Girvan-Newman Algorithm

CSCI 4964 - Final Presentation

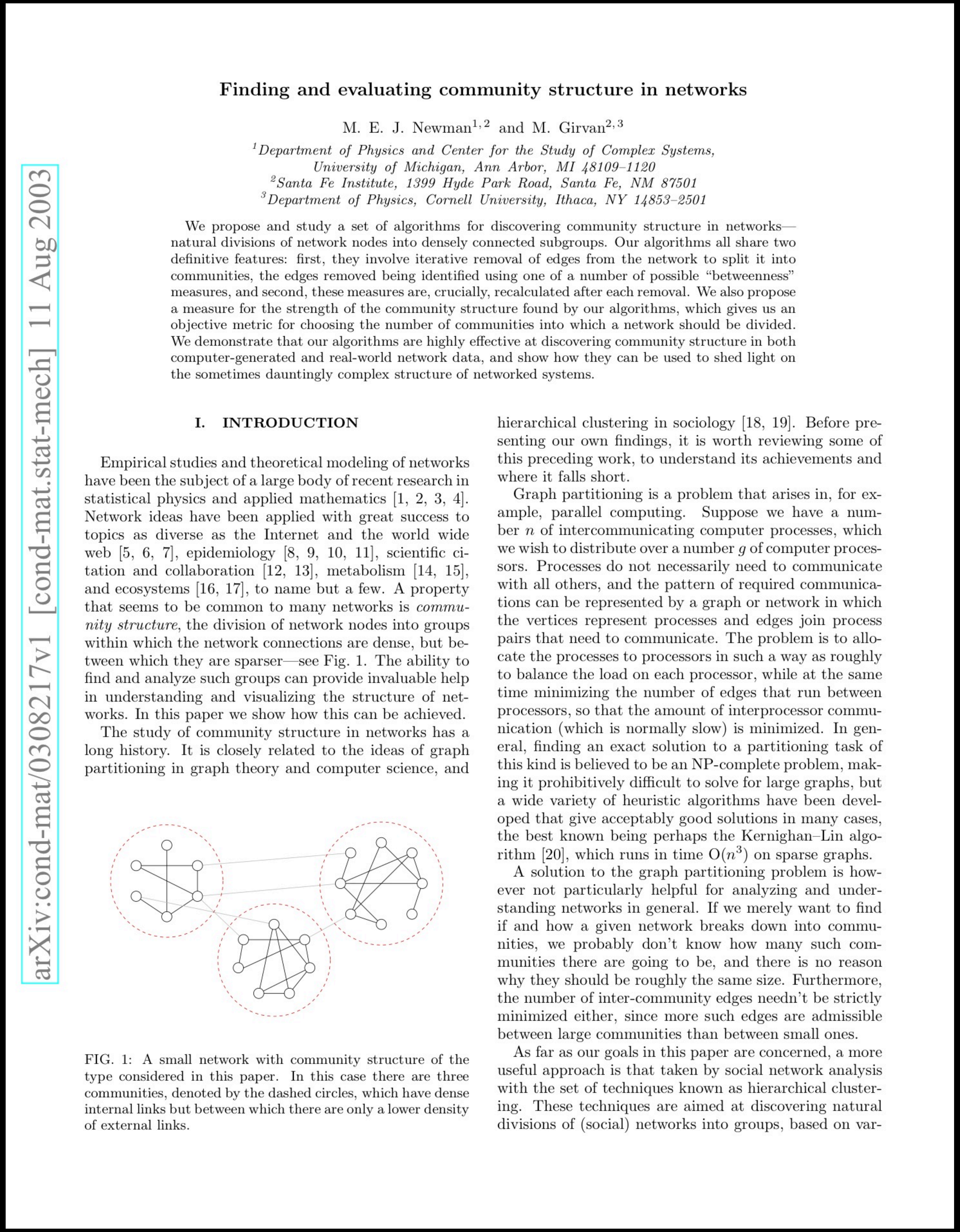
YIFAN (ELIO) LI

Overview

Paper Selection

Title: Finding and evaluating community structure in networks

Authors: Newman, Mark EJ, and Michelle Girvan.



arXiv:cond-mat/0308217v1 [cond-mat.stat-mech] 11 Aug 2003

Finding and evaluating community structure in networks

M. E. J. Newman[1,2] and M. Girvan[2,3]

[1]Department of Physics and Center for the Study of Complex Systems, University of Michigan, Ann Arbor, MI 48109–1120
[2]Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM 87501
[3]Department of Physics, Cornell University, Ithaca, NY 14853–2501

We propose and study a set of algorithms for discovering community structure in networks—natural divisions of network nodes into densely connected subgroups. Our algorithms all share two definitive features: first, they involve iterative removal of edges from the network to split it into communities, the edges removed being identified using one of a number of possible "betweenness" measures, and second, these measures are, crucially, recalculated after each removal. We also propose a measure for the strength of the community structure found by our algorithms, which gives us an objective metric for choosing the number of communities into which a network should be divided. We demonstrate that our algorithms are highly effective at discovering community structure in both computer-generated and real-world network data, and show how they can be used to shed light on the sometimes dauntingly complex structure of networked systems.

I. INTRODUCTION

Empirical studies and theoretical modeling of networks have been the subject of a large body of recent research in statistical physics and applied mathematics [1, 2, 3, 4]. Network ideas have been applied with great success to topics as diverse as the Internet and the world wide web [5, 6, 7], epidemiology [8, 9, 10, 11], scientific citation and collaboration [12, 13], metabolism [14, 15], and ecosystems [16, 17], to name but a few. A property that seems to be common to many networks is *community structure*, the division of network nodes into groups within which the network connections are dense, but between which they are sparser—see Fig. 1. The ability to find and analyze such groups can provide invaluable help in understanding and visualizing the structure of networks. In this paper we show how this can be achieved.

The study of community structure in networks has a long history. It is closely related to the ideas of graph partitioning in graph theory and computer science, and hierarchical clustering in sociology [18, 19]. Before presenting our own findings, it is worth reviewing some of this preceding work, to understand its achievements and where it falls short.

Graph partitioning is a problem that arises in, for example, parallel computing. Suppose we have a number n of intercommunicating computer processes, which we wish to distribute over a number g of computer processors. Processes do not necessarily need to communicate with all others, and the pattern of required communications can be represented by a graph or network in which the vertices represent processes and edges join process pairs that need to communicate. The problem is to allocate the processes to processors in such a way as roughly to balance the load on each processor, while at the same time minimizing the number of edges that run between processors, so that the amount of interprocessor communication (which is normally slow) is minimized. In general, finding an exact solution to a partitioning task of this kind is believed to be an NP-complete problem, making it prohibitively difficult to solve for large graphs, but a wide variety of heuristic algorithms have been developed that give acceptably good solutions in many cases, the best known being perhaps the Kernighan–Lin algorithm [20], which runs in time $O(n^3)$ on sparse graphs.

A solution to the graph partitioning problem is however not particularly helpful for analyzing and understanding networks in general. If we merely want to find if and how a given network breaks down into communities, we probably don't know how many such communities there are going to be, and there is no reason why they should be roughly the same size. Furthermore, the number of inter-community edges needn't be strictly minimized either, since more such edges are admissible between large communities than between small ones.

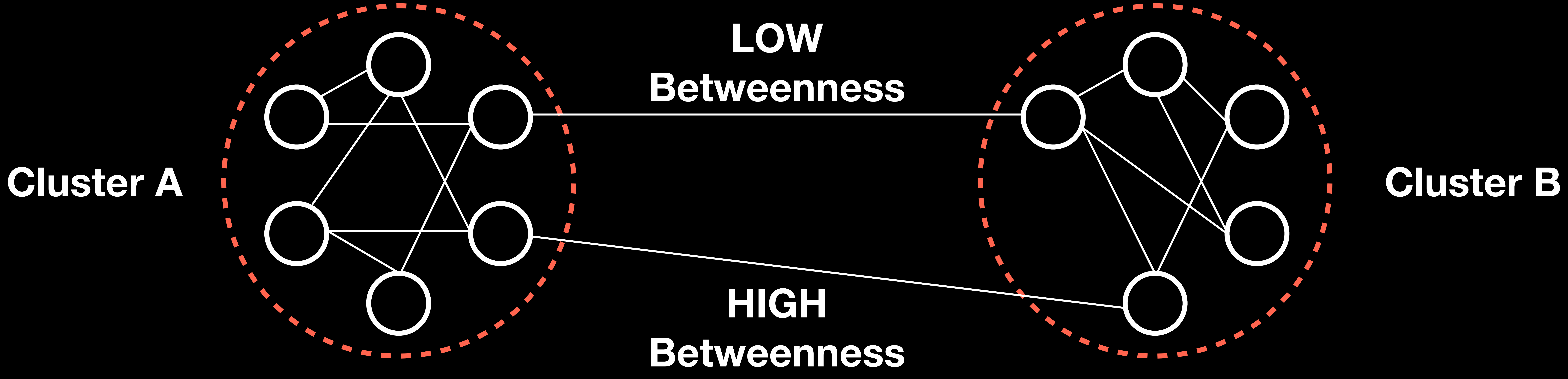
As far as our goals in this paper are concerned, a more useful approach is that taken by social network analysis with the set of techniques known as hierarchical clustering. These techniques are aimed at discovering natural divisions of (social) networks into groups, based on var-

FIG. 1: A small network with community structure of the type considered in this paper. In this case there are three communities, denoted by the dashed circles, which have dense internal links but between which there are only a lower density of external links.

Overview

Finding Communities in a Network

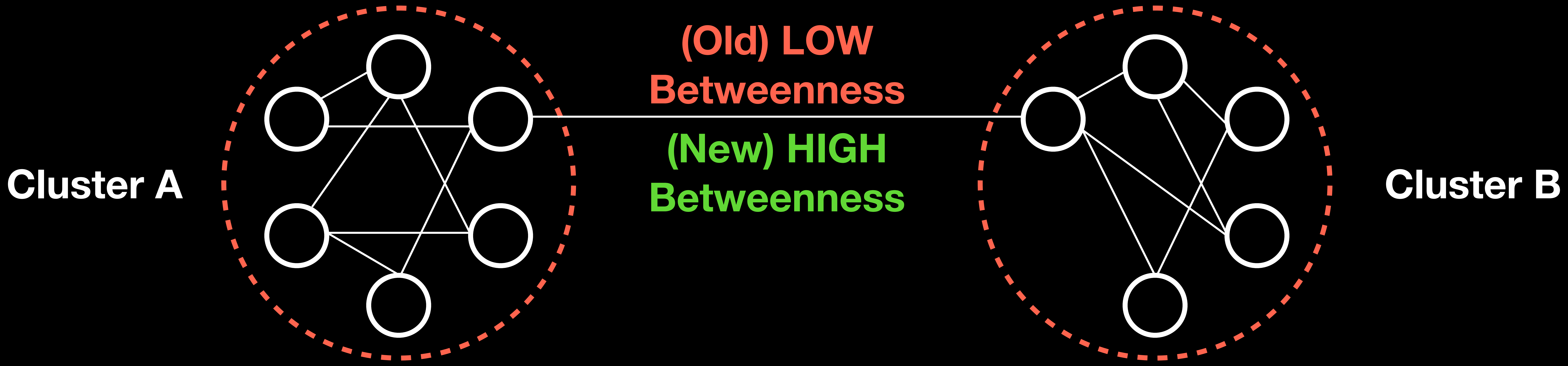
1. For every edge in a graph, calculate the edge betweenness centrality.
2. Remove the edge with the highest betweenness centrality.
3. Repeat steps 2 until there are no more edges left.



Overview

Finding Communities in a Network

1. For every edge in a graph, calculate the edge betweenness centrality.
2. Remove the edge with the highest betweenness centrality.
3. Repeat steps 2 until there are no more edges left.



Overview

Proposed Algorithm

1. For every edge in a graph, calculate the edge betweenness centrality.
2. Remove the edge with the highest betweenness centrality.
3. Calculate the betweenness centrality for every remaining edge.
4. Repeat steps 2-3 until there are no more edges left.

Overview

Betweenness

- A measure that favors edges that lie between communities and disfavors those that lie inside communities.
- 3 Types of Betweenness Measures
  - I. Shortest-Path Betweenness
  - II. Random-Walk Betweenness
  - III. Current-Flow Betweenness

Overview

Modularity (Q)

- e_{ii} : Probability of Edge is in the Same Module i
- a_i^2 : Probability of Random Edge Belongs in Module i
- $TR(e) = \sum_i e_{ii}$: Trace of e, Same as e_{ii}
- $\|e^2\|$: Sum of the Matrix Elements

$$Q = \sum_i (e_{ii} - a_i^2) = TR(e) - \|e^2\|$$

Network

Les Misrables Character
Directed

Node - Character in the Novel
Edge - Co-Appearance of the Corresponding Characters in One or More Scenes

Nodes: 77
Edges: 254

Average Degree: 3.299
Diameter: 5
Modularity: Varies

Weakly Connected Components: 1
Strongly Connected Components: 77

Network Visualizations

Approaches

1. **NetworkX Library Function (`girvan_newman()`)**

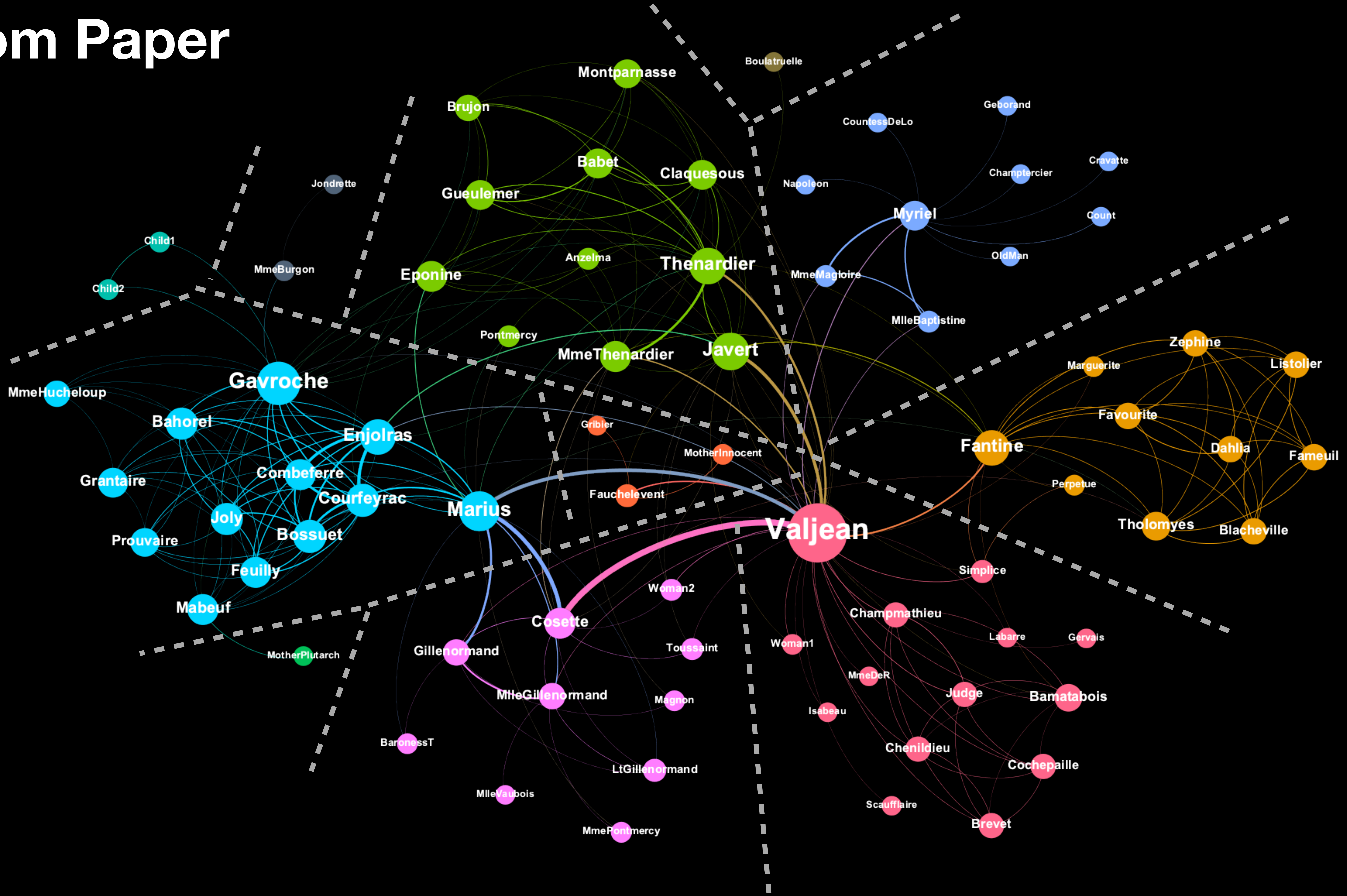
2. **Gephi**

3. **My Own Implementation**

Network Visualizations

Visualization From Paper

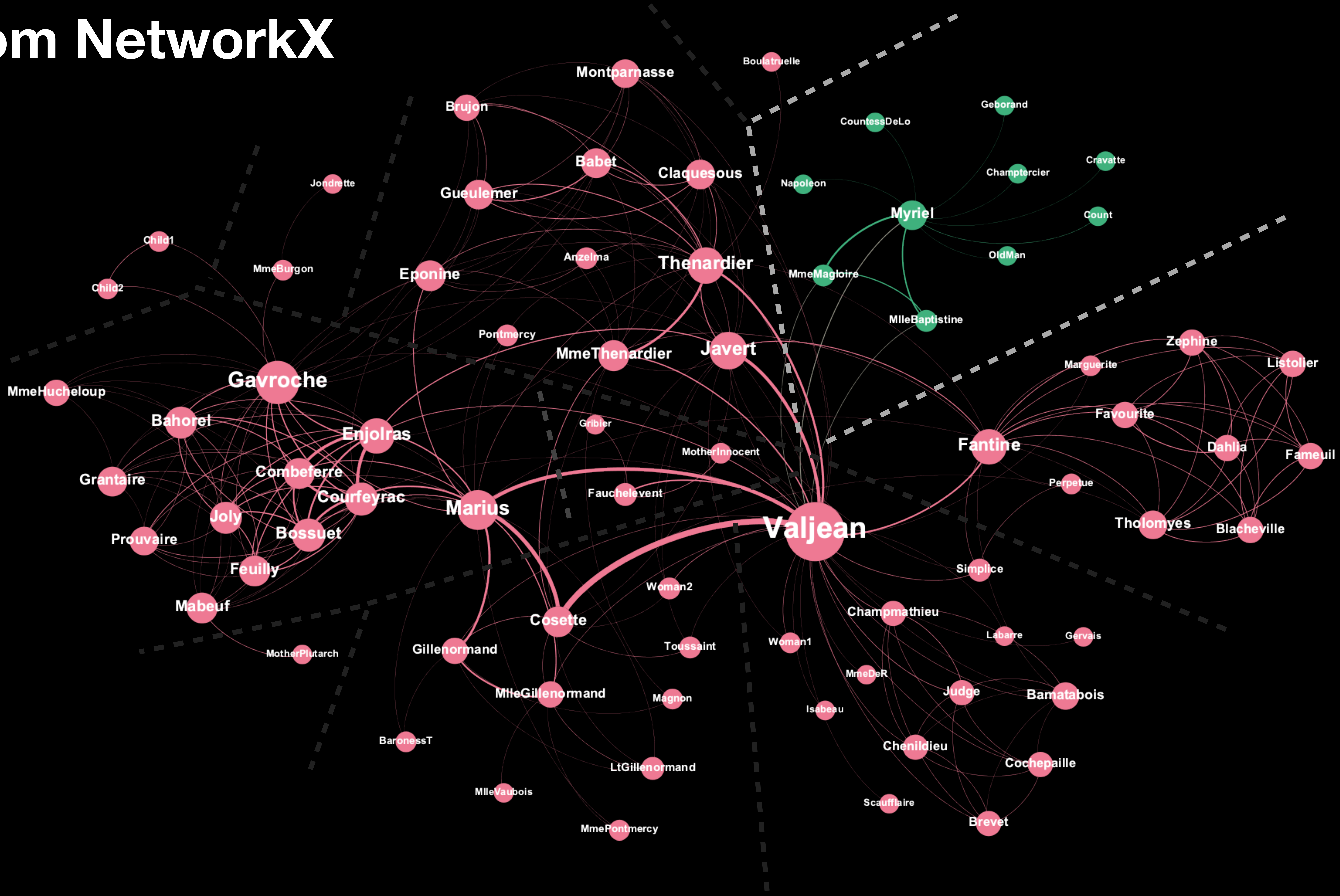
- **11 Communities**
- **Modularity: 0.54**



Network Visualizations

Visualization From NetworkX

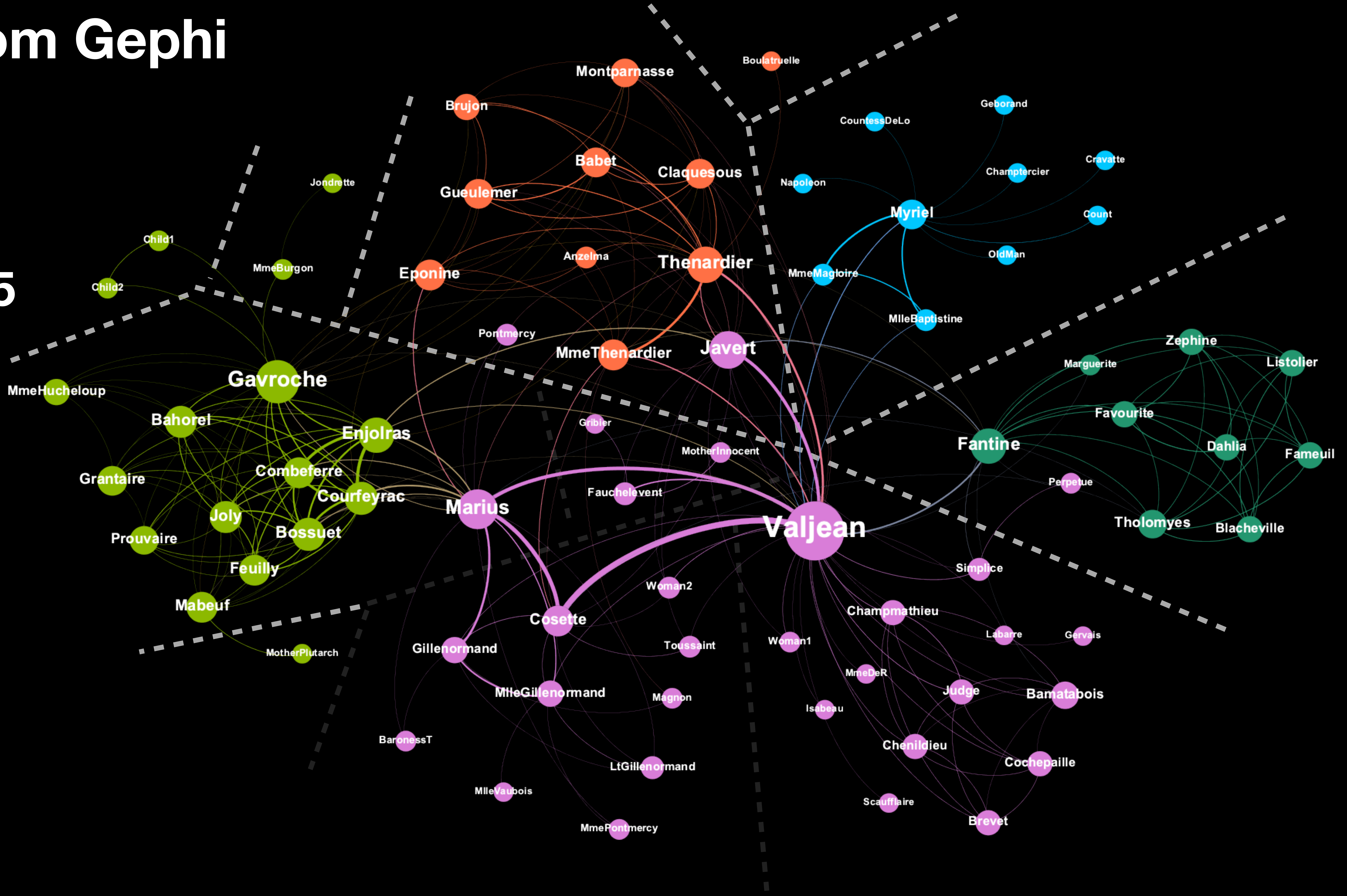
- 2 Communities



Network Visualizations

Visualization From Gephi

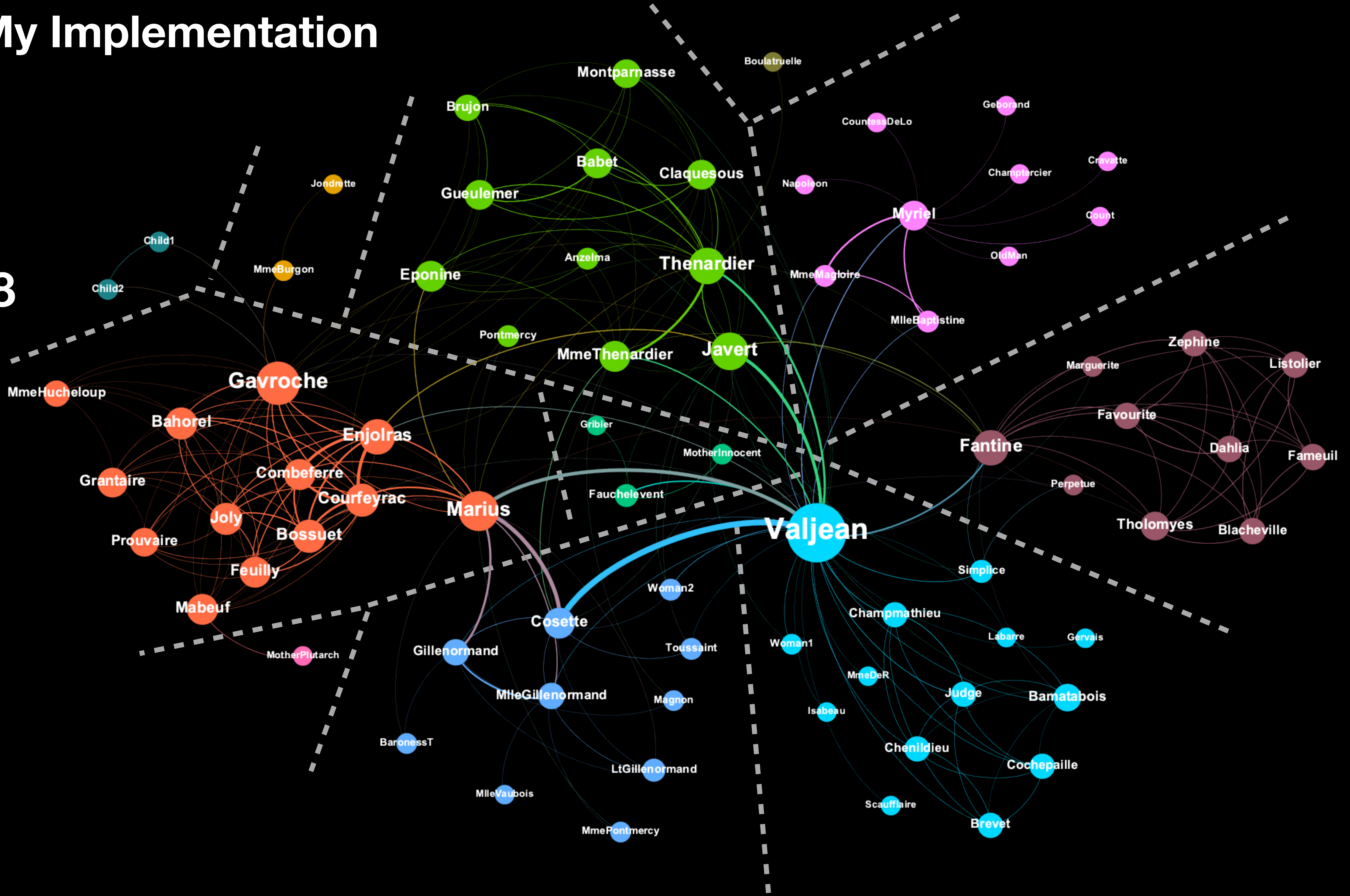
- 5 Communities
- Modularity: 0.565



Network Visualizations

Visualization From My Implementation

- 11 Communities
- Modularity: 0.538



Codes

Edge Removal

```
def removeEdges(G):
    connected_comp_init = nx.number_connected_components(G)
    connected_comp = connected_comp_init
    while connected_comp <= connected_comp_init:
        betweenness = nx.edge_betweenness_centrality(G)
        max_betweenness = max(betweenness.values())
        for k, v in betweenness.items():
            if float(v) == max_betweenness:
                G.remove_edge(k[0],k[1])
        connected_comp = nx.number_connected_components(G)
```

Codes

Get Modularity

```
def getModularity(G, degrees):
    A_update = nx.adjacency_matrix(G)
    degrees_update = {node:val for (node, val) in G.degree()}
    connected_comp = nx.connected_components(G)

    modularity = 0.0
    for component in connected_comp:
        edges, random_edges = 0.0, 0.0
        for i in component:
            edges += degrees_update[ i ]
            random_edges += degrees[ i ]
        modularity += (edges - (random_edges**2)/(2*m))
    modularity /= float(2*m)
    return modularity
```

Codes

Driver Function

```
def GirvanNewman(G, degrees):
    modularity, max_modularity = 0.0, 0.0
    component = [ ]
    while True:
        removeEdges(G)
        modularity = getModularity(G, degrees)
        if modularity > max_modularity:
            max_modularity = modularity
            component = list(nx.connected_components(G))
        if G.number_of_edges() == 0:
            break
    return component
```

Thanks