DOCUMENTATION

INTRODUCTION

I. THE TASK

We consider a graph that models a network (social, transport, etc.). The graph will be taken from a data set stored in a file. To write an application with the following requirements:

a) using the input data from the file, create a representation of the graph in the form of an adjacency list, adjacency matrix, edge list, etc.

b) at least 4 functions that apply one algorithm each to solve four real problems related to the studied graph. The algorithms must each belong to the following problems:

   i. Connectivity, topological sorting
   ii. Shortest path
   iii. Minimal spanning tree
   iv. Eulerian, Hamlitonian cycles
   v. Couplings in graphs (Matching)
   vi. Flows and transport networks (Maximum flow)

c) the project will also contain a documentation describing the domain that the graph models, the problems that will be solved by using algorithms, observations regarding performance, operating time, optimizations, etc.

PROJECT DESCRIPTION

I. THE DATASET

I got the data set from a site called: Network Repository. There are a lot of huge data sets in different formats on it: https://networkrepository.com/index.php.

My dataset is on the topic „Infrastructure Networks": https://networkrepository.com/inf-power.php..

II. PROJECT

My project consists of two sections:

- Coding
- Network visualisation

In the coding section I develop OOP based algorithms in C++ in order to solve the tasks.

The network visualisation is a suplimentary thing. It's a small python project that helps me visualise the graph and export the adjacency list, adjacency matrix and incidence matrix faster (task a)).

III. TOOL USED

For the coding part I used the following tools:

2C++ programming language

1Visual Studio 2022

For the network visualisation I used the following tools:

4Python programming language

3Visual Studio Code

6Pandas python library

5Networkx python library

8Matplotlib python library

7Numpy python library

Other tools:

10Excel

9Git CVS

TASK SOLVING

I. GENERAL

The tasks are solved in the coding section. In the „Coding/Graph network" folder we can find all the files used to solve the tasks: sources, headers and resources.

There are two sources files:

- Source.cpp
- inputData.cpp

Source.cpp is used to declare an object of the Graph class and call the method used to solve the tasks.

inputData.cpp contains the method used for importing the edges from dataset.

There are three header files:

- Graph.h
- sparseMatrix.h
- stringToType.h

Graph.h contains the structure of our class and it's main method.

sparseMatrix.h contains the sparseMatrix class wich helps us store the data for the adjacency and incidence matrices of our graph.

stringToType.h contains a template class used for converting data from string to a template type.

The resources folder contains another folder in wich we have our dataset in csv format. In the coding section, the inf-power-1.csv file is used.

II. ALGORITHMS

1. Task a) – data storage

This is the graph class:

```
template <class Type>
class Graph
{
private:
	unsigned long long vertices, edges, numberOfComponents;
	mutable std::list<Type>* adjacencyList;
	sparseMatrix <bool> adjacencyMatrix;
	sparseMatrix <bool> incidenceMatrix;
public:
	Graph();
	Graph(const unsigned long long&, unsigned long long edges = 0);
	void inputData();
	void addEdge(const unsigned long long&, const unsigned long long&);
	void addEdge(const unsigned long long&, const unsigned long long&, const unsigned long
long&);
	unsigned long long& getVertices();
	template <class Type> friend std::ostream& operator<<(std::ostream&, Graph <Type>&);
	void printAdjacencyList();
	void printAdjacencyMatrix();
	void printIncidenceMatrix();
	bool BFS(const Type&, const Type&, Type[], unsigned long long[]);
	void printShortestDistance(const Type&, const Type&);
	void printMST();
	void DFS(unsigned long long v, bool[], std::vector<Type>&);
	unsigned long long countAndPrintConnectedComponents();
	bool isConnected();
	bool isEulerian();
	void printEulerianCycles();
	bool isAdjacent(Type, Type);
	void printHamiltonianCycle(const Type&, const Type&);
	void printMaxMatching() const;
};
```

In the inputData.cpp file we import the edges of the graph:

```
#include "../Headers/Graph.h"
#include "../Headers/stringToType.h"
#include <fstream>
#include <vector>
#include <string>
```

```
#include <sstream>
#include <iostream>

template <class Type> void Graph <Type>::inputData()
{
	try
	{
		std::ifstream file;
		file.open("Resources/inf-power/inf-power-1.csv");
		std::string titles;
		getline(file, titles);
		std::string line;
		unsigned long long edge = 0;
		while (getline(file, line))
		{
			std::istringstream ss(line);
			std::string firstColumn, secondColumn;
			getline(ss, firstColumn, ',');
			getline(ss, secondColumn, ',');
			Type u = convert_to <Type>(firstColumn);
			Type v = convert_to <Type>(secondColumn);
			edge++;
			(*this).addEdge(u, v, edge);
		}
	}
	catch (...)
	{
		throw new std::exception("File error!");
	}
}
```

The process is simple: we try to open the file and if there aren't any errors we can start to read from it. We skip the first line because it contains the title of the columns and then with a while loop we get the nodes that form the current edge. Using the following lines of code we can convert the string data to our template type:

```
Type u = convert_to <Type>(firstColumn);
Type v = convert_to <Type>(secondColumn);
#pragma once
#include <sstream>
#include <string>

template <typename Type> Type convert_to(const std::string& string)
{
    std::istringstream ss(string);
    Type number;
    ss >> number;
    return number;
}
```

To add an edge we used the `void addEdge(const unsigned long long&, const unsigned long long&, const unsigned long long&)` method from our `Graph` class.

```
template <class Type> void Graph <Type>::addEdge(const unsigned long long& u, const unsigned
long long& v, const unsigned long long& edge)
{
      (*this).adjacencyList[u].push_back(v);
      (*this).adjacencyList[v].push_back(u);
      (*this).adjacencyMatrix.add(u, v, true);
      (*this).adjacencyMatrix.add(v, u, true);
      (*this).incidenceMatrix.add(u, edge, true);
      (*this).incidenceMatrix.add(v, edge, true);
}
```

For the adjacency list I opted for the STL library: `std::list<Type>* adjacencyList;`

I used sparse matrices to create the adjacency matrix and the incidence matrix. This helps us because in our graph there are many nodes that aren't connected with an edge and therefore, a lot of elements in our matrix would be equal to 0.

Here is the `sparseMatrix` class:

```
#include <iostream>
#include <vector>
#include <algorithm>

template <class Type> class sparseMatrix {
private:
    std::vector<Type> values;
    std::vector<unsigned long long> row_indices;
    std::vector<unsigned long long> column_indices;
    unsigned long long numberOfRows;
    unsigned long long numberOfColumns;
public:
    sparseMatrix() {};
    sparseMatrix(unsigned long long numberOfRows, unsigned long long numberOfColumns) :
numberOfRows(numberOfRows), numberOfColumns(numberOfColumns) {}
    void add(unsigned long long row, unsigned long long column, Type value) {
        values.push_back(value);
        row_indices.push_back(row);
        column_indices.push_back(column);
    }
    Type get(unsigned long long row, unsigned long long column) const {
        for (unsigned long long index = 0; index < values.size(); index++) {
            if (row_indices[index] == row && column_indices[index] == column) {
                return values[index];
            }
        }
        return 0;
    }
    void print() const{
        for (unsigned long long row = 0; row < 10; row++) {
            for (unsigned long long column = 0; column < 10; column++) {
                std::cout << get(row, column) << " ";
            }
            std::cout << std::endl;
        }
    }
    unsigned long long& getNumberOfRows()
    {
        return (*this).numberOfRows;
    }
    unsigned long long& getNumberOfColumns()
    {
```

```
        return (*this).numberOfColumns;
    }
};
```

This helps us enormously in terms of memory usage and speed.

To print the data we call the necessary methods in Source.cpp:

```
#include <iostream>
#include "inputData.cpp"

int main()
{
	Graph <unsigned long long> G(4941, 6954);
	std::cout << "The graph is processed...\n";
	G.inputData();
	G.countAndPrintConnectedComponents();
	G.printShortestDistance(2, 870);
	G.printMST();
	G.printEulerianCycles();
	G.printMaxMatching();
	std::cout << G;
	return 0;
}
```

Because it takes a lot of time to print the data in the csv files we print a message and then we first solve the task b). After that, by overloading the '<<' operator we export our data in csv format.

```
template <class Type> std::ostream& operator<<(std::ostream& output, Graph <Type>& graph)
{
	graph.printAdjacencyList();
	graph.printAdjacencyMatrix();
	graph.printIncidenceMatrix();
	return output;
}

template <class Type> void Graph <Type>::printAdjacencyList()
{
	std::ofstream file("Output/Adjacency List.csv");
	for (unsigned long long vertex = 0; vertex < (*this).vertices; vertex++)
	{
		file << vertex + 1 << ",";
		typename std::list<Type>::iterator i;
		for (i = adjacencyList[vertex].begin(); i != adjacencyList[vertex].end(); ++i)
		{
			file << *i;
			typename std::list<Type>::iterator j;
			j = i;
			j++;
			if (j != adjacencyList[vertex].end())
			{
				file << ",";
			}
		}
```

```
			file << "\n";
		}
		std::cout << "The adjacency list has been printed.\n";
		file.close();
}

template <class Type> void Graph <Type>::printAdjacencyMatrix()
{
		std::ofstream file("Output/Adjacency Matrix.csv");
		for (unsigned long long row = 0; row < (*this).vertices; row++)
		{
			for (unsigned long long column = 0; column < (*this).vertices ; column++)
			{
				file << (*this).adjacencyMatrix.get(row, column);
				if (column < (*this).vertices - 1)
				{
					file << ",";
				}
			}
			file << "\n";
		}
		std::cout << "The adjacency matrix has been printed.\n";
		file.close();
}

template <class Type> void Graph <Type>::printIncidenceMatrix()
{
		std::ofstream file("Output/Incidence Matrix.csv");
		for (unsigned long long row = 0; row < (*this).vertices; row++)
		{
			for (unsigned long long column = 0; column < (*this).vertices; column++)
			{
				file << (*this).incidenceMatrix.get(row, column);
				if (column < (*this).vertices - 1)
				{
					file << ",";
				}
			}
			file << "\n";
		}
		std::cout << "The incidence matrix has been printed.\n";
		file.close();
}
```

Note that the .csv files are not on GitHub because they are too big.

2. Task b)

I solved the first 5 tasks of the problem (with some restrictions because my graph is undirected and unweighted):

i. Connectivity

The algorithm for this task is implemented in the method countAndPrintConnectedComponents().

```
template <class Type> unsigned long long Graph<Type>::countAndPrintConnectedComponents()
{
	std::ofstream file("Output/Connected components.csv");
	bool* visited = new bool[(*this).vertices];
	for (unsigned long long index = 0; index < (*this).vertices; index++)
	{
		visited[index] = false;
	}
	for (unsigned long long index = 0; index < (*this).vertices; index++)
	{
		if (!visited[index])
		{
			(*this).numberOfComponents++;
			std::vector<Type> component;
			DFS(index, visited, component);
			file << "Component " << (*this).numberOfComponents << ", ";
			for (typename std::vector<Type>::iterator it = component.begin(); it != 
component.end(); ++it)
			{
				file << *it + 1 << ", ";
			}
			file << "\n";
		}
	}
	file << "Number of connected components: " << (*this).numberOfComponents << std::endl;

	delete[] visited;
	return (*this).numberOfComponents;
}
```

The algorithm uses depth-first search (DFS) to traverse the graph and count the number of connected components. It initializes an array of visited vertices and sets all values to false. Then, it iterates through all vertices, and for each unvisited vertex, it performs DFS and stores the vertices in the connected component. The algorithm has a time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because the DFS algorithm takes $O(V + E)$ time to traverse the graph, and the algorithm iterates through all vertices once. The space complexity of the algorithm is $O(V)$, where V is the number of vertices, due to the use of the visited array. Overall, the algorithm provides an efficient way to count and print the connected components of a graph.

ii. Shortest path

The algorithm for this task is implemented in the method printShortestDistance().

```
template <class Type> void Graph <Type>::printShortestDistance(const Type& source, const Type&
destination)
{
	std::ofstream file("Output/Shortest distance.csv");
	Type* predecessor = new Type[vertices];
	unsigned long long* distance = new unsigned long long[vertices];
	if (BFS(source, destination, predecessor, distance))
	{
		file << "The shortest distance between " << source << " and " << destination << "
is " << distance[destination] << ".\n";
		std::vector<Type> path;
		Type currentVertex = destination;
		path.push_back(currentVertex);
		while (predecessor[currentVertex] != -1)
		{
			currentVertex = predecessor[currentVertex];
			path.push_back(currentVertex);
		}
		file << "Shortest path is:,";
		for (typename std::vector<Type>::reverse_iterator it = path.rbegin(); it !=
path.rend(); ++it)
		{
			file << *it << ",";
		}
		file << ".\n";
	}
	else
	{
		file << "No path exists between " << source << " and " << destination << "\n";
	}
	delete[] predecessor;
	delete[] distance;
}

template <class Type> bool Graph<Type>::BFS(const Type& source, const Type& destination,  Type
predecessor[], unsigned long long distance[])
{
	std::queue<Type> queue;
	bool* visited = new bool[vertices];
	for (unsigned long long i = 0; i < vertices; i++)
	{
		visited[i] = false;
		predecessor[i] = -1;
		distance[i] = UINT_MAX;
	}
	visited[source] = true;
	distance[source] = 0;
	queue.push(source);
	while (!queue.empty()) {
		Type currentVertex = queue.front();
		queue.pop();
		for (auto it = adjacencyList[currentVertex].begin(); it !=
adjacencyList[currentVertex].end(); ++it) {
			if (!visited[*it]) {
				visited[*it] = true;
				distance[*it] = distance[currentVertex] + 1;
				predecessor[*it] = currentVertex;
				queue.push(*it);
				if (*it == destination) {
					delete[] visited;
					return true;
				}
			}
		}
	}
	delete[] visited;
	return false;
```

```
}
```

The algorithm uses breadth-first search (BFS) to find the shortest path between the source and destination nodes in the graph. It initializes an array of predecessor vertices and an array of distances from the source vertex to all other vertices in the graph. The BFS algorithm takes O(V + E) time, where V is the number of vertices and E is the number of edges in the graph. After BFS, the algorithm checks if a path exists between the source and destination nodes by checking if the distance array has been updated for the destination node. If a path exists, the algorithm traces back the path using the predecessor array and stores the path in a vector. The time complexity of tracing back the path is O(V), where V is the number of vertices in the graph. Therefore, the overall time complexity of the algorithm is O(V + E). The space complexity of the algorithm is O(V), where V is the number of vertices, due to the use of the predecessor and distance arrays.

Overall, the algorithm provides an efficient way to find the shortest path between two nodes in a graph.

iii. Minimal spanning tree

The algorithm for this task is implemented in the method printMST ().

```
template <class Type> void Graph<Type>::printMST()
{
	std::ofstream file("Output/MST.csv");
	std::priority_queue<std::pair<int, std::pair<unsigned long long, unsigned long long>>,
		std::vector<std::pair<int, std::pair<unsigned long long, unsigned long long>>>>
pq;
	for (unsigned long long index = 0; index < vertices; index++)
	{
		for (typename std::list<Type>::iterator it = adjacencyList[index].begin(); it !=
adjacencyList[index].end(); ++it) {
```

```
                        unsigned long long v = index;
                        unsigned long long w = *it;
                        if (v < w)
                        {
                                pq.push(std::make_pair(1, std::make_pair(v, w)));
                        }
                }
        }
        std::vector<unsigned long long> predecessor((*this).vertices);
        for (unsigned long long index = 0; index < vertices; index++)
        {
                predecessor[index] = index;
        }
        std::vector<std::pair<unsigned long long, unsigned long long>> mst;
        while (!pq.empty() && mst.size() < (*this).vertices - 1)
        {
                std::pair<int, std::pair<unsigned long long, unsigned long long>> curr =
pq.top();
                pq.pop();
                unsigned long long u = curr.second.first;
                unsigned long long v = curr.second.second;
                while (predecessor[u] != u)
                {
                        u = predecessor[u];
                }
                while (predecessor[v] != v)
                {
                        v = predecessor[v];
                }
                if (u != v)
                {
                        mst.push_back(std::make_pair(curr.second.first, curr.second.second));
                        predecessor[u] = v;
                }
        }
        file << "Minimum Spanning Tree:" << std::endl;
        for (typename std::vector<std::pair<unsigned long long, unsigned long long>>::iterator
it = mst.begin(); it != mst.end(); ++it)
        {
                file << it->first << "," << it->second << std::endl;
        }
}
```

This algorithm uses Prim's algorithm to find the minimal spanning tree of an undirected and unweighted graph. The time complexity of the algorithm is $O(E \log V)$, where E is the number of edges and V is the number of vertices in the graph. This is because the algorithm processes each edge once and uses a priority queue to keep track of the next smallest edge to add to the MST. The priority queue operations take $O(\log V)$ time, and since there can be at most E edges, the overall time complexity is $O(E \log V)$. The space complexity of the algorithm is $O(V)$, as it uses a disjoint set to keep track of the connected components of the graph, which requires a vector of size V to store the predecessors.

The use of a priority queue to process the edges in ascending order of weight is a common optimization technique for finding the minimal spanning tree.

iv. Eulerian cycles

The algorithm for this task is implemented in the method printEulerianCycles().

```
template <class Type> void Graph<Type>::printEulerianCycles()
{
	std::ofstream file("Output/Eulerian cycles.csv");
	if (!isEulerian())
	{
		file << "The graph is not Eulerian.\n";
		return;
	}
	std::vector<unsigned long long> circuit, nodes;
	circuit.reserve(edges);
	nodes.reserve(vertices);
	nodes.push_back(0);
	while (!nodes.empty()) {
		unsigned long long node = nodes.back();
		if (adjacencyList[node].empty())
		{
			for (auto u : circuit)
			{
				file << u << ",";
			}
			file << node << "\n";
			circuit.pop_back();
			nodes.pop_back();
			continue;
		}
		circuit.push_back(node);
		auto edge = adjacencyList[node].begin();
		nodes.push_back(*edge);
		adjacencyList[node].erase(edge);
	}
}

template <class Type> bool Graph<Type>::isEulerian()
{
	if (!(*this).isConnected())
	{
		return false;
	}
	for (unsigned long long i = 0; i < (*this).vertices; i++)
	{
		if ((*this).adjacencyList[i].size() % 2 != 0)
		{
			return false;
		}
	}
	return true;
}
```

The complexity of the printEulerianCycles() function is O(E), where E is the number of edges in the graph. This is because the function traverses each edge in the graph exactly once to construct the Eulerian cycle. Additionally, the function calls isEulerian() which has a complexity of O(V).

The complexity of the isEulerian() function is O(V), where V is the number of vertices in the graph. This is because the function checks each vertex in the graph to see if it has an odd degree, which takes O(1) time for each vertex.

v. Couplings in graphs (Matching)

The algorithm for this task is implemented in the method printMaxMatching().

```
template <class Type> void Graph<Type>::printMaxMatching() const
{
	std::ofstream file("Output/Maximum matching.csv");
	std::vector<std::list<Type>> adjList(vertices);
	for (unsigned long long i = 0; i < vertices; i++)
	{
		adjList[i] = adjacencyList[i];
	}
	std::unordered_map<Type, Type> matching;
	for (Type u = 0; u < vertices; u++)
	{
		for (auto it = adjList[u].begin(); it != adjList[u].end(); it++)
		{
			Type v = *it;
			if (matching.find(u) == matching.end() && matching.find(v) ==
matching.end())
			{
				matching[u] = v;
				matching[v] = u;
				adjList[u].clear();
				adjList[v].clear();
				for (Type w = 0; w < vertices; w++) {
					adjList[w].remove(u);
					adjList[w].remove(v);
				}
				break;
			}
		}
	}
	file << "Maximum matching:\n";
	for (auto it = matching.begin(); it != matching.end(); ++it)
	{
		if (it->first < it->second)
		{
			file << it->first << "," << it->second << "\n";
		}
	}
}
```

This algorithm used for finding a maximum matching of a graph has a time complexity of O(V^3), where V is the number of vertices in the graph. This is because the algorithm involves iterating over all vertices in the graph and checking all possible edges between those vertices. In the worst case, the algorithm could potentially check all possible pairs of vertices, resulting in O(V^2) iterations, and for each iteration, it could potentially remove all edges incident to those vertices, resulting in another O(V) operations. Thus, the total time complexity of the algorithm is O(V^3).

3. Network visualization

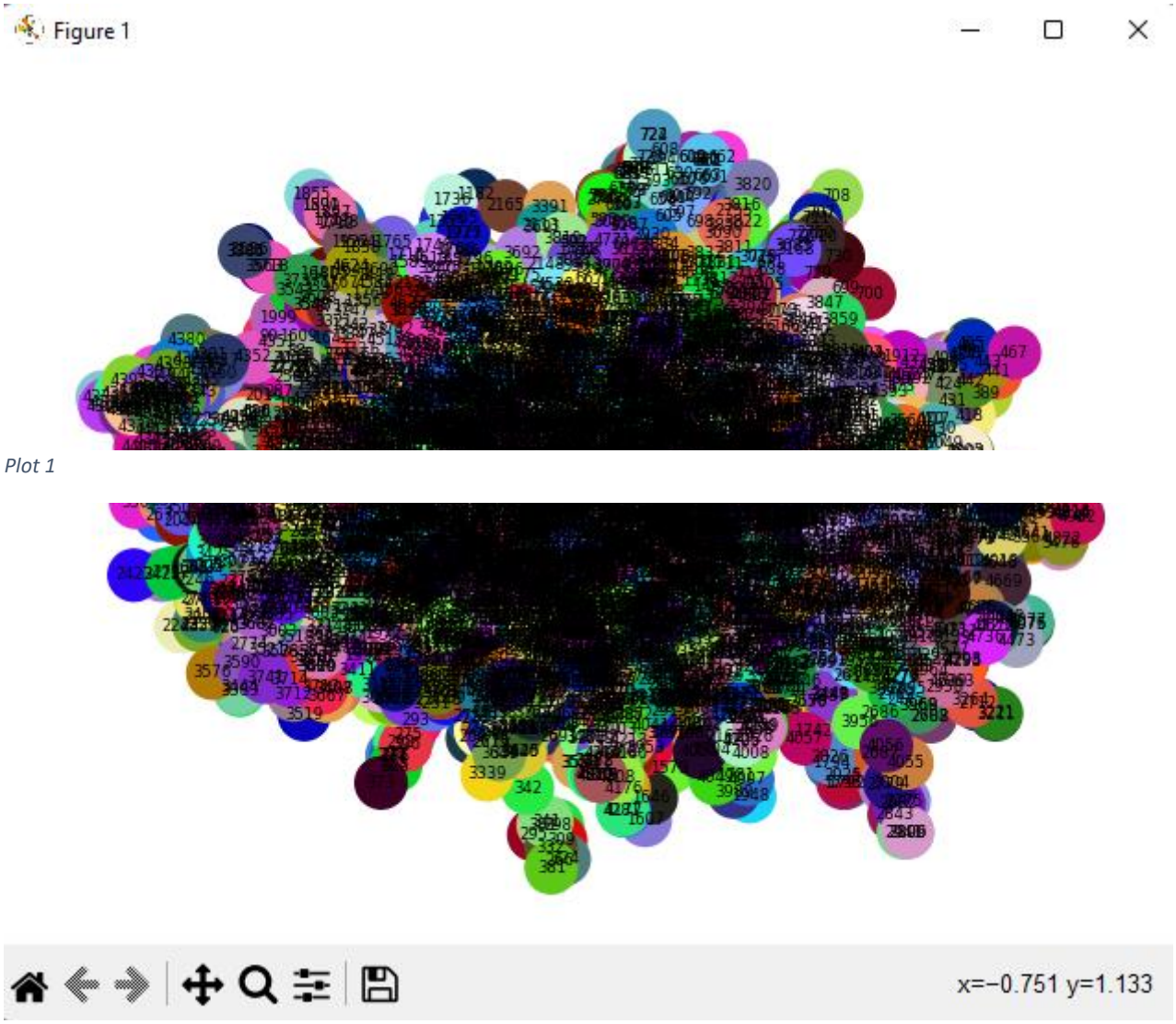
To plot the graph I used the mentioned libraries (see TOOLS USED). The algorithm is short and simple:

```
import csv
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import random
import numpy as np

data = pd.read_csv("Resources/inf-power/inf-power.csv")
# print(data)

G = nx.Graph()
for index in range(4941):
    G.add_node(index + 1)
for index, row in data.iterrows():
```

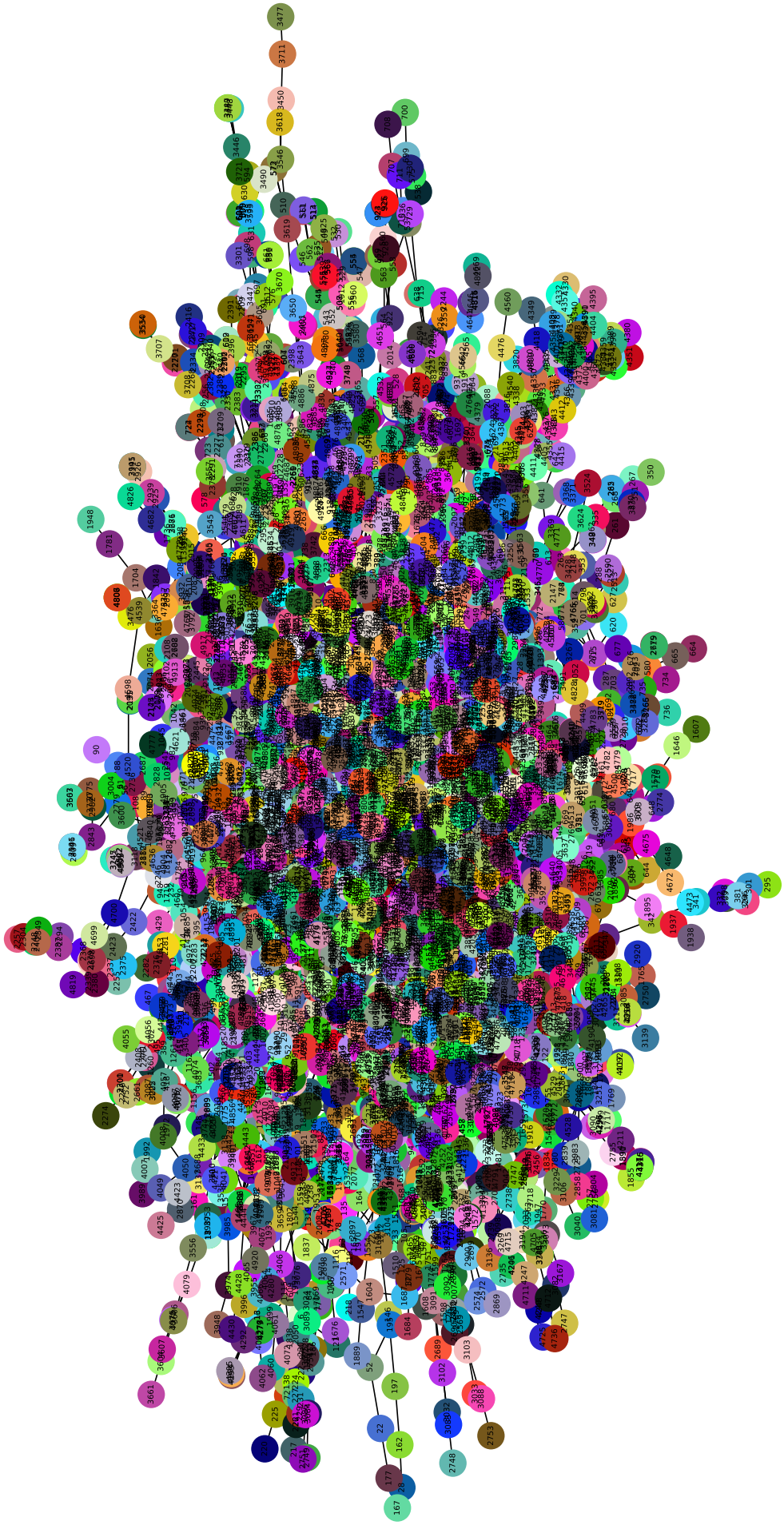
```
    G.add_edge(row[0], row[1])
node_colors = {}
for node in G.nodes():
    node_colors[node] = "#" + ''.join([random.choice('0123456789ABCDEF') for j in
range(6)])
nx.draw(G, with_labels = True, node_size = 400, node_color = list(node_colors.values()),
font_size = 6)

adjacency_matrix = pd.DataFrame(nx.adjacency_matrix(G).todense())
adjacency_matrix.to_csv('Output/adjacency_matrix.csv', index = False, header = False)

incidence_matrix = nx.incidence_matrix(G, nodelist=None, edgelist=None, oriented=False,
weight=None).todense()
df = pd.DataFrame(incidence_matrix, columns=[e for e in G.edges()])
df.insert(0, 'Node', [n for n in G.nodes()])
df.to_csv('Output/incidence_matrix.csv', index=False)

adjacency_list = nx.to_dict_of_lists(G)
with open('Output/adjacency_list.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['Node', 'Neighbors'])
    for node, neighbors in adjacency_list.items():
        writer.writerow([node] + neighbors)

plt.show()
```

First, the code reads in the CSV file and creates a graph object using the networkx library. It then assigns a unique color to each node in the graph and uses the matplotlib library to visualize the graph. The adjacency matrix and incidence matrix of the graph are computed using networkx functions and then saved to separate CSV files. Finally, the adjacency list of the graph is computed and saved to a CSV file using Python's built-in csv library. The graph visualization is displayed using matplotlib's show() function.

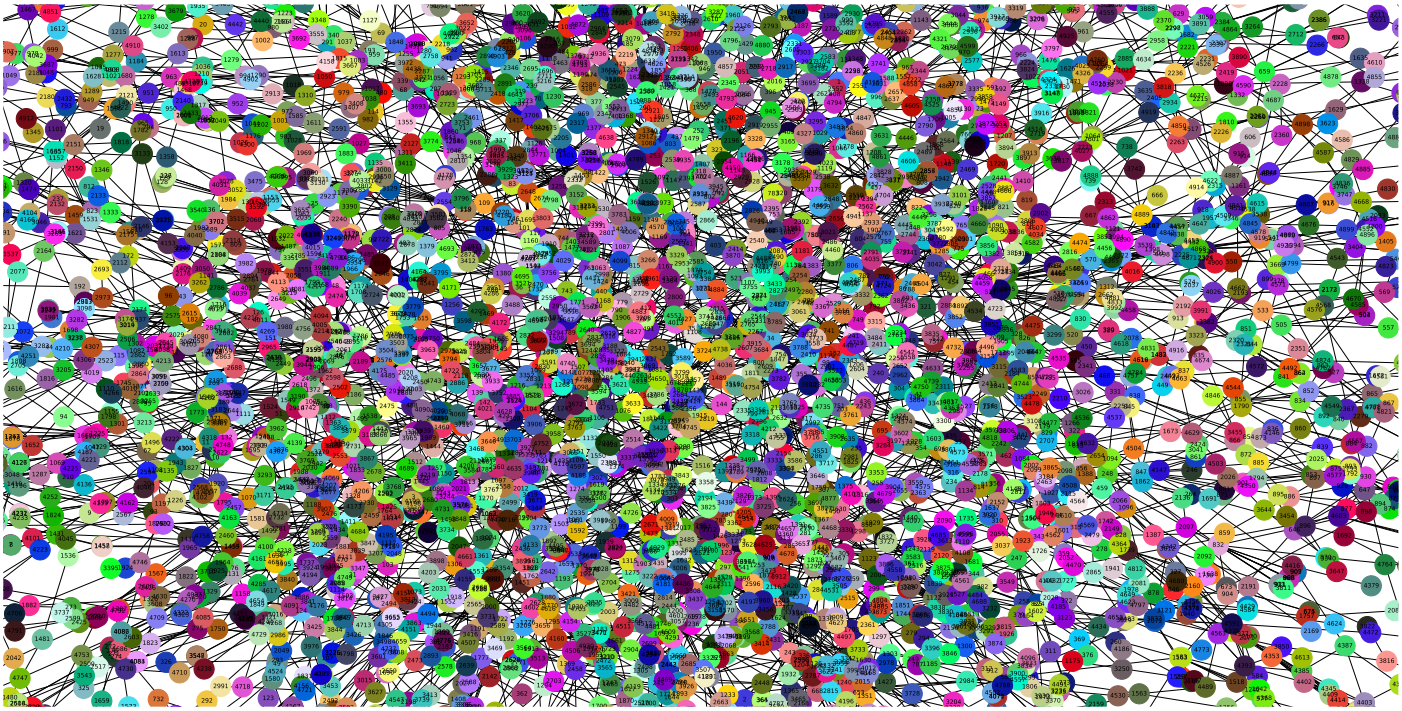
This is how the graph looks like:



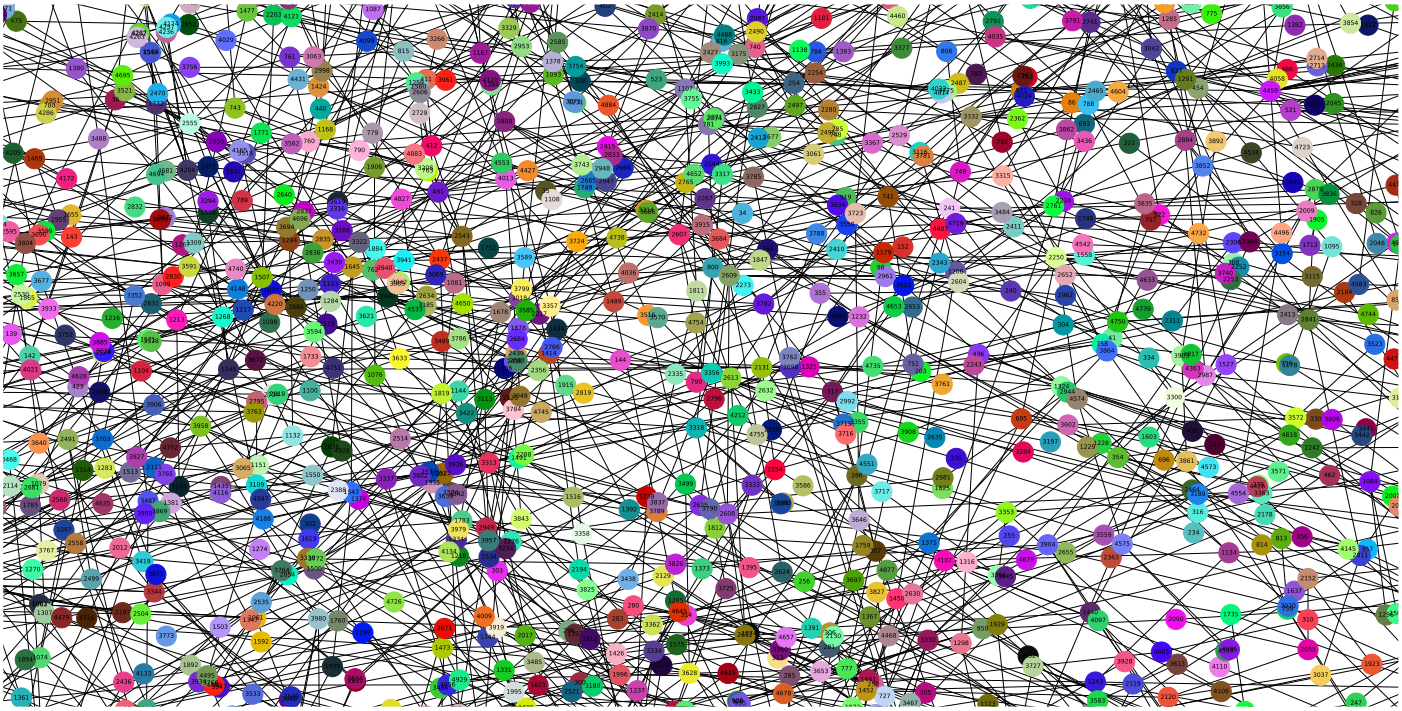
Plot 1



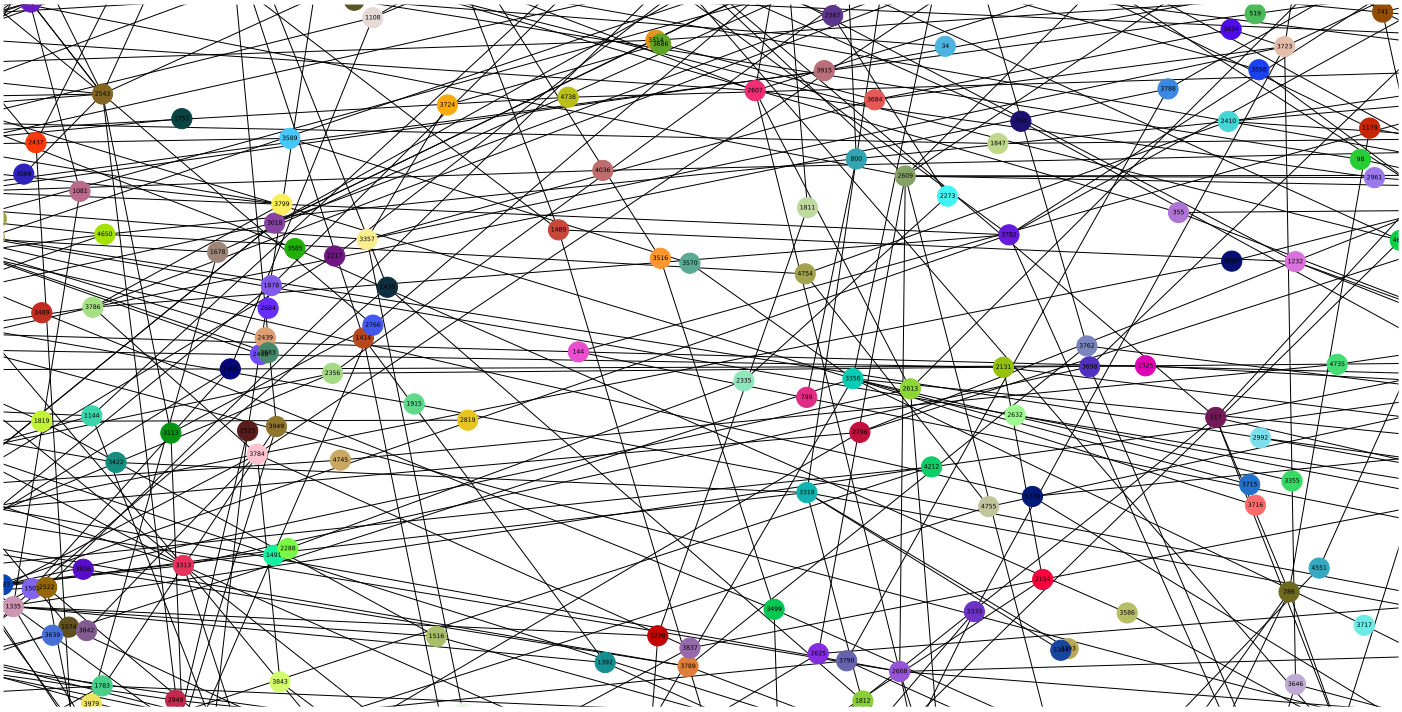
Plot 2



Plot 3



Plot 4



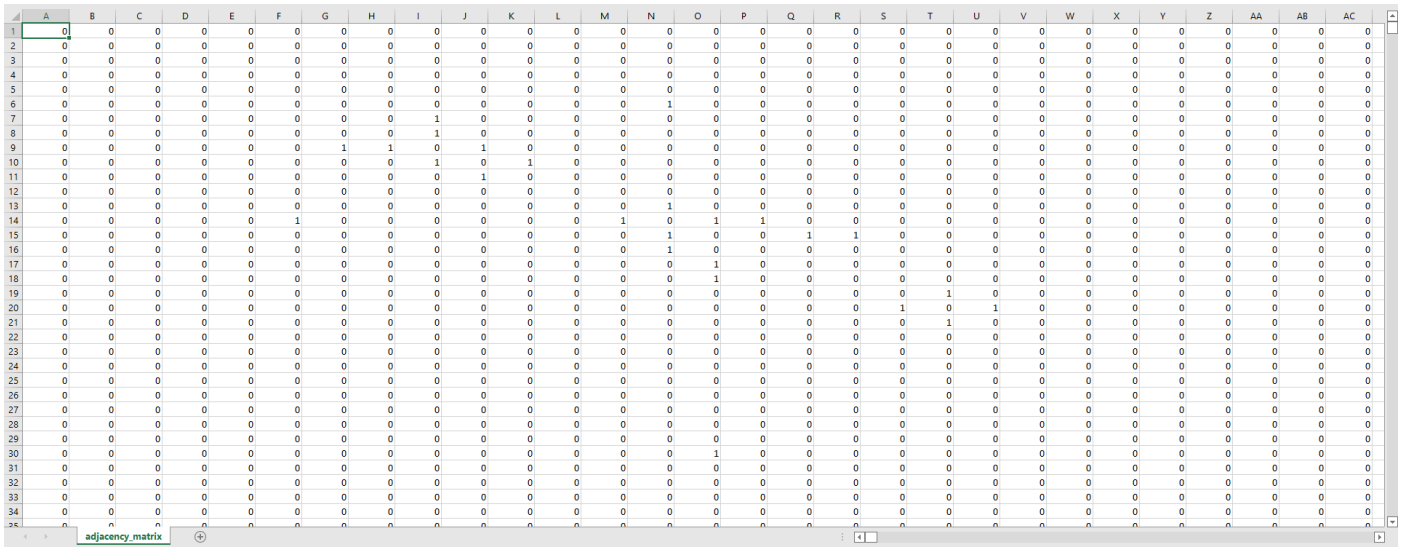
Plot 5

This code actually exports the adjacency list, adjacency matrix and incidence matrix a lot faster. Here is how they look like:

Node	Neighbors						
1	387	396	452				
2	3554	3587	3588	3638			
3	3584						
4	4931						
5	89						
6	14	121					
7	9						
8	9						
9	7	8	10				
10	9	11	62	76	206	209	
11	10	51					
12	95	126					
13	14						
14	6	13	15	16	84		
15	14	17	18	30	148		
16	14						
17	15						
18	15						
19	20	191					
20	19	21	131				
21	20	200					
22	52	177					
23	4446						
24	4447						
25	54	4462					
26	59	137					
27	95	132	140	204			
28	162	167					
29	99						
30	15	131	161				
31	124	141					
32	63	105					
33	156	175					

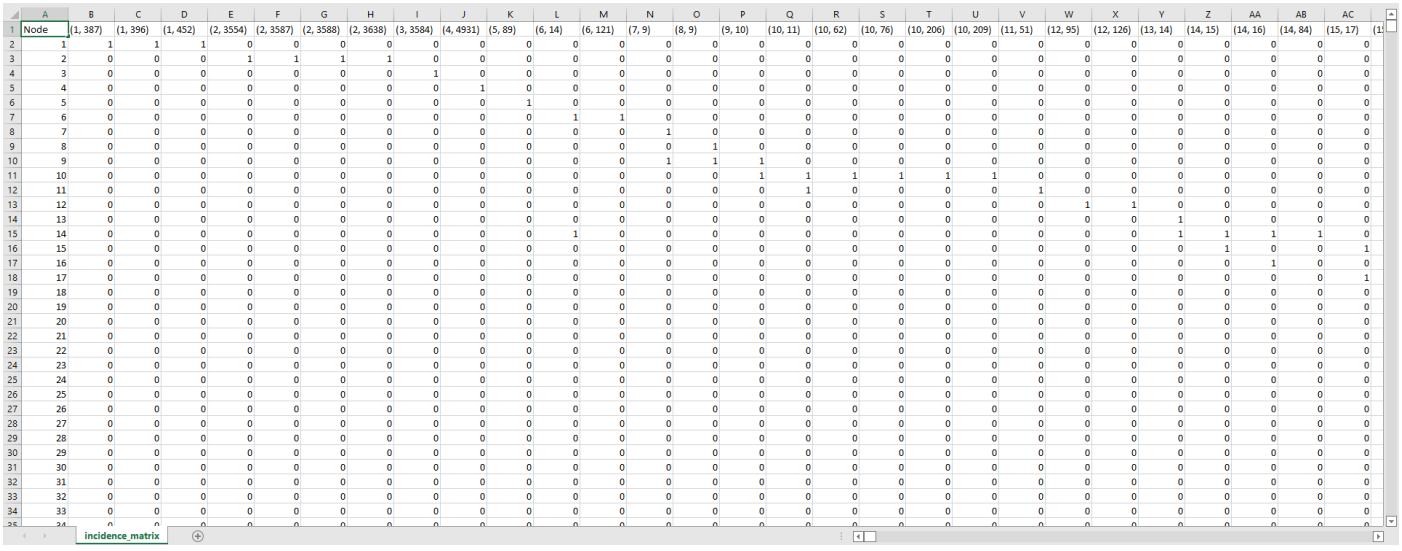
adjacency_list

1Adjacecny list



adjacency_matrix

2Adjacency matrix



incidence_matrix

3Incidence matrix

CONTENTS

FIGURES

TOOLS

NETWORK VISUALISATION

DATA