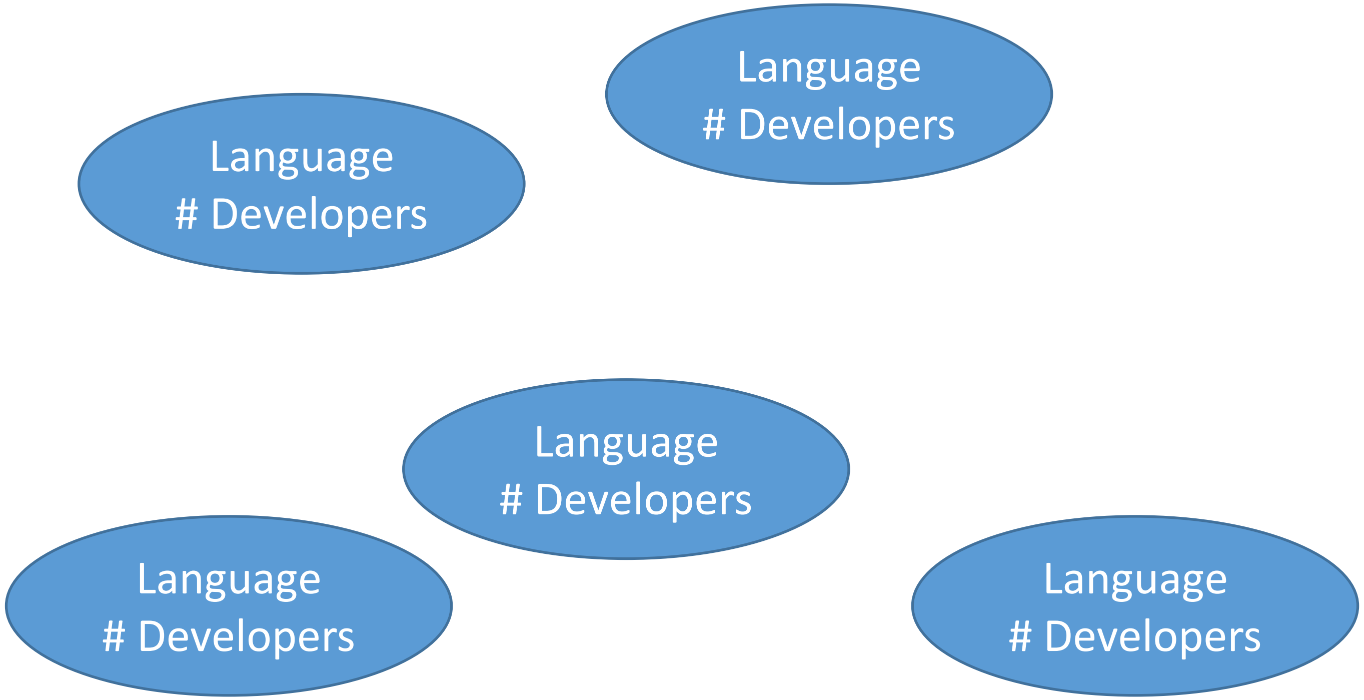
WARWICK BUSINESS SCHOOL
THE UNIVERSITY OF WARWICK

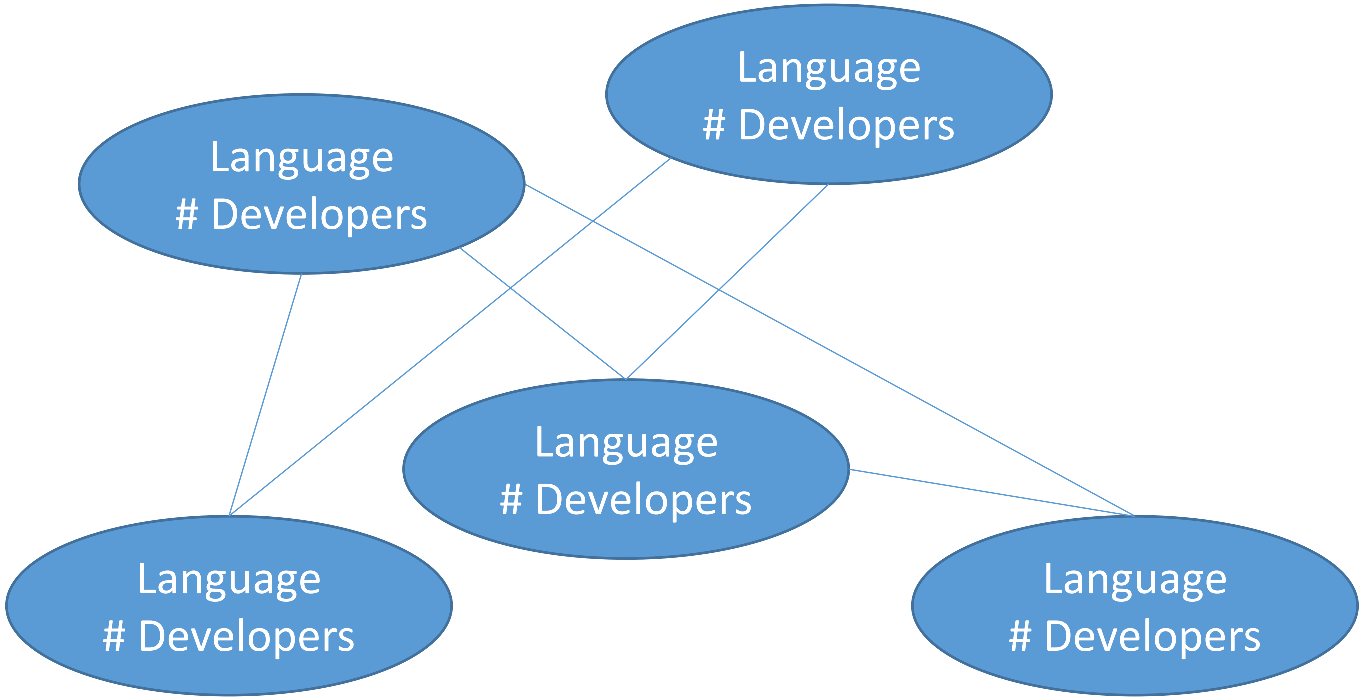
For the Change Makers

# Advanced Programming for Data Science

Week 7: Network Analysis
Information Systems and Management
Warwick Business School

# Network Analysis


Language
# Developers
Language
# Developers
Language
# Developers
Language
# Developers
Language
# Developers


Language
# Developers
Language
# Developers
Language
# Developers
Language
# Developers
Language
# Developers

# Structuralism

- phenomena of human life are not intelligible except through their **interrelations**. These relations constitute a structure, and behind local variations in the surface phenomena there are constant laws of abstract **structure**. (Blackburn 2008)

# Network Analysis

- A network (or graph) is:
  - a collection of connected individuals or entities, each called a vertex or node.
  - a list of pairs of vertices that are neighbors, representing edges or links.
- Examples:
  1. nodes are mathematicians, links represent coauthorship relationships
  2. nodes are Facebook users, links represent Facebook friendships
  3. nodes are news articles, links represent word overlap

# Network Analysis with Python

- One of the most popular Python package to manipulate, analyze and visualize network.
- Vast graph algorithms.
- Requires Numpy, Pandas and Matplotlib.
- Load and save network datasets in different formats.

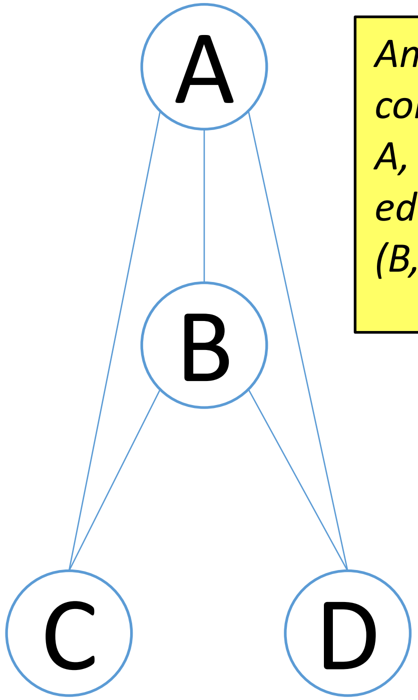
NetworkX
Network Analysis in Python

# Types of network

- A network is represented in the form of graph.
  - Nodes are represented as vertices.
  - Links or ties are represented as edges
- Two types of graphs
  - Undirected graphs
  - Directed graphs

# Undirected Graphs

- A graph G(V, E) is said to be an undirected graph if every edge is bidirectional.
- Every edge is symmetrical and reciprocal.
- Example: Collaborations and friendships on social media applications.




*An undirected graph comprising of 4 vertices A, B, C and D, and 5 edges, (A,B), (A,C), (A,D), (B,C), (B,D)*

# Undirected Graphs

- The **Graph()** class creates an undirected network object. A Graph() object can grow by adding nodes and edges.

```
import networkx as nx
net = nx.Graph()
```

# Adding Nodes

**.add_node(node)** adds one node at a time. Node can be any hashable objects, such number, string, image or even another graph.

```
net.add_node(1)
```

**.add_nodes_from([node, …])** adds multiple nodes stored in an iterable container, such as a list.

```
net.add_nodes_from([3,4,5])
```

**.nodes()** shows current nodes in the graph object.

```
net.nodes()
```

# Adding Edges

**.add_edge(node1, node2)** adds one edge, node1 to node 2, at a time.

```
net.add_edge(1, 2)
```

**.add_edges_from([(node1, node2), …])** adds multiple edges stored in an iterable container, such as a list, of edges in tuples of two nodes.

```
net.add_edges_from([(2,3), (3,4), (4,5)])
```

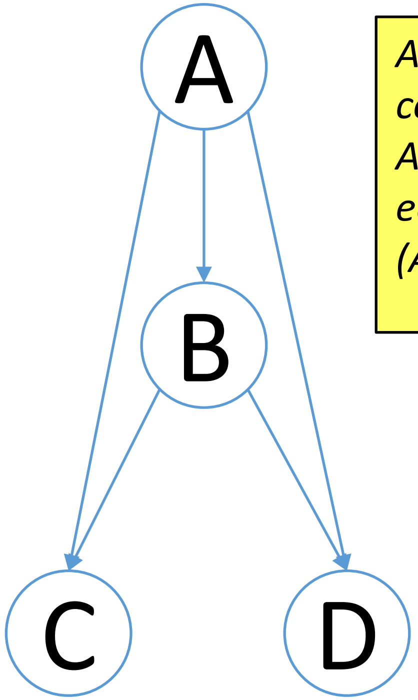
**.edges()** shows current edges in the graph object.

```
net.edges()
```

When adding edges, node that does not exist in the graph will be added automatically.

# Directed Graphs

- A graph G(V, E) is said to be an undirected graph if every edge is directional.
- Every edge is asymmetrical and non-reciprocal.
- Example: Voting and followings on social media applications.




*A directed graph comprising of 4 vertices A, B, C and D, and 5 edges, (A,B), (A,C), (A,D), (B,C), (B,D)*

# Directed Graphs

- The **DiGraph()** class creates a directed network object. A DiGraph() object can grow by adding nodes and edges in the same way as Graph() object.

```
dnet = nx.DiGraph()
dnet.add_node(1)
dnet.add_nodes_from([3,4,5])
```

- When adding edges, the order of nodes matters and indicates the direction.

```
dnet.add_edge(1, 2)
dnet.add_edges_from([(2,3), (3,4), (4,5)])
```

# Node and Edge Attributes

- Nodes and edges may also have attributes to store some information about the entity or relationship.
  - Name: The name of the node.
  - Weight: The frequency of communication between the connected vertices, the strength of this connection, etc.
  - Type: The type of the relationship between the connected vertices. Eg: Family, friends, colleagues.
  - Ranking: Best friend, second best friend, third best friend, so on.
  - Sign: Friend vs foe, trust vs distrust, etc.

# Node Attributes

- Node with attributes can be added by using attributes as arguments.

```
net.add_node(2, name = 'John')
```

- Nodes with attributes can be added as list of tuples: [(node, node_attribute_dict)]

```
net.add_nodes_from([(6, {'name':'Jane'}),
                    (7, {'name':'Jerry'})])
```

# Edge Attributes

- Edge with attributes can be added by using attributes as arguments.

```
net.add_edge(2, 3, weight = 2)
```

- Edges with attributes can be added as list of tuples: [(node1, node2, edge_attribute_dict)]

```
net.add_edge_from([(2, 4, {'weight':3}),
                   (1, 4, {'weight':4})])
```

- Most functions/methods covered in this lecture have an optional argument weight that can be passed with the edge attribute name for weighted edge.

# Reading and Writing Graphs

- NetworkX can read different forms of graphs and write graphs into different formats of files.
- The simplest format of a graph is edge list where each edge is stored in a separate line as two separated nodes.
- Edge list can be read by calling read_edgelist() with arguments:
  - path: this can be a file object or a string of file path.
  - delimiter: default is whitespace, use comma if reading csv file
  - create_using: default is .Graph() for undirected graph, change to DiGraph if creating directed graph
- Graph object can be written to file of edgelist by calling write_edgelist(graph) with arguments like path and delimiter.

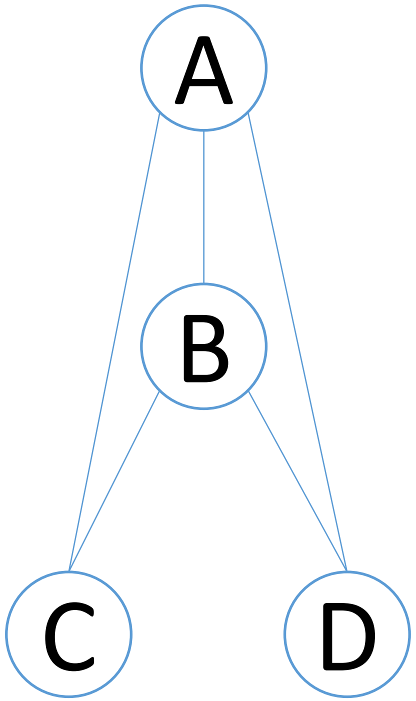
| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 5 | 6 |
| 7 | 8 |

# Key network characteristics

- Degree
- Path
- Clustering coefficient

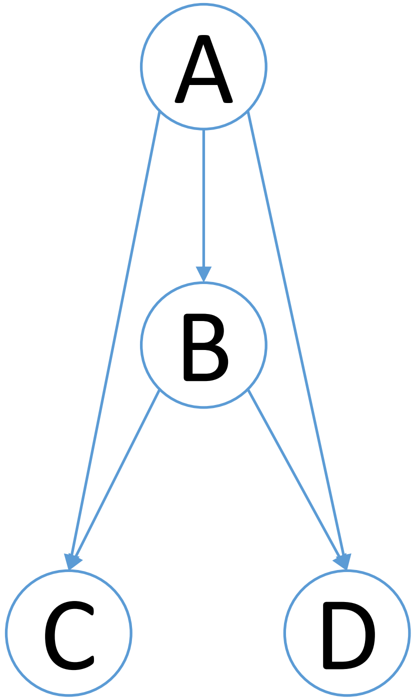
# Degree

- The number of nodes connected to the focal node.
- The degree of a vertex is defined as the number of edges that are adjacent to this vertex.
- For directed graph, in-degree counts the number of degrees directed towards the vertex; while out-degree counts the number of degrees directed away from the vertex. The degree of a vertex is the sum of in-degree and out-degree.




| Vertex | Degree |
|---|---|
| A | 3 |
| B | 3 |
| C | 2 |
| D | 2 |
| Total | 10 |

| degree | frequency |
|---|---|
| 2 | 2/4 |
| 3 | 2/4 |




| Vertex | In-degree | Out-degree | Degree |
|---|---|---|---|
| A | 0 | 3 | 3 |
| B | 1 | 2 | 3 |
| C | 2 | 0 | 2 |
| D | 2 | 0 | 2 |
| Total | 5 | 5 | 10 |

# Calculating Degree for Undirected Graph

- Degrees of a Graph can be easily calculated by calling Graph method **.degree()**.

```
degrees = net.degree()
```

- You may provide a list of nodes if you are only interested in degrees of these nodes.

```
degrees = net.degree([1, 2, 3])
```

- If the graph is weighted, i.e. there is an edge attribute containing the weight information, you can pass the name of weight attribute to argument weight.

```
weighted_degrees = net.degree(weight = 'weight')
```

- **.degree()** returns an iterable container of tuples: (node, degree).

# Calculating Degree for Directed Graph

- Total degrees, in-degrees, out-degrees of a DiGraph can be calculated by calling DiGraph method **.degree()**, **.in_degree()** and **out_degree()**, respectively. They can be used in the similar way.

```
degrees = net.degree()
in_degrees = net.in_degree()
out_degrees = net.out_degree()
```

# Degree Distribution

- Degree distribution is defined as the probability that a random chosen vertex has degree k.
  - A frequency count of the occurrence of each degree.
- The average degree of a graph G(V, E), denoted by k, is defined as the average of the degrees of all the vertices in G.

| Vertex | Degree |
|---|---|
| A | 3 |
| B | 3 |
| C | 2 |
| D | 2 |
| Total | 10 |
| Average | 2.5 |

| degree | frequency |
|---|---|
| 2 | 2/4 |
| 3 | 2/4 |

# Calculate and Plot Degree Distribution

- To understand degree distribution, degree_histogram() can be called to generate a list of frequencies of degrees with the list index being the degree values.

```
degree_freq = nx.degree_histogram(net)
```

- We can then convert the result to DataFrame or ndarray to plot the distribution in Seaborn or Matplotlib.

```
df_degree = pd.DataFrame(degree_freq,columns=['count'])
df_degree['degree'] = pd.DataFrame(degrees)
```

# Why do we care about degree?

- Degree centrality: the number of ties that a node has.
- The simplest centrality measure.
- We want to know the most well connected nodes in a network, which are usually most important nodes.
- **Example**
  - Centrality was a stronger direct predictor of performance than the individual characteristics.

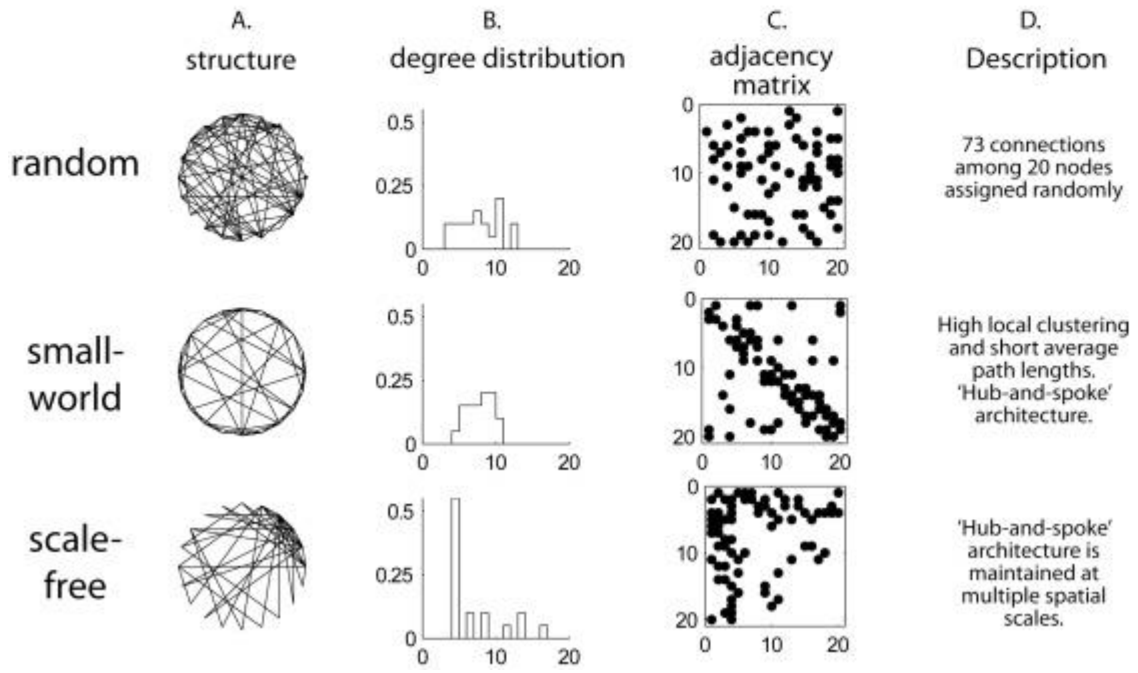
Ahuja, Manju K., Dennis F. Galletta, and Kathleen M. Carley. "Individual centrality and performance in virtual R&D groups: An empirical study." Management science 49.1 (2003): 21-38.

# Calculating Degree Centrality

- Degree centrality of a node essentially can be seen as the degree of that node. But generally we would calculate the degree centrality by normalizing the node degree. NetworkX provides three functions to calculate degree centrality, with Graph/DiGraph object as the only argument, based on graph type:
  1. `degree_centrality()`
  2. `in_degree_centrality()`
  3. `out_degree_centrality()`
- A dictionary of nodes with degree centrality as the value will be returned.

# Why do we care about degree

- Degree distribution is useful to understand the types of network and the underlying formation mechanisms.

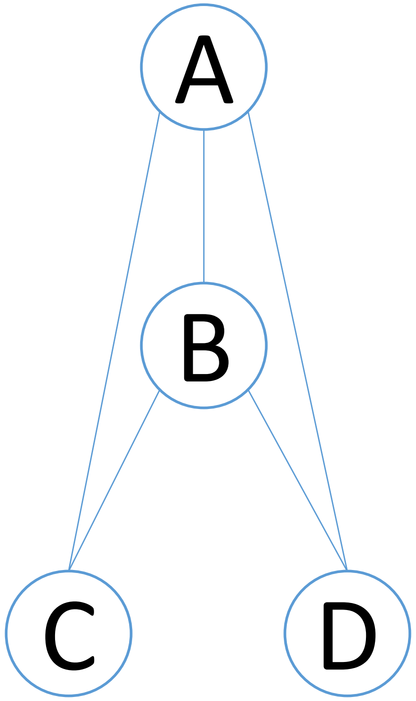



Stobb, Michael & Peterson, Joshua & Mazzag, Bori & Gahtan, Ethan. (2012). Graph Theoretical Model of a Sensorimotor Connectome in Zebrafish. PloS one.

# Path

- A path from a vertex u to a vertex v is defined either as a sequence of vertices in which each vertex is linked to the next, {u,u1,u2,. . .,uk ,v}.
  - A path that does not contain any repetition in either the edges or the nodes is called a simple path.
  - The length of a path is the number of edges in the sequence that comprises this path.
  - The distance between a pair of vertices u and v is defined as the number of edges along the shortest path connecting u and v.
  - Average path length is the average of the distances between all pairs of vertices in the graph

- From A to C, there are multiple path, such as {A,B,C} and {A, C}.
- The path length of {A,B,C} is 2.
- The distance between A and C is 1.



# Finding Simple Paths

- Simple paths between two nodes in a graph can be generated by calling function all_simple_paths(G, source, target)
  - G should be a Graph or DiGraph object.
  - source should be the starting node.
  - target should be the ending node.
  - An iterable container of paths will be returned.

```
paths = all_simple_paths(net, 1, 3)
list(paths)
```

# Finding Shortest Paths

- There two functions that can be used to find shortest paths: **shortest_path(G, source, target)** and **all_shortest_paths(G, source, target)**. The former only returns one shortest path between the source node and ending node, while the latter returns all shortest paths.
  - Both source and target are optional: if not provided, the function will use all nodes as source nodes or target nodes, whichever is omitted. They can be omitted simultaneously.
  - A list or dictionary of paths will be returned, depending on whether both source and target are specified. If not, use the node as the key for the returned dictionary.

```
paths = shortest_path(net, 1, 3)
paths
```

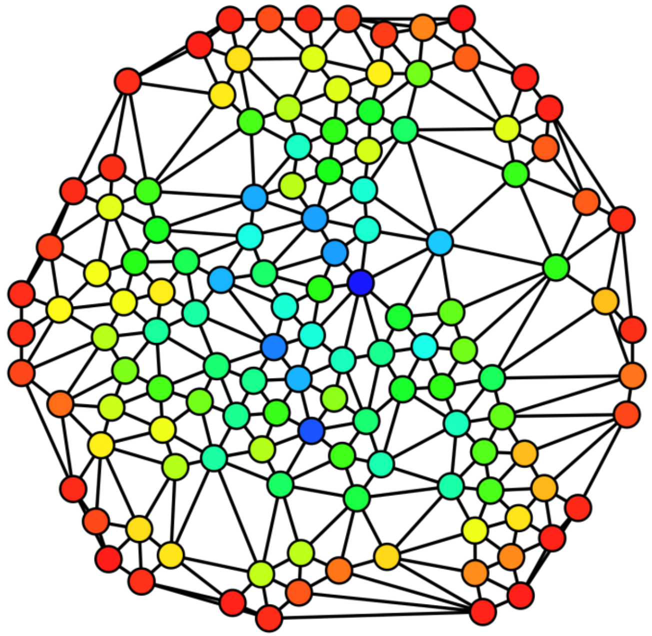
# Calculating Distance

- Distance between two nodes can be calculated by calling **shortest_path_length(G, source, target)**, which behaves similarly to **shortest_path()**, and source and target are optional.
- **average_shortest_path_length(G)** can be handy if we are only interested in the average distance of the network.

# Why do we care about path?

- Betweenness centrality is a measure of centrality in a graph based on shortest paths.
- It is the number of the shortest paths that pass through the vertex.
- It means connectivity and captures the indirect interactions in a network, and individual nodes benefit from indirect relationships because friends might provide access to favors from their friends and information might spread through the links of a network.
- Example:

  The difference between bid amounts and the secret reserve price decreases with increasing betweenness centrality. (Hinz and Spann 2008)



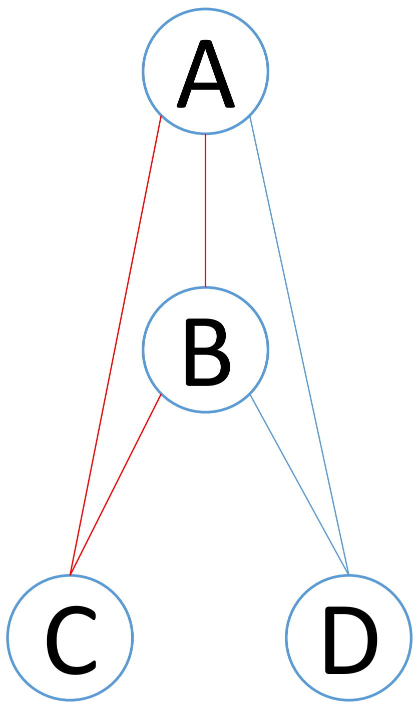
# Betweenness Centrality

- We can use function betweenness_centrality(G, k, normalized) to compute the betweenness centrality for all the nodes in a graph.
  - k can be passed with an integer to specify the number of node sample to estimate the betweenness.
  - Set normalized to **True** will normalize the betweenness values.
  - A dictionary of nodes with betweenness centrality as the value will be returned.

```
BW_centrality = nx.betweenness_centrality(G)
```

# Clustering coefficient

- Clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together.
- A triplet consists of three nodes that are connected by either two (open triplet) or three(closed triplet) undirected ties.
- The global clustering coefficient is the number of closed triplets over the total number of triplets (both open and closed).
- The local clustering coefficient for a node is defined as the ratio of existing links to the maximum number of possible links between the neighbors of that node.



# Computing Clustering Coefficient

- We can use function **clustering(G, nodes)** to compute the clustering coefficient for the nodes in a graph.
  - List of nodes can be passed to nodes to compute subset of nodes.
  - A dictionary of nodes with clustering coeeficient as the value will be returned.

  ```
  coef = nx.clustering(G)
  ```

- **average_clustering(G, nodes)** can be used to compute the average clustering coefficient for the graph.

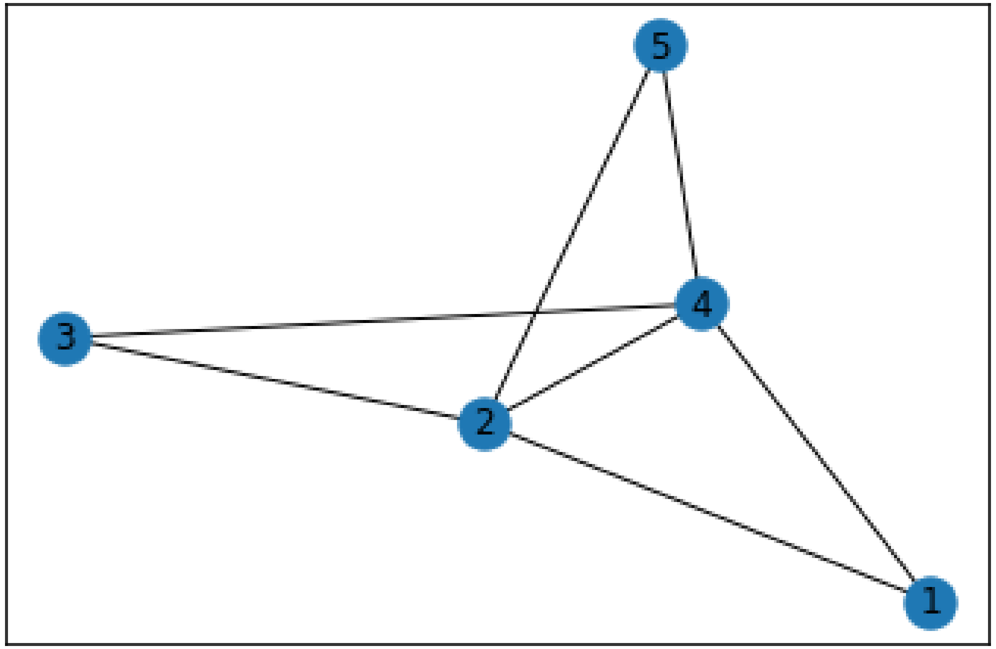
# Why do we care about clustering coefficient?

- Identify communities and groups within the network.
- Example:
  - greater-than-expected clustering coefficient observed in consumer-product network, suggesting a non random network.

    Huang, Zan, Daniel D. Zeng, and Hsinchun Chen. "Analyzing consumer-product graphs: Empirical findings and applications in recommender systems." Management science 53.7 (2007): 1146-1164.

# Visualization in NetworkX

- draw()
- Different layouts:
  - draw_circular()
  - draw_random()
  - draw_spring()
- Best for smaller networks



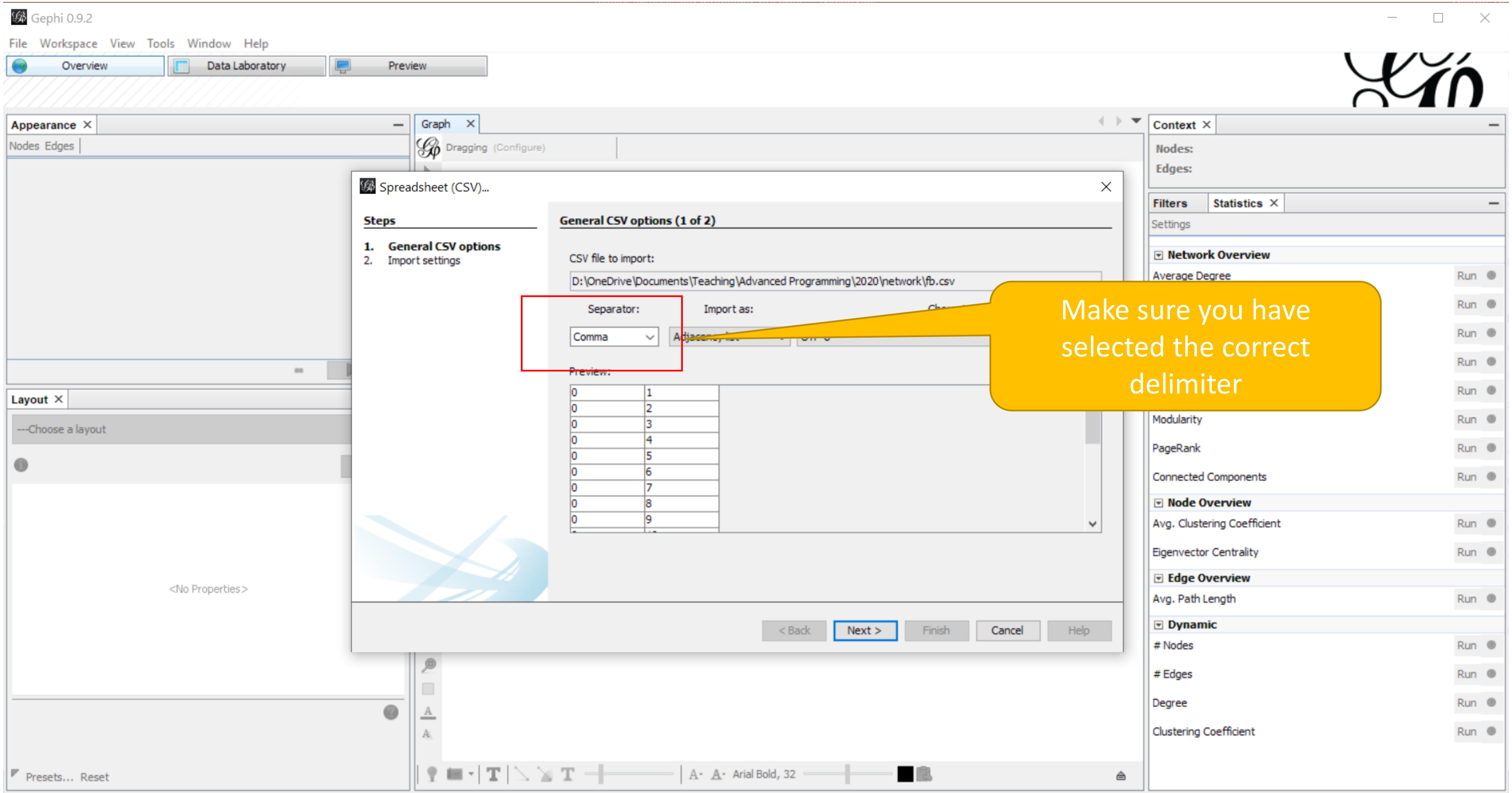

https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html

# Case Study: Facebook Network

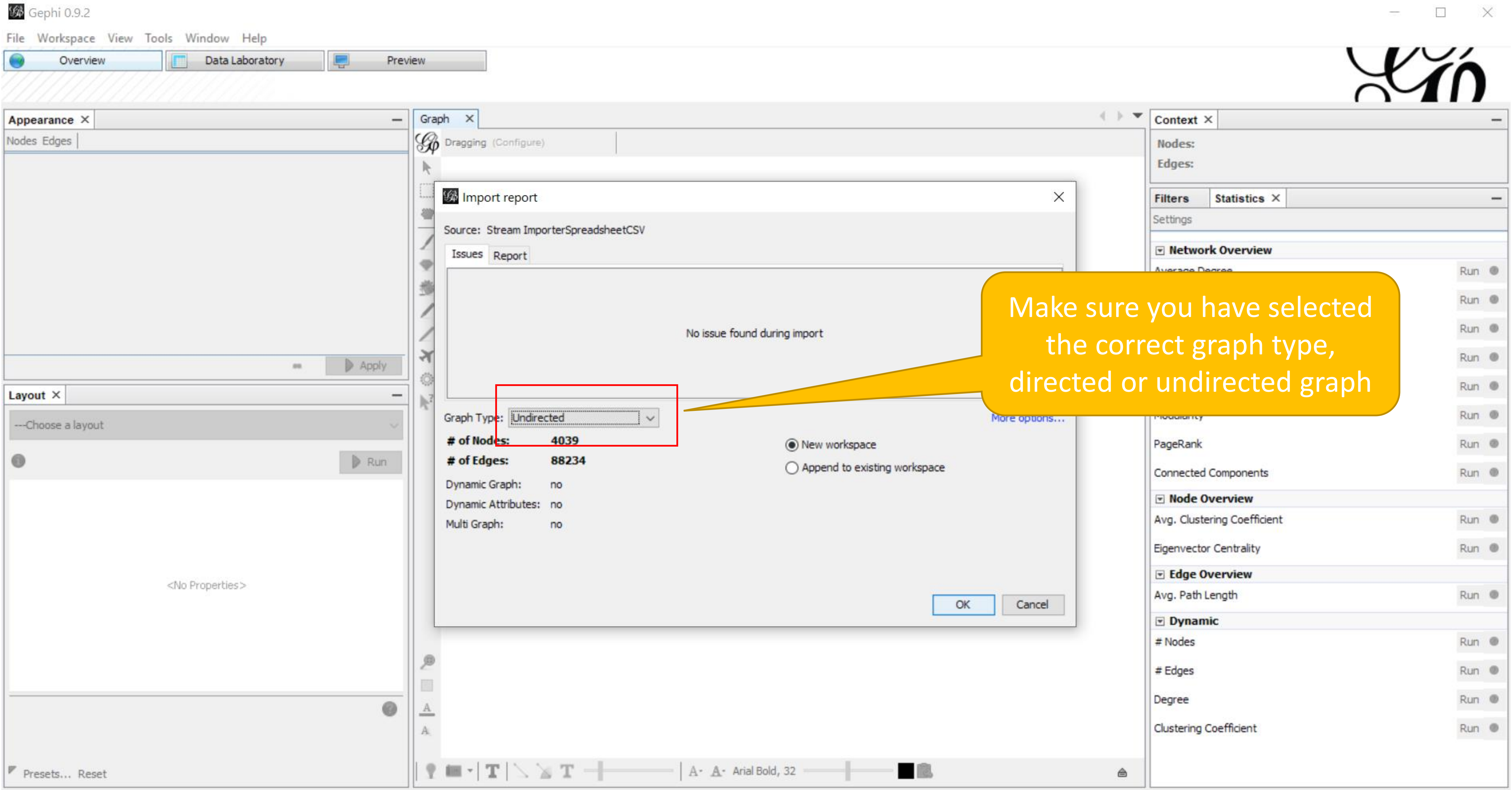
- What is the structure of social network on Facebook?
- What are the communities in Facebook network?
- How can we identify key opinion leaders and influencers?

# Gephi

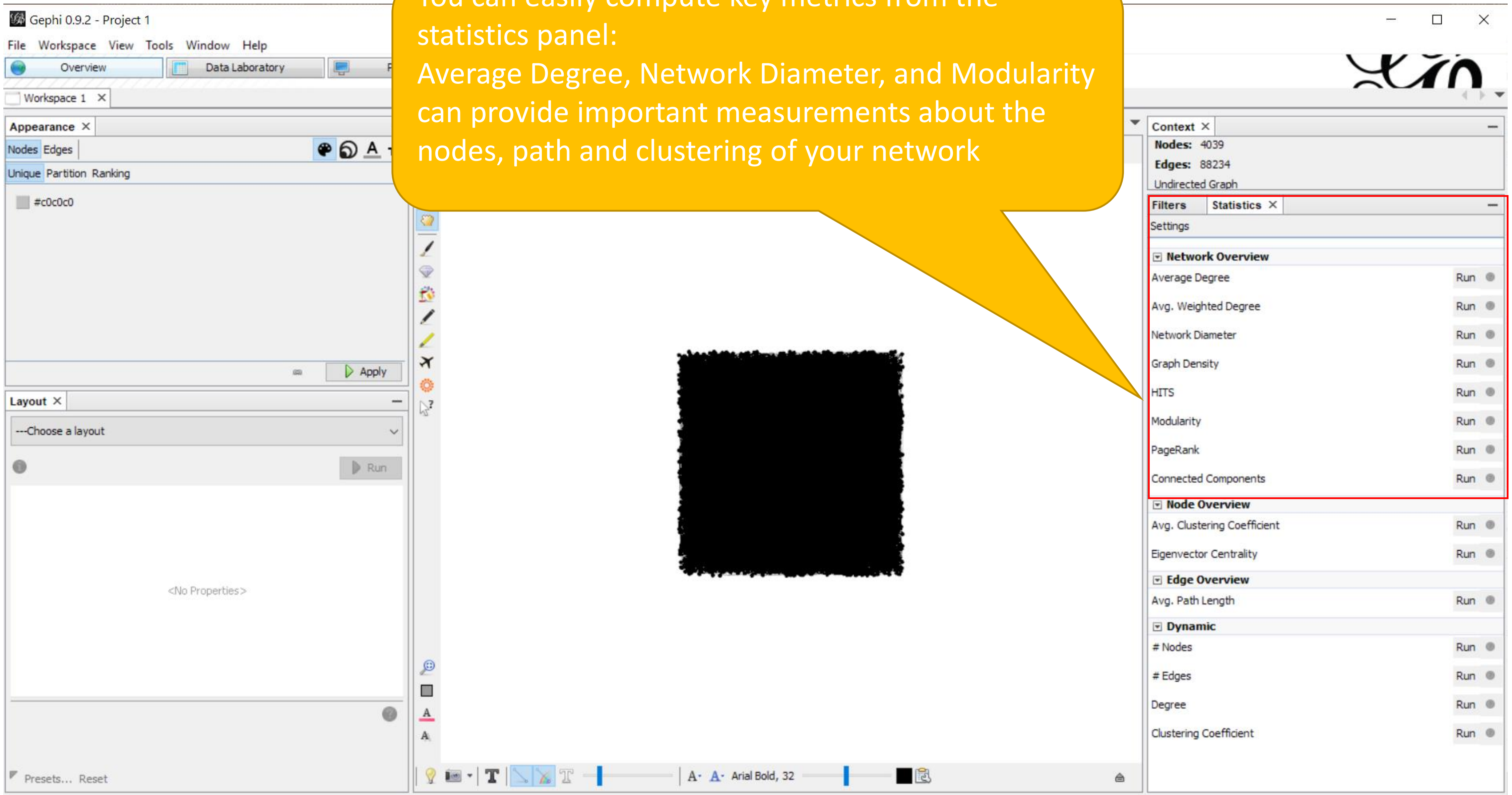
- Open source software for network analysis and visualization.
- User-friendly interface
- Fancy visualization



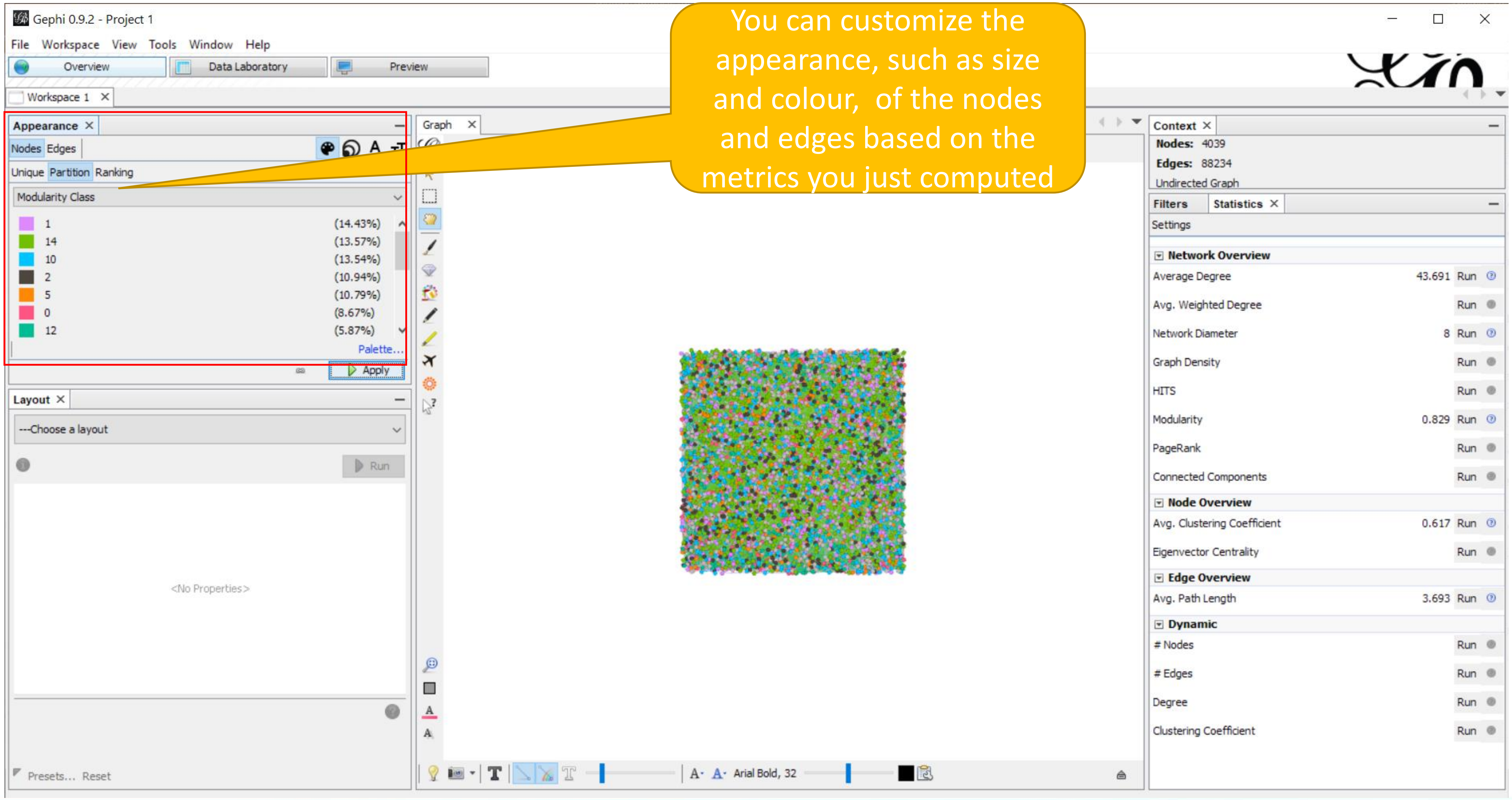

Make sure you have selected the correct delimiter

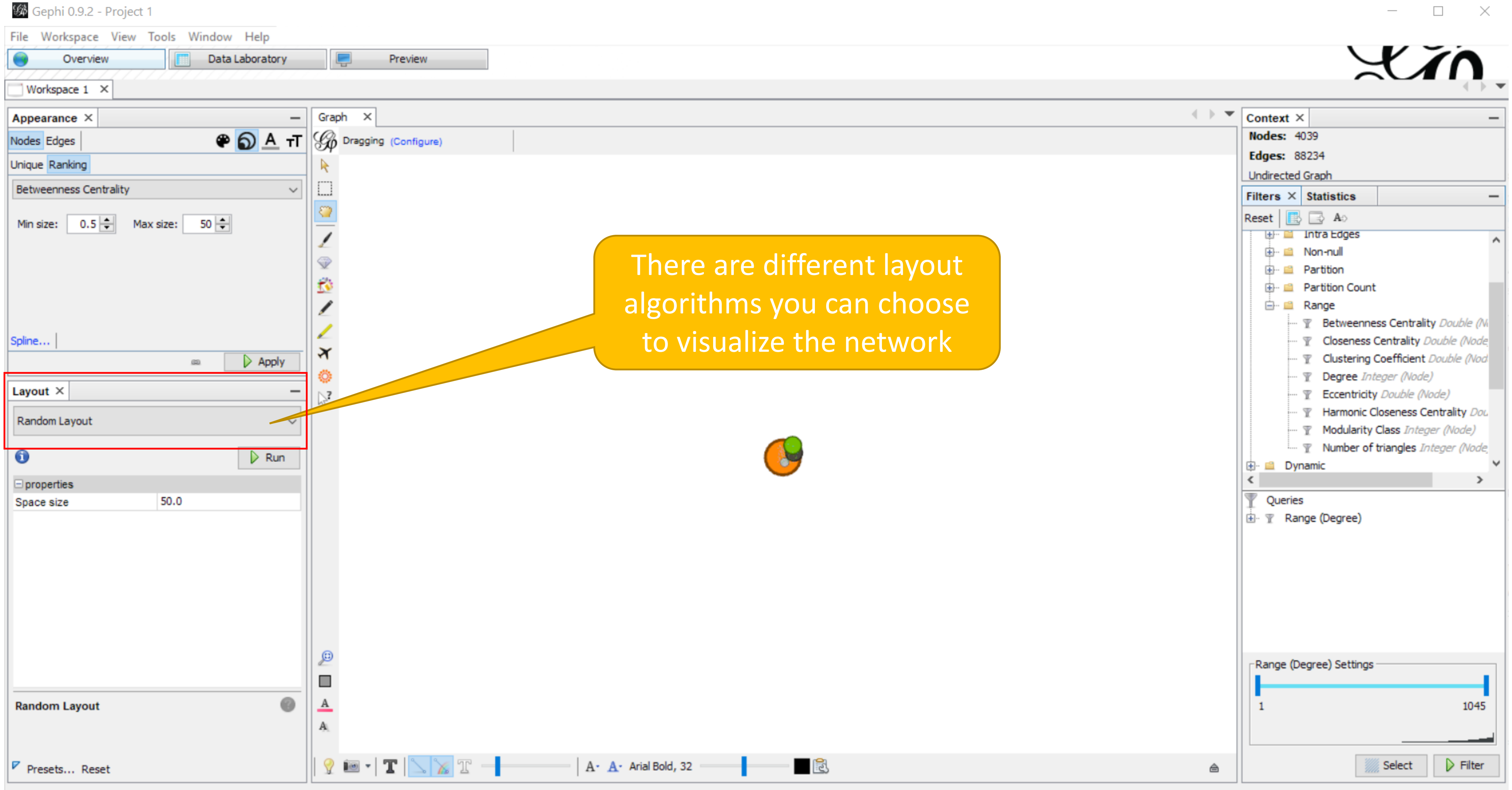

Gephi 0.9.2
File Workspace View Tools Window Help
Overview
Data Laboratory
Preview
Appearance
Nodes Edges
Apply
Layout
---Choose a layout
Run
<No Properties>
Presets... Reset
Graph
Dragging (Configure)
Import report
Source: Stream ImporterSpreadsheetCSV
Issues
Report
No issue found during import
Graph Type: Undirected
More options...
# of Nodes: 4039
# of Edges: 88234
Dynamic Graph: no
Dynamic Attributes: no
Multi Graph: no
New workspace
Append to existing workspace
OK
Cancel
Arial Bold, 32
Context
Nodes:
Edges:
Filters
Statistics
Settings
Network Overview
PageRank
Connected Components
Node Overview
Avg. Clustering Coefficient
Eigenvector Centrality
Edge Overview
Avg. Path Length
Dynamic
# Nodes
# Edges
Degree
Clustering Coefficient
Run
Make sure you have selected the correct graph type, directed or undirected graph

You can easily compute key metrics from the statistics panel:
Average Degree, Network Diameter, and Modularity can provide important measurements about the nodes, path and clustering of your network



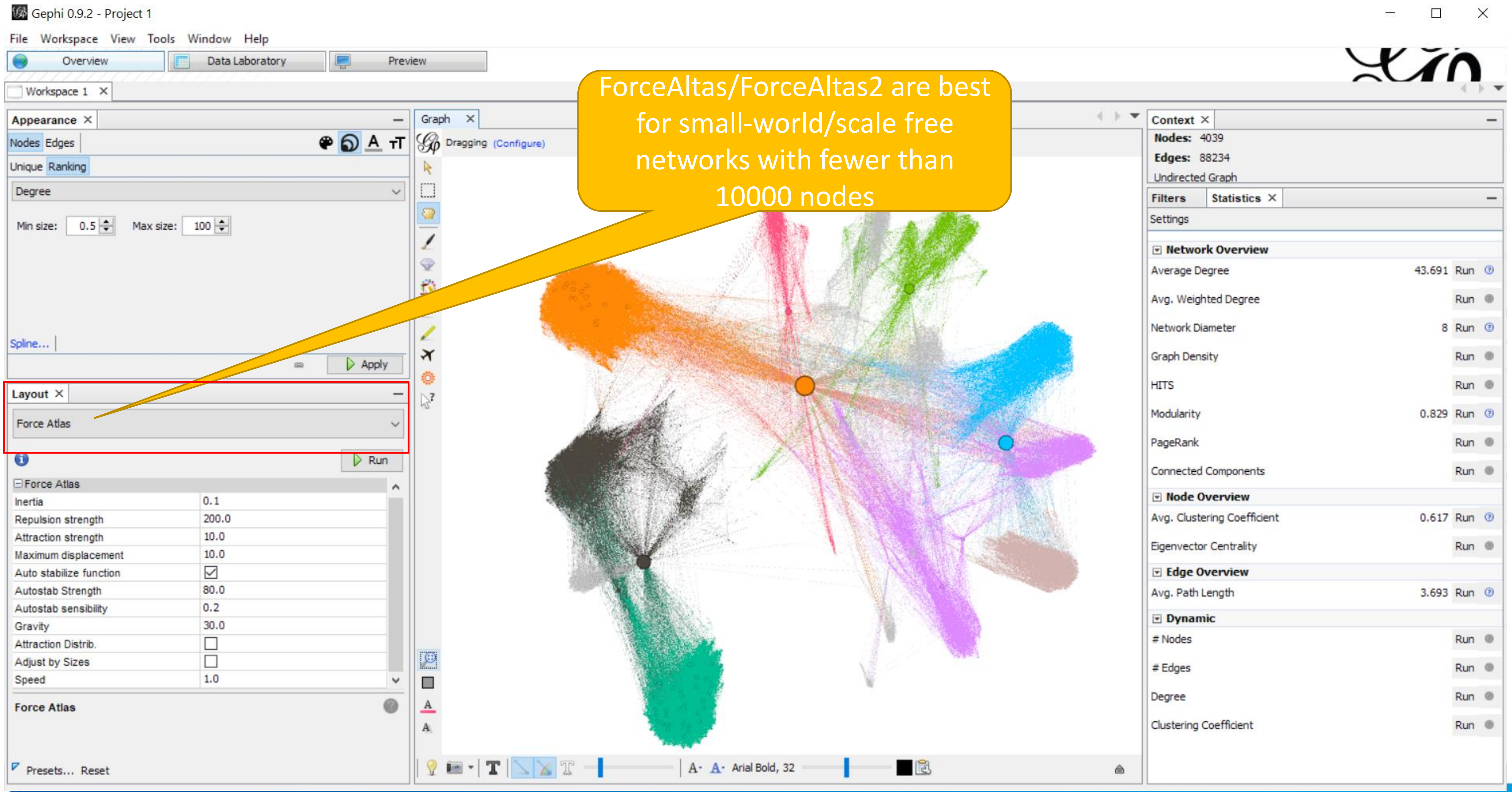
You can customize the appearance, such as size and colour, of the nodes and edges based on the metrics you just computed



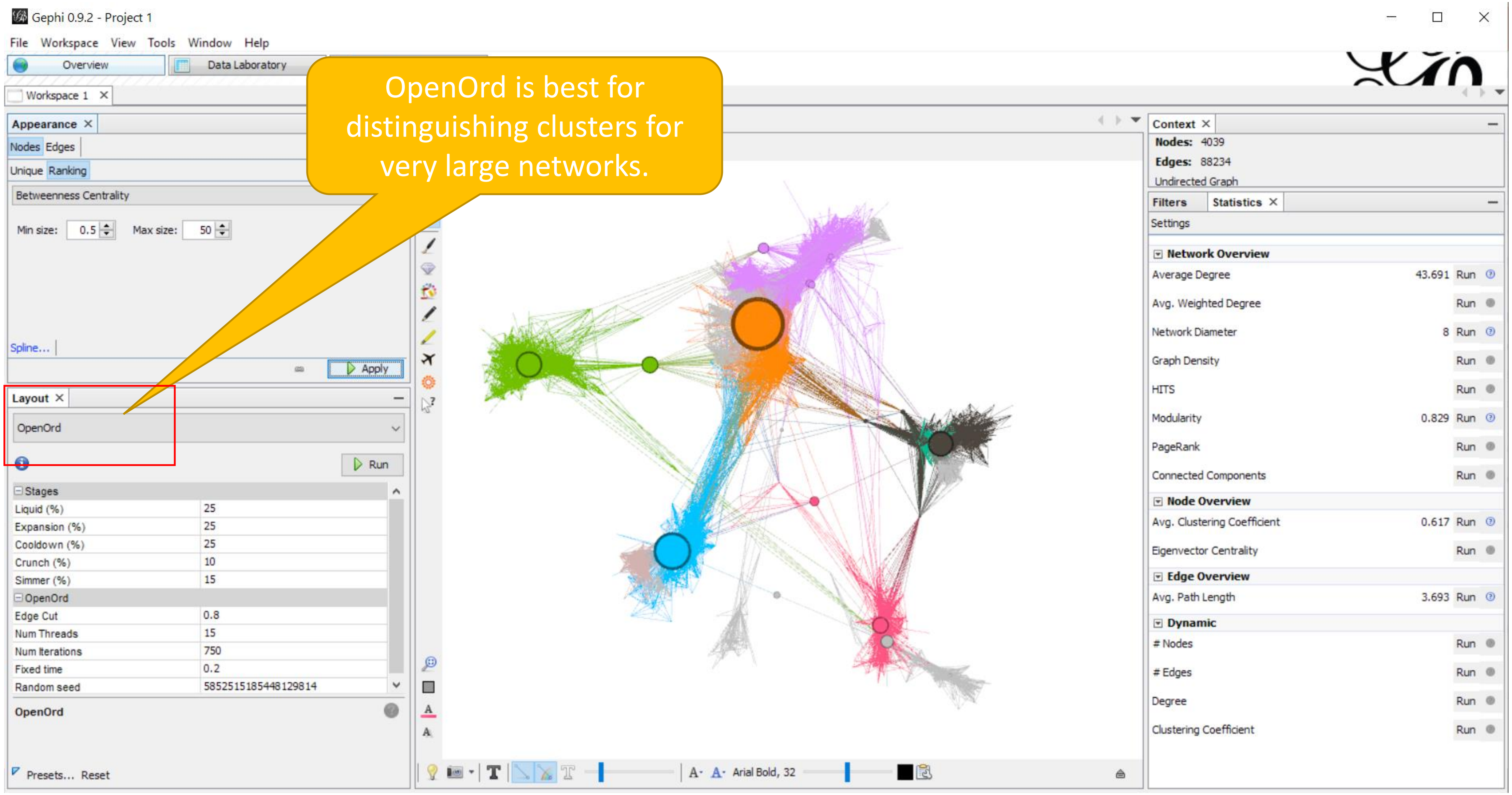


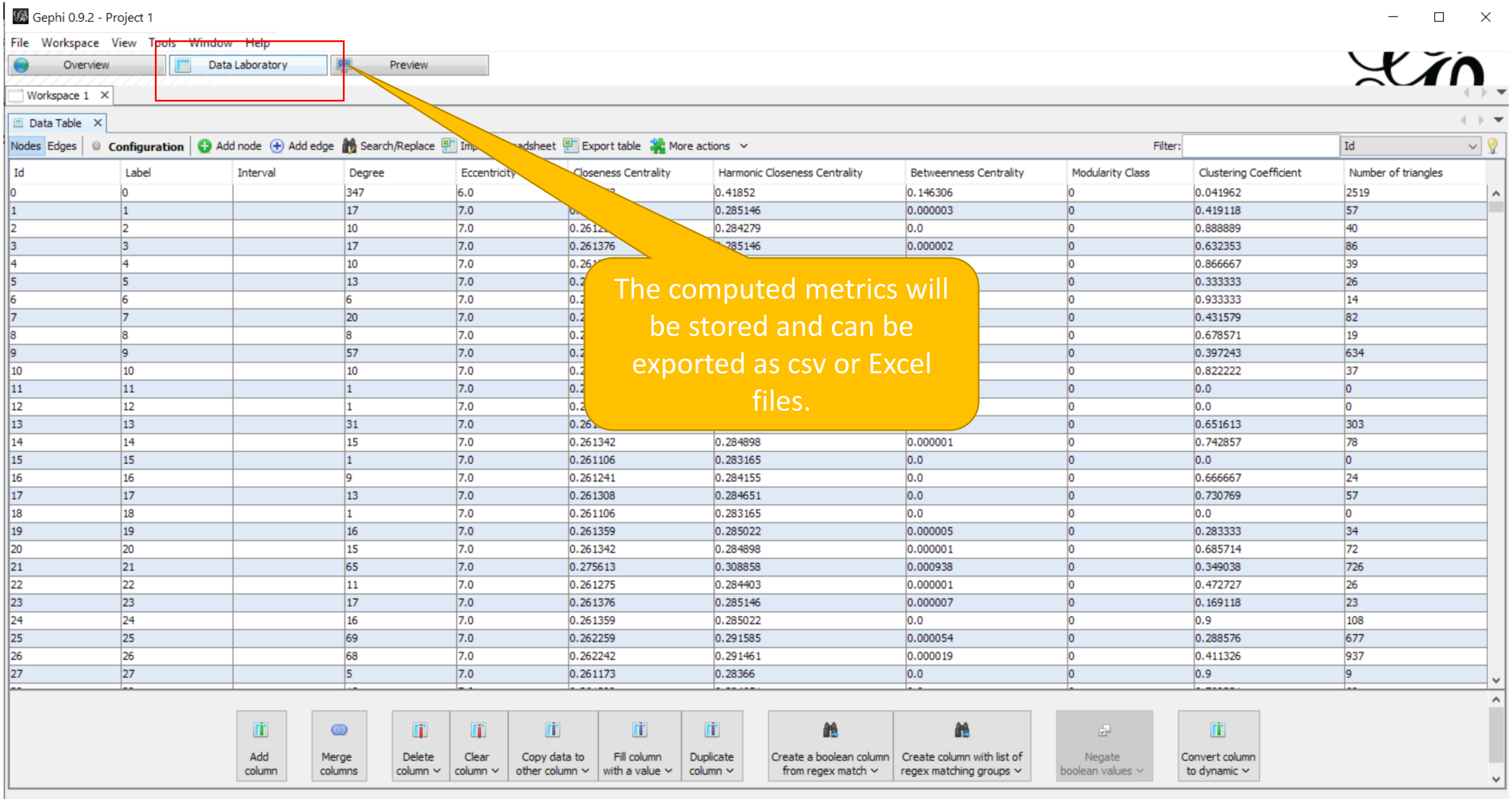

There are different layout algorithms you can choose to visualize the network

ForceAltas/ForceAltas2 are best for small-world/scale free networks with fewer than 10000 nodes



OpenOrd is best for distinguishing clusters for very large networks.






The computed metrics will be stored and can be exported as csv or Excel files.

# Social Network Analysis Tools

- NetworkX, a Python module.

 (https://www.datacamp.com/community/tutorials/social-network-analysis-python)

Snap.py

- Gephi, a standalone software.

 (https://gephi.org/)

- Pajek, a standalone software.

 (http://mrvar.fdv.uni-lj.si/pajek/)

# Exercise

- Try to analyze Facebook example using Python in Jupyter Notebook.