Graph Visualization

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실 NETWORK SCIENCE LAB

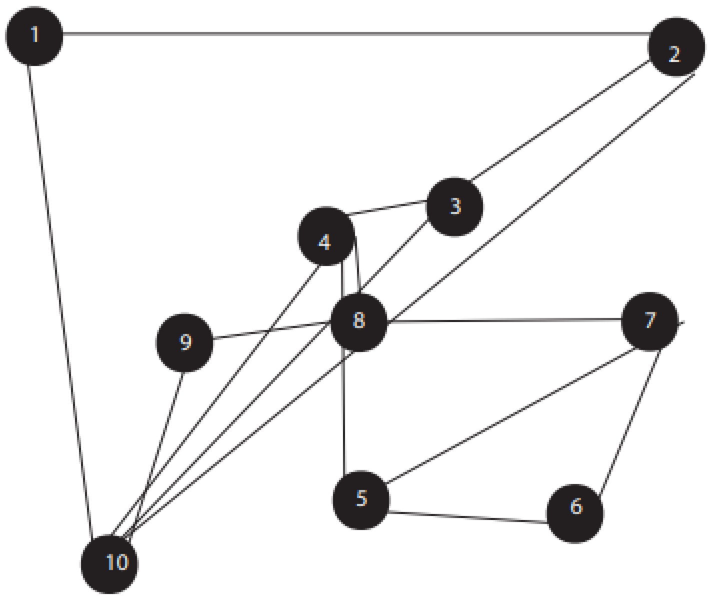
가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA

Contents

- Visualization techniques:
  - Spring-embedded
  - Circular, etc.
- Tools for graph exploration and visualization:
  - Gephi, Cytoscape, etc.

General Layout Problem

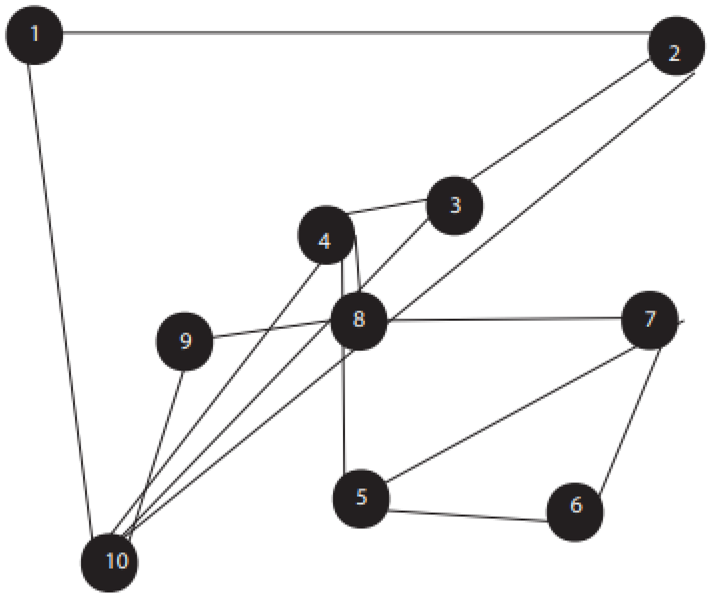
- Input: Graph G = (V, E)

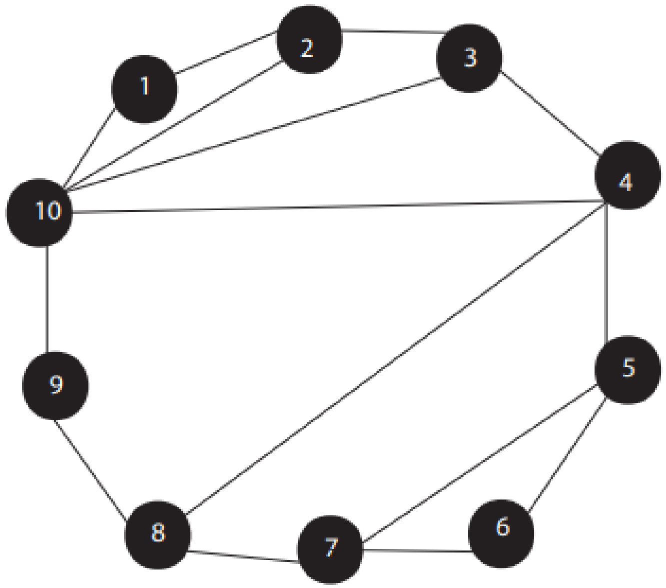


https://www.csd.uoc.gr/~hy583/reviewed_notes/circular.pdf

General Layout Problem

- **Input**: Graph G = (V, E)
- **Output**: Clear and readable drawing of G

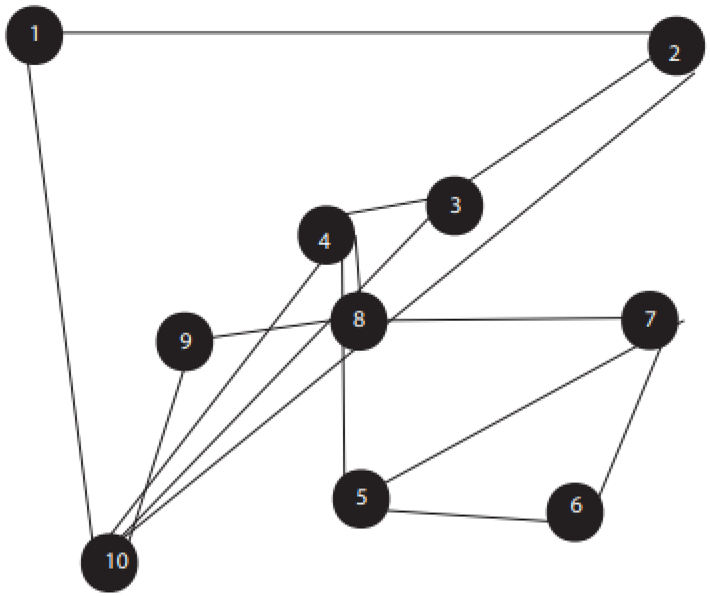


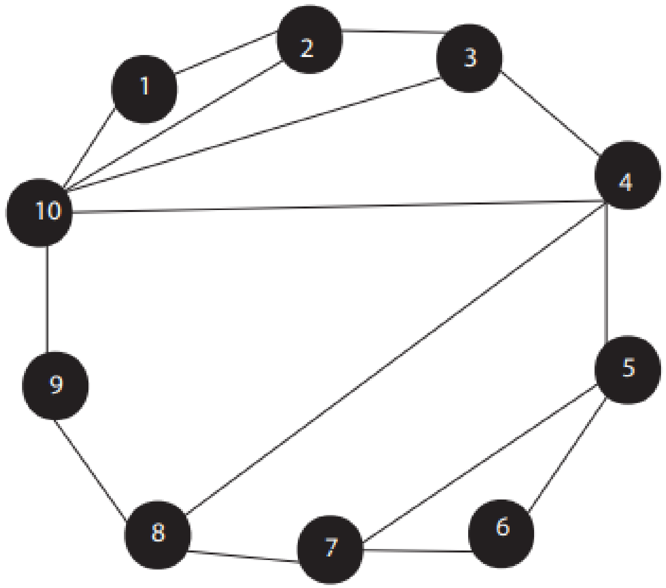


https://www.csd.uoc.gr/~hy583/reviewed_notes/circular.pdf

General Layout Problem

- Input: Graph G = (V, E)
- Output: Clear and readable drawing of G





- Which criteria would you optimize?

https://www.csd.uoc.gr/~hy583/reviewed_notes/circular.pdf

General Layout Problem

- Input: Graph G = (V, E)
- Output: Creating clear and readable drawings of graph G.
- Criteria:
  - Adjacent nodes are close.
  - Non-adjacent nodes are far.
  - The preservation of edge length.
  - Densely connected nodes tend to close.
  - Draw G with as few crossings as possible.

https://www.csd.uoc.gr/~hy583/reviewed_notes/circular.pdf

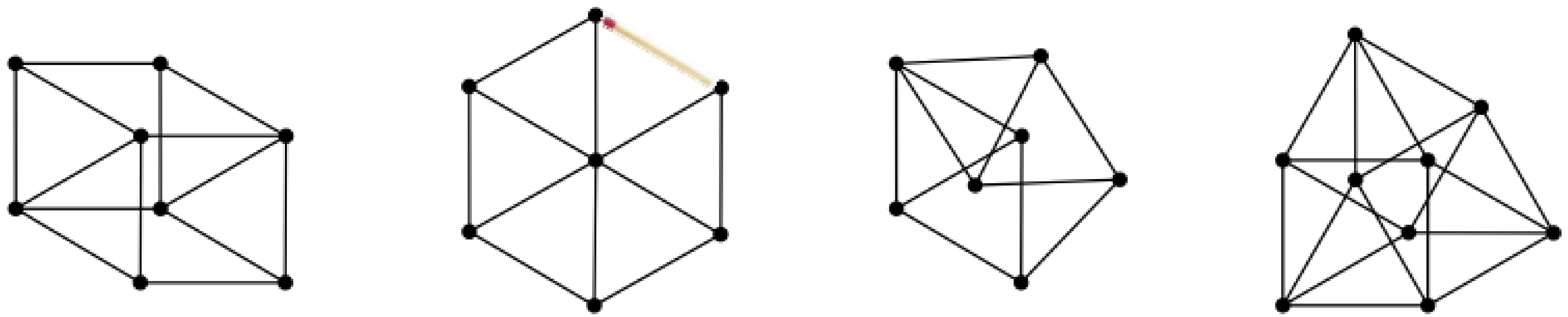
General Layout Problem

- Input: Graph G = (V, E)
- Output: Creating clear and readable drawings of graph G.
- Criteria:
  - Adjacent nodes are close.
  - Non-adjacent nodes are far.
  - The preservation of edge length: similar length.
  - Densely connected nodes tend to close.
  - Draw G with as few crossings as possible.

Let's take an example

Fixed edge lengths?

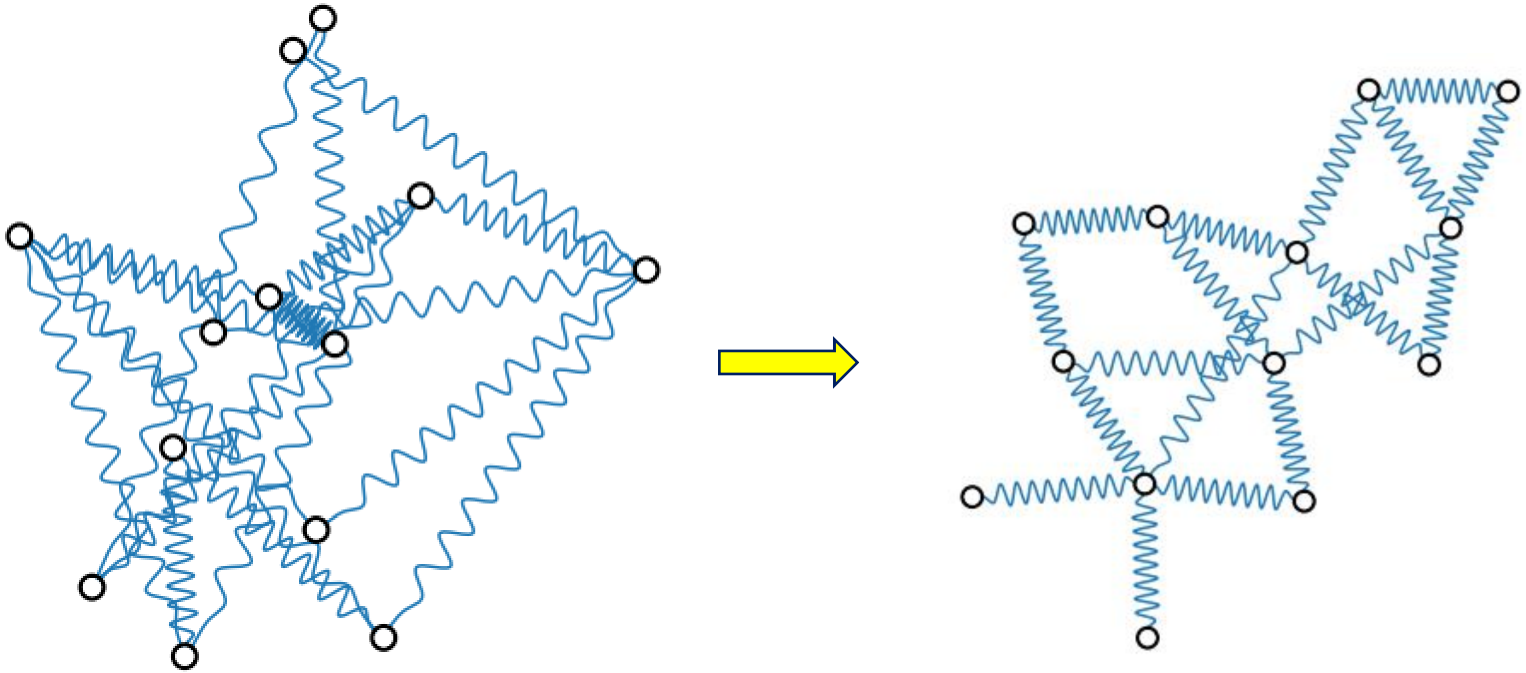
- **Input:** Graph G = (V, E), required edge length $l(e)$, $\forall e \in E$
- **Output:** Drawing of G which realizes all the edge lengths



- NP-hard problem for:
  - Uniform edge lengths in any dimension.
  - Uniform edge lengths in planar drawing.

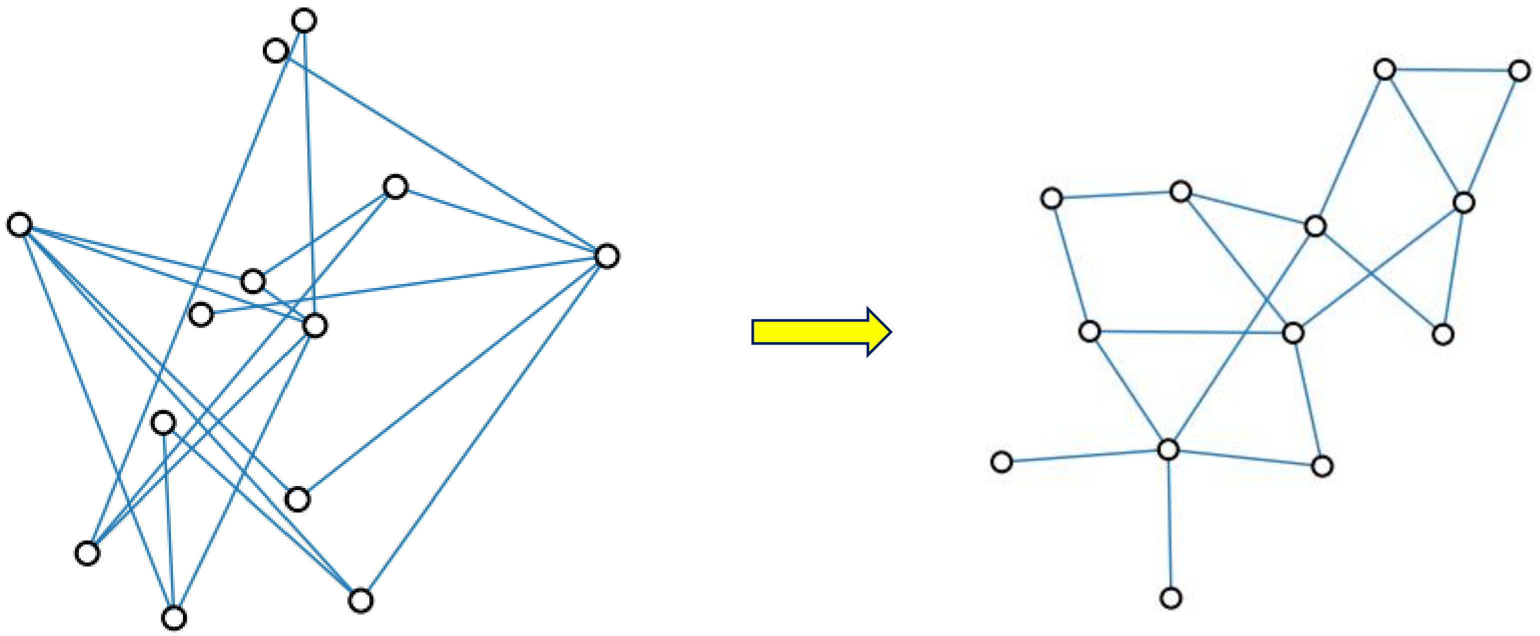
1. Spring-embedded: The main idea

- To embed a graph, we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system.
  - The nodes are placed in some initial layout.
  - The spring forces on the rings move the system to a minimal energy state.



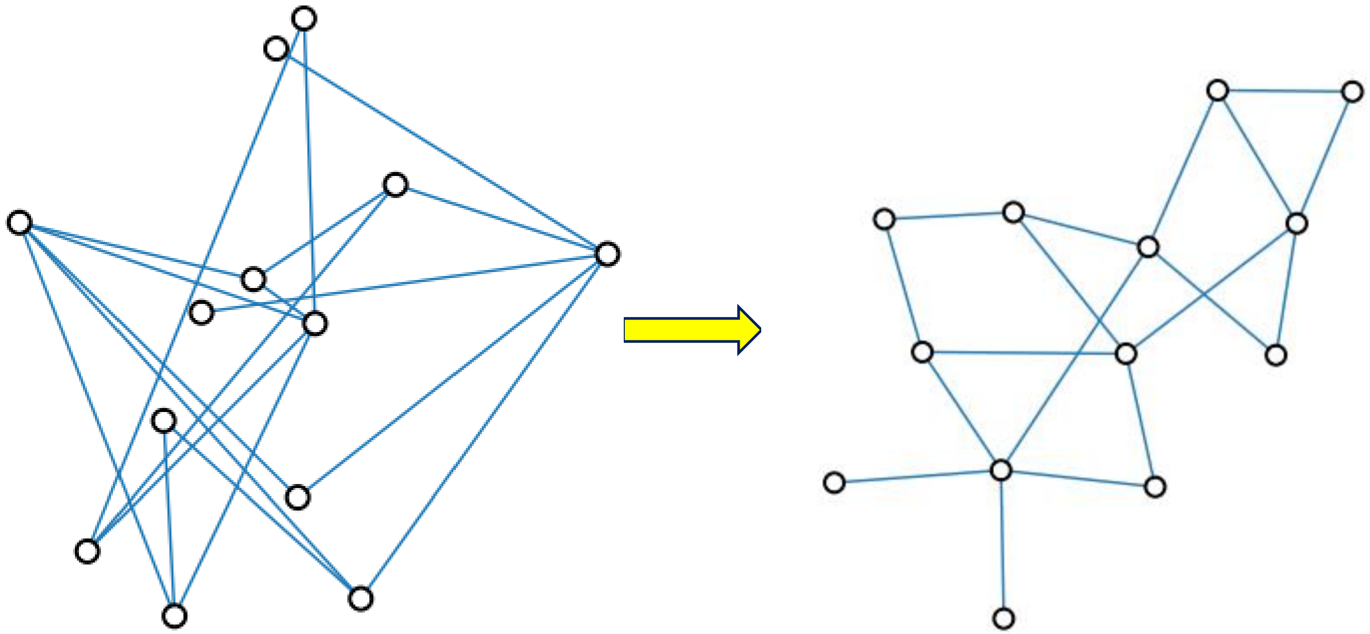
1. Spring-embedded: The main idea

- To embed a graph, we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system.
  - The nodes are placed in some initial layout.
  - The spring forces on the rings move the system to a minimal energy state.



Spring-embedded: diea, constraints

- To embed a graph, we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system.
  - The nodes are placed in some initial layout.
  - The spring forces on the rings move the system to a minimal energy state.



- Adjacent nodes u and v: f_{spring}

- Repulsive forces:

  non-adjacent nodes u and v: f_{rep}

Spring-embedded by Eades – main functions

- Repulsive force between two non-adjacent node pairs v_i and v_j:

$$f_{rep}(p_i, p_j) = \frac{c_{rep}}{||p_i - p_j||^2} \cdot \vec{p_i p_j}$$

- Attractive force between two adjacent vertices v_i and v_j :

$$f_{spring}(p_i, p_j) = c_{spring} \log \frac{||p_i - p_j||}{l} \cdot \vec{p_i p_j}$$

- Resulting displacement vector for node v_i

$$F_i(t) \leftarrow \sum_{(v_i, v_j) \notin E} f_{rep}(p_j, p_i) + \sum_{(v_i, v_j) \in E} f_{spring}(p_j, p_i)$$

Where:

- $l = l(e)$: the ideal spring length of edge e.
- $||p_i - p_j||$: Distance between v_i and v_j.
- $\vec{p_i p_j}$: unit vector pointing from v_i to v_j.
- c_{rep}: repulsion constant (e.g. 1.0).
- c_{spring}: spring constant (e.g. 2.0)

[Fruchterman, Reingold 1991] Graph drawing by force-directed placement

Spring-embedded by Eades – Algorithm

Initial layout with random positions of nodes in the layout

Algorithm 1: SpringEmbedder

$G = (V, E)$, $\boxed{p = (p_i)}$, $v_i \in V, \epsilon > 0, K \in N$

Input: p: initial layout, ϵ: threshold

Output: p: is end layout

1

2

3

4

5

6

7

8 Return p

End layout

- Spring forces:
  - Adjacent nodes u and v: f_{spring}
- Repulsive forces:
  - non-adjacent nodes u and v: f_{rep}

[Eades 1984] A heuristic for graph drawing

Spring-embedded by Eades – Algorithm

Initial layout with random positions of nodes in the layout

Algorithm 1: SpringEmbedder
$G = (V, E)$, $\boxed{p = (p_i)}$, $v_i \in V, \epsilon > 0, K \in N$

Input: p: initial layout, ϵ: threshold
Output: p: is end layout

1 $t \leftarrow 1$
2
3
4
5
6
7
8 Return p

End layout

- Spring forces:
  Adjacent nodes u and v: f_{spring}
- Repulsive forces:
  non-adjacent nodes u and v: f_{rep}

[Eades 1984] A heuristic for graph drawing

Spring-embedded by Eades – Algorithm

Initial layout with random positions of nodes in the layout

Algorithm 1: SpringEmbedder

$G = (V, E)$, $\boxed{p = (p_i)}$, $v_i \in V, \epsilon > 0, K \in N$

Input: p: initial layout, ϵ: threshold

Output: p: is end layout

$t \leftarrow 1$

while $t < K$ *and* $\boldsymbol{MAX}_{v_i \in V} ||F_i(t)|| > \epsilon$ **do**

  for $v \in V$ **do**

    $F_i(t) \leftarrow \sum_{(v_i,v_j) \notin E} f_{rep}(p_j, p_i) + \sum_{(v_i,v_j) \in E} f_{spring}(p_j, p_i)$

  for $v \in V$ **do**

    $p_i \leftarrow p_i + \delta(t) \cdot F_{i(t)}$

  $t \leftarrow t + 1$

Return p

Update new location of node

cooling factor

End layout

- Spring forces:
  Adjacent nodes u and v: f_{spring}
- Repulsive forces:
  non-adjacent nodes u and v: f_{rep}

Where:

- $l = l(e)$: the ideal spring length of edge e.
- $||p_i - p_j||$: Distance between v_i and v_j.
- $\vec{p_i p_j}$: unit vector pointing from v_i to v_j.
- c_{rep}: repulsion constant (e.g. 1.0).
- c_{spring}: spring constant (e.g. 2.0)

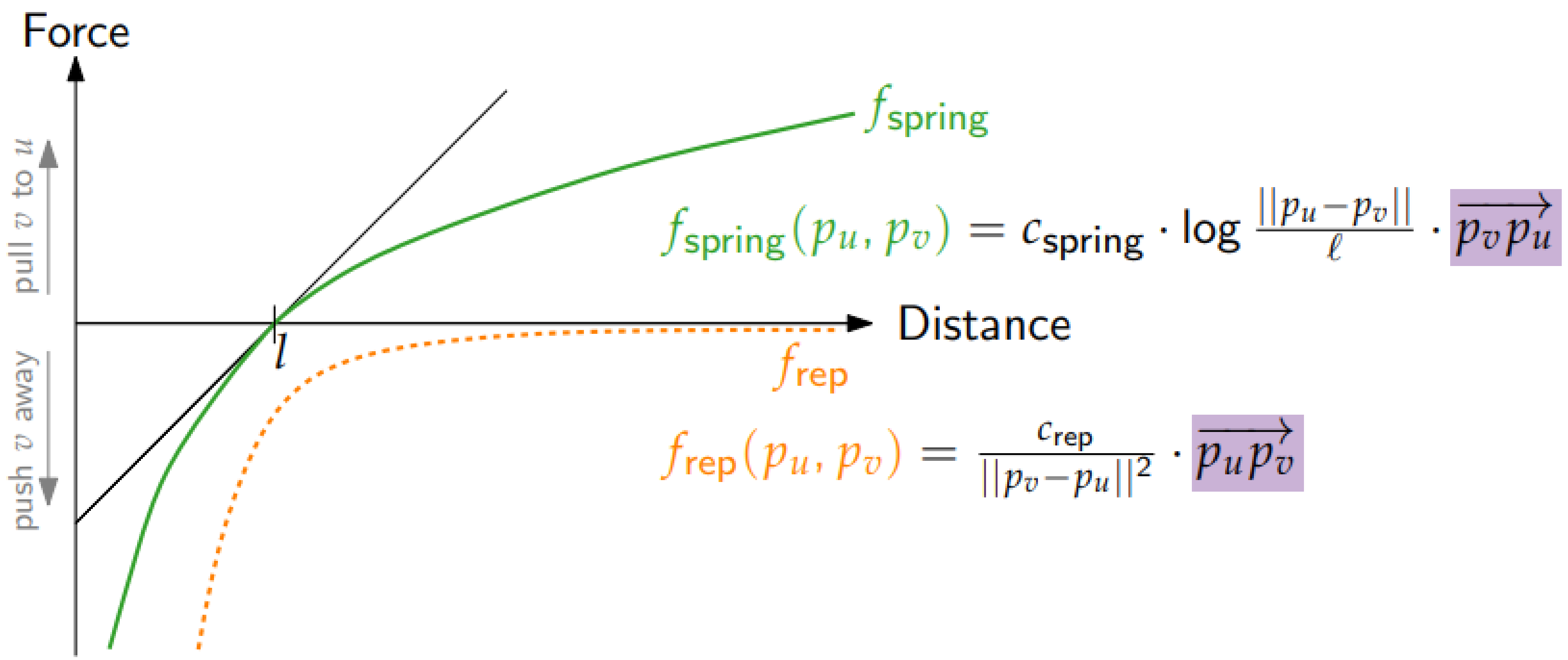
$$f_{rep}(p_i, p_j) = \frac{c_{rep}}{||p_i - p_j||^2} \cdot \vec{p_i p_j}$$

$$f_{spring}(p_i, p_j) = c_{spring} \log \frac{||p_i - p_j||}{l} \cdot \vec{p_i p_j}$$

[Eades 1984] A heuristic for graph drawing

Spring-embedded by Eades – Force diagram

- Spring forces (f_{spring}): pull node v close to node u (u and v are adjacent)
- Repulsive forces (f_{rep}): push node v far away node u (u and v are non-adjacent)



Spring-embedded by Eades – Discussion

- Advantages.
  - Simple algorithm.
  - Good results for small and medium-sized graphs.
  - Good representation of symmetry and structure.
- Disadvantages.
  - System is not stable at the end.
  - Converging to local minimal.

Variant by Fruchterman & Reingold

- Repulsive force between **all** vertex pairs v_i and v_j:

$$f_{rep}(p_i, p_j) = \frac{l}{||p_i - p_j||^2} \cdot \vec{p_i p_j}$$

- Attractive force between two adjacent vertices v_i and v_j :

$$f_{attactive}(p_i, p_j) = \frac{||p_i - p_j||^2}{l} \cdot \vec{p_i p_j}$$

- Resulting force between adjacent vertices v_i and v_j:

$$f_{spring}(p_i, p_j) = f_{rep}(p_i, p_j) + f_{attactive}(p_i, p_j)$$

Algorithm 1: SpringEmbedder

$G = (V, E), p = (p_i), v_i \in V, \epsilon > 0, K \in N$

Input: p: initial layout, ϵ: threshold

Output: p: is end layout

$t \leftarrow 1$

while $t < K$ *and* $\boldsymbol{MAX}_{v_i \in V} ||F_{i(t)}|| > \epsilon$ **do**

  for $v \in V$ **do**

    $F_i(t) \leftarrow \sum_{(v_i, v_j) \notin E} f_{rep}(p_j, p_i) + \sum_{(v_i, v_j) \in E} f_{spring}(p_j, p_i)$

  for $v \in V$ **do**

    $p_i \leftarrow p_i + \delta(t) \cdot F_{i(t)}$

  $t \leftarrow t + 1$

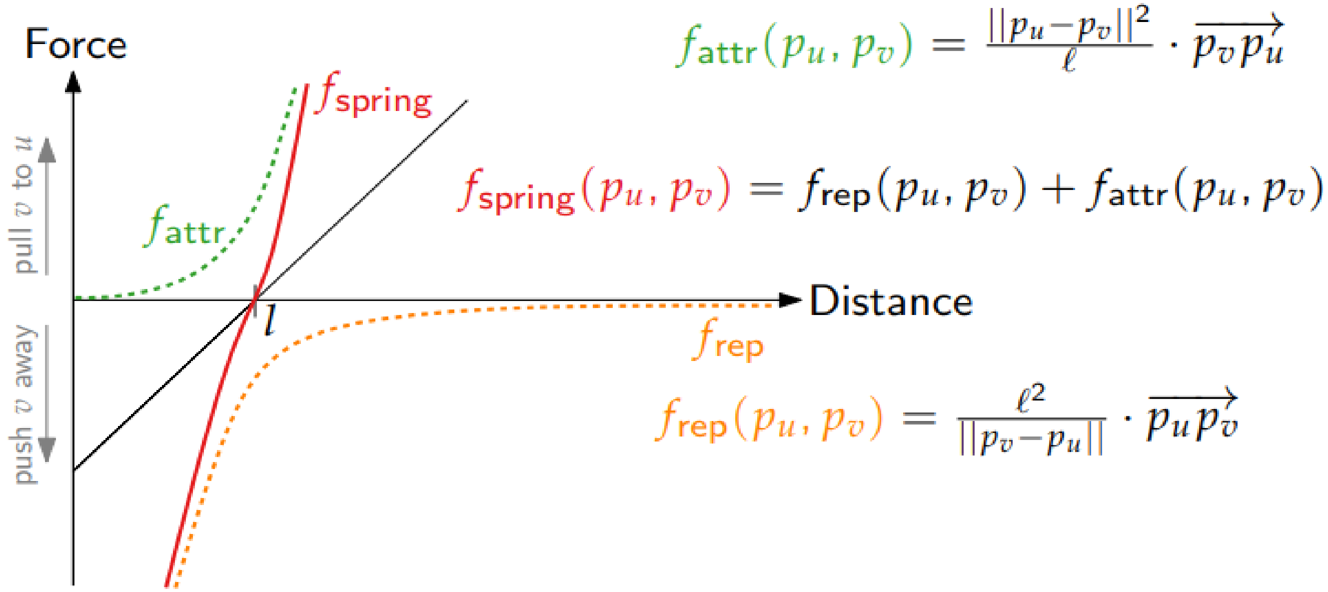
Return p

[Fruchterman, Reingold 1991] Graph drawing by force-directed placement

Variant by Fruchterman & Reingold

There are three forces:

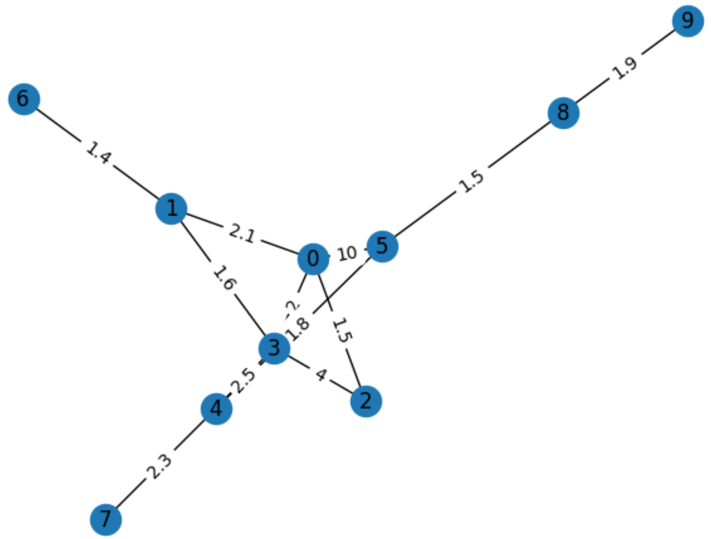
- Spring forces (f_{spring}): pull node v close to node u (u and v are adjacent)
- Attractive force between two adjacent nodes v_i and v_j (f_{attr}): pull node v close to node u (u and v are adjacent)
- Repulsive forces (f_{rep}): push node v far away node u (u and v are non-adjacent)



Sample code: Visualizing a graph using NetworkX

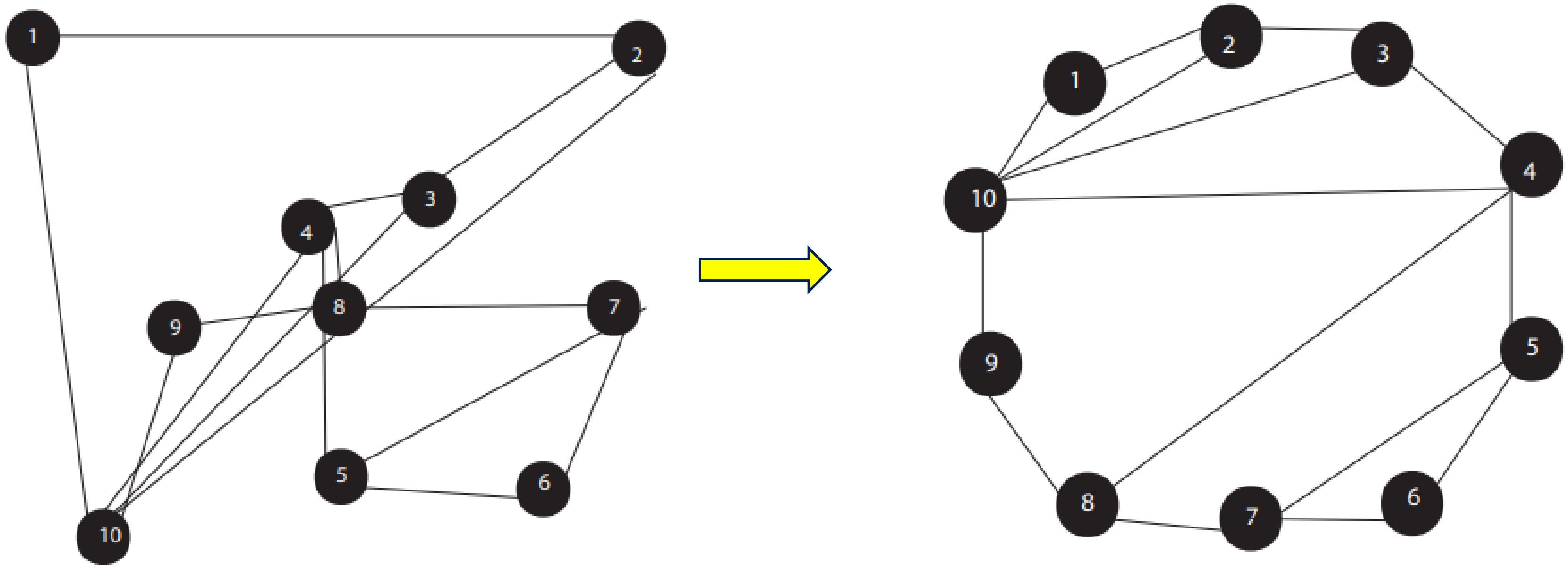
- Spring layout

```
# Instantiate the graph
G = nx.Graph()

edges = [(0, 1, 2.1), (0, 2, 1.5), (0, 3, 2), (0, 5, 10), (1, 3, 1.6), (1, 6, 1.4),
         (2, 3, 4), (3, 4, 2.5), (4, 5, 1.8), (4, 7, 2.3), (5, 8, 1.5), (8, 9, 1.9)]
# add node/edge pairs
G.add_weighted_edges_from(edges)

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels = True)

labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
```



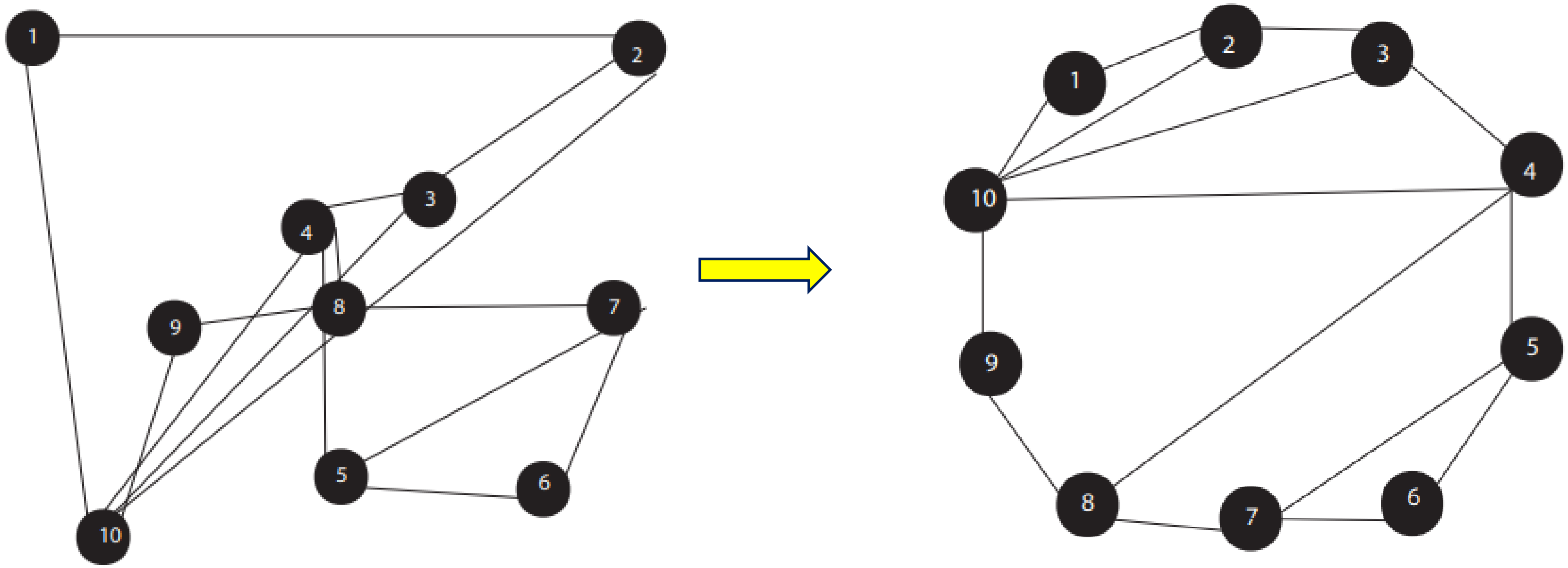
2. Circular

- **Input**: Graph G = (V, E)
- **Output**: Creating clear and readable drawings of graph G.



Algorithm Circular

- Input: A biconnected graph, ***G* = (*V*, *E*).**
- Output: A circular drawing ***Γ*** of ***G*** such that each node in ***V*** lies on the periphery of a single embedding circle.



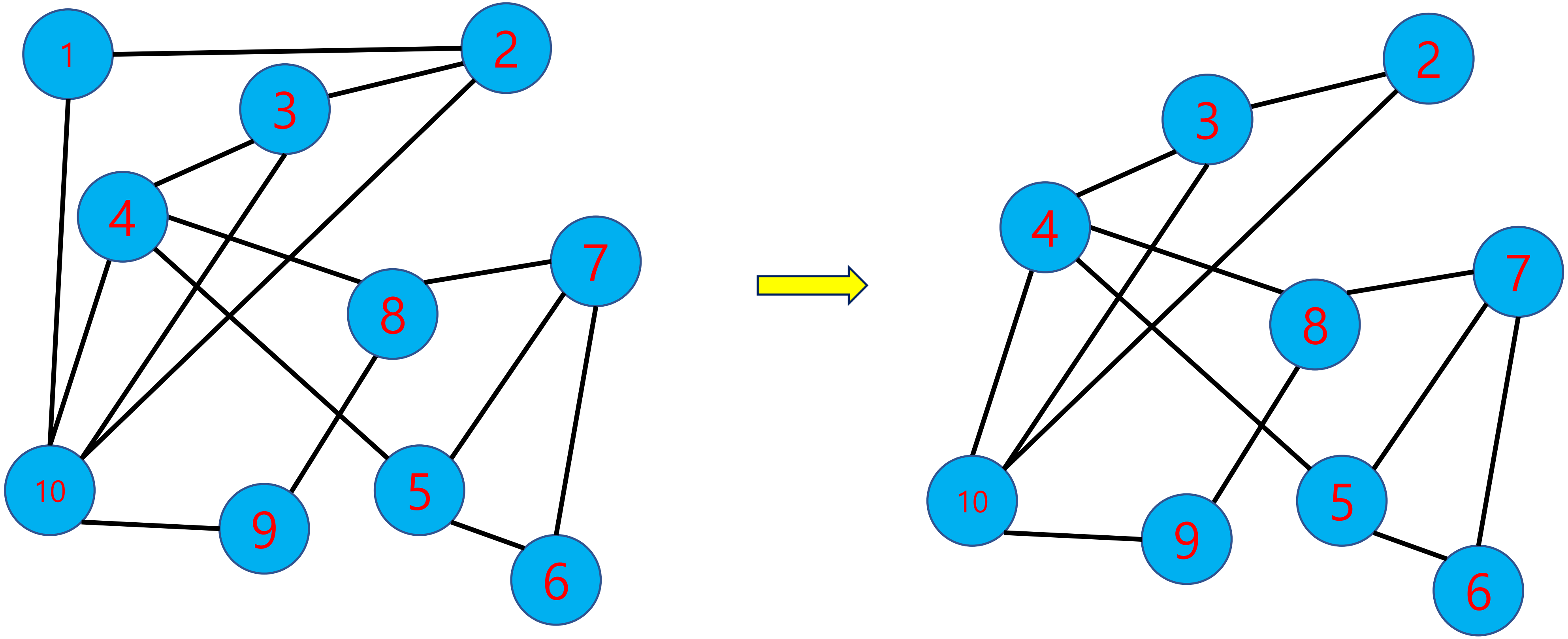
Algorithm Circular

- In circular graphs, close nodes should not be connected:
  - Idea: Finding and store nodes that have two non-connected neighbours by using BFS algorithm.
  - Implement:
    - Starting at a random node, store nodes that do not have non-connected neighbours in a stack.
    - Restore the graph to generate a circle graph

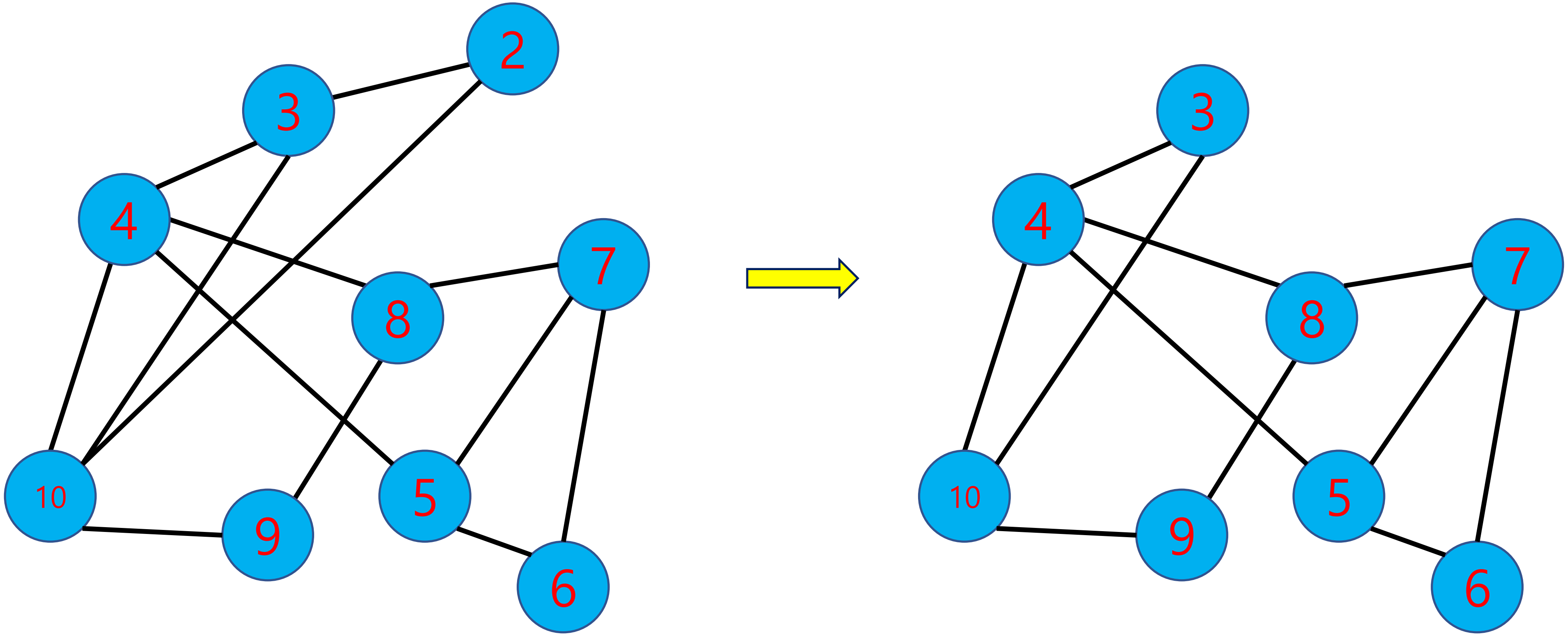
1. Bucket sort the nodes by ascending degree into a table T.
2. Set *counter* to 1.
3. While $counter \leq n - 3$
4. If a wave front node u has lowest degree then $currentNode = u$.
5. Else If a wave center node v has lowest degree then $currentNode = v$.
6. Else set *currentNode* to be some node with lowest degree.
7. Visit the adjacent nodes consecutively. For each two nodes,
8. If a pair edge exists place the edge into *removalList*.
9. Else place a triangulation edge between the current pair of neighbors and also into *removalList*.
10. Update the location of *currentNode*'s neighbors in T.
11. Remove *currentNode* and incident edges from G.
12. Increment *counter* by 1.
13. Restore G to its original topology.
14. Remove the edges in *removalList* from G.
15. Perform a DFS (or a longest path heuristic) on G.
16. Place the resulting longest path onto the embedding circle.
17. If there are any nodes which have not been placed then place the remaining nodes into the embedding order with the following priority:
    (i) between two neighbors, (ii) next to one neighbor, (iii) next to zero neighbors.

An example of the execution of the algorithm.

- First, we choose randomly Node 1.
- We check for edge(2,10), which exists. We store it and remove Node 1.

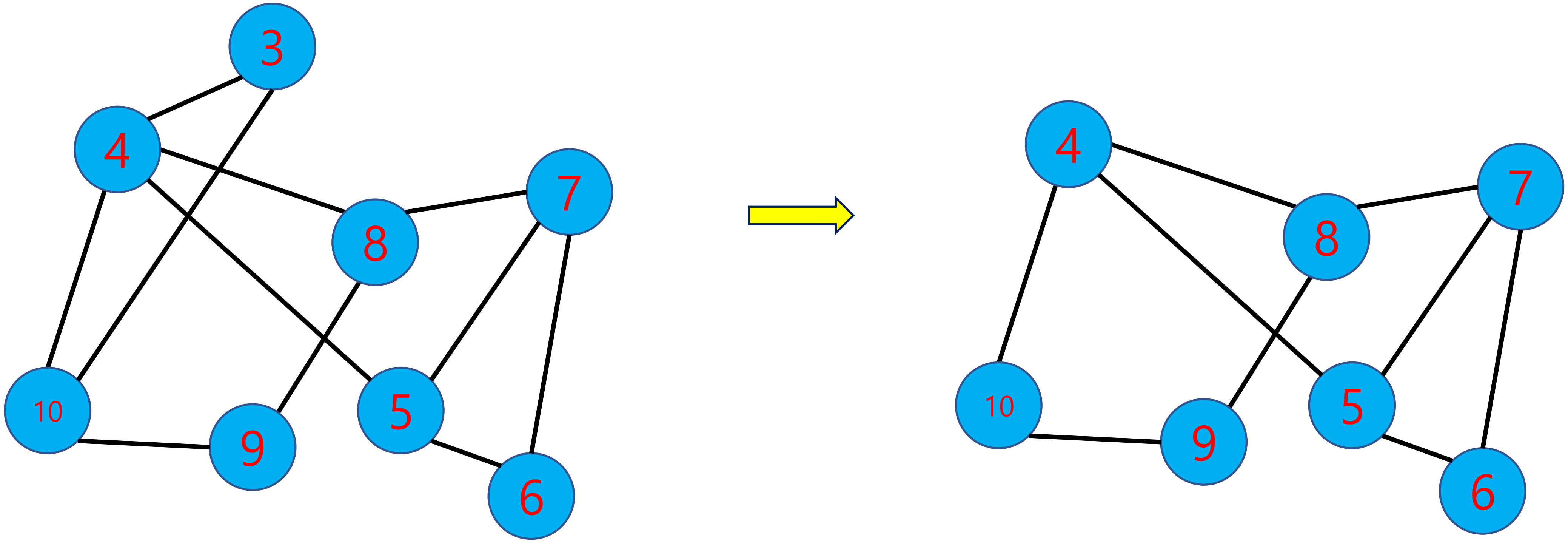


- Next, we choose a lowest-degree neighbor of the removed Node 1, which is 2.
- Check for edge (3,10) which exists. We store it and remove Node 2.



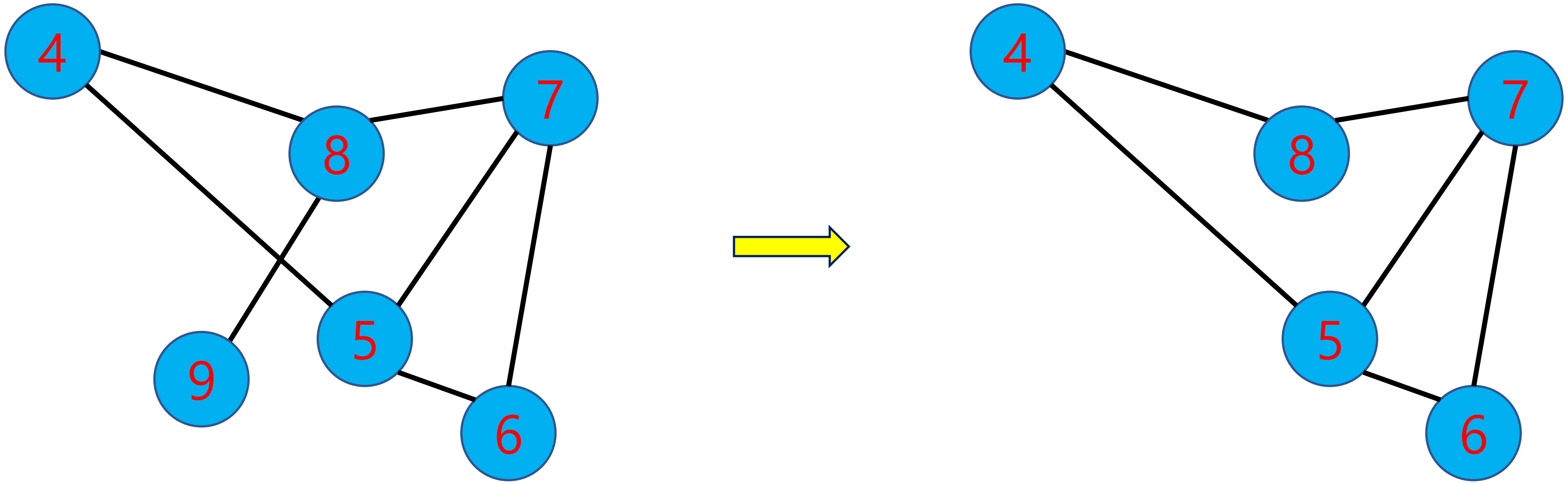
An example of the execution of the algorithm.

- Next we select a lowest degree neighbor of Node 2. This is Node 3. We check for edge (4,10).
- It exists so we store it and remove Node 3.



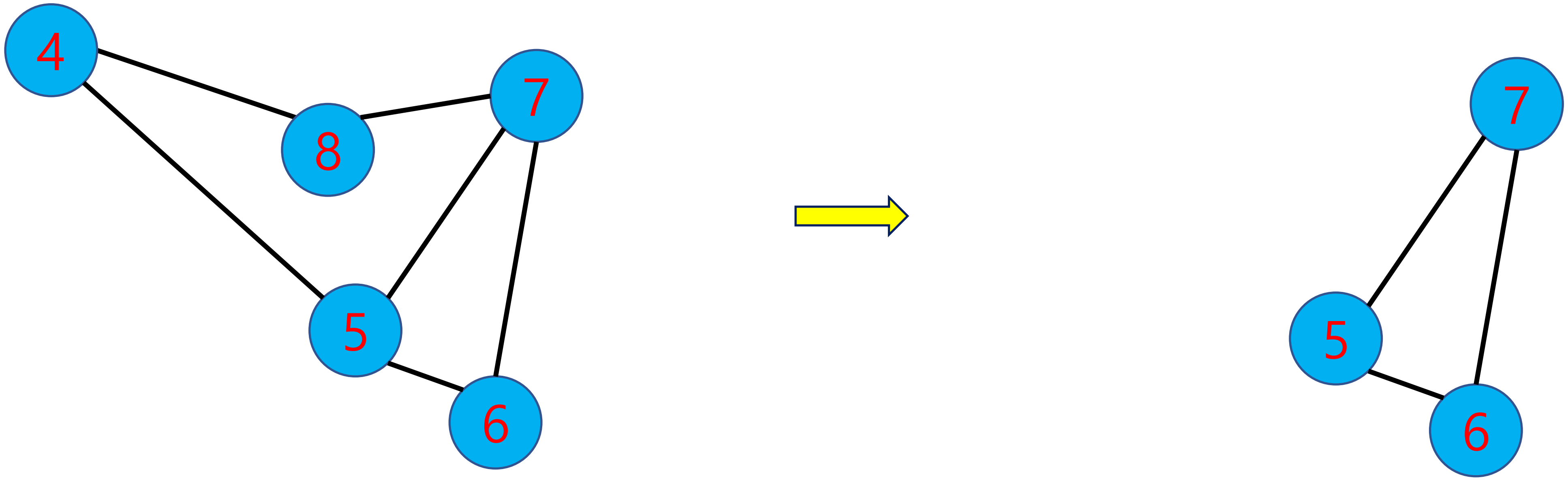
An example of the execution of the algorithm.

- Similarly we can select Node 10 and check for edge (4,9). It does not exist. So we add edge (4,9) which is a triangulation edge, store it and remove Node 10.
- We continue choosing Node 9, and check for edge (4,8). It exists so we store it and remove Node 9. Next, for Node 8 we check for edge(4,7) which does not exist. We add it to the graph and store it. After this, we remove Node 8.



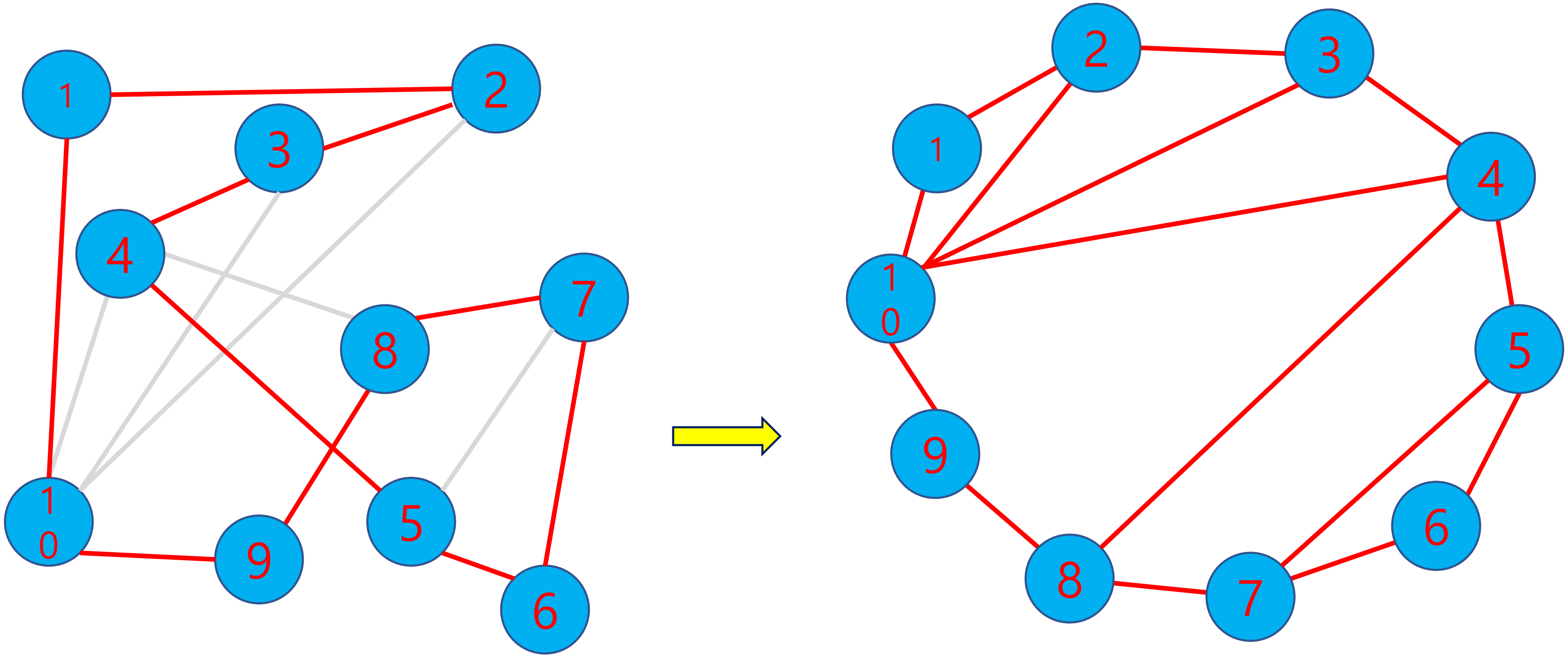
An example of the execution of the algorithm.

- In the same way, we select vertex 4 and check for edge (5,7), which exists. So we mark.
- Now we have only three vertices left, so this phase of the algorithm is completed.



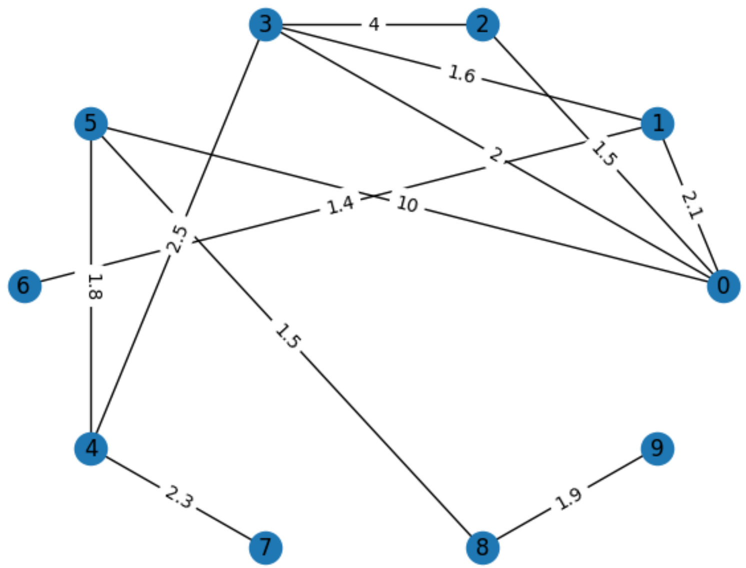
An example of the execution of the algorithm.

- Now we restore the graph and remove all stored edges.
- Since the graph is outerplanar, we have the Hamilton circle left.



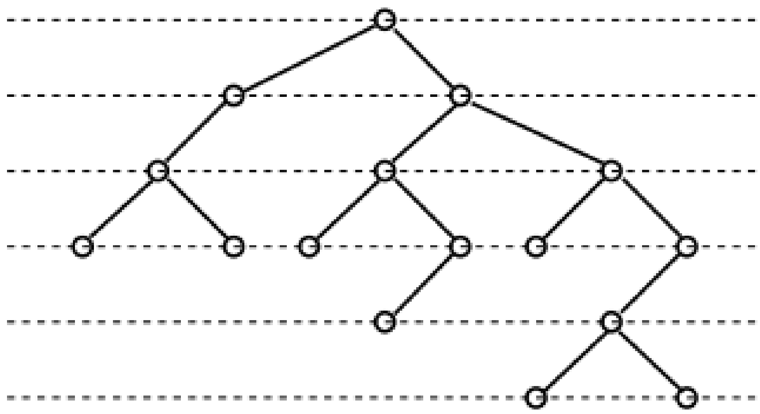
- Circular layout

```
# Instantiate the graph
G = nx.Graph()

edges = [(0, 1, 2.1), (0, 2, 1.5), (0, 3, 2), (0, 5, 10), (1, 3, 1.6), (1, 6, 1.4),
         (2, 3, 4), (3, 4, 2.5), (4, 5, 1.8), (4, 7, 2.3), (5, 8, 1.5), (8, 9, 1.9)]
# add node/edge pairs
G.add_weighted_edges_from(edges)

pos = nx.circular_layout(G)
nx.draw(G, pos, with_labels = True)

labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
```

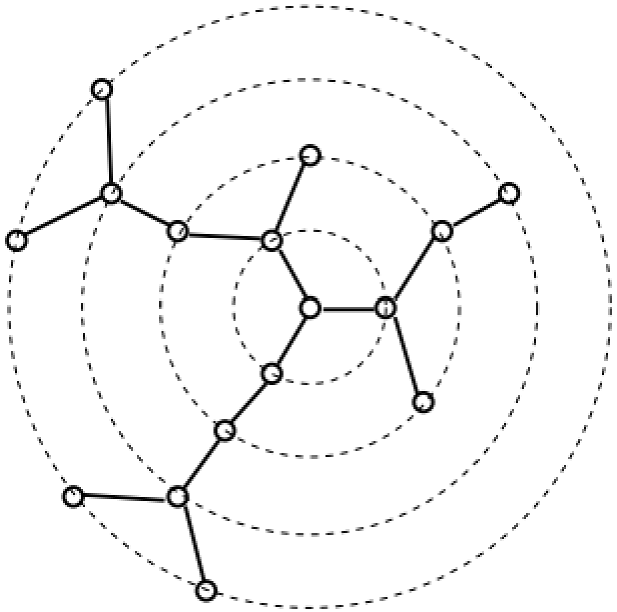


Another graph visualizations

- Trees:
  - Requirements:
    - No two edges cross
    - A child should be placed below its parent in the y-direction.
    - Strongly order-preserving drawings



a layered tree drawing.

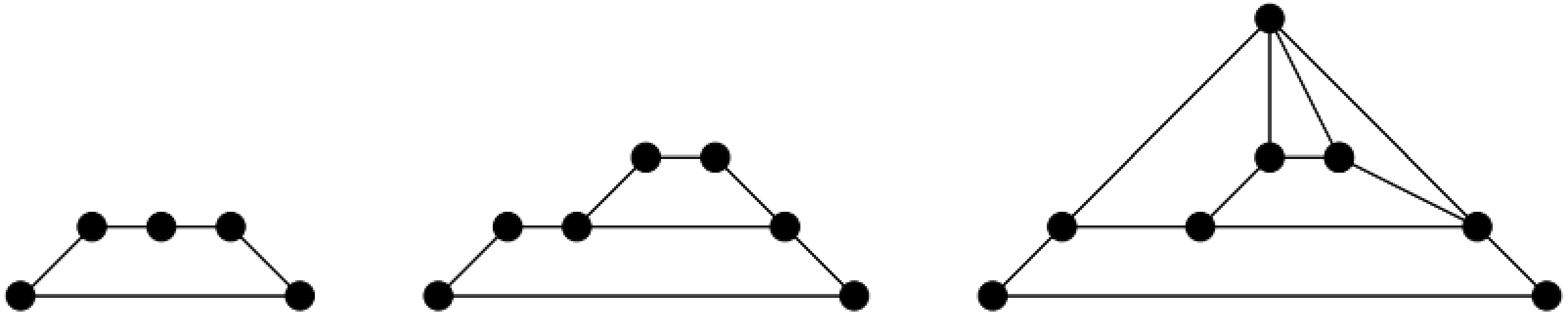


a radial tree drawing.

Another graph visualizations

- Given an input graph G = (V, E):

- Kant used the canonical ordering approach to develop straight-line algorithm.
- The algorithm aims to form a chain and give them the same *y*-coordinate



An example for the straight-line algorithm of Kant.

Chrobak, M., and Kant, G. (1997). Convex grid drawings of 3-connected planar graphs. International Journal of Computational Geometry and Applications, 7(3):211–224

Tools For Data Visualisation

- There are sevreal open source tools for network analysis:
  - NetworkX
  - iGraph packages in R
  - Gephi
  - Cytoscape

Tools For Data Visualisation: Gephi

- An open-source visualization exploration software without having coding skills.
- Functions:
  - Interaction with the final representation
  - Manipulation with structures
  - Appearance properties
  - Understand patterns in visualization.

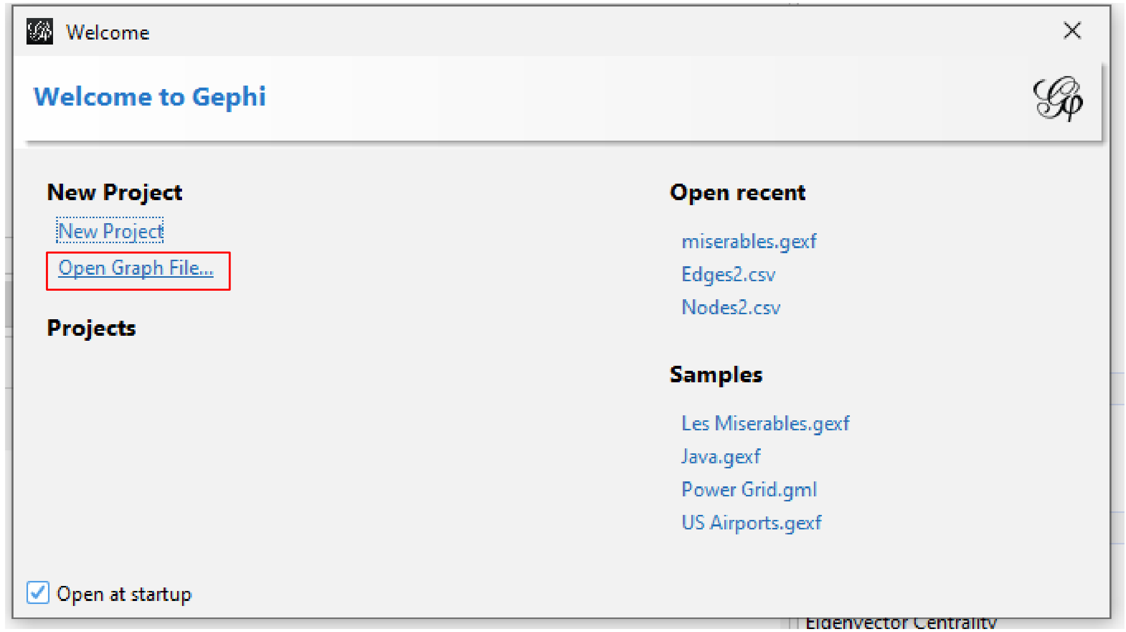
https://github.com/gephi/gephi

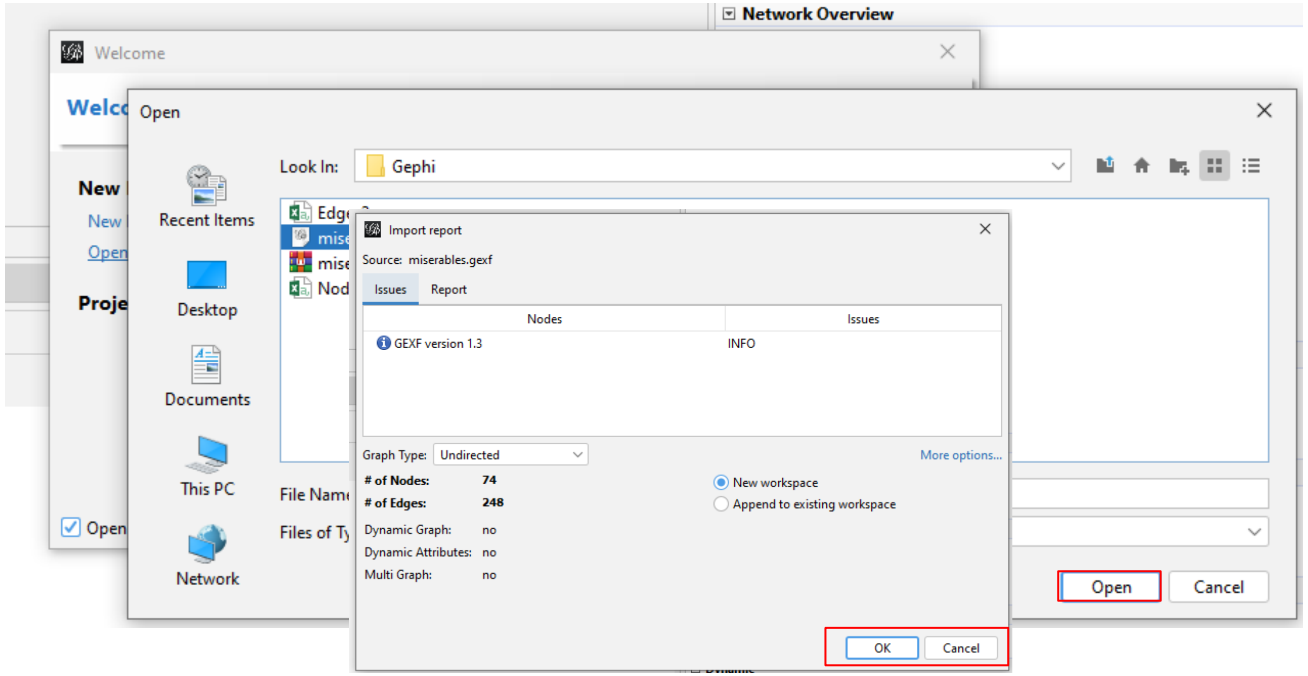
Tools For Data Visualisation: Gephi

- Applications of Gephi:
  - Exploratory Data Analysis
  - Link Analysis
  - Social Network Analysis
  - Biological Network Analysis
  - Poster Creation
- Different layouts:
  - Circular Layout
  - Noverlap Layout
  - Expansion…

Visualizing a graph using Gephi

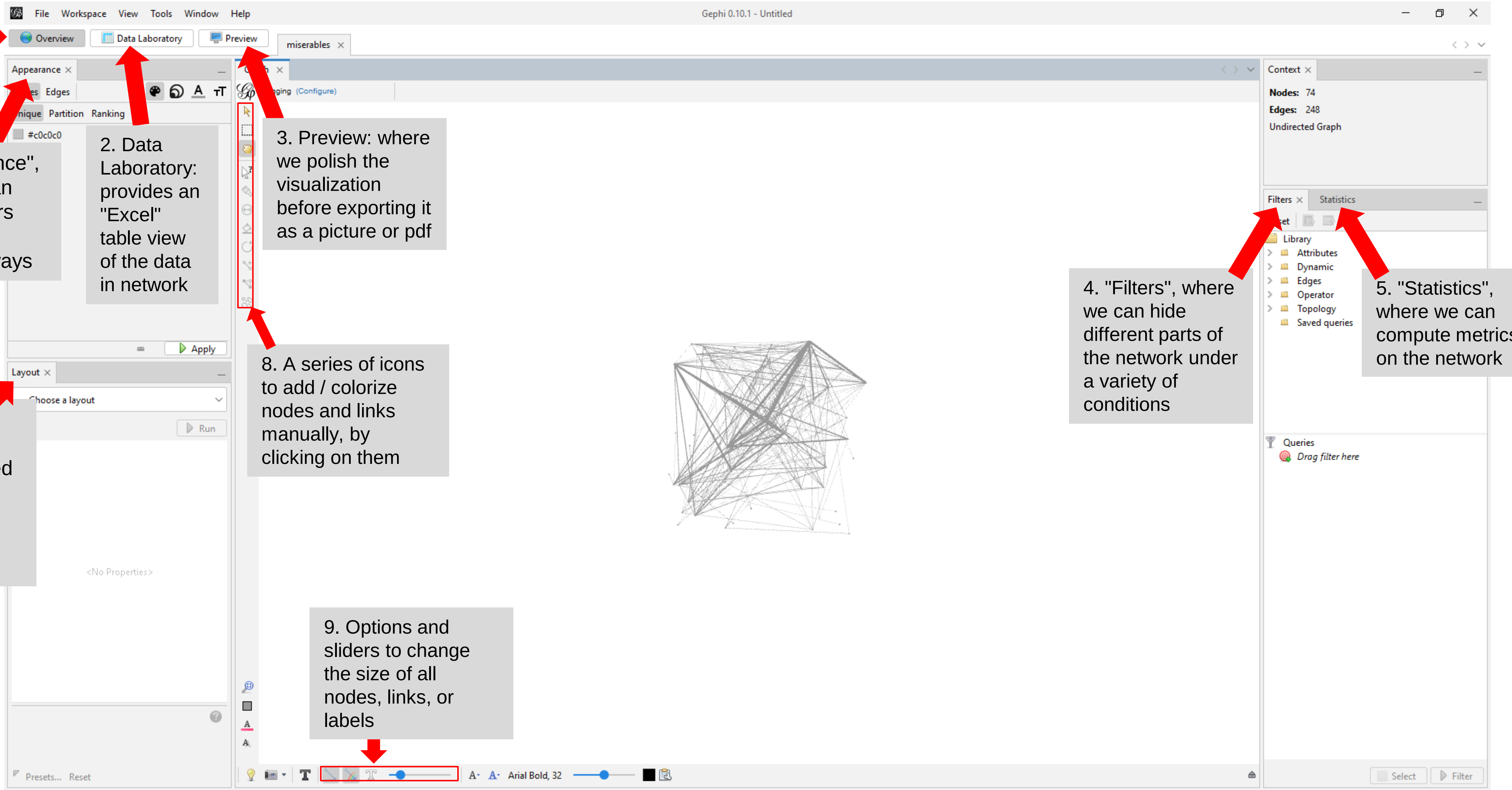
- Prepare:
- Sample graph: miserables.gexf (download in class github)
- Open Gephi.
- On the Welcome screen that appears, click on Open Graph File.
- Open miserables.gexf and click OK





Visualizing a graph using Gephi

➢ Gephi's interface:



1. Overview: where we can explore the graph visually

2. Data Laboratory: provides an "Excel" table view of the data in network

3. Preview: where we polish the visualization before exporting it as a picture or pdf

4. "Filters", where we can hide different parts of the network under a variety of conditions

5. "Statistics", where we can compute metrics on the network

6. "Appearance", where we can change colors and sizes in interesting ways

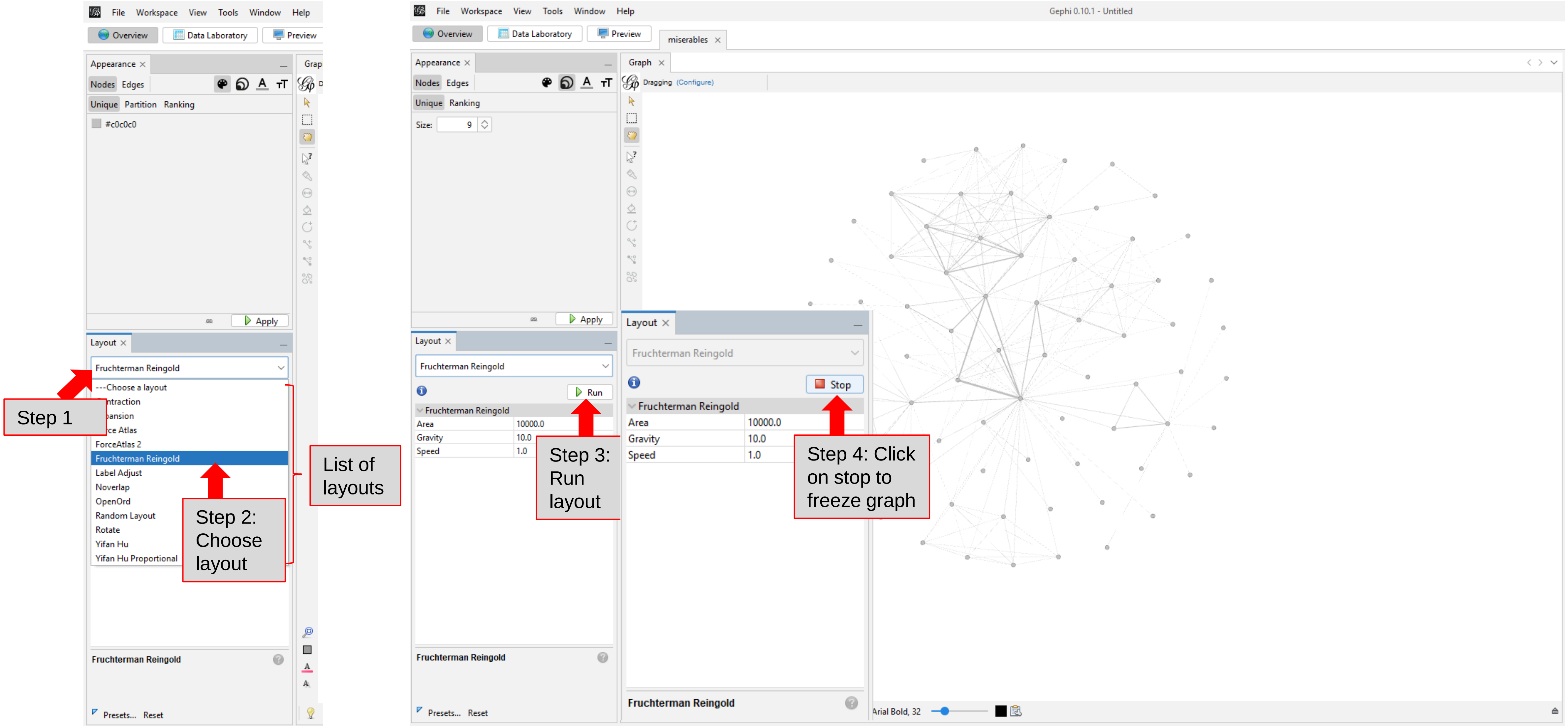
7. "Layouts", where we can apply automated procedures to change the position of the network

8. A series of icons to add / colorize nodes and links manually, by clicking on them

9. Options and sliders to change the size of all nodes, links, or labels

Visualizing a graph using Gephi

- Gephi's layout:



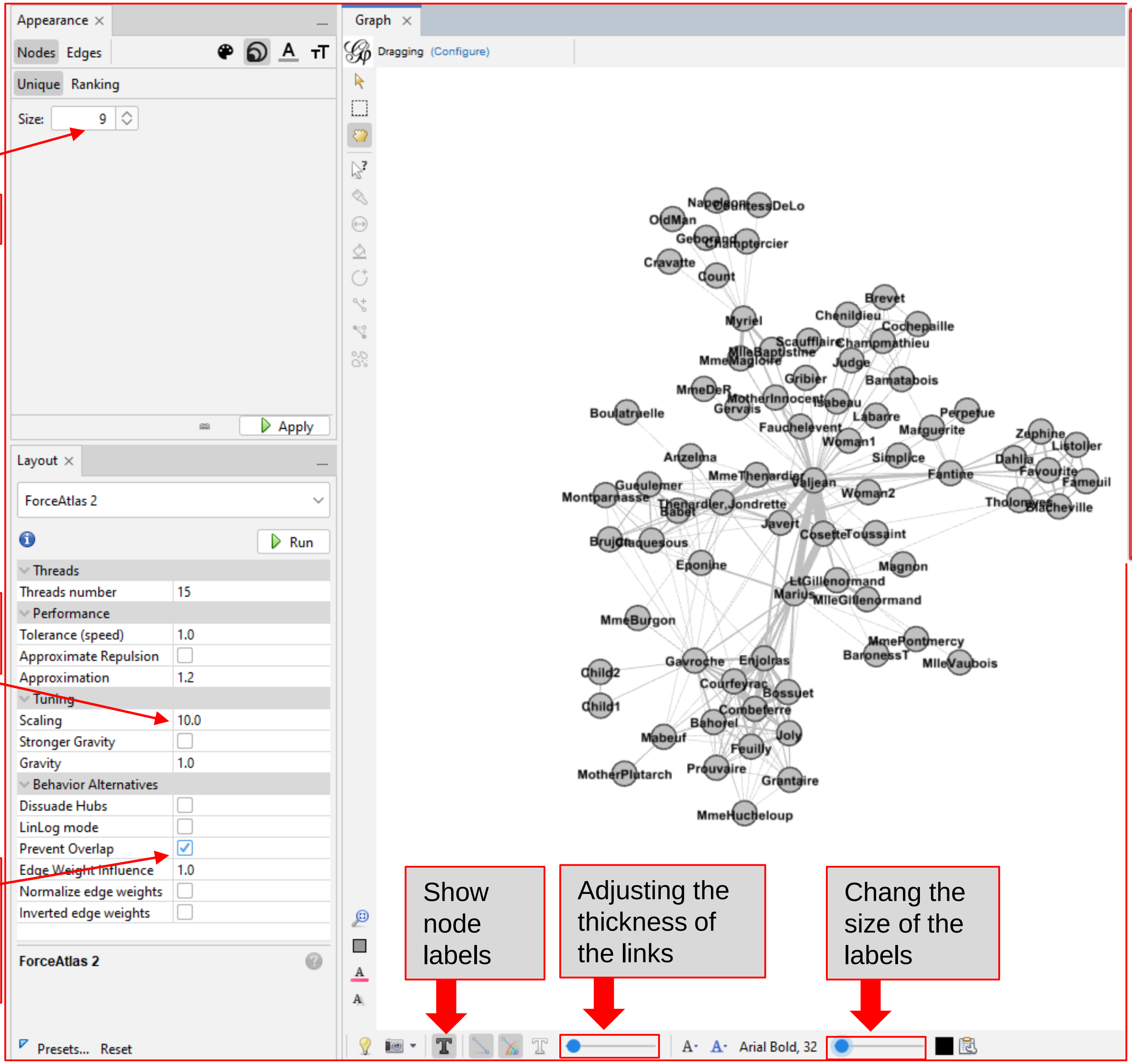
Visualizing a graph using Gephi

- Force atlas 2 layout:

1. Size: change node size.

2. Scaling: a parameter to control how "wide" the graph will be.

3. Prevent overlap: a parameter to avoid that nodes are on top of each other. To make it easier to read the network. Check the box.



Force Atlas 2 is a layout which:

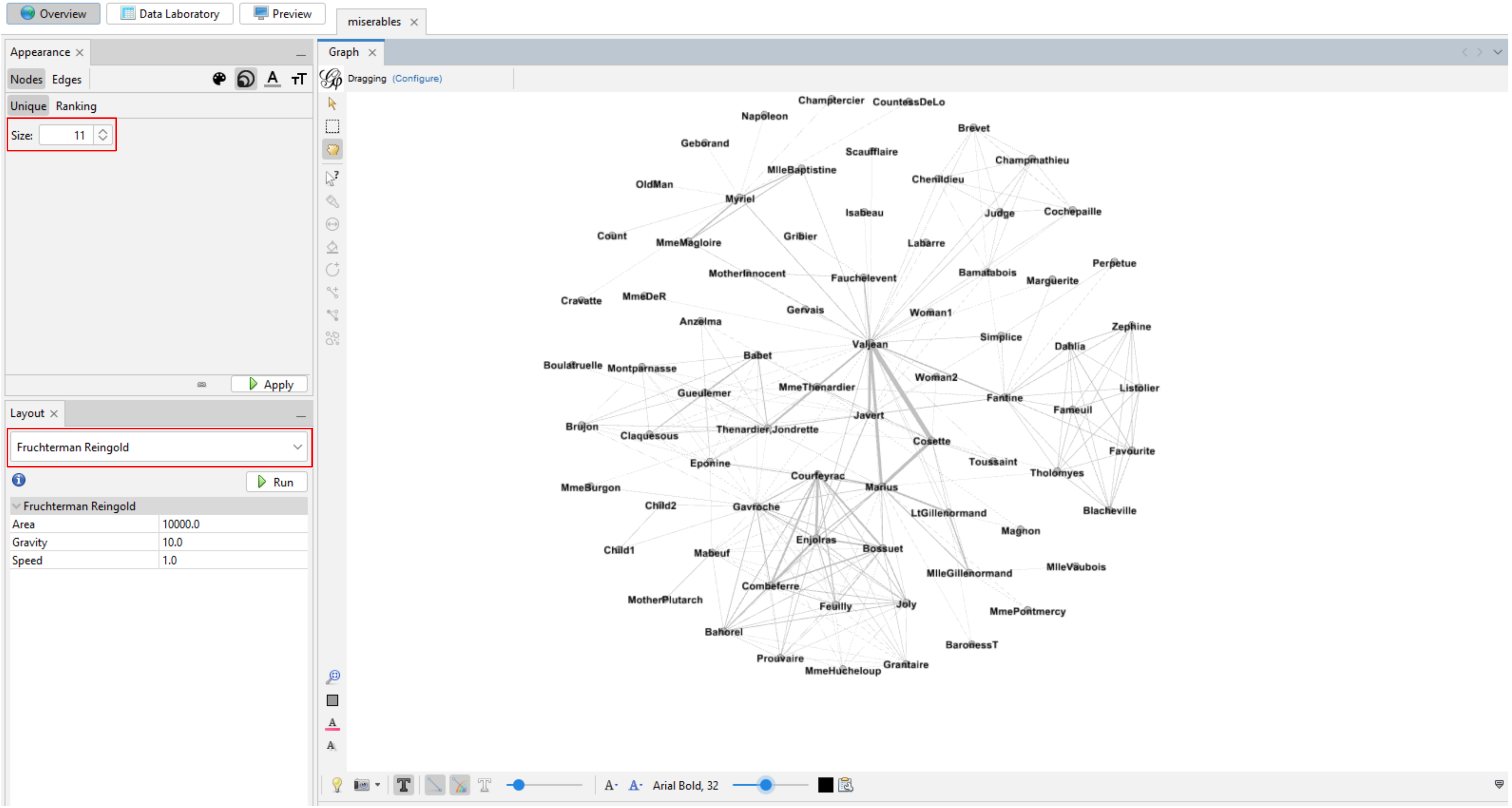
1. brings together nodes which are connected

2. spreads apart unconnected nodes

As an effect, we can easily detect communities of nodes

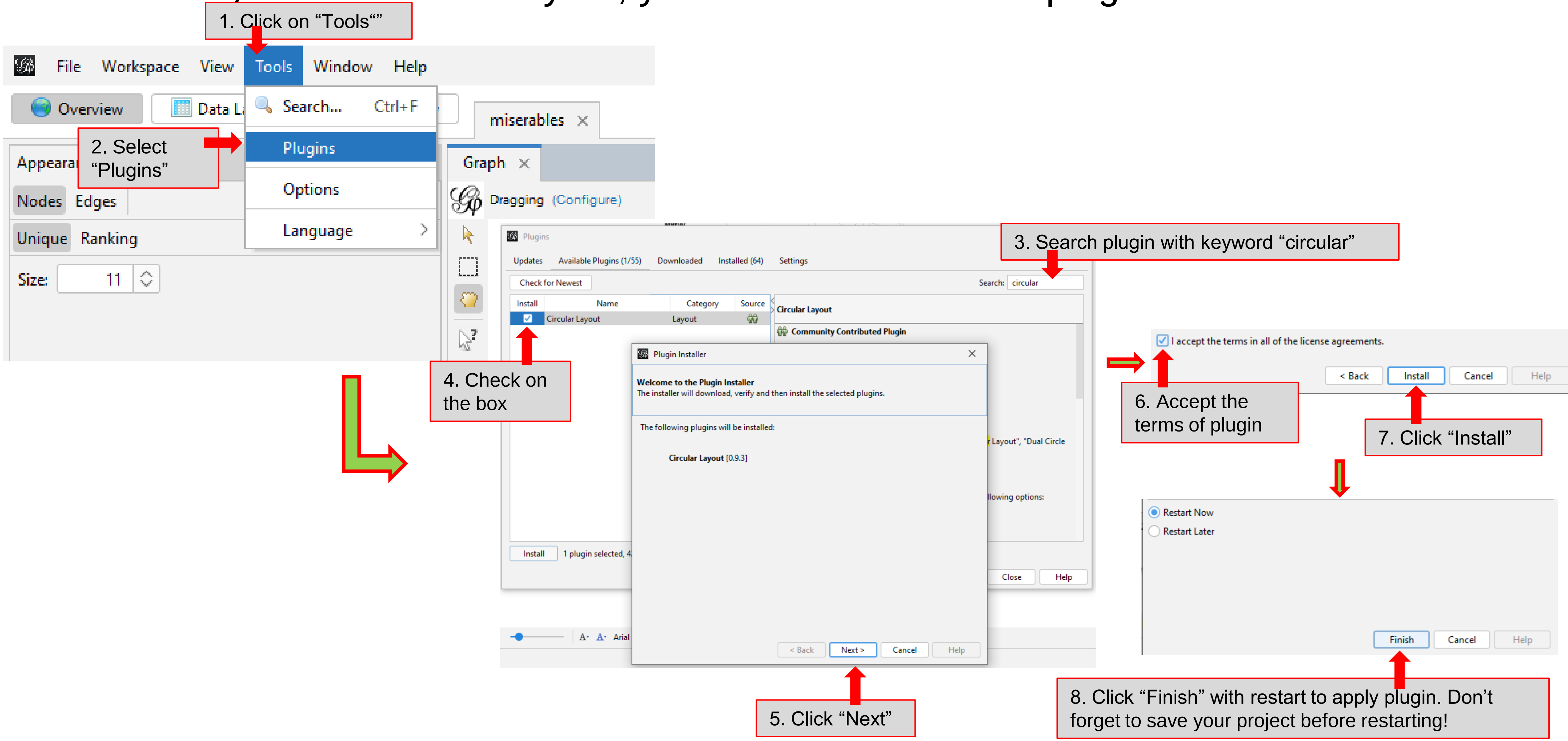
Visualizing a graph using Gephi

- Fruchterman and Reingold relies on spring layout



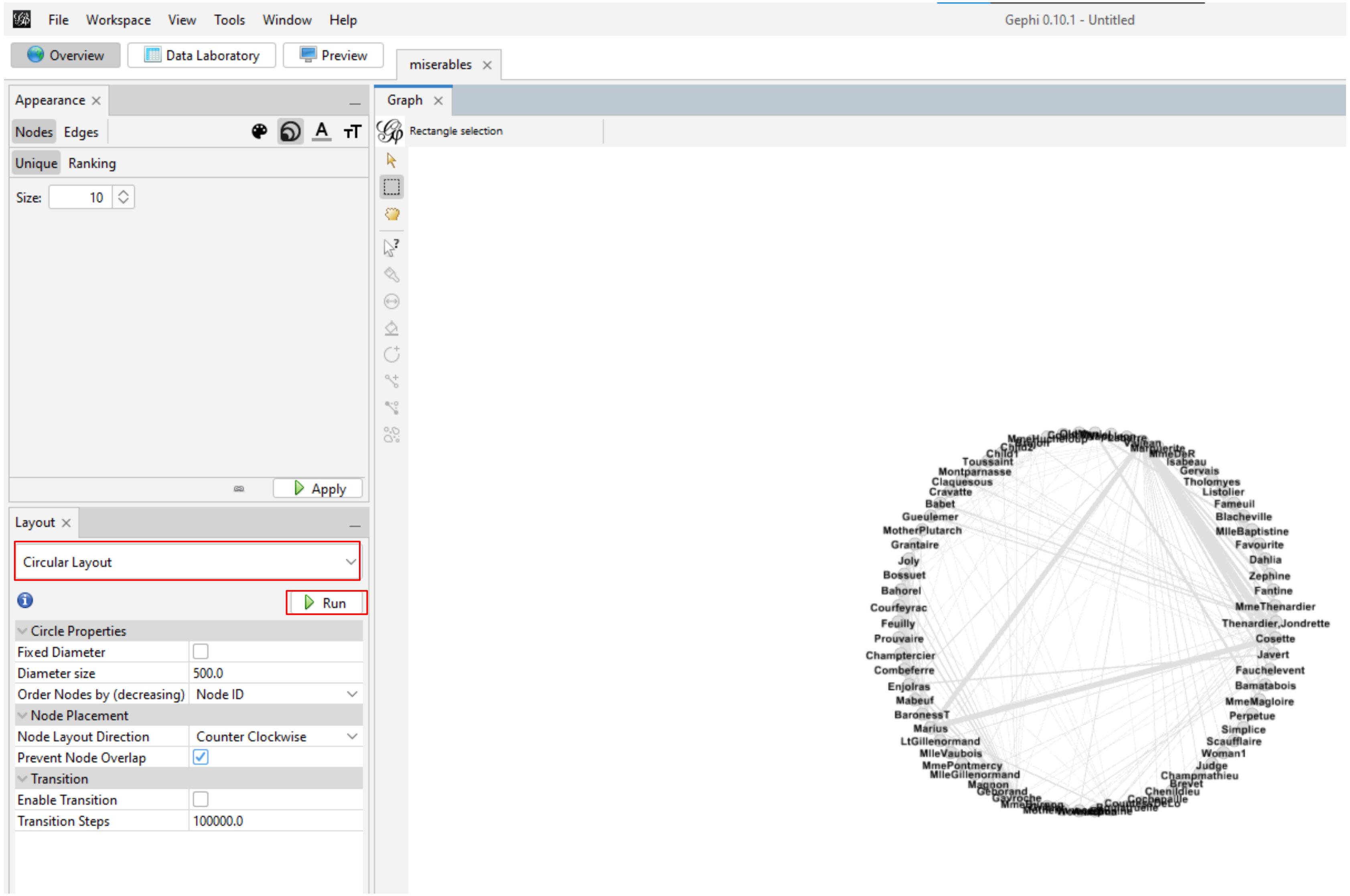
Visualizing a graph using Gephi

- ➢ Circular layout: to use this layout, you have to install new plugin



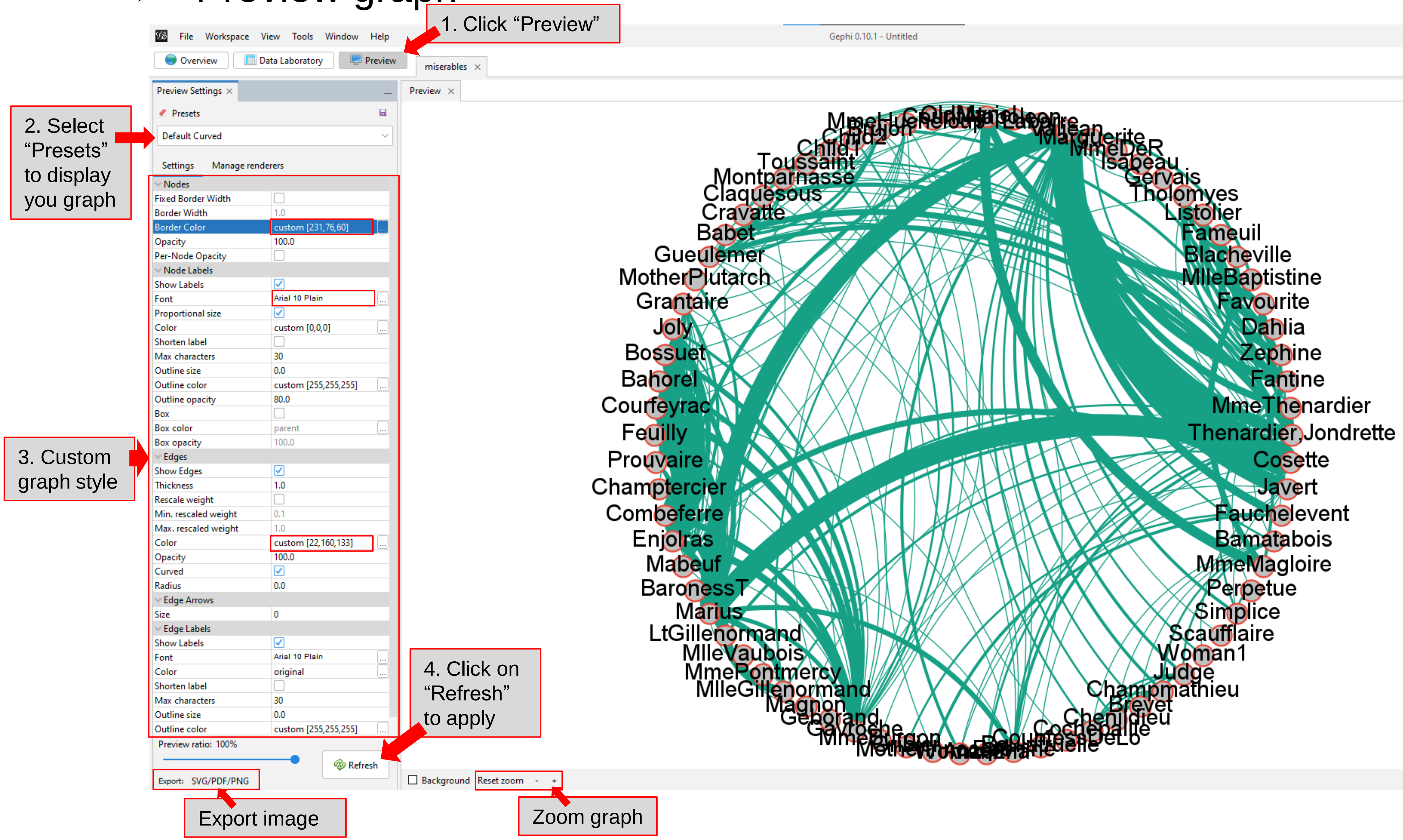
Visualizing a graph using Gephi

➢ Circular layout:



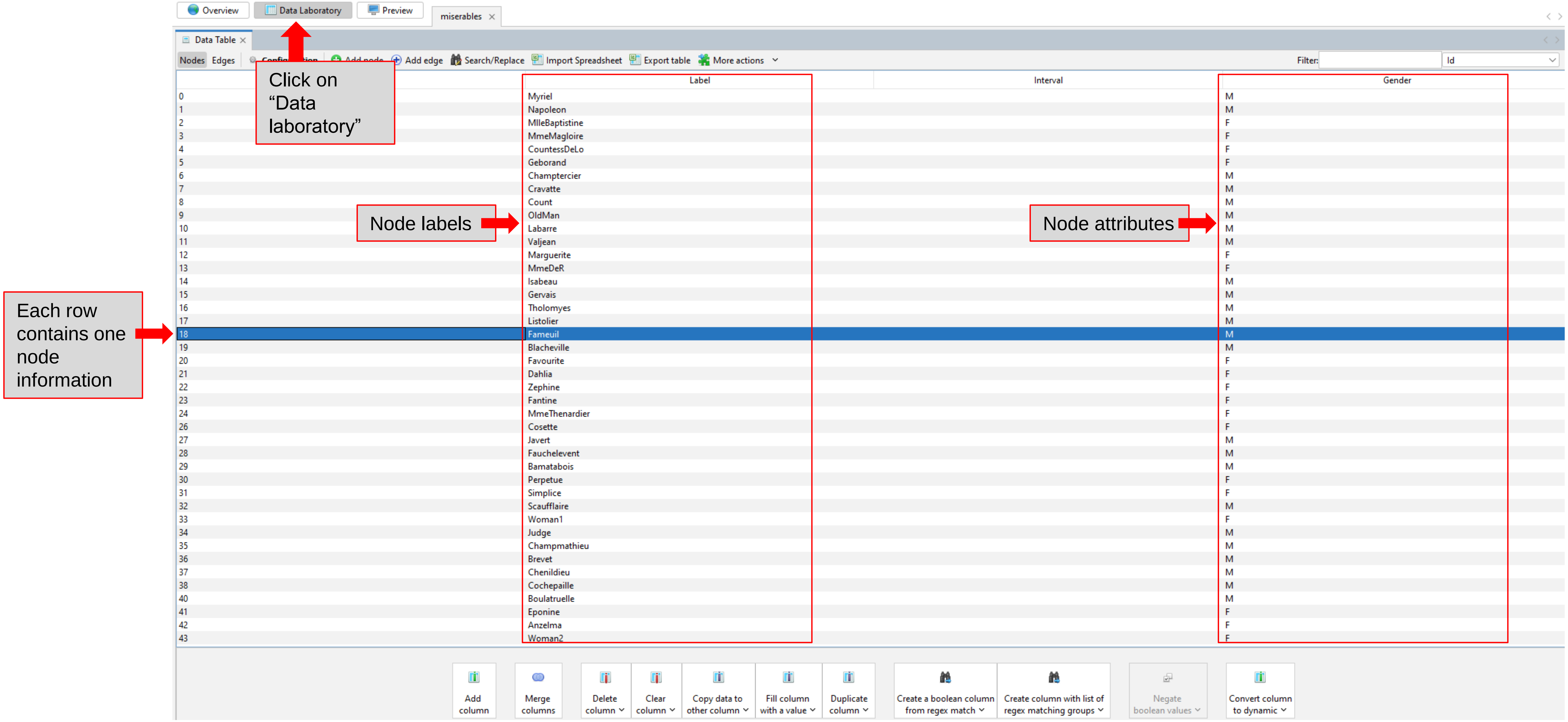
Visualizing a graph using Gephi

➢ Preview graph



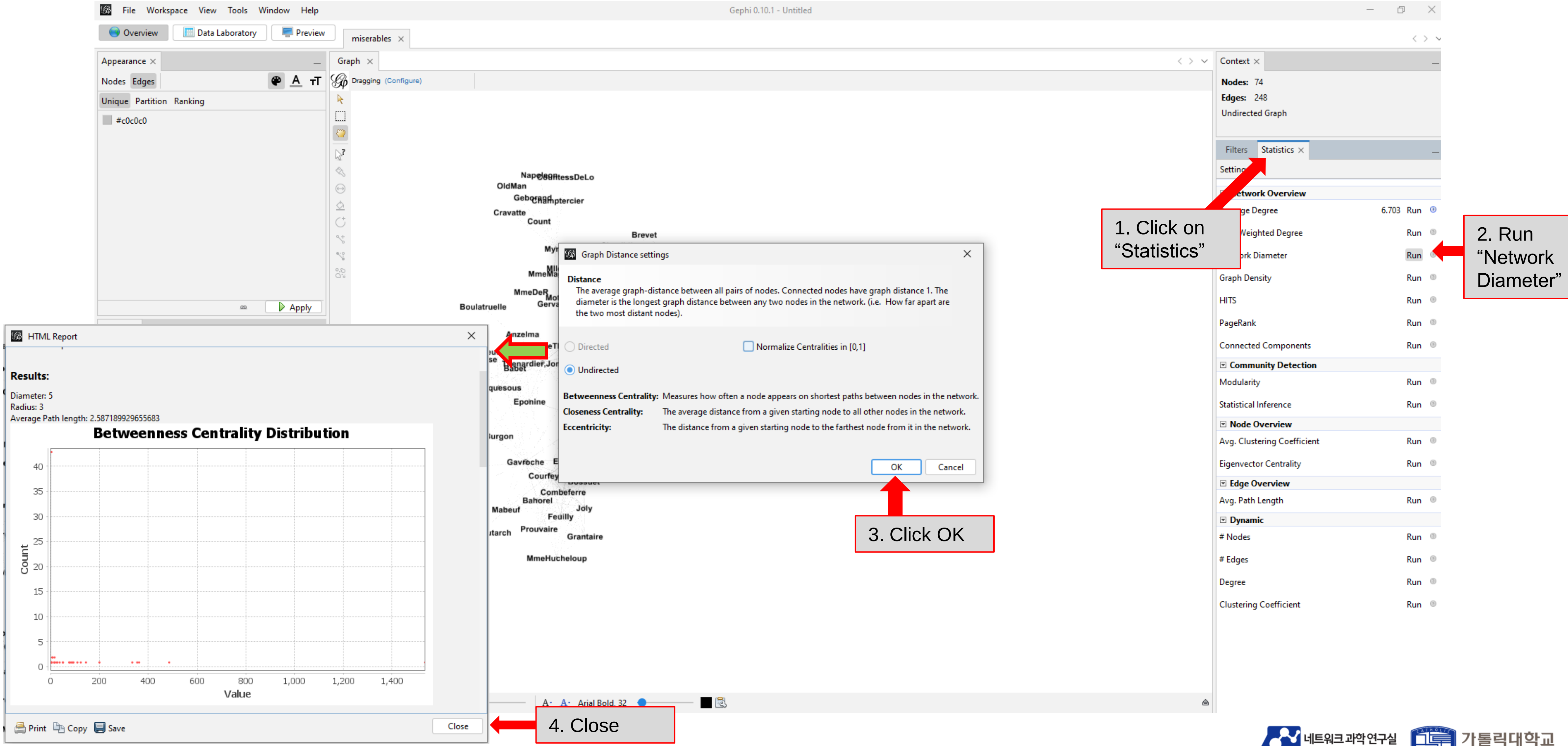
Visualizing a graph using Gephi

- Switching the view to the data laboratory:



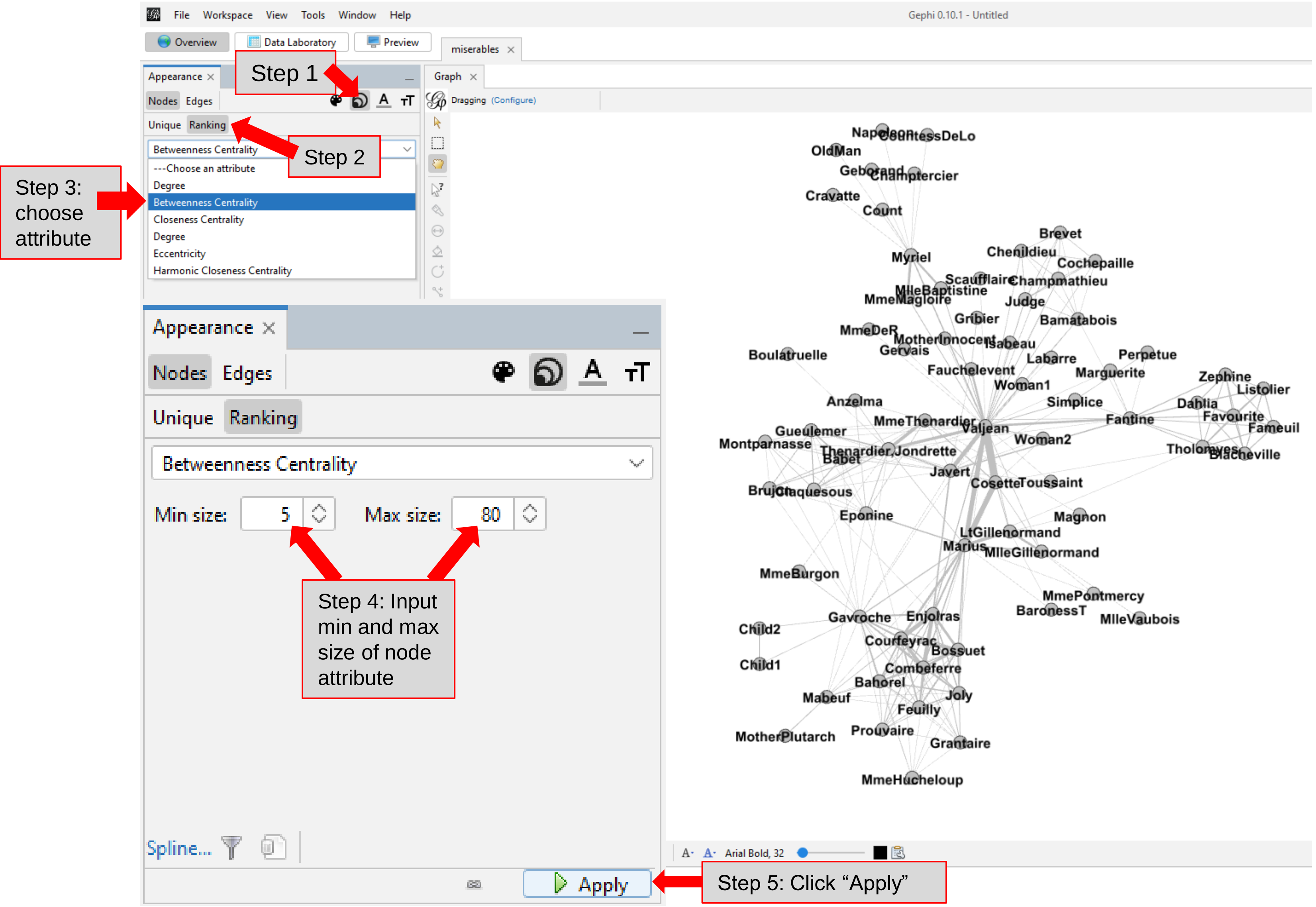
Visualizing a graph using Gephi

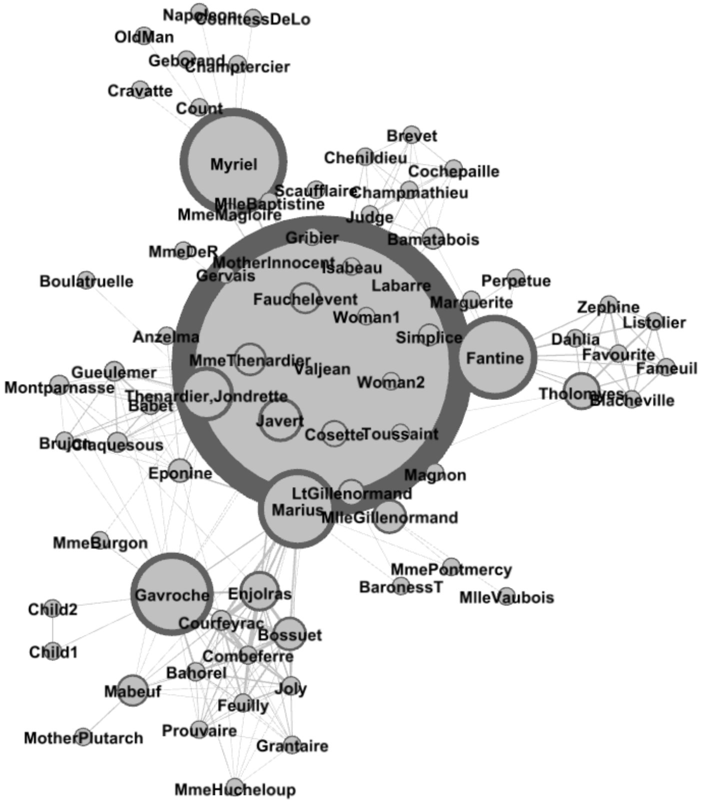
- Computing betweenness centrality with Gephi:



Visualizing a graph using Gephi

➢ View graph attribute: Betweenness Centrality





네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA