Modularity-Driven Hypergraph Clustering via Flow-Based Cuts

Leonie Beer

May 23, 2025

4204583

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisors:
Henrik Reinstädtler
Adil Chhabra

Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Schulz for providing the opportunity to write this thesis under his supervision, for the valuable guidance throughout the project, and for sparking my enthusiasm for graph theory in his lectures. I am especially grateful to Henrik Reinstädtler and Adil Chhabra for their support, patience, and quick responses with insightful feedback during the course of my work. Deepest thanks go to my parents for their continuous support, encouragement, and for inspiring me to pursue computer science in the first place, and to my friends for their motivation and occasional distractions, which were both much appreciated.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, May 23, 2025

Leonie Beer

Abstract

Clustering data represented by hypergraphs continues to gain significance in several different and overlapping fields of research, including pattern recognition, bio-informatics, and machine learning. The hypergraph clustering problem seeks to divide the vertices of a hypergraph into densely edge-connected subsets, called clusters, without predefining their number or size. To solve this problem, we propose a two-phase algorithm with focus on modularity maximization, to which end we introduce two modularity functions, one defined on a graph and one defined on a hypergraph, supported by several proofs.

The first phase consists of a flow-based approach inspired by the concept of Natural Cuts [15], iteratively computing minimum cuts on subhypergraphs to construct an initial clustering. The second phase refines this clustering using a greedy strategy which moves vertices between neighboring clusters to improve overall modularity. Both phases are designed for parallel execution, which we present and analyze.

In our experimental evaluation, we compare numerous parameter settings, algorithm variations, and modularity functions to determine the optimal configuration of our algorithm. We further assess the efficiency and quality of our parallelization, and compare our algorithm to state-of-the-art clustering strategies, demonstrating that our algorithm consistently produces higher-quality clusterings.

Contents

CHAPTER 1

Introduction

Graph Clustering is an essential technique used for analyzing data, but more often than not, real world data engages in far more complex relations than graphs are able to represent. This has led to the growing use of hypergraphs, which allow us to model multi-way relations more effectively. However, clustering hypergraphs remains significantly more challenging, both in terms of adequacy and efficiency.

1.1 Motivation

In recent years, the use of hypergraphs has gained a growing amount of attention across a wide range of applications, from bio-informatics to social sciences. Traditionally, relational data has been modeled using graphs, relying solely on pairwise connections. However, it has become increasingly obvious that such models are mostly insufficient to capture the complexity of real-world interactions. In all contexts, from brain networks to studies of social dynamics, relations frequently occur among multiple data points simultaneously. This realization has led to an increasing amount of research over the past decade emphasizing the importance of higher-order connections. Hypergraphs, which generalize conventional graphs by allowing edges to connect more than two vertices, are ideal for modeling such multi-way relations.

A lot of research focuses on showing the importance of modeling complex systems as hypergraphs: In neuroscience, for example, representing functional brain networks through multi-vertex connections has been shown to reveal organizational principles of brain functions that cannot be found by modeling them using pairwise connections [23, 42, 45]. Other research showing the importance of hypergraphs in medical informatics include researching the effects of combining three or more drugs [56] and metabolic engineering [29]. Similarly, in social sciences, studies prove that for both humans [8] and animals [37], modeling higher-order interactions is indispensable for characterizing and analyzing social dynamics,

and this idea also extends to other areas, for example the analysis of scholarly collaboration networks [41].

With the steadily growing popularity of hypergraphs arises the need for efficient and effective algorithms to analyze the data they represent. Among the most important analysis techniques is clustering, which has the goal of splitting vertices into clusters while maximizing edges inside and minimizing edges in between said clusters. For graphs, there already exist a large number of highly efficient clustering algorithms, and they are frequently applied to examine social, biological or neural networks represented by graphs. For instance, clustering gene expression helps with the identification of functionally related genes [52], grouping cell lines assists in analyzing drug responses [12], and clustering protein sequences aids in understanding protein families [36]. Analysis of online social networks benefits from clustering techniques [31], and it is also used to cluster ego networks [33]. Additional applications in other fields of study include analyzing air transportation networks to improve planning and coordination [48], and enhancing personal online recommendations[53].

Despite the recognized advantages of hypergraphs, clustering them in application remains rarely used since clustering methods specifically designed for hypergraphs have not been properly developed. Existing works include hypergraph spectral clustering applied to protein-protein interaction networks between multiple species[35], partitioning of chromatin domains represented by hypergraphs [24] and clustering genotypic data [49]. Still, these remain relatively sparse compared to the extensive research using graph clustering. One option to cluster hypergraphs consists of transforming them into graphs to apply graph clustering methods. However, this transformation leads to a loss of multi-vertex connections since they are replaced with pairwise ones, losing the additional information they provide and effectively negating the advantages of the hypergraph representation. Consequently, there is a growing need to develop clustering algorithms that operate, if possible, directly on hypergraphs, thus preserving the structure of higher-order connections.

1.2 Our Contribution

In this work, we propose a clustering algorithm using flow-based minimum cuts adapted to hypergraphs. We apply different modularity functions to ensure high quality and refine results by a greedy strategy. Our key contributions are:

- Modularity Functions: We present, use and compare two modularity functions for measuring clustering quality, one of them defined directly on the hypergraph to avoid loss of information, and prove why the latter one is preferable in comparison to the original versions it is based on. We develop and prove modularity gain functions for both to efficiently compute the best choice with the highest modularity increase for two different kinds of cluster adjustments.

- Flow-Based Clustering Algorithm for Hypergraphs: We extend the Natural Cuts [15]

strategy for graph partitioning to hypergraph clustering by sampling and converting subhypergraphs into flow networks, computing their minimum cuts and merging the so-found clusters into a clustering of the entire hypergraph. We incorporate modularity gain computations to select the most beneficial cuts.

- Greedy Refinement Procedure: We propose a local refinement algorithm that iteratively improves the initial clustering by moving vertices into neighboring clusters if modularity increases.
- Parallelization Strategies: We design and analyze multiple parallelization approaches for both the flow-based clustering and the greedy refinement phase, making our algorithm more efficient for large hypergraphs.

These contributions provide a scalable modularity-driven framework for hypergraph clustering that combines global structural clustering with local optimization and comes with efficient parallelization.

1.3 Structure

The remainder of this thesis is organized as follows. Section 2.2 sets the foundation by introducing key concepts and definitions used throughout this work. We introduce the clustering problem in Section 2.2, define two hypergraph conversion techniques in Section 2.4 and introduce flow networks in Section 2.5, which are a main component of our algorithm. Section 3 lists different strategies for graph and hypergraph clustering, motivating a new approach that is presented in the following section. Section 4 presents our main contributions as described in 1.2. We start by introducing the criteria for usable clustering quality functions, present two modularity functions, one of which is usable directly on hypergraphs, and provide detailed mathematical proofs of why these functions satisfy the criteria in Sections 4.1.1 and 4.1.2. We then continue with describing the first phase of our algorithm in Section 4.2.1, where we explain how flow networks are sampled and derived from hypergraphs and how we use minimum cuts to split clusters. In Section 4.2.2, we introduce the greedy refinement strategy. Section 4.3.1 and Section 4.3.2 present and analyze parallelization techniques, specifically three different approaches for the flow-based algorithm. Section 5 begins by outlining implementation details and the experimental setup. We then present a wide range of experiment results, in parts aimed at fine-tuning our algorithm: we compare different choices for input parameters and stopping criteria to identify the ones providing the highest quality clustering, which are then chosen for the final version of our algorithm. We also assess efficiency and scalability of our parallelization. Additionally, we compare our algorithm to the well-known and frequently used greedy modularity maximization algorithms. Finally, we summarize and discuss our results and provide an outlook on future research in Section 6.

CHAPTER 2

Fundamentals

We begin by providing general definitions of graphs and hypergraphs, followed by an introduction to the clustering problem and the closely related graph partitioning problem. Then, we define some fundamental concepts, such as two different graph conversion techniques and the maximum flow problem, that we will use later in our algorithm.

2.1 General Definitions

An undirected weighted graph $G = (V, E, \omega)$ consists of a vertex set V, an edge set E where all $e \in E$ are subsets of V of size 2, and a weight function $\omega : E \to \mathbb{R}^+$. We denote the number of vertices and edges by $|V|$ and $|E|$, respectively. Hypergraphs are a generalization of graphs where one hyperedge can connect an arbitrary number of vertices. A weighted hypergraph $H = (V, E, \omega)$ is defined similarly to G, the only difference being the hyperedge set E with $e \subset V \; \forall e \in E$.

For both graphs and hypergraphs, the degree of a vertex $v \in V$ is defined as

$$d(v) := |(e \in E : v \in e)|,$$

i. e., the number of hyperedges containing v. The weighted degree is defined as

$$k_v = d_\omega(v) := \sum_{e \in E \,:\, v \in e} \omega(e).$$

The cardinality or size of a hyperedge $e \in E$ is defined as

$$|e| := |(v \in V \mid v \in e)|,$$

which counts the number of vertices in e. For graphs, we always have $|e| = 2$. The neighborhood of a vertex $v \in V$ is defined as

$$\mathcal{N}(v) := \{i \in V \setminus \{v\} \mid \exists e \in E \text{ s.t. } \{v, i\} \subseteq e\},$$

i. e., all vertices that share at least one hyperedge with v.

We define the total edge weight as

$$m := \sum_{e \in E} \omega(e).$$

In the case of unweighted graphs, this reduces to $m = |E|$. We also define the nontrivial edge weight

$$m' := m - \sum_{e \in E \colon |e|=1} \omega(e),$$

which excludes contributions from edges containing only one vertex, called self-loops (graphs) or singleton edges (hypergraphs).

2.2 Clustering

For a given hypergraph H with vertex set V, hyperedge set E and positive edge weights, the hypergraph clustering problem aims to partition V into densely connected components so-called clusters. The goal is to maximize the sum of edge weights within the clusters while minimizing the sum of weights of edges being cut. The number and size of clusters are not predetermined, but rather depend on the network. Formally, a clustering by our definition is a collection $\mathcal{C} = \{C_1, \ldots, C_k\}$ of disjoint subsets of V satisfying

$$V = \bigcup_{C \in \mathcal{C}} C \quad \text{and} \quad C_i \cap C_j = \emptyset \quad \text{for all } C_i \neq C_j \in \mathcal{C}.$$

Other definitions and methods might allow overlapping clusters [3], where one vertex can be assigned to multiple clusters at the same time.

During and after the clustering process, each vertex $v \in V$ is assigned to a cluster denoted by $C(v)$.

For each vertex $v \in V$, we define its neighboring clusters, i. e., the set of clusters containing neighbors of v, as

$$\mathcal{N}_{\mathcal{C}}(v) := \{C \in \mathcal{C} \mid \exists i \in \mathcal{N}(v) \text{ s.t. } C(i) = C\}.$$

The volume of a set or cluster $S \subseteq V$ is defined as

$$\mathrm{vol}(S) := \sum_{v \in S} d(v),$$

where $d(v)$ denotes the unweighted degree of v. For weighted hypergraphs, we define the weighted volume analogously as

$$\mathrm{vol}_{\omega}(S) := \sum_{v \in S} d_{\omega}(v),$$

where $d_\omega(v)$ is the weighted degree of v. In most use cases on this paper, we will be interested in the volume of the entire hypergraph, i. e., $S = V$.

To account for self-loops (edges of the form $e = \{v\}$), we define an adjusted weighted volume as the counterpart of m':

$$\mathrm{vol}'_\omega(S) := \mathrm{vol}_\omega(S) - \sum_{\substack{e \in E \\ e = \{v\},\, v \in S}} \omega(e).$$

We also define the normalized volume of a set S as

$$\mathrm{nvol}(S) := \sum_{v \in S} \sum_{\substack{e \in E \\ v \in e}} \frac{1}{|e|},$$

and its weighted version by treating a hyperedge e of weight $\omega(e)$ as $\omega(e)$ parallel unit-weight edges:

$$\mathrm{nvol}_\omega(S) := \sum_{v \in S} \sum_{\substack{e \in E \\ v \in e}} \frac{\omega(e)}{|e|}.$$

For $S = V$, this evaluates to

$$\mathrm{nvol}_\omega(V) = \sum_{e \in E} |e| \cdot \frac{\omega(e)}{|e|} = \sum_{e \in E} \omega(e) =: m.$$

Finally, we define the loyalty of a hyperedge e to a cluster C as

$$\ell(e, C) := \frac{|e \cap C|}{|e|},$$

that is, the fraction of the hyperedge's vertices that are contained in the cluster.

2.2.1 Related Problem: Partitioning

The graph partitioning problem, which asks to divide a given graph into k disjoint parts of roughly equal size, is closely related to graph clustering with no overlapping clusters allowed. For graph partitioning, the focus lies on dividing a graph efficiently, often with size constraints, as opposed to analyzing its structure. Still, a lot of techniques can be used for and adjusted to either of these problems, e. g., multilevel partitioning tools like KaHIP [44]. The Natural Cuts algorithm [15], which we adapt for our approach, is also designed to solve the graph partitioning problem rather than graph clustering problem. For further information on both theoretical aspects and main applications of graph partitioning, we refer the reader to Bichot et al. [4].

2.3 Modularity Functions

In this section, we introduce two main clustering quality functions, one of which is defined on a regular graph requiring previous hypergraph conversion, the other one defined directly on hypergraphs. These quality functions measure the difference between actual and expected innercluster edges, a strategy which is called modularity. In-depth explanations and proofs on why they are suitable can be found in Section 4.1.

2.3.1 Graph Modularity

To measure clustering quality on a clustered graph G, we will be using the modularity function [39] below, which is maximized when the edge weight inside of clusters is higher than expected and the edge weight in between clusters is lower than expected.
The modularity of a clustering C is computed by

$$Q_1(C) = \frac{1}{2m} \sum_{i,j} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

with adjacency matrix A of G, so $A_{i,j}$ is the edge weight between vertices i and j. $\delta(c_i, c_j)$ is the Kronecker delta function, i. e., it is 1 iff i and j are in the same cluster, and 0 if not. As defined in Section 2.1, k_i is the sum of the weights of the edges incident to i, the same goes for k_j. The term $\frac{k_i k_j}{2m}$ calculates the expected edge weight between i, j in the configuration model based on G, a graph generation model which works by randomly adding edges between vertices until the previously provided sum of incident edge weights k_n per vertex n is reached.

2.3.2 Hypergraph Modularity

Although applying modularity measures to the converted graph derived from a hypergraph is straightforward and uses well-established methods, it reduces the complexity of the original data by transforming higher-order relations into simple pairwise connections. Therefore, despite the algorithmic convenience, we additionally use the following function to measure clustering quality directly on a hypergraph, a modified version of the so-called PI-modularity by Feng et al. [21]:

$$Q_2(\mathcal{C}) = \frac{1}{m} \sum_{C \in \mathcal{C}} \sum_{e \in E_\theta(C)} \frac{\omega(e) l(e, C)}{log_2(\frac{1}{l(e,C)} + 1)} - \frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \frac{(1 - \eta_C)^2}{1 + \frac{\gamma \eta_C}{1 - \gamma}}$$

$$\text{with} \qquad \gamma := \frac{\text{vol}'_\omega(H) - 2m'}{\text{vol}'_\omega(H) - m'} \qquad \text{and} \qquad \eta_C := \theta \left(1 - \frac{\text{nvol}_\omega(C)}{m} \right)$$

Similar to the classical notion of graph modularity [39], the proposed function measures the discrepancy between the actual contribution of hyperedges to a cluster $C \in \mathcal{C}$ and the

expected contribution according to the hypergraph expansion model (HEM), also proposed by Feng et al. [21], which generates a random hypergraph on a given set of vertices with a predetermined number of edges. The expected contribution is defined as

$$\exp(C) := \frac{(1-\eta_C)^2}{1+\frac{\gamma\eta_C}{1-\gamma}},$$

while the actual contribution is given by

$$\text{supt}(C) := \sum_{e\in E_\theta(C)} \frac{\omega(e)\, l(e,C)}{\log_2\left(\frac{1}{l(e,C)}+1\right)},$$

where supt is short for support.

Here, $E_\theta(C)$ denotes the set of θ-innercluster hyperedges for cluster C, i. e., those hyperedges e for which at least a θ-fraction of their vertices are contained in C (i. e., $|e \cap C| \geq \theta|e|$). This relaxed criterion generalizes the commonly used All-Or-Nothing (AON) approach, e. g., by Chodrow et al. [13] and Kaminski et al. [28], where a hyperedge is only considered to be part of cluster only if all of its vertices lie within it. While AON simplifies computation, it neglects even those hyperedges that are almost entirely contained within a single cluster, losing important structural information. The θ-based definition allows for a more flexible and nuanced inclusion of hyperedges, making the measure more adaptable to varying hypergraph structures. The parameter $\theta \in (0, 1]$ can be adjusted depending on if a stricter or looser definition of internal connectivity is required. Since we consider weighted hypergraphs with integer weights $\omega(e)$, each hyperedge e of weight $\omega(e)$ is interpreted as $\omega(e)$ parallel hyperedges of weight 1. This transformation enables us to derive all theoretical results in the unweighted setting without loss of generality; all proofs extend naturally to the weighted case. Finally, we recall that m is defined as the sum of all hyperedge weights as described in Section 2.1, as opposed to the commonly used definition of $m = |E|$.

2.4 Hypergraph to Graph Conversion

Throughout the proposed algorithm introduced later in this work, we will encounter multiple sections where the input hypergraph must be transformed into a graph. This is necessary because some of the techniques and formulas we employ are inherently defined for graphs with edges of cardinality 2 ($|e| = 2$) and are not directly applicable to hypergraphs. The process of converting a hypergraph into a graph, often called reduction, is not uniquely defined, and the most appropriate conversion method can vary depending on the specific objective or structural property that should be preserved.

2.4.1 Clique Reduction

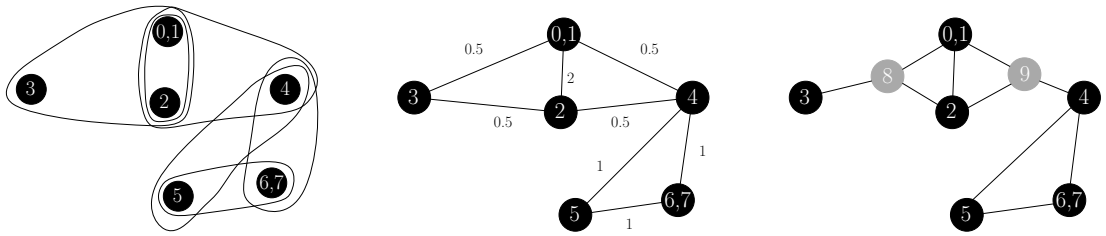
The first and most intuitive conversion strategy is called clique reduction [26], and it works exactly as the name suggests. Given a hypergraph $H = (V, E, \omega)$, every hyperedge $e \in E$ is replaced by a clique, meaning it is replaced by $\binom{|e|}{2}$ graph edges, one for every pair of vertices of e. Every edge of said clique is assigned the same weight of $\frac{\omega(e)}{|e|-1}$, ensuring the preservation of k_n for all vertices, i. e., for any vertex $i \in V$, the sum of weights of incident edges is maintained, meaning k_i in $H = k_i$ in G. Occurring parallel edges are combined into one single edge by summing up the edge weights. Self-loops are removed as they do not affect the clustering since they will always be contained entirely inside of a cluster. Figure 2.1b depicts an example: hyperedges of size 2 become normal graph edges, larger hyperedges are replaced by clique, in this case with edges of weight 0.5 since they are all size 3, and parallel edges are merged into one by summing up the weight.

Clique reduction, as opposed to other reduction approaches, does not add any new vertices, so it allows direct application of the later introduced graph modularity without modifying the modularity definition. Additionally, graph modularity computes the expected number of edges between pairs of vertices based on weighted vertex degree, so it is necessary to ensure this does not change during conversion, which clique reduction does. Even though the quadratic blow-up of edges (one edge of cardinality n is replaced by $\frac{1}{2}n(n-1)$ edges) can be seen as a drawback in terms of computational efficiency, it emphasizes the strength of association among vertices within each former hyperedge, which is important for computing an accurate modularity.

2.4.2 Star Expansion

The second type of conversion [11] is more efficient: given a hypergraph $H = (V, E, \omega)$, every hyperedge $e \in E$ with $|e| > 2$ is replaced by one new auxiliary vertex v and $|e|$ graph edges connecting every single one of the vertices formerly contained in the hyperedge e to v. Each one of these new edges is assigned the weight of the original hyperedge e. Hyperedges of size 2 are simply replaced by a normal edge since they are equivalent. Figure 2.1c depicts an example: Hyperedges of size 2 become normal graph edges, and larger hyperedges are replaced by a new vertex (gray) and one edge per vertex contained in former hyperedge.

Because it works by inserting additional vertices, star expansion is not suitable for modularity computations. However, it is a fitting choice for preparing the hypergraph for minimum cut computations, a strategy employed in the Natural Cuts component of our algorithm described in Section 4.2.1. Given that computing a min-cut requires preserving as much hyperedge information as possible, star expansion is the obvious choice. It upholds the global connectivity of a hyperedge rather than decomposing it into pairwise relations, which can lead to a loss of structural information.



(a) Original Hypergraph **(b)** Clique Reduction **(c)** Star Expansion

Figure 2.1: Hypergraph to Graph Conversion

In this model, cutting a converted hyperedge requires cutting all connections to the auxiliary vertex, whereas clique expansion allows individual edges to be cut independently, potentially underestimating the true cost of cutting the original hyperedge.

2.5 Flow Networks

We consider a capacitated flow network $F = (V, E, c, s, t)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of directed edges, also often called arcs. Each edge $(i, j) \in E$ has a nonnegative capacity $c_{ij} \geq 0$. Two distinguished vertices are the source $s \in V$ and the sink $t \in V$. This construct can be interpreted as a regular directed graph with nonnegative weights ω to represent the capacity.

A flow is a vector $x = \{x_{ij}\}_{(i,j)\in E}$ that satisfies:

$$\begin{aligned}
\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} &= f && \text{if } i = s \\
\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} &= 0 && \text{if } i \in V \setminus \{s, t\} \\
\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} &= -f && \text{if } i = t \\
0 \leq x_{ij} &\leq c_{ij} && \text{for all } (i, j) \in E
\end{aligned}$$

The goal of the maximum flow problem is to find a feasible flow x that maximizes the total flow value f from the source s to the sink t.

We refer the reader to Ahuja et al. [2] for more detailed descriptions of flow algorithms and applications.

The minimum cut problem on the other hand asks to partition the set of vertices $V(F)$ into two disjoint sets S and T where $s \in S$ and $t \in T$ s.t. the capacity of the cut (S, T) is minimized. The capacity is defined as:

$$\text{cap}(S, T) := \sum_{\substack{(i,j) \in E \\ i \in S,\, j \in T}} c(u, v).$$

Ford and Fulkerson [22] showed that in any capacitated flow network, the maximum value of a feasible s-t flow is equal to the minimum capacity of an s-t cut. This enables us to apply efficient state-of-the-art maximum flow algorithms to compute minimum cuts, which we will need in our algorithm later.

CHAPTER 3

Related Work

Many different approaches for solving both the graph and the hypergraph clustering problem have been proposed in literature [47, 6, 27, 30]. These methods can be broadly grouped into several categories based on how they define cluster quality, whether they allow overlapping clusters, and whether they operate globally or locally on the input structure.

3.1 Graph Clustering Methods

Among all the different ways of clustering a graph, working with modularity functions [38], which aim at maximizing innercluster edges while simultaneously minimizing edges in between clusters, remains the most frequently used method. The preferred method on regular graphs is the Louvain algorithm by Blondel et al. [6], a greedy modularity maximization approach which starts with one vertex per cluster and uses local search to move vertices to new clusters such that modularity is optimized. Extensions like the Leiden algorithm [50] further improve upon Louvain by addressing disconnected clusters and refining local movement, which makes it useful as a robust modularity maximization baseline for both graphs and converted hypergraphs.

Another quality measure which can be used to cluster a graph is called conductance, measuring how well-separated a group of vertices (cluster) is from the rest of a graph [32]. To accurately measure the quality of less densely clustered graphs, another metric to use is cluster path length, which measures quality based on the length of paths inside of clusters [54].

Spectral clustering [27], [55] is a fundamentally different approach to clustering a graph as compared to modularity optimization. This method makes use of the eigenvectors of the graph Laplacian to project vertices into a k-dimensional space, where it becomes easier to detect clusters. Then, traditional clustering methods such as k-means can be applied, which for each cluster computes the average position of all points currently assigned to

that cluster (centroid) and assigns each point to the closest centroid's cluster, minimizing within-cluster variance, i. e., the total distance from each point to its cluster.

Another method for clustering a graph is modeling it using mathematical programming. For instance, linear or integer programming formulations can optimize cluster quality based on flow or cut-based metrics [9]. These methods are often computationally expensive but can provide strong guarantees.

Although many methods require disjoint clusters, some clustering strategies allow overlapping communities. One of these techniques is called seed expansion, where clusters are grown from small sets of vertices by adding nearby vertices that meet certain connectivity criteria while allowing them to belong to multiple clusters at the same time [34, 51]. The Label Propagation Algorithm (LPA) [43] and Clique Percolation [40], where overlapping communities are found via percolation of k-cliques, are other well-known methods that naturally yield overlapping clusters.

Finally, one approach is to take advantage of the efficient computability of max-flow solvers and the max-flow min-cut theorem as described in Section 2.5. Since graph clustering aims to identify clusters with as few edges between them as possible, finding the minimum cut that separates sets of vertices in the graph is a useful strategy, used by Feng et al. [21] and Sarcheshmehpour et al. [46] among others, and it forms the foundation of our algorithm as well.

3.2 Hypergraph Clustering Methods

Hypergraph clustering presents additional challenges due to the complexity of hyperedges connecting multiple vertices. The most intuitive approach is clustering a hypergraph by converting it into a graph and then applying graph clustering methods like Louvain [6], as described in the work of Kumar et al. [30].

Instead of transforming the hypergraph to match the modularity function, it is also possible to extend the definition of modularity to hypergraphs. The simpler but less meaningful option is to only count a hyperedge towards a cluster if it is completely contained, also called the All-Or-Nothing approach (AON) [13, 28]. Feng et al. [21] avoid this by introducing the Partial Innerclusteredge modularity (PI) based on a new random hypergraph model called the Hyperedge Expansion Model (HEM). In their paper, they propose the so-called clustering algorithm Partial Innerclusteredge Clustering (PIC) which optimizes PI-modularity (defined in Section 4.1.2).

To simplify cluster analysis, one can look at k-uniform hypergraphs which restrict hyperedge sizes to k, i. e., every hyperedge hyperedge contains exactly k vertices [1, 7]. This constraint allows generalization of graph-based concepts such as conductance or modularity.

A method working directly on hypergraphs without conversion or restriction is called motif-based clustering [10, 11] which groups vertices into one cluster based on recurring small patterns (motifs) in a hypergraph. This is especially useful for hypergraphs and com-

plex networks where relations involve more than two vertices. Other hypergraph clustering strategies work via merging clusters by using a so-called linkage function, computing the similarity between pairs of clusters and merging those with highest similarity [17].

CHAPTER 4

Clustering Hypergraphs

In this chapter, we describe our strategy for clustering hypergraphs, making decisions based on different modularity functions measuring clustering quality. We begin by offer some proofs on why our modularity functions are suitable. We then continue with descriptions of the flow-based cuts algorithm and the greedy improvement strategy used, and discuss impact of parameter variations. Finally, we present different parallelization techniques and their strengths and drawbacks.

4.1 Measuring Clustering Quality

To measure the quality of a hypergraph clustering, we need a formula with the following properties:

1. It needs to be bounded to ensure interpretability by keeping values in a known range and comparability of different clusterings on the same hypergraph.

2. It should take on values ≤ 0 for both trivial solutions, where either all vertices are in a single cluster or every vertex is in its separate cluster, since none of these two clusterings convey any information about the connection of the represented data.

3. Most importantly, it needs to fulfill the criterion for a good/bad clustering: The goal is to maximize the sum of edge weights within clusters while simultaneously minimizing the sum of edge weights in between clusters, so the quality measurement formula should be high when these conditions are met, and low when they are not.

A well-known measure meeting all the listed criteria is called modularity, originally introduced by Newman et al. [39], and it is defined as the difference between the expected fraction of edges and the fraction of edges that actually fall within clusters. For graphs with only edges of size 2, defining a modularity formula is straightforward and commonly

used, which is why we are introducing it here, even though it requires rather inefficient hypergraph-to-graph conversion, losing information in the process. To avoid the downsides of computing clustering quality of a hypergraph on its graph counterpart, we are additionally using another modularity formula defined directly on a hypergraph. Since translating the concept of modularity on graphs with edges of any size is rather ambiguous, there is not one single hypergraph modularity function that is commonly used, but many different definitions.

The hypergraph modularity function we use here is a modified version of the so-called PI modularity [21]. The original function is not suitable for our case because it was defined for unweighted edges and does not meet all the criteria mentioned above, proofs can be found in Lemma 5 and Lemma 6. In the following, we will prove some main qualities for both quality functions and derive two gain functions each to simplify and optimize their later usage in our clustering algorithm.

4.1.1 Graph Modularity

As already introduced in Section 2.3.1, we will be using the following modularity function [39]:

$$Q_1(C) = \frac{1}{2m} \sum_{i,j} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

The modularity Q_1 is high if actual weight between i, j in G is higher than the expected weight for all (or a majority of) i, j in same cluster. Evidently, maximizing the modularity function is the same as maximizing edge weight sums in clusters and minimizing them in between clusters, which is exactly what we want. As we will see in the following two lemmas, Q_1 also meets the other requirements for quality measuring functions.

Lemma 1
The modularity function Q_1 is bounded; more specifically, its values always lie within the interval $[-1, 1]$.

Proof. First we need to prove that $Q_1 \leq 1$. This part is easy to see: Q_1 is maximized when the number of innercluster-edges is maximized, so we assume all edges are inside of clusters. We get

$$\frac{1}{2m} \sum_{i,j} A_{i,j} \delta(c_i, c_j) = \frac{1}{2m} \sum_{i,j} A_{i,j} = 1,$$

and since $\frac{k_i k_j}{2m}$ is obviously always positive, Q_1 is bounded from above by 1, so $Q_1 \leq \frac{1}{2m} \sum_{i,j} A_{i,j} \delta(c_i, c_j) \leq 1$ Now we need to show that $Q_1 \geq -1$. We know that Q_1 is minimized when all edges are between clusters, so none are in the same cluster. This means $\sum_{i,j} A_{i,j} \delta(c_i, c_j) = 0$, and

$$\frac{1}{2m} \sum_{i,j} \frac{k_i k_j}{2m} \delta(c_i, c_j) \leq \frac{1}{2m} \sum_{i,j} \frac{k_i k_j}{2m} = \frac{1}{(2m)^2} \sum_{i,j} k_i k_j = \frac{1}{(2m)^2} \left[\sum_i k_i \right]^2 = \frac{(2m)^2}{(2m)^2} = 1$$

so

$$Q_1 = \frac{1}{2m}\sum_{i,j} A_{i,j}\delta(c_i, c_j) - \frac{1}{2m}\sum_{i,j}\frac{k_i k_j}{2m}\delta(c_i, c_j) = 0 - \frac{1}{2m}\sum_{i,j}\frac{k_i k_j}{2m}\delta(c_i, c_j) \geq -1.$$

■

Lemma 2
For trivial clusterings carrying no meaningful information (assigning all vertices to the same cluster or placing each vertex in its own cluster), $Q_1 = 0$.

Proof. For $|\mathcal{C}| = 1$, all edges lie inside of a cluster, so $\delta(c_i, c_j)$ is always 1, and by transforming the expression as in the proof above using

$$\sum_{i,j} k_i k_j = \left[\sum_i k_i\right]^2 = (2m)^2,$$

we get $Q_1 = 0$. For $c_i \neq c_j$ for all vertices i, j, all edges fall in between clusters, meaning we simply have $\delta(c_i, c_j) = 0\ \forall c_i, c_j$, so $Q_1 = 0$ as well. ■

Since a quality value of less than 0 ($Q_1 \leq 0$) would imply that this division into clusters is worse than if assigned randomly, we work with a clustering quality between 0 and 1, and ensure this by starting with a null model (e. g., all vertices in the same cluster) and only allowing changes that increase clustering quality.

Modularity Gain. To be able to use this formula efficiently in our algorithm later, we need to avoid recomputing the entire modularity as much as possible by only calculating the change in modularity resulting from the cluster changes. We will be dealing with two different kinds of clustering modifications:

In one case, we need to calculate the change in modularity when "layering" a new cluster S on top of the already existing clusters, splitting each old cluster C into two new ones $C_1 := C \cap S$, $C_2 := C \setminus C_1$. Instead of recomputing the entire graph modularity, it is enough to only consider the clusters getting split. In most cases, this strategy is more efficient since rarely all clusters are affected. We compute the changes in modularity of clustering $\mathcal{C}$ with the following formula with input set S:

$$\Delta Q_1(\mathcal{C}, S) = \frac{1}{m}\sum_{\substack{i\in S, j\notin S \\ j\in\text{former cluster}(i)}}\left[\frac{k_i k_j}{2m} - A_{i,j}\right]$$

Lemma 3
$\Delta Q_1(\mathcal{C}, S)$ *correctly computes the change in modularity when splitting of* $\mathcal{C}$ *according to* S *as described above.*

Proof. We begin by showing that $\Delta Q_1(\mathcal{C}, S)$ properly determines the modularity adjustment for splitting one cluster of the original clustering $\mathcal{C}$. The contribution of one cluster $C \in \mathcal{C}$ to the total modularity of $\mathcal{C}$ is

$$\frac{1}{2m} \sum_{\substack{i \in C \\ j \in C}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right],$$

and by adding $\Delta Q_1(\mathcal{C}, S)$ for cluster C we get

$$\begin{aligned}
&\frac{1}{2m} \sum_{\substack{i \in C \\ j \in C}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] + \frac{1}{m} \sum_{\substack{i \in S \cap C, j \notin S \\ j \in \text{former cluster}(i)}} \left[\frac{k_i k_j}{2m} - A_{i,j} \right] \\
&= \frac{1}{2m} \left(\sum_{\substack{i \in C \\ j \in C}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] - 2 \sum_{\substack{i \in C_1 \\ j \in C_2}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \right) \\
&= \frac{1}{2m} \Bigg(\sum_{\substack{i \in C_1 \\ j \in C_1}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] + \sum_{\substack{i \in C_2 \\ j \in C_2}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \\
&\qquad + \sum_{\substack{i \in C_1 \\ j \in C_2}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] + \sum_{\substack{i \in C_2 \\ j \in C_1}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] - 2 \sum_{\substack{i \in C_1 \\ j \in C_2}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \Bigg) \\
&= \frac{1}{2m} \left(\sum_{\substack{i \in C_1 \\ j \in C_1}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] + \sum_{\substack{i \in C_2 \\ j \in C_2}} \left[A_{i,j} - \frac{k_i k_j}{2m} \right] \right)
\end{aligned}$$

which is exactly the contribution of the two new clusters C_1, C_2 to the modularity of the entire graph. By summing up over $i \in S, j \notin S, j \in \text{former cluster}(i)$, we adjust the modularity for all clusters being split an thus get

$$Q_1(\mathcal{C}) + \Delta Q_1(\mathcal{C}, S) = Q_1(\mathcal{C}')$$

with $\mathcal{C}'$ being the new clustering after applying the changes as described above. ■

This gain function is very intuitive as we simply adjust the modularity for all i, j that used to be in the same cluster but are now separated. Only pairs of vertices in the same cluster have an impact on Q_1, and the algorithm only performs cluster splits, without merging clusters or moving individual vertices. Therefore, no new pairs i, j of vertices in the same cluster are introduced, and thus no other changes to the modularity need to be considered.

In the other case, we want to compute the quality change for moving a single vertex to one of its neighboring clusters, i. e., clusters that contain neighboring vertices. Here, we have an even smaller change in modularity, and directly computing the modularity gain is

decidedly more efficient. The gain of moving vertex i from its old cluster C_{old} to a new cluster C_{new} can be computed as follows:

$$\Delta Q_1(i, C_{\text{old}} \to C_{\text{new}}) = \frac{k_{i,\text{in, new}} - k_{i,\text{in, old}}}{m} - \frac{k_i(\Sigma_{\text{tot, new}} + \Sigma_{\text{in, new}} - \Sigma_{\text{tot, old}} - \Sigma_{\text{in, old}} + k_i)}{2m^2}$$

We denote by $\sum_{\text{in}}$ the sum of the weights of all edges with both endpoints in cluster C, and by $\sum_{\text{tot}}$ the sum of the weights of all edges incident to vertices in C. Furthermore, for a vertex i, we define $k_{i,\text{in}}$ as the sum of the weights of all edges between i and other vertices in C. The old / new in the subscript refers to which cluster this variable is in reference to, e. g., $\sum_{\text{in, new}}$ is defined as sum of the weights of all edges with both endpoints in cluster C_{new}.

Lemma 4
$\Delta Q_1(i, C_{old} \to C_{new})$ correctly computes the change in modularity when moving a vertex i from its current cluster C_{old} to C_{new}.

Proof.

To construct this formula, we decompose the process of moving vertex i from its original cluster to a new cluster into two steps: First, i is removed from its current cluster C_{old} and placed into a new isolated cluster (singleton). Then, in a second step, i is moved from this singleton into the target cluster C_{new}. As a singleton has no internal edges and hence does not contribute to the overall modularity, this decomposition into two parts simplifies the understanding of the gain formula without adding unnecessary computations to the function. Moreover, moving a vertex from a singleton cluster into an existing cluster yields a modularity gain that is equal in magnitude but opposite in sign to the gain obtained when moving the same vertex from that cluster back into the singleton, so it suffices to consider only one direction of the move in the analysis. To compute the contribution of a given vertex i in cluster $C \in \mathcal{C}$ to the total modularity of $\mathcal{C}$, it is only necessary to consider vertices $j \in C$ and thus can be written as follows:

$$\frac{1}{m} \sum_{j \in C} \left(A_{i,j} - \frac{k_i k_j}{2m} \right)$$

Here, the expression is multiplied by $\frac{1}{m}$ instead of $\frac{1}{2m}$, since each vertex pair (i, j) is considered only once. In contrast, the original modularity function Q_1 accounts for both (i, j) and its symmetric counterpart (j, i), effectively double-counting each edge.

Additionally, observe that

$$(1) \quad \sum_{j \in C} A_{i,j} = k_{i,\text{in}}, \qquad \text{and} \qquad (2) \quad \sum_{j \in C} k_j = \Sigma_{\text{tot}} + \Sigma_{\text{in}}.$$

Identity (1) holds by definition of $k_{i,\text{in}}$. For (2), note that we are summing the weighted degrees of all vertices in C, which includes each internal edge twice, once from each endpoint in C, and each external edge once. Thus, the result equals the sum of all edge weights incident to vertices in C, i. e., $\Sigma_{\text{tot}} + \Sigma_{\text{in}}$. Combining these results, we get

$$\frac{1}{m}\sum_{j\in C}\left(A_{i,j}-\frac{k_i k_j}{2m}\right)=\frac{1}{m}\left(\sum_{j\in C}A_{i,j}-\frac{k_i}{2m}\sum_{j\in C}k_j\right)=\frac{1}{m}\left(k_{i,\text{in}}-\frac{k_i}{2m}(\Sigma_{\text{tot}}+\Sigma_{\text{in}})\right).$$

We have now determined the change in modularity associated with moving a vertex i from an isolated cluster into an existing one. By symmetry, the modularity change for moving i out of a cluster into its own singleton follows directly, with the only remaining adjustment being Σ_{tot} and Σ_{in} for the original cluster. Since our formula assumes that i is no longer part of the cluster, we must account for the fact that its edges should no longer contribute to Σ_{tot} and Σ_{in}: edges that were previously internal now only contribute to Σ_{tot}, and edges that were previously external are no longer connected to the cluster at all. Overall, this change is captured by subtracting k_i from the sum, i. e., we get $\Sigma_{\text{itot}}+\Sigma_{\text{in}}-k_i$. Summing up the modularity of both earlier described steps and performing some trivial simplifications yields

$$\begin{aligned}
&\frac{1}{m}\left(k_{i,\text{in, new}}-\frac{k_i}{2m}(\Sigma_{\text{tot, new}}+\Sigma_{\text{in, new}})\right)-\frac{1}{m}\left(k_{i,\text{old}}-\frac{k_i}{2m}(\Sigma_{\text{tot, old}}+\Sigma_{\text{in, old}}-k_i)\right)\\
&=\frac{k_{i,\text{in, new}}-k_{i,\text{in, old}}}{m}-\frac{k_i(\Sigma_{\text{tot, new}}+\Sigma_{\text{in, new}}-\Sigma_{\text{tot, old}}-\Sigma_{\text{in, old}}+k_i)}{2m^2}\\
&=\Delta Q_1(i, C_{\text{old}}\to C_{\text{new}})
\end{aligned}$$

■

Observe that if vertex i has no neighboring vertices in C_{new}, then $\frac{k_{i,\text{in, new}}}{m}=0$. In this case, even if the third and fourth terms in the modularity gain function representing moving i out of C_{old} were positive, it would still be preferable to assign i to a singleton cluster. This is because the expression

$$-\frac{k_i(\Sigma_{\text{tot, new}}+\Sigma_{\text{in, new}})}{2m^2}$$

would evaluate to zero, rather than contributing a negative value to $\Delta Q_1(i, C_{\text{old}}\to C_{\text{new}})$.

4.1.2 Hypergraph Modularity

We will begin by introducing the original PI-modularity functions by Feng et al. [21], one from the paper and one from the implementation, and prove that they do not meet all criteria from 4.1. We start with the definition offered in the paper:

$$\text{PI}(\mathcal{C})=\frac{1}{m}\sum_{C\in\mathcal{C}}\left(\sum_{e\in E_\theta(C)}\frac{l(e,C)}{log_2(\frac{1}{l(e,C)}+1)}-\frac{m(1-\eta_C)^2}{1+\frac{\gamma\eta_C}{1-\gamma}}\right)$$

Lemma 5
The original function PI($\mathcal{C}$) for unweighted graphs [21] by Feng et al. is unbounded.

Proof. We begin by bounding the relevant parameters and expressions. Note that for all $0 < x \leq 1$, the inequality

$$0 < \frac{x}{\log_2\left(\frac{1}{x}+1\right)} \leq x$$

holds. Furthermore, for all $e \in E$, by definition of $l(e, C)$ we have

$$0 \leq \sum_{C\,:\,e\in E_\theta(C)} l(e,C) \leq 1, \quad \text{with} \quad 0 < l(e,C) \leq 1.$$

It follows that

$$\begin{aligned} 0 &< \frac{1}{m}\sum_{C\in\mathcal{C}}\sum_{e\in E_\theta(C)} \frac{l(e,C)}{log_2(\frac{1}{l(e,C)}+1)} \\ &= \frac{1}{m}\sum_{e\in E}\sum_{C:e\in E_\theta(C)} \frac{l(e,C)}{log_2(\frac{1}{l(e,C)}+1)} \\ &\leq \frac{1}{m}\sum_{e\in E}\sum_{C:e\in E_\theta(C)} l(e,C) \leq \frac{1}{m}\sum_{e\in E} 1 = 1 \end{aligned}$$

Next, we bound the parameters γ and η_C. By definition, trivial edges are excluded from $\text{vol}'(H)$ and m', implying $\text{vol}'(H) \geq 2m'$. Hence,

$$0 \leq \frac{\text{vol}'(H) - 2m'}{\text{vol}'(H) - m'} < 1,$$

where $\gamma \to 1$ as $\text{vol}'(H) \to \infty$.

For η_C, we use the bound $\text{nvol}(C) \leq m$, which gives

$$0 < \frac{\text{nvol(C)}}{m} \leq 1 \Rightarrow 0 \leq \theta\left(1 - \frac{\text{nvol}(C)}{m}\right) < \theta \leq 1$$

Let

$$T(\eta_C, \gamma) := \frac{(1-\eta_C)^2}{1+\frac{\gamma\eta_C}{1-\gamma}}.$$

Since $(1-\eta_C)^2 > 0$ for $0 \leq \eta_C < 1$ and $1 + \frac{\gamma\eta_C}{1-\gamma} > 1$ for $0 \leq \gamma < 1$, we immediately obtain $T(\eta_C, \gamma) \geq 0$.

To show $T(\eta_C, \gamma) < 1$, consider:

$$\frac{(1-\eta_C)^2}{1+\frac{\gamma\eta_C}{1-\gamma}} < 1 \quad \Longleftrightarrow \quad (1-\eta_C)^2 < 1 + \frac{\gamma\eta_C}{1-\gamma}.$$

Expanding the left-hand side gives:

$$1 - 2\eta_C + \eta_C^2 < 1 + \frac{\gamma\eta_C}{1-\gamma} \quad \Longleftrightarrow \quad \eta_C\left(\eta_C - 2 - \frac{\gamma}{1-\gamma}\right) < 0.$$

Since $\eta_C \geq 0$ and $\eta_C - 2 - \frac{\gamma}{1-\gamma} < 0$ for all $0 < \eta_C < 1$ and $0 \leq \gamma < 1$, the inequality holds. Hence, $0 \leq T(\eta_C, \gamma) < 1$.

What remains to be shown is that the term

$$\frac{1}{m} \sum_{C \in \mathcal{C}} \frac{m(1-\eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}} = \sum_{C \in \mathcal{C}} \frac{(1-\eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}} = \sum_{C \in \mathcal{C}} T(\eta_C, \gamma)$$

is unbounded as the number of clusters grows. To demonstrate this, we consider a graph partitioned into a large number of clusters, where each cluster contains exactly one vertex (i. e., singleton clusters), and each vertex has degree 1. For simplicity, we assume a graph, where each edge connects exactly two vertices. In this setting, since the graph is binary, we have $\gamma = 0$. Moreover, the normalized volume of each cluster satisfies $\mathrm{nvol}(C) = \frac{1}{2}$, and thus

$$\begin{aligned} \sum_{C \in \mathcal{C}} T(\eta_C, \gamma) = |\mathcal{C}|(1-\eta_C)^2 &= 2m(1 - \theta(1 - \frac{0.5}{m}))^2 \\ &= 2m(1-\theta)^2 + 2\theta(1-\theta) + \frac{\theta^2}{2m} \xrightarrow{m \to \infty} \infty \end{aligned}$$

which shows that $\mathrm{PI}(\mathcal{C})$ is not bounded from below. ■

Evidently, this definition of PI-modularity proves unsuitable for our purposes, as it may produce arbitrarily low quality scores, thereby undermining the comparability of clustering results.

A slightly modified version of this modularity function can be found in the implementation [20] by Feng et al. associated with their publication [21] :

$$\mathrm{PI}_{\mathrm{impl}}(\mathcal{C}) = \frac{1}{m} \sum_{C \in \mathcal{C}} \left(\sum_{e \in E_\theta(C)} \frac{l(e,C)}{log_2(\frac{1}{l(e,C)} + 1)} - \frac{(1-\eta_C)^2}{m(1 + \frac{\gamma \eta_C}{1-\gamma})} \right)$$

This version is preferable, as it is bounded. This is not proven here, but follows readily from the reasoning and estimations used in the proof of Lemma 5, suitably adapted to this formula. However, it exhibits another shortcoming that renders it unsuitable for our purposes, as will be demonstrated below.

Lemma 6

The function $\mathrm{PI}_{impl}(\mathcal{C})$ for unweighted graphs found in the implementation [20] by Feng et al. associated with their publication [21] approaches 1 very quickly for the trivial clustering of all vertices in the same cluster, when number of edges approaches infinity.

Proof. Assume all vertices are in the same cluster. In that case, $l(e, C) = 1$ and $e \in E_\theta(C)$ for all edges, and thus

$$\frac{1}{m} \sum_{C \in \mathcal{C}} \sum_{e \in E_\theta(C)} \frac{l(e, C)}{log_2(\frac{1}{l(e,C)} + 1)} = \frac{1}{m} \sum_{e \in E} 1 = 1.$$

Additionally, we get $\text{nvol}(C) = m$, so $\frac{\text{nvol}(C)}{m} = 1$ and $\eta_C = 0$, which leads to

$$\frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \frac{(1 - \eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}} = 1$$

so it is obvious that

$$\begin{aligned} \text{PI}_{\text{impl}}(\mathcal{C}) &= \frac{1}{m} \sum_{C \in \mathcal{C}} \left(\sum_{e \in E_\theta(C)} \frac{l(e, C)}{log_2(\frac{1}{l(e,C)} + 1)} - \frac{(1 - \eta_C)^2}{m(1 + \frac{\gamma \eta_C}{1-\gamma})} \right) \\ &= 1 - \frac{1}{m} \left(\frac{(1 - \eta_C)^2}{m(1 + \frac{\gamma \eta_C}{1-\gamma})} \right) = 1 - \frac{1}{m^2} \xrightarrow{m \to \infty} 1 \end{aligned}$$

■

Since this function does also not meet our criteria from Section 4.1, we have proven that our suggested changes resulting in the function

$$Q_2(\mathcal{C}) = \frac{1}{m} \sum_{C \in \mathcal{C}} \sum_{e \in E_\theta(C)} \frac{\omega(e) l(e, C)}{log_2(\frac{1}{l(e,C)} + 1)} - \frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \frac{(1 - \eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}},$$

as already introduced in Section 2.3.2, are necessary. We remains to show is that this new function Q_2 does meet all criteria from Section 4.1, which we will do in the following.

Lemma 7
The modularity function Q_2 is bounded; more specifically, its values always lie within the interval $(-1, 1]$.

Proof. For this proof, we primarily rely on the bounds established in the proof of Lemma 5. From that, we know

$$0 < \frac{1}{m} \sum_{C \in \mathcal{C}} \sum_{e \in E_\theta(C)} \frac{l(e, C)}{log_2(\frac{1}{l(e,C)} + 1)} \leq 1$$

and

$$T(\eta_C, \gamma) := \frac{(1 - \eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}}, \quad 0 \leq T(\eta_C, \gamma) < 1$$

It follows that

$$\frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \frac{(1 - \eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}} < \frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} 1 = 1$$

Combining all results, we get

$$-1 < Q_2(\mathcal{C}) \leq 1$$

which concludes the proof. ■

Lemma 8
For trivial clusterings carrying no meaningful information, i. e., assigning all vertices to the same cluster or placing each vertex in its own cluster, $Q_2 \leq 0$.

Proof. We begin by showing this for the easier case, where all vertices are inside of one large cluster. From the proof of Lemma 6, we already know that

$$\frac{1}{m} \sum_{C \in \mathcal{C}} \sum_{e \in E_\theta(C)} \frac{l(e,C)}{log_2(\frac{1}{l(e,C)} + 1)} = 1$$

and

$$\frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} \frac{(1-\eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}} = 1,$$

so we get $Q_2(\mathcal{C}) = 1 - 1 = 0$ as required.
The second case, where every vertex is in a singleton, is more complicated to show. For a hyperedge e with $|e|$ vertices, we have $|e \cap C| \leq 1$ for all C. Thus, for any cluster C,

$$l(e,C) = \frac{|e \cap C|}{|e|} \leq \frac{1}{|e|}.$$

Therefore, if we assume $\theta > \frac{1}{\min_{e \in E} |e|}$, no edge satisfies the θ-inner-cluster condition in any cluster. As a result, the first term in $Q_2(\mathcal{C})$ evaluates to 0. For the second term, we use $\text{nvol}(C) \ll m$ to get $\eta_C \to \theta$ and consequently,

$$\frac{(1-\eta_C)^2}{1 + \frac{\gamma \eta_C}{1-\gamma}} \to (1-\theta)^2.$$

hence it approaches 0 as $\theta \to 1$. Therefore, $Q_2(\mathcal{C}) \leq 0$ under singleton clustering when θ is chosen large enough. If we pick a smaller θ, more edges will be counted as being inside of a cluster, increasing the first term of Q_1 only marginally because singleton clusters imply small values for $l(e,c)$). Simultaneously, the second term approaches 1 as $\theta \to 1$, ensuring that $Q_1 \leq 0$. ■

We conclude that Q_2 serves as a suitable function for measuring clustering quality, as it satisfies the essential criteria of a modularity function that we demanded at the beginning of this chapter in Section 4.1: (1) it is bounded, (2) it assigns low quality scores to trivial clusterings, and (3) it computes the difference between expected and actual edges inside of clusters.

Modularity Gain. As for the graph modularity function Q_1, we aim to minimize redundant computations when evaluating changes in clustering quality. To this end, deriving modularity gain functions is a key step toward developing an efficient algorithm. We again introduce two expressions for the modularity gain ΔQ_2, each corresponding to a different type of cluster modification, the same ones as introduced in Section 4.1.1 Both expressions follow directly from the definition of Q_2, using arguments analogous to those employed in Lemmas 3 and 4, which is why we omit formal proofs to avoid repetition and only provide explanations.

The first gain function, denoted by $\Delta Q_2(\mathcal{C}, S)$, measures the change in clustering quality resulting from splitting each affected cluster $C \in \mathcal{C}$ into two disjoint new clusters $C \cap S$ and $C \setminus S$, where $S \subset V$ is a given set of vertices. We define $\mathcal{C}' \subset \mathcal{C}$, $C \in \mathcal{C}'$ if $C \cap S \neq \emptyset$ and $C \cap S \neq S$, and get the following definition:

$$\begin{aligned}\Delta Q_2(\mathcal{C}, S) =& \frac{1}{m} \sum_{C \in \mathcal{C}'} \Big(\text{supt}_\theta(C \cap S) + \text{supt}_\theta(C \setminus S) - \text{supt}_\theta(C) \Big) \\ & - \frac{1}{|\mathcal{C}'|} \sum_{C \in \mathcal{C}'} \Big(\text{exp}(C \cap S) + \text{exp}(C \setminus S) - \text{exp}(C) \Big)\end{aligned}$$

This clustering adjustment affects all clusters C sharing at least one and at most $|C| - 1$ vertices with set S, because in those cases, C is being split into two new clusters. Therefore, the actual and expected edge contributions must be updated for all such clusters $C \in \mathcal{C}'$ by subtracting their previous contributions and adding the respective contributions of the resulting clusters $C \cap S$ and $C \setminus S$.

The second gain function, denoted $\Delta Q_2(i, C_{\text{old}} \to C_{\text{new}})$, captures the change in quality resulting from moving vertex i from cluster C_{old} to cluster C_{new}:

$$\begin{aligned}\Delta Q_2(i, C_{\text{old}} \to C_{\text{new}}) = \frac{1}{m} \Bigg(& \text{supt}_\theta(C_{\text{new}} \cup \{i\}) - \text{supt}_\theta(C_{\text{new}}) - \text{supt}_\theta(\{i\}) \\ & - \Big(\text{supt}_\theta(C_{\text{old}}) - \text{supt}_\theta(C_{\text{old}} \setminus \{i\}) - \text{supt}_\theta(\{i\}) \Big) \Bigg) \\ & - \Bigg(\text{exp}(C_{\text{new}} \cup \{i\}) - \text{exp}(C_{\text{new}}) - \text{exp}(\{i\}) \\ & - \Big(\text{exp}(C_{\text{old}}) - \text{exp}(C_{\text{old}} \setminus \{i\}) - \text{exp}(\{i\}) \Big) \Bigg) \\ = \frac{1}{m} \Bigg(& \text{supt}_\theta(C_{\text{new}} \cup \{i\}) - \text{supt}_\theta(C_{\text{new}}) - \text{supt}_\theta(C_{\text{old}}) + \text{supt}_\theta(C_{\text{old}} \setminus \{i\}) \Bigg) \\ & - \Bigg(\text{exp}(C_{\text{new}} \cup \{i\}) - \text{exp}(C_{\text{new}}) - \text{exp}(C_{\text{old}}) + \text{exp}(C_{\text{old}} \setminus \{i\}) \Bigg)\end{aligned}$$

Since this vertex movement affects only two clusters (C_{old} and C_{new}), adjustments are required only for these. Specifically, we subtract both the actual (supt) and expected (exp) contributions corresponding to the clusters prior to the move (C_{old} and C_{new}), and then add the updated values after the move has been applied: $C_{\text{old}} \setminus \{i\}$ and $C_{\text{new}} \cup \{i\}$).

4.2 Finding Clusters

In this section, we introduce our strategy for obtaining a high-quality clustering, which combines two complementary approaches: The first is based on the so-called Natural Cuts strategy [15] that repeatedly samples parts of the hypergraph, computes their minimum cuts 2.5, and aggregates the resulting partitions to construct an initial clustering capturing global structural patterns. The second is a greedy refinement strategy that improves this preliminary result by reassigning vertices to neighboring clusters to improve quality. Both strategies make use of the previously introduced modularity functions in between iterations to ensure an increase in quality. Additionally, both are deliberately chosen to support efficient parallel execution, enabling scalability to large hypergraphs.

4.2.1 Clustering using Flow-Based Cuts

Our clustering approach builds on the Natural Cuts method introduced by Delling et al. [15] for graph partitioning. This technique takes advantage of the property that maximum-flow-minimum-cut algorithms can effectively identify clusters by splitting networks where the smallest possible sum of edge weights is being cut, which is precisely the criterion we seek in high-quality clusterings. To apply this strategy to hypergraphs, we iteratively process subgraphs of a predetermined size by transforming them into flow networks using star expansion (Section 2.4.2), computing their minimum cuts, and merging the resulting partitions to form clusters. Initially, all vertices are assigned to the same cluster. Connected components are identified before the algorithm begins. A summary of the steps of this algorithm is provided in Algorithm 1.

Input Parameters. Our input is a hypergraph $H = (V, E, \omega)$. Before running the algorithm, we specify three parameters:

- $\alpha \in (0, 1]$: The quantity $\alpha|V(H)|$ defines an upper bound on the number of vertices $|V(H'_i)|$ in the subgraphs $H'_i = (V, E, \omega)$ that we will sample. If the connected component is smaller, then $|V(H'_i)|$ is simply the size of that component. When $\alpha = 1$, the entire hypergraph (or component) will be sampled.
- $\beta \in (0, \frac{1}{2})$: The value $\beta|V(H'_i)|$ denotes the size of the so-called core, which will later be contracted into into the source vertex, and ring, which will later be contracted into the sink vertex. The constraint $\beta < \frac{1}{2}$ ensures that the resulting flow network

is functional, i.e., it contains at least one vertex aside from the source and sink. Moreover, $\beta > 0$ guarantees the existence of both a source and a sink vertex.

- $\sigma \in (0, 1]$: This is a stopping criterion which will be checked at the beginning of each iteration. The algorithm terminates once the number of vertices that have been included in a core at any point during execution reaches at least $\sigma|V(H)|$.

For each sampled subgraph H'_i, we construct a flow network $F_i = (V, E, \omega)$ with the following vertex bound:

$$|V(F_i)| \leq (1 - \beta)|V(H'_i)| + 2 + |E(V(H'_i))|,$$

where F_i includes all vertices not in the core or ring, one source vertex, one sink vertex, and at most one additional vertex per hyperedge in H'_i, resulting from the hypergraph-to-network transformation via star expansion.

Construction of the Flow Network. Each iteration i begins by randomly selecting a vertex v that has not yet been included in a core during past iterations, ensuring that every vertex lies inside of a found cluster at least once. We build H'_i starting at v via breadth-first search (BFS) to explore the surrounding component until either $\alpha|V(H)|$ vertices or the entire component have been discovered, depending on what is smaller. During this traversal, we keep track of the number of visited vertices and identify two subsets to be contracted into source and sink later to create a flow network: The first $\beta|V(H'_i)|$ vertices discovered by BFS form the core, the final $\beta|V(H'_i)|$ vertices form the ring. During the exploration of the hypergraph, we simultaneously contract core and ring into one single vertex each called source and sink respectively. This vertex contraction process consists of replacing a set of vertices with one new vertex while preserving all edges incident to the former set. Parallel edges (i. e., multiple edges containing exactly the same vertices) might occur; in that case, they are simply merged into a single edge by summing up their weights. This contraction is necessary in larger hypergraphs with thousands of vertices and edges, since selecting individual vertices as source and sink would often result in trivial cuts where only the sink or source is separated from the rest of the network. By contracting groups of vertices instead, we can force the cut to split the graph more meaningfully, being able to use it for finding a high quality clustering.

Figure 4.1 visualizes this step: the overall gray region denotes the hypergraph, while the darker gray zones highlight the core (merged into source vertex s) and ring (merged into sink vertex t). The orange dashed line illustrates the minimum cut separating these regions.

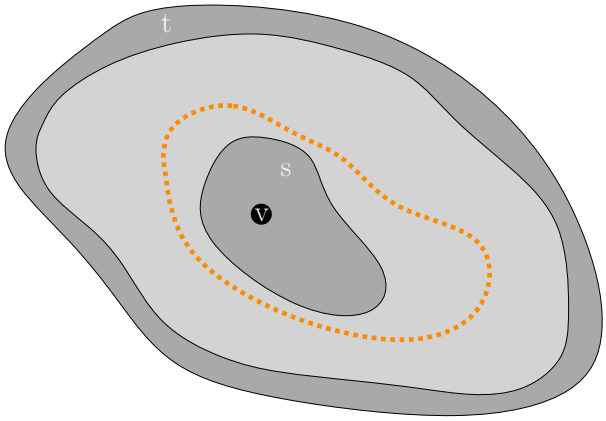


Figure 4.1: high level representation of one iteration of the Natural Cuts algorithm

Computation of the Minimum Cut. Before computing the minimum cut, the contracted hypergraph must be transformed into a graph. For this purpose, we apply the star expansion technique as introduced in Section 2.4.2. The resulting graph uses edge weights as capacities and interprets every undirected edge as a pair of directed edges, one in each direction, both carrying the same weight equal to the original edge weight. These weights are then interpreted as capacities in the flow network, enabling the use of classical max-flow algorithms to split the vertices into two sets S, T such that edges with the least weight possible lie in between these sets.

In our implementation tested in this paper, Dinitz's algorithm [18] is used, which works by repeatedly computing a blocking flow on the residual network until no longer possible. However, this partitioning is constrained by the core and ring: the core must lie on one side of the cut, and the ring on the other, which may prevent the discovery of a globally optimal split.

Repetition and Aggregation. To construct a complete clustering of the hypergraph, the min-cut procedure is repeated across different regions, which is ensured by randomly choosing the starting vertex form all vertices that have not yet been part of a core. The iterations continue until the stopping criterion is met: all but a fraction of vertices $(1 - \sigma)|V(H)|$ must have been contracted into a source vertex in at least one iteration, meaning they have been inside of a cut at least once. Lower values of σ reduce the number of iterations, but clustering quality might suffer.

Instead of using the parameter σ to control the number of iterations, we also consider an alternative stopping criterion based on the number of consecutive unsuccessful clustering attempts, i. e., min-cut computations that do not lead to an improvement in the overall clustering quality. This strategy offers greater flexibility: If many high-quality cuts are found, the algorithm continues for more iterations, whereas a fixed threshold σ might prematurely terminate the process despite potential for further improvement.

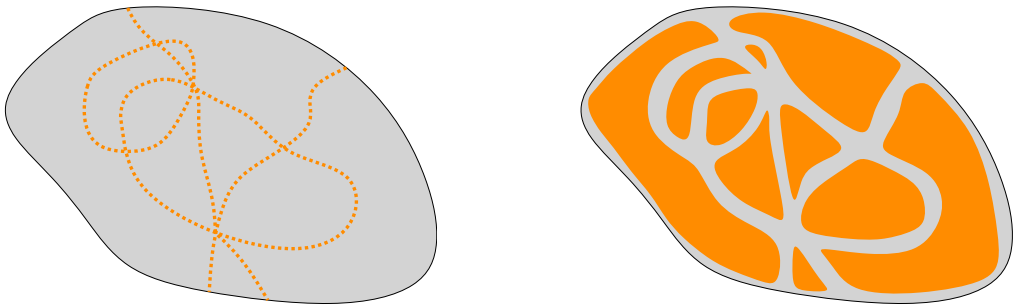


Figure 4.2: After several minimum cut computations, cuts are used to form clusters

However, this alternative approach does not guarantee convergence to an optimal clustering. After a certain number of non-improving iterations, it is still possible that better cuts remain undiscovered. Nevertheless, this heuristic improves computational efficiency with only a marginal loss in clustering quality (shown in experiments).

Once the iterations are finished, the accumulated cuts are combined into a final clustering of the original hypergraph. This step is illustrated in Figure 4.2.

Improving Quality: Applying Modularity Functions. Since the algorithm described above does not necessarily find optimal cuts for the sampled subgraphs due to the predetermined choice of source and sink vertices as well as the core and ring sizes, applying a newly computed cut (S, T) to the clustering might reduce overall hypergarph clustering quality. To avoid this, we evaluate the hypothetical change in modularity at the end of each iteration and only apply the cut if it results in an improvement. We make use of the previously introduced hypergraph modularity gain functions $\Delta Q_1(\mathcal{C}, S)$, where we first apply clique reduction from Section 2.4.1 to transform the hypergraph into a regular graph, and $\Delta Q_2(\mathcal{C}, S)$. These allow for an efficient evaluation of the modularity change and ensure that the algorithm maintains or improves the overall clustering quality.

Improving Quality: Comparing Results for Multiple Input Parameters. Furthermore, the quality of the resulting clustering can be improved by varying the parameters α and β across iterations rather than fixing them throughout the entire algorithm. Specifically, for each starting vertex , we may perform multiple iterations using different combinations of α and β. Each resulting cut is evaluated using one of the modularity functions $\Delta Q_1(\mathcal{C}, S)$ and $\Delta Q_2(\mathcal{C}, S)$, and only the best outcome, i. e., the cut that yields the highest modularity gain, is applied. If no parameter setting leads to a positive modularity gain, then no update is made in that iteration. Since different vertices may benefit from different parameter choices, this additional computational effort may be justified by the corresponding improvements in clustering quality.

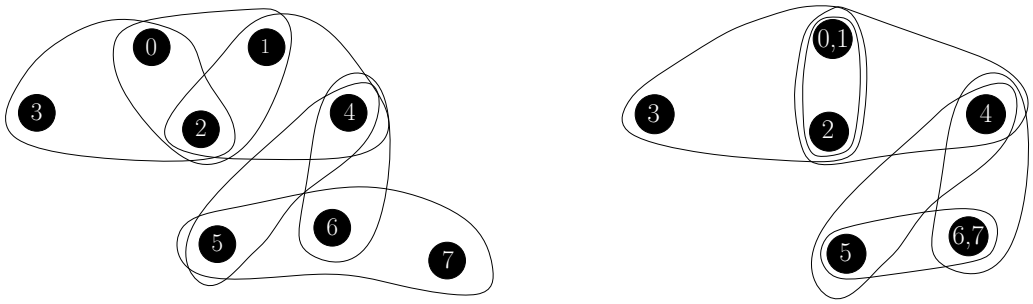


Figure 4.3: Sampling and contraction of core and ring

Example. Consider a hypergraph with eight vertices and input parameters $\alpha = 1$ $\beta = \frac{1}{4}$, meaning that the sample size equals $\alpha|V(H)| = |V(H)| = 8$, and the core and ring each have size $\beta(\text{sample size}) = 2$. Suppose we select vertex 0 as the starting point. Performing a breadth-first search, we track the number of visited vertices and contract the first two discovered vertices (core) consisting of vertices $\{0, 1\}$ into s and the last two discovered vertices (ring) consisting of vertices $\{6, 7\}$ into t (Figure 4.3) To compute a minimum cut, we transform the hypergraph into a graph using star expansion technique, e. g., replacing the hyperedge $\{s, 2, 3\}$ with auxiliary vertex 8 and three edges $\{s, 8\}, \{2, 8\}, \{3, 8\}$, while hyperedge $\{5, t\}$ can simply be replaced by an edge, since they are equivalent. After having constructed this flow network with designated source and sink vertices, classical minimum cut algorithms such as Dinitz's [18] or Ford-Fulkerson's[22] can be applied. In this example, since all edges are unweighted, the minimum cut is straightforward to identify. Assuming this is the first iteration of the algorithm, this leaves us with two new clusters, splitting the old cluster which contained the entire graph along the minimum cut. (Figure 4.4)

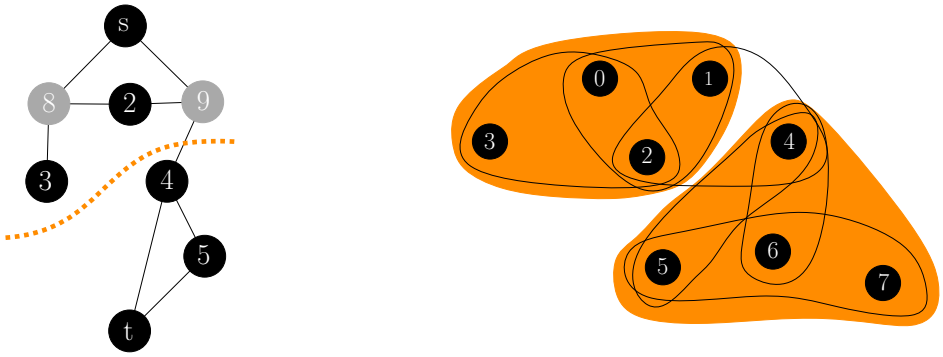


Figure 4.4: Computation and application of the minimum cut, gray vertices are auxiliary vertices resulting from star expansion 2.4.2, min cut and clusters are indicated by orange line and background

Algorithm 1 Clustering using Flow-Based Cuts

Function `MinCut` (H, v, α, β)**:**
- $H'_i \leftarrow \text{BFS}(H, v, \alpha|V|)$
- $s \leftarrow$ contract first $\lfloor \beta|V(H'_i)| \rfloor$ in V'
- $t \leftarrow$ contract last $\lfloor \beta|V(H'_i)| \rfloor$ in V'
- $F \leftarrow$ star expansion of H'_i with s, t
- **return** Dinitz(F, s, t)

Function `FlowCuts` (H, α-*set*, β-*set*, σ, Q_i)**:**
- Initialize clustering $\mathcal{C}$ with $C(v) = 0 \; \forall v \in V$
- Initialize $clusQual \leftarrow 0$
- Mark all vertices as `not in core`
- **while** *less than* $\sigma|V(H)|$ *vertices have been part of a core* **do**
  - Select $v \in V(H)$ uniformly at random such that v is not yet in any core
  - Initialize $bestCut \leftarrow$ `null`, $bestGain \leftarrow 0$
  - **foreach** $\alpha \in \alpha$-*set* **do**
    - **foreach** $\beta \in \beta$-*set* **do**
      - $cut \leftarrow$ `MinCut` (H, v, α, β)
      - Compute modularity gain ΔQ_i for cut
      - **if** $\Delta Q_i > bestGain$ **then**
        - $bestGain \leftarrow \Delta Q_i$
        - $bestCut \leftarrow cut$
      - **end**
    - **end**
  - **end**
  - **if** $bestCut \neq$ `null` **then**
    - Update $\mathcal{C}$ with $bestCut$
    - $clusQual \leftarrow clusQual + bestGain$
    - Mark vertices in core of this iteration as `in core`
  - **end**
- **end**
- **return** $\mathcal{C}$

4.2.2 Greedy Refinements

To further improve the clustering results obtained by the minimum cuts method described above, we propose a simple greedy improvement procedure. As the minimum cut computations are restricted by fixed core and ring sizes and predetermined source and sink vertices, the resulting clustering is likely not globally optimal, failing to maximize modularity. Additionally, by merging a new cluster in the already existing clustering, no small exact adjustments are possible, instead a larger number of vertices changes cluster at once during each iteration. This makes it suitable for identifying coarse partitions but neglects improvements possible through subtle changes. Hence, applying a greedy refinement procedure to the initial clustering results is a logical step.

Algorithm 2 Greedy Refinements

Function `ImproveModularity(`$H, \mathcal{C}, clusQual, Q_i$`)`**:**
- Initialize $improved \leftarrow$ `true`
- **while** $improved$ **do**
  - $improved \leftarrow$ `false`
  - **foreach** $v \in V(H)$ **do**
    - $C_{\text{old}} \leftarrow \mathcal{C}(v)$
    - $best\_gain \leftarrow 0$, $best\_cluster \leftarrow C_{\text{old}}$
    - **foreach** $C \in \mathcal{N}_{\mathcal{C}}(v)$ **do**
      - **if** $C \neq C_{old}$ **then**
        - $gain \leftarrow \Delta Q_i(v, C_{\text{old}} \rightarrow C)$
        - **if** $gain > bestGain$ **then**
          - $bestGain \leftarrow gain$
          - $bestCluster \leftarrow C$
        - **end**
      - **end**
    - **end**
    - **if** $best\_cluster \neq C_{old}$ **then**
      - move v from C_{old} to $bestCluster$
      - $improved \leftarrow$ `true`
      - $clusQual \leftarrow clusQual + bestGain$
    - **end**
  - **end**
- **end**
- **return** $\mathcal{C}$

Strategy. We start with the initial clustering computed by the min-cut-based approach, iterate over all vertices $v \in V$, and consider all clusters $C \in \mathcal{N}_{\mathcal{C}}(v)$, where $\mathcal{N}_{\mathcal{C}}(v)$ denotes the set of clusters that contain at least one neighbor of v. For each such candidate cluster C, we evaluate the modularity gain obtained by removing v from its current cluster

$C(v)$ and inserting it into C. Among all candidate clusters, vertex v is reassigned to the cluster C that yields the maximum positive gain in modularity. If the gain in modularity is negative for all possible clusters, v remains in its old cluster. This local optimization process is repeated iteratively until no further improvement is possible (strict version), or until only negligible improvements remain (efficient version), indicating that a local optimum has been reached. The modularity gain resulting from moving a vertex from one cluster to another is computed using one of the previously introduced modularity functions, either $\Delta Q_1(v, C_{\text{old}} \to C_{\text{new}})$ or $\Delta Q_2(v, C_{\text{old}} \to C_{\text{new}})$. This strategy is summarized in Algorithm 2.

4.3 Parallelization

Both parts of our algorithm, the initial coarse clustering phase using minimum cuts and the greedy refinement phase, consist of repeating the same series of operations applied to individual vertices. These operations are largely independent except for shared quality and clustering adjustments. This inherent structure makes the algorithm particularly well-suited for parallelization.

4.3.1 Parallelizing Flow-Based Clustering

To make efficient parallelization of the Flow-Based strategy from Section 4.2.1 possible, we need to slightly modify the original iteration criterion of the Natural Cuts approach, which restricts the selection to vertices that have not yet been part of a core. In the original sequential setting, this condition can be enforced easily; however, in a parallel setting, it becomes problematic: Determining whether a vertex belongs to a core depends on building the flow network during each iteration and cannot be computed ahead of time. Every thread would need to wait for all earlier threads to finish parts of their computation which would significantly decrease parallelization efficiency.

To avoid this, we adopt a simpler and more parallel-friendly strategy. Before starting the parallel computation, we shuffle the list of all vertex ids in the hypergraph and assign iterations to threads based on this predefined order. Each thread processes vertices from the shuffled list independently and stops once no further improvements are found. This approach determines which thread processes which vertex before the actual computation begins and eliminates the need for generating random variables during parallel execution. As a result, accidentally processing the same vertex with the same parameters can be avoided.

In the following, we first introduce and compare three parallelization strategies for the minimum cut phase of the algorithm. The first is a strictly deterministic approach preserving the exact behavior and results of the sequential version but offering limited parallel efficiency. The second is a fully asynchronous version that prioritizes parallel performance at the cost of potentially reduced clustering quality. The third is a mix of both, aiming to balance correctness and efficiency by selectively synchronizing key operations.

All strategies naturally parallelize the iterations consisting of the sampling of a sub-hypergraph, the construction of the corresponding flow network, and the computation of the minimum cut. These iterations can originate from different starting vertices, or in some cases even from the same one, when multiple parameter configurations are evaluated per vertex as outlined in Section 4.2.1.

Deterministic Strategy. In this strategy, parallelization is applied only in completely independent sections to preserve the sequential results of the original algorithm. For different parameter settings associated with the same starting vertex, the corresponding minimum cut and modularity gain computations are performed in parallel. Once all threads finish, a single thread compares the results, selects the cut yielding the highest gain in modularity and applies the changes, while all other threads wait.

In the case of different starting vertices, modularity gain evaluations are conducted sequentially: After all threads finish their minimum cut computation, a single thread iteratively calculates the gain the first optional cluster change and, if beneficial, applies it before continuing to the next vertex.

This strategy guarantees the same results as if executed using only one thread. It avoids the risk of decreasing quality due to concurrent updates. Specifically, it ensures that a change that initially appears beneficial does not later result in a net loss of modularity due to subsequent, concurrently applied changes.

The downside is however, that most threads spend a lot of time waiting, resulting in very limited parallel efficiency. This strategy is summarized in Algorithm 3.

Algorithm 3 Flow-Based Clustering: Deterministic Parallel Strategy

Function `DeterministicParallelClustering(`H, α-*set*, β-*set*, Q_i`)`**:**
- **foreach** *vertex* v ***in parallel*** **do**
  - **foreach** *pair* $(\alpha, \beta) \in \alpha$-*set* $\times \beta$-*set* **do**
    - $cut_{v,\alpha,\beta} \leftarrow$ `MinCut(`H, v, α, β`)`
  - **end**
  - wait until released (designated thread has finished loop below sequentially)
  - **foreach** *vertex* v *with params* (α, β) *already processed* **do**
    - $gain_{v,\alpha,\beta} \leftarrow$ modularity gain ΔQ_i for $cut_{v,\alpha,\beta}$
    - `SelectAndApplyBestChange(`$\{cut_{v,\alpha,\beta}\}$`)`
  - **end**
- **end**

Asynchronous Strategy. The asynchronous strategy prioritizes speed and maximal parallel utilization. All computations (minimum cuts, modularity gain evaluations, and

clustering updates) are performed independently and in parallel. For the same vertex with multiple parameter settings, this does not make much of a difference, but for different starting vertices, changes with positive modularity gain are applied immediately by each thread.

While this approach ensures minimal waiting between threads, it accepts incorrect gain computations. Concurrent threads may operate on outdated versions of the clustering: For instance, it is possible that while thread 1 is in the middle of calculating modularity gain, thread 2 is adjusting clusters, resulting in thread 1 computing modularity gain based on a clustering that changes halfway through. This can result in applying outdated or even counterproductive updates, ultimately reducing clustering quality. This strategy is summarized in Algorithm 4.

Algorithm 4 Flow-Based Clustering: Asynchronous Parallel Strategy

Function `AsynchronousParallelClustering(`H, α-*set*, β-*set*, Q_i`)`**:**
- **foreach** *vertex* v ***in parallel*** **do**
  - **foreach** *pair* $(\alpha, \beta) \in \alpha$-*set* $\times \beta$-*set* **do**
    - $cut \leftarrow$ `MinCut(`H, v, α, β`)`
    - $gain \leftarrow$ modularity gain ΔQ_i for $cut_{v,\alpha,\beta}$
  - **end**
  - apply best modularity change per vertex immediately in parallel, no waiting
  - `SelectAndApplyBestChange(`$\{cut_{v,\alpha,\beta}\}$`)`
- **end**

Combined Strategy. The hybrid approach aims to balance correctness and efficiency. For different parameter settings of the same vertex, modularity gain computations are performed in parallel. Once completed, a single thread selects and applies the best cut while others wait. For different starting vertices, minimum cuts and modularity gain evaluations are also computed in parallel, but all updates to the clustering are applied sequentially afterward. All threads wait until this update step is complete before proceeding. This strategy allows for substantial parallelism in the most time-consuming part (minimum cut and modularity gain computation) while ensuring that updates are applied in a coordinated, consistent manner. There remains a risk that the application of one cut might render another cut suboptimal or invalid since gains are computed before any updates are applied However, this risk is generally small, especially when only a moderate number of threads operate on different starting vertices simultaneously, and parameter α dictating the flow network size, and thus, maximum cluster size, is relatively small. In summary, we allow a slight potential loss in clustering quality in exchange for significantly improved efficiency compared to the fully deterministic strategy. This strategy is summarized in Algorithm 5.

Algorithm 5 Flow-Based Clustering: Combined Parallel Strategy

Function `CombinedParallelClustering(`H, α-*set*, β-*set*, Q_i`)`**:**
  foreach *vertex* v ***in parallel*** **do**
    foreach *pair* $(\alpha, \beta) \in \alpha$-*set* $\times \beta$-*set* **do**
      $cut_{v,\alpha,\beta} \leftarrow$ `MinCut(`H, v, α, β`)`
      $gain_{v,\alpha,\beta} \leftarrow$ modularity gain ΔQ_i for $cut_{v,\alpha,\beta}$
    end
    wait until all changes have been applied sequentially
    `SelectAndApplyBestChange(`$\{cut_{v,\alpha,\beta}\}$`)`
  end

4.3.2 Parallelizing Greedy Refinements

In this section, we briefly discuss the simple parallelization technique of the greedy refinement algorithm. We could again consider different parallelization strategies similar to those used in the flow-based clustering algorithm. However, in this case, such strategies have limited impact. The greedy algorithm makes only minor changes by moving one vertex at a time to a neighboring cluster, rather than constructing entirely new clusters potentially involving up to two-thirds of all graph vertices and affecting all clusters. Because of this, delaying the application of cluster changes until after computing the modularity gain for other vertices barely affects the modularity outcome. Therefore, we parallelize the algorithm by performing all iterations in parallel, without any stopping points: Each thread computes the best modularity gain for a single vertex across all candidate clusters. After evaluating all vertices, the changes are applied sequentially and modularity is updated. This process can be repeated until no further improvement is possible, i. e., a local maximum is reached. This strategy is summarized in Algorithm 6.

Algorithm 6 Greedy Refinement: Parallel Strategy

Function `GreedyRefinement(`H, Q_i`)`**:**
  repeat
    foreach *vertex* $v \in V$ ***in parallel*** **do**
      $(C_v^*, \Delta Q_v) \leftarrow$ `EvaluateModularityGain(`v, C, Q_i`)`
    end
    `ApplyMovesSequentially(`$\{(v, C_v^*, \Delta Q_I)\}$`)`
  until *no improvement in modularity*

CHAPTER 5

Experimental Evaluation

This section presents the experimental evaluation of the algorithms developed and described in the previous chapters. We begin by presenting the evaluation methodology and implementation details, and introduce the hypergraph instances used. Next, we present our experimental results, starting with tuning experiments to identify the optimal parameters for our flow-based clustering algorithm. We then show the efficiency of our parallelization for both the coarser min-cut clustering phase and the greedy refinement phase. Finally, we compare our algorithm to state-of-the-art clustering frameworks.

5.1 Methodology

The algorithms described in Section 4 are implemented in C++. All experiments were carried out on an Intel Xeon Silver 4216 16-Core Processor running at 2.10 GHz equipped with 93 GB. We are using performance profiles [19] to plot our results. They visualize for which fraction of instances a clustering with quality $\geq \tau \cdot Q_{\text{best}}$ is computed by the respective methods, where Q_{best} denotes the highest modularity obtained across all methods. Similarily, they display for which fraction of instances the algorithm terminates in $\tau \cdot T_{\text{best}}$, with T_{best} being the lowest running time observed.

5.1.1 Parallelization

For the parallel implementation of our algorithm, we use OpenMP (Open Multi-Processing), an API (Application Programming Interface) that supports multi-platform shared-memory parallel programming in C++ among others. OpenMP allows the programmer to specify parallel regions in the code using simple compiler instructions, eliminating the need for manual thread management while offering control over concurrency and synchronization when needed. It is particularly useful for our use case because it enables efficient parallel execution of loops and independent task such as the repeated vertex-wise

operations in both phases of our algorithm on multi-core CPUs. Furthermore, OpenMP can be easily integrated into existing sequential code, so complete restructuring of the algorithm is not necessary.

5.1.2 Instances

In Section 5.2 and Section 5.4, we use hypergraph instances that are constructed from incident matrices obtained from the suite sparse collection by Davis and Hu [14]. They gather sparse matrices from real-world applications across various domains, including structural engineering, circuit simulation, computational fluid dynamics, and optimization. These matrices are contributed by researchers and practitioners and are not synthetically generated.

In Section 5.3, we use five hypergraphs 5.1 from the collection provided by Gottesbüren et al. [25] containing general hypergraphs.

Hypergraph	#Edges	#Vertices
astro-ph.mtx.hgr	16,046	16,706
av41092.mtx.hgr	41,092	41,092
bibd_49_3.mtx.hgr	1,176	18,424
bips07_1998.mtx.hgr	15,066	15,066
fd18.mtx.hgr	16,428	16,428

Table 5.1: Hypergraph instances from [25]

5.2 Optimizing the Flow-based Clustering Algorithm

We optimize the quality and running time of our algorithm by systematically varying the sample size (α) and core size (β) used during the maximum flow phase of our algorithm. Additionally, we evaluate the change in performance when applying small algorithmic variations such as two different stopping criteria for min-cut iterations and two options for selecting vertices.

5.2.1 Comparing Input Parameter Variations

We present a comprehensive set of performance profiles to identify the best overall choices for α and β, as well as strong alternatives. These runner-up configurations allow us to assign multiple parameter variations to each vertex and select the best one individually. In Figures 5.1, 5.2, 5.3, 5.5 and 5.6, we use the graph modularity measure Q_1 (see Section 4.1.1) to evaluate clustering quality. In Figure 4.1.2), we instead apply the hypergraph modularity Q_2.

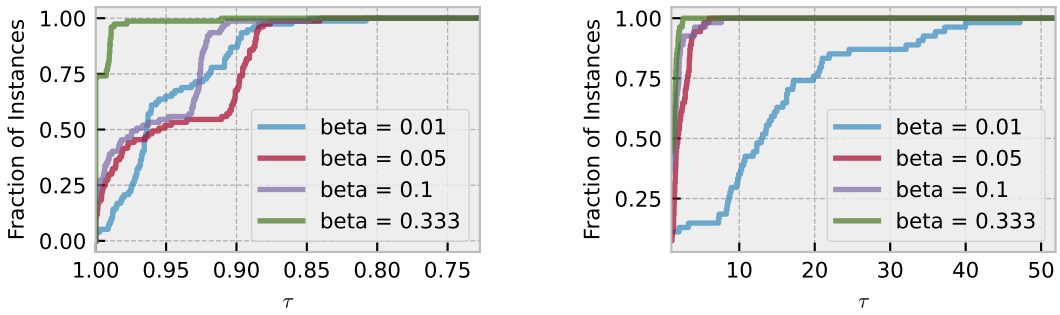
Fixed α, Varying β. The performance profiles in Figures 5.1, 5.2 and 5.3 show how different values of β perform for fixed values of α. Across all experiments, we use $\beta \in \{0.01, 0.05, 0.1, 0.333\}$, with 0.333 being the highest possible choice for beta that ensures the number of core vertices does not exceed that of the flow network vertices. Hypergraph sample size α is set to 1, 0.5, and 0.1 in Figures 5.1, 5.2, and 5.3, respectively.

In all three cases, $\beta = 0.333$ consistently outperforms smaller values in both clustering quality and runtime. The quality gain likely stems from a larger core reducing the chance of trivial cuts, while the runtime improvement results from fewer vertices in the flow graph, thus reducing computational overhead. Given that $\beta = 0.1$ is the next-best option, it may be worthwhile to explore additional values between 0.1 and 0.333 for further tuning.

Performance profiles in Figure 5.4 present results using the same input parameters as in Figures 5.1 and 5.2, but this time the decisions are based on the hypergraph modularity measure Q_2. We omit running time plots here, as they closely match those shown previously.

Interestingly, the outcomes differ significantly: lower values of β lead to higher modularity scores, with the smallest tested value, $\beta = 0.01$, outperforming all others. A smaller β typically results in smaller average cluster sizes, since each new cluster must at least contain all core vertices. Fewer core vertices allow the algorithm to form smaller clusters without requiring a split.

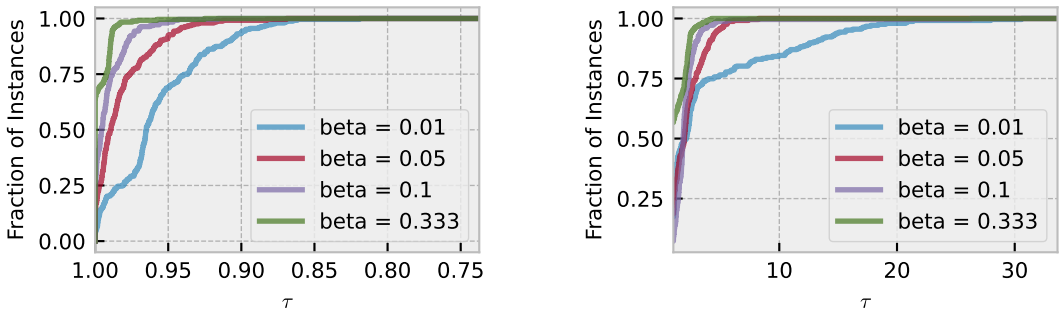
These results suggest that Q_2 favors smaller clusters, in contrast to Q_1, which tends to reward larger cluster formations.



(a) graph modularity performance profile

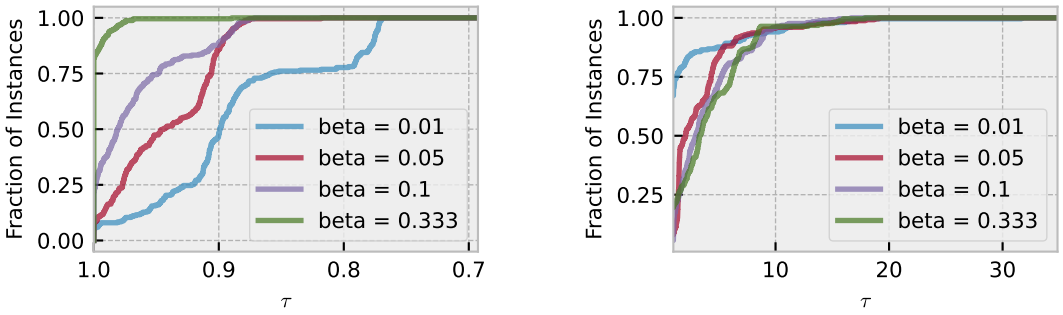
(b) running time performance profile

Figure 5.1: Hypergraph sample size parameter $\alpha = 1$



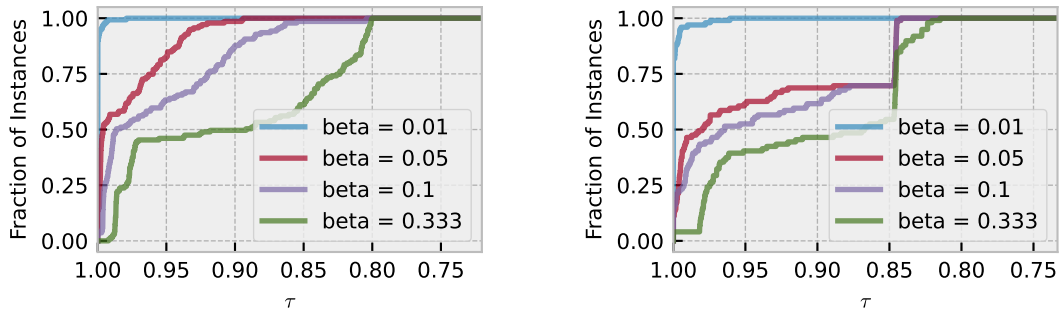
(a) graph modularity performance profile **(b)** running time performance profile

Figure 5.2: Hypergraph sample size parameter $\alpha = 0.5$



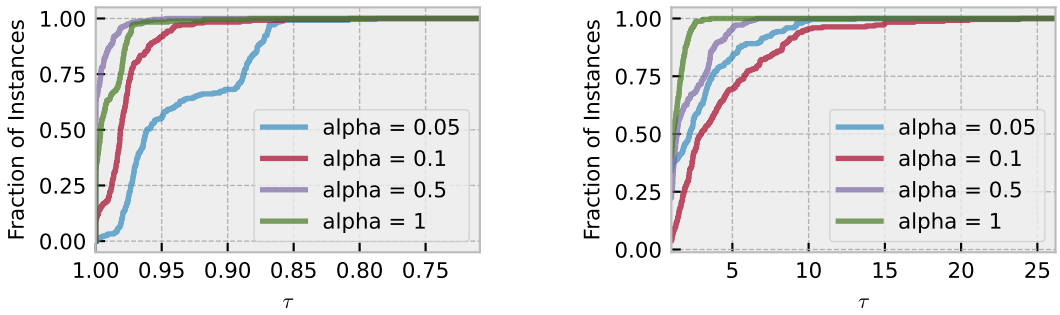
(a) graph modularity performance profile **(b)** running time performance profile

Figure 5.3: Hypergraph sample size parameter $\alpha = 0.1$



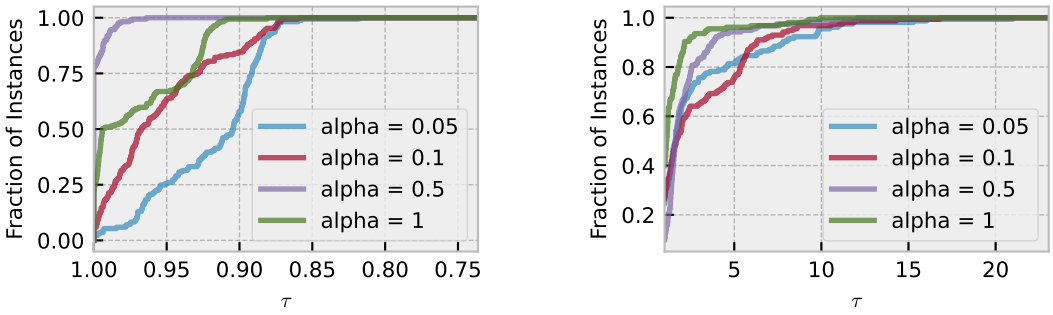
(a) hypergraph modularity performance profile **(b)** hypergraph modularity performance profile

Figure 5.4: Hypergraph sample size parameter (a) $\alpha = 1$, (b) $\alpha = 0.5$



(a) graph modularity performance profile **(b)** running time performance profile

Figure 5.5: Core and ring size parameter $\beta = 0.333$



(a) graph modularity performance profile **(b)** running time performance profile

Figure 5.6: Core and ring size parameter $\beta = 0.1$

Fixed β, Varying α. In Figures 5.5 and 5.6 we repeat the evaluation with fixed values of β, specifically 0.333 in 5.5 and 0.1 in 5.6, while varying $\alpha \in \{0.05, 0.1, 0.5, 1\}$. In both cases, we observe that $\alpha = 0.5$ achieves the best overall clustering quality.

In terms of running time, the highest value α performs best, likely because a larger sample of the hypergraph results in more vertices being marked as already included in a core early on, reducing the number of required iterations. However, the runtime difference between $\alpha = 0.5$ and $\alpha = 1$ is relatively small, making $\alpha = 0.5$ a worthwhile trade-off for its superior quality.

It may be beneficial to explore additional values of α between 0.1 and 1, as values lower than 0.1 yield both poorer quality and longer runtimes compared to the others.

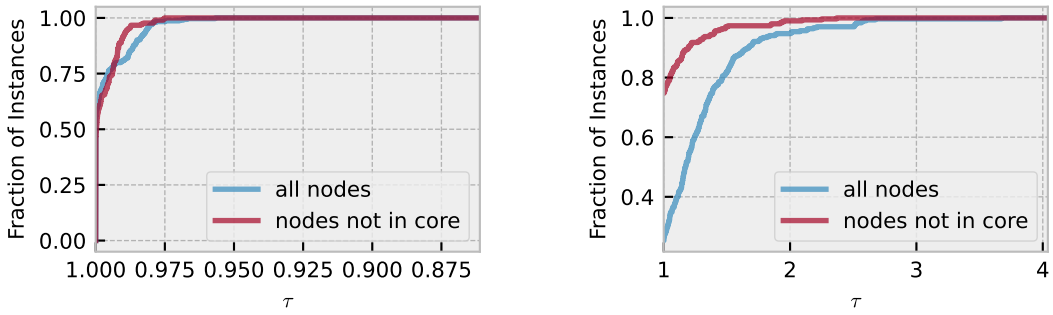
5.2.2 Analyzing Vertex Selection and Stopping Criterion Strategies

For our parallelization strategy, we proposed selecting vertices not based on whether they have already been part of a core, but instead by choosing randomly from those that have not yet been used as a starting vertex for sampling the hypergraph. Figure 5.7 confirms the adequacy of this modification: although running time increases, the clustering quality remains nearly unchanged.

To further improve efficiency, we explore alternative stopping criteria, as waiting until every vertex has been part of a core is inefficient. Figures 5.8 and 5.9 demonstrate that simply lowering the threshold ω (i. e., allowing more vertices to be skipped) significantly reduces runtime, but leads to unacceptable losses in clustering quality.

A more promising approach is to monitor the number of consecutive iterations without improvement in modularity, and terminate once a given threshold is reached. Figure 5.10 shows that this method achieves substantial speed-ups with only minimal loss in quality.

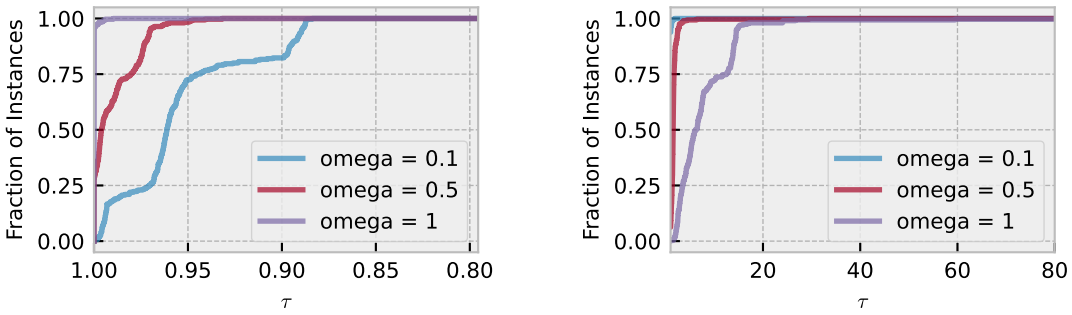
Finally, Figures 5.11 and 5.12 illustrate that after a certain number of non-improving iterations, further gains in quality are negligible. When using the original vertex selection strategy (only vertices not yet in a core), terminating after 10 non-improving iterations appears sufficient. Extending this to 100 only increases quality marginally (Figure 5.11). In the alternative selection strategy (choosing from vertices not yet used as a starting point), a slightly higher threshold is advisable, as shown in Figure 5.11. This difference likely stems from the more uniform vertex coverage in the former case, whereas the latter may select neighboring vertices in close succession.



(a) graph modularity performance profile

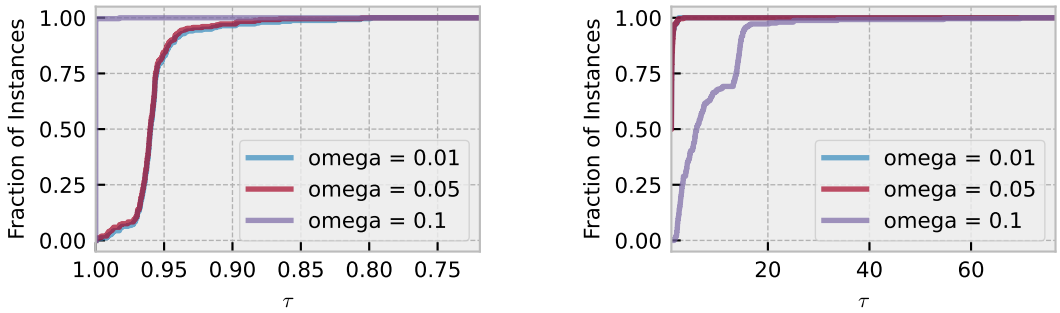
(b) running time performance profile

Figure 5.7: choosing a vertex that has not been a starting vertex yet vs. choosing a vertex that has not been in a core yet (terminating after 10 consecutive iterations without quality improvement)



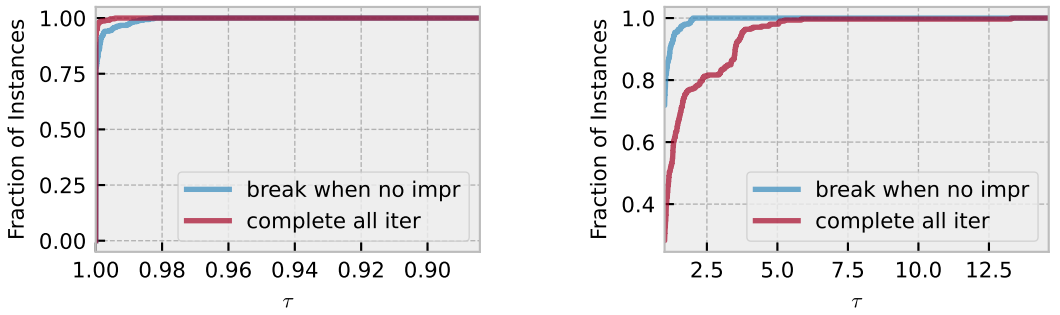
(a) graph modularity performance profile **(b)** running time performance profile

Figure 5.8: Terminating when $< \omega|V|$ vertices have not been in a core yet



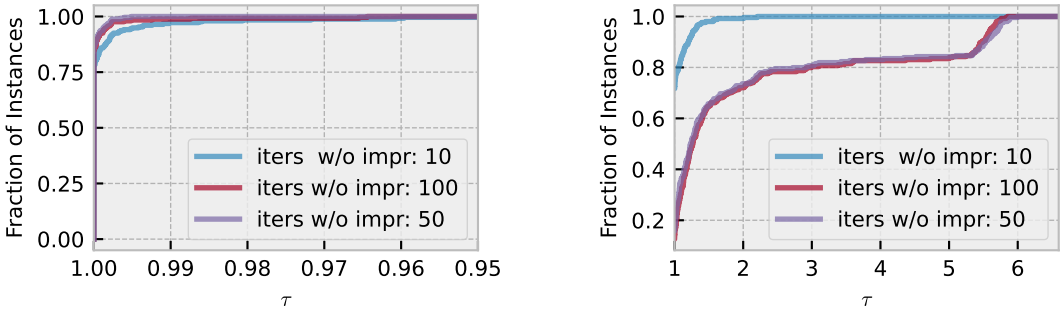
(a) graph modularity performance profile **(b)** running time performance profile

Figure 5.9: Terminating when $< \omega|V|$ vertices have not been a starting vertex yet



(a) graph modularity performance profile **(b)** running time performance profile

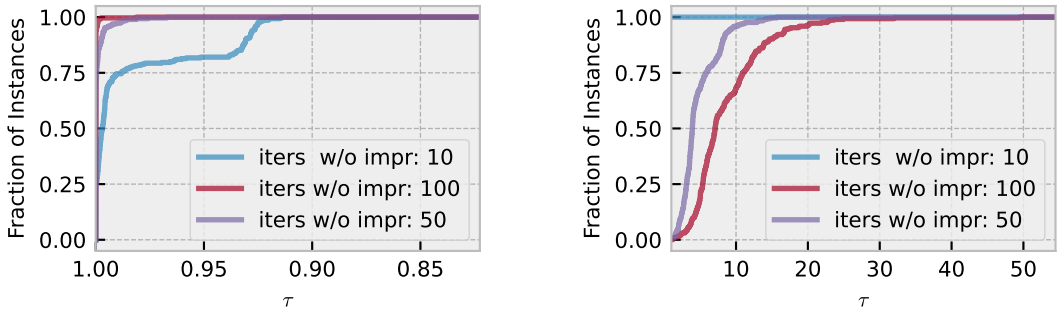
Figure 5.10: Terminating when all vertices have been a starting vertex once vs. terminating after 10 consecutive iterations without quality improvement



(a) graph modularity performance profile

(b) running time performance profile

Figure 5.11: Randomly choosing vertices from all that have not been in a core yet, terminating after varying numbers of consecutive iterations without quality improvement



(a) graph modularity performance profile

(b) running time performance profile

Figure 5.12: Randomly choosing vertices from all that have not been a starting vertex yet, terminating after varying numbers of consecutive iterations without quality improvement

5.3 Efficiency of our Parallelization

We showcase the efficiency of our parallelization technique using five hypergraph instances from [25]. Both the deterministic strategy (Section 4.3.1) and the asynchronous strategy (Section 4.3.1) exhibited major drawbacks, the former being highly inefficient, and the latter suffering from a loss in clustering quality. Therefore, we only present the results of the combined strategy (Section 4.3.1), which minimizes the downsides of the individual approaches.

For these computations, we used the best parameter settings identified in Section 5.2, namely $\alpha = 0.5$ and $\beta = 0.333$. As a stopping criterion, we terminate after a fixed number of successive iterations without quality improvement. Since we select vertices from the set of all that have not yet served as a starting vertex rather than only those not previously included in a core, we set this threshold to 30.

As shown in Table 5.2, the parallelization yields satisfactory results, with an efficiency of approximately 0.5 (and often higher) when using 16 threads, and even better for fewer threads. For up to 4 threads, the clustering quality is largely unaffected. With a higher number of threads, the risk of quality loss increases, though in some cases, even improvements are observed.

This parallelization technique offers a useful speedup. Moreover, incorporating a set of parameter choices per vertex could further reduce or eliminate the risk of quality loss.

Hypergraph	1 thread	2 threads	4 threads	8 threads	16 threads
astro-ph.mtx.hgr	67 s 0.411	53 s 0.412	14 s 0.512	8 s 0.531	5 s 0.543
av41092.mtx.hgr	523 s 0.348	310 s 0.348	203 s 0.348	141 s 0.415	109 s 0.427
bibd_49_3.mtx.hgr	23 s 0.269	12 s 0.342	7 s 0.259	4 s 0.325	3 s 0.314
bips07_1998.mtx.hgr	18 s 0.855	10 s 0.855	5 s 0.855	3 s 0.851	2 s 0.849
fd18.mtx.hgr	30 s 0.855	16 s 0.855	9 s 0.855	5 s 0.838	3 s 0.843

Table 5.2: Running time and modularity values Q_1 for various thread counts.

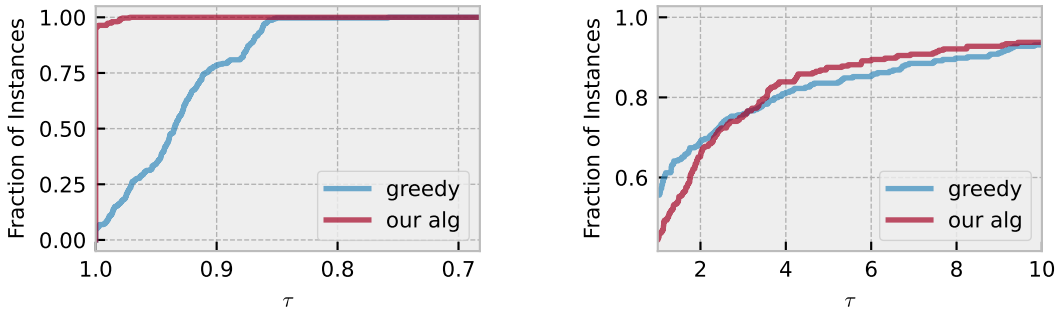
5.4 Comparison to another Clustering Strategy

The most widely used clustering strategy is the greedy modularity maximization approach, which starts by assigning each vertex to its own cluster and iteratively moves vertices to neighboring clusters whenever an increase in modularity is possible. This strategy has been applied not only to graphs but also to hypergraphs, either by transforming hypergraphs into graphs before clustering or by adapting the graph modularity function to hypergraphs.

As illustrated in Figure 5.13a, our flow-based approach using the modularity measure Q_1 (Section 4.1.1) finds higher quality clusterings in more than 90% of the cases. Moreover, for 50% of all instances, the greedy strategy finds clusterings with quality at least 10% worse. Despite these improvements, our method achieves comparable running times on average, as also shown in Figure 5.13b.

In Figure 5.14, we compare the greedy strategy to our approach using the hypergraph modularity measure Q_2 (Section 4.1.2). For the greedy strategy, we still use the modularity function Q_1, since due to its definition of θ-innercluster edges, Q_2 is not useful in this context. Therefore, it is unsurprising that our approach outperforms the greedy strategy in

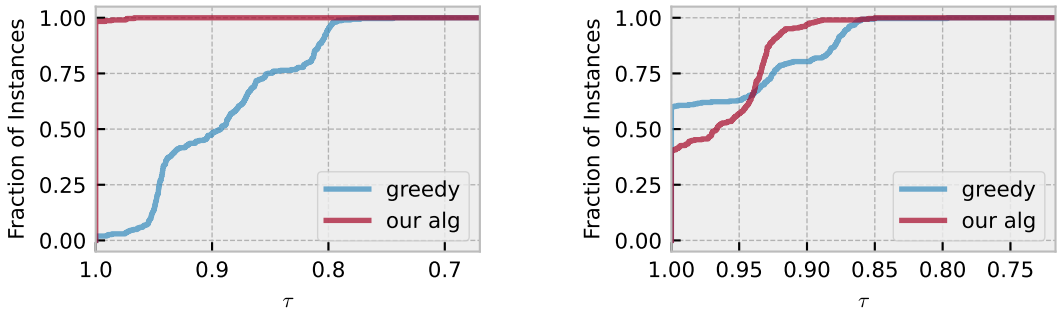
Figure 5.14a, where evaluation is based on Q_2. Conversely, in Figure 5.14b, we compare results using Q_1 as the quality metric, although our algorithm optimizes a different function. Nonetheless, our algorithm performs well in comparison, despite evidence from other experiments (Figure 5.1 and 5.4) indicating that the quality functions Q_1 and Q_2 tend to favor very different clustering structures.



(a) graph modularity performance profile

(b) running time performance profile

Figure 5.13: Comparison of our algorithm using Q_1 to the modularity maximization approach as used in Louvain [6] and Leiden [50]



(a) compare clustering results using Q_2

(b) compare clustering results using Q_1

Figure 5.14: Comparison of our algorithm using Q_2 to the modularity maximization approach as used in Louvain [6] and [50]

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we introduced a novel algorithm for clustering hypergraphs that combines a flow-based coarse clustering phase with a greedy local refinement strategy. By making use of the Max-Flow-Min-Cut theorem [22] and repeatedly solving maximum flow problems using Dinitz's algorithm [18], our approach identifies meaningful substructures that can be interpreted as clusters. These clusters are then improved through a greedy refinement strategy which finds a local optimum.

To emphasize the representational richness of hypergraphs over standard graphs, we proposed a new variation of a hypergraph modularity measure defined directly on hypergraphs [21]. We demonstrated that our adapted function satisfies the criteria for clustering quality functions, as discussed in Section 4.1, making it suitable for optimization within our algorithm.

To ensure scalability, we developed three parallelization strategies for the flow-based clustering component and an additional parallelization approach for the refinement phase. Among these, one combined technique proved particularly effective and was used in our final version. Experiments confirmed the efficiency of this parallelization.

Furthermore, we conducted a thorough evaluation of algorithmic variations, including parameter tuning for building the flow network, vertex selection strategies, and stopping criteria. These experiments allowed us to identify the most robust and high-performing configuration of our algorithm.

Overall, our method decisively outperforms the classical graph and hypergraph clustering strategy of greedy modularity maximization, both in terms of clustering quality and scalability. The presented results demonstrate the usefulness of combining flow-based techniques with refinement heuristics for effective hypergraph clustering.

6.2 Future Work

As we have seen in the experiments, there is no best choice of α and β for all hypergraphs, and not even for all vertices in the same hypergraph. While some decidedly outperform others, we still notice a clear improvement in quality when trying out an array of parameter combinations for one vertex compared to having them as fixed parameters for the entire computation. Instead of having to try out all variations for every vertex, we can focus on analyzing if there are any relations between best parameter choice and vertex or hypergraph characteristics, such as vertex degree or current state of the clustering. This would allow the algorithm to better adapt to varying hypergraph structures and avoid unnecessary computations.

Similarly, we can try to find connections between vertex properties and their potential to improve clustering quality. This can be applied in both parts of our algorithm, choosing to process vertices with high potential to improve clustering quality first.

Inspired by other hierarchical clustering methods [16, 17, 5], we could also adapt this idea to our algorithm: We start by contracting vertices based on certain similarity criteria, apply the flow-based algorithm to the coarse hypergraph, and then iteratively expand the hypergraph back into its original state while applying both the minimum cut and the greedy refinement strategy. This could possibly improve quality and efficiency through the "divide-and-conquer" approach.

In addition to trying various modularity functions, we could also test quality metrics beyond modularity such as conductance [32] or cluster path length [54] and possibly combine them with the currently used metrics.

The greedy refinement strategy could be improved by not only allowing one vertex but also a set of vertices at a time to change cluster. Especially in combination with Q_2 this might be a good idea, because this function only counts hyperedges as part of a cluster if a θ-fraction of its vertices lies in said cluster, and when only moving one vertex at a time, it is unlikely that loyalty changes, as could be seen in the experiments. This strategy might also help to prevent local maxima determined by allowing only single vertices to move.

Zusammenfassung

Das Clustern von Daten, die durch Hypergraphen dargestellt werden, gewinnt in den verschiedensten Forschungsbereichen zunehmend an Bedeutung, darunter Mustererkennung, Bioinformatik und Maschinelles Lernen.

Das Hypergraph-Clustering-Problem besteht darin, die Knotenmenge eines gegebenen Hypergraphen in eng durch Kanten verbundene Teilmengen, sogenannte Cluster, zu unterteilen, ohne dabei deren Anzahl oder Größe vorzugeben. Zur Lösung dieses Problems schlagen wir einen aus zwei Phasen bestehenden Algorithmus mit Fokus auf der Maximierung der Modularität vor. Hierfür führen wir zwei Modularitätsfunktionen ein, eine auf einem Graphen und eine direkt auf einem Hypergraphen definiert, und untermauern diese durch mehrere mathematische Beweise.

Die erste Phase unseres Verfahrens basiert auf einem Flow-orientierten Ansatz, der von dem Konzept der Natural-Cuts-Methode [15] inspiriert ist. Dabei werden sukzessive Minimum-Cuts von Teilhypergraphen berechnet, um daraus ein initiale Clustering zu konstruieren. In der zweiten Phase wird dieses Clustering mittels eines Greedy-Verfahrens verbessert, welches einzelne Knoten zwischen benachbarten Clustern verschiebt, um die Modularität des gesamten Hpergraphen zu verbessern.

Beide Phasen wurden explizit mit Blick auf parallele Ausführbarkeit entworfen; entsprechende Parallelisierungsstrategien werden vorgestellt und analysiert. Im Rahmen unserer experimentellen Evaluation untersuchen wir eine Vielzahl an Parameterkonfigurationen, Abwandlungen unseres Algorithmus sowie unterschiedliche Modularitätsfunktionen, um eine optimale Einstellung zu identifizieren. Darüber hinaus evaluieren wir die Effizienz und Qualität unserer Parallelisierung und vergleichen unseren Ansatz mit state-of-the-art Verfahren. Unsere Ergebnisse zeigen, dass unser Algorithmus in der Lage ist, konsistent Clusterings von höherer Qualität zu erzeugen.

Bibliography

[1] S. Agarwal, Jongwoo Lim, L. Zelnik-Manor, P. Perona, D. Kriegman, and S. Belongie. Beyond pairwise clustering. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 838–845 vol. 2, 2005.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] Said Baadel, Fadi Thabtah, and Joan Lu. Overlapping clustering: A review. In *2016 SAI Computing Conference (SAI)*, pages 233–237, 2016.

[4] Charles-Edmond Bichot and Patrick Siarry. *Graph Partitioning*. ISTE Ltd and John Wiley & Sons, London, UK, 2011.

[5] Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster. Vienna graph clustering. In Stefan Canzar and Francisca Rojas Ringeling, editors, *Protein-Protein Interaction Networks, Methods and Protocols*, volume 2074 of *Methods in Molecular Biology*, pages 215–231. Springer, 2020.

[6] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of community hierarchies in large networks. *CoRR*, abs/0803.0476, 2008.

[7] Samuel Rota Bulò and Marcello Pelillo. A game-theoretic approach to hypergraph clustering. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(6):1312–1327, 2013.

[8] Giulia Cencetti, Federico Battiston, Bruno Lepri, and Márton Karsai. Temporal properties of higher-order interactions in social networks, 2020.

[9] William Y. C. Chen, Andreas Dress, and Winking Q. Yu. *Detecting Community Structures in Networks Using a Linear-Programming Based Approach: a Review*, pages 1–19. Springer International Publishing, Cham, 2014.

[10] Adil Chhabra, Marcelo Fonseca Faraj, and Christian Schulz. Local motif clustering via (hyper)graph partitioning. In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pages 261–263. AAAI Press, 2022.

[11] Adil Chhabra, Marcelo Fonseca Faraj, and Christian Schulz. Faster local motif clustering via maximum flows, 2023.

[12] Calvin Chi, Yuting Ye, Bin Chen, and Haiyan. Bipartite graph-based approach for clustering of cell lines by gene expression-drug response associations. *Bioinformatics*, 37, 03 2021.

[13] Philip S. Chodrow, Nate Veldt, and Austin R. Benson. Generative hypergraph clustering: From blockmodels to modularity. *Science Advances*, 7(28):eabh1303, 2021.

[14] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[15] Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato Fonseca F. Werneck. Graph partitioning with natural cuts. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 1135–1146. IEEE, 2011.

[16] Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. Hierarchical agglomerative graph clustering in nearly-linear time, 2021.

[17] Laxman Dhulipala, Jakub Lacki, Jason Lee, and Vahab Mirrokni. Terahac: Hierarchical agglomerative clustering of trillion-edge graphs. *Proc. ACM Manag. Data*, 1(3):221:1–221:27, 2023.

[18] Yefim Dinitz. *Dinitz' Algorithm: The Original Version and Even's Version*, pages 218–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[19] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002.

[20] Zijin Feng. Modularity-based hypergraph clustering code. `https://github.com/Gavinzj/Modularity_based_Hypergraph_Clustering_code`, 2025. Accessed: 2025-05-18.

[21] Zijin Feng, Miao Qiao, and Hong Cheng. Modularity-based hypergraph clustering: Random hypergraph model, hyperedge-cluster relation, and computation. *Proc. ACM Manag. Data*, 1(3), November 2023.

[22] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[23] Chad Giusti, Robert Ghrist, and Danielle S. Bassett. Two's company, three (or more) is a simplex: Algebraic-topological tools for understanding higher-order structure in neural data, 2016.

[24] Haiyan Gong, Sichen Zhang, Xiaotong Zhang, and Yang Chen. A method for chromatin domain partitioning based on hypergraph clustering. *Computational and structural biotechnology journal*, 23:1584–1593, 04 2024.

[25] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning, 2023.

[26] S.W. Hadley, B.L. Mark, and A. Vannelli. An efficient eigenvector approach for finding netlist partitions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):885–892, 1992.

[27] Jeffrey Q. Jiang, Andreas W.M. Dress, and Genke Yang. A spectral clustering-based framework for detecting community structures in complex networks. *Applied Mathematics Letters*, 22(9):1479–1482, 2009.

[28] Bogumił Kamiński, Valérie Poulin, Paweł Prałat, Przemysław Szufel, and François Théberge. Clustering via hypergraph modularity. *PLOS ONE*, 14(11):e0224307, November 2019.

[29] Steffen Klamt, Utz-Uwe Haus, and Fabian Theis. Hypergraphs and cellular networks. *PLoS computational biology*, 5:e1000385, 06 2009.

[30] Tarun Kumar, Sankaran Vaidyanathan, Harini Ananthapadmanabhan, Srinivasan Parthasarathy, and Balaraman Ravindran. Hypergraph clustering: A modularity maximization approach. *CoRR*, abs/1812.10869, 2018.

[31] Vang Le and Vaclav Snasel. Community detection in online social network using graph embedding and hierarchical clustering. In Ajith Abraham, Sergey Kovalev, Valery Tarassov, Vaclav Snasel, and Andrey Sukhanov, editors, *Proceedings of the Third International Scientific Conference "Intelligent Information Technologies for Industry" (IITI'18)*, pages 263–272, Cham, 2019. Springer International Publishing.

[32] Longlong Lin, Ronghua Li, and Tao Jia. Scalable and effective conductance-based graph clustering. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):4471–4478, Jun. 2023.

[33] Julian Mcauley and Jure Leskovec. Discovering social circles in ego networks. *ACM Trans. Knowl. Discov. Data*, 8(1), February 2014.

[34] Aaron F. McDaid and Neil J. Hurley. Using model-based overlapping seed expansion to detect highly overlapping community structure. *CoRR*, abs/1011.1970, 2010.

[35] Tom Michoel and Bruno Nachtergaele. Alignment and integration of complex networks by hypergraph-based spectral clustering. *Physical Review E*, 86(5), November 2012.

[36] Pooja Mishra and P. N. Pandey. A graph-based clustering method applied to protein sequences. *Bioinformation*, 6:372 – 374, 2011.

[37] Federico Musciotto, Danai Papageorgiou, Federico Battiston, and Damien R. Farine. Beyond the dyad: uncovering higher-order structure within cohesive animal groups. *bioRxiv*, 2022.

[38] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, June 2006.

[39] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), February 2004.

[40] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, June 2005.

[41] Alice Patania, Giovanni Petri, and Francesco Vaccarino. The shape of collaborations. *EPJ Data Science*, 6:18, 08 2017.

[42] Giovanni Petri, Paul Expert, Federico Turkheimer, Robin Carhart-Harris, David Nutt, Peter Hellyer, and Francesco Vaccarino. Homological scaffolds of brain functional networks. *Journal of The Royal Society Interface*, 11:20140873, 12 2014.

[43] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), September 2007.

[44] Peter Sanders and Christian Schulz. Kahip v0.53 - karlsruhe high quality partitioning - user guide. *CoRR*, abs/1311.1714, 2013.

[45] Andrea Santoro, Federico Battiston, Giovanni Petri, and Enrico Amico. Higher-order organization of multivariate time series. *Nature Physics*, January 2023.

[46] Y. SarcheshmehPour, Y. Tian, L. Zhang, and A. Jung. Flow-based clustering and spectral clustering: A comparison. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, pages 1292–1296, 2021.

[47] Satu Elisa Schaeffer. Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007.

[48] Min Geun Song and Gi Tae Yeo. Analysis of the air transport network characteristics of major airports. *The Asian Journal of Shipping and Logistics*, 33(3):117–125, 2017.

[49] Tricia Thornton-Wells, Jason Moore, and Jonathan Haines. Dissecting trait heterogeneity: A comparison of the clustering methods applied to genotypic data. *BMC bioinformatics*, 7:204, 02 2006.

[50] V. A. Traag, L. Waltman, and N. J. van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1), March 2019.

[51] Fang Wei, Chen Wang, Li Ma, and Aoying Zhou. Detecting overlapping community structures in networks with global partition and local expansion. In Yanchun Zhang, Ge Yu, Elisa Bertino, and Guandong Xu, editors, *Progress in WWW Research and Development*, pages 43–55, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[52] Ying Xu, Victor Olman, and Dong Xu. Clustering gene expression data using a graph-theoretic approach: An application of minimum spanning trees. *Bioinformatics (Oxford, England)*, 18:536–45, 05 2002.

[53] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, page 974–983. ACM, July 2018.

[54] Faraz Zaidi, Guy Melançon, and Daniel Archambault. Evaluating the quality of clustering algorithms using cluster path lengths. 07 2010.

[55] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems*, volume 19. MIT Press, 2006.

[56] Anat Zimmer, Itay Katzir, Erez Dekel, Avraham E. Mayo, and Uri Alon. Prediction of multidimensional drug dose responses based on measurements of drug pairs. *Proceedings of the National Academy of Sciences*, 113(37):10442–10447, 2016.