MTH5001: Introduction to Computer Programming 2018/19

Final Report Project: "Networks"

Instructions:

First, please type your name and student number into the Markdown cell below:

Name:

Student number:

You must write your answers in this Jupyter Notebook, using either Markdown or Python code as appropriate. (You should create new code and/or Markdown cells in the appropriate places, so that your answers are clearly visible.)

Your code must be well documented. As a rough guide, you should aim to include one line of comments for each line of code (but you may include more or fewer comments depending on the situation). You should also use sensible variable names, so that your code is as clear as possible. If your code works but is unduly difficult to read, then you may lose marks.

For this project, you will need to use the Python package NetworkX (https://networkx.github.io/) extensively. However, to test your coding skills, in certain questions you will be restricted to using only specific functions. These restrictions are made clear below (see questions 4 and 8).

Submission deadline:

You must submit your work via QMPlus (to the "Final Report Project" assignment in the "Final Report Project" section).

The submission deadline is **11:55pm on Monday 29 April, 2019**. Late submissions will be penalised according to the School's guidelines (https://qmplus.qmul.ac.uk/mod/book/view.php?id=807735&chapterid=89105).

Your lecturers will respond to project-related emails until 5:00pm on Friday 26 April, 2019, only. You should aim to have your project finished by this time.

Marking:

The project is worth 70% of your final mark for this module.

The total number of marks available for the project is 100.

Attempt all parts of all questions.

When writing up your project, good writing style is even more important than in written exams. According to the advice in the student handbook (https://qmplus.qmul.ac.uk/mod/book/view.php?id=807735&chapterid=87786),

> To get full marks in any assessed work (tests or exams) you must normally not only give the right answers but also explain your working clearly and give reasons for your answers by writing legible and grammatically correct English sentences. Mathematics is about logic and reasoned arguments and the only way to present a reasoned and logical argument is by writing about it clearly. Your writing may include numbers and other mathematical symbols, but they are not enough on their own. You should copy the writing style used in good mathematical textbooks, such as those recommended for your modules. **You can expect to lose marks for poor writing (incorrect grammar and spelling) as well as for poor mathematics (incorrect or unclear logic).**

Plagiarism warning:

Your work will be tested for plagiarism, which is an assessment offence, according to the School's policy on Plagiarism (https://qmplus.qmul.ac.uk/mod/book/view.php?id=807735&chapterid=87787). In particular, while only academic staff will make a judgement on whether plagiarism has occurred in a piece of work, we will use the plagiarism detection software "Turnitin" to help us assess how much of work matches other sources. You will have the opportunity to upload your work, see the Turnitin result, and edit your work accordingly before finalising your submission.

You may summarise relevant parts of books, online notes, or other resources, as you see fit. However, you must use your own words as far as possible (within reason, e.g. you would not be expected to change the wording of a well-known theorem), and you **must** reference (https://qmplus.qmul.ac.uk/mod/book/view.php?id=807735&chapterid=87793) any sources that you use. Similarly, if you decide to work with other students on parts of the project, then you **must** write up your work individually. You should also note that most of the questions are personalised in the sense that you will need to import and manipulate data that will be unique to

Background information

In this project you will learn about a field of mathematics called graph theory (https://en.wikipedia.org/wiki/Graph_theory). A **graph** (or **network**) is simply a a collection of **nodes** (or **vertices**), which may or may not be joined by **edges**. (Note that this is not the same as the 'graph' of a function.)

Graphs can represent all sorts of real-world (and, indeed, mathematical) objects, e.g.

- social networks (nodes represent people, edges represent 'friendship'),
- molecules in chemistry/physics (nodes represent atoms, edges represent bonds),
- communications networks, e.g. the internet (nodes represent computers/devices, edges represent connections).

In this project we will only consider **undirected** graphs (see the above Wikipedia link for a definition).

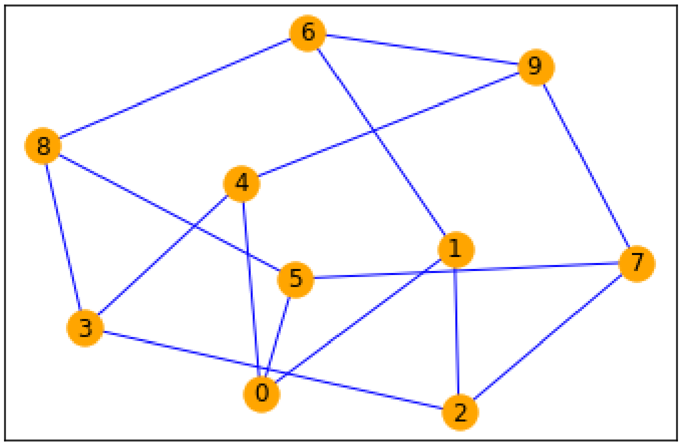
Conventiently, Python has a package, called NetworkX (https://networkx.github.io/), for constructing and analysing graphs. Let's look at an example. Below we create the famous Petersen graph (https://en.wikipedia.org/wiki/Petersen_graph) and use some basic NetworkX functions to learn a bit about it.

In [2]:
```
# import NetworkX and other useful packages
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import networkx as nx

# create the Petersen graph, storing it in a variable called "PG"
PG = nx.petersen_graph()
```

Before we doing anything else, it would make sense to draw the graph, to get an idea of what it looks like. We can do this using the NetworkX function `draw_networkx` (together with our old favourtie, matplotlib).

```
In [2]: nx.draw_networkx(PG, node_color = 'orange', edge_color = 'blue', with_la
        bels=True)
        plt.xticks([])
        plt.yticks([])
        plt.show()
```



We can see that the graph has 10 nodes, labelled by the integers $0, 1, \ldots, 9$. It is also possible to label nodes with other data types, most commonly strings, but we won't do that in this project. The nodes of a graph can be accessed via the method `nodes()` :

```
In [3]: PG.nodes()

Out[3]: NodeView((0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
```

You can convert this to a Python list if you need to:

```
In [4]: list(PG.nodes())

Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This (hopefully) makes it clear that the node labels do in fact have type `int` , at least in our example. You can also see from the picture that the graph has 15 edges. These can be accessed using the method `edges()` :

```
In [5]: PG.edges()

Out[5]: EdgeView([(0, 1), (0, 4), (0, 5), (1, 2), (1, 6), (2, 3), (2, 7), (3,
        4), (3, 8), (4, 9), (5, 7), (5, 8), (6, 8), (6, 9), (7, 9)])
```

Again, you can convert this to a list if you need to (try it), and you will see that the elements of the list are tuples. In either case, if you compare the output with the picture, it should become clear what it means, i.e. two nodes labelled i and j are joined by an edge if and only if the pair (i, j) appears in `PG.edges()` .

So far we haven't said much about how graphs are related to **mathematics**. It turns out that a graph can be completely defined by its adjacency matrix (https://en.wikipedia.org/wiki/Adjacency_matrix). This is simply a matrix A defined as follows:

- A has size $n \times n$, where n is the number of nodes in the graph;
- if the nodes labelled i and j form an edge, then the (i, j)-entry of A is 1; if they don't form an edge, the (i, j)-entry of A is 0.

This idea is the foundation of algebraic graph theory (https://en.wikipedia.org/wiki/Algebraic_graph_theory), a field of mathematics used to study graphs by analysing certain matrices.

Not surprisingly, you can compute the adjacency matrix of a graph using an appropriate NetworkX function. Let's do this for the Petersen graph:

```
In [6]: A = nx.adjacency_matrix(PG)
```

Note that if you print this 'adjacency matrix' (try it), it doesn't actually look much like a matrix. This is because it doesn't have type `numpy.ndarray` like the matrices/arrays we've worked with in class:

```
In [7]: type(nx.adjacency_matrix(PG))

Out[7]: scipy.sparse.csr.csr_matrix
```

However, you can convert it to a `numpy.ndarray` by calling the method `toarray()` :

```
In [8]: A = A.toarray()
```

```
In [9]: type(A)

Out[9]: numpy.ndarray
```

After doing this, the adjacency matrix looks like you would expect, so you can use all the usual `numpy.linalg` functions on it:

```
In [10]: print(A)

[[0 1 0 0 1 1 0 0 0 0]
 [1 0 1 0 0 0 1 0 0 0]
 [0 1 0 1 0 0 0 1 0 0]
 [0 0 1 0 1 0 0 0 1 0]
 [1 0 0 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 1 1 0]
 [0 1 0 0 0 0 0 0 1 1]
 [0 0 1 0 0 1 0 0 0 1]
 [0 0 0 1 0 1 1 0 0 0]
 [0 0 0 0 1 0 1 1 0 0]]
```

Make sure that you understand what all these 0's and 1's mean: the (i, j)-entry of the adjacency matrix is 1 if and only if the edges labelled i and j form an edge in the graph; otherwise, it is 0. For example (remembering that Python starts counting from 0, not from 1): the $(0, 4)$ entry is 1, and in the picture above we see that nodes 0 and 4 form an edge; on the other hand, the $(1, 7)$ entry is 0, and accordingly nodes 1 and 7 don't form an edge.

You will be working with matrices related to graphs quite a lot in this project, so before you begin you should make sure that you understand what the code we've given you above is doing. You may also like to work through the official NetworkX tutorial (https://networkx.github.io/documentation/stable/tutorial.html) before attempting the project, bearing in mind that not everything in the tutorial is relevant to the project. (Alternatively, Google for another tutorial if you don't like that one.)

A final remark before we get to the project itself:

You can rest assured that the graphs we consider this project all have the following nice properties:

- They are **connected**. This means that for every pair of nodes i and j, there is a 'path' of edges joining i to j. For example, the Petersen graph is connected, e.g. the nodes labelled 6 and 7 do not form an edge, but we can still reach node 7 from node 6 via the edges $(6, 9)$ and $(9, 7)$.
- They are **simple**. This means that there is never an edge from a node to itself.

You may come across these terms when you are researching the relevant mathematics for various parts of the project, so you should know what they mean.

The project

As we have already mentioned, in this project you will make extensive use of the Python package NetworkX (https://networkx.github.io/), which allows you to create and analyse graphs. You are expected to read the relevant parts of the NetworkX documentation, or otherwise learn how to use whatever Python functions you need to complete the project. However, the mini-tutorial which we have provided above should be enough to get you started.

You will also need to research and summarise some mathematics related to graphs, and to use your findings to write certain pieces of code 'from scratch', instead of of using NetworkX functions. In these cases (questions 4 and 8), it is **strongly recommended** that you use NetworkX to check your answers.

You should structure your report as follows:

Part I: Data import and preliminary investigation [10 marks]

You have been provided with a Python file called **"data.py"** on QMPlus, which you should save in the same directory as this Jupyter notebook. This file contains a function `create_graph` which constructs a random graph that you will be analysing throughout the project. By following the instructions in question 1 (below), you will create a graph that is unique to you, i.e. no two students will have the same graph.

1. [5 marks] Execute the following code cell to create your graph, storing it in a variable called `G` (you can change the variable name if you like, but we recommend leaving it as it is). You **must** replace the number "123456789" with your 9-digit student number.

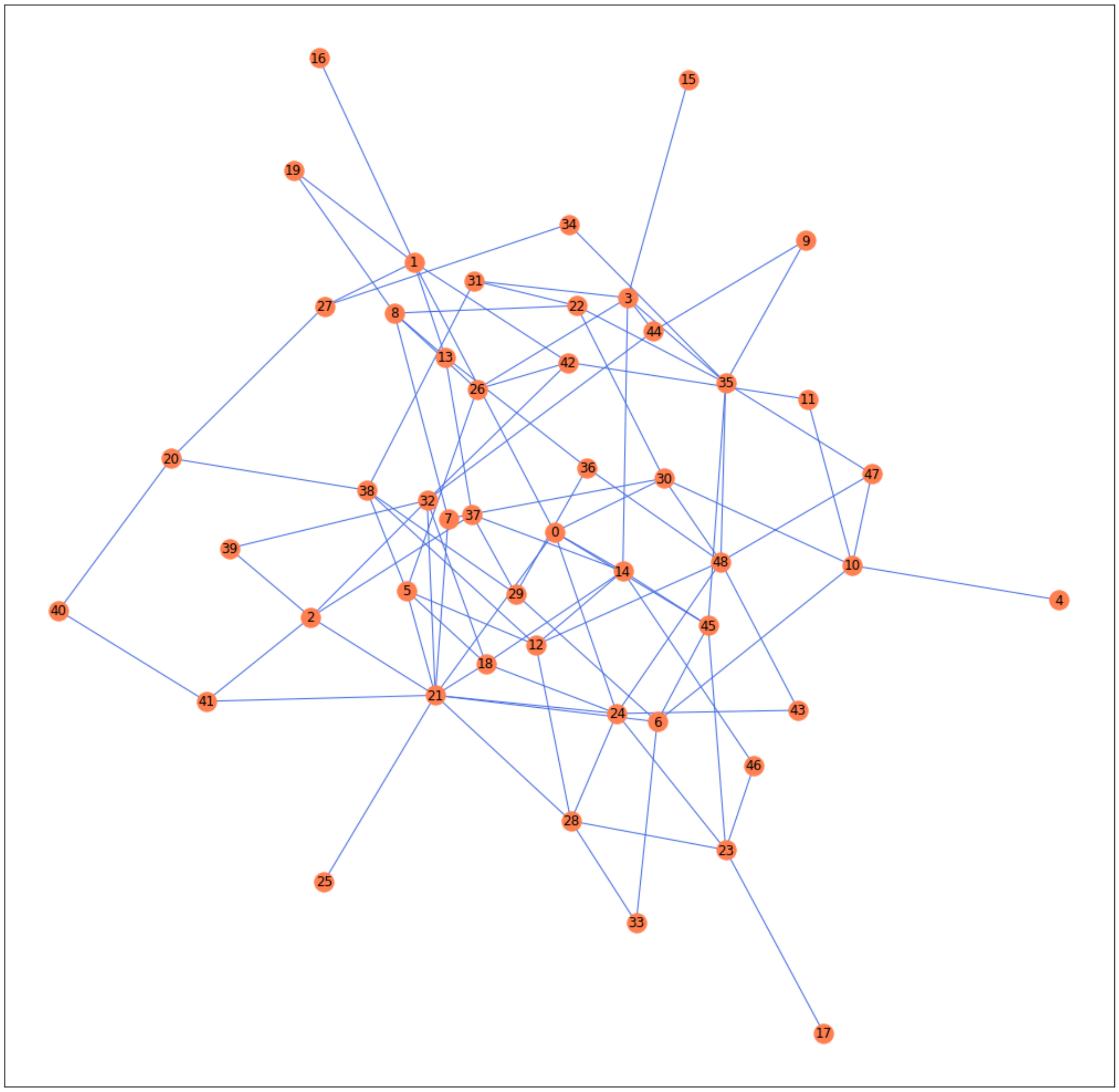
Important note: If you do not do this correctly, you will score 0 for this question, and if you are found to have used the same input as another student (rather than your individual student number), then your submission will be reviewed for plagiarism.

In [3]:
```
from data import create_graph
# import NetworkX and other useful packages
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import networkx as nx

# Replace "123456789" below with your 9-digit student number.
# My student number is "160626247".
G = create_graph(160626247)
```

2. [5 marks] Draw your graph, and calculate how many nodes and edges it has.

In [188]:
```
# Draw the graph
plt.figure(figsize=(18,18))
nx.draw_networkx(G, node_color = 'Coral', edge_color = 'RoyalBlue', with
_labels=True)
plt.xticks([])
plt.yticks([])
plt.show()
```



In [5]:
```
G.nodes()
```

Out[5]:
```
NodeView((0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48))
```

```
In [6]: list(G.nodes())
```

```
Out[6]: [0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25,
 26,
 27,
 28,
 29,
 30,
 31,
 32,
 33,
 34,
 35,
 36,
 37,
 38,
 39,
 40,
 41,
 42,
 43,
 44,
 45,
 46,
 47,
 48]
```

```
In [7]: # Calculate number of nodes
print("number of nodes is: ",G.number_of_nodes())
```

```
number of nodes is:  49
```

```
In [8]: # Calculate number of edges of this graph G.
        print("number of edges is: ",G.number_of_edges())
```

```
number of edges is:  97
```

Part II: Distance matrices and shortest paths [30 marks]

Many properties of graphs can be analysed by using matrices/linear algebra. The rest of your report will involve researching/summarising some of the relevant mathematics, and writing/using Python code to analyse certain properties of your graph from Part I. As explained above, you are allowed to summarise information from books and/or web pages, but you must use your own words and clearly reference any sources you use.

3. [10 marks] Explain what a "path" between two nodes in a graph is, and what the "distance" between two nodes is. Explain also what the "distance matrix" of a graph is, and how it can be computed using the adjacency matrix. Here you should discuss arbitrary (undirected, simple, connected) graphs, not your specific graph from Part I.

Note: You do **not** need to give any proofs, but you must reference any material you use, as explained in the plagiarism warning above.

Answer:

- 3.1 path:
  - a path between two nodes in a graph:
    - Generally, a path between two nodes in a graph could be considered as a way to get to a node from another node, by passing through other nodes and edges linked them together.
    - in undirected graphs, a path between two nodes could be understood as a double-ended linked list of the nodes connected by edges, which could link these two nodes together, where one node is the start point and the other is the end point, and vice verse.
    - in connected graphs, a path between two nodes could be understood as a single direction linked list of nodes, connected by directed edges.
- 3.2 distance matrix
  - In graph theory, a distance matrix is a two-dimensioanl array that contains the pairwise distances between any two connected nodes in a graph, which primary designed to show the distance between a pair of two directly connected nodes.

4. [10 marks] Write a function `distance_matrix` which computes the distance matrix of a graph. Your function should return a matrix, represented as an array of type `numpy.ndarray` , of the same shape as the adjacency matrix of the graph. You may use the NetworkX function `adjacency_matrix` to compute the adjacency matrix of the input graph, but you **must not use any other NetworkX functions**.

In [9]:
```
def getMinDistance(dist, queue):
    # Initialize the minimum value and its index
    min_val = float("Inf")
    min_idx = -1
    length = len(dist)
    # get the minimum value from the distance array
    for i in range(length):
        # update the min_val and min_idx
        if (dist[i] < min_val and i in queue):
            min_val = dist[i]
            min_idx = i
    return min_idx
```

In [10]:
```
def dijkstraAlg(adj_matrix, src):
    row = len(adj_matrix)
    col = len(adj_matrix[0])

    # distanc array: dist[i] is the shortest distance from the src node
 to i.
    # initialize the distances to be INFINTE.
    dist = [float("Inf")] * col
    parent = [-1] * col

    # the distance from the src node to itself is 0.
    dist[src] = 0

    queue = []
    for i in range(col):
        # add all nodes in queue.
        queue.append(i)

    # Find the shortest path for all the nodes in the graph.
    while queue:
        # get index of the min_val from the nodes in the queue.
        idx = getMinDistance(dist, queue)
        # remove the min element from the queue.
        queue.remove(idx)

        # update
        for i in range(col):
            if (adj_matrix[idx][i] and i in queue):
                if (dist[idx] + adj_matrix[idx][i] < dist[i]):
                    dist[i] = dist[idx] + adj_matrix[idx][i]
                    parent[i] = idx
    return dist
```

In [11]:
```
def computeDistMat(graph):
    adj_matrix = nx.adjacency_matrix(graph)
    adj_matrix = adj_matrix.toarray()

    row = len(adj_matrix)
    col = len(adj_matrix[0])

    dist_matrix = [[float("Inf")] * col] * row

    for i in range(row):
        row_dist = dijkstraAlg(adj_matrix,i)
        dist_matrix[i] = row_dist

    return dist_matrix
```

In [12]:
```
dist_matrix = computeDistMat(G)
print(dist_matrix)
```

```
[[0, 1, 2, 2, 3, 2, 2, 2, 3, 3, 2, 3, 2, 2, 1, 3, 2, 3, 2, 2, 3, 1, 2,
2, 1, 2, 3, 2, 2, 3, 1, 3, 2, 3, 3, 2, 3, 2, 3, 3, 3, 2, 2, 2, 3, 1, 2,
3, 2], [1, 0, 3, 3, 4, 3, 3, 3, 2, 4, 3, 2, 3, 1, 2, 4, 1, 4, 3, 1, 2,
2, 3, 3, 2, 3, 2, 1, 3, 3, 2, 4, 2, 4, 2, 3, 2, 2, 3, 3, 3, 3, 1, 3, 3,
2, 3, 4, 3], [2, 3, 0, 3, 4, 2, 2, 2, 3, 3, 3, 3, 3, 2, 2, 4, 4, 4, 2,
4, 3, 1, 4, 3, 2, 2, 3, 4, 2, 2, 3, 4, 1, 3, 5, 4, 3, 1, 3, 1, 2, 1, 2,
3, 2, 3, 3, 4, 3], [2, 3, 3, 0, 4, 2, 3, 3, 2, 2, 3, 3, 2, 3, 1, 1, 4,
4, 2, 3, 3, 3, 2, 3, 3, 4, 1, 3, 3, 3, 3, 1, 2, 4, 2, 1, 3, 2, 2, 3, 4,
4, 2, 3, 1, 2, 2, 2, 2], [3, 4, 4, 4, 0, 4, 2, 3, 4, 4, 1, 2, 4, 5, 4,
5, 5, 5, 4, 5, 5, 3, 3, 4, 4, 4, 4, 5, 4, 3, 2, 4, 4, 3, 4, 3, 4, 4, 4,
5, 5, 4, 3, 4, 5, 3, 5, 2, 3], [2, 3, 2, 2, 4, 0, 2, 2, 2, 4, 3, 3, 1,
3, 2, 3, 4, 4, 1, 3, 2, 1, 3, 3, 2, 2, 1, 3, 2, 2, 3, 2, 2, 3, 4, 3, 3,
3, 1, 3, 3, 2, 2, 3, 3, 3, 3, 3, 2], [2, 3, 2, 3, 2, 2, 0, 2, 3, 3, 1,
2, 3, 3, 2, 4, 4, 3, 2, 4, 3, 1, 3, 2, 2, 2, 3, 4, 2, 1, 2, 3, 2, 1, 3,
2, 2, 2, 2, 3, 3, 2, 3, 3, 3, 1, 3, 2, 3], [2, 3, 2, 3, 3, 2, 2, 0, 1,
4, 2, 3, 3, 2, 3, 4, 4, 4, 2, 2, 4, 1, 2, 3, 2, 2, 2, 4, 2, 3, 1, 3, 2,
3, 4, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3, 4, 3, 2], [3, 2, 3, 2, 4, 2, 3,
1, 0, 3, 3, 3, 3, 1, 3, 3, 3, 5, 3, 1, 4, 2, 1, 4, 3, 3, 1, 3, 3, 3, 2,
2, 3, 4, 3, 2, 2, 2, 3, 4, 4, 3, 2, 4, 3, 3, 4, 3, 3], [3, 4, 3, 2, 4,
4, 3, 4, 3, 0, 3, 4, 3, 4, 3, 3, 5, 4, 3, 4, 4, 3, 2, 3, 3, 4, 3, 3, 4,
4, 3, 3, 2, 4, 2, 1, 3, 4, 4, 3, 5, 4, 3, 3, 1, 2, 4, 2, 2], [2, 3, 3,
3, 1, 3, 1, 2, 3, 3, 0, 1, 3, 4, 3, 4, 4, 4, 3, 4, 4, 2, 2, 3, 3, 3, 3,
4, 3, 2, 1, 3, 3, 2, 3, 2, 3, 3, 3, 4, 4, 3, 2, 3, 4, 2, 4, 1, 2], [3,
2, 3, 3, 2, 3, 2, 3, 3, 4, 1, 0, 4, 3, 4, 4, 3, 5, 3, 3, 4, 3, 3, 4, 4,
4, 2, 3, 4, 3, 2, 4, 2, 3, 4, 3, 4, 4, 4, 3, 5, 4, 1, 4, 3, 3, 5, 2,
3], [2, 3, 3, 2, 4, 1, 3, 3, 3, 3, 3, 4, 0, 3, 1, 3, 4, 3, 2, 4, 2, 2,
3, 2, 2, 3, 2, 3, 1, 2, 2, 2, 3, 2, 3, 2, 2, 2, 1, 4, 3, 3, 3, 2, 3, 2,
2, 2, 1], [2, 1, 2, 3, 5, 3, 3, 2, 1, 4, 4, 3, 3, 0, 2, 4, 2, 5, 3, 2,
3, 3, 2, 4, 3, 4, 2, 2, 4, 2, 3, 3, 3, 4, 3, 3, 1, 1, 3, 3, 4, 3, 2, 3,
4, 3, 3, 3, 2], [1, 2, 2, 1, 4, 2, 2, 3, 3, 3, 3, 4, 1, 2, 0, 2, 3, 3,
1, 3, 3, 2, 3, 2, 2, 3, 2, 3, 2, 2, 2, 2, 2, 3, 3, 2, 3, 1, 2, 3, 4, 3,
3, 3, 2, 1, 1, 3, 2], [3, 4, 4, 1, 5, 3, 4, 4, 3, 3, 4, 4, 3, 4, 2, 0,
5, 5, 3, 4, 4, 4, 3, 4, 4, 5, 2, 4, 4, 4, 4, 2, 3, 5, 3, 2, 4, 3, 3, 4,
5, 5, 3, 4, 2, 3, 3, 3, 3], [2, 1, 4, 4, 5, 4, 4, 4, 3, 5, 4, 3, 4, 2,
3, 5, 0, 5, 4, 2, 3, 3, 4, 4, 3, 4, 3, 2, 4, 4, 3, 5, 3, 5, 3, 4, 3, 3,
4, 4, 4, 4, 2, 4, 4, 3, 4, 5, 4], [3, 4, 4, 4, 5, 4, 3, 4, 5, 4, 4, 5,
3, 5, 3, 5, 5, 0, 3, 5, 5, 3, 4, 1, 2, 4, 5, 5, 2, 4, 4, 5, 4, 3, 4, 3,
4, 4, 4, 5, 5, 4, 5, 3, 5, 2, 2, 4, 3], [2, 3, 2, 2, 4, 1, 2, 2, 3, 3,
3, 3, 2, 3, 1, 3, 4, 3, 0, 4, 3, 1, 4, 2, 1, 2, 2, 4, 2, 3, 3, 3, 1, 3,
4, 3, 3, 2, 2, 2, 3, 2, 2, 2, 2, 2, 3, 2], [2, 1, 4, 3, 5, 3, 4, 2,
1, 4, 4, 3, 4, 2, 3, 4, 2, 5, 4, 0, 3, 3, 2, 4, 3, 4, 2, 2, 4, 4, 3, 3,
3, 5, 3, 3, 3, 3, 4, 4, 4, 4, 2, 4, 4, 3, 4, 4, 4], [3, 2, 3, 3, 5, 2,
3, 4, 4, 4, 4, 4, 2, 3, 3, 4, 3, 5, 3, 3, 0, 3, 3, 4, 4, 4, 3, 1, 3, 2,
4, 2, 4, 4, 2, 3, 3, 3, 1, 4, 1, 2, 3, 4, 4, 4, 4, 4, 3], [1, 2, 1, 3,
3, 1, 1, 1, 2, 3, 2, 3, 2, 3, 2, 4, 3, 3, 1, 3, 3, 0, 3, 2, 1, 1, 2, 3,
1, 2, 2, 3, 1, 2, 4, 3, 3, 2, 2, 2, 2, 1, 2, 2, 2, 2, 3, 3, 2], [2, 3,
4, 2, 3, 3, 3, 2, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 4, 2, 3, 3, 0, 3, 3, 4,
2, 3, 4, 3, 1, 1, 4, 4, 2, 1, 3, 3, 2, 5, 4, 4, 3, 3, 3, 2, 4, 2, 2],
[2, 3, 3, 3, 4, 3, 2, 3, 4, 3, 3, 4, 2, 4, 2, 4, 4, 1, 2, 4, 4, 2, 3,
0, 1, 3, 4, 4, 1, 3, 3, 4, 3, 2, 3, 2, 3, 3, 3, 4, 4, 3, 4, 2, 4, 1, 1,
3, 2], [1, 2, 2, 3, 4, 2, 2, 2, 3, 3, 3, 4, 2, 3, 2, 4, 3, 2, 1, 3, 4,
1, 3, 1, 0, 2, 3, 3, 1, 3, 2, 4, 2, 2, 3, 2, 2, 3, 3, 3, 3, 2, 3, 1, 3,
2, 2, 2, 1], [2, 3, 2, 4, 4, 2, 2, 2, 3, 4, 3, 4, 3, 4, 3, 5, 4, 4, 2,
4, 4, 1, 4, 3, 2, 0, 3, 4, 2, 3, 3, 4, 2, 3, 5, 4, 4, 3, 3, 3, 3, 2, 3,
3, 3, 3, 4, 4, 3], [3, 2, 3, 1, 4, 1, 3, 2, 1, 3, 3, 2, 2, 2, 2, 2, 3,
5, 2, 2, 3, 2, 2, 4, 3, 3, 0, 3, 3, 3, 3, 2, 2, 4, 3, 2, 3, 3, 2, 3, 4,
3, 1, 4, 2, 3, 3, 3, 3], [2, 1, 4, 3, 5, 3, 4, 4, 3, 3, 4, 3, 3, 2, 3,
4, 2, 5, 4, 2, 1, 3, 3, 4, 3, 4, 3, 0, 4, 3, 3, 3, 3, 5, 1, 2, 3, 3, 2,
4, 2, 3, 2, 4, 4, 3, 4, 3, 3], [2, 3, 2, 3, 4, 2, 2, 2, 3, 4, 3, 4, 1,
4, 2, 4, 4, 2, 2, 4, 3, 1, 4, 1, 1, 2, 3, 4, 0, 3, 3, 3, 2, 1, 4, 3, 3,
3, 2, 3, 3, 2, 3, 2, 3, 2, 2, 3, 2], [3, 3, 2, 3, 3, 2, 1, 3, 3, 4, 2,
3, 2, 2, 2, 4, 4, 4, 3, 4, 2, 2, 3, 3, 3, 3, 3, 3, 3, 0, 3, 2, 3, 2, 4,
3, 1, 1, 1, 3, 3, 3, 4, 3, 4, 2, 3, 3, 2], [1, 2, 3, 3, 2, 3, 2, 1, 2,
3, 1, 2, 2, 3, 2, 4, 3, 4, 3, 3, 4, 2, 1, 3, 2, 3, 3, 3, 3, 3, 0, 2, 3,
3, 3, 2, 2, 3, 3, 4, 4, 3, 3, 2, 4, 2, 3, 2, 1], [3, 4, 4, 1, 4, 2, 3,
3, 2, 3, 3, 4, 2, 3, 2, 2, 5, 5, 3, 3, 2, 3, 1, 4, 4, 4, 2, 3, 3, 2, 2,
0, 3, 4, 3, 2, 3, 3, 1, 4, 3, 4, 3, 4, 2, 3, 3, 3, 3], [2, 2, 1, 2, 4,
2, 2, 2, 3, 2, 3, 2, 3, 3, 2, 3, 3, 4, 1, 3, 4, 1, 4, 3, 2, 2, 2, 3, 2,
3, 3, 3, 0, 3, 4, 3, 4, 2, 3, 1, 3, 2, 1, 3, 1, 3, 3, 4, 3], [3, 4, 3,
4, 3, 3, 1, 3, 4, 4, 2, 3, 2, 4, 3, 5, 5, 3, 3, 5, 4, 2, 4, 2, 2, 3, 4,
5, 1, 2, 3, 4, 3, 0, 4, 3, 3, 3, 3, 4, 4, 3, 4, 3, 4, 2, 3, 3, 3], [3,
2, 5, 2, 4, 4, 3, 4, 3, 2, 3, 4, 3, 3, 3, 3, 3, 4, 4, 3, 2, 4, 2, 3, 3,
5, 3, 1, 4, 4, 3, 3, 4, 4, 0, 1, 3, 4, 3, 5, 3, 4, 3, 3, 3, 2, 4, 2,
2], [2, 3, 4, 1, 3, 3, 2, 3, 2, 1, 2, 3, 2, 3, 2, 2, 4, 3, 3, 3, 3, 3,
1, 2, 2, 4, 2, 2, 3, 3, 2, 2, 3, 3, 1, 0, 2, 3, 3, 4, 4, 4, 3, 2, 2, 1,
3, 1, 1], [3, 2, 3, 3, 4, 3, 2, 3, 2, 3, 3, 4, 2, 1, 3, 4, 3, 4, 3, 3,
3, 3, 3, 3, 2, 4, 3, 3, 3, 1, 2, 3, 4, 3, 3, 2, 0, 2, 2, 4, 4, 4, 3, 2,
4, 3, 4, 2, 1], [2, 2, 1, 2, 4, 3, 2, 3, 2, 4, 3, 4, 2, 1, 1, 3, 3, 4,
2, 3, 3, 2, 3, 3, 3, 3, 3, 3, 3, 1, 3, 3, 2, 3, 4, 3, 2, 0, 2, 2, 3, 2,
3, 4, 3, 2, 2, 4, 3], [3, 3, 3, 2, 4, 1, 2, 3, 3, 4, 3, 4, 1, 3, 2, 3,
4, 4, 2, 4, 1, 2, 2, 3, 3, 3, 2, 2, 2, 1, 3, 1, 3, 3, 3, 3, 2, 2, 0, 4,
2, 3, 3, 3, 3, 3, 3, 3, 2], [3, 3, 1, 3, 5, 3, 3, 3, 4, 3, 4, 3, 4, 3,
3, 4, 4, 5, 2, 4, 4, 2, 5, 4, 3, 3, 3, 4, 3, 3, 4, 4, 1, 4, 5, 4, 4, 2,
4, 0, 3, 2, 2, 4, 2, 4, 4, 5, 4], [3, 3, 2, 4, 5, 3, 3, 3, 4, 5, 4, 5,
3, 4, 4, 5, 4, 5, 3, 4, 1, 2, 4, 4, 3, 3, 4, 2, 3, 3, 4, 3, 3, 4, 3, 4,
4, 3, 2, 3, 0, 1, 4, 4, 4, 4, 5, 5, 4], [2, 3, 1, 4, 4, 2, 2, 2, 3, 4,
3, 4, 3, 3, 3, 5, 4, 4, 2, 4, 2, 1, 4, 3, 2, 2, 3, 3, 2, 3, 3, 4, 2, 3,
4, 4, 4, 2, 3, 2, 1, 0, 3, 3, 3, 3, 4, 4, 3], [2, 1, 2, 2, 3, 2, 3, 3,
2, 3, 2, 1, 3, 2, 3, 3, 2, 5, 2, 2, 3, 2, 3, 4, 3, 3, 1, 2, 3, 4, 3, 3,
1, 4, 3, 3, 3, 3, 3, 2, 4, 3, 0, 4, 2, 3, 4, 3, 4], [2, 3, 3, 3, 4, 3,
3, 3, 4, 3, 3, 4, 2, 3, 3, 4, 4, 3, 2, 4, 4, 2, 3, 2, 1, 3, 4, 4, 2, 3,
2, 4, 3, 3, 3, 2, 2, 4, 3, 4, 4, 3, 4, 0, 4, 3, 3, 2, 1], [3, 3, 2, 1,
5, 3, 3, 3, 3, 1, 4, 3, 3, 4, 2, 2, 4, 5, 2, 4, 4, 2, 3, 4, 3, 3, 2, 4,
3, 4, 4, 2, 1, 4, 3, 2, 4, 3, 3, 2, 4, 3, 2, 4, 0, 3, 3, 3, 3], [1, 2,
3, 2, 3, 3, 1, 3, 3, 2, 2, 3, 2, 3, 1, 3, 3, 2, 2, 3, 4, 2, 2, 1, 2, 3,
3, 3, 2, 2, 2, 3, 3, 2, 2, 1, 3, 2, 3, 4, 4, 3, 3, 3, 3, 0, 2, 2, 2],
[2, 3, 3, 2, 5, 3, 3, 4, 4, 4, 4, 5, 2, 3, 1, 3, 4, 2, 2, 4, 4, 3, 4,
1, 2, 4, 3, 4, 2, 3, 3, 3, 3, 3, 4, 3, 4, 2, 3, 4, 5, 4, 4, 3, 3, 2, 0,
4, 3], [3, 4, 4, 2, 2, 3, 2, 3, 3, 2, 1, 2, 2, 3, 3, 3, 5, 4, 3, 4, 4,
3, 2, 3, 2, 4, 3, 3, 3, 3, 2, 3, 4, 3, 2, 1, 2, 4, 3, 5, 5, 4, 3, 2, 3,
2, 4, 0, 1], [2, 3, 3, 2, 3, 2, 3, 2, 3, 2, 2, 3, 1, 2, 2, 3, 4, 3, 2,
4, 3, 2, 2, 2, 1, 3, 3, 3, 2, 2, 1, 3, 3, 3, 2, 1, 1, 3, 2, 4, 4, 3, 4,
1, 3, 2, 3, 1, 0]]
```

5. [5 marks] Using your function from Question 4, find a pair of nodes (i, j) in your graph from Part I with the property that the distance from i to j is maximal amongst all pairs of nodes in the graph.

Note: This means that for every *other* pair of nodes (i', j'), the distance from i' to j' is less than or equal to the distance from i to j.

```
In [13]: def computeMaxDistPair(dist_matrix):
             row = len(dist_matrix)
             col = len(dist_matrix[0])
             # Initialize the pair of nodes(i,j) index as [0,0]
             max_pair = [0]*2
             max_val = dist_matrix[0][0]

             for i in range(row):
                 for j in range(i, col):
                     if dist_matrix[i][j] > max_val:
                         max_val = dist_matrix[i][j]
                         max_pair = [i, j]
             return max_val, max_pair
```

```
In [14]: max_val = computeMaxDistPair(dist_matrix)[0]
         max_pair = computeMaxDistPair(dist_matrix)[1]
         print("The maximal value amongst all pairs of nodes is", max_val, ",\nth
         e index of nodes is : ", max_pair)
```

```
The maximal value amongst all pairs of nodes is 5 ,
the index of nodes is :  [2, 34]
```

6. [5 marks] Find a shortest path between your nodes from Question 5, i.e. one with the shortest possible length, and re-draw your graph so that this path is clearly visible. You should use one colour for the nodes and edges in the path, and a different colour for all other nodes and edges.

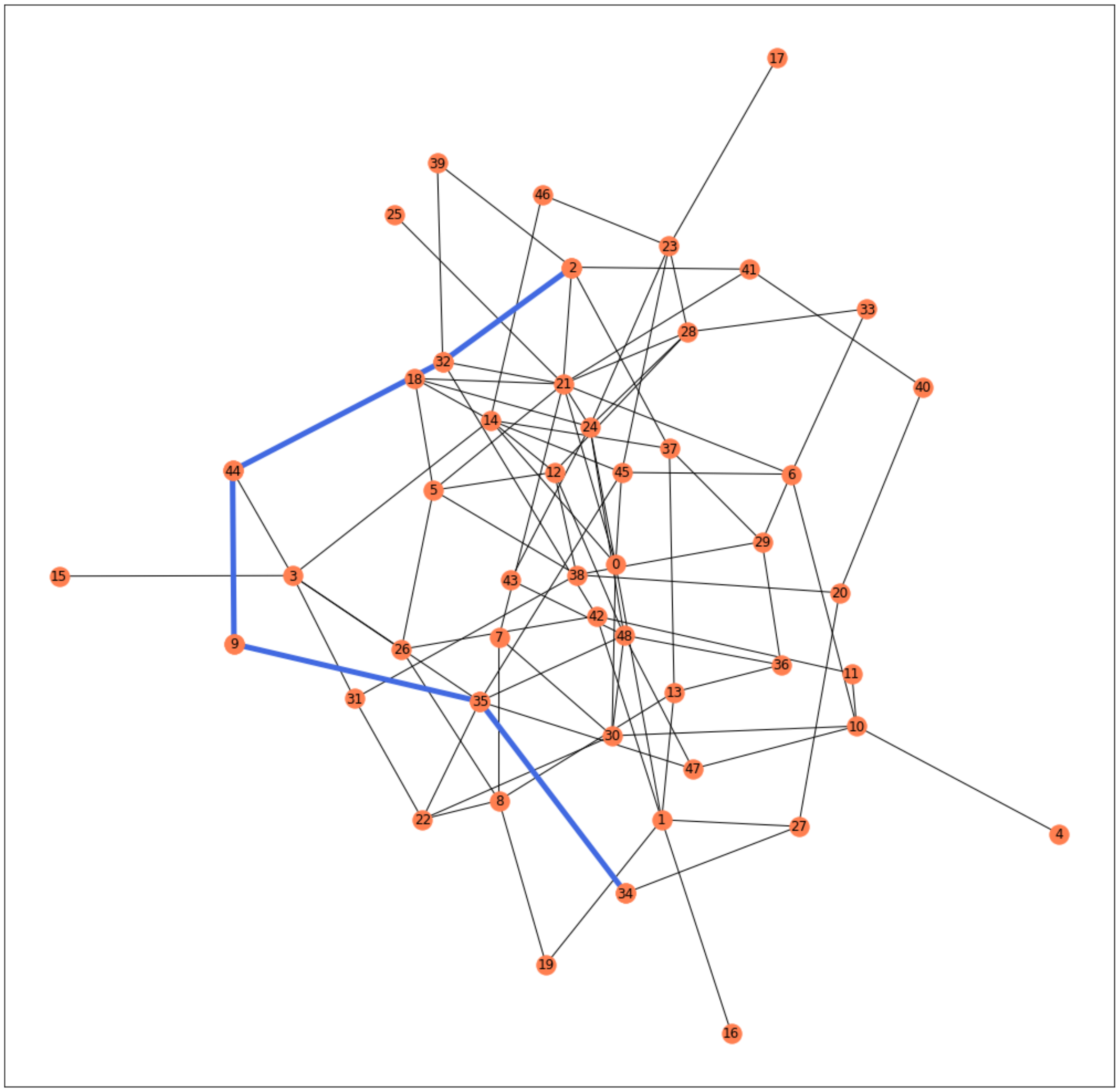
Hint: You should be able to find a NetworkX function that computes a shortest path.

```
In [15]: def printShortestPath(G):
             shortest_paths =nx.shortest_path(G, source=2, target=34)
             return shortest_paths
```

In [196]:
```
shortest_paths = printShortestPath(G)
print(shortest_paths)

pos = nx.spring_layout(G)
plt.figure(figsize=(18,18))
# nx.draw(G,pos,node_color='k')
nx.draw_networkx(G, pos, node_color = 'Coral', edge_color = 'black', wit
h_labels=True)
path = nx.shortest_path(G,source=2,target=34)
path_edges = zip(path,path[1:])
path_edges = set(path_edges)
nx.draw_networkx_nodes(G,pos,nodelist=path,node_color='Coral')
nx.draw_networkx_edges(G,pos,edgelist=path_edges,edge_color='RoyalBlue',
width=5)
plt.xticks([])
plt.yticks([])
plt.show()
```

```
[2, 32, 44, 9, 35, 34]
```



Part III: Laplacian matrices and spanning trees [30 marks]

So far you have learned about two matrices associated with a graph: the adjacency matrix, and the distance matrix. Now you will study a third matrix: the Laplacian.

7. [10 marks] Explain what the "degree" of a node in a graph is, what the "Laplacian matrix" of a graph is, what a "spanning tree" of a graph is, and how the Laplacian matrix can be used to calculate the number of spanning trees in a graph. Here, again, you should discuss arbitrary (undirected, simple, connected) graphs, not your specific graph from Part I.

Note: You do **not** need to give any proofs, but you must reference any material you use, as explained in the plagiarism warning above.

Answer:

- the "degree" of a node in a graph:
  - the degree of a node in a graph is the number of edges connected to this node. Usually, we count the number of edges which has the node as an endpoint. In addition, the degree of a graph is the largest node degree among all the nodes in the graph.
- the "Laplacian matrix" of a graph:
  - For a simple graph G with n nodes, the Laplacian matrix of this graph is L = D - A where D is the degree matrix and A is the adjacency matrix of this graph.
  - For undirected, unweighted graph with n nodes, without loop from one node to itself or multiple edges from one node to another node, the Laplacian matrix of this graph is L = D - A.
  - Briefly, the "Laplacian matrix" of a graph is equal to the difference between the degree matrix of this graph and its adjacency matrix.
- a "spanning tree" of a graph:
  - a "spanning tree" of a graph G could be considered as a subset of G, that is a tree, which contains all the nodes of G, where all of the nodes are covered with minimum number of edges. a spanning tree does not have loops and it cannot be disconnected. A graph might have several spanning trees (the graph should not be connected in order to have a spanning tree.)
- how the Laplacian matrix can be used to calculate the number of spanning trees in a graph
  - According to Kirchhoff's theorem which relates the number of spanning trees of a graph with its Laplacian matrix, the number of spanning trees of a graph G is equal to any cofactor of the Laplacian matrix of G. Represents as:

In [19]:
```
from IPython.display import Latex
Latex(r"$t(G) = \frac{1}{n}* \lambda_1 \lambda_2 ... \lambda_{n-1}$")
```

Out[19]: $t(G) = \frac{1}{n}* \lambda_1 \lambda_2 ... \lambda_{n-1}$

where λ_1, λ_2 ,..., λ_{n-1} are non-zero eigenvalues of the Laplacian matrix.

8. [10 marks] Write a function `number_of_spanning_trees` which takes as input a graph G and returns the number of spanning trees in G. You may use the NetworkX function `adjacency_matrix` to compute the adjacency matrix of the input graph, but you **may not use any other NetworkX functions.**

Note: You will probably need to compute the determinant of a certain matrix somewhere in your code. If you use the function `numpy.linalg.det` then your determinant will only be computed approximately, i.e. to a certain numerical precision. This is fine; you will not lose any marks if your code is otherwise correct.

In [33]:
```
def find_degree(G, node):
    adj_matrix = nx.adjacency_matrix(G).toarray()
    num_nodes = G.number_of_nodes()
    degree = 0
    for d in range(num_nodes):
        if adj_matrix[node][d] == 1:
            degree = degree + 1
    return degree
```

In [45]:
```
# Cayley's formula
import math
def number_of_spanning_trees(G):
    adj_matrix = nx.adjacency_matrix(G).toarray()
    num_nodes = G.number_of_nodes()
    # Replace all the diagonal elements with the degree of nodes
    for n in range(num_nodes):
        adj_matrix[n][n] = find_degree(G, n)

    # Replace all non-diagonal 1's with -1.
    for r in range(num_nodes):
        for c in range(num_nodes):
            if adj_matrix[r][c] == 1:
                adj_matrix[r][c] = -1

    print("Matrix: ", adj_matrix)
    # Calculate co-factor for any element.
    det = np.linalg.det(adj_matrix)
    print("co-factor", math.pow(-1, 1)*det)
    return math.pow(-1, 1)*det
```

9 [5 marks] Use your function from Question 8 to calculate the number of spanning trees in your graph from Part I.

In [46]:
```
number_of_spanning_trees(G)
```

```
Matrix:  [[ 6 -1  0 ...  0  0  0]
 [-1  6  0 ...  0  0  0]
 [ 0  0  5 ...  0  0  0]
 ...
 [ 0  0  0 ...  2  0  0]
 [ 0  0  0 ...  0  3 -1]
 [ 0  0  0 ...  0 -1  7]]
co-factor 1.1961427653389379e+23
```

Out[46]:
```
1.1961427653389379e+23
```

10 [5 marks] Find a minimal spanning tree of your graph from Part I, i.e. one with the smallest possible number of edges. Re-draw your graph in such a way that this spanning tree is clearly visible. You should use one colour for the edges in the spanning tree, and a different colour for all other edges.

Hint: You should be able to find a NetworkX function that computes a minimal spanning tree.

In [26]:
```
def primsAlg(G):
    adj_matrix = nx.adjacency_matrix(G).toarray()
    num_nodes = G.number_of_nodes()
    # arbitrarily choose initial node from graph
    node = 0
    # initialize empty edges array and empty MST
    MST = [];
    edges = [];
    visited = [];
    miniEdge = [None,None,float("Inf")];

    # run prims algorithm
    # until we create an MST that contains every vertex from the graph
    while (len(MST) != num_nodes-1):
        # mark this vertex as visited
        visited.append(node)

        for n in range(0, num_nodes):
            if adj_matrix[node][n] != 0:
                edges.append([node,n,adj_matrix[node][n]])

        # find the edge that has the smallest weight to an un-visited no
de
        for e in range(0, len(edges)):
            if edges[e][2] < miniEdge[2] and edges[e][1] not in visited:
                miniEdge = edges[e]

        edges.remove(miniEdge)
        MST.append(miniEdge)

        # restart, at a new node and reset the mini edge
        node = miniEdge[1]
        miniEdge = [None,None,float("Inf")];
    return MST
```

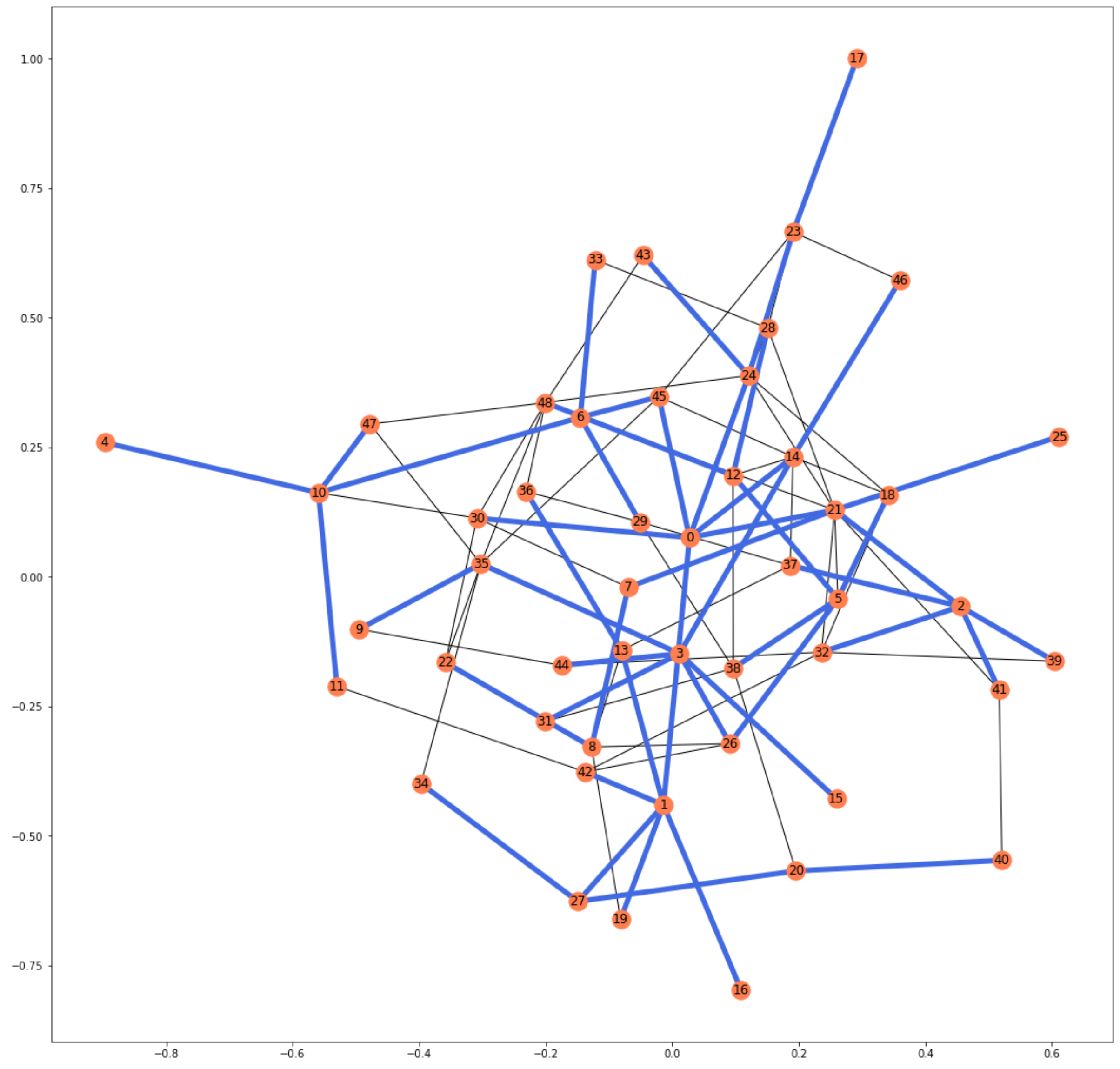
```
In [25]: primsAlg(G)
```

```
Out[25]: [[0, 1, 1],
          [0, 14, 1],
          [0, 21, 1],
          [0, 24, 1],
          [0, 30, 1],
          [0, 45, 1],
          [1, 13, 1],
          [1, 16, 1],
          [1, 19, 1],
          [1, 27, 1],
          [1, 42, 1],
          [14, 3, 1],
          [14, 12, 1],
          [14, 18, 1],
          [14, 37, 1],
          [14, 46, 1],
          [21, 2, 1],
          [21, 5, 1],
          [21, 6, 1],
          [21, 7, 1],
          [21, 25, 1],
          [21, 28, 1],
          [21, 32, 1],
          [21, 41, 1],
          [24, 23, 1],
          [24, 43, 1],
          [24, 48, 1],
          [30, 10, 1],
          [30, 22, 1],
          [45, 35, 1],
          [13, 8, 1],
          [13, 36, 1],
          [27, 20, 1],
          [27, 34, 1],
          [42, 11, 1],
          [42, 26, 1],
          [3, 15, 1],
          [3, 31, 1],
          [3, 44, 1],
          [12, 38, 1],
          [37, 29, 1],
          [2, 39, 1],
          [6, 33, 1],
          [41, 40, 1],
          [23, 17, 1],
          [48, 47, 1],
          [10, 4, 1],
          [35, 9, 1]]
```

```
In [189]: # Kruskal's algorithm
          mst = nx.minimum_spanning_edges(G,data=False) # a generator of MST edges
          edgelist=list(mst) # make a list of the edges
          print(sorted(edgelist))
          path_edges = set(edgelist)
          pos = nx.spring_layout(G)
          plt.figure(figsize=(18,18))
          nx.draw_networkx(G, pos, node_color = 'Coral', edge_color = 'black', wit
          h_labels=True)
          nx.draw_networkx_edges(G,pos,edgelist=path_edges,edge_color='RoyalBlue',
          width=5)
```

```
[(0, 1), (0, 14), (0, 21), (0, 24), (0, 30), (0, 45), (1, 13), (1, 16),
(1, 19), (1, 27), (1, 42), (2, 21), (2, 32), (2, 37), (2, 39), (2, 41),
(3, 14), (3, 15), (3, 26), (3, 31), (3, 35), (3, 44), (4, 10), (5, 12),
(5, 18), (5, 26), (5, 38), (6, 10), (6, 29), (6, 33), (6, 45), (7, 8),
(7, 21), (8, 22), (9, 35), (10, 11), (10, 47), (12, 28), (12, 48), (13,
36), (14, 46), (17, 23), (20, 27), (20, 40), (21, 25), (23, 24), (24, 4
3), (27, 34)]
```

```
Out[189]: <matplotlib.collections.LineCollection at 0x10d42ad30>
```



Part IV: Eigenvalues and triangles [30 marks]

By now you have seen that certain matrices associated with a graph can tell us a lot about the structure of the graph. In this final part of the project, you will investigate this further, by learning how eigenvalues can be used to reveal even more information about graphs.

11. [5 marks] Explain what a "triangle" in a graph is, and quote a formula for calculating the number of triangles in a graph from the eigenvalues of the adjacency matrix.

Note: You do **not** need to give any proofs, but you must reference any material you use, as explained in the plagiarism warning above.

Answer:

- a "triangle" in a graph:
  - In an undirected graph, a triangle of a graph is formed by 3 nodes adjacent in G that connected by 3 edges, which makes itself a cycle graph, a complete graph.
- Formula to calculate the number of triangles:
  - Algorithm: each node, v, of the graph G:
    - for each pair u, w in nodes(v), where u and w are distinct neighbors of v that both have higher degree than v:
      - if u,v,w form a triangle, increment a running count of the triangles of the graph.
  - references:
    - [1] Suri, S. and Vassilvitskii, S., 2011, March. Counting triangles and the curse of the last reducer. In Proceedings of the 20th international conference on World wide web (pp. 607-614). ACM.
    - [2] Kolountzakis, M.N., Miller, G.L., Peng, R. and Tsourakakis, C.E., 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. Internet Mathematics, 8(1-2), pp.161-185.

12. [5 marks] Calculate the number of triangles in your graph from Part I, using the formula discussed in question 11. Your answer **must** be an integer.

Hint: What is the "trace" of a matrix and how is it related to the eigenvalues?

In [106]:
```
# matrix multiplecation
def matrix_multiple(G, M1, M2, M3):
    num_nodes = G.number_of_nodes()

    for r in range(num_nodes):
        for c in range(num_nodes):
            M3[r][c] = 0
            for i in range(num_nodes):
                M3[r][c] += M1[r][i] * M2[i][c]
    return M3
```

In [107]:
```
# calculate the trace of a matrix (sum of diagnonal elements)
def find_trace(G,adj_matrix):
    trace = 0
    num_nodes = G.number_of_nodes()
    for i in range(num_nodes):
        trace += adj_matrix[i][i]
    return trace
```

In [138]:
```
def compute_num_of_triangles(G):
    adj_matrix = nx.adjacency_matrix(G).toarray()
    num_nodes = G.number_of_nodes()

    # To Store graph^2
    graphv2 = [[None] * num_nodes for i in range(num_nodes)]
    # To Store graph^3
    graphv3 = [[None] * num_nodes for i in range(num_nodes)]

    # Initialize
    for r in range(num_nodes):
        for c in range(num_nodes):
            graphv2[r][c] = graphv3[r][c] = 0

    graphv2 = matrix_multiple(G,adj_matrix, adj_matrix, graphv2)
    graphv3 = matrix_multiple(G,adj_matrix, graphv2, graphv3)

    trace = find_trace(G,graphv3)
    return int(trace/6)
```

In [139]:
```
adj_matrix = nx.adjacency_matrix(G).toarray()
print(adj_matrix)
compute_num_of_triangles(G)
print("Number of Triangles in Graph G: ", compute_num_of_triangles(G))
```

```
[[0 1 0 ... 0 0 0]
 [1 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 1]
 [0 0 0 ... 0 1 0]]
Number of Triangles in Graph G:  13
```

13. [10 marks] Write a function `all_triangles` which finds all of the triangles in a graph. Use your function to count the number of triangles in your graph, and compare with your answer to question 12. (The two answers should, of course, be the same.)

Note: You will need to use your function in the next question, so you should think carefully about what kind of data structure you want it to output.

```
In [135]: num_nodes = G.number_of_nodes()
          count_triangles = 0
          for i in range(num_nodes):
              count_triangles += nx.triangles(G, nodes=i)
          count_triangles = int(count_triangles/3)
          print(count_triangles)
```

```
13
```

```
In [150]: def all_triangles(G):
              num_nodes = G.number_of_nodes()
              adj_matrix = nx.adjacency_matrix(G).toarray()
              count_triangles = 0
              list_triangles = []
              # for i in range(num_nodes):
                  # count_triangles += nx.triangles(G, nodes=i)
              # count_triangles = int(count_triangles/3)

              # Consider every possible triangles in graph
              for i in range(num_nodes):
                  for j in range(i+1,num_nodes):
                      for k in range(j+1,num_nodes):
                          # check if it is a cycle
                          if( i != j and i != k and j != k and adj_matrix[i][j] an
          d adj_matrix[j][k] and adj_matrix[k][i]):
                              count_triangles += 1
                              list_triangles.append([i,j,k])
              return count_triangles, list_triangles
```

```
In [151]: all_triangles(G)
```

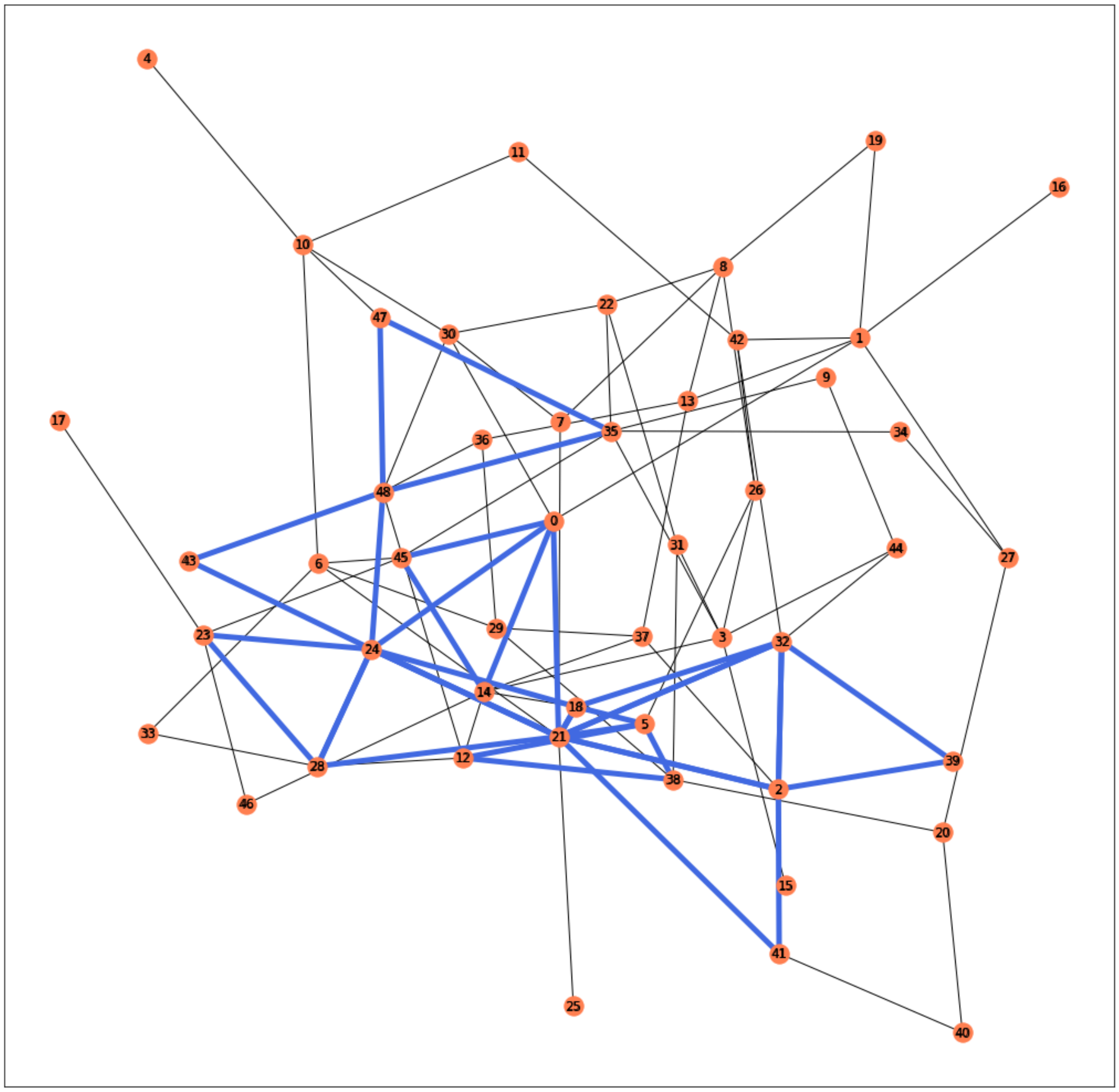
```
Out[151]: (13,
           [[0, 14, 45],
            [0, 21, 24],
            [2, 21, 32],
            [2, 21, 41],
            [2, 32, 39],
            [5, 12, 38],
            [5, 18, 21],
            [18, 21, 24],
            [18, 21, 32],
            [21, 24, 28],
            [23, 24, 28],
            [24, 43, 48],
            [35, 47, 48]])
```

14. [10 marks] Re-draw your graph from Part I once more, so that all of its triangles are clearly visible. You should use one colour for the edges that appear in at least one triangle, and a different colour for all other edges.

In [197]:
```
triangle_list = all_triangles(G)[1]
print(triangle_list)
edges = []
for i in range(len(triangle_list)):
    edges.append([triangle_list[i][0], triangle_list[i][1]])
    edges.append([triangle_list[i][1], triangle_list[i][2]])
    edges.append([triangle_list[i][2], triangle_list[i][0]])
print(edges)

pos = nx.spring_layout(G)
plt.figure(figsize=(18,18))
nx.draw_networkx(G, pos, node_color = 'Coral', edge_color = 'black', wit
h_labels=True)
nx.draw_networkx_edges(G,pos,edgelist=edges,edge_color='RoyalBlue',width
=5)
nx.draw_networkx_labels(G, pos, font_size=10, font_family='sans-serif')
plt.xticks([])
plt.yticks([])
plt.show()
```

```
[[0, 14, 45], [0, 21, 24], [2, 21, 32], [2, 21, 41], [2, 32, 39], [5, 1
2, 38], [5, 18, 21], [18, 21, 24], [18, 21, 32], [21, 24, 28], [23, 24,
28], [24, 43, 48], [35, 47, 48]]
[[0, 14], [14, 45], [45, 0], [0, 21], [21, 24], [24, 0], [2, 21], [21,
32], [32, 2], [2, 21], [21, 41], [41, 2], [2, 32], [32, 39], [39, 2],
[5, 12], [12, 38], [38, 5], [5, 18], [18, 21], [21, 5], [18, 21], [21,
24], [24, 18], [18, 21], [21, 32], [32, 18], [21, 24], [24, 28], [28, 2
1], [23, 24], [24, 28], [28, 23], [24, 43], [43, 48], [48, 24], [35, 4
7], [47, 48], [48, 35]]
```



```
In [ ]:
```