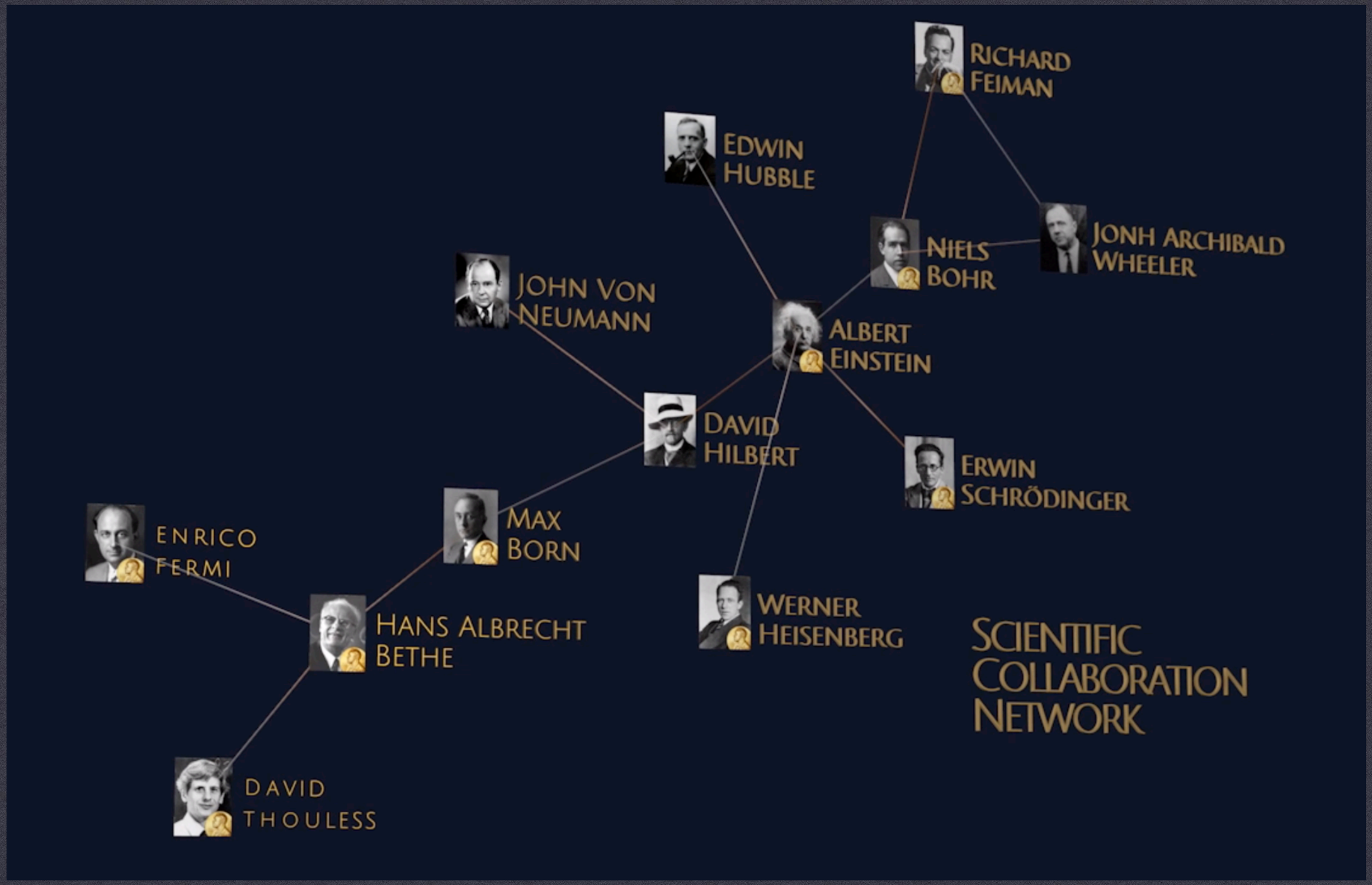


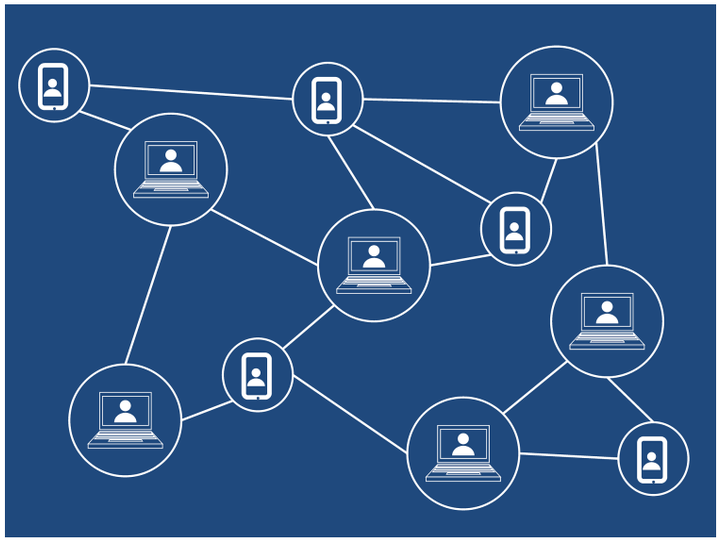
ANALYZING MASSIVE GRAPHS - PART I

DR. MICHAEL FIRE

Graphs/Networks are Everywhere

Some examples:

- Online Social Networks
- Computer Networks
- Financial Transactions Networks
- Protein Networks
- Neuron Networks
- Animal Social Networks



Networks/Graphs are Well-Studied

Due to their cross-disciplinary usefulness today, there are:

- Numerous graph algorithms
- Many tools/frameworks to analyze graphs
- Many tools to visualize graphs

igraph

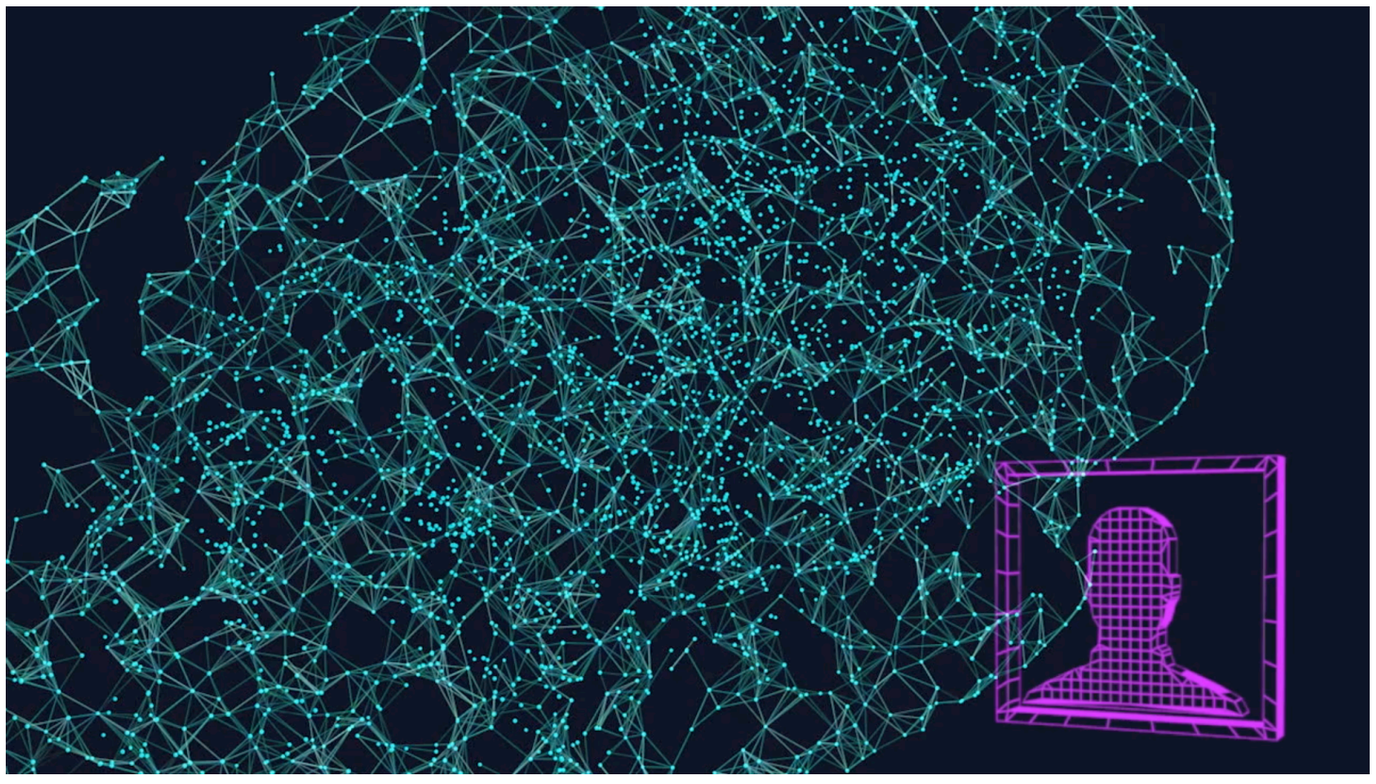
Gephi

Networkx

Graphs - The Basics

A graph consists of a set of vertices (nodes), and a set of links (edges). The links connect vertices.

There are direct graphs, like Twitter, and undirected graphs, like Facebook.



Practical Network Analysis

There are countless interesting things that we can learn from graphs:

- Graph Algorithms and Their Complexity
- Network Evolution Models
- Dynamic Networks
- Spreading Phenomena
- Temporal Networks
- Motifs

In today's lecture, we are going to learn how to use networks as data structures to analyze and infer insights from data

Brief History of Network Models

- In 1959, Erdős and Rényi developed random graph generation model
- In 1965, Price observed a network in which the degree distribution followed a power law. Later, in 1976, Price provided an explanation of the creation of these types of networks: "Success seems to breed success"
- In 1998, Watts and Strogatz introduced model for generating small-world networks
- In 1999, Baraba´si and Albert observed that degree distributions that follow power laws exist in a variety of networks, including the World Wide Web. Baraba´si and Albert coined the term "scale-free networks" for describing such networks.

Network Analysis Tools

- **Networkx -** a Python Library for graph analysis
- **iGraph -** a library collection for creating and manipulating graphs and analyzing networks. It is written in C and also exists as Python and R packages.
- **NodeXL** - NodeXL Basic is an open-source template that makes it easy to explore network graphs
- **NetworKit** - NetworKit is an open-source tool suite for high-performance network analysis. Its aim is to provide tools for the analysis of large networks.
- **SGraph** - A scalable graph data structure. The SGraph provides flexible vertex and edge query functions, and seamless transformation to and from an SFrame object

Networkx

Pros:

- Easy and fun to use
- Mature library
- Many implemented graph algorithms
- Vertices can be anything and edges can store attributes
- Easy to visualize graphs and to save them in many formats

Cons:

- Doesn’t work well with large graphs
- Slow

iGraph

Pros:

- Mature library
- Many implemented graph algorithms
- Pretty fast

Cons:

- Little tricky to use
- Doesn’t work well with very large graphs

SGraph

Pros:

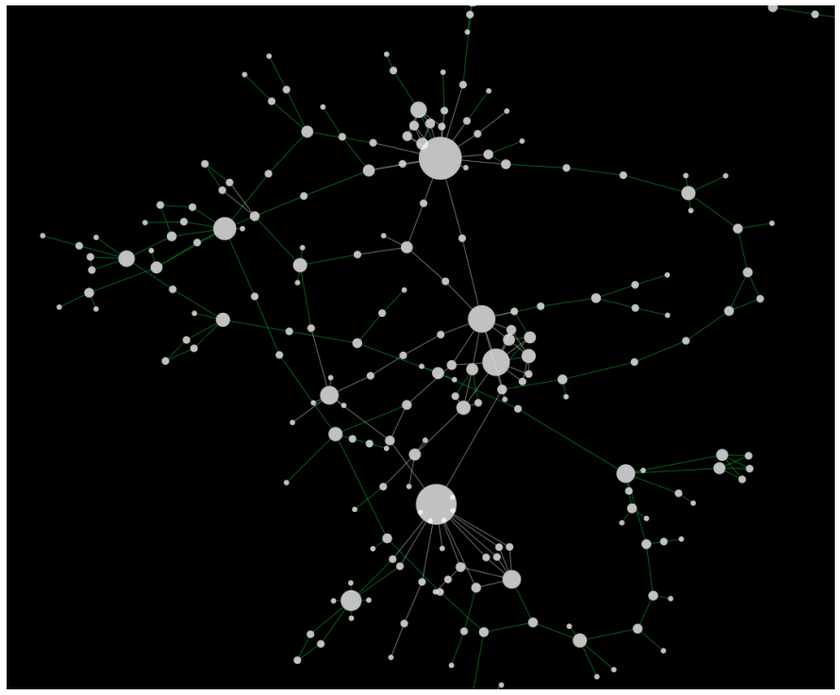
- Can work with very large graphs (billions of links)
- Part of the TuriCreate Eco-system (works great with SFrame)

Cons:

- Relatively little functionality

Network Visualization Tools

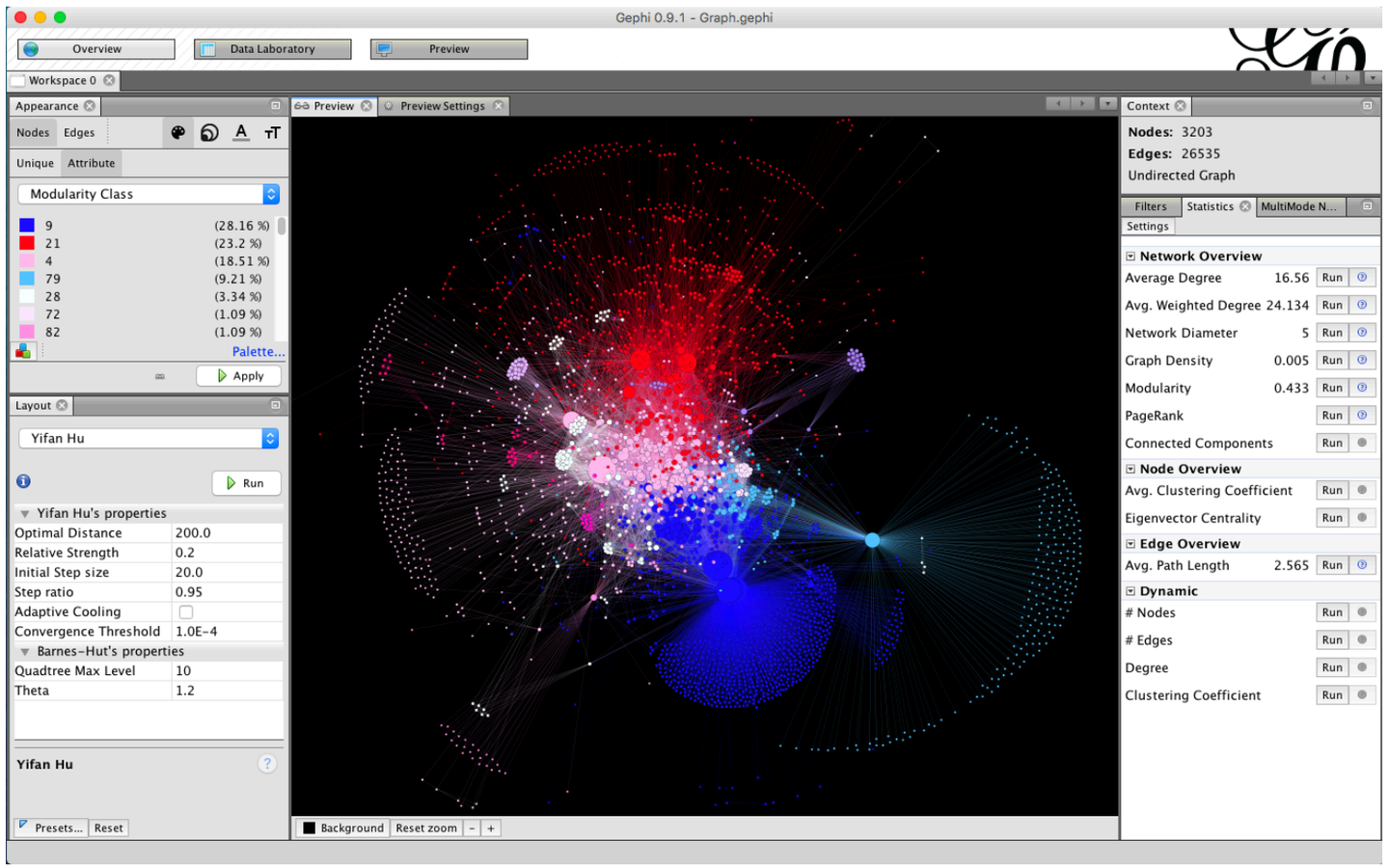
Cytoscape is an open source bioinformatics software platform for visualizing molecular interaction networks. Cytoscape can be used to visualize and analyze network graphs of any kind.



Yeast Protein–protein/Protein–DNA interaction network visualized by Cytoscape.

Network Visualization Tools

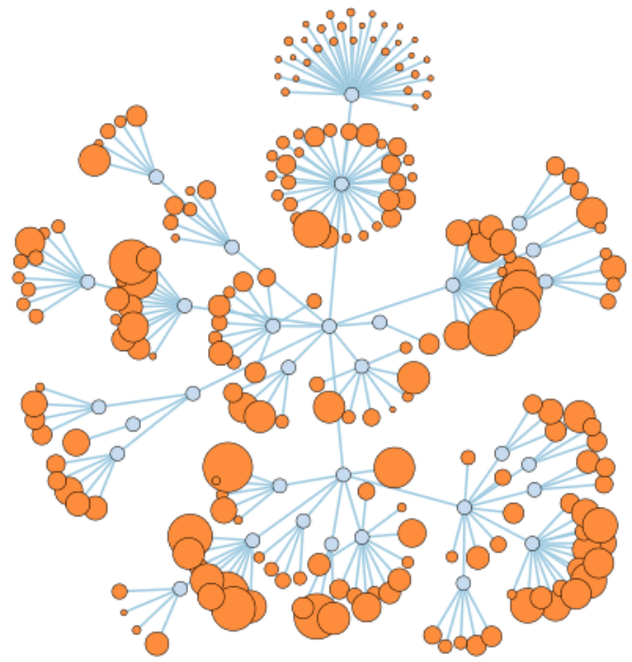
Gephi is an open-source network
analysis and visualization software package written in Java



Network Visualization Tools

D3.js is a JavaScript library for producing dynamic, interactive data visualizations in web browsers.

We can use D3.js to dynamically visualize networks



Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

`shortest_path` (G[, source, target, weight, ...])	Compute shortest paths in the graph.
`all_shortest_paths` (G, source, target[, ...])	Compute all shortest paths in the graph.
`shortest_path_length` (G[, source, target, ...])	Compute shortest path lengths in the graph.
`average_shortest_path_length` (G[, weight, method])	Return the average shortest path length.
`has_path` (G, source, target)	Return *True* if *G* has a path from *source* to *target*.

Advanced Interface

Shortest path algorithms for unweighted graphs.

`single_source_shortest_path` (G, source[, cutoff])	Compute shortest path between source and all other
`single_source_shortest_path_length` (G, source)	Compute the shortest path lengths from source to all
`single_target_shortest_path` (G, target[, cutoff])	Compute shortest path to target from all nodes that re
`single_target_shortest_path_length` (G, target)	Compute the shortest path lengths to target from all r
`bidirectional_shortest_path` (G, source, target)	Return a list of nodes in a shortest path between sour
`all_pairs_shortest_path` (G[, cutoff])	Compute shortest paths between all nodes.
`all_pairs_shortest_path_length` (G[, cutoff])	Computes the shortest path lengths between all node
`predecessor` (G, source[, target, cutoff, ...])	Returns dict of predecessors for the path from source

Shortest path algorithms for weighed graphs.

Some Terminology

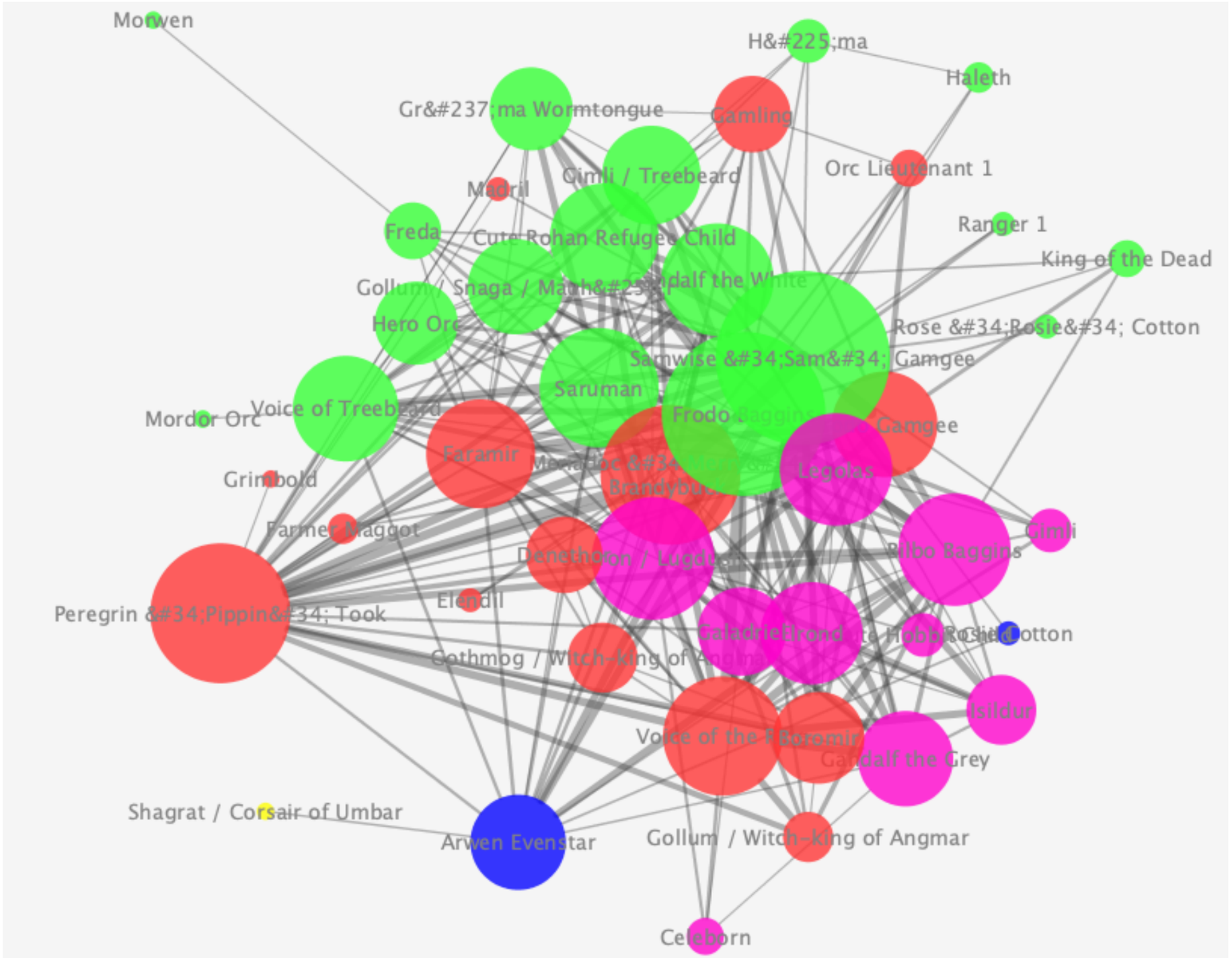
There are many interesting vertex/links/networks properties that are widely used in the context of analyzing networks. Here is a short list of terms we will need for the following examples:

- **Degree** **-** the number of edges incident to a vertex
- **Clustering Coefficient -** a measure of the degree to which nodes in a graph tend to cluster together.
- **Betweenness centrality** - a centrality measure which quantifies the number of times a vertex/edge acts as a bridge along the shortest path between two other nodes/edges.
- **Modularity -** measures the strength of division of a graph into modules (also called groups, clusters or communities)
- **Strongly Connected Component -** a maximal group of vertices that are mutually reachable
- **Weakly Connected Component** - a maximal group of vertices that are mutually reachable by **violating the edge directions.**
- **Common Friends** - Given two vertices u and v, the common friends of u and v is the size of the common set of friends that both u and v possess.

Some Useful Algorithms

- **Community Detection**
- **Shortest Path**
- **Closeness Centrality**
- **PageRank**
- **k-Core Decomposition**

Let’s move to running code using Jupyter Notebook


Morwen
Háma
Haleth
Gríma Wormtongue
Gamling
Orc Lieutenant 1
Madril
Gimli / Treebeard
Freda
Ranger 1
King of the Dead
Hero Orc
Rose "Rosie" Cotton
Samwise "Sam" Gamgee
Saruman
Mordor Orc
Voice of Treebeard
Gamgee
Faramir
Legolas
Grimbold
Farmer Maggot
Gimli
Bilbo Baggins
Peregrin "Pippin" Took
Elendil
Isildur
Boromir
Gandalf the Grey
Shagrat / Corsair of Umbar
Arwen Evenstar
Gollum / Witch-king of Angmar
Celeborn

Example: Closeness Centrality

Closeness centrality (or **closeness**) of a node is a measure of centrality in a network, calculated as the reciprocal of the sum of the length of the shortest paths **between the node and all other nodes in the graph**.

Thus, the more central a node is, the closer it is to all other nodes.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}$$

The Networkx Implementation

```
if distance is not None:
    # use Dijkstra's algorithm with specified attribute as edge weight
    path_length = functools.partial(nx.single_source_dijkstra_path_length,
                                    weight=distance)
else:
    path_length = nx.single_source_shortest_path_length

if u is None:
    nodes = G.nodes()
else:
    nodes = [u]
closeness_centrality = {}
for n in nodes:
    sp = path_length(G,n)
    totsp = sum(sp.values())
    if totsp > 0.0 and len(G) > 1:
        closeness_centrality[n] = (len(sp)-1.0) / totsp
        # normalize to number of nodes-1 in connected part
        if normalized:
            s = (len(sp)-1.0) / ( len(G) - 1 )
            closeness_centrality[n] *= s
    else:
        closeness_centrality[n] = 0.0
if u is not None:
    return closeness_centrality[u]
else:
    return closeness_centrality
```

The Networkx Implementation

single_source_shortest_path_length(*G, source, cutoff=None*) [source]

Compute the shortest path lengths from source to all reachable nodes.

```
    """
    seen={}                  # level (number of hops) when seen in BFS
    level=0                  # the current level
    nextlevel={source:1}  # dict of nodes to check at next level
    while nextlevel:
        thislevel=nextlevel  # advance to next level
        nextlevel={}         # and start a new list (fringe)
        for v in thislevel:
            if v not in seen:
                seen[v]=level # set the level of vertex v
                nextlevel.update(G[v]) # add neighbors of v
        if (cutoff is not None and cutoff <= level):  break
        level=level+1
    return seen  # return all path lengths as dictionary
```

Example: Grivan-Newman Community Detection Algorithm

The idea behind the algorithm:

- Iteratively remove edges
- focuses on edges that are most likely “between" communities, i.e. edges with the highest betweenness
- Stop were there are no-more edges
- As the graph breaks down into pieces, the tightly knit community structure is exposed and the result can be depicted as a dendrogram.

The Networkx Implementation

```
    """
    # If the graph is already empty, simply return its connected
    # components.
    if G.number_of_edges() == 0:
        yield tuple(nx.connected_components(G))
        return
    # If no function is provided for computing the most valuable edge,
    # use the edge betweenness centrality.
    if most_valuable_edge is None:
        def most_valuable_edge(G):
            """Returns the edge with the highest betweenness centrality
            in the graph `G`.

            """
            # We have guaranteed that the graph is non-empty, so this
            # dictionary will never be empty.
            betweenness = nx.edge_betweenness_centrality(G)
            return max(betweenness, key=betweenness.get)
    # The copy of G here must include the edge weight data.
    g = G.copy().to_undirected()
    # Self-loops must be removed because their removal has no effect on
    # the connected components of the graph.
    g.remove_edges_from(nx.selfloop_edges(g))
    while g.number_of_edges() > 0:
        yield _without_most_central_edges(g, most_valuable_edge)
```

The Networkx Implementation

```python
def _without_most_central_edges(G, most_valuable_edge):
    """Returns the connected components of the graph that results from
    repeatedly removing the most "valuable" edge in the graph.

    `G` must be a non-empty graph. This function modifies the graph `G`
    in-place; that is, it removes edges on the graph `G`.

    `most_valuable_edge` is a function that takes the graph `G` as input
    (or a subgraph with one or more edges of `G` removed) and returns an
    edge. That edge will be removed and this process will be repeated
    until the number of connected components in the graph increases.

    """
    original_num_components = nx.number_connected_components(G)
    num_new_components = original_num_components
    while num_new_components <= original_num_components:
        edge = most_valuable_edge(G)
        G.remove_edge(*edge)
        new_components = tuple(nx.connected_components(G))
        num_new_components = len(new_components)
    return new_components
```

most_valuable_edge -> usually the default is the edge with the highest betweenness centrality

Edge Betweenness Centrality

Betweenness centrality of an edge e is the sum of the fraction of all-pairs shortest paths that pass through e:

$$c_B(e) = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)}$$

where V is the set of nodes,`sigma(s, t)` is the number of shortest (s,t)-paths, and $\sigma(s,t|e)$ is the number of those paths passing through edge e [2].

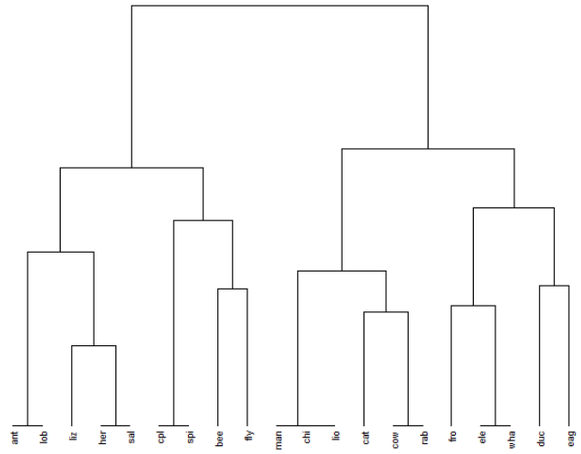
Using Girvan-Newman

```
In [1]: import networkx as nx

In [2]: from networkx.algorithms.community.centrality import *

In [3]: g =  nx.path_graph(10)

In [4]: list(girvan_newman(g))
Out[4]:
[({0, 1, 2, 3, 4}, {5, 6, 7, 8, 9}),
 ({0, 1}, {2, 3, 4}, {5, 6, 7, 8, 9}),
 ({0, 1}, {2, 3, 4}, {5, 6}, {7, 8, 9}),
 ({0, 1}, {2}, {3, 4}, {5, 6}, {7, 8, 9}),
 ({0, 1}, {2}, {3, 4}, {5, 6}, {7}, {8, 9}),
 ({0}, {1}, {2}, {3, 4}, {5, 6}, {7}, {8, 9}),
 ({0}, {1}, {2}, {3}, {4}, {5, 6}, {7}, {8, 9}),
 ({0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8, 9}),
 ({0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9})]
```



Recommended Read:

- Michael Fire: The Science of networks: Big Data in action
- Network Science Course by Carlos Castillo
- Top 30 Social Network Analysis and Visualization Tools by Devendra Desale
- Lecture 24 — Community Detection in Graphs - Motivation | Stanford University
- k-Core Decomposition: A tool for the visualization of Large Scale Networks by Alvarez-Hamelin, José Ignacio, et al.