# Learning Module – Clustering and Similarity

Clustering is all about identifying communities within larger networks. In this module, we will cover:

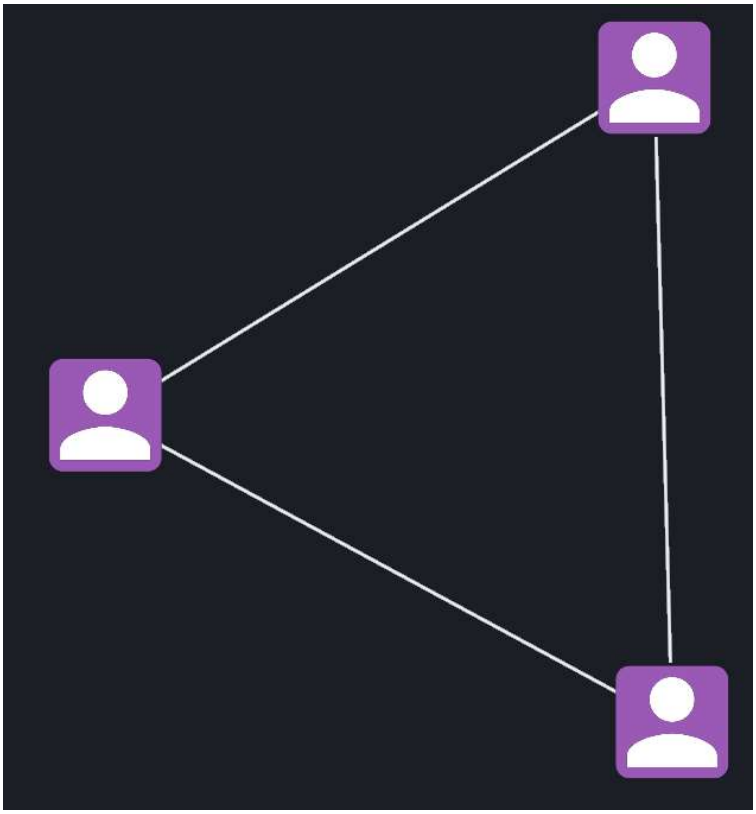
- Ego-Networks
- Half-Hop and One-Hop Induced Subgraphs
- K-Trusses
- Hierarchical Clustering
- Similarity
- Jaccard Index
- Levenshtein Distance

---

## Ego-Networks

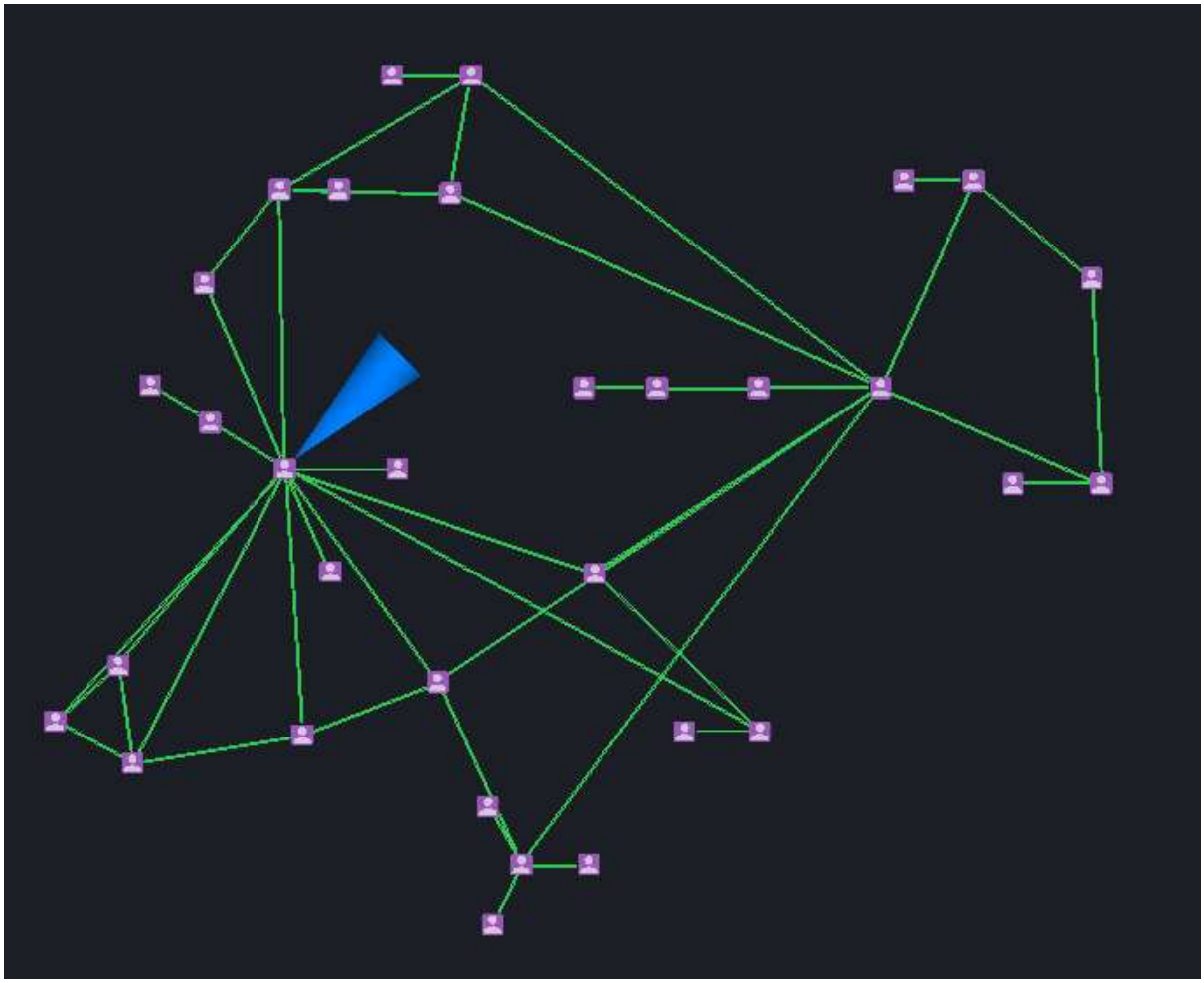
Earlier, we looked at identifying important relationships between entities using metrics such as Multiplexity, Ratio of Reciprocity, and Weight. We can actually go further and use other network structures to determine the importance of a relationship.

Triangles, for example, are key structures in social networks. Studies have shown that one-sided relationships are weak – they do not survive the test of time and with only one person putting in the effort, it eventually stops being worth the trouble. Two sided relationships tend to be stronger but can collapse if subjected to or pressured by external circumstances or conflict. However, when there is a third-party that both entities are connected to, bonds are less likely to break when conflicts as arise as the group keeps people together.

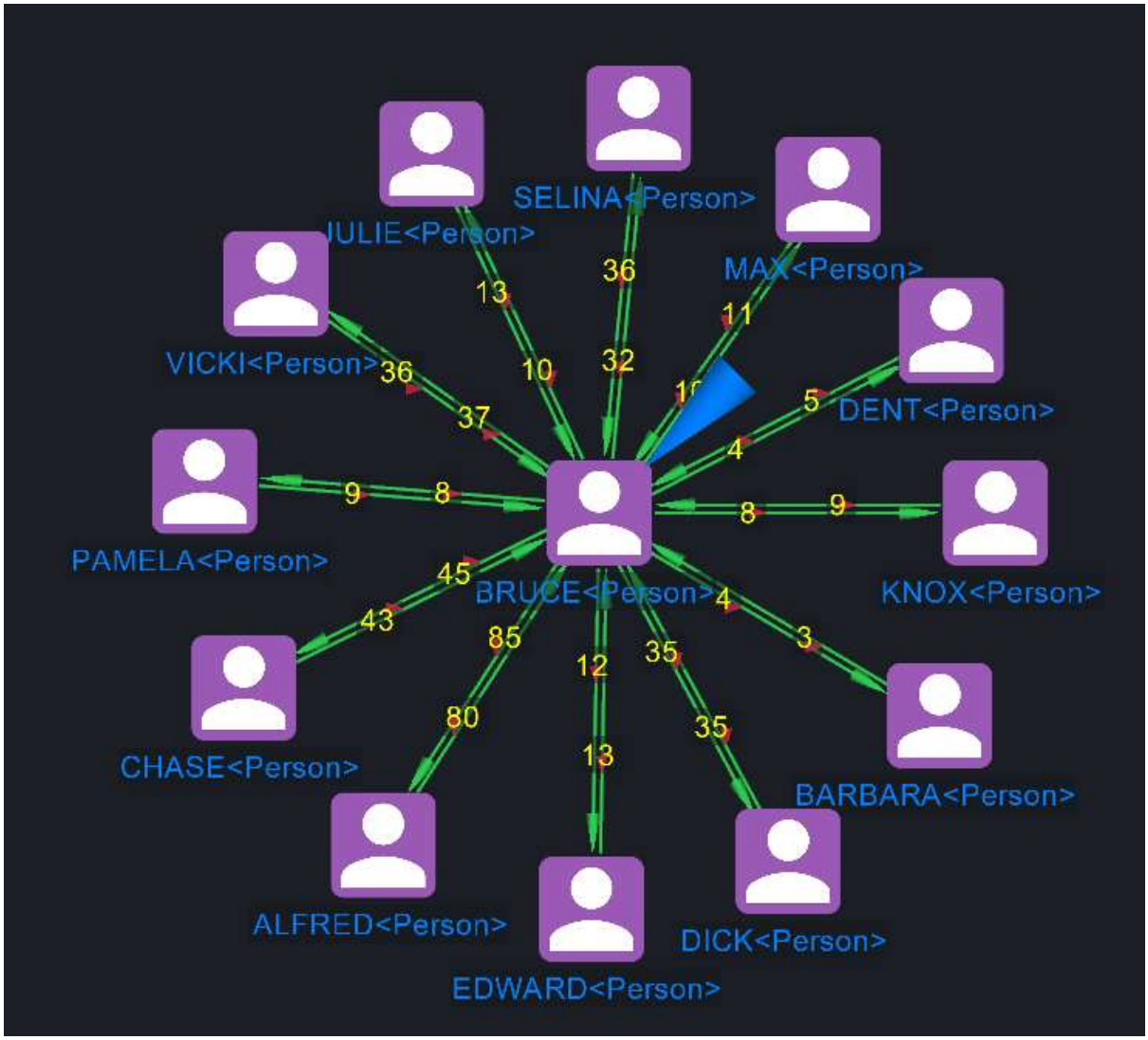


An ego-network identifies triangles in a node's local neighbourhood (1-hop network). It helps indicate closer ties, and also communities.

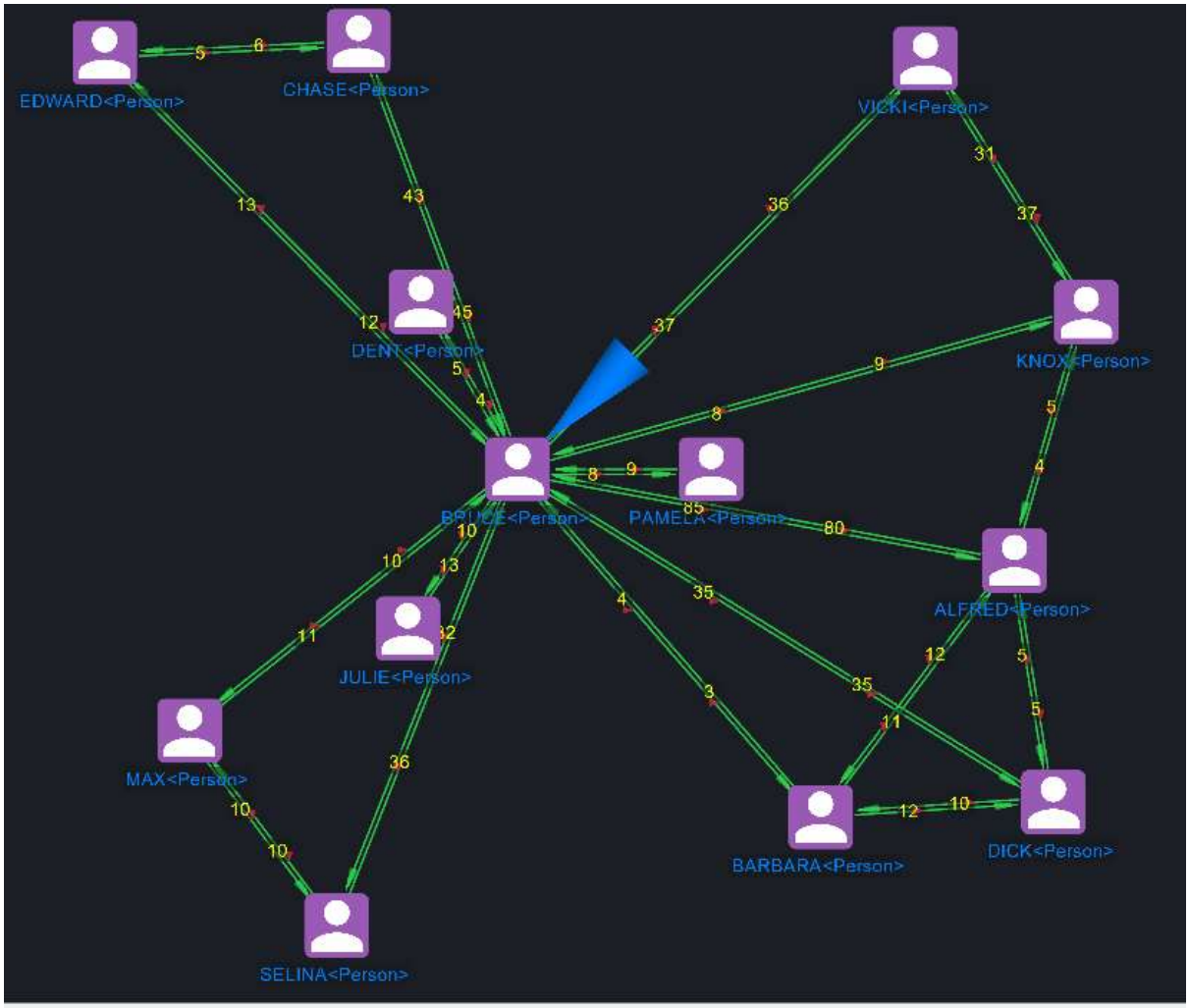
Here is a sample network.



If we have a look at the blazed node’s 1-hop network, we can use weight, reciprocity and multiplexity to determine the closest neighbour.




But if we expand the network 1.5 hops, we can see the triangles in the blazed node's neighbourhood.



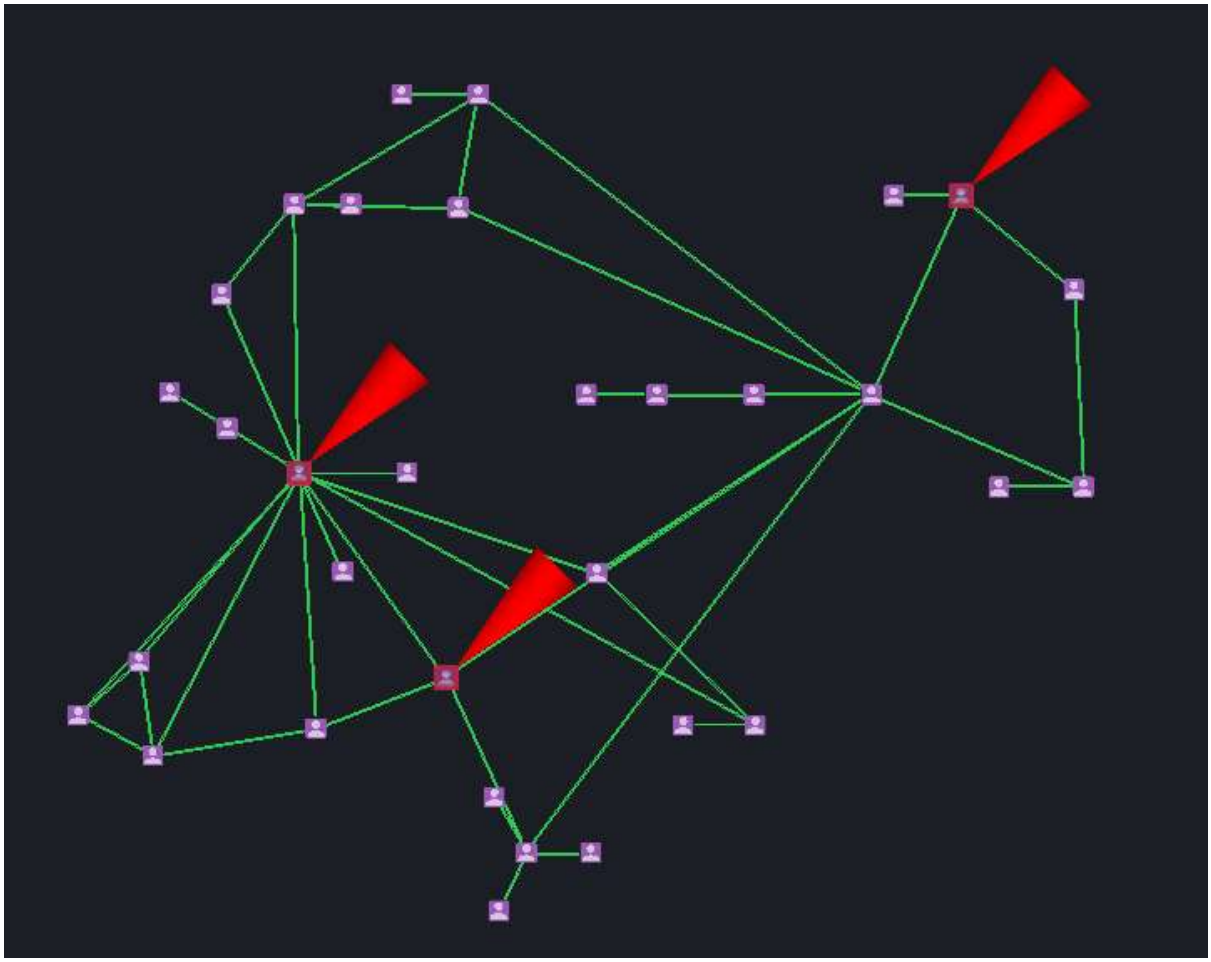

This indicates 3 potential communities that the blazed node is connected to.

To create an Ego-Network in Constellation, expand the selection by 1 hop and then select the Half-Hop Induced Subgraph.

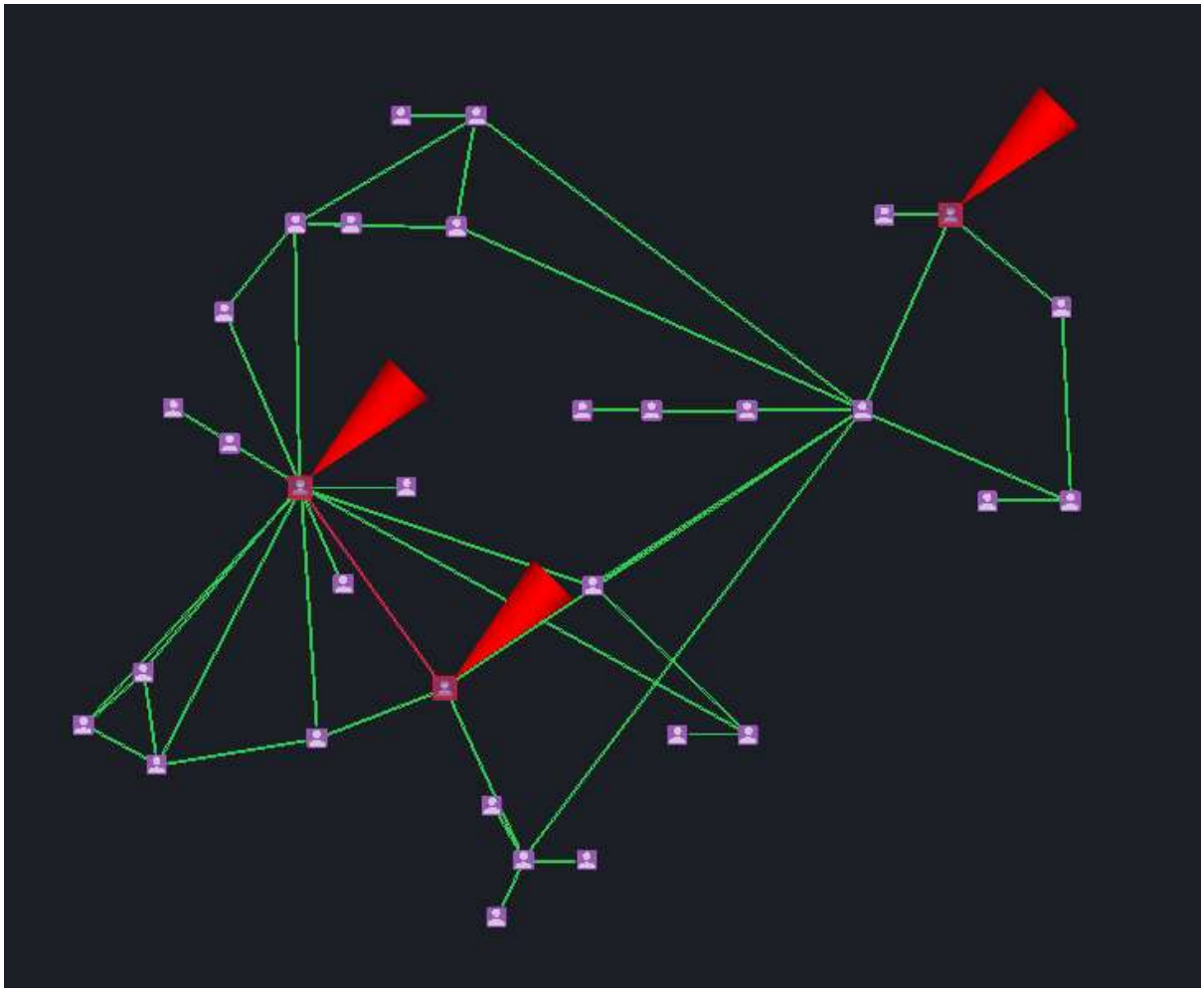
## Half-Hop and One-Hop Induced Subgraphs

A subgraph is a graph within a graph. Any selection made in Constellation is an example of creating a subgraph.

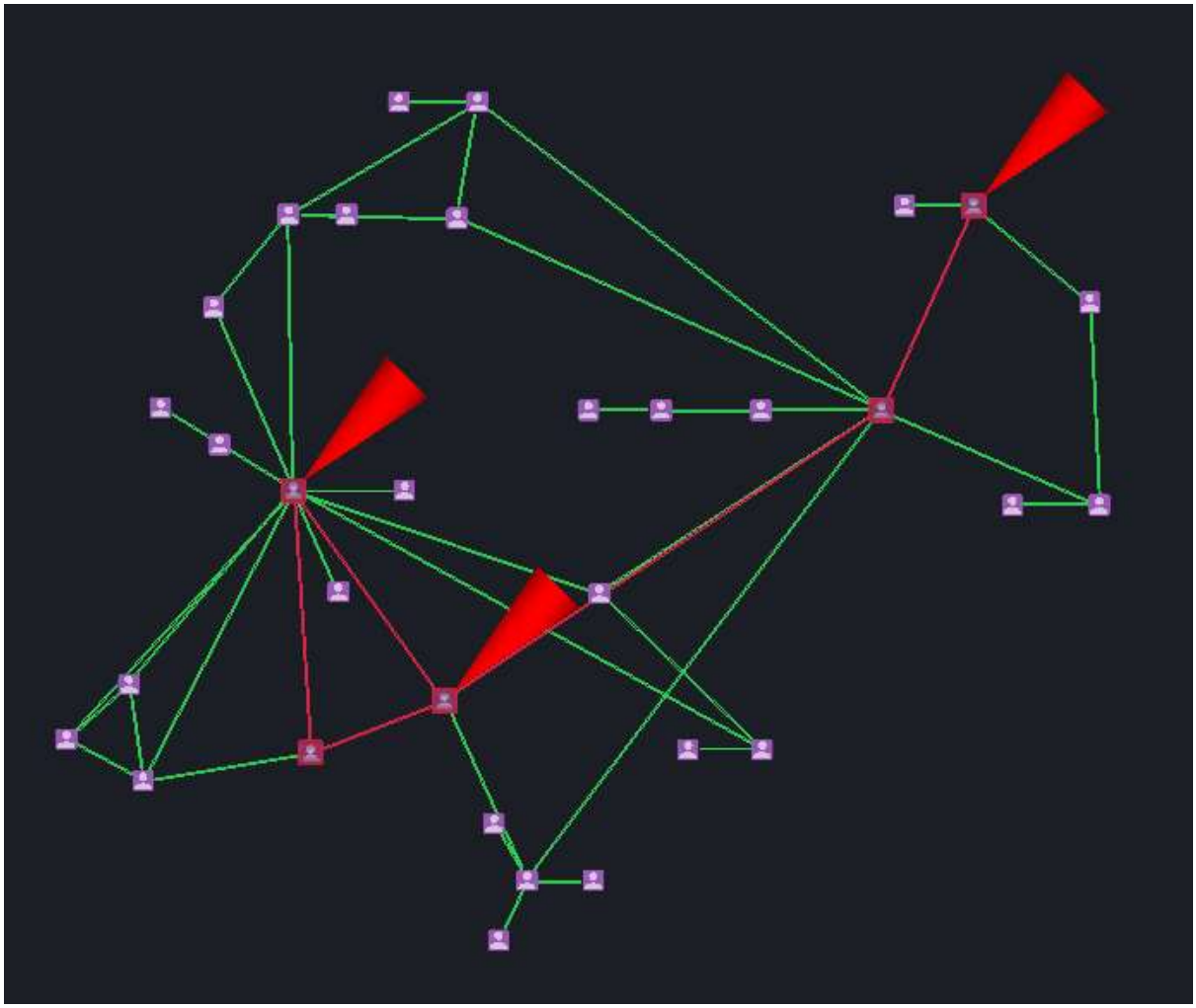
An induced subgraph is a subgraph that is created by meeting certain conditions. Firstly, some nodes of interest need to be selected.



A half-hop induced subgraph expands the selection by half a hop, ONLY between the selected nodes. It tells us whether the nodes are connected/one hop away.

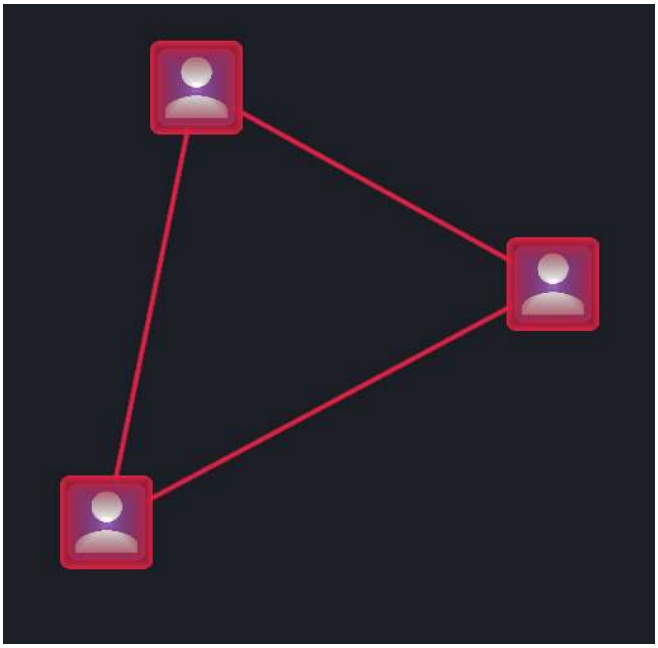


A one-hop induced subgraph expands the selection by one hop, ONLY between common neighbours of the selected nodes. It tells us whether the nodes have neighbours in common/are two hops away.

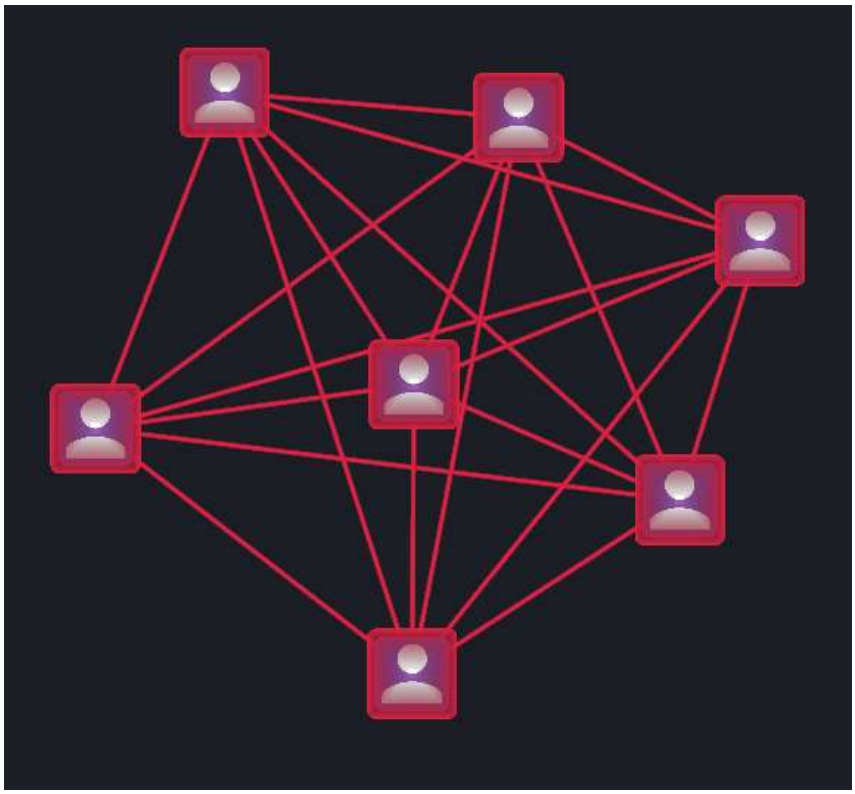


## K-Trusses

K-Trusses help us identify groups of interconnected triangles in a graph. A k-Truss is a graph structure that has the property that each edge is found in k-2 triangles.

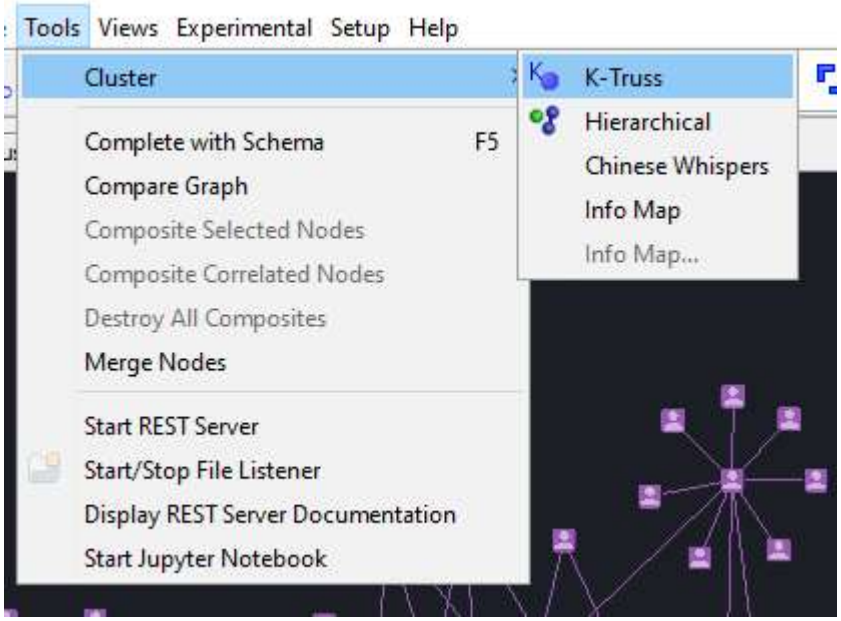


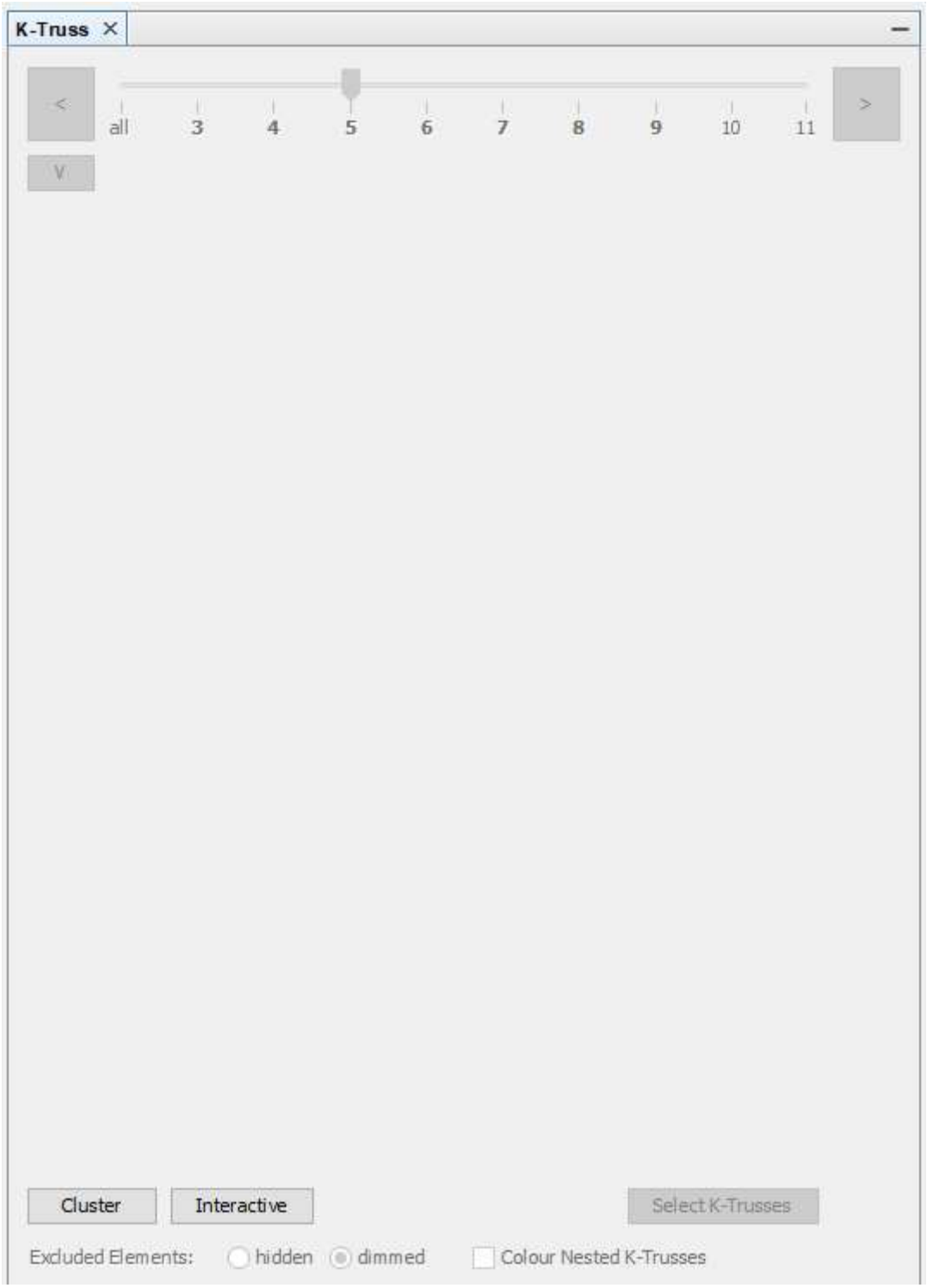
Above is an example of a 3-truss. Each edge is found in 1 triangle. Below is an example of a 5 truss – each edge can be found in 3 triangles.



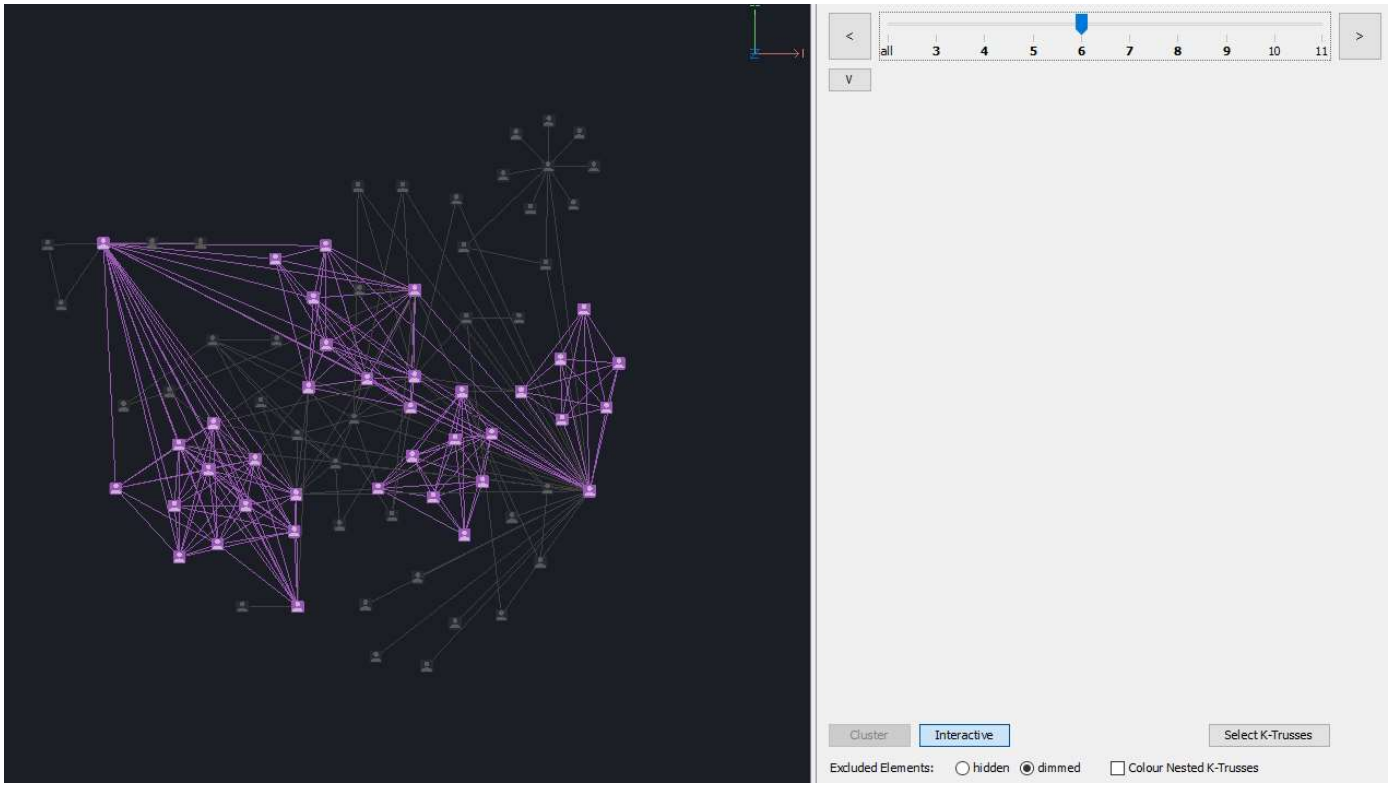
If you consider the backbone of the graph as being the first step to centring towards an interconnected network core, k-Trusses take it to the next level, helping to identify even more densely connected groups within a network.

The k-Truss view can be found under Tools menu within the Clustering submenu.

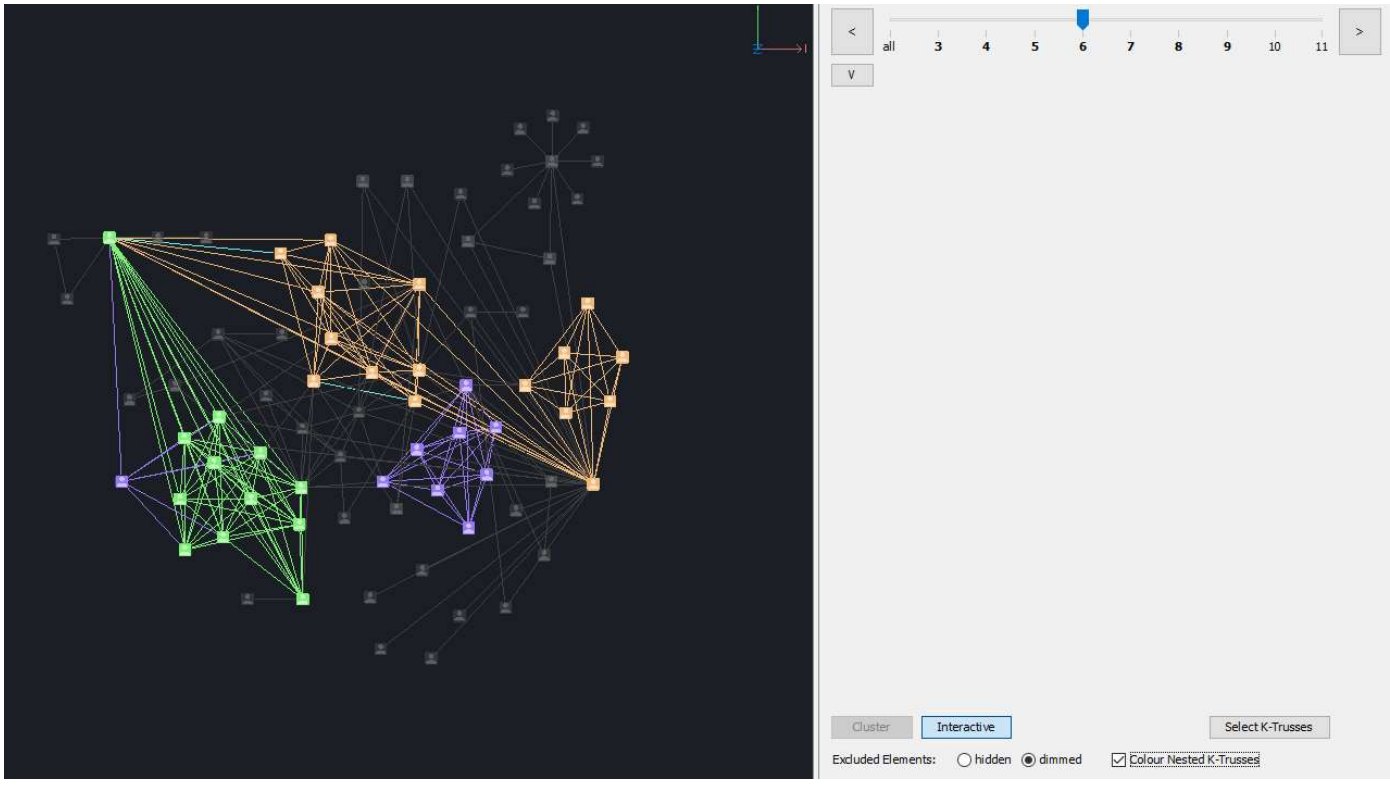

Tools
Views
Experimental
Setup
Help
Cluster
Complete with Schema
F5
Compare Graph
Composite Selected Nodes
Composite Correlated Nodes
Destroy All Composites
Merge Nodes
Start REST Server
Start/Stop File Listener
Display REST Server Documentation
Start Jupyter Notebook
K-Truss
Hierarchical
Chinese Whispers
Info Map
Info Map...




Press **Cluster** to calculate the k-Trusses, and then press **Interactive** to move the slider to highlight them in the graph.




Toggle Colour Nested K-Trusses to highlight more densely connected groups visually.



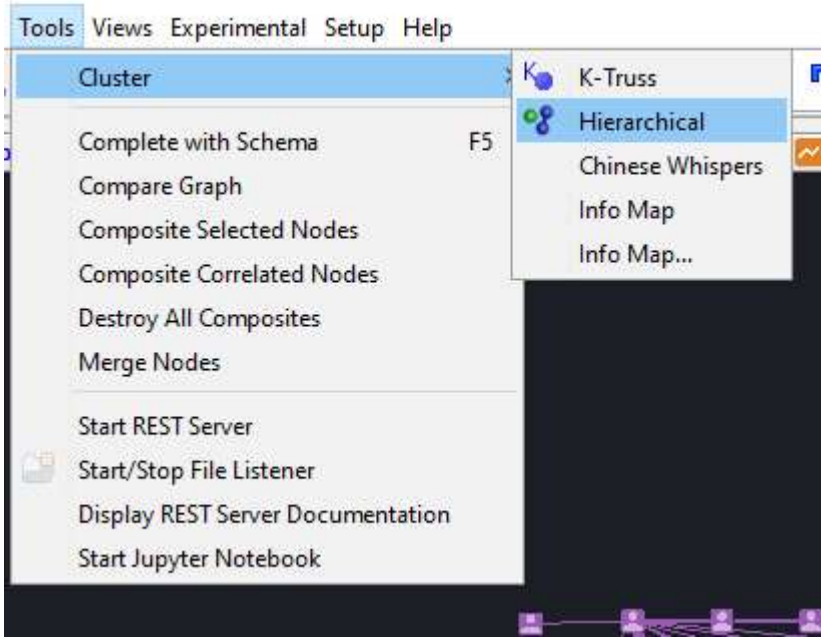

Close the View to reset the graph back to the default schema style.

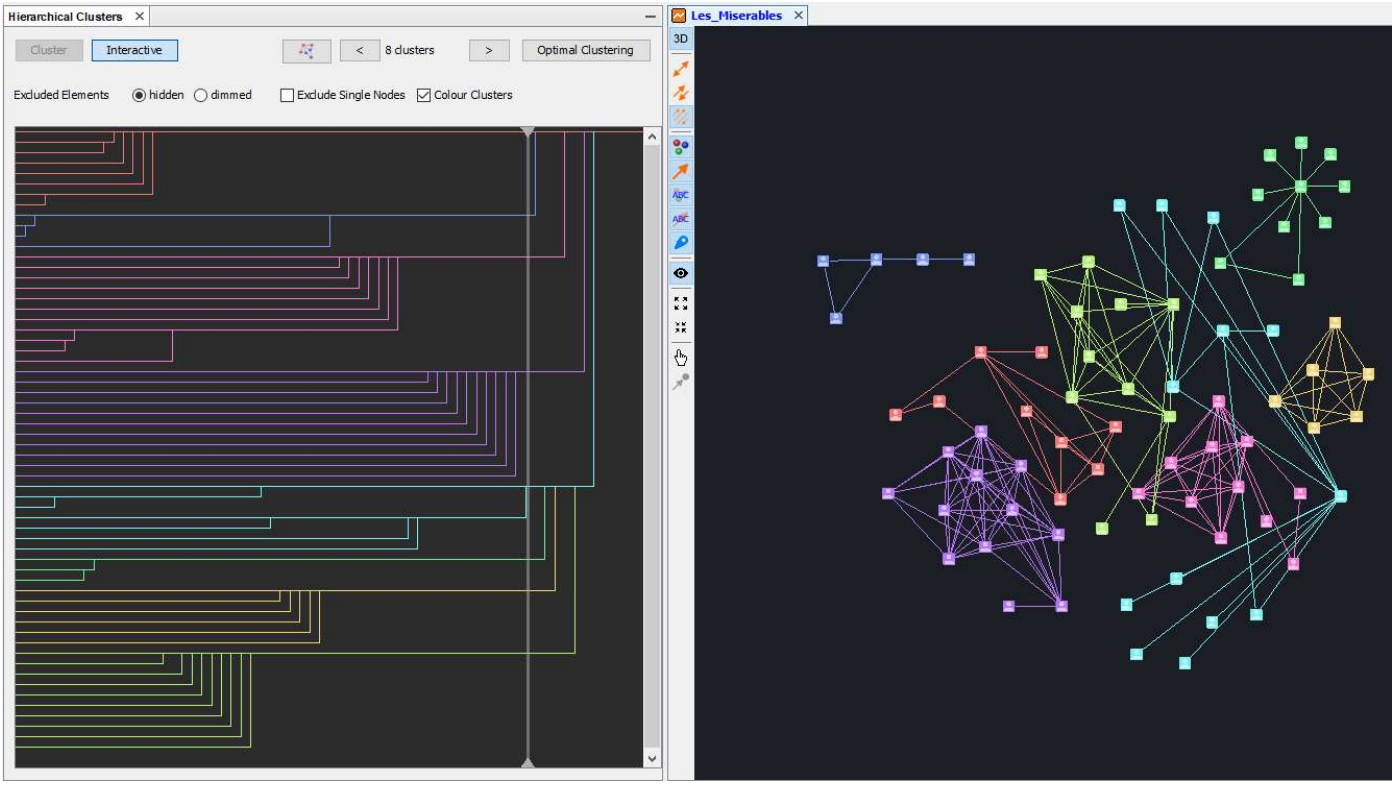
# Hierarchical Clustering

One of the limitations of k-Trusses is that it will only find communities if triangles exist. Hierarchical Clustering is an example of a clustering algorithm that puts **every** node in a group. It is based on the concept that communities are more likely to share connections **within** the group than to entities **outside** the group.

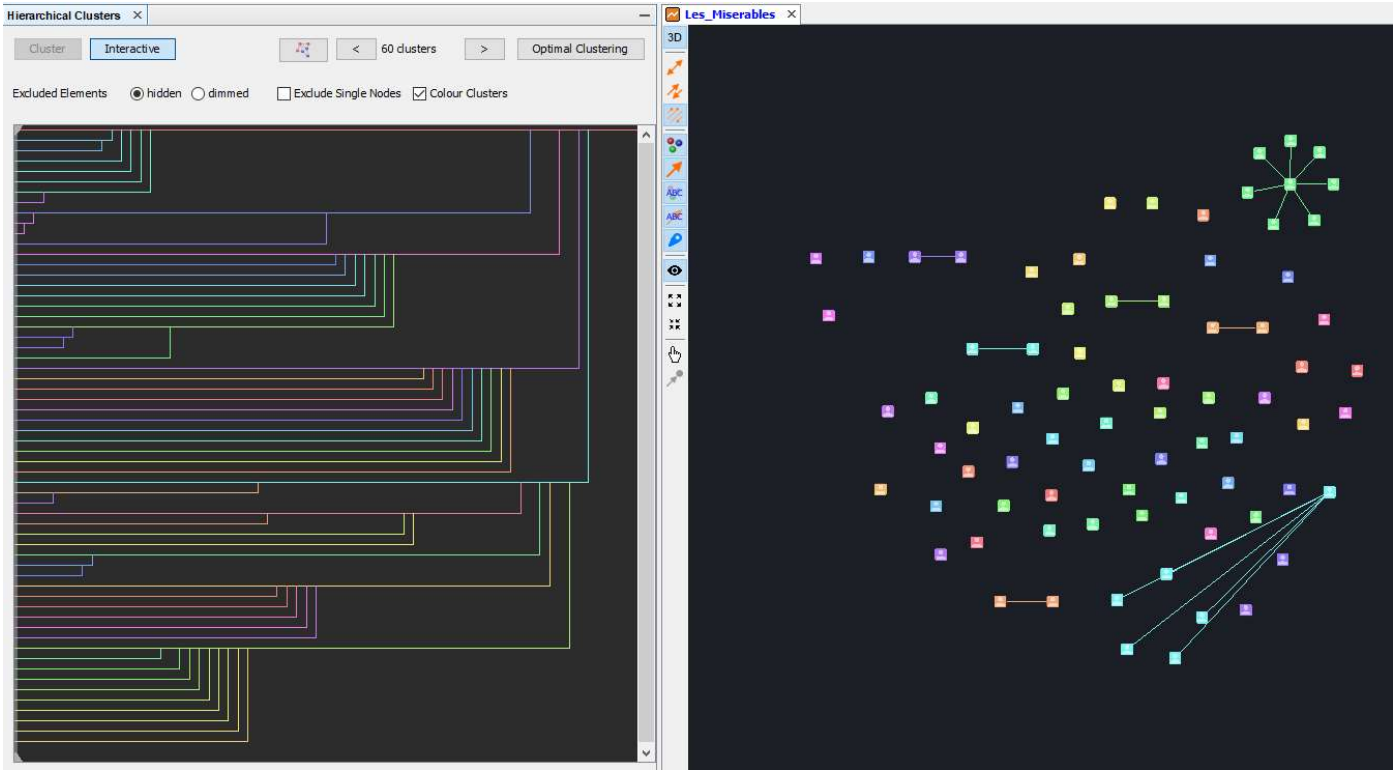
It starts by assigning every node a community. Then, it randomly groups nodes together along edges, and then checks to see whether the internal cohesiveness of the new group is stronger than the overall cohesiveness of the network outside. If yes, it keeps going, if no, it tries again, and it keeps repeating this process until every node is in one group. At the end of this process, it goes back to the point where it calculated optimal cohesion between groups.

Hierarchical Clustering can be found just below k-Truss in the Tools -> Clustering menu.

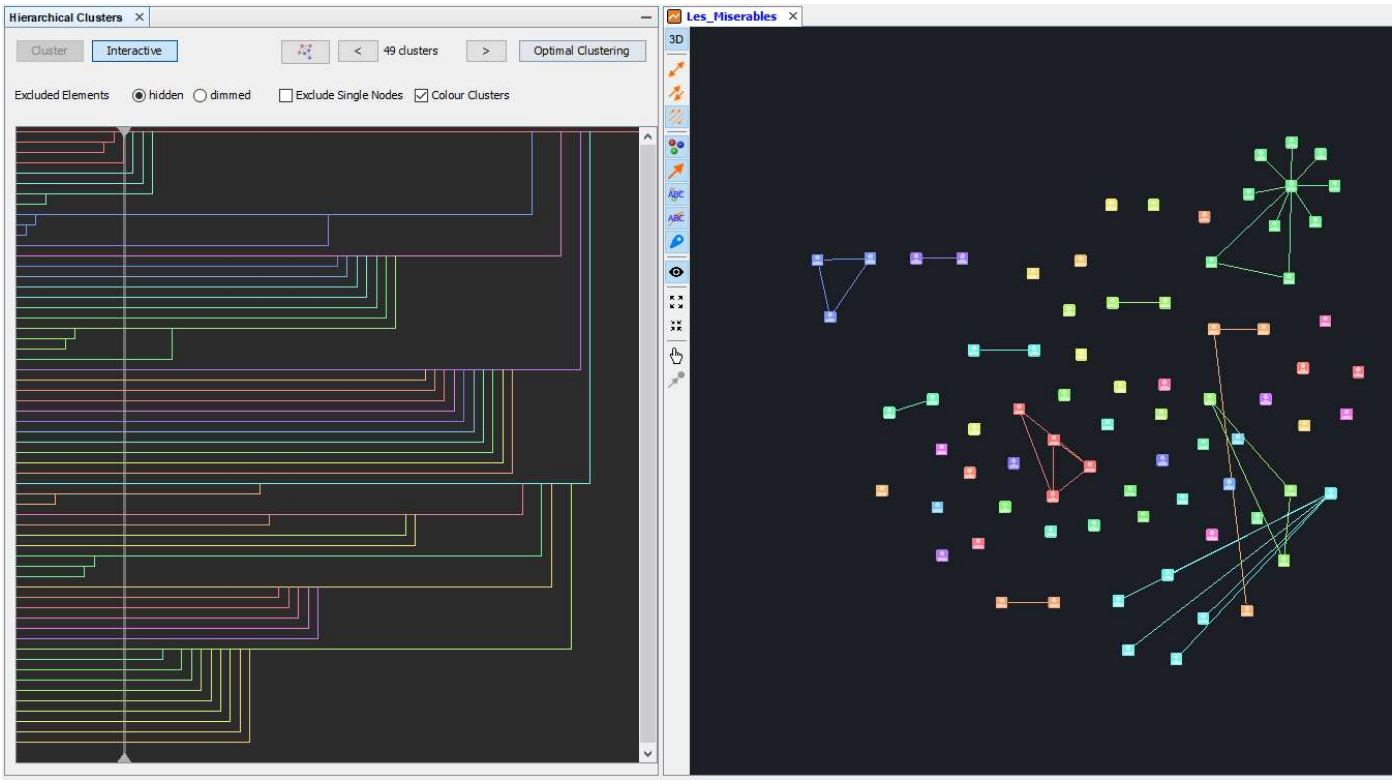


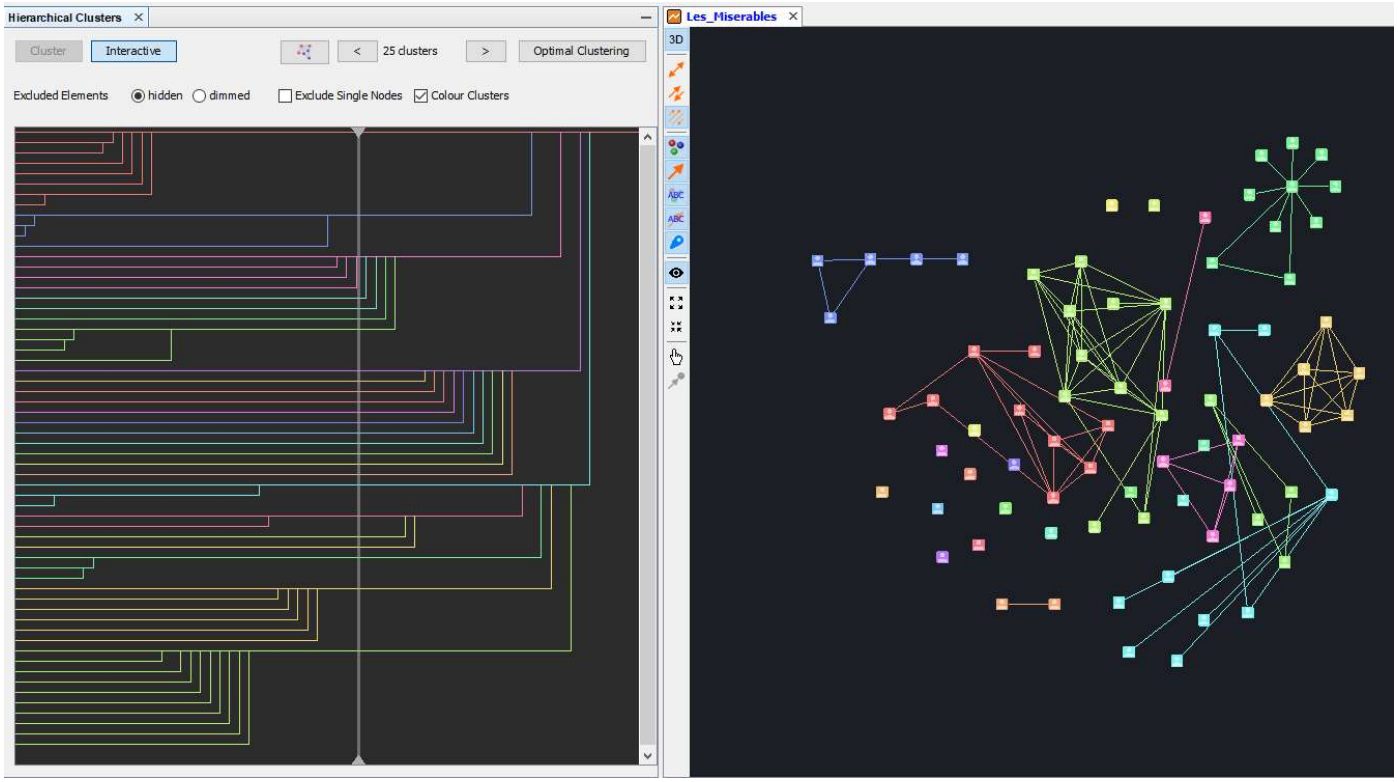



After hitting Cluster, it generates the communities. You can drag the slider to the left to see all the initial communities, and slowly move it to the right to see how the algorithm went about creating communities.

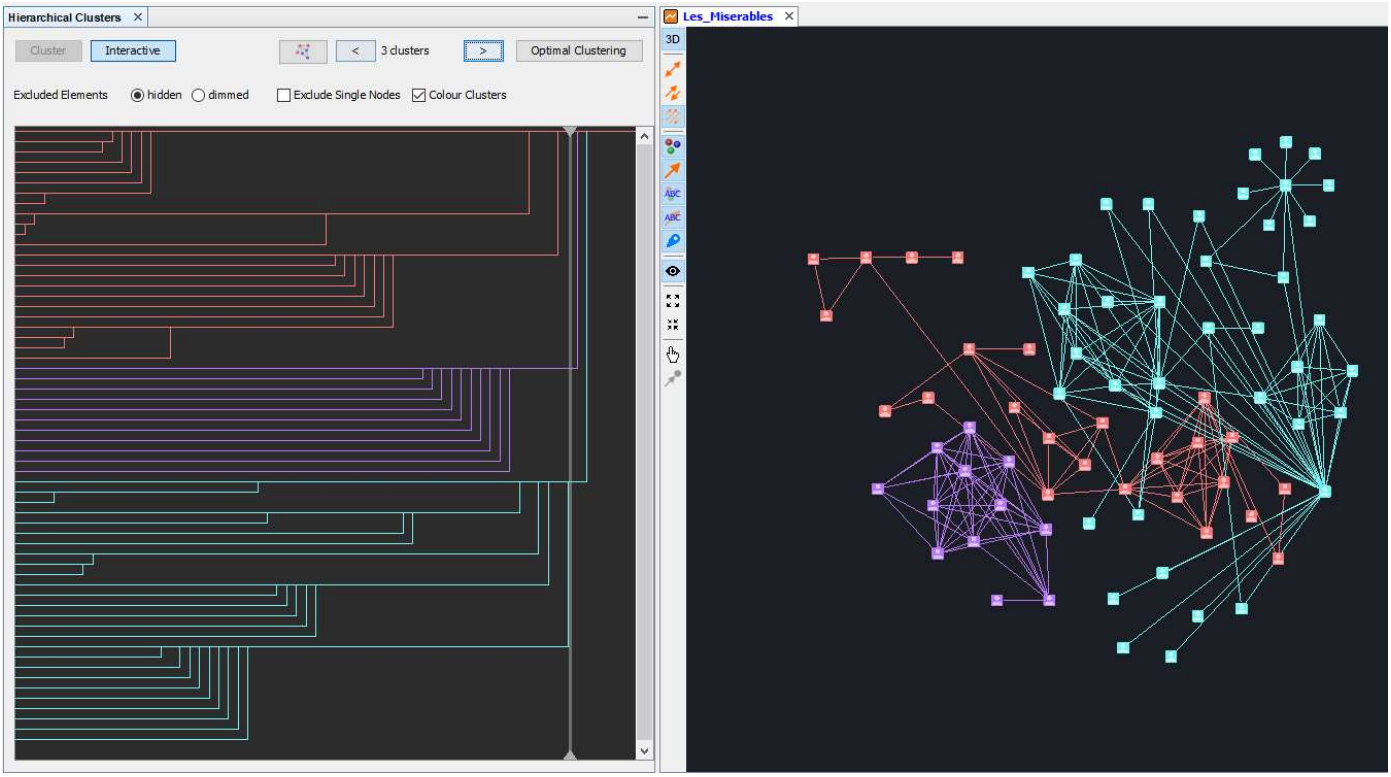


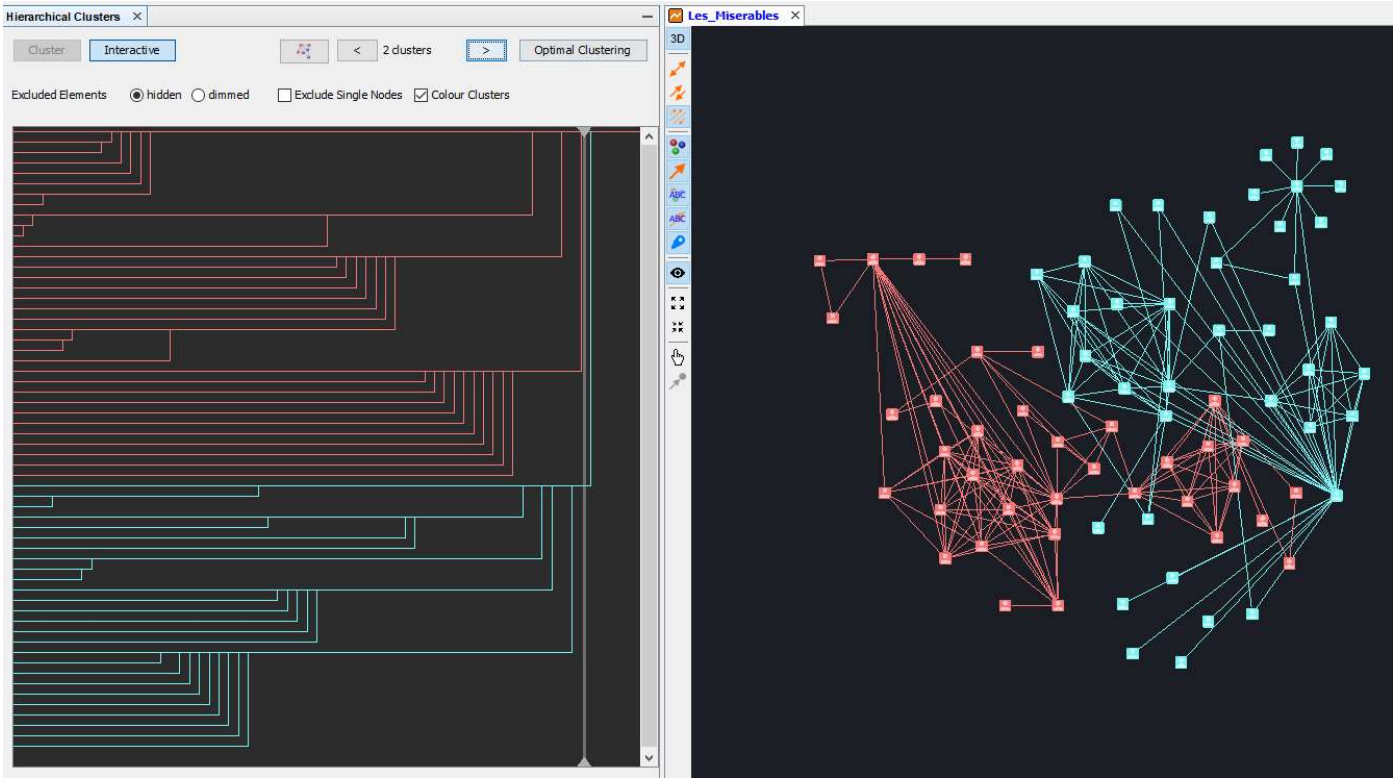

Above, every node is in its own community (except pendants)

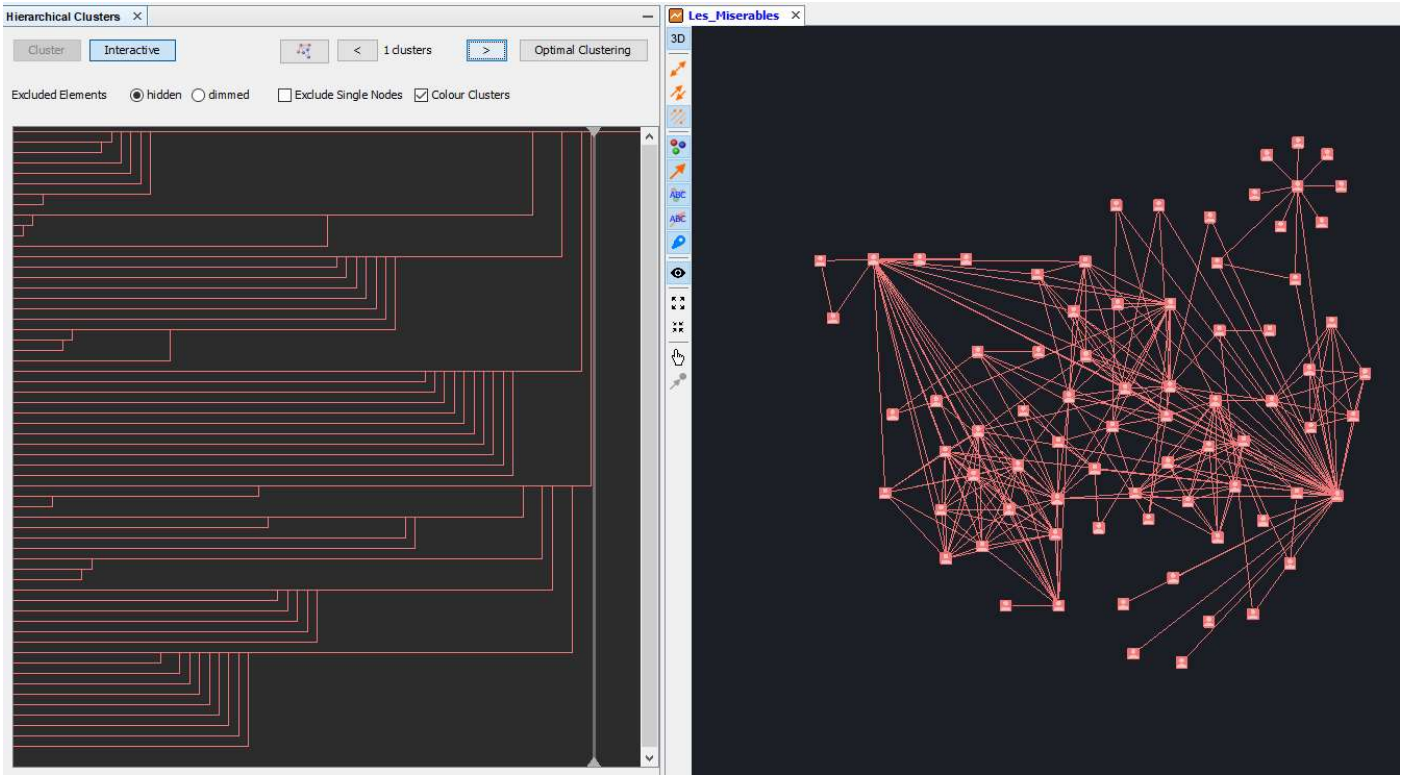


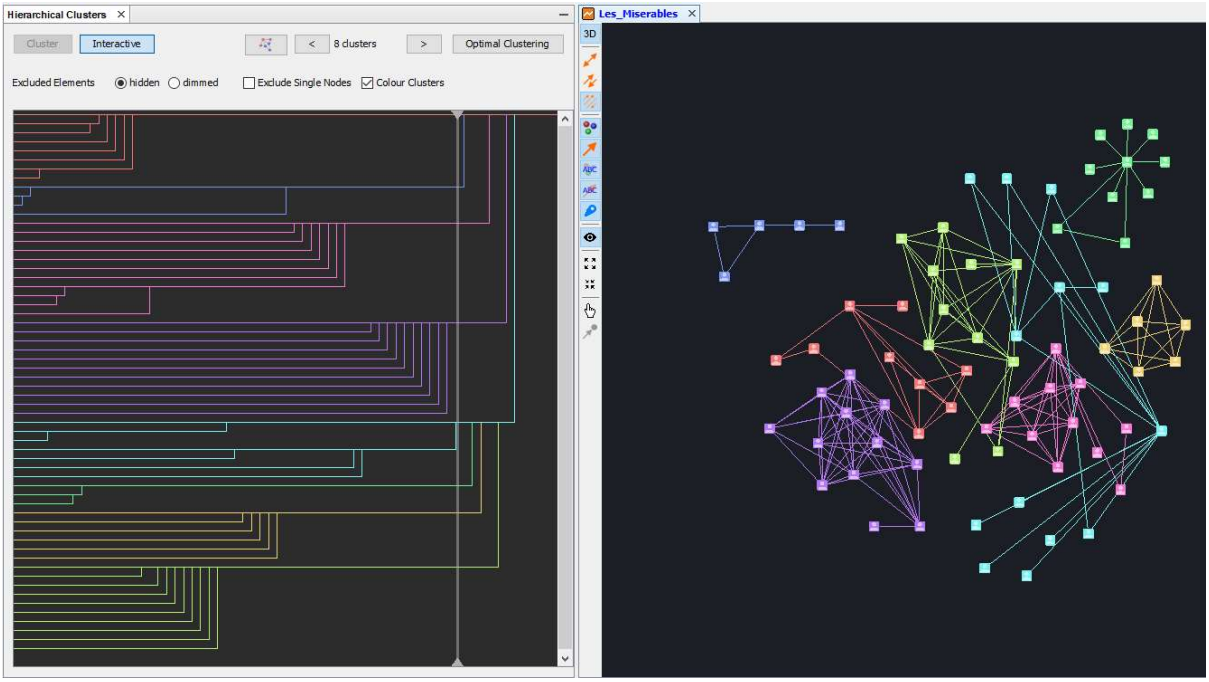

As we get closer to the right, we can see how the Hierarchical Clustering is partitioning the graph clearly.


Hierarchical Clusters
Cluster
Interactive
< 3 clusters >
Optimal Clustering
Excluded Elements
hidden
dimmed
Exclude Single Nodes
Colour Clusters
Les_Miserables
3D

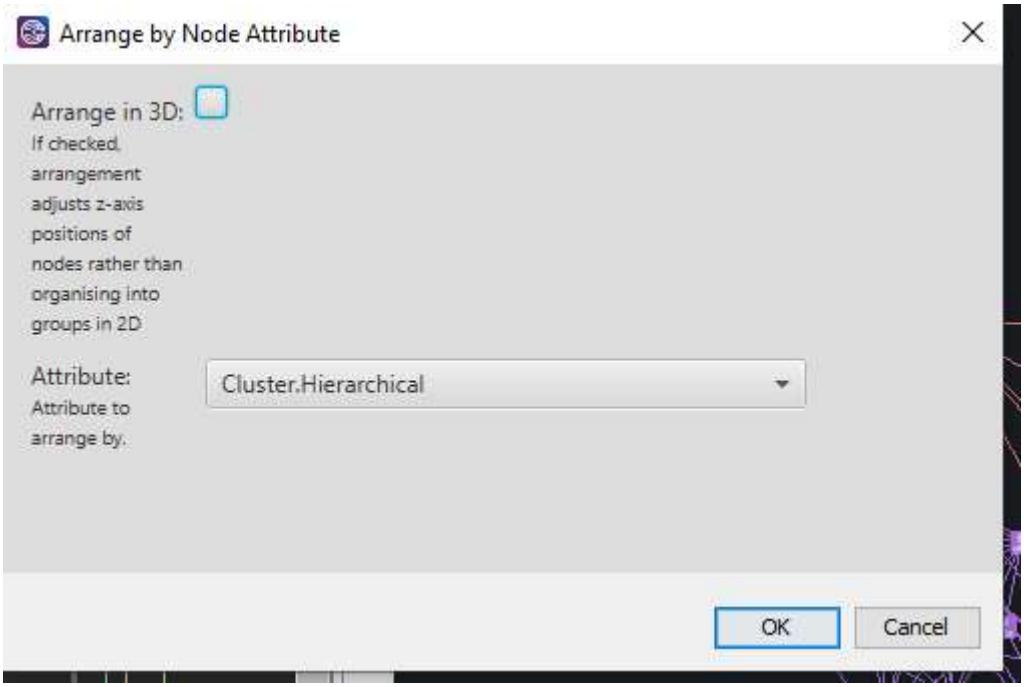

Hierarchical Clusters
Cluster
Interactive
< 2 clusters >
Optimal Clustering
Excluded Elements
hidden
dimmed
Exclude Single Nodes
Colour Clusters
Les_Miserables
3D

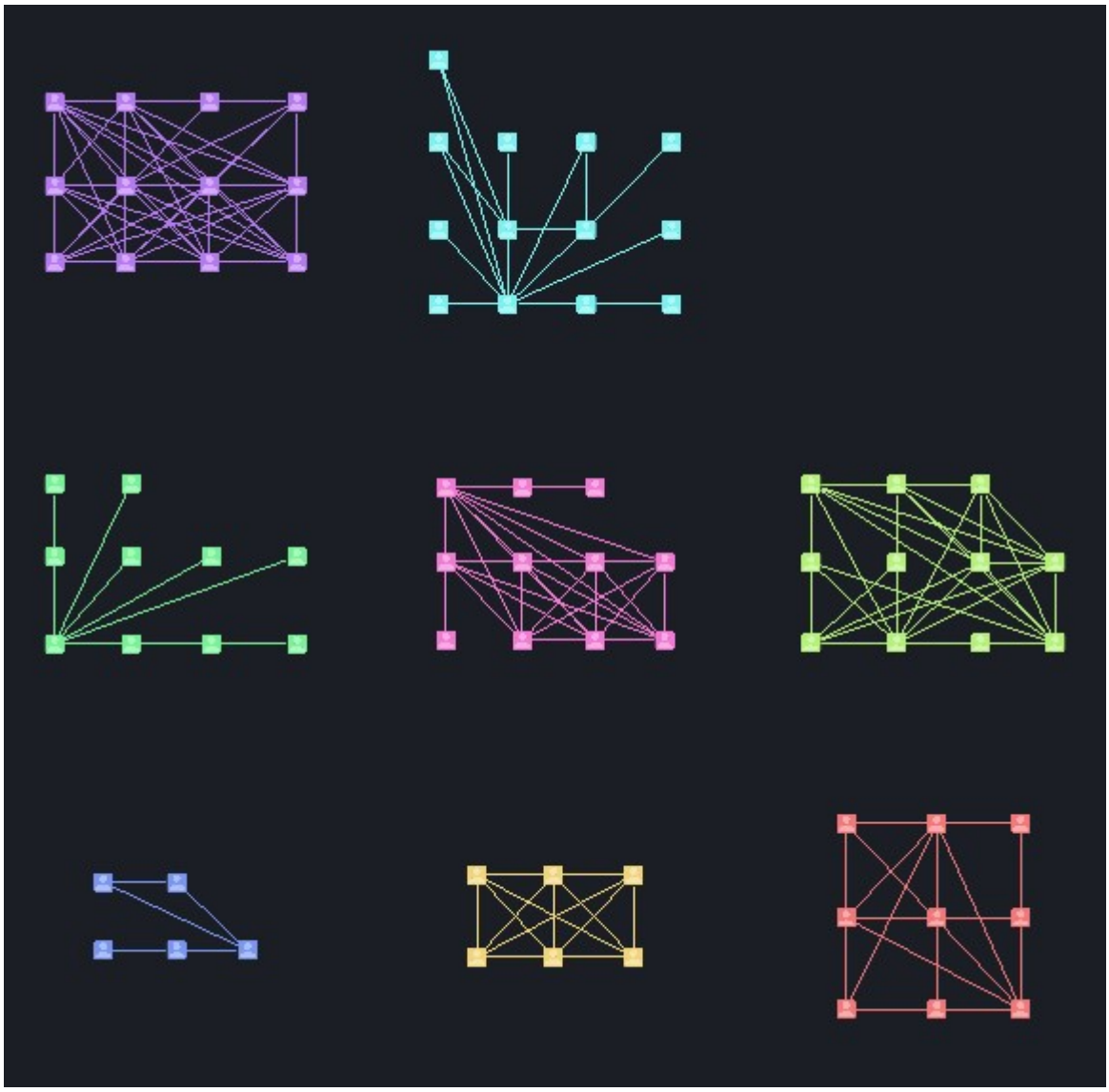



Press Optimal Clustering to go back to where it felt it did the best job of creating communities. The shortest paths button will identify the shortest paths between clusters, and the nodes that act as **gateways** between communities.




A useful way of visualising the communities is to Arrange By Node Attribute (found under the Arrange Menu)


Arrange by Node Attribute
Arrange in 3D:
If checked, arrangement adjusts z-axis positions of nodes rather than organising into groups in 2D
Attribute:
Attribute to arrange by.
Cluster.Hierarchical
OK
Cancel

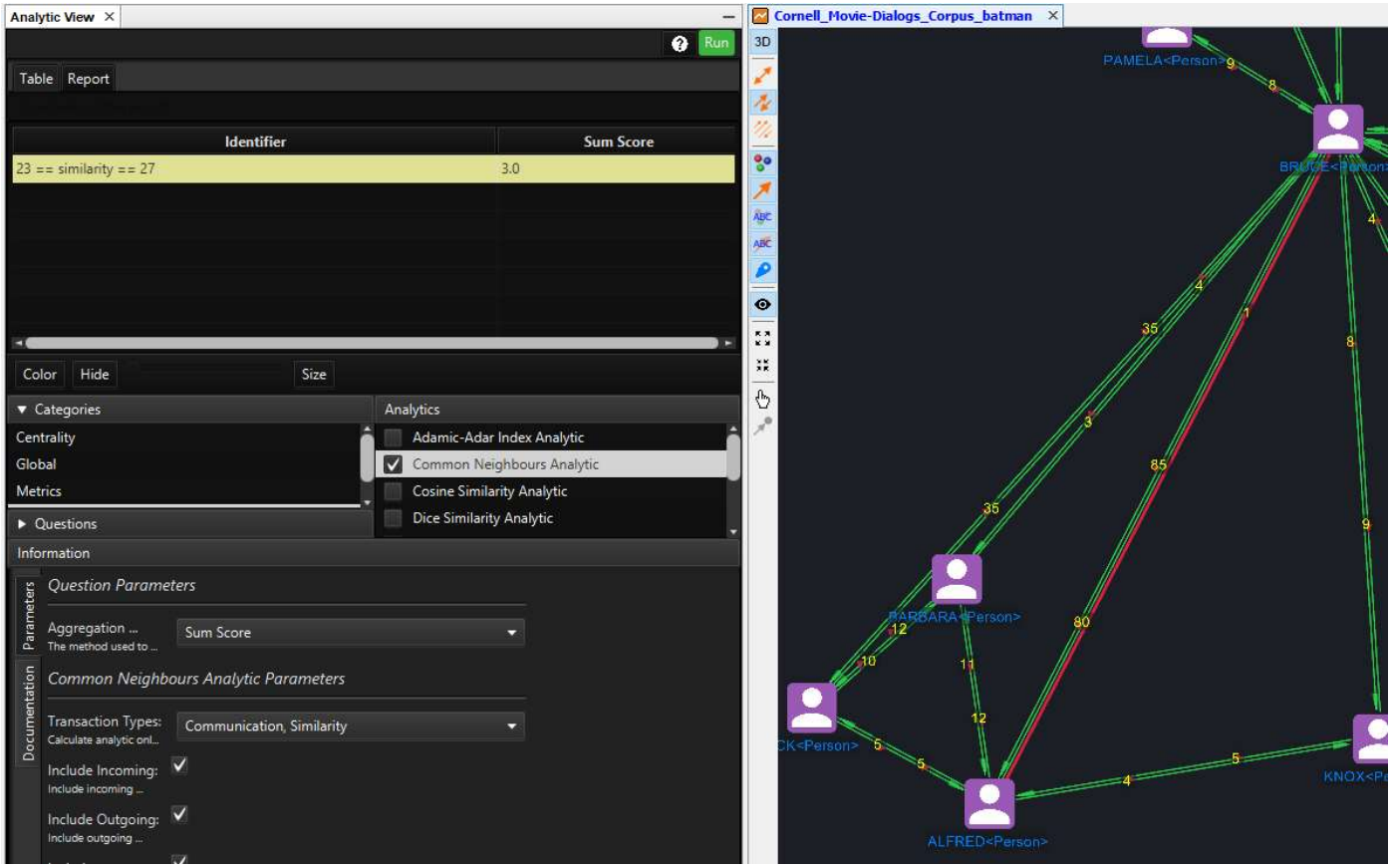


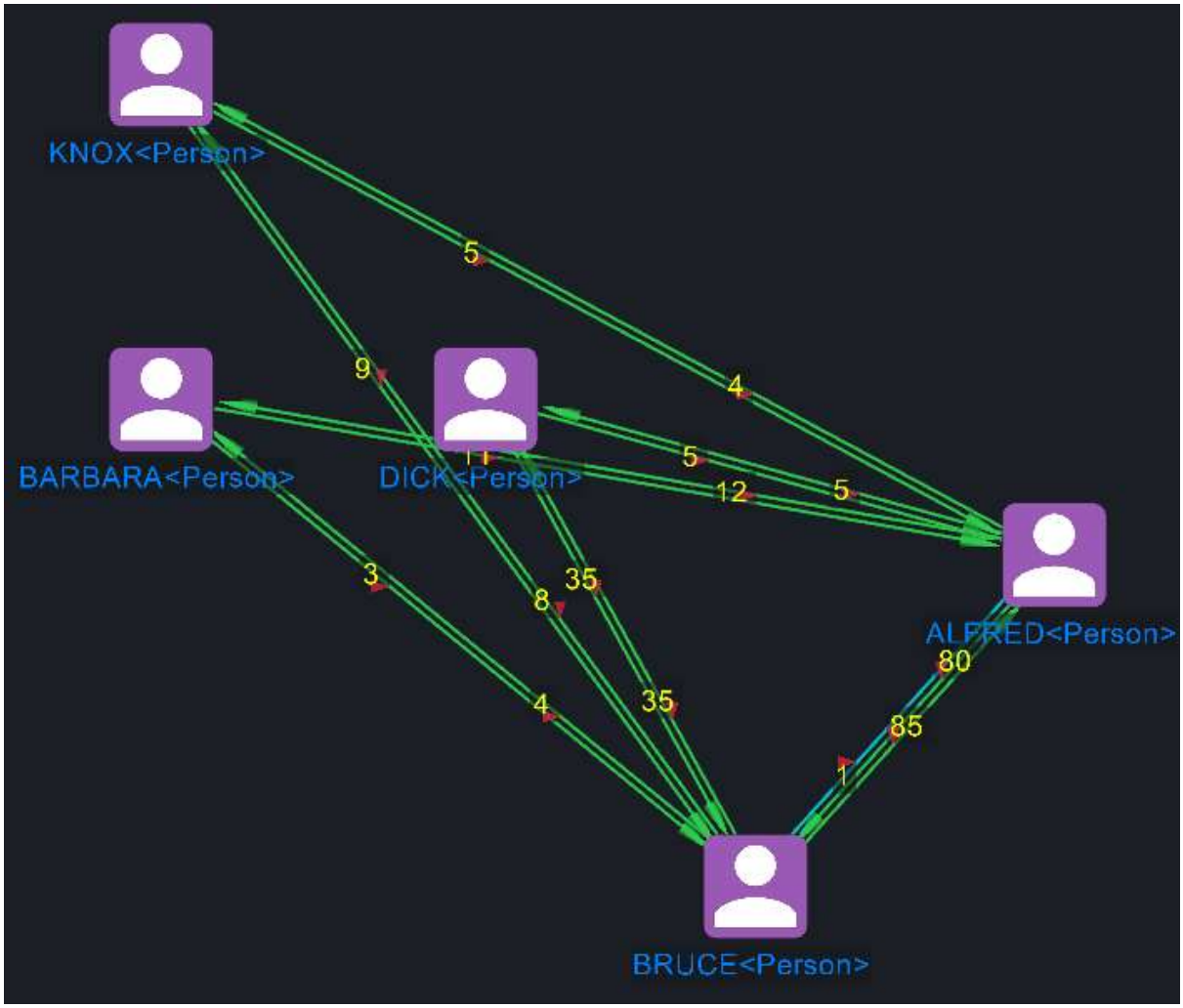
## Similarity

Similarity is about measuring how many **features** two entities have in common. In graph theory, this is usually based on the neighbours two nodes have in common. This is useful because it can tell us whether two entities potentially belong to the same group or community, but it can also flag instances where links **might** exist between two entities (Link Prediction) as well as highlighting where two entities might be the **same**, based on their behaviours (Entity Resolution)

Earlier, we used the One-Hop Induced Subgraph to select common neighbours between entities. In the Analytic View, the Common Neighbours Similarity Analytic will calculate scores for every pair of nodes on the graph which is based on the number of common neighbours for each pair.

In the example below, it has identified that Bruce and Alfred have 3 common neighbours.

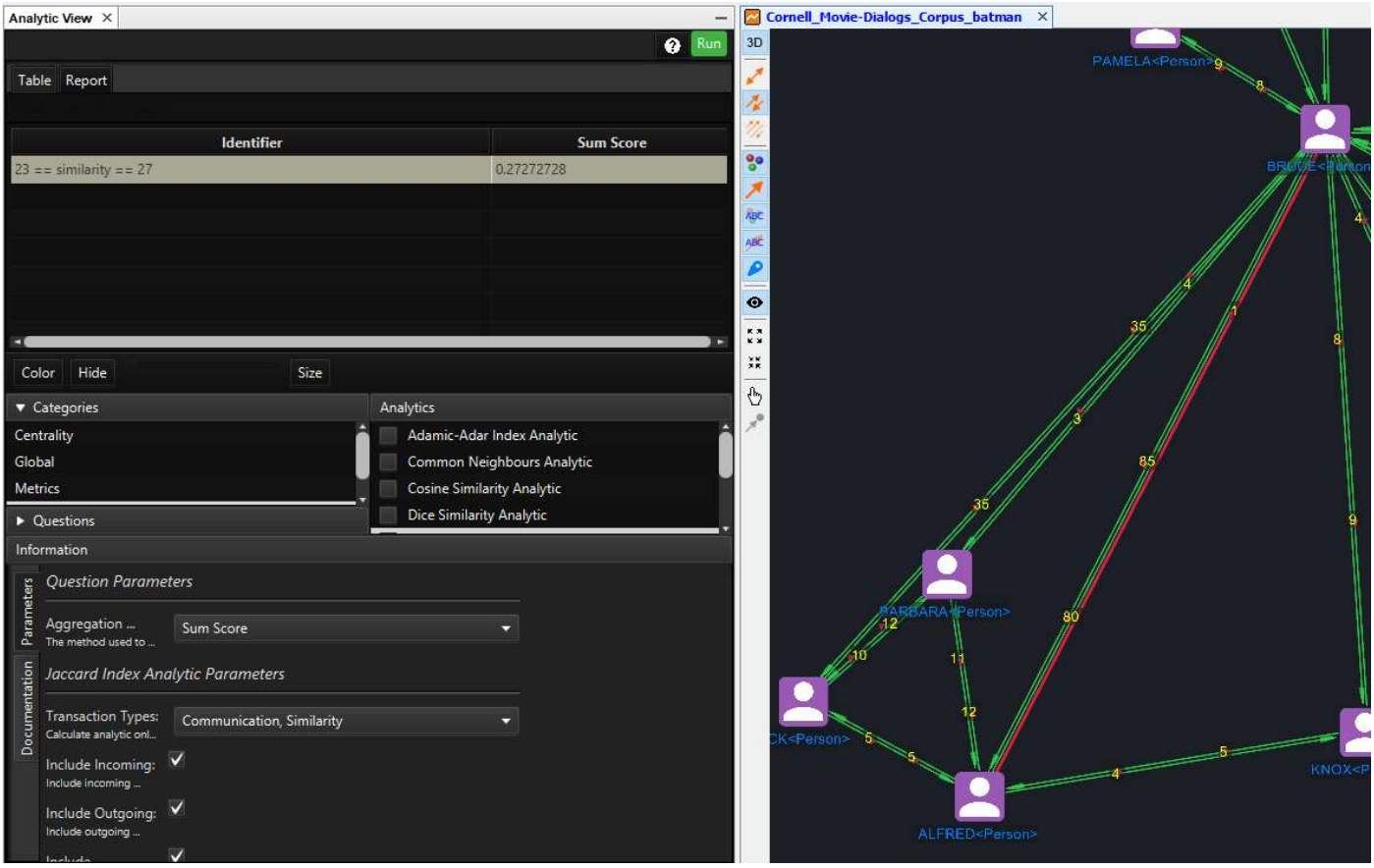


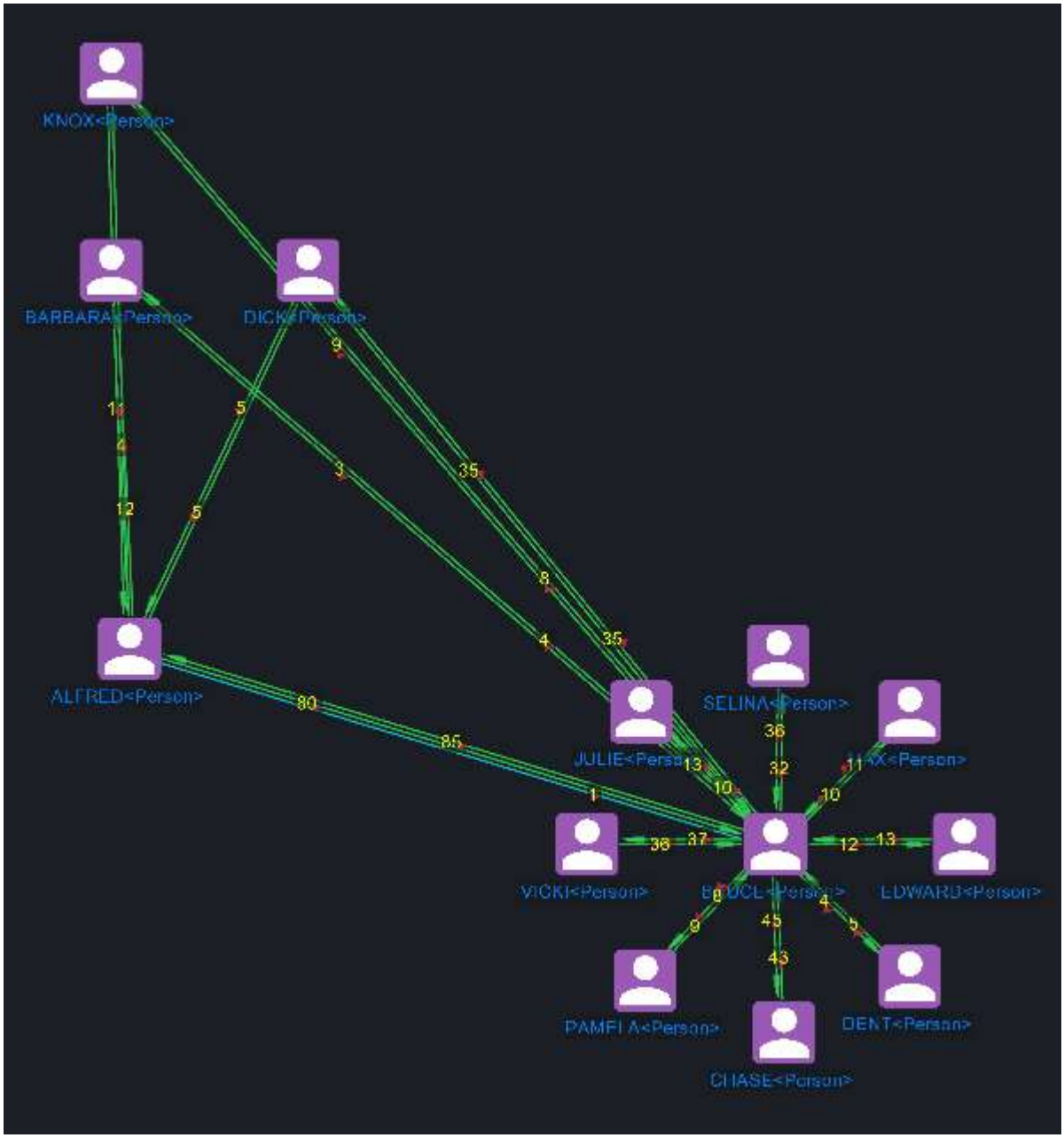



## Jaccard Index

Jaccard Index is a slightly more robust way of measuring similarity. It is the ratio of how many neighbours/features two nodes have in common, over the total number of features/neighbours they have. This is a commonly used metric to measure the distance between sets.

Below, Bruce and Alfred have a Jaccard Index of .2727, which is equal to 3/11, the number of neighbours they have in common over the total number of neighbours they have.


Analytic View
Run
Table
Report
Identifier
Sum Score
23 == similarity == 27
0.27272728
Color
Hide
Size
Categories
Centrality
Global
Metrics
Analytics
Adamic-Adar Index Analytic
Common Neighbours Analytic
Cosine Similarity Analytic
Dice Similarity Analytic
Questions
Information
Parameters
Documentation
Question Parameters
Aggregation ...
The method used to ...
Sum Score
Jaccard Index Analytic Parameters
Transaction Types:
Calculate analytic onl...
Communication, Similarity
Include Incoming:
Include incoming ...
Include Outgoing:
Include outgoing ...
Cornell_Movie-Dialogs_Corpus_batman
3D
PAMELA<Person>
BRUCE<Person>
BARBARA<Person>
ALFRED<Person>
KNOX<P



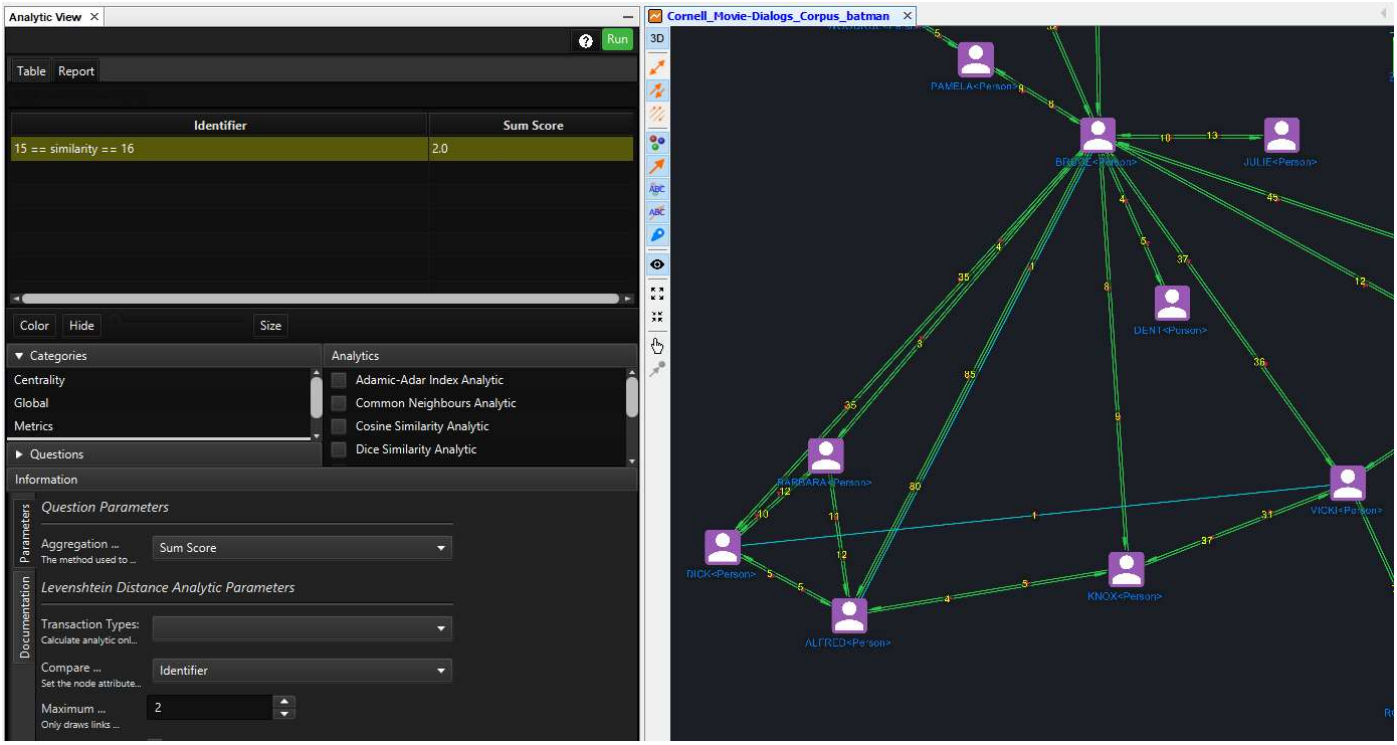

## Levenshtein Distance

Levenshtein distance is another distance measure which is based on the character strings. When comparing two strings, distance can be calculated by measuring how many changes you need to apply to one string in order to change it to another. Changes can include adding a character/letter, deleting a character/letter, or substituting a character/letter.

This example (from Wikipedia) highlights the Levenshtein distance between "kitten" and "sitting"

The Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. **k**itten → **s**itten (substitution of "s" for "k")
2. sitt**e**n → sitt**i**n (substitution of "i" for "e")
3. sittin → sittin**g** (insertion of "g" at the end).




In the Analytic View, you just need to specify which Attribute to compare (default is Identifier) and also the maximum number of changes to set before the strings are two different to be considered possibly the same entity (default is 1, above it has been set to 2.) We can see that it has linked Dick and Vicki, as these two strings have a Levenshtein Distance of 2.