Analysing the performance of Flow-Capacity-Based Preprocessing Methods for Commmunity Detection Algorithms

BEP Draft

Felipe F.B Cintra

sn. 1605399

March 20th 2024

Contents

0.1 Abstract

Loren Ipsun

Chapter 1

1.1 Introduction

The internet, social media, academic citations, and roads, are all examples of networks that appear in our daily lives. In light of their big impact, it is only natural that we study them. An important field, in the study of networks, is community detection[1]. The applications of the area include finding metabolic communities, that help with the understanding of protein-protein interactions [2], and communities of scientific research, allowing for new possibilities of collaboration[3] .
Formally, community detection is concerned with finding sets/clusters of nodes that are more similar and/or connected within, as opposed to the rest of the network[4]. In undirected and non-valued networks, an ideal community will be a dense set of nodes with high intra-homogeneity and a high inter-heterogeneity. However, in directed and valued networks, the idea of a community is not as clear, and many times they can be context-dependent. Barroso et al. [5] found at least four different, non-formal, concepts about what can be considered communities in those types of networks. These include density-based communities, that follow the traditional definition. Random-walk communities, where each group is formed by nodes that are more likely to remain connected in a random walk. Co-citation communities, in which a group of nodes is identified by their followers, even if these aren't connected. Flow networks, where a community is based on the amount of information that can be moved within it, this concept is somewhat related to density-based clustering, but it places more importance on the structure and location of the edges, which can impact massively the perception of the community. We will be focusing on this last interpretation, as it has a considerable number of applications and literature related to it.
To apply this, flow-based, interpretation, we have either to create new algorithms, that find communities based on strictly flow criteria or to adapt existing ones to incorporate flow in their approach. This second philosophy, is more appealing, as it builds upon the existing body of work and takes advantage of already performing and well-researched algorithms and methods. In light of this, Guttierez et al. [6] proposed a prepossessing method, that utilizes the flow-capacity of a network, to augment the input of already existing community detection algorithms, ideally leading to more realistic communities in directed networks.
The article proposing this new pre-processing method provides a solid theoretical, and mathematical underpinning for its performance, but does not provide experimental results/ implementation relating to its findings. Our goal with this work, is to implement and benchmark this method, testing to see if this approach is suitable for practical applications. With this goal in mind, we formulate the following research questions to guide our work:

- How does the flow-capacity pre-processing method behave in regarding changes in its parameters?
- Is the flow-capacity pre-processing method suitable for practical applications, as it pertains to gains in performance vs computational costs?

To answer these questions, the rest of this work is split into the following chapters: Related Work where we go over the existing literature as it relates to our current work. Methodology where

we explore the relevant methods, and algorithms we will be using throughout our experiments. Experimental Setup, where we outline the experiments that will be performed, and the rationale behind them. Results, where we go over the results obtained by the experiments. Conclusion where we will be answering our research questions, and evaluate the process as to what we could have done differently.

1.2 Related Work

In this section, we will be going over the literature that relates to, more generally, community detection algorithms, then we will briefly discuss the evaluation of community detection methods, and, finally, flow capacity-related community detection.

1.2.1 Community Detection Algorithms and Evaluation

The body of work relating to community detection methods is extensive, many methods have been proposed to solve the problem, each with its philosophy, upsides, and drawbacks. [**?**] compiles 11 methods that have seen considerable adoption in the literature. These include, but are not limited to, the Girvan-Newman algorithm [7], which performs hierarchical partitions of the network, by removing edges based on the betweenes of the pairs of nodes. The Louvain Algorithm [8], that places nodes into communities that maximize the overall modularity of the network. The Leiden Algorithm [9], is an iteration of the aforementioned Louvain algorithm, and it addresses a flaw in the detection of disconnected communities. The other approaches cited include clustering by diffusion processes and by optimality of compression. In this work we will be focusing on, the Leiden, Louvain, GN algorithms, as they provide a good balance of performance, computational efficiency, interpretability and, simplicity, which allows us to better understand the performance of the pre-processing method, without the need to study extensively interaction effects between the pre-processing and the specificities of a particular algorithm. The article uses, for the comparisons between algorithms the Lancichinetti–Fortunato–Radicchi(LFR) benchmark.
The LFR benchmark was proposed in [10], it is a tool to generate random graphs, where each node is assigned to a community. This method of generating synthetic networks with built-in ground-truth communities have seen wide adoption for the testing and benchmarking of community detection algorithms[11].

1.2.2 Flow, Networks and, Community Detection

The idea of combining the concepts of flow and networks is not a new one. In fact some of the most important algorithms, such as Dijkstra‘s, take this approach. Other examples include assignment and optimization problems [**?**]. In terms of network analysis, this idea has also been somewhat studied, for example, Borgatti, [12] describes how centrality measures such as modularity and eigenvector centrality, which are used by many community detection algorithms, are dependent on assumptions of how information flows between nodes, and what are the implications of these assumptions. Despite the connection in other network analysis fields, the idea of flow and community detection wasn't explored until Barroso et al. [5] defined groups under networks, using flow capacity as a fuzzy measure. In the paper, the authors also propose an extension of the Louvain algorithm that uses the flow capacity to add more information to the modularity optimization procedure. Later the same authors, use the idea of the flow-capacity measure to propose a pre-processing method, that mixes the flow information with the adjacency matrix, which is inputted into the community detection algorithm being used. The authors claim this approach generates communities that map more closely to reality [6]. And that is what we are going to be investigating in this work.

Chapter 2

2.1 Preliminaries and Theoretical Background

We have so far talked about networks, flow, communities, and, flow capacity, but we haven't defined them in the formal sense, we will define all these terms appropriately. We will also present all the algorithms that will be used for the rest of the work. The algorithms and definitions in this section will be presented in a relatively context-free manner, as their context and application will be explored whenever they become relevant to our work.

2.1.0.1 Definitions

Definition 2.1.1 (Directed Network)**.** A directed network or a *digraph* is represented by a double $G = (N, E)$, where N is a set of nodes and E is a set of edges i, j, linking the nodes i and j, with the source being node i and the target node j. A *weighted network*, is described by the triple (N, E, w), where w is a set of weights associated with each edge. Directed networks can be represented by a non-symmetric *Adjacency Matrix* A_{ij}, such that

$$A_{ij} = \begin{cases} w_{i,j} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

In the case where edges aren't weighted, replace $w_{i,j}$ by 1.

Despite there being no formal definition for a community, there are informal definitions that still prove to be useful, one is given by [4] and it follows:

Definition 2.1.2 (Community)**.** one can consider it (community) as a set of entities that are closer to each other compared to the rest of the dataset. The notion of closeness is based on a similarity measure, which is usually defined over a set of entities.

Now that we have the definition of community, we can define a metric used to determine the quality of partitions (communities) in a network.

Definition 2.1.3 (Directed Modularity Q_d[13])**.** Let $G = (V, E)$ denote a directed graph and let P denote a partition of the nodes. The directed modularity of the partition P in the directed graph G is calculated as:

$$Q_d = \frac{1}{m}\Sigma_{i,j}[A_{ik} - \frac{k_i^{in}k_j^{out}}{m}]\delta(c_i, c_j) \tag{2.2}$$

where $\delta(c_i, c_j) = 1$ if i belongs to the same community as j and $\delta(c_i, c_j) = 0$ otherwise, m denotes the number of edges, denotes the in-degree of node i (number of links coming into i) and k_j out denotes the out-degree of node j (number of links going out of j).

We have also been referring to flow. In a network flow is defined as follows:

Definition 2.1.4 (Flow, Flow Conservation Rule, s-t Flow [14])**.** Given a digraph G with capacities $u : E(G) \to \mathbb{R}_+$, a flow is a function $f : E(G) \to \mathbb{R}_+$ with $f(e) \leq u(e) \forall_{e \in E(G)}$. We say that f satisfies the flow conservation rule at vertex v if:

$$\Sigma_{e \in \delta^-(v)} f(e) = \Sigma_{e \in \delta^+(v)} f(e) \tag{2.3}$$

Now, given a network (G, u, s, t), and s-t-flow is a flow satisfying the flow conservation rule at all vertices except s and t. We define the value of a s-t-flow f by:

$$\texttt{value}(f) := \Sigma_{e \in \delta^-(v)} f(e) - \Sigma_{e \in \delta^+(v)} f(e) \tag{2.4}$$

Now that we know what flow means, in the context of a network, we can define the flow capacity measure which is used by the pre-processing algorithm. It follows:

Definition 2.1.5 (Flow Capacity Measure μ^F[6])**.** Given a directed graph $G = (V, E)$ let $(i, j) \in E$ denote a directed edge, and let $f_i j$ denote the flow among nodes $i, j \in V$. For any subset $S \in V$ the flow capacity measure μ^F is:

$$\mu^F(S) = \frac{\Sigma_{i,j \in S} f_{i,j}}{\Sigma_{i,j \in V} f_{i,j}} \tag{2.5}$$

This measure gives us information about the intensity of the flow in a certain set of nodes and a super-set containing it. But in order to be able to use the measure, we need to first find a way of passing that information along to the algorithms, so we define the flow-directed interaction index, which is the measure applied to the overall graph.

Definition 2.1.6 (Flow Directed Interaction Index[6])**.** Given the directed graph $G = (V, E)$ whose flow function is f_i, j and being μ^F it's flow capacity measure, given a directed edge$(i, j) \in E$, the flow directed interaction index is defined as:

$$I^D_{i,j} = \frac{f_{i,j}}{\Sigma_{k,l \in V} f_{kl}}$$

Considering every pair $i, j \in V$ from values I^D_{ij} we obtain the matrix $(I^D_{ij})_{i,j \in V}$ which summarizes μ^F

The following definitions concern the evaluation of the partitions obtained by community detection algorithms.

Definition 2.1.7 (κ index [11])**.** For a classification problem, if $P(y = \hat{y}$ is the probability a predicted label $\hat{y}$ matches a label y, we can say for a accuracy measure $A(y_i, \hat{y}_i$ and a expected accuracy $\mathbb{E}_a = \Sigma^n_{i=1} P(y_i = \hat{y}_i) A(y_i, \hat{y}_i)$,

$$\kappa = \frac{\Sigma^n_{i=1} A(y_i, \hat{y}_i) - \mathbb{E}_a}{1 - \mathbb{E}_a} \tag{2.6}$$

2.1.0.2 Algorithms

In this subsection, we will be introducing the algorithms we will be using throughout our work.

2.1.0.2.1 Flow-Capacity Preprocessing (FCP)

The flow capacity pre-processing method, is a supervised method, meaning it follows the machine learning paradigm that uses data with known outcomes. This data is split into a training set and a test set, the training set is used by the method to "learn" the optimal parameters for the model. The performance of this new, learned, parameter is then tested in the test set. The best parameter values are then kept and, the model with the fitted parameters is then used for desired applications [15].

Algorithm 1 Flow Capacity Preprocessing (FCP)

Input $A = [A_1, A_2...]$: Array of adjacency matrices of training set, I_{ij}: set of flow interaction indexes corresponding to each matrix of A, `ALG`: community detection algorithm to be pre-processed, Q_d quality function to be maximized N number of maximum iterations, `find_next`: function to find the next α

Output α: optimal mixing parameter between the I_{ij} and A

```
1:  procedure FLOW CAPACITY ALG
2:      n ← 0
3:      maxQ ← −∞
4:      prev ← −∞
5:      alpha ← 0
6:      while α ≠ prev and n ≤ N do
7:          F ← αA + (1 − α)I_ij
8:          out_1, out_2, ... ← run ALG(F_1), ALG(F_2)...
9:          newQ ← average(Q(out_1, out_2, ...)
10:         if newQ ≥ maxQ then
11:             prev ← α
12:             maxQ ← newQ
13:             α ← find_next(α)
14:         else
15:             α ← find_next(α)
        return α
```

The algorithm presented here is a slight modification of the one originally proposed by Gutiérrez et al. [6], as here we present the method with a choice for a function to find the next α, and a choice of the quality function Q_d, while in the original work, α was found through a linear search with fixed step size, and the quality function Q_d to be maximized was fixed as modularity. We've chosen to present the algorithm in its modified version, as users might want to maximize quality functions other than modularity, and might want to utilize more efficient ways of finding α other than linear search.

2.1.0.2.2 FCP and running time

Given that the FCP is a supervised method, there are two running times that concern us when applying it, the training and the utilization running time. The training running time is a function of the maximum set number of iterations N, running time of quality function r, length of the training array l, and the maximum time complexity of the algorithm chosen for the input array $\max \mathcal{O}(A)$. Such that the upper bound can be given by $\mathcal{O}_{FCP}(A, N, r, l) = \mathcal{O}_{\texttt{ALG}}(A)Nrl$. This means that the choice of algorithm, training set, and quality function are all important, to prevent the training running time from becoming unreasonably large. On the other hand, the pre-processing does not change the time complexity of the algorithm, so once a ideal α is known for a particular application, there's no extra cost associated with using the FCP version.

2.1.0.3 Louvain Algorithm [8]

The Louvain Algorithm is a fast and popular community detection algorithm [**?**]. For a network $G = (V, E)$, The algorithm works in two phases. In phase 1 the algorithm assigns each vertex a community. Nodes are then stochastically moved to neighboring communities. If modularity increases after this move, the node is kept in the new community, otherwise, it is returned to the original community. In phase 2 the new-found communities are aggregated into super nodes. Phases 1 and 2 are then repeated until modularity cannot be increased any further.

Algorithm 2 Louvain Algorithm

Input $A \leftarrow G = (V, E)$
Output P

procedure PHASE 1
  $n \leftarrow len(V)$
  $u \leftarrow [\{\}_1, ...\{\}_n]$
  for $i \in 1, 2...., n$ **do**
    $u[i] \leftarrow V[i]$
  $o \leftarrow permutation(V)$
  while modularity optimization process is not done **do**
    find out-edges j of node $o[i]$
    $j^* \leftarrow \{l|\Delta Q_i^d(j^*) = \max_{l \in \{1,..,h\}}\{\Delta Q_i^d(l)\}$
    if $\Delta Q_i^d(j^*) > 0$ **then**
      move node i to community $j*$
    else
      node i stays in its community
procedure PHASE 2
  Aggregate communities into super-nodes by summing intra-community and inter-community edge weights
return P (set of communities)

2.1.0.4 Leiden Algorithm[9]

The Leiden algorithm is a refinement of the Louvain algorithm, proposed by Traag et al., it seeks to correct the limitation of the Louvain algorithm, which in some cases finds badly connected/disconnected communities. The algorithm achieves that goal by the introduction of auxiliary functions, *Move Nodes Fast(G, P)* and *Refine Partition(G, P). Move Nodes Fast(G,P)*, starts from a given partition P and moves nodes in trying to increase modularity. *Refine Partition(G, P)* refines the partitions obtained, reducing the amount of badly connected communities. *Move Nodes Fast* and *Refined Partitions* are relatively complex functions, and going in-depth into them is outside the scope of this work.

Algorithm 3 Leiden Algorithm

Input $A \leftarrow G = (V, E)$
Output P

procedure LEIDEN
  $P \leftarrow MoveNodesFast(G, P)$
  done $\leftarrow$ mod $P =$ mod V
  if not done **then**
    $P_{refined} \leftarrow RefinePartition(G, P)$
    $G_F^* \leftarrow$ aggregate the graph $G$by grouping nodes into the communities in $P_{refined}$
  while not done **do**
    $returnflat^*(P)$

2.1.0.5 Girven-Newman Algorithm [7]

The Girven-Newman algorithm takes a different approach than the Leiden and Louvain Algorithms. The Girven-Newman algorithm operates by removing edges, using a centrality criterion, usually, SP betweenness centrality [11]. As the edges are removed, the community structure is revealed.

Algorithm 4 Girven-Newman Algorithm

Input $A \leftarrow G = (V, E)$
Output P

1: **procedure** GIRVAN-NEWMAN
2: $G_F = (V, E)$
3: $max_B \leftarrow 0$
4: $max_B\_edge \leftarrow (i, j) \in E$
5: **while** number of edges in G_F is 0 **do**
6: $m \leftarrow$ mod E
7: **for** $l = 0$ to $m - 1$ **do**
8: l denotes an edge (i, j)
9: calculate betweeness cetrality of (i, j) $w^B_{(i,j)}$ in G_F
10: **if** $w^B_{(i,j)} > max_B$ **then**
11: $max_B \leftarrow w^B_{(i,j)}$
12: $max_B\_edge \leftarrow (i, j)$

2.1.0.6 LFR Benchmarks Algorithm [10]

The LFR benchmark algorithm proposed by Lancichinetti, Fortunato, and Radicchi[10], is an algorithm that generates random networks with ground-truth communities. These graphs are unweighted and undirected. A network created by the LFR algorithm is generated using a power-law distribution, nodes are allocated to communities iteratively, following a set of constraints imposed by the parameters set by the user. The parameters for the algorithm are as follows: `n`: indicating the size of the network, τ_1: 1st exponent of the distribution, τ_2 second exponent of the distribution, μ: indicating mean community size, `average_degree`: the average degree of a node, optional parameters that can be set are `min_degree`: minimum degree of a node in the network, `max_degree`: maximum degree of a node, `min_community`: minimum community size, `max_community`: maximum size of a community. We will not be using these optional parameters unless explicitly described.

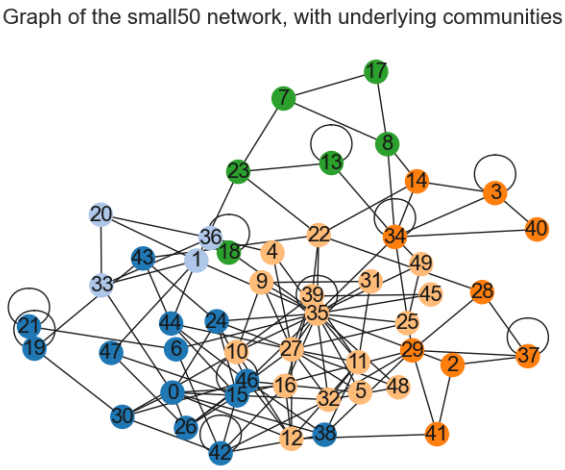

Figure 2.1: Depiction of the small50 network and ground communities

2.2 Methodology

Since there are many parts to the FCP method, various aspects might impact its performance. In this work, we will experiment with the following: Choice in the training set, choice of algorithm, choice in quality function, Network application, and method of mixing. With multiple features of interest, we have to create a consistent and invariant testing setup. This setup, should, ideally, allow us to investigate how feature changes impact the performance outcomes, free of any externalities.

2.2.0.1 Synthetic Networks

To account for the impact of network size, density, number of communities, etc. We will test the algorithms and variations in synthetic and real networks of different sizes. Thus we have generated 3 synthetic networks, using the LFR benchmark. We will refer to these as "small50", "medium250" and "large2500", with approximately 50, 250 and 2500 nodes respectively. The choice in sizes was motivated, partly due to limitations in the LFR algorithm, which may not be able to generate networks that are too small [10]. And in line with the literature, regarding evaluating community detection algorithms [**?**].

As for the other parameters of the LFR benchmark, μ (mixing parameter) was chosen in the range of (0.25,0.35). This choice was motivated by our need for communities that are difficult enough to detect so that differences in the performance of the algorithms are evident. But not too obfuscated, otherwise, we lose again the ability to differentiate performance reliably. For the other parameters, we will be a mix of the parameters outlined in [11], and parameters that allow us to consistently generate networks, with the networkx implementation of the algorithm. The overview of the parametrization can be found in the table [**??**]. The network small50 can be visualized in fig 2.1

2.2.0.2 Flow Induction

The LFR benchmark generates, by design, undirected, unweighted graphs. That do not inherently suit our purpose to evaluate the performance of the FCP, on weighted, directed graphs. To circumvent this limitation, we will be inducing a flow in the networks using the max-flow algorithm. We chose this approach, instead of randomly assigning weights given a given probability distribution because we want to preserve the intuition, that the weights represent the flow of information inside the network. Given that we are interested in a flow, we have to assign weights that follow

Parameter	Value
τ_1	3
τ_2	2
average_degree(large2500)	8
average_degree(medium250)	6
average_degree(small25)	4

Table 2.1

network/ value parameter	small50	medium250	large2500
# nodes	50	250	2500
# edges	130	754	8143
avg. community size	10.0	22.7	69.4
var community size	23.6	128.9	4653.5
avg. deg.	5.2	6.0	6.5
min deg.	2.0	2.0	4.0
max deg.	24	53	346
avg. btw. centrality	31×10^{-2}	31×10^{-3}	1×10^{-3}

Table 2.2

Table 2.3: Parametrization and key metrics of the generated networks

the flow conservation rule, given in definition 2.1.4. So, for a network, $G = (V, E)$ and an edge e_{ij}, where the nodes i, j are determined as the source and sink nodes respectively, the weight of the edge w_{ij} will be given by

$$w_{i,j} = \texttt{maxflow}(i, j) \tag{2.7}$$

we will be calling the weighted adjacency matrix,

$$F_{ij} = \begin{bmatrix} w_{1,1} & w_{1,2} & ... & w_{1,n} \\ . & . & & . \\ . & & . & . \\ w_{n,1} & . & . & w_{n,n} \end{bmatrix} \tag{2.8}$$

with the induced flow information, $F_{i,j}$ for the rest of this work.

We've applied this process to the following toy example: we first generate a network G using the LFR benchmark. In our case, the adjacency matrix for $A^G_{i,j}$ is given by 2.9, and by inducing the flow we get the weighted matrix $F^G_{i,j}$

$$A^G_{i,j} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} F^G_{i,j} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 2 & 2 & 3 \\ 0 & 0 & 1 & 0 & 2 & 1 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.9}$$

Note that the flow between nodes is not the same as a weighted adjacency matrix, since there can be flow between nodes that aren't connected by an edge. This is only natural, given that many networks are a way of facilitating the move of resources, without strict direct connection requirements. From this flow matrix $F^G_{i,j}$, we can then calculate the flow interaction index (2.1.6) I^d_{ij} which will be used as input for the FCP.

We can see in 2.2, that we manage to capture the dynamics of how information flows in the network. We can see that nodes that are more central to their communities, transfer more flow, as expected.

2.2.0.3 Real World Data

To verify that the FCP method applies to real-world data, and to minimize the impact of possible limitations of our flow-induction approach. We will also be testing the method in two real-world

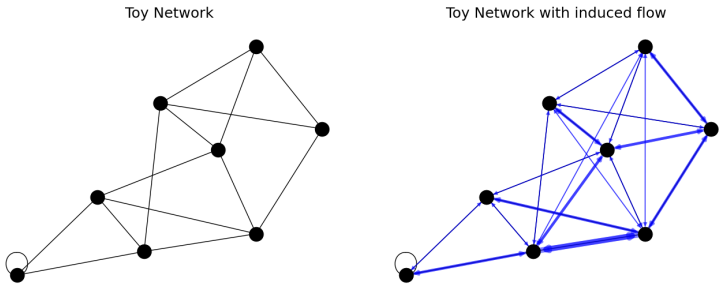

Figure 2.2: network without and with flow information

datasets. The datasets chosen were the EU-core and the CISCO22 datasets, together these networks should provide insights on how the performance of the FCP translates to large and small datasets.

2.2.0.3.1 EU-Core Dataset

The EU-Core dataset [16], is a dataset containing the anonymized email information of an undisclosed large European research institute. It is openly available through SNAP (Stanford Network Analysis Project) [17]. Each node represents a member of the institute, and each edge represents emails sent between members. The dataset also contains information about which department, of the institution, each node belongs to. These departments can be perceived as the ground-truth communities of the network. The dataset's key metrics are described in table [2.6]

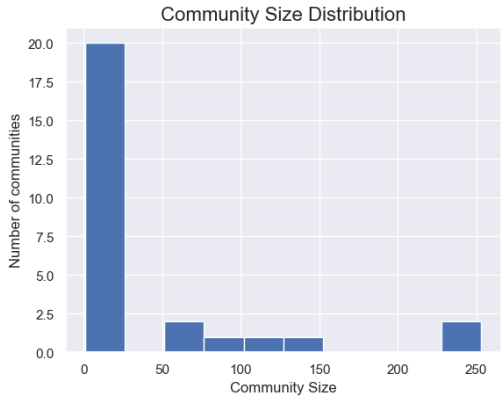

Figure 2.3: Size Distribution of Communities

Metric	Value
Nodes	1005
Edges	25571
Number of Communities	42
Mean Community Size	37.1851
std Community Size	62.2544
max Community Size	183
min Community Size	1
Average clustering coefficient	0.3994
Diameter (longest shortest path)	7
Average Betweeness Centrality	0.001522

Table 2.4: EU-Core dataset metrics

We can see that the dataset has communities of varying sizes, and ideally, we want an algorithm that can detect, with enough resolution the larger and smaller communities.

2.2.0.3.2 Cisco 22-Network

The other real-world dataset we are using is one of the 22 networks provided by Modani [18] The graphs represent the communications among hosts running various distributed applications. An edge in any graph corresponds to a connection from a client node to a server node. The edges also provide the traffic of information between the nodes. The ground communities in the graph we will be using represent the steps in the author's workload. he workload includes flow data processing pipelines, various graph algorithms as well as other tasks. Thus the sensors and the system collect, process, and report their activity. The grouping is based on the node function.

Metric	Value
Nodes	52
Edges	628
Number of Communities	21
Mean Community Size	2.75
std Community Size	1.442
max Community Size	7
min Community Size	1
Average clustering coefficient	-
Diameter (longest shortest path)	6

Table 2.5: Key Metrics for the CISCO22 network

Data node, processing node, and so on. The statistics describing the dataset can be found in table 2.5

2.2.1 FCP Fitting and application

Now that we have described the data being used, we will explore how we will be applying the FCP 1. The diagram 2.4 describes the process.

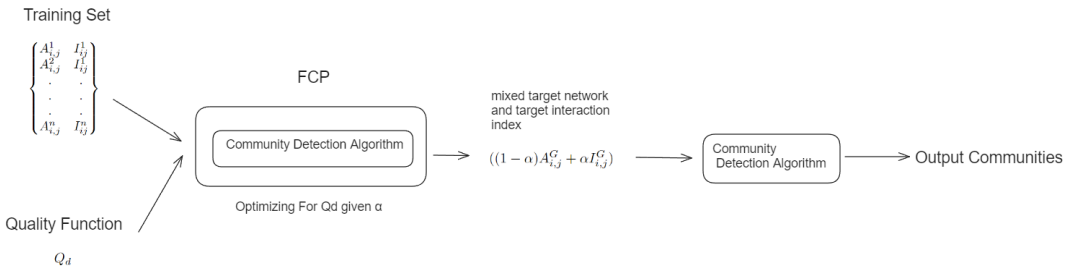

Figure 2.4: Process of the FCP

The first step is to find a suitable training set and quality function to fit the method. The training set can be the target network itself, but in case of technical limitations, such as a very large network, where the iterated application of community detection methods is too costly, one can train the preprocessing method in synthetic networks that share characteristics (i.e centrality measures, average degree etc.) with the target network. We will be evaluating how the choice of training set impacts the overall performance of the method.

Once the training set has been chosen, one must choose a quality function, to be used as an optimization objective for the preprocessing method. But, before we discuss the choice quality functions, we will, briefly, go over the evaluation metrics of community detection algorithms.

2.2.1.1 Evaluation of Community Detection Algorithms

An important part of the community detection problem is the evaluation of obtained communities. Given that, in applications, communities aren't known beforehand, many quality measures for the partitions have been proposed, most prominently, but not without criticism, modularity has been the standard [8], with definition 2.1.3. But, if we have more information, namely the underlying ground communities of the network, we can evaluate the method's performance with a higher level of detail. One of the most popular metrics for community detection algorithms has been

normalized mutual information NMI:**??**,[19]. It compares the closeness of the partitions obtained, to the ground-truth communities. Despite its wide adoption throughout the literature, NMI has been shown to display biased behavior [19] and suffer from the finite-size network effect. In light of that In light of that, Zhang et al. [11], proposed a machine-learning inspired approach, that uses the κ **??** index. This approach first translates the labels of the communities obtained to the ground-truth labels. To do that we first create a cost mapping, from a set of labels $A = A_1, A_2, ..., A_{c'}$ to a set of labels $B = B_1, B_2, ...B_c$, st.

$$l_{i,j} = |B_i \cup A_i| - |B_i \cap A_j| \tag{2.10}$$

We use this mapping to formulate a cost matrix $L = (l'_{i,j})_{c',c'}$ such that:

$$l'_{ij} = \begin{cases} l_i j, & \text{if } i \leq c \\ 0, & \text{if } i > c \end{cases} \tag{2.11}$$

We use L to formulate a standard assignment problem, which can be solved using the Munkres algorithm [**?**]. After translating the labels, we can then apply the κ to compare the algorithm's performance, to the random assignment of labels. The κ index, does not suffer from the same bias as NMI.

Since κ and NMI, are metrics of the quality of the results obtained by the community detection algorithms, we can test using them as quality functions to be optimized, whenever we have access to the ground communities in our training set. If we have no access to the real-partitions, in our training set, we will be using modularity as a function to be optimized, as it is the standard in the literature.

2.2.1.2 Fitting and Algorithm Selection

Now that we have discussed the choice of datasets and quality functions, we will discuss the fitting process and the algorithms we will preprocess and evaluate. We will first explore the algorithms, as the fitting process depends on the choice of method used.

2.2.1.2.1 Algorithms

Given that the FCP is a preprocessing method, it is only natural that we apply it to a variety of community detection methods to evaluate it. In this work we will be applying it to 3 methods, the Louvain method [8]2, the Leiden method [9] 4 and the Girven-Newman method[7] 4. We chose these methods because they are fast, consistent, and well-understood methods [**?**]. Enabling us to conduct more experiments, to get a deeper understanding of the FCP, the object of our study. Another property that lends itself to these methods, is that they are built on optimizing a metric. Louvain and Leiden optimize the partitions based on modularity, and the standard Girven-Newman partitions the partitions given the between-centrality of the edges. These approaches lead to predictable behavior and lend themselves well to the fitting process of the FCP.

2.2.1.2.2 Fitting

The goal of the fitting process of the FCP (lines 6-15 in 1), is to find the ideal α for a given algorithm and dataset. Thus, it is natural that for different types of algorithms, the process varies slightly. For methods that yield stochastic partitions, in this work the Louvain and Leiden methods, we need to take the uncertainty into account. Meaning, that to increase our certainty that the value of α is indeed what we expect, we need to run the algorithm a predetermined n times. Then calculate the the average of the quality of the partitions obtained. We compare the averages obtained, to determine the best *alpha*.

To determine an appropriate n, we first examined the distribution of the modularity obtained by running the Louvain algorithm in the medium250 network. We can see the distribution obtained in 2.5

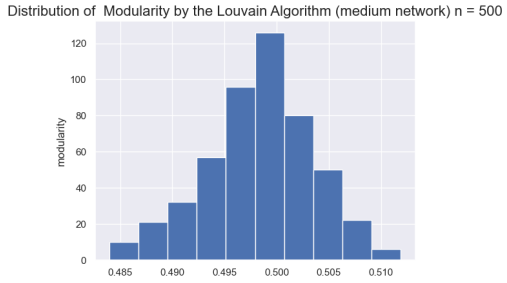

Figure 2.5: Distribution of Modularity of Louvain Algorithm on medium250 (500 runs)

Metric	Value
μ	0.4983
σ	4401×10^{-6}

Table 2.6: Mean and variance of the Modularity (500 runs)

To orient ourselves, we set an arbitrary, but reasonable, goal that with 95% of confidence, the mean modularity of the trials would be within 1% of the absolute value of the modularity distribution. Using the central limit theorem, we arrive at the number of trials $n = 15$. This approach allows us to set some guarantees that the modularity, as a quality function, obtained by a choice of α, is representative of the underlying impact of the parameter. Based on these results, we will be fitting the FCP for Leidein and the Louvain algorithms, with the average value of the quality function for 15 runs for each α.

For deterministic algorithms, the process is simpler. For each α We have only to perform 1 run of the algorithm, and use the value of the quality function on the partitions obtained, to orient our choice of α.

2.2.2 Application & Evaluation

After the FCP is fitted on a specific algorithm $\mathtt{A}(G)$, we apply $\mathtt{A}^{FCP}(G)$ to the target network. To evaluate the quality of the partitions obtained, we will be using the same quality measure functions, that we used to fit the algorithm α, namely modularity, NMI, and κ. For stochastic algorithms, we are also going to perform repeated experiments and use the average to indicate our results.

2.3 Experimental Setup

In this section, we will be detailing the experiments that were performed, we will also give a detailed description of the tools used for the implementation.

2.3.1 Baseline

To evaluate the performance of the FCP, we first benchmarked the performance of the "vanilla" Leiden, Louvain, and GN algorithms. We applied the methods to our synthetic and real-world networks and recorded the results. Given that, the Leiden and Louvain algorithms have a stochastic nature, the results presented will be averages over a number of 10 runs.

2.3.2 Same Network, Modularity

The first set of experiments performed, were the closest to what was described in [6]. For each of our test benchmark networks, we fitted the FCP in the same network that would be the target. For F_{ij} we used a linear combination of A and $I_{i,j}$, $F_{ij} = \alpha A + (1-\alpha)I_{ij}$, and we used modularity Q_d as a quality function to optimize α. These experiments were performed so we could evaluate, in its most basic form, the performance of the FCP. We can use this set of experiments to calculate

the treatment effects of introducing other configurations, by comparing the outcomes of this set results with the subsequent ones and to the baseline results.

2.3.3 Same Network, κ/NMI

The second set of experiments performed, deals with the choice of quality function, as an optimization objective. Different evaluation metrics capture different information about how the partitions reflect reality. Since we don't know which metric will provide better data, to inform the choice of α, we decided to experiment with the metrics we are using, namely modularity, κ, and NMI to understand how they impact FCP performance.

2.3.4 Multiple Training Sets, Modularity

The last set of experiments performed had the goal to test the practical application of the use of different training sets other than the target network. In this set of experiments, we apply the technique to two different applications,

- the target network is too big for timely fitting of α
- the target network has no known ground-community

In both scenarios a training set of synthetic networks is constructed, and we parametrize these networks, intending to preserve some characteristics of the target network, such as average degree.

2.3.5 Tooling

The project has been fully implemented in Python 3.12. The auxiliary libraries used were, Numpy [20], for computation, especially with matrices, scikit-learn [21] used in obtaining metrics. We also used extensively NetworkX, [22], for managing and plotting networks. The library was also used to create the LFR benchmark network graphs, apply the baseline Louvain and Girven-Newman algorithms, and calculate the `maxflow` in between nodes. For the Leiden algorithm, we used a combination of the Igraph [23] and the Leidenalg packages[24]. Finally, we used the Munkres[25] package, to solve the assignment problem described in 2.11.

Inducing flow, remapping communities, calculating the κ index, calculating both the flow capacity metric and the flow interaction index, and the flow capacity pre-processing method for the Girven-Newman, Leiden, and Louvain algorithms are all new contributions and have been implemented by the authors. All the code for this project is available in the GitHub repository: https://github.com/felipefbcintra/Bachelor-End-Project

Chapter 3

3.1 Results

In this section, we will detail the results obtained from out experiments.

3.1.1 Baseline

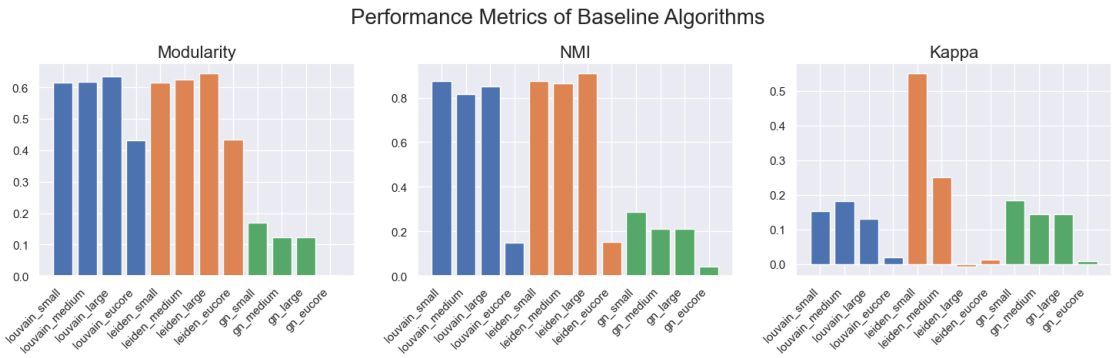

Figure 3.1: Baseline Metrics

3.2 Conclusion and Evaluation

Bibliography

[1] W. Hu, "Finding statistically significant communities in networks with weighted label propagation," *Social Networking*, vol. 02, pp. 138–146, 01 2013.

[2] G. O. Consortium, "The gene ontology project in 2008," *Nucleic acids research*, vol. 36, no. suppl_1, pp. D440–D444, 2008.

[3] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, p. 7821–7826, June 2002.

[4] F. D. Malliaros and M. Vazirgiannis, "Clustering and community detection in directed networks: A survey," *Physics Reports*, vol. 533, p. 95–142, Dec. 2013.

[5] M. Barroso, I. Gutiérrez García-Pardo, D. Gomez, J. Castro, and R. Espínola, *Group Definition Based on Flow in Community Detection*, pp. 524–538. 06 2020.

[6] I. Gutiérrez García-Pardo, M. Barroso, D. Gomez, and J. Castro, "Social networks analysis and fuzzy measures: a general approach to improve community detection in directed graphs," in *The 20th World Congress of the International Fuzzy Systems Association (IFSA 2023)*, (EXCO, Daegu, Korea), 08 2023.

[7] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, pp. 113–126, Feb 2004.

[8] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. P10008, Oct. 2008.

[9] V. A. Traag, L. Waltman, and N. J. Van Eck, "From louvain to leiden: guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, p. 5233, 2019.

[10] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, Oct. 2008.

[11] X. Liu, H.-M. Cheng, and Z.-Y. Zhang, "Evaluation of community detection methods," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 9, pp. 1736–1746, 2020.

[12] S. P. Borgatti, "Centrality and network flow," *Social Networks*, vol. 27, no. 1, pp. 55–71, 2005.

[13] E. A. Leicht and M. E. J. Newman, "Community structure in directed networks," *Physical Review Letters*, vol. 100, Mar. 2008.

[14] B. Korte and J. Vygen, *Network Flows*, pp. 153–184. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[15] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. 2018. Second Edition.

[16] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, p. 2–es, mar 2007.

[17] Leskovec, J, ""snap:stanford network analysis project"." https://snap.stanford.edu/data/email-EuAll.html, 2024. Accessed: 2024-11-06.

[18] O. Madani, S. A. Averineni, and S. Gandham, "A dataset of networks of computing hosts," in *Proceedings of the 2022 ACM on International Workshop on Security and Privacy Analytics*, pp. 100–104, 2022.

[19] P. Zhang, "Evaluating accuracy of community detection using the relative normalized mutual information," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2015, no. 11, p. P11006, 2015.

[20] Numpy Team, ""numpy: The fundamental package for scientific computing with python"." https://numpy.org/, 2024. Accessed: 2024-11-06.

[21] scikit-learn developers, ""scikit-learn: Machine learning in python"." https://scikit-learn.org/stable/index.html, 2024. Accessed: 2024-11-06.

[22] NetworkX developers, ""networkx: Network analysis in python"." https://networkx.org/, 2024. Accessed: 2024-11-06.

[23] igraph: the network analysis package, ""scikit-learn: Machine learning in python"." https://igraph.org/, 2024. Accessed: 2024-11-06.

[24] Traag V., ""leidenalg"." https://pypi.org/project/leidenalg/, 2024. Accessed: 2024-11-06.

[25] Clapper B., ""munkres1.1.4"." https://scikit-learn.org/stable/index.html, 2024. Accessed: 2024-11-06.