1

Getting Started with Graphs

Graphs are mathematical structures that are used for describing relations between entities and are used almost everywhere. For example, social networks are graphs, where users are connected depending on whether one user "*follows*" the updates of another user. They can be used for representing maps, where cities are linked through streets. Graphs can describe biological structures, web pages, and even the progression of neurodegenerative diseases.

Graph theory, the study of graphs, has received major interest for years, leading people to develop algorithms, identify properties, and define mathematical models to better understand complex behaviors.

This chapter will review some of the concepts behind graph-structured data. Theoretical notions will be presented, together with examples to help you understand some of the more general concepts and put them into practice. In this chapter, we will introduce and use some of the most widely used libraries for the creation, manipulation, and study of the structure dynamics and functions of complex networks, specifically looking at the Python `networkx` library.

The following topics will be covered in this chapter:

- Introduction to graphs with `networkx`
- Plotting graphs
- Graph properties
- Benchmarks and repositories
- Dealing with large graphs

Technical requirements

We will be using Jupyter Notebooks with *Python 3.8* for all of our exercises. In the following code snippet, we show a list of Python libraries that will be installed for this chapter using `pip` (for example, run `pip install networkx==2.5` on the command line, and so on):

```
Jupyter==1.0.0
networkx==2.5
snap-stanford==5.0.0
matplotlib==3.2.2
pandas==1.1.3
scipy==1.6.2
```

In this book, the following Python commands will be referred to:

- `import networkx as nx`
- `import pandas as pd`
- `import numpy as np`

For more complex data visualization tasks, Gephi (`https://gephi.org/`) is also required. The installation manual is available here: `https://gephi.org/users/install/`. All code files relevant to this chapter are available at `https://github.com/PacktPublishing/Graph-Machine-Learning/tree/main/Chapter01`.

Introduction to graphs with networkx

In this section, we will give a general introduction to graph theory. Moreover, in order to merge theoretical concepts with their practical implementation, we will enrich our explanation with code snippets in Python, using `networkx`.

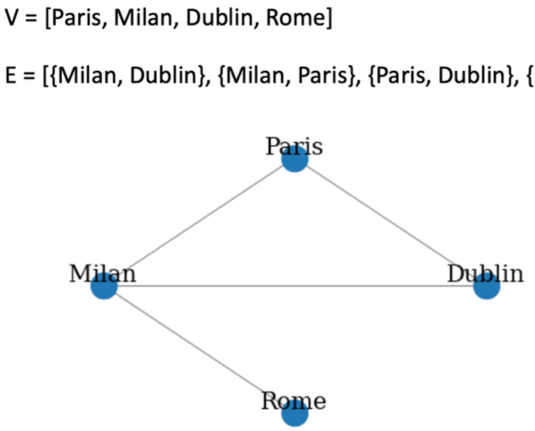
A **simple undirected graph** (or simply, a graph) G is defined as a couple $G=(V,E)$, where $V=\{v_1, .., v_n\}$ is a set of nodes (also called **vertices**) and $E=\{\{v_k, v_w\} .., \{v_i, v_j\}\}$ is a set of two-sets (set of two elements) of edges (also called **links**), representing the connection between two nodes belonging to V.

It is important to underline that since each element of E is a two-set, there is no order between each edge. To provide more detail, $\{v_k, v_w\}$ and $\{v_w, v_k\}$ represent the same edge.

We now provide definitions for some basic properties of graphs and nodes, as follows:

- The **order** of a graph is the number of its vertices $|V|$. The **size** of a graph is the number of its edges $|E|$.
- The **degree** of a vertex is the number of edges that are adjacent to it. The **neighbors** of a vertex v in a graph G is a subset of vertex V' induced by all vertices adjacent to v.
- The **neighborhood graph** (also known as an ego graph) of a vertex v in a graph G is a subgraph of G, composed of the vertices adjacent to v and all edges connecting vertices adjacent to v.

An example of what a graph looks like can be seen in the following screenshot:



V = [Paris, Milan, Dublin, Rome]

E = [{Milan, Dublin}, {Milan, Paris}, {Paris, Dublin}, {Milan, Rome}]

Figure 1.1 – Example of a graph

According to this representation, since there is no direction, an edge from **Milan** to **Paris** is equal to an edge from **Paris** to **Milan**. Thus, it is possible to move in the two directions without any constraint. If we analyze the properties of the graph depicted in *Figure 1.1*, we can see that it has *order* and *size* equal to `4` (there are, in total, four vertices and four edges). The **Paris** and **Dublin** vertices have degree `2`, **Milan** has degree `3`, and **Rome** has degree `1`. The neighbors for each node are shown in the following list:

- `Paris` = {`Milan`, `Dublin`}
- `Milan` = {`Paris`, `Dublin`, `Rome`}
- `Dublin` = {`Paris`, `Milan`}
- `Rome` = {`Milan`}

The same graph can be represented in `networkx`, as follows:

```
import networkx as nx
G = nx.Graph()
V = {'Dublin', 'Paris', 'Milan', 'Rome'}
E = [('Milan','Dublin'), ('Milan','Paris'), ('Paris','Dublin'),
('Milan','Rome')]
G.add_nodes_from(V)
G.add_edges_from(E)
```

Since by default, the `nx.Graph()` command generates an undirected graph, we do not need to specify both directions of each edge. In `networkx`, nodes can be any hashable object: strings, classes, or even other `networkx` graphs. Let's now compute some properties of the graph we previously generated.

All the nodes and edges of the graph can be obtained by running the following code:

```
print(f"V = {G.nodes}")
print(f"E = {G.edges}")
```

Here is the output of the previous commands:

```
V = ['Rome', 'Dublin', 'Milan', 'Paris']
E = [('Rome', 'Milan'), ('Dublin', 'Milan'), ('Dublin',
'Paris'), ('Milan', 'Paris')]
```

We can also compute the graph order, the graph size, and the degree and neighbors for each of the nodes, using the following commands:

```
print(f"Graph Order: {G.number_of_nodes()}")
print(f"Graph Size: {G.number_of_edges()}")
print(f"Degree for nodes: { {v: G.degree(v) for v in G.nodes}
}")
print(f"Neighbors for nodes: { {v: list(G.neighbors(v)) for v
in G.nodes} }")
```

The result will be the following:

```
Graph Order: 4
Graph Size: 4
Degree for nodes: {'Rome': 1, 'Paris': 2, 'Dublin':2, 'Milan':
3}
Neighbors for nodes: {'Rome': ['Milan'], 'Paris': ['Milan',
'Dublin'], 'Dublin': ['Milan', 'Paris'], 'Milan': ['Dublin',
'Paris', 'Rome']}
```

Finally, we can also compute an ego graph of a specific node for the graph `G`, as follows:

```
ego_graph_milan = nx.ego_graph(G, "Milan")
print(f"Nodes: {ego_graph_milan.nodes}")
print(f"Edges: {ego_graph_milan.edges}")
```

The result will be the following:

```
Nodes: ['Paris', 'Milan', 'Dublin', 'Rome']
Edges: [('Paris', 'Milan'), ('Paris', 'Dublin'), ('Milan',
'Dublin'), ('Milan', 'Rome')]
```

The original graph can be also modified by adding new nodes and/or edges, as follows:

```
#Add new nodes and edges
new_nodes = {'London', 'Madrid'}
new_edges = [('London','Rome'), ('Madrid','Paris')]
G.add_nodes_from(new_nodes)
G.add_edges_from(new_edges)
print(f"V = {G.nodes}")
print(f"E = {G.edges}")
```

This would output the following lines:

```
V = ['Rome', 'Dublin', 'Milan', 'Paris', 'London', 'Madrid']
E = [('Rome', 'Milan'), ('Rome', 'London'), ('Dublin',
'Milan'), ('Dublin', 'Paris'), ('Milan', 'Paris'), ('Paris',
'Madrid')]
```

Removal of nodes can be done by running the following code:

```
node_remove = {'London', 'Madrid'}
G.remove_nodes_from(node_remove)
print(f"V = {G.nodes}")
print(f"E = {G.edges}")
```

This is the result of the preceding commands:

```
V = ['Rome', 'Dublin', 'Milan', 'Paris']
E = [('Rome', 'Milan'), ('Dublin', 'Milan'), ('Dublin',
'Paris'), ('Milan', 'Paris')]
```

As expected, all the edges that contain the removed nodes are automatically deleted from the edge list.

Also, edges can be removed by running the following code:

```
node_edges = [('Milan','Dublin'), ('Milan','Paris')]
G.remove_edges_from(node_edges)
print(f"V = {G.nodes}")
print(f"E = {G.edges}")
```

The final result will be as follows:

```
V = ['Dublin', 'Paris', 'Milan', 'Rome']
E = [('Dublin', 'Paris'), ('Milan', 'Rome')]
```

The `networkx` library also allows us to remove a single node or a single edge from a graph `G` by using the following commands: `G. remove_node('Dublin')` and `G.remove_edge('Dublin', 'Paris')`.

Types of graphs

In the previous section, we described how to create and modify simple undirected graphs. Here, we will show how we can extend this basic data structure in order to encapsulate more information, thanks to the introduction of **directed graphs** (**digraphs**), weighted graphs, and multigraphs.

Digraphs

A digraph *G* is defined as a couple *G=(V, E)*, where $V=\{v_1, .., v_n\}$ is a set of nodes and $E=\{(v_k, v_w)$ .., $(v_i, v_j)\}$ is a set of ordered couples representing the connection between two nodes belonging to *V*.

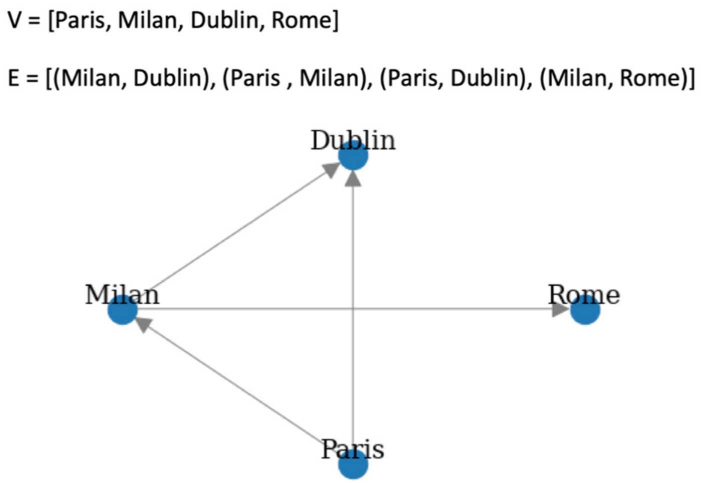
Since each element of *E* is an ordered couple, it enforces the direction of the connection. The edge (v_k, v_w) means *the node* v_k *goes into* v_w. This is different from (v_w, v_k) since it means *the node* v_w *goes to* v_k.The starting node v_w is called the *head*, while the ending node is called the *tail*.

Due to the presence of edge direction, the definition of node degree needs to be extended.

Indegree and outdegree

For a vertex *v*, the number of head ends adjacent to *v* is called the **indegree** (indicated by $deg^{-}(v)$ of *v*, while the number of tail ends adjacent to *v* is its **outdegree** (indicated by $deg^{+}(v)$).

An example of what a digraph looks like is available in the following screenshot:



V = [Paris, Milan, Dublin, Rome]

E = [(Milan, Dublin), (Paris , Milan), (Paris, Dublin), (Milan, Rome)]

Figure 1.2 – Example of a digraph

The direction of the edge is visible from the arrow—for example, **Milan** -> **Dublin** means from **Milan** to **Dublin**. **Dublin** has $deg^{-}(v) = 2$ and $deg^{+}(v) = 0$, **Paris** has $deg^{-}(v) = 0$ and $deg^{+}(v) = 2$, **Milan** has $deg^{-}(v) = 1$ and $deg^{+}(v) = 2$, and **Rome** has $deg^{-}(v) = 1$ and $deg^{+}(v) = 0$.

The same graph can be represented in `networkx`, as follows:

```
G = nx.DiGraph()
V = {'Dublin', 'Paris', 'Milan', 'Rome'}
E = [('Milan','Dublin'), ('Paris','Milan'), ('Paris','Dublin'),
('Milan','Rome')]
G.add_nodes_from(V)
G.add_edges_from(E)
```

The definition is the same as that used for simple undirected graphs; the only difference is in the `networkx` classes that are used to instantiate the object. For digraphs, the `nx.DiGraph()` class is used.

`Indegree` and `Outdegree` can be computed using the following commands:

```
print(f"Indegree for nodes: { {v: G.in_degree(v) for v in
G.nodes} }")
print(f"Outdegree for nodes: { {v: G.out_degree(v) for v in
G.nodes} }")
```

The results will be as follows:

```
Indegree for nodes: {'Rome': 1, 'Paris': 0, 'Dublin': 2,
'Milan': 1}
Outdegree for nodes: {'Rome': 0, 'Paris': 2, 'Dublin': 0,
'Milan': 2}
```

As for the undirected graphs, `G.add_nodes_from()`, `G.add_edges_from()`, `G.remove_nodes_from()`, and `G.remove_edges_from()` functions can be used to modify a given graph `G`.

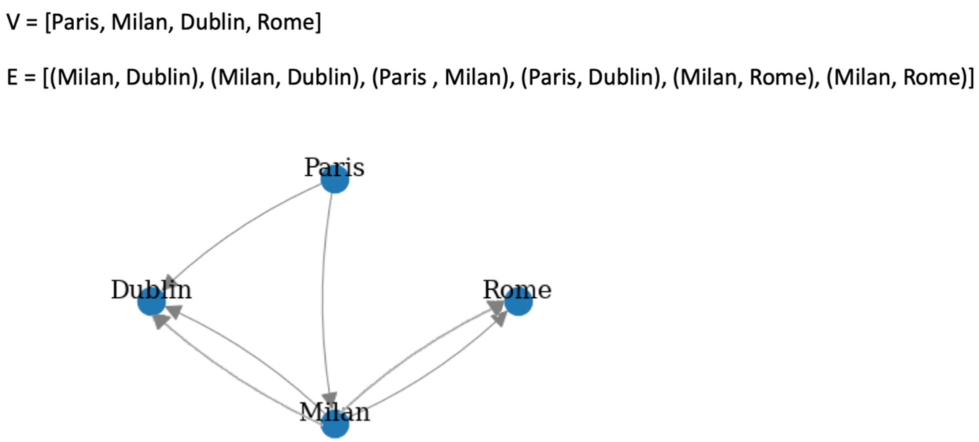
Multigraph

We will now introduce the multigraph object, which is a generalization of the graph definition that allows multiple edges to have the same pair of start and end nodes.

A **multigraph G** is defined as *G=(V, E)*, where *V* is a set of nodes and *E* is a multi-set (a set allowing multiple instances for each of its elements) of edges.

A multigraph is called a **directed multigraph** if *E* is a multi-set of ordered couples; otherwise, if *E* is a multi-set of two-sets, then it is called an **undirected multigraph**.

An example of a directed multigraph is available in the following screenshot:



V = [Paris, Milan, Dublin, Rome]

E = [(Milan, Dublin), (Milan, Dublin), (Paris , Milan), (Paris, Dublin), (Milan, Rome), (Milan, Rome)]

Figure 1.3 – Example of a multigraph

In the following code snippet, we show how to use `networkx` in order to create a directed or an undirected multigraph:

```
directed_multi_graph = nx.MultiDiGraph()
undirected_multi_graph = nx.MultiGraph()
V = {'Dublin', 'Paris', 'Milan', 'Rome'}
E = [('Milan','Dublin'), ('Milan','Dublin'), ('Paris','Milan'),
('Paris','Dublin'), ('Milan','Rome'), ('Milan','Rome')]
directed_multi_graph.add_nodes_from(V)
undirected_multi_graph.add_nodes_from(V)
directed_multi_graph.add_edges_from(E)
undirected_multi_graph.add_edges_from(E)
```

The only difference between a directed and an undirected multigraph is in the first two lines, where two different objects are created: `nx.MultiDiGraph()` is used to create a directed multigraph, while `nx.MultiGraph()` is used to build an undirected multigraph. The function used to add nodes and edges is the same for both objects.

Weighted graphs

We will now introduce directed, undirected, and multi-weighted graphs.

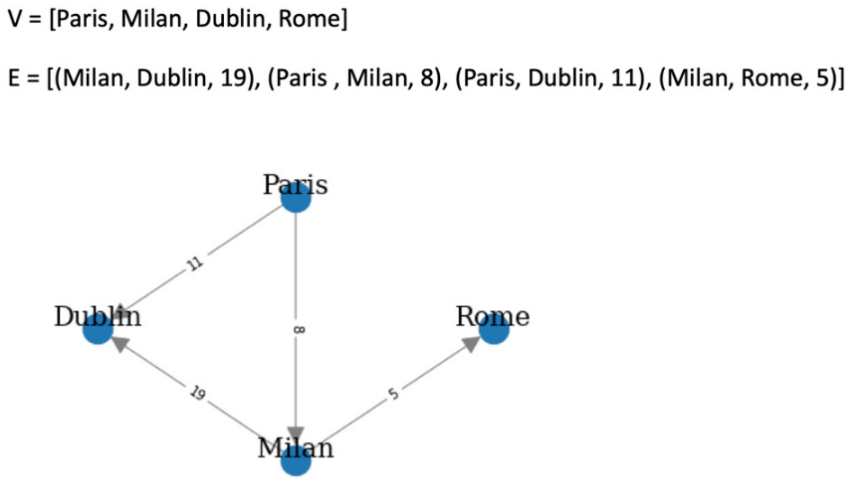
An **edge-weighted graph** (or simply, a weighted graph) G is defined as $G=(V, E ,w)$ where V is a set of nodes, E is a set of edges, and $w: E \rightarrow \mathbb{R}$ is the weighted function that assigns at each edge $e \in E$ a weight expressed as a real number.

A **node-weighted graph** G is defined as $G=(V, E ,w)$,where V is a set of nodes, E is a set of edges, and $w: V \rightarrow \mathbb{R}$ is the weighted function that assigns at each node $v \in V$ a weight expressed as a real number.

Please keep the following points in mind:

- If E is a set of ordered couples, then we call it a **directed weighted graph**.
- If E is a set of two-sets, then we call it an **undirected weighted graph**.
- If E is a multi-set, we will call it a **weighted multigraph** (**directed weighted multigraph**).
- If E is a multi-set of ordered couples, it is an **undirected weighted multigraph**.

An example of a directed edge-weighted graph is available in the following screenshot:



V = [Paris, Milan, Dublin, Rome]

E = [(Milan, Dublin, 19), (Paris , Milan, 8), (Paris, Dublin, 11), (Milan, Rome, 5)]

Figure 1.4 – Example of a directed edge-weighted graph

From *Figure 1.4*, it is easy to see how the presence of weights on graphs helps to add useful information to the data structures. Indeed, we can imagine the edge weight as a "cost" to reach a node from another node. For example, reaching **Dublin** from **Milan** has a "cost" of `19`, while reaching **Dublin** from **Paris** has a "cost" of `11`.

In networkx, a directed weighted graph can be generated as follows:

```
G = nx.DiGraph()
V = {'Dublin', 'Paris', 'Milan', 'Rome'}
E = [('Milan','Dublin', 19), ('Paris','Milan', 8),
('Paris','Dublin', 11), ('Milan','Rome', 5)]
G.add_nodes_from(V)
G.add_weighted_edges_from(E)
```

Bipartite graphs

We will now introduce another type of graph that will be used in this section: multipartite graphs. Bi- and tripartite graphs—and, more generally, kth-partite graphs—are graphs whose vertices can be partitioned in two, three, or more k-th sets of nodes, respectively. Edges are only allowed across different sets and are not allowed within nodes belonging to the same set. In most cases, nodes belonging to different sets are also characterized by particular node types. In *Chapters 7, Text Analytics and Natural Language Processing Using Graphs,* and *Chapter 8*, *Graphs Analysis for Credit Cards Transaction*, we will deal with some practical examples of graph-based applications and you will see how multipartite graphs can indeed arise in several contexts—for example, in the following scenarios:

- When processing documents and structuring the information in a bipartite graph of documents and entities that appear in the documents
- When dealing with transactional data, in order to encode the relations between the buyers and the merchants

A bipartite graph can be easily created in networkx with the following code:

```
import pandas as pd
import numpy as np
n_nodes = 10
n_edges = 12
bottom_nodes = [ith for ith in range(n_nodes) if ith % 2 ==0]
 top_nodes = [ith for ith in range(n_nodes) if ith % 2 ==1]
iter_edges = zip(
    np.random.choice(bottom_nodes, n_edges),
    np.random.choice(top_nodes, n_edges))
edges = pd.DataFrame([
    {"source": a, "target": b} for a, b in iter_edges])
B = nx.Graph()
```

```
B.add_nodes_from(bottom_nodes, bipartite=0)
 B.add_nodes_from(top_nodes, bipartite=1)
 B.add_edges_from([tuple(x) for x in edges.values])
```

The network can also be conveniently plotted using the `bipartite_layout` utility function of `networkx`, as illustrated in the following code snippet:

```
from networkx.drawing.layout import bipartite_layout
pos = bipartite_layout(B, bottom_nodes)
 nx.draw_networkx(B, pos=pos)
```

The `bipatite_layout` function produces a graph, as shown in the following screenshot:

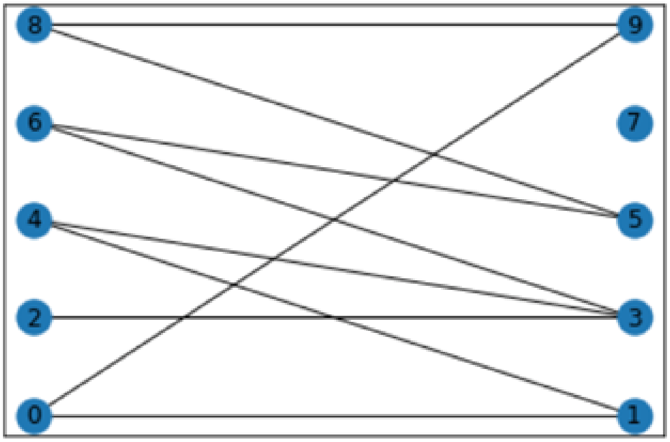


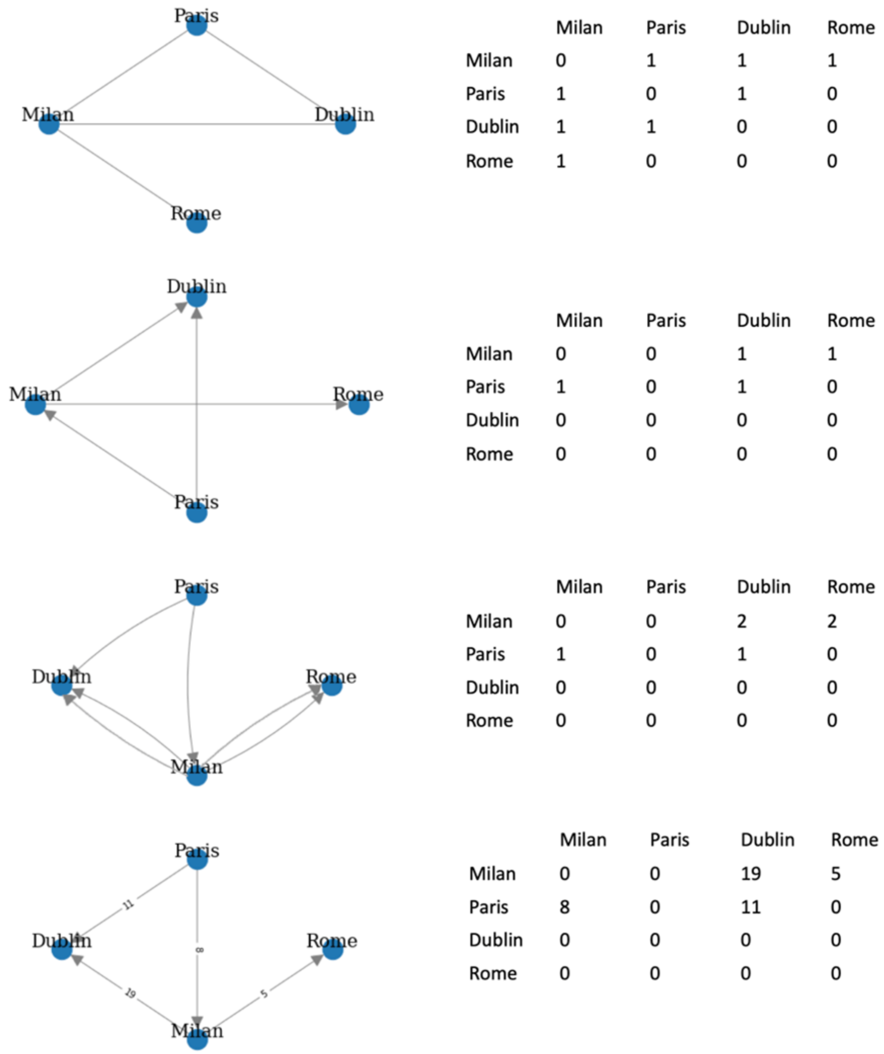
Figure 1.5 – Example of a bipartite graph

Graph representations

As described in the previous sections, with `networkx`, we can actually define and manipulate a graph by using node and edge objects. In different use cases, such a representation would not be as easy to handle. In this section, we will show two ways to perform a compact representation of a graph data structure—namely, an adjacency matrix and an edge list.

Adjacency matrix

The **adjacency matrix** *M* of a graph *G=(V,E)* is a square matrix ($|V| \times |V|$) matrix such that its element M_{ij} is 1 when there is an edge from node *i* to node *j*, and 0 when there is no edge. In the following screenshot, we show a simple example where the adjacency matrix of different types of graphs is displayed:



	Milan	Paris	Dublin	Rome
Milan	0	1	1	1
Paris	1	0	1	0
Dublin	1	1	0	0
Rome	1	0	0	0

	Milan	Paris	Dublin	Rome
Milan	0	0	1	1
Paris	1	0	1	0
Dublin	0	0	0	0
Rome	0	0	0	0

	Milan	Paris	Dublin	Rome
Milan	0	0	2	2
Paris	1	0	1	0
Dublin	0	0	0	0
Rome	0	0	0	0

	Milan	Paris	Dublin	Rome
Milan	0	0	19	5
Paris	8	0	11	0
Dublin	0	0	0	0
Rome	0	0	0	0

Figure 1.6 – Adjacency matrix for an undirected graph, a digraph, a multigraph, and a weighted graph

It is easy to see that adjacency matrices for undirected graphs are always symmetric, since no direction is defined for the edge. The symmetry instead is not guaranteed for the adjacency matrix of a digraph due to the presence of constraints in the direction of the edges. For a multigraph, we can instead have values greater than 1 since multiple edges can be used to connect the same couple of nodes. For a weighted graph, the value in a specific cell is equal to the weight of the edge connecting the two nodes.

In `networkx`, the adjacency matrix for a given graph can be computed in two different ways. If `G` is the `networkx` of *Figure 1.6*, we can compute its adjacency matrix as follows:

```
nx.to_pandas_adjacency(G) #adjacency matrix as pd DataFrame
nt.to_numpy_matrix(G) #adjacency matrix as numpy matrix
```

For the first and second line, we get the following results respectively:

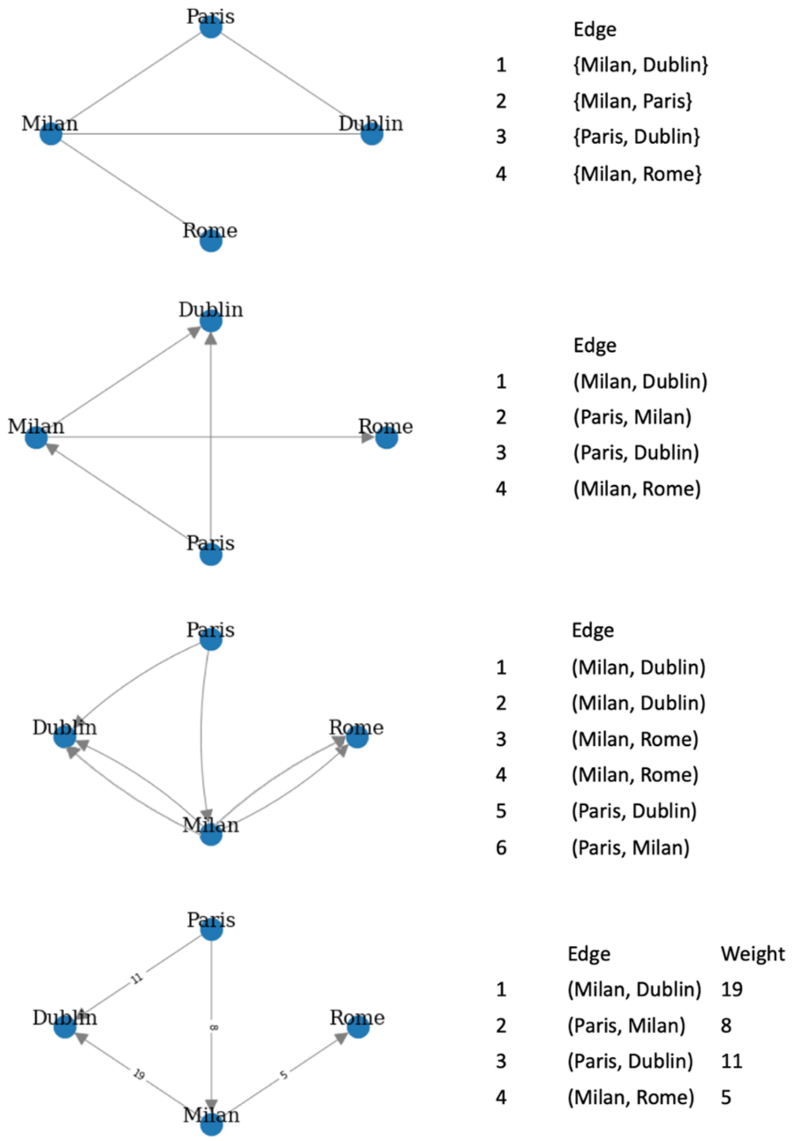
```
           Rome  Dublin  Milan  Paris
Rome       0.0     0.0     0.0    0.0
Dublin     0.0     0.0     0.0    0.0
Milan      1.0     1.0     0.0    0.0
Paris      0.0     1.0     1.0    0.0

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

Since a `numpy` matrix cannot represent the name of the nodes, the order of the element in the adjacency matrix is the one defined in the `G.nodes` list.

Edge list

As well as an adjacency matrix, an edge list is another compact way to represent graphs. The idea behind this format is to represent a graph as a list of edges.

The **edge list** L of a graph $G=(V,E)$ is a list of size $|E|$ matrix such that its element L_i is a couple representing the tail and the end node of the edge i. An example of the edge list for each type of graph is available in the following screenshot:



	Edge
1	{Milan, Dublin}
2	{Milan, Paris}
3	{Paris, Dublin}
4	{Milan, Rome}

	Edge
1	(Milan, Dublin)
2	(Paris, Milan)
3	(Paris, Dublin)
4	(Milan, Rome)

	Edge
1	(Milan, Dublin)
2	(Milan, Dublin)
3	(Milan, Rome)
4	(Milan, Rome)
5	(Paris, Dublin)
6	(Paris, Milan)

	Edge	Weight
1	(Milan, Dublin)	19
2	(Paris, Milan)	8
3	(Paris, Dublin)	11
4	(Milan, Rome)	5

Figure 1.7 – Edge list for an undirected graph, a digraph, a multigraph, and a weighted graph

In the following code snippet, we show how to compute in `networkx` the edge list of the simple undirected graph *G* available in *Figure 1.7*:

```
print(nx.to_pandas_edgelist(G))
```

By running the preceding command, we get the following result:

```
  source  target
0  Milan  Dublin
1  Milan    Rome
2  Paris   Milan
3  Paris  Dublin
```

Other representation methods, which we will not discuss in detail, are also available in `networkx`. Some examples are `nx.to_dict_of_dicts(G)` and `nx.to_numpy_array(G)`, among others.

Plotting graphs

As we have seen in previous sections, graphs are intuitive data structures represented graphically. Nodes can be plotted as simple circles, while edges are lines connecting two nodes.

Despite their simplicity, it could be quite difficult to make a clear representation when the number of edges and nodes increases. The source of this complexity is mainly related to the position (space/Cartesian coordinates) to assign to each node in the final plot. Indeed, it could be unfeasible to manually assign to a graph with hundreds of nodes the specific position of each node in the final plot.

In this section, we will see how we can plot graphs without specifying coordinates for each node. We will exploit two different solutions: `networkx` and Gephi.

networkx

`networkx` offers a simple interface to plot graph objects through the `nx.draw` library. In the following code snippet, we show how to use the library in order to plot graphs:

```
def draw_graph(G, nodes_position, weight):
      nx.draw(G, pos_ position, with_labels=True, font_size=15,
node_size=400, edge_color='gray', arrowsize=30)
             if plot_weight:
             edge_labels=nx.get_edge_attributes(G,'weight')
         nx.draw_networkx_edge_labels(G, pos_ position, edge_
labels=edge_labels)
```

Here, nodes_position is a dictionary where the keys are the nodes and the value assigned to each key is an array of length 2, with the Cartesian coordinate used for plotting the specific node.

The nx.draw function will plot the whole graph by putting its nodes in the given positions. The with_labels option will plot its name on top of each node with the specific font_size value. node_size and edge_color will respectively specify the size of the circle, representing the node and the color of the edges. Finally, arrowsize will define the size of the arrow for directed edges. This option will be used when the graph to be plotted is a digraph.

In the following code example, we show how to use the draw_graph function previously defined in order to plot a graph:

```
G = nx.Graph()
V = {'Paris', 'Dublin','Milan', 'Rome'}
E = [('Paris','Dublin', 11), ('Paris','Milan', 8),
     ('Milan','Rome', 5), ('Milan','Dublin', 19)]
G.add_nodes_from(V)
G.add_weighted_edges_from(E)
node_position = {"Paris": [0,0], "Dublin": [0,1], "Milan":
[1,0], "Rome": [1,1]}
draw_graph(G, node_position, True)
```

The result of the plot is available to view in the following screenshot:

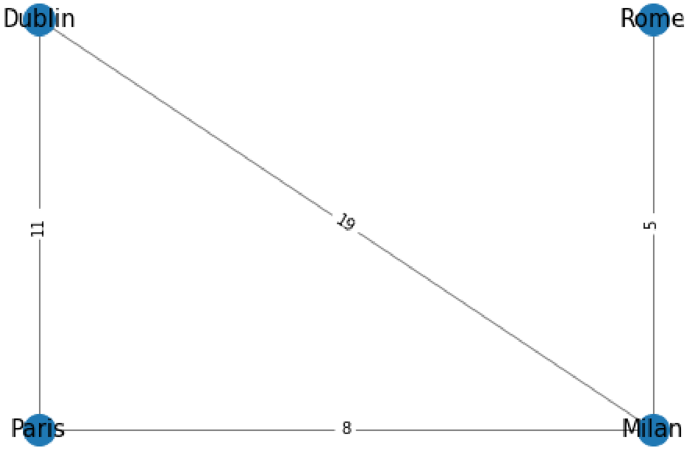


Figure 1.8 – Result of the plotting function

The method previously described is simple but unfeasible to use in a real scenario since the `node_position` value could be difficult to decide. In order to solve this issue, `networkx` offers a different function to automatically compute the position of each node according to different layouts. In *Figure 1.9*, we show a series of plots of an undirected graph, obtained using the different layouts available in `networkx`. In order to use them in the function we proposed, we simply need to assign `node_position` to the result of the layout we want to use—for example, `node_position = nx.circular_layout(G)`. The plots can be seen in the following screenshot:

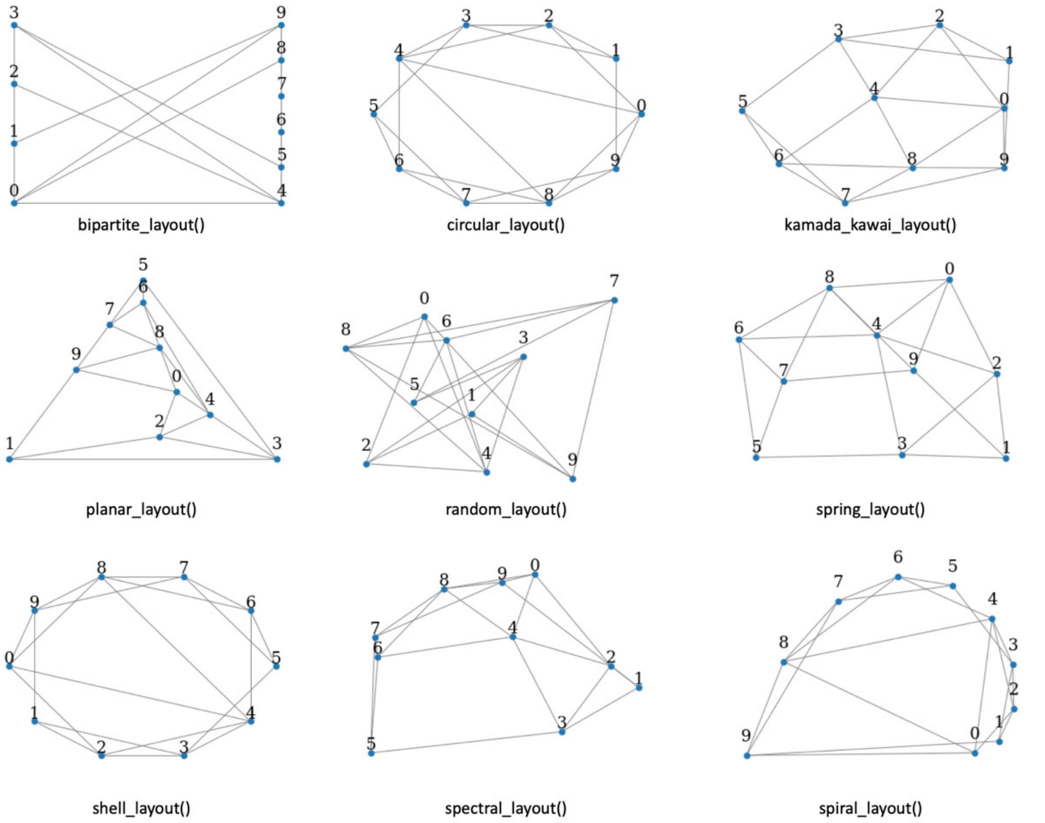


Figure 1.9 – Plots of the same undirected graph with different layouts

`networkx` is a great tool for easily manipulating and analyzing graphs, but it does not offer good functionalities in order to perform complex and good-looking plots of graphs. In the next section, we will investigate another tool to perform complex graph visualization: Gephi.

Gephi

In this section, we will show how **Gephi** (an open source network analysis and visualization software) can be used for performing complex, fancy plots of graphs. For all the examples showed in this section, we will use the `Les Miserables.gexf` sample (a weighted undirected graph), which can be selected in the **Welcome** window when the application starts.

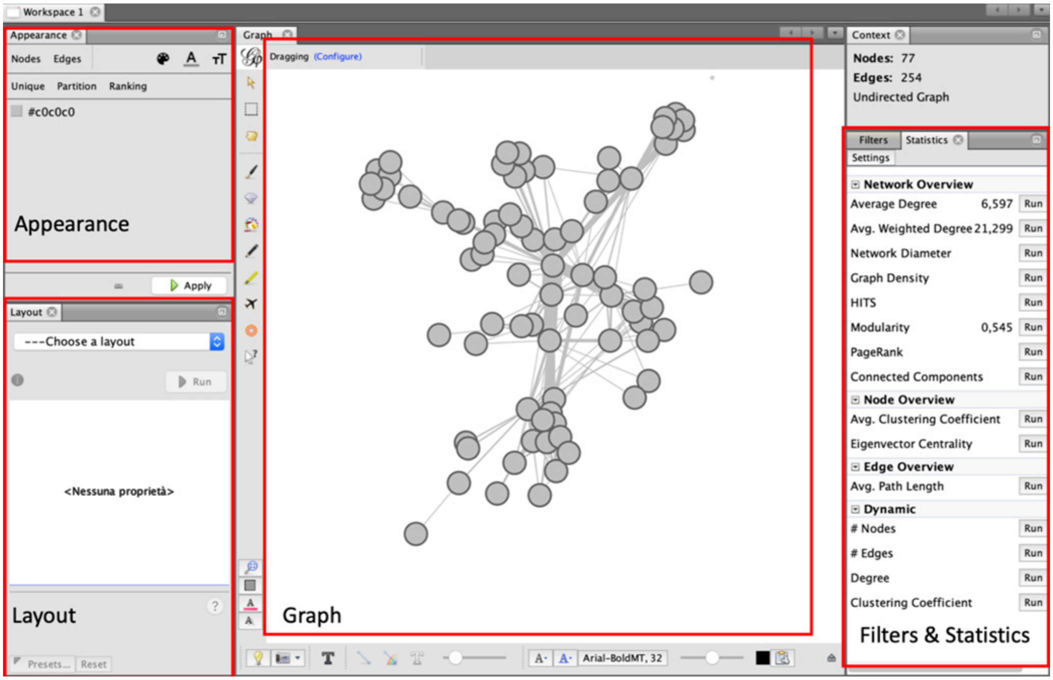
The main interface of Gephi is shown in *Figure 1.10*. It can be divided into four main areas, as follows:

1. **Graph**: This section shows the final plot of the graph. The image is automatically updated each time a filter or a specific layout is applied.
2. **Appearance**: Here, it is possible to specify the appearance of nodes and edges.
3. **Layout**: In this section, it is possible to select the layout (as in `networkx`) to adjust the node position in the graph. Different algorithms, from a simple random position generator to a more complex Yifan Hu algorithm, are available.
4. **Filters & Statistics**: In this set area, two main functions are available, outlined as follows:

   a. **Filters:** In this tab, it is possible to filter and visualize specific subregions of the graph according to a set property computed using the **Statistics** tab.

   b. **Statistics**: This tab contains a list of available graph metrics that can be computed on the graph using the **Run** button. Once metrics are computed, they can be used as properties to specify the edges' and nodes' appearance (such as node and edge size and color) or to filter a specific subregion of the graph.

You can see the main interface of Gephi in the following screenshot:



Nodes: 77
Edges: 254
Undirected Graph

Average Degree 6,597
Avg. Weighted Degree 21,299
Network Diameter
Graph Density
HITS
Modularity 0,545
PageRank
Connected Components
Avg. Clustering Coefficient
Eigenvector Centrality
Avg. Path Length
Nodes
Edges
Degree
Clustering Coefficient

Figure 1.10 – Gephi main window

Our exploration of Gephi starts with the application of different layouts to the graph. As previously described, in `networkx` the layouts allow us to assign to each node a specific position in the final plot. In Gephi 1.2, different layouts are available. In order to apply a specific layout, we have to select from the **Layout** area one of the available layouts, and then click on the **Run** button that appears after the selection.

The graph representation, visible in the **Graph** area, will be automatically updated according to the new coordinates defined by the layout. It should be noted that some layouts are parametric, hence the final graph plot can significantly change according to the parameters used. In the following screenshot, we propose several examples for the application of three different layouts:

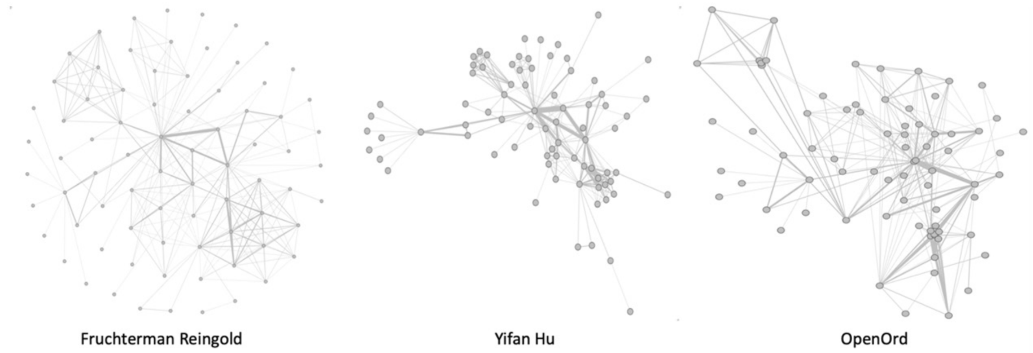


Figure 1.11 – Plot of the same graph with different layout

We will now introduce the available options in the **Appearance** menu visible in *Figure 1.10*. In this section, it is possible to specify the style to be applied to edges and nodes. The style to be applied can be static or can be dynamically defined by specific properties of the nodes/edges. We can change the color and the size of the nodes by selecting the **Nodes** option in the menu.

In order to change the color, we have to select the color palette icon and decide, using the specific button, if we want to assign a **Unique** color, a **Partition** (discrete values), or a **Ranking** (range of values) of colors. For **Partition** and **Ranking**, it is possible to select from the drop-down menu a specific **Graph** property to use as reference for the color range. Only the properties computed by clicking **Run** in the **Statistics** area are available in the drop-down menu. The same procedure can be used in order to set the size of the nodes. By selecting the concentric circles icon, it is possible to set a **Unique** size to all the nodes or to specify a **Ranking** of size according to a specific property.

As for the nodes, it is also possible to change the style of the edges by selecting the **Edges** option in the menu. We can then select to assign a **Unique** color, a **Partition** (discrete values), or a **Ranking** (range of values) of colors. For **Partition** and **Ranking**, the reference value to build the color scale is defined by a specific **Graph** property that can be selected from the drop-down menu.

It is important to remember that in order to apply a specific style to the graph, the **Apply** button should be clicked. As a result, the graph plot will be updated according to the style defined. In the following screenshot, we show an example where the color of the nodes is given by the **Modularity Class** value and the size of each node is given by its degree, while the color of each edge is defined by the edge weight:

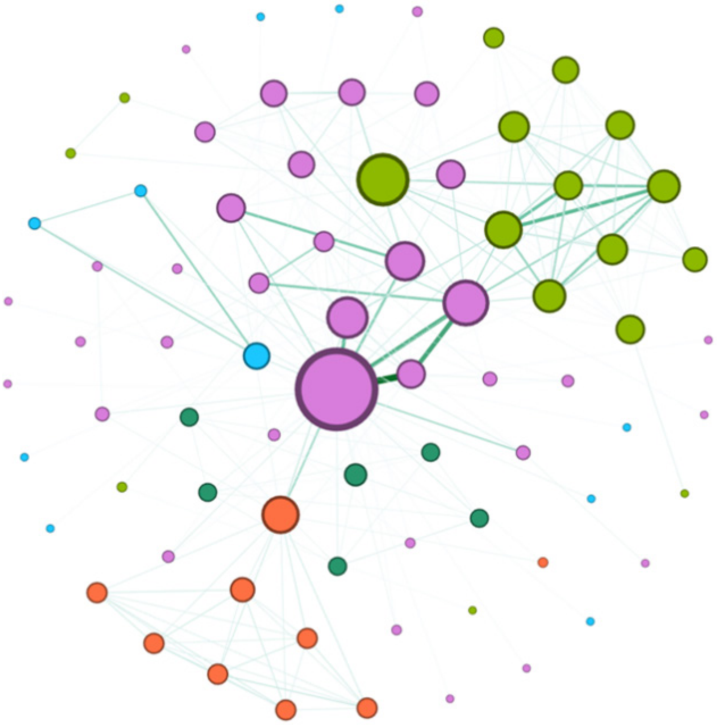


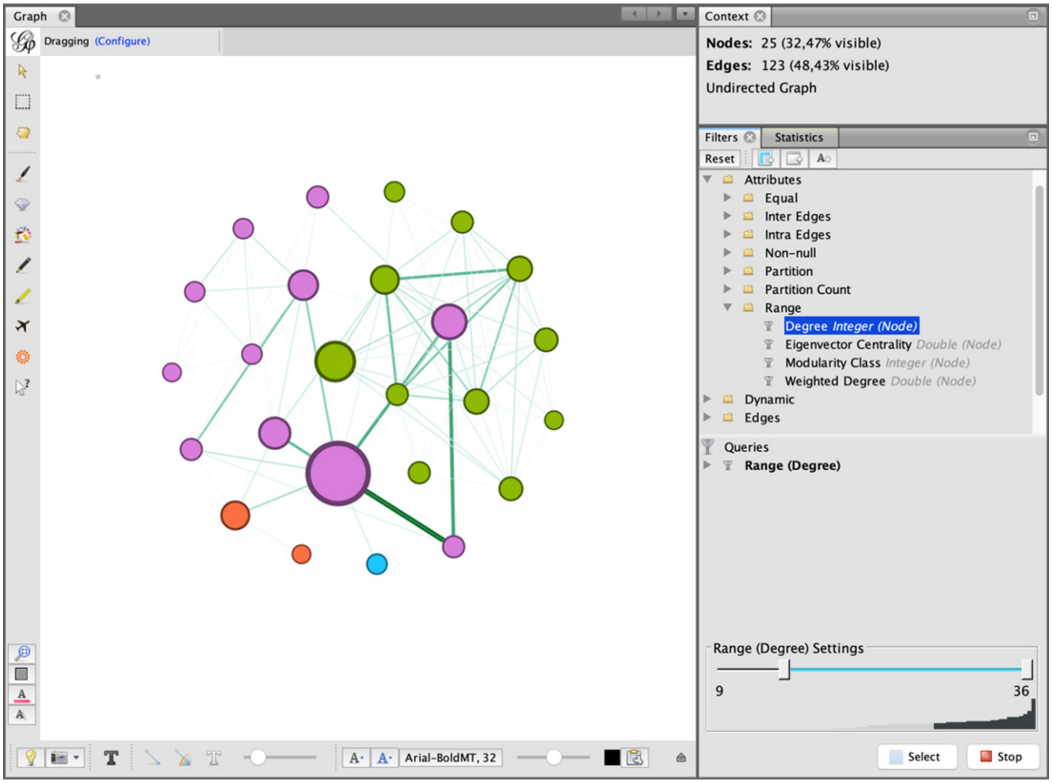
Figure 1.12 – Example of graph plot changing nodes' and edges' appearance

Another important section that needs to be described is **Filters & Statistics**. In this menu, it is possible to compute some statistics based on graph metrics.

Finally, we conclude our discussion on Gephi by introducing the functionalities available in the **Statistics** menu, visible in the right panel in *Figure 1.10*. Through this menu, it is possible to compute different statistics on the input graph. Those statistics can be easily used to set some properties of the final plot, such as nodes'/edges' color and size, or to filter the original graph to plot just a specific subset of it. In order to compute a specific statistic, the user then needs to explicitly select one of the metrics available in the menu and click on the **Run** button (*Figure 1.10*, right panel).

Moreover, the user can select a subregion of the graph, using the options available in the **Filters** tab of the **Statistics** menu, visible in the right panel in *Figure 1.10*. An example of filtering a graph can be seen in *Figure 1.13*. To provide more details of this, we build and apply to the graph a filter, using the **Degree** property. The result of the filters is a subset of the original graph, where only the nodes (and their edges) having the specific range of values for the degree property are visible.

This is illustrated in the following screenshot:



Nodes: 25 (32,47% visible)
Edges: 123 (48,43% visible)
Undirected Graph

Range (Degree) Settings
9 36

Figure 1.13 – Example of a graph filtered according to a range of values for Degree

Of course, Gephi allows us to perform more complex visualization tasks and contains a lot of functionalities that cannot be fully covered in this book. Some good references to better investigate all the features available in Gephi are the official Gephi guide (`https://gephi.org/users/`) or the *Gephi Cookbook* book by Packt Publishing.

Graph properties

As we have already learned, a *graph* is a mathematical model that is used for describing relations between entities. However, each complex network presents intrinsic properties. Such properties can be measured by particular metrics, and each measure may characterize one or several local and global aspects of the graph.

In a graph for a social network such as Twitter, for example, users (represented by the *nodes* of the graph) are connected to each other. However, there are users that are more connected than others (influencers). On the Reddit social graph, users with similar characteristics tend to group into communities.

We have already mentioned some of the *basic features* of graphs, such as the *number of nodes and edges* in a graph, which constitute the size of the graph itself. Those properties already provide a good description of the structure of a network. Think about the Facebook graph, for example: it can be described in terms of the number of nodes and edges. Such numbers easily allow it to be distinguished from a much smaller network (for example, the social structure of an office) but fail to characterize more complex dynamics (for example, how *similar* nodes are connected). To this end, more advanced graph-derived **metrics** can be considered, which can be grouped into four main categories, outlined as follows:

- **Integration metrics**: These measure how nodes tend to be interconnected with each other.
- **Segregation metrics**: These quantify the presence of groups of interconnected nodes, known as communities or modules, within a network.
- **Centrality metrics**: These assess the importance of individual nodes inside a network.
- **Resilience metrics**: These can be thought of as a measure of how much a network is able to maintain and adapt its operational performance when facing failures or other adverse conditions.

Those metrics are defined as **global** when expressing a measure of an overall network. On the other hand, **local** metrics measure values of individual network elements (nodes or edges). In weighted graphs, each property may or may not account for the *edge weights*, leading to **weighted and unweighted metrics**.

In the following section, we describe some of the most commonly used metrics that measure global and local properties. For simplicity, unless specified differently in the text, we illustrate the global unweighted version of the metric. In several cases, this is obtained by averaging the local unweighted properties of the node.

Integration metrics

In this section, some of the most frequently used integration metrics will be described.

Distance, path, and shortest path

The concept of **distance** in a graph is often related to the number of edges to traverse in order to reach a target node from a given source node.

In particular, consider a source node i and a target node j. The set of edges connecting node i to node j is called a **path**. When studying complex networks, we are often interested in finding the **shortest path** between two nodes. A shortest path between a source node i and a target node j is the path having the lowest number of edges compared to all the possible paths between i and j. The **diameter** of a network is the number of edges contained in the longest shortest path among all possible shortest paths.

Take a look at the following screenshot. There are different paths to reach **Tokyo** from **Dublin**. However, one of them is the shortest (the edges on the shortest path are highlighted):

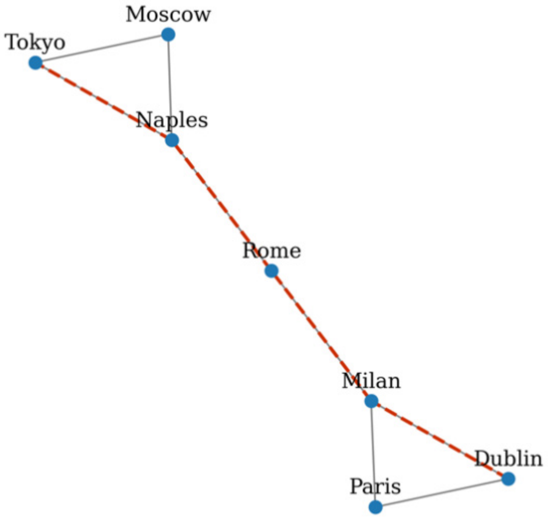


Figure 1.14 – The shortest path between two nodes

The `shortest_path` function of the `networkx` Python library enables users to quickly compute the shortest path between two nodes in a graph. Consider the following code, in which a seven-node graph is created by using `networkx`:

```
G = nx.Graph()
nodes = {1:'Dublin',2:'Paris',3:'Milan',4:'Rome',5:'Naples',
         6:'Moscow',7:'Tokyo'}
G.add_nodes_from(nodes.keys())
G.add_edges_from([(1,2),(1,3),(2,3),(3,4),(4,5),(5,6),(6,7),(7
,5)])
```

The shortest path between a source node (for example, `'Dublin'`, identified by the key 1) and a target node (for example, `'Tokyo'`, identified by the key 7) can be obtained as follows:

```
path = nx.shortest_path(G,source=1,target=7)
```

This should output the following:

```
 [1,3,4,5,6]
```

Here, `[1,3,4,5,7]` are the nodes contained in the shortest path between `'Tokyo'` and `'Dublin'`.

Characteristic path length

The **characteristic path length** is defined as the average of all the shortest path lengths between all possible pair of nodes. If l_i is the average path length between the node i and all the other nodes, the characteristic path length is computed as follows:

$$\frac{1}{q(q-1)} \sum_{i \in V} l_i$$

Here, V is the set of nodes in the graph and $q = |V|$ represents its *order*. This is one of the most commonly used measures of how efficiently information is spread across a network. Networks having shorter characteristic path lengths promote the quick transfer of information and reduce costs. Characteristic path length can be computed through `networkx` using the following function:

```
nx.average_shortest_path_length(G)
```

This should give us the following:

```
2.1904761904761907
```

However, this metric cannot be always defined since it is not possible to compute a path among all the nodes in *disconnected graphs*. For this reason, **network efficiency** is also widely used.

Global and local efficiency

Global efficiency is the average of the inverse shortest path length for all pairs of nodes. Such a metric can be seen as a measure of how efficiently information is exchanged across a network. Consider that l_{ij} is the shortest path between a node i and a node j. The network efficiency is defined as follows:

$$\frac{1}{q(q-1)}\sum_{i\in V}\frac{1}{l_{ij}}$$

Efficiency is at a maximum when a graph is fully connected, while it is minimal for completely disconnected graphs. Intuitively, the shorter the path, the lower the measure.

The **local efficiency** of a node can be computed by considering only the neighborhood of the node in the calculation, without the node itself. Global efficiency is computed in `networkx` using the following command:

```
nx.global_efficiency(G)
```

The output should be as follows:

```
0.6111111111111109
```

Average local efficiency is computed in `networkx` using the following command:

```
nx.local_efficiency(G)
```

The output should be as follows:

```
0.6666666666666667
```

In the following screenshot, two examples of graphs are depicted. As observed, a fully connected graph on the left presents a higher level of efficiency compared to a circular graph on the right. In a fully connected graph, each node can be reached from any other node in the graph, and information is exchanged rapidly across the network. However, in a circular graph, several nodes should instead be traversed to reach the target node, making it less efficient:

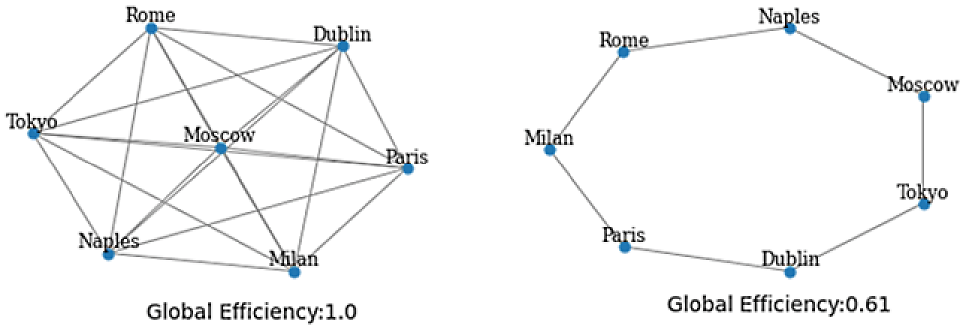


Figure 1.15 – Global efficiency of a fully connected graph (left) and a circular graph (right)

Integration metrics well describe the connection among nodes. However, more information about the presence of groups can be extracted by considering segregation metrics.

Segregation metrics

In this section, some of the most common segregation metrics will be described.

Clustering coefficient

The **clustering coefficient** is a measure of how much nodes cluster together. It is defined as the fraction of **triangles** (complete subgraph of three nodes and three edges) around a node and is equivalent to the fraction of the node's *neighbors* that are neighbors of each other. A global clustering coefficient is computed in `networkx` using the following command:

```
nx.average_clustering(G)
```

This should output the following:

```
0.6666666666666667
```

The local clustering coefficient is computed in `networkx` using the following command:

```
nx.clustering(G)
```

This should output the following:

```
{1: 1.0,
 2: 1.0,
 3: 0.3333333333333333,
 4: 0,
 5: 0.3333333333333333,
 6: 1.0,
 7: 1.0}
```

The output is a Python dictionary containing, for each node (identified by the respective key), the corresponding value. In the graph represented in *Figure 1.16*, two clusters of nodes can be easily identified. By computing the clustering coefficient for each single node, it can be observed that **Rome** has the lowest value. **Tokyo** and **Moscow**, as well as **Paris** and **Dublin**, are instead very well connected within their respective groups (notice the size of each node is drawn proportionally to each node's clustering coefficient). The graph can be seen in the following screenshot:

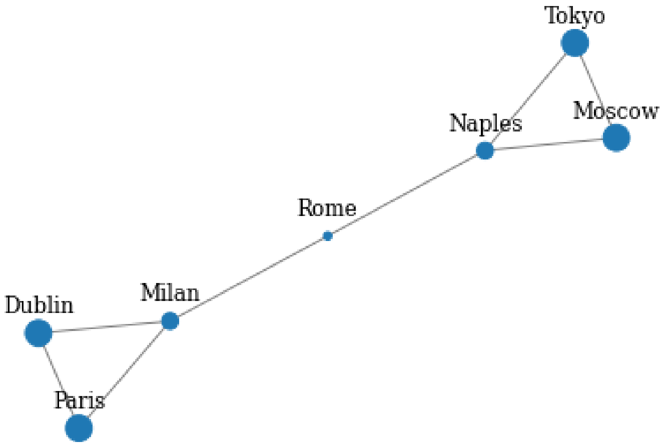


Figure 1.16 – Local clustering coefficient representation

Transitivity

A common variant of the clustering coefficient is known as **transitivity**. This can simply be defined as the ratio between the observed number of **closed triplets** (complete subgraph with three nodes and two edges) and the maximum possible number of closed triplets in the graph. Transitivity can be computed using `networkx`, as follows:

```
nx.transitivity(G)
```

The output should be as follows:

```
0.5454545454545454
```

Modularity

Modularity was designed to quantify the division of a network in aggregated sets of highly interconnected nodes, commonly known as **modules**, **communities**, **groups**, or **clusters**. The main idea is that networks having high modularity will show dense connections within the module and sparse connections between modules.

Consider a social network such as Reddit: members of communities related to video games tend to interact much more with other users in the same community, talking about recent news, favorite consoles, and so on. However, they will probably interact less with users talking about fashion. Differently from many other graph metrics, modularity is often computed by means of optimization algorithms.

Modularity in `networkx` is computed using the `modularity` function of the `networkx.algorithms.community` module, as follows:

```
import networkx.algorithms.community as nx_comm
nx_comm.modularity(G, communities=[{1,2,3}, {4,5,6,7}])
```

Here, the second argument—`communities`—is a list of sets, each representing a partition of the graph. The output should be as follows:

```
0.3671875
```

Segregation metrics help to understand the presence of groups. However, each node in a graph has its own *importance*. To quantify it, we can use centrality metrics.

Centrality metrics

In this section, some of the most common centrality metrics will be described.

Degree centrality

One of the most common and simple centrality metrics is the **degree centrality** metric. This is directly connected with the *degree* of a node, measuring the number of *incident* edges on a certain node i.

Intuitively, the more a node is connected to an other node, the more its degree centrality will assume high values. Note that, if a graph is *directed*, the **in-degree centrality** and **out-degree centrality** will be considered for each node, related to the number of *incoming* and *outcoming* edges, respectively. Degree centrality is computed in `networkx` by using the following command:

```
nx.degree_centrality(G)
```

The output should be as follows:

```
{1: 0.3333333333333333, 2: 0.3333333333333333, 3: 0.5,
4: 0.3333333333333333, 5: 0.5, 6: 0.3333333333333333, 7:
0.3333333333333333}
```

Closeness centrality

The **closeness centrality** metric attempts to quantify how much a node is close (well connected) to other nodes. More formally, it refers to the average distance of a node i to all other nodes in the network. If l_{ij} is the shortest path between node i and node j, the closeness centrality is defined as follows:

$$\frac{1}{\sum_{i \in V, i!=j} l_{ij}}$$

Here, V is the set of nodes in the graph. Closeness centrality can be computed in `networkx` using the following command:

```
nx.closeness_centrality(G)
```

The output should be as follows:

```
{1: 0.4, 2: 0.4, 3: 0.5454545454545454, 4: 0.6, 5:
0.5454545454545454, 6: 0.4, 7: 0.4}
```

Betweenness centrality

The **betweenness centrality** metric evaluates how much a node acts as a **bridge** between other nodes. Even if poorly connected, a node can be strategically connected, helping to keep the whole network connected.

If L_{wj} is the total number of shortest paths between node w and node j and $L_{wj}(i)$ is the total number of shortest paths between w and j passing through node i, then the betweenness centrality is defined as follows:

$$\sum_{w\,!=i\,!=j} \frac{L_{wj}(i)}{L_{wj}}$$

If we observe the formula, we can notice that the higher the number of shortest paths passing through node i, the higher the value of the betweenness centrality. Betweenness centrality is computed in `networkx` by using the following command:

```
nx.betweenness_centrality(G)
```

The output should be as follows:

```
{1: 0.0, 2: 0.0, 3: 0.5333333333333333, 4: 0.6, 5:
0.5333333333333333, 6: 0.0, 7: 0.0}
```

In *Figure 1.17*, we illustrate the difference between *degree centrality*, *closeness centrality*, and *betweenness centrality*. **Milan** and **Naples** have the highest degree centrality. **Rome** has the highest closeness centrality since it is the closest to any other node. It also shows the highest betweenness centrality because of its crucial role in connecting the two visible clusters and keeping the whole network connected.

You can see the differences here:

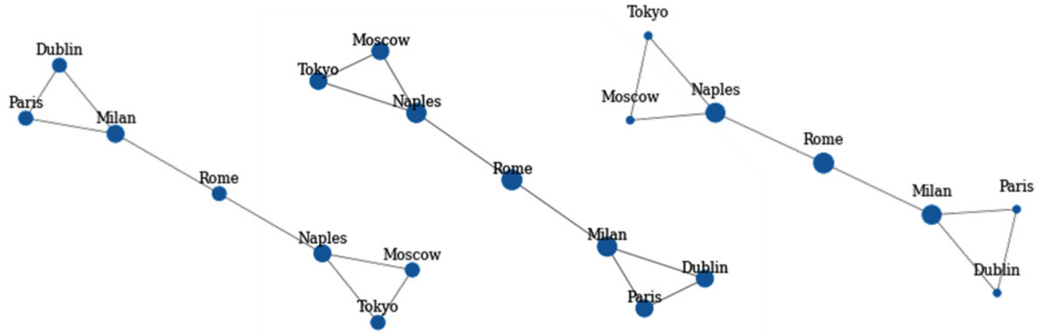


Figure 1.17 – Degree centrality (left), closeness centrality (center), and betweenness centrality (right)

Centrality metrics allow us to measure the importance of a node inside the network. Finally, we will mention resilience metrics, which enable us to measure the vulnerability of a graph.

Resilience metrics

There are several metrics that measure a network's resilience. Assortativity is one of the most used.

Assortativity coefficient

Assortativity is used to quantify the tendency of nodes being connected to similar nodes. There are several ways to measure such correlations. One of the most commonly used methods is the **Pearson correlation coefficient** between the degrees of directly connected nodes (nodes on two opposite ends of a link). The coefficient assumes positive values when there is a correlation between nodes of a similar degree, while it assumes negative values when there is a correlation between nodes of a different degree. Assortativity using the Pearson correlation coefficient is computed in `networkx` by using the following command:

```
nx.degree_pearson_correlation_coefficient(G)
```

The output should be as follows:

```
-0.6
```

Social networks are mostly assortative. However, the so-called *influencers* (famous singers, football players, fashion bloggers) tend to be *followed* (incoming edges) by several standard users, while tending to be connected with each other and showing a disassortative behavior.

It is important to remark that the previously presented properties are a subset of all the possible metrics used to describe graphs. A wider set of metrics and algorithms can be found at `https://networkx.org/documentation/stable/reference/algorithms/`.

Benchmarks and repositories

Now that we have understood the basic concepts and notions about graphs and network analysis, it is now time to dive into some practical examples that will help us to start to put into practice the general concepts we have learned so far. In this section, we will present some examples and toy problems that are generally used to study the properties of networks, as well as benchmark performances and effectiveness of networks' algorithms. We will also provide some useful links of repositories where network datasets can be found and downloaded, together with some tips on how to parse and process them.

Examples of simple graphs

We start by looking at some very simple examples of networks. Fortunately, `networkx` already comes with a number of graphs already implemented, ready to be used and played with. Let's start by creating a **fully connected undirected graph**, as follows:

```
complete = nx.complete_graph(n=7)
```

This has $\frac{n \cdot (n-1)}{2} = 21$ edges and a clustering coefficient *C=1*. Although fully connected graphs are not very interesting on their own, they represent a fundamental building block that may arise within larger graphs. A fully connected subgraph of *n* nodes within a larger graph is generally referred to as a **clique** of size *n*.

Definition

A **clique**, *C*, in an undirected graph is defined a subset of its vertices, $C \subseteq V$, such that every two distinct vertices in the subset are adjacent. This is equivalent to the condition that the induced subgraph of *G* induced by *C* is a fully connected graph.

Cliques represent one of the basic concepts in graph theory and are often also used in mathematical problems where relations need to be encoded. Besides, they also represent the simplest unit when constructing more complex graphs. On the other hand, the task of finding cliques of a given size *n* in larger graphs (clique problem) is of great interest and it can be shown that it is a **nondeterministic polynomial-time complete** (**NP-complete**) problem often studied in computer science.

Some simple examples of `networkx` graphs can be seen in the following screenshot:

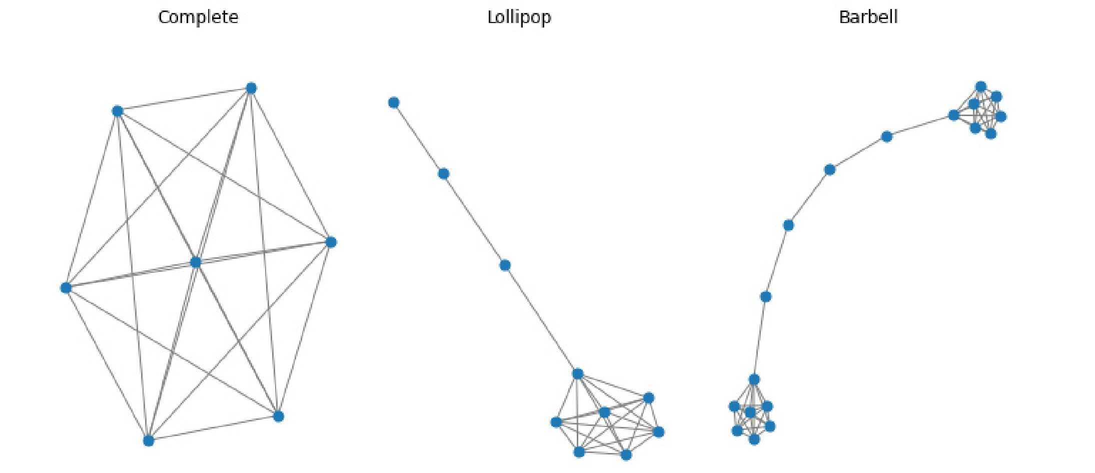


Figure 1.18 – Simple examples of graphs with networkx: (left) fully connected graph; (center) lollipop graph; (right) barbell graph

In *Figure 1.18*, we showed a complete graph along with two other simple examples containing cliques that can be easily generated with `networkx`, outlined as follows:

- A **lollipop graph** formed by a clique of size *n* and a branch of *m* nodes, as shown in the following code snippet:

```
lollipop = nx.lollipop_graph(m=7, n=3)
```

- A **barbell graph** formed by two cliques of size *m1* and *m2* joined by a branch of nodes, which resembles the sample graph we used previously to characterize some of the global and local properties. The code to generate this is shown in the following snippet:

```
barbell = nx.barbell_graph(m1=7, m2=4)
```

Such simple graphs are basic building blocks that can be used to generate more complex networks by combining them. Merging subgraphs is very easy with `networkx` and can be done with just a few lines of code, as shown in the following code snippet, where the three graphs are merged together into a single graph and some random edges are placed to connect them:

```
def get_random_node(graph):
    return np.random.choice(graph.nodes)
allGraphs = nx.compose_all([complete, barbell, lollipop])
allGraphs.add_edge(get_random_node(lollipop), get_random_
node(lollipop))
allGraphs.add_edge(get_random_node(complete), get_random_
node(barbell))
```

Other very simple graphs (that can then be merged and played around with) can be found at https://networkx.org/documentation/stable/reference/generators.html#module-networkx.generators.classic.

Generative graph models

Although creating simple subgraphs and merging them is a way to generate new graphs of increasing complexity, networks may also be generated by means of **probabilistic models** and/or **generative models** that let a graph grow by itself. Such graphs usually share interesting properties with real networks and have long been used to create benchmarks and synthetic graphs, especially in times when the amount of data available was not as overwhelming as today. Here, we present some examples of random generated graphs, briefly describing the models that underlie them.

Watts and Strogatz (1998)

This model was used by the authors to study the behavior of **small-world networks**—that is to say, networks that resemble, to some extent, common social networks. The graph is generated by first displacing n nodes in a ring and connecting each node with its k neighbors. Each edge of such a graph then has a probability p of being rewired to a randomly chosen node. By ranging p, the Watts and Strogatz model allows a shift from a regular network ($p=0$) to a completely random network ($p=1$). In between, graphs exhibit small-world features; that is, they tend to bring this model closer to social network graphs. These kinds of graphs can be easily created with the following command:

```
graph = nx.watts_strogatz_graph(n=20, k=5, p=0.2)
```

Barabási-Albert (1999)

The model proposed by Albert and Barabási is based on a generative model that allows the creation of random scale-free networks by using a **preferential attachment** schema, where a network is created by progressively adding new nodes and attaching them to already existing nodes, with a preference for nodes that have more neighbors. Mathematically speaking, the underlying idea of this model is that the probability for a new node to be attached to an existing node i depends on the degree of the i-th node, according to the following formula:

$$p_i = \frac{k_i}{\sum k_j}$$

Thus, nodes with a large number of edges (hubs) tend to develop even more edges, whereas nodes with few links will not develop other links (periphery). Networks generated by this model exhibit a *power-law distribution* for the connectivity (that is, degree) between nodes. Such a behavior is also found in real networks (for example, the **World Wide Web** (**WWW**) network and the actor collaboration network), interestingly showing that it is the popularity of a node (how many edges it already has) rather than its intrinsic node properties that influences the creation of new connections. The initial model has then been extended (and this is the version that is available on `networkx`) to also allow the preferential attachment of new edges or rewiring of existing edges.

The Barabási-Albert model is illustrated in the following screenshot:

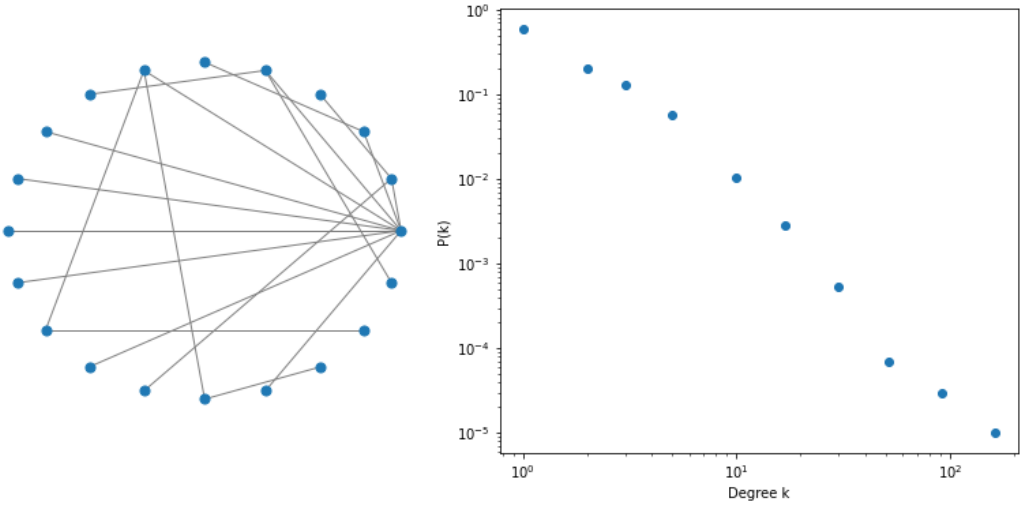


Figure 1.19 – Barabási-Albert model (left) with 20 nodes (right) distribution of connectivity with n=100.000 nodes, showing the scale-free power law distribution

In *Figure 1.19*, we showed an example of the Barabasi-Albert model for a small network, where you can already observe the emergence of hubs (on the left), as well as the probability distribution of the degree of the nodes, which exhibits a scale-free power-law behavior (on the right). The preceding distribution can easily be replicated in `networkx`, as follows:

```
ba_model = nx.extended_barabasi_albert_graph(n,m=1,p=0,q=0)
degree = dict(nx.degree(ba_model)).values()
```

```
bins = np.round(np.logspace(np.log10(min(degree)),
np.log10(max(degree)), 10))
```

```
cnt = Counter(np.digitize(np.array(list(degree)), bins))
```

Benchmarks

Digitalization has profoundly changed our lives, and today, any activity, person, or process generates data, providing a huge amount of information to be drilled, analyzed, and used to promote data-driven decision making. A few decades ago, it was hard to find datasets ready to be used to develop or test new algorithms. On the other hand, there exist today plenty of repositories that provide us with datasets, even of fairly large dimensions, to be downloaded and analyzed. These repositories, where people can share datasets, also provide a benchmark where algorithms can be applied, validated, and compared with each other.

In this section, we will briefly go through some of the main repositories and file formats used in network science, in order to provide you with all the tools needed to import datasets—of different sizes—to analyze and play around with.

In such repositories, you will find network datasets coming from some of the common areas of network science, such as social networks, biochemistry, dynamic networks, documents, co-authoring and citations networks, and networks arising from financial transactions. In *Part 3*, *Advanced Applications of Graph Machine Learning*, we will discuss some of the most common type of networks (social networks, graphs arising when processing corpus documents, and financial networks) and analyze them more thoroughly by applying the techniques and algorithms described in *Part 2*, *Machine Learning on Graphs*.

Also, `networkx` already comes with some basic (and very small) networks that are generally used to explain algorithms and basic measures, which can be found at `https://networkx.org/documentation/stable/reference/generators.html#module-networkx.generators.social`. These datasets are, however, generally quite small. For larger datasets, refer to the repositories we present next.

Network Data Repository

The **Network Data Repository** is surely one of the largest repositories of network data (`http://networkrepository.com/`) with several thousand different networks, featuring users and donations from all over the world and top-tier academic institutions. If a network dataset is freely available, chances are that you will find it there. Datasets are classified in about *30 domains*, including biology, economics, citations, social network data, industrial applications (energy, road), and many others. Besides providing the data, the website also provides a tool for interactive visualization, exploration, and comparison of datasets, and we suggest you check it out and explore it.

The data in the Network Data Repository is generally available under the **Matrix Market Exchange Format** (**MTX**) file format. The MTX file format is basically a file format for specifying dense or sparse matrices, real or complex, via readable text files (**American Standard Code for Information Interchange**, or **ASCII**). For more details, please refer to `http://math.nist.gov/MatrixMarket/formats.html#MMformat`.

A file in MTX format can be easily read in Python using `scipy`. Some of the files we downloaded from the Network Data Repository seemed slightly corrupted and required a minimal fix on a 10.15.2 OSX system. In order to fix them, just make sure the header of the file is compliant with the format specifications; that is, with a double `%` and no spaces at the beginning of the line, as in the following line:

```
%%MatrixMarket matrix coordinate pattern symmetric
```

Matrices should be in coordinate format. In this case, the specification points also to an unweighted, undirected graph (as understood by `pattern` and `symmetric`). Some of the files have some comments after the first header line, which are preceded by a single `%`.

As an example, we consider the **Astro Physics** (**ASTRO-PH**) collaboration network. The graph is generated using all the scientific papers available from the e-print *arXiv* repository published in the *Astrophysics* category in the period from January 1993 to April 2003. The network is built by connecting (via undirected edges) all the authors that co-authored a publication, thus resulting in a clique that includes all authors of a given paper. The code to generate the graph can be seen here:

```
from scipy.io import mmread
adj_matrix = mmread("ca-AstroPh.mtx")
graph = nx.from_scipy_sparse_matrix(adj_matrix)
```

The dataset has 17,903 nodes, connected by 196,072 edges. Visualizing so many nodes cannot be done easily, and even if we were to do it, it might not be very informative, as understanding the underlying structure would not be very easy with so much information. However, we can get some insights by looking at specific subgraphs, as we will do next.

First, we can start by computing some basic properties we described earlier and put them into a pandas `DataFrame` for our convenience to later use, sort, and analyze. The code to accomplish this is illustrated in the following snippet:

```
stats = pd.DataFrame({
    "centrality": nx.centrality.betweenness_centrality(graph),
    "C_i": nx.clustering(graph),
    "degree": nx.degree(graph)
})
```

We can easily find out that the node with the largest **degree centrality** is the one with ID `6933`, which has 503 neighbors (surely a very popular and important scientist in astrophysics!), as illustrated in the following code snippet:

```
neighbors = [n for n in nx.neighbors(graph, 6933)]
```

Of course, also plotting its **ego network** (the node with all its neighbors) would still be a bit messy. One way to produce some subgraphs that can be plotted is by sampling (for example, with a 0.1 ratio) its neighbors in three different ways: random (sorting by index is a sort of random sorting), selecting the most central neighbors, or selecting the neighbors with the largest `C_i` values. The code to accomplish this is shown in the following code snippet:

```
nTop = round(len(neighbors)*sampling)
idx = {
    "random": stats.loc[neighbors].sort_index().index[:nTop],
    "centrality": stats.loc[neighbors]\
         .sort_values("centrality", ascending=False)\
         .index[:nTop],
    "C_i": stats.loc[neighbors]\
         .sort_values("C_i", ascending=False)\
         .index[:nTop]
}
```

We can then define a simple function for extracting and plotting a subgraph that includes only the nodes related to certain indices, as shown in the following code snippet:

```
def plotSubgraph(graph, indices, center = 6933):
    nx.draw_kamada_kawai(
        nx.subgraph(graph, list(indices) + [center])
    )
```

Using the preceding function, we can plot the different subgraphs, obtained by filtering the ego network using the three different criteria, based on random sampling, centrality, and the clustering coefficient we presented previously. An example is provided here:

```
plotSubgraph(graph, idx["random"])
```

In *Figure 1.20*, we compare these results where the other networks have been obtained by changing the key value to `centrality` and `C_i`. The random representation seems to show some emerging structure with separated communities. The graph with the most central nodes clearly shows an almost fully connected network, possibly made up of all full professors and influential figures in astrophysics science, publishing on multiple topics and collaborating frequently with each other. Finally, the last representation, on the other hand, highlights some specific communities, possibly connected with a specific topic, by selecting the nodes that have a higher clustering coefficient. These nodes might not have a large degree of centrality, but they very well represent specific topics. You can see examples of the ego subgraph here:

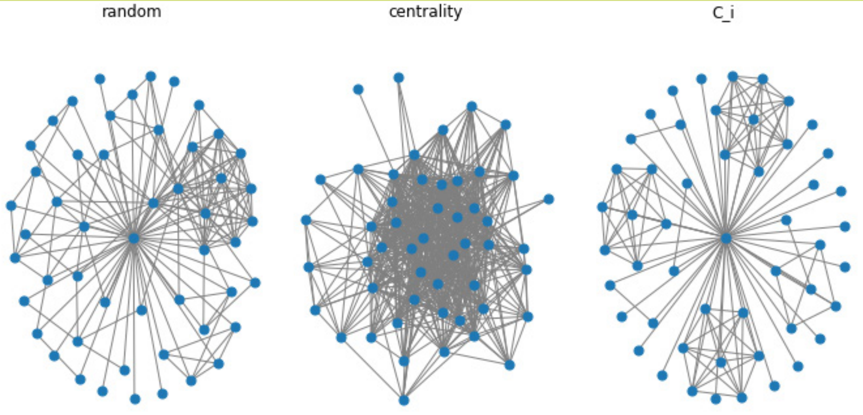


Figure 1.20 – Examples of the ego subgraph for the node that has largest degree in the ASTRO-PH dataset. Neighbors are sampled with a ratio=0.1. (left) random sampling; (center) nodes with largest betweenness centrality; (right) nodes with largest clustering coefficient

Another option to visualize this in `networkx` could also be to use the *Gephi* software that allows for fast filtering and visualizations of graphs. In order to do so, we need to first export the data as **Graph Exchange XML Format** (**GEXF**) (which is a file format that can be imported in Gephi), as follows:

```
nx.write_gext(graph, "ca-AstroPh.gext")
```

Once data is imported in Gephi, with few filters (by centrality or degree) and some computations (modularity), you can easily do plots as nice as the one shown in *Figure 1.21*, where nodes have been colored using modularity in order to highlight clusters. Coloring also allows us to easily spot nodes that connect the different communities and that therefore have large betweenness.

Some of the datasets in the Network Data Repository may also be available in the **EDGE file format** (for instance, the citations networks). The EDGE file format slightly differs from the MTX file format, although it represents the same information. Probably the easiest way to import such files into `networkx` is to convert them by simply rewriting its header. Take, for instance, the **Digital Bibliography and Library** (**DBLP**) citation network.

A sample plot can be seen in the following screenshot:

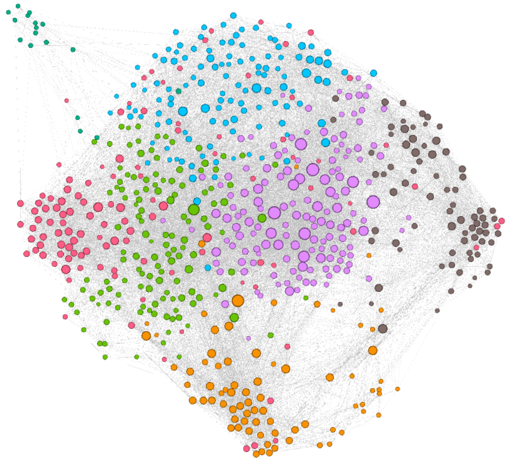


Figure 1.21 – Example of the visualization ASTRO-PH dataset with Gephi. Nodes are filtered by degree centrality and colored by modularity class; node sizes are proportional to the value of the degree

Here is the code for the header of the file:

```
% asym unweighted
% 49743 12591 12591
```

This can be easily converted to comply with the MTX file format by replacing these lines with the following code:

```
%%MatrixMarket matrix coordinate pattern general
12591 12591 49743
```

Then, you can use the import functions described previously.

Stanford Large Network Dataset Collection

Another valuable source of network datasets is the website of the **Stanford Network Analysis Platform (SNAP)** (`https://snap.stanford.edu/index.html`), which is a general-purpose network analysis library that was written in order to handle even fairly large graphs, with hundreds of millions of nodes and billions of edges. It is written in *C++* to achieve top computational performance, but it also features interfaces with Python in order to be imported and used in native Python applications.

Although `networkx` is currently the main library to study `networkx`, SNAP or other libraries (more on this shortly) can be orders of magnitude faster than `networkx`, and they may be used in place of `networkx` for tasks that require higher performance. In the SNAP website, you will find a specific web page for **Biomedical Network Datasets** (`https://snap.stanford.edu/biodata/index.html`), besides other more general networks (`https://snap.stanford.edu/data/index.html`), covering similar domains and datasets as the Network Data Repository described previously.

Data is generally provided in a **text file format** containing a list of edges. Reading such files can be done with `networkx` in one code line, using the following command:

```
g = nx.read_edgelist("amazon0302.txt")
```

Some graphs might have extra information, other than about edges. Extra information is included in the archive of the dataset as a separated file—for example, where some metadata of the nodes is provided and is related to the graph via the *id* node.

Graphs can also be read directly using the SNAP library and its interface via Python. If you have a working version of SNAP on your local machine, you can easily read the data as follows:

```
from snap import LoadEdgeList, PNGraph
graph = LoadEdgeList(PNGraph, "amazon0302.txt", 0, 1, '\t')
```

Keep in mind that at this point, you will have an instance of a `PNGraph` object of the SNAP library, and you can't directly use `networkx` functionalities on this object. If you want to use some `networkx` functions, you first need to convert the `PNGraph` object to a `networkx` object. To make this process simpler, in the supplementary material for this book (available at `https://github.com/PacktPublishing/Graph-Machine-Learning`), we have written some functions that will allow you to seamlessly swap back and forth between `networkx` and SNAP, as illustrated in the following code snippet:

```
networkx_graph = snap2networkx(snap_graph)
snap_graph = networkx2snap(networkx_graph)
```

Open Graph Benchmark

This is the most recent update (dated May 2020) in the graph benchmark landscape, and this repository is expected to gain increasing importance and support in the coming years. The **Open Graph Benchmark** (**OGB**) has been created to address one specific issue: current benchmarks are actually too small compared to real applications to be useful for **machine learning** (**ML**) advances. On one hand, some of the models developed on small datasets turn out to not be able to scale to large datasets, proving them unsuitable in real-world applications. On the other hand, large datasets also allow us to increase the capacity (complexity) of the models used in ML tasks and explore new algorithmic solutions (such as neural networks) that can benefit from a large sample size to be efficiently trained, allowing us to achieve very high performance. The datasets belong to diverse domains and they have been ranked on three different dataset sizes (small, medium, and large) where the small-size graphs, despite their name, already have more than 100,000 nodes and/or more than 1 million edges. On the other hand, large graphs feature networks with more than 100 million nodes and more than 1 billion edges, facilitating the development of scalable models.

Beside the datasets, the OGB also provides, in a *Kaggle fashion*, an end-to-end ML pipeline that standardizes the data loading, experimental setup, and model evaluation. OGB creates a platform to compare and evaluate models against each other, publishing a *leaderboard* that allows tracking of the performance evolution and advancements on specific tasks of node, edge, and graph property prediction. For more details on the datasets and on the OGB project, please refer to `https://arxiv.org/pdf/2005.00687.pdf`.

Dealing with large graphs

When approaching a use case or an analysis, it is very important to understand how large the data we focus on is or will be in the future, as the dimension of the datasets may very well impact both the technologies we use and the analysis that we can do. As already mentioned, some of the approaches that have been developed on small datasets hardly scale to real-world applications and larger datasets, making them useless in practice.

When dealing with (possibly) large graphs, it is crucial to understand potential bottlenecks and limitation of the tools, technologies, and/or algorithms we use, assessing which part of our application/analysis may not scale when increasing the number of nodes or edges. Even more importantly, it is crucial to structure a data-driven application, however simple or at early **proof of concept** (**POC**) stages, in a way that would allow its scaling out in the future when data/users would increase, without rewriting the whole application.

Creating a data-driven application that resorts to graphical representation/modeling is a challenging task that requires a design and implementation that is a lot more complicated than simply importing `networkx`. In particular, it is often useful to decouple the component that processes the graph—named **graph processing engine**—from the one that allows querying and traversing the graph—the **graph storage layer**. We will further discuss these concepts in *Chapter 9*, *Building a Data-Driven Draft-Powered Application*. Nevertheless, given the focus of the book on ML and analytical techniques, it makes sense to focus more on graph processing engines than on graph storage layers. We therefore find it useful to provide you already at this stage with some of the technologies that are used for graph processing engines to deal with large graphs, crucial when scaling out an application.

In this respect, it is important to classify graph processing engines into two categories (that impact the tools/libraries/algorithms to be used), depending whether the graph can fit a *shared memory machine* or requires *distributed architectures* to be processed and analyzed.

Note that there is no absolute definition of large and small graphs, but it also depends on the chosen architecture. Nowadays, thanks to the vertical scaling of infrastructures, you can find servers with **random-access memory** (**RAM**) larger than 1 **terabyte** (**TB**) (usually called *fat nodes*), and with tens of thousands of **central processing units** (**CPUs**) for multithreading in most cloud-provider offerings, although these infrastructures might not be economically viable. Even without scaling out to such extreme architectures, graphs with millions of nodes and tens of millions of edges can nevertheless be easily handled in single servers with ~100 **gigabytes** (**GB**) of RAM and ~50 CPUs.

Although networkx is a very popular, user-friendly, and intuitive library, when scaling out to such reasonably large graphs it may not be the best available choice. networkx, being natively written in pure Python, which is an interpreted language, can be substantially outperformed by other graph engines fully or partly written in more performant programming languages (such as C++ and Julia) and that make use of multithreading, such as the following:

- **SNAP** (http://snap.stanford.edu/), which we have already seen in the previous section, is a graph engine developed at Stanford and is written in C++ with available bindings in Python.
- **igraph** (https://igraph.org/) is a C library and features bindings in Python, R, and Mathematica.
- **graph-tool** (https://graph-tool.skewed.de/), despite being a Python module, has core algorithms and data-structures written in C++ and uses OpenMP parallelization to scale on multi-core architectures.
- **NetworKit** (https://networkit.github.io/) is also written in C++ with OpenMP boost for parallelization for its core functionalities, integrated in a Python module.
- **LightGraphs** (https://juliagraphs.org/LightGraphs.jl/latest/) is a library written in Julia that aims to mirroring networkx functionalities in a more performant and robust library.

All the preceding libraries are valid alternatives to networkx when achieving better performance becomes an issue. Improvements can be very substantial, with speed-ups varying from 30 to 300 times faster, with the best performance generally achieved by LightGraphs.

In the forthcoming chapters, we will mostly focus on networkx in order to provide a consistent presentation and provide the user with basic concepts on network analysis. We want you to be aware that other options are available, as this becomes extremely relevant when pushing the edge from a performance standpoint.

Summary

In this chapter, we refreshed concepts such as graphs, nodes, and edges. We reviewed graph *representation* methods and explored how to *visualize* graphs. We also defined *properties* that are used to characterize networks, or parts of them.

We went through a well-known Python library to deal with graphs, networkx, and learned how to use it to apply theoretical concepts in practice.

We then ran examples and toy problems that are generally used to study the properties of networks, as well as benchmark performance and effectiveness of network algorithms. We also provided you with some useful links of repositories where network datasets can be found and downloaded, together with some tips on how to parse and process them.

In the next chapter, we will go beyond defining notions of ML on graphs. We will learn how more advanced and latent properties can be automatically found by specific ML algorithms.