. bij = |utd id i| - |utd id j|.

Call BellmanFord.java to get the shortest cost distances for each pair of source and sink. BellmanFordEvaluation in the BellmanFord.java file will return an array of distances. Also call ShortVertices.java file to find out the paths used such that the cost of the path is the shortest for each sources and sinks. This will return an adjacency matrix of 0 and 1. 0 if the link is utilized by any of the shortest path or 1 if the link is not utilized. This will give a count of all the used and unused links in the graph.

Finally cost is calculated by simply adding cost of all the shortest paths. Capacity is calculated by multiplying the traffic demand (bij) and cost. Density is calculated by count of all the used links/total number of links in the graph ie n\*(n-1).

Cost, Capacity and density of the graph is printed along with the final resulted graph with only used links.

Inputs :

Number of vertices : 20

K value : 3 .. 13

Utdid : 1 2 3 4 5 6 6 7 8 9 (10 digit number with a space)

1. **Program Flow: - flowcharts**

Vertex

Driver

FastSolution

BellmanFord

ShortVertices

Print Capacity, Cost, Density and Result Graph

Print Input Graph

Graph

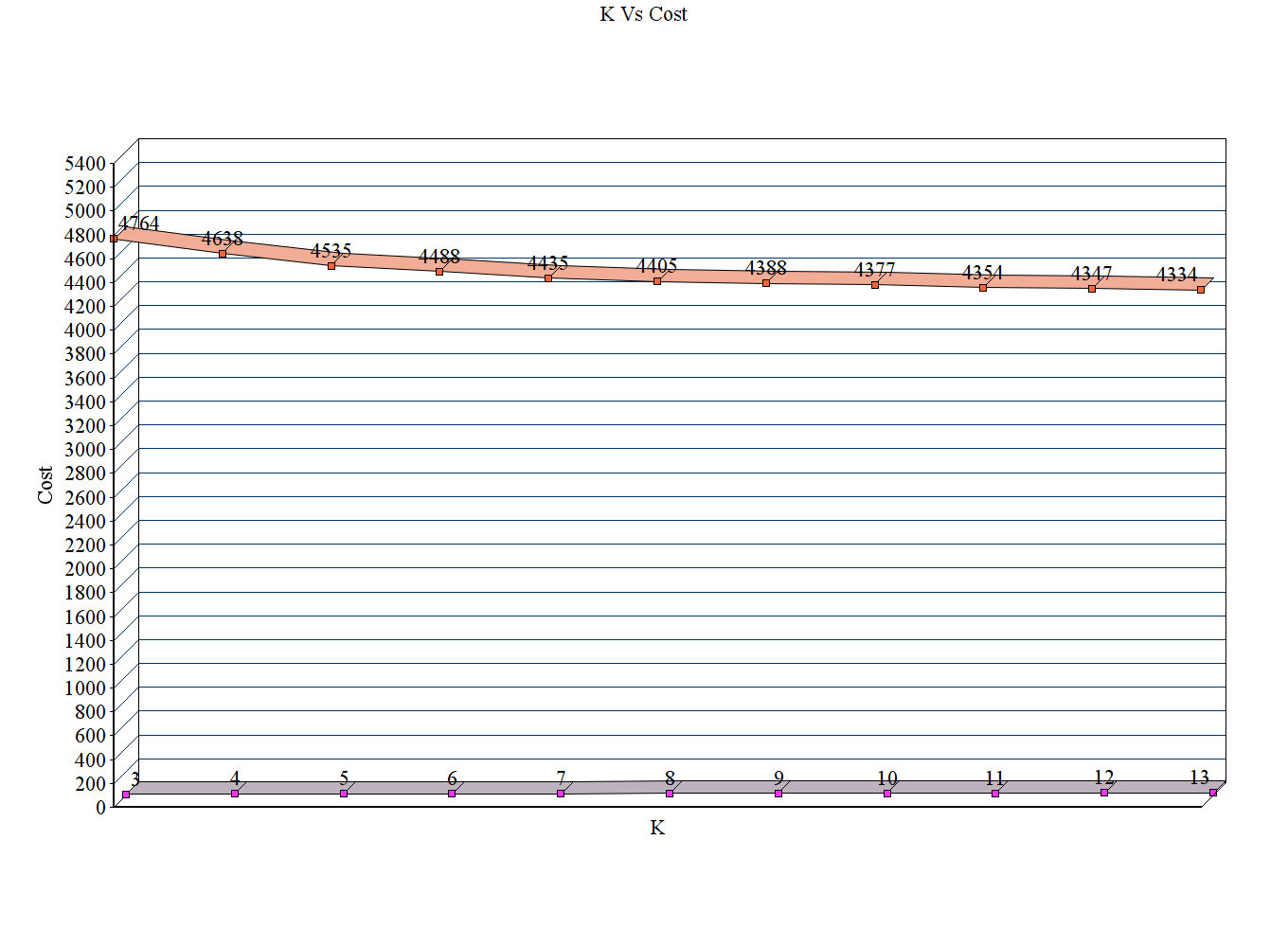
Edge

1. **Graphs**

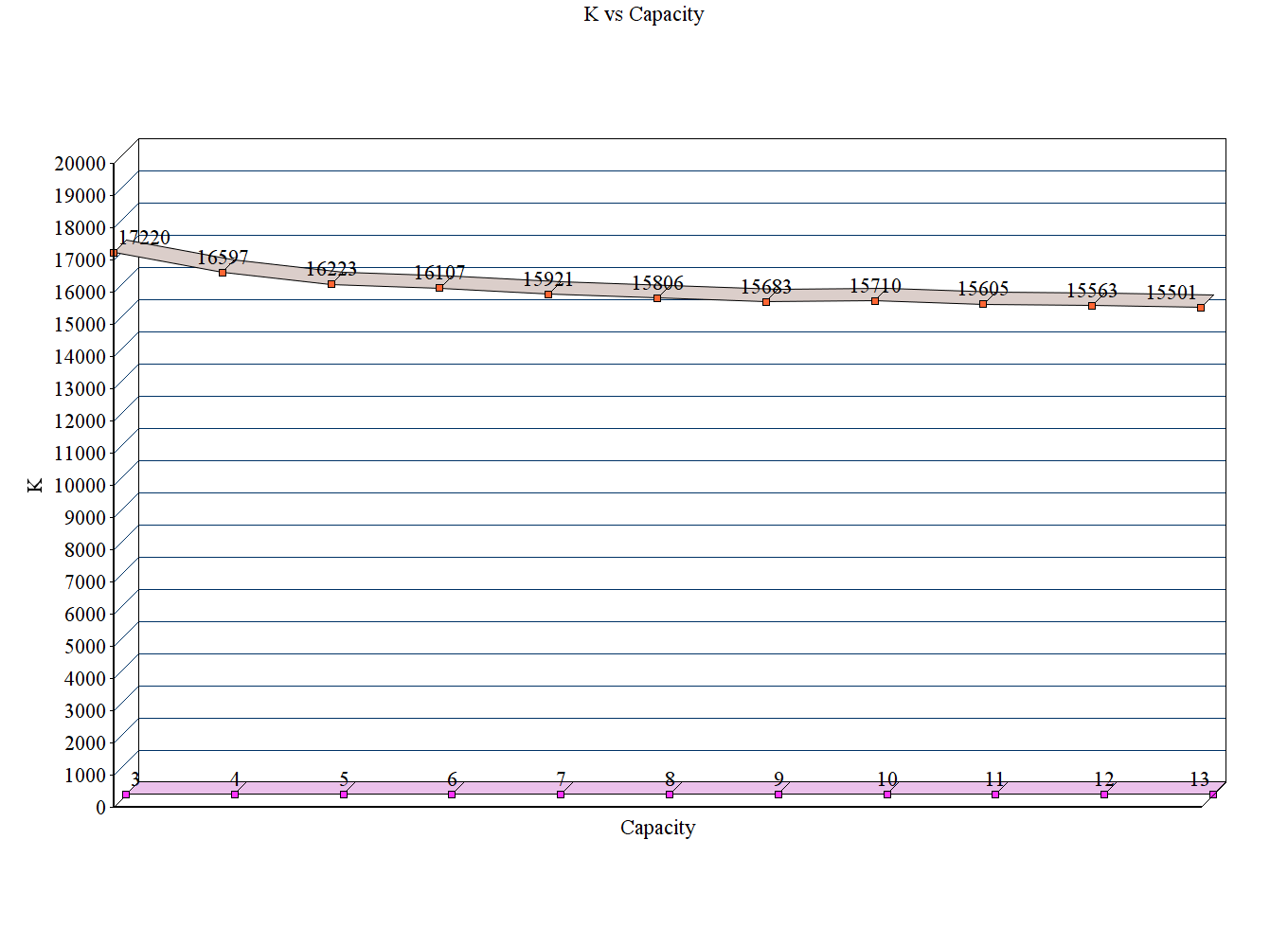
Below does giving the value of the program output plot figures that will plot the graphs with actual program results.

1. k value versus cost
2. k value versus capacity
3. k value versus density
4. **k value versus cost**

Cost will decrease as the k value increases. They are inversely proportional to each other. So graph below will clearly show a dip in cost values as the k value increases. It starts from 4764 for k=3 and dips to 4334 when k=13.



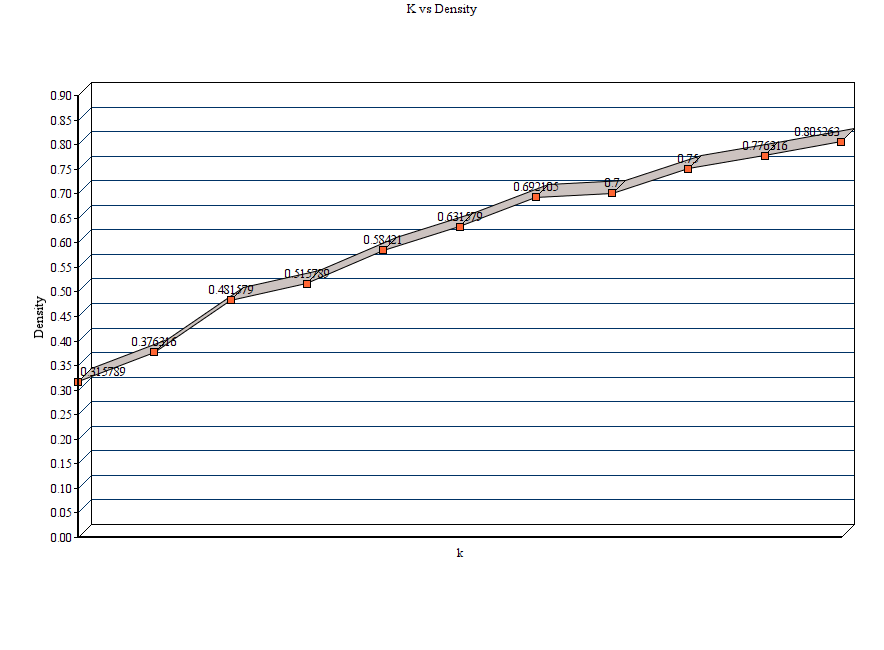
1. **k value versus capacity**



Capacity will decrease as the k value increases. They are inversely proportional to each other. So graph below will clearly show a dip in capacity values as the k value increases. It starts from 17220 for k=3 and dips to 15501 when k=13.

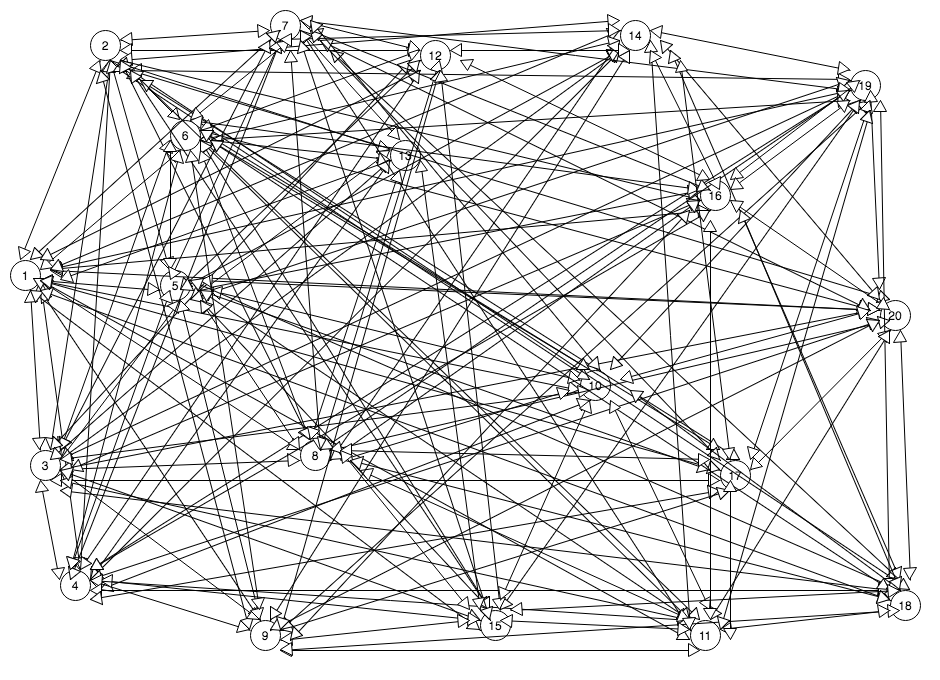
1. **k value versus density**

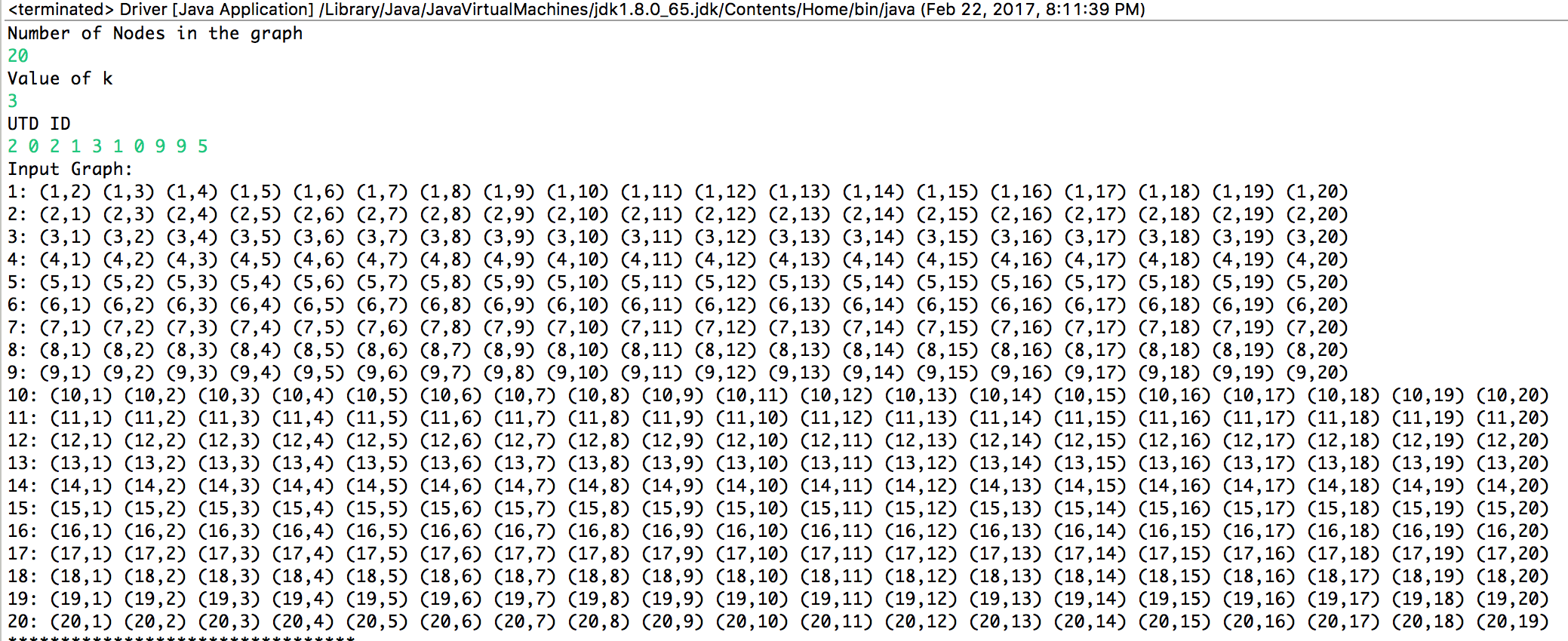
Density is between 0 and 1. When the k value is very small as in case of 3 it is small and it is .31789. As the k value increases and approaches the number of nodes density increases and reaches to 1. Here it is plotted only up to k=13 and it is .805263.



1. Input graph

Due to the complexity of the graph, input graph is printed as adjacency matrix.



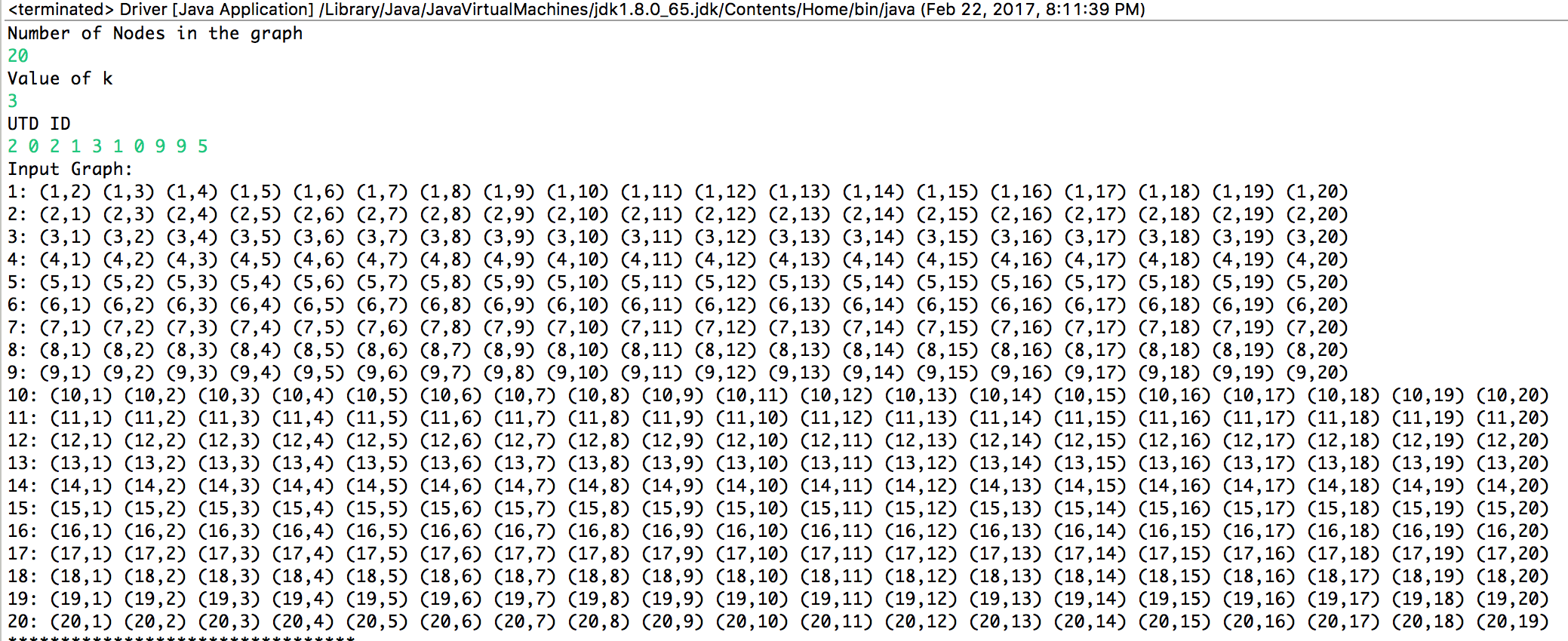


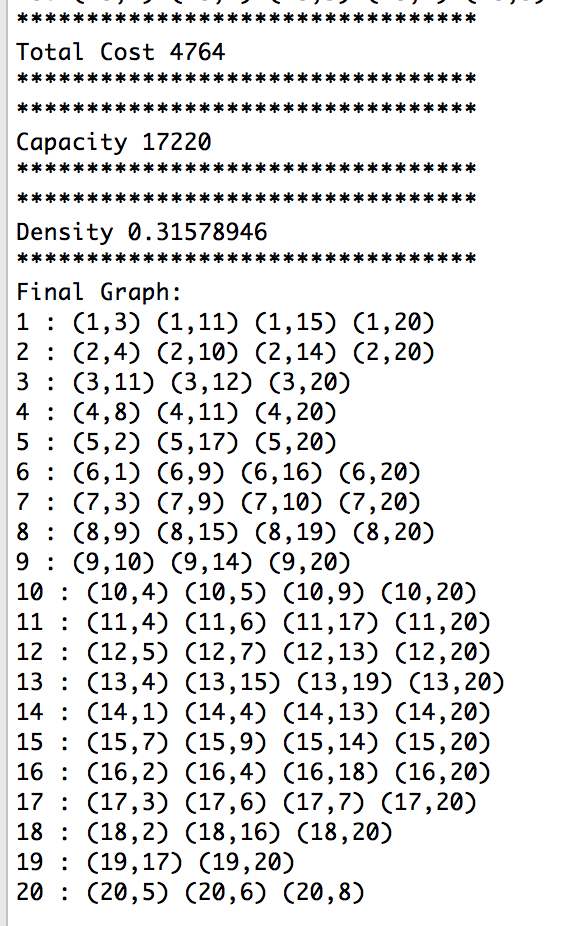
Input graph when k=3

1. **Output graph**

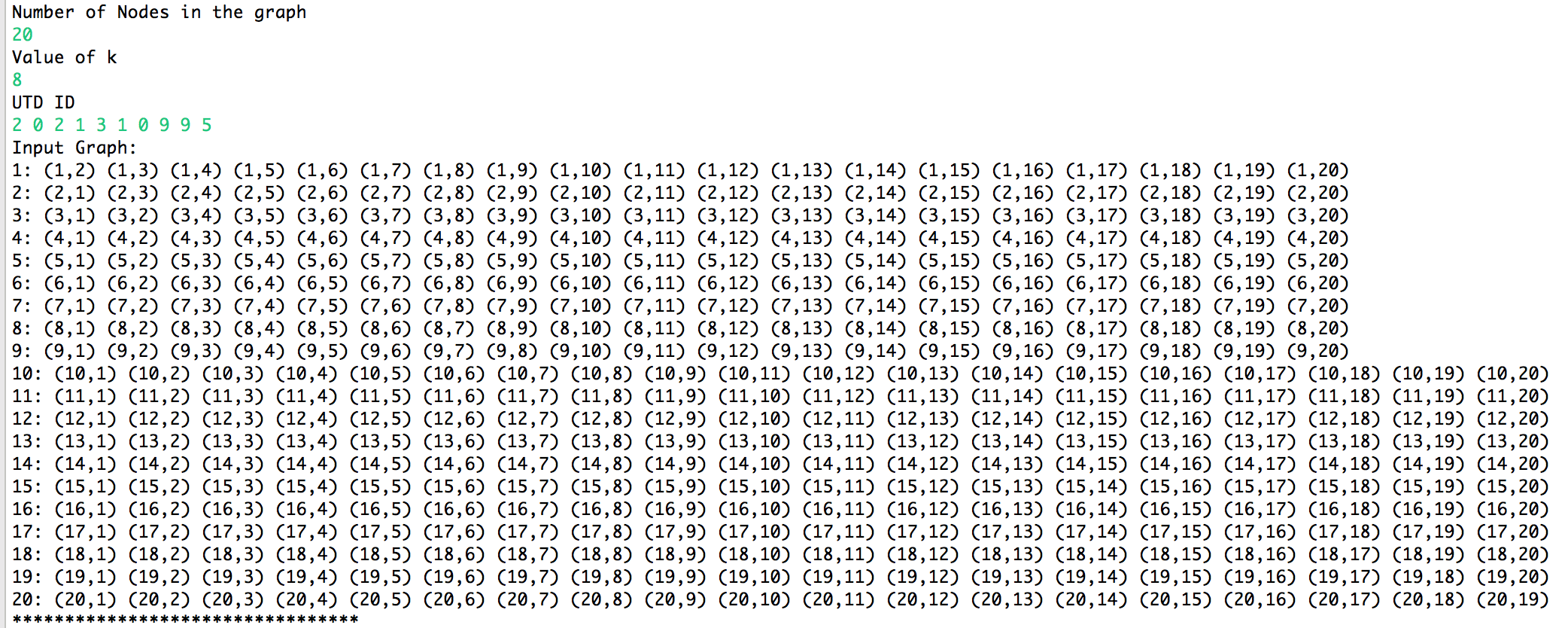
Due to the complexity of the graph, output graph is printed as adjacency matrix.

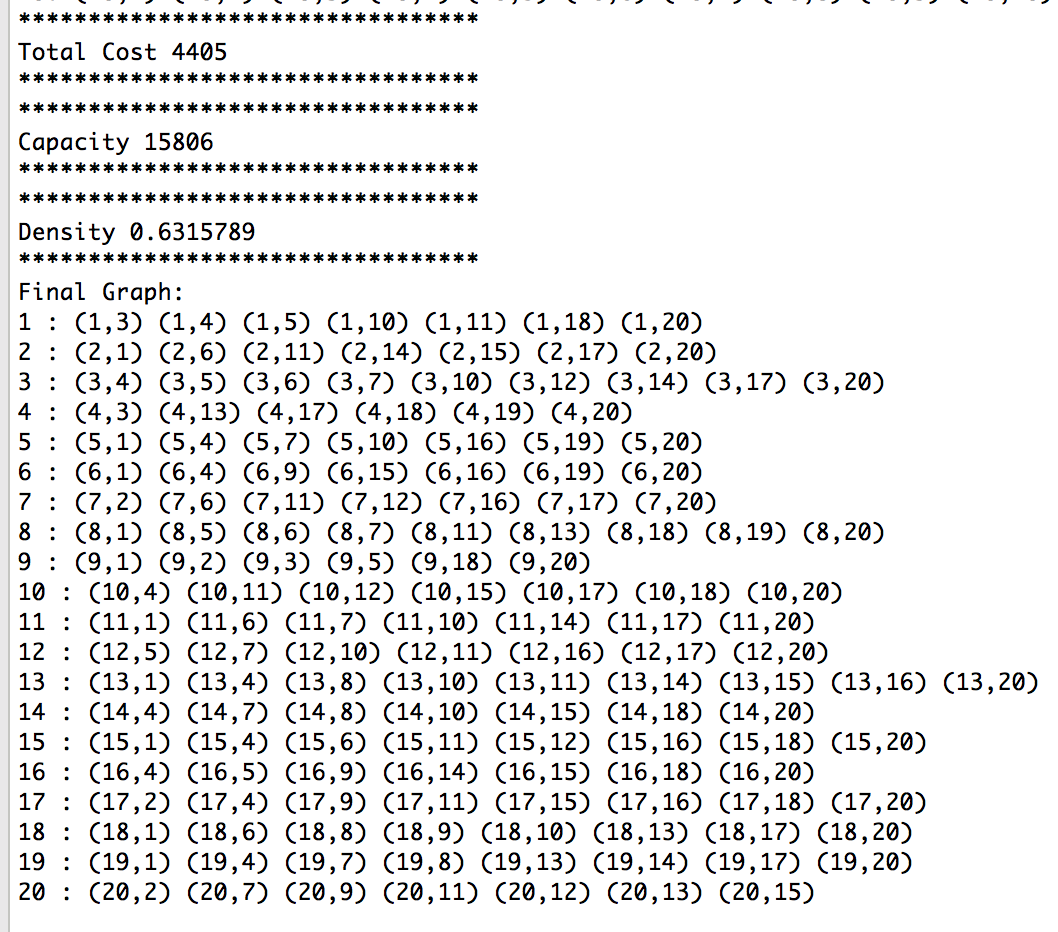
**K=3**



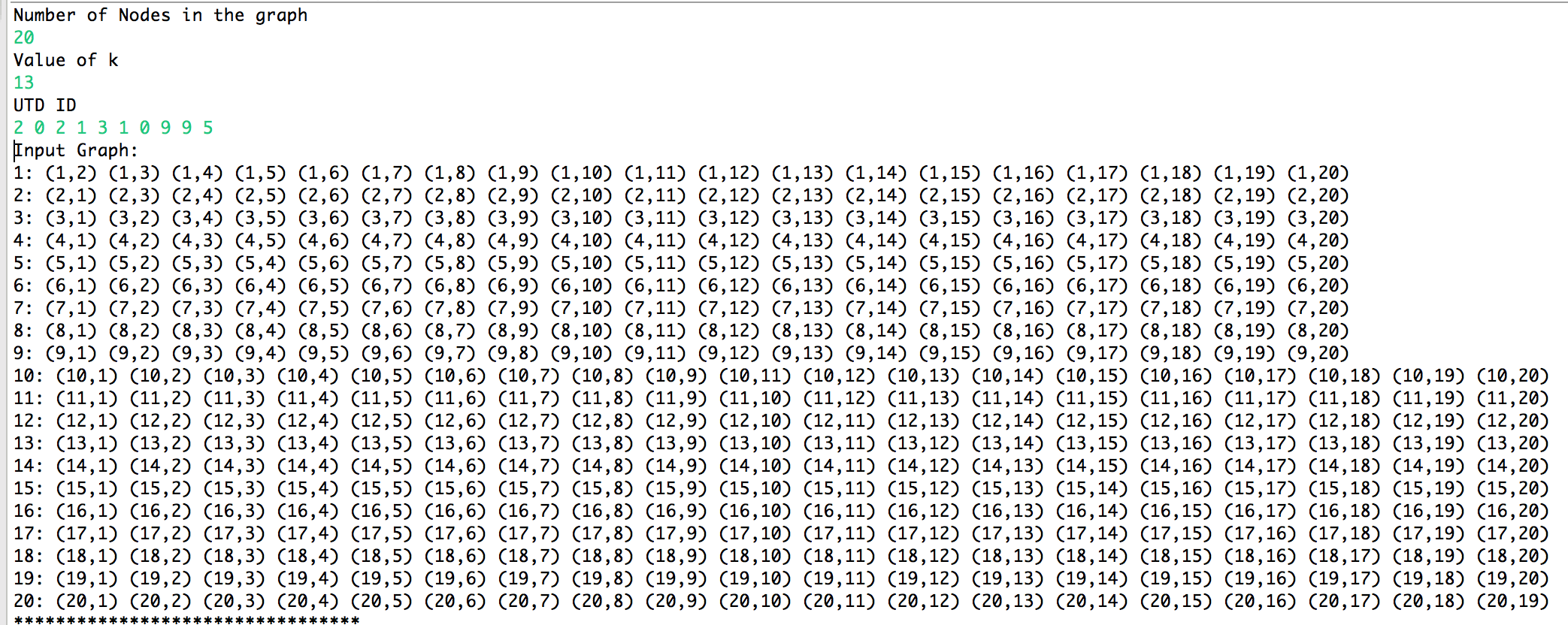


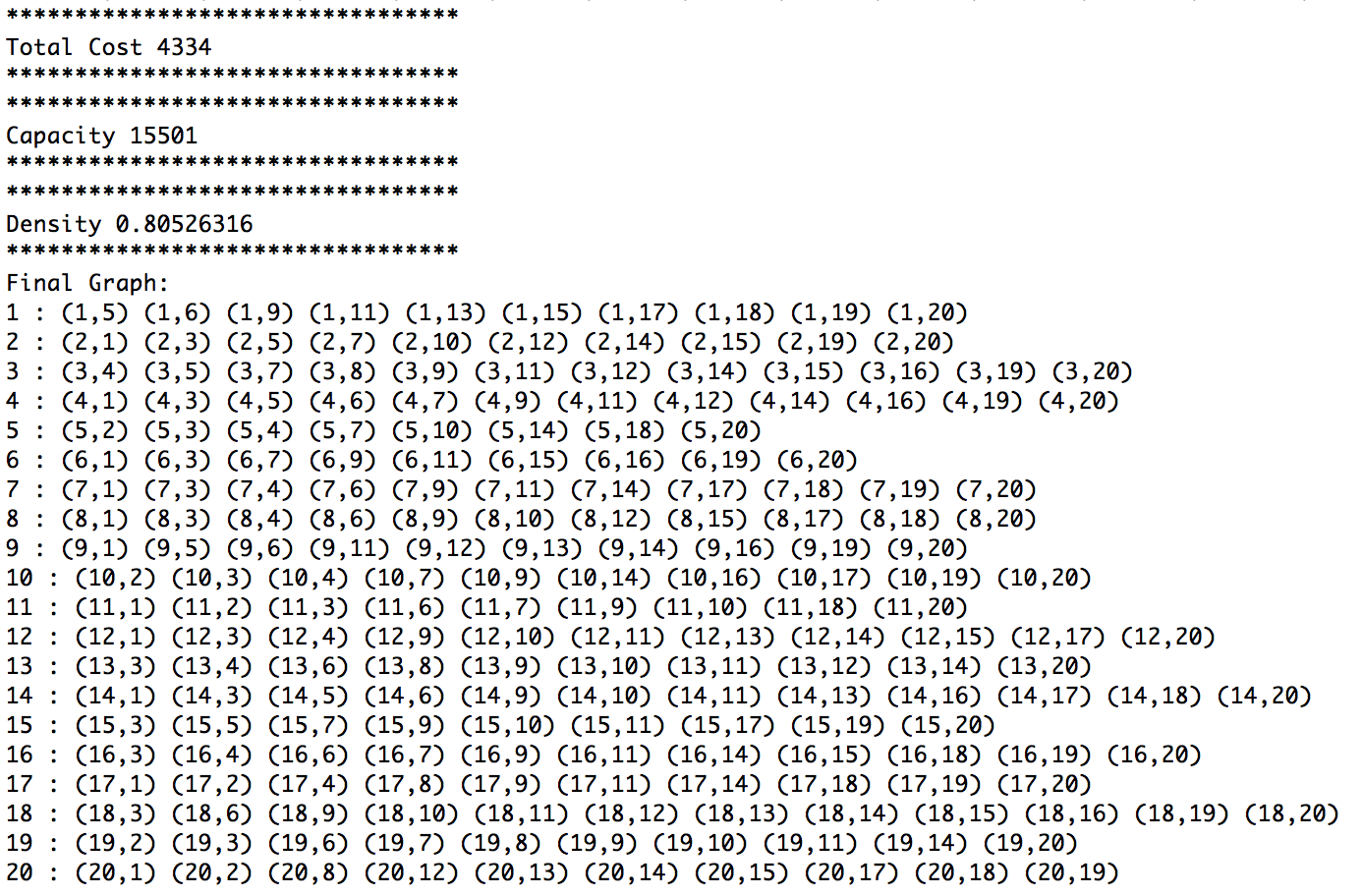
**k=8**





**k=13**





1. **Appendix**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Driver.java

/\*\* Sample driver program using the graph class

\*

\*/

import java.io.FileNotFoundException;

import java.util.Scanner;

import java.io.File;

public class Driver {

public static void main(String[] args) throws FileNotFoundException {

Scanner in;

if (args.length > 0) {

File inputFile = new File(args[0]);

in = new Scanner(inputFile);

} else {

in = new Scanner(System.in);

}

Graph.readDirectedGraph(in);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Graph.java

/\*\*

\* Class to represent a graph

\*

\*

\*/

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**import** java.util.List;

**import** java.util.Queue;

**import** java.util.Scanner;

**public** **class** Graph **implements** Iterable<Vertex> {

List<Vertex> v; // vertices of graph

**int** size; // number of verices in the graph

**boolean** directed; // true if graph is directed, false otherwise

**int** min=Integer.***MAX\_VALUE***;

/\*\*

\* Constructor for Graph

\*

\* **@param** size

\* : int - number of vertices

\*/

Graph(**int** size) {

**this**.size = size;

**this**.v = **new** ArrayList<>(size + 1);

**this**.v.add(0, **null**); // Vertex at index 0 is not used

**this**.directed = **false**; // default is undirected graph

// create an array of Vertex objects

**for** (**int** i = 1; i <= size; i++)

**this**.v.add(i, **new** Vertex(i));

}

/\*\*

\* Find vertex no. n

\* **@param** n

\* : int

\*/

Vertex getVertex(**int** n) {

**return** **this**.v.get(n);

}

/\*\*

\* Method to add an edge to the graph

\*

\* **@param** a

\* : int - one end of edge

\* **@param** b

\* : int - other end of edge

\* **@param** weight

\* : int - the weight of the edge

\*/

**void** addEdge(Vertex from, Vertex to, **int** weight) {

Edge e = **new** Edge(from, to, weight);

**if**(**this**.directed) {

from.adj.add(e);

to.revAdj.add(e);

} **else** {

from.adj.add(e);

to.adj.add(e);

}

}

/\*\*

\* Method to create iterator for vertices of graph

\*/

**public** Iterator<Vertex> iterator() {

Iterator<Vertex> it = **this**.v.iterator();

it.next(); // Index 0 is not used. Skip it.

**return** it;

}

// read a directed graph using the Scanner interface

**public** **static** Graph readDirectedGraph(Scanner in) {

**return** *readGraph*(in, **true**);

}

// read an undirected graph using the Scanner interface

**public** **static** Graph readGraph(Scanner in) {

**return** *readGraph*(in, **false**);

}

**public** **static** Graph readGraph(Scanner in, **boolean** directed) {

// read the graph related parameters

System.***out***.println("Number of Nodes in the graph");

**int** n = in.nextInt(); // number of vertices in the graph - nodes

//System.out.println("Number of Edges in the graph");

//int m = in.nextInt(); // number of edges in the graph - edges

**int** m=n\*(n-1);

System.***out***.println("Value of k");

**int** k = in.nextInt();

// create a graph instance

Graph g = **new** Graph(n);

g.directed = directed;

//System.out.println("From To");

**for**(**int** i=1;i<=n;i++){

**for**(**int** j=1;j<=n;j++){

**if**(i!=j){

**int** u=i;

**int** v=j;

**int** w;

**if**(v<=k){

w=1;

}

**else**

{

w=200;

}

g.addEdge(g.getVertex(u), g.getVertex(v), w);

}

}

}

//////////////////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* utd id \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Integer[] utdid=**new** Integer[n];

System.***out***.println("UTD ID");

**for**(**int** i=0;i<n/2;i++){

**int** x=in.nextInt();

utdid[i]=x;

utdid[i+n/2]=x;

}

System.***out***.println("Input Graph:");

**for**(Vertex u: g) {

System.***out***.print(u + ": ");

**for**(Edge e: u.adj) {

e.otherEnd(u);

System.***out***.print(e + " ");

}

System.***out***.println();

}

FastSolution.*fastsoln*(n, k, utdid,g);

**return** g;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Edge.java

/\*\*

\* Class that represents an edge of a Graph

\*

\*

\*/

**import** java.lang.Exception;

**public** **class** Edge {

Vertex from; // head vertex

Vertex to; // tail vertex

**int** weight;// weight of edge

**int** capcity;

Edge(Vertex u, Vertex v, **int** w) {

from = u;

to = v;

weight = w;

capcity = 0;

}

**public** Vertex otherEnd(Vertex u) {

**assert** from == u || to == u;

// if the vertex u is the head of the arc, then return the tail else

// return the head

**if** (from == u) {

**return** to;

} **else** {

**return** from;

}

}

@Override

**public** **int** hashCode() {

**int** h1 = **this**.from.hashCode() + **this**.to.hashCode();

**return** h1;

}

@Override

**public** **boolean** equals(Object o) {

Edge e = (Edge) o;

**if**(o==**null** || o.getClass()!=**this**.getClass()){

**return** **false**;

}

**if** ((**this**.to == e.to) && (**this**.from == e.from)) {

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Return the string "(x,y)", where edge goes from x to y

\*/

**public** String toString() {

**return** "(" + from + "," + to + ")";

}

**public** String stringWithSpaces() {

**return** from + " " + to + " " + weight;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Vertex.java

/\*\*

\* Class to represent a vertex of a graph

\*

\*

\*/

import java.util.List;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.Iterator;

public class Vertex implements Iterable<Edge> {

int name; // name of the vertex

boolean seen; // flag to check if the vertex has already been visited

int d; Vertex p; // fields used in algorithms of Prim and Dijkstra

List<Edge> adj, revAdj; // adjacency list; use LinkedList or ArrayList

/\*\*

\* Constructor for the vertex

\*

\* @param n

\* : int - name of the vertex

\*/

Vertex(int n) {

name = n;

seen = false;

d = Integer.MAX\_VALUE;

p = null;

adj = new ArrayList<Edge>();

revAdj = new ArrayList<Edge>(); /\* only for directed graphs \*/

}

public Iterator<Edge> iterator() { return adj.iterator(); }

/\*\*

\* Method to represent a vertex by its name

\*/

public String toString() {

return Integer.toString(name);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FastSolution.java

import java.awt.Label;

import java.awt.List;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Random;

import java.util.Scanner;

public class FastSolution {

public static void fastsoln(int n,int k,Integer[] utdid,Graph g){

int numberofvertices = n;

int distances[] = new int[numberofvertices + 1];

Random r = new Random();

int Low = 1;

int High = n;

int adjacencymatrix[][] = new int[numberofvertices + 1][numberofvertices + 1];

//System.out.println("Enter the adjacency matrix");

for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++)

{

int krand[]=new int[k+1];

for(int rand=1;rand<=k;rand++){

krand[rand]=r.nextInt(High-Low) + Low;

//System.out.print("Random "+krand[rand]);

}

for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++)

{

int flag=0;

for(int rand=1;rand<=k;rand++){

if(krand[rand]==destinationnode)

flag=1;

}

if(flag==1){

adjacencymatrix[sourcenode][destinationnode] = 1;

}

else{

adjacencymatrix[sourcenode][destinationnode] = 200;

}

if (sourcenode == destinationnode)

{

adjacencymatrix[sourcenode][destinationnode] = 0;

continue;

}

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Adjacency Matrix\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

/\*for (int node = 1; node <= numberofvertices; node++)

{

for (int sn = 1; sn <= numberofvertices; sn++){

System.out.print(adjacencymatrix[node][sn]+" ");

}

System.out.println();

}\*/

ShortVertices shortvertices = new ShortVertices(numberofvertices);

int den[][]=shortvertices.shorts(adjacencymatrix, utdid);

int countden=0;

for (int node = 1; node <= numberofvertices; node++)

{

for (int sn = 1; sn <= numberofvertices; sn++){

//System.out.print(den[node][sn]+" "); ///// \*\*\*\* to print length of each path \*\*\*\* ////

if(den[node][sn]==1){

countden++;

}

}

//System.out.println();

}

int capacity=0;

int cost=0; int dens=0;

for(int i=1;i<=n;i++){

BellmanFord bellmanford = new BellmanFord(numberofvertices);

distances=bellmanford.BellmanFordEvaluation(i, adjacencymatrix,utdid);

for(int j=1;j<=n;j++){

if(i!=j){

Integer x=utdid[i-1]-utdid[j-1]; //bij value calculated

x=Math.abs(x);

int c=x\*distances[j];

capacity=capacity+c;

cost=cost+distances[j];

}

//System.out.print(distances[j]+" ");

if(distances[j]==1 || distances[j]==200){

dens++;

den[i][j]=1;

}

}

// System.out.println();

}

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Total Cost "+cost);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Capacity "+capacity);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

float edge=n\*(n-1);

if(countden+dens<(n\*(n-1)))

countden=countden+dens;

else if(countden<dens)

countden=dens;

float density;

density=countden/edge;

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Density "+density);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

// return distances;

System.out.println("Final Graph:");

if(countden<edge){

for(int i=1;i<=n;i++){

System.out.print(i+ " : ");

for(int j=1;j<=n;j++){

if(den[i][j]==1)

System.out.print("("+i+","+j+") ");

}

System.out.println();

}

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ShortVertices.java

**import** java.util.Scanner;

**public** **class** ShortVertices

{

**private** **int** distances[];

**private** String[] dist;

**private** **int** numberofvertices;

**public** **static** **final** **int** ***MAX\_VALUE*** = 9999999;

**public** ShortVertices(**int** numberofvertices)

{

**this**.numberofvertices = numberofvertices;

distances = **new** **int**[numberofvertices + 1];

dist = **new** String[numberofvertices + 1];

}

**public** **int**[][] shorts(**int** adjacencymatrix[][],Integer[] utdid)

{

**int** den[][] = **new** **int**[numberofvertices + 1][numberofvertices + 1];

**for** (**int** node = 1; node <= numberofvertices; node++)

{

**for** (**int** sn = 1; sn <= numberofvertices; sn++){

den[node][sn] = 0;

}

}

**for**(**int** j=1;j<=numberofvertices;j++){

dist[j]="";

}

**for**(**int** srcs=1;srcs<=numberofvertices;srcs++)

{

**int** src=srcs;

**for** (**int** node = 1; node <= numberofvertices; node++)

{

distances[node] = ***MAX\_VALUE***;

}

distances[srcs] = 0;

//System.out.println("Start "+srcs);

**for** (**int** node = 1; node <= numberofvertices - 1; node++)

{

//System.out.println("node : "+node);

**for** (**int** sourcenode = 1; sourcenode <= numberofvertices; sourcenode++)

{

//System.out.println("sourcenode : "+sourcenode);

**for** (**int** destinationnode = 1; destinationnode <= numberofvertices; destinationnode++)

{

//System.out.println("destinationnode : "+destinationnode);

**if** (distances[destinationnode] > (distances[sourcenode] + adjacencymatrix[sourcenode][destinationnode]))

{

distances[destinationnode] = distances[sourcenode] + adjacencymatrix[sourcenode][destinationnode];

**if**(srcs!=sourcenode)

den[sourcenode][destinationnode]=1;

}

}

}

}

}

**return** den;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BellmanFord.java

**import** java.util.Scanner;

**public** **class** BellmanFord

{

**private** **int** distances[];

**private** **int** numberofvertices;

**public** **static** **final** **int** ***MAX\_VALUE*** = 9999999;

**public** BellmanFord(**int** numberofvertices)

{

**this**.numberofvertices = numberofvertices;

distances = **new** **int**[numberofvertices + 1];

}

**public** **int**[] BellmanFordEvaluation(**int** source, **int** adjacencymatrix[][],Integer[] utdid)

{

**for** (**int** node = 1; node <= numberofvertices; node++)

{

distances[node] = ***MAX\_VALUE***;

}

**int** den[][] = **new** **int**[numberofvertices + 1][numberofvertices + 1];

**for** (**int** node = 1; node <= numberofvertices; node++)

{

**for** (**int** sn = 1; sn <= numberofvertices; sn++){

den[node][sn] = 0;

}

}

distances[source] = 0;

//System.out.println("Source : "+source);

**for** (**int** node = 1; node <= numberofvertices - 1; node++)

{

**int** src=source;

//System.out.println("node : "+node);

**for** (**int** sourcenode = 1; sourcenode <= numberofvertices; sourcenode++)

{

//System.out.println("sourcenode : "+sourcenode);

**for** (**int** destinationnode = 1; destinationnode <= numberofvertices; destinationnode++)

{

//System.out.println("destinationnode : "+destinationnode);

**if** (distances[destinationnode] > (distances[sourcenode] + adjacencymatrix[sourcenode][destinationnode]))

{

distances[destinationnode] = distances[sourcenode] + adjacencymatrix[sourcenode][destinationnode];

}

}

}

}

**return** distances;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **References**

Bellman Ford Algorithm is taken from the below given site.

1. <http://www.sanfoundry.com/java-program-implement-bellmanford-algorithm/>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Readme**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 1: Keep all the .java files in separate .java named files and save it.

Step 2: Compile all the java files. Eg : javac \*.java

Step3: Run the Driver.java file. Eg : Java Driver

Step 4: Program will prompt for inputs.

Step 5: Give the number of vertices. It should be even number as the utd id input is number of vertices by 2.

Step 6: Give the k value.

Step 7: Give the utd id. If the number of vertices is not 20 then give utd id as number of vertices/2 digits. Eg: 2 1 3 4 2 2 4 4 3 3. Note: give digits with a space.

Step 8: See the output.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Readme**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*