Karlsruhe Institute of Technology

Community Detection in Hypergraphs with Application to Partitioning

Bachelor Thesis of

Robert Krause

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Peter Sanders
	PD Dr. Torsten Ueckerdt
Advisors:	M.Sc. Tobias Heuer
	M.Sc. Lars Gottesbüren

Date of Submission: June 4, 2021

Statement of Authorship

Karlsruhe, June 4, 2021

Abstract

Hypergraph partitioning is an important tool for optimizing electric circuits and improving the parallelization of sparse matrix-vector multiplications in scientific computing. The goal of hypergraph partitioning is to distribute nodes to a fixed number of blocks while maintaining a balance constraint on the block size. The most common heuristic for hypergraph partitioning is the multi-level paradigm. It consists of three phases: coarsening, initial partitioning and refinement. The coarsening phase aims to construct successively smaller and structural similar hypergraphs by contracting pairs of nodes that will likely end up in the same partition. These contractions are usually performed using rating functions that only incorporate the direct neighborhood of the node. Therefore, the recently introduced community-aware coarsening algorithm restricts contractions to densely coupled regions of the hypergraph. The current approach first transforms the hypergraph into the bipartite graph representation and then evaluates its community structure. Such a projection on a dyadic graph representation loses some of the higher-order structure we want to capture.
Therefore, in this work, we present different approaches to community detection on hypergraphs directly and evaluate them as a preprocessing step for the coarsening phase. We present a modularity maximization approach that optimizes a modularity formulation that we derived by incorporating the connectivity of hyperedges. We optimize this hypergraph modularity using the parallel Louvain method. Additionally, we present formulations for the map equation on hypergraphs based on different random walk models. We integrate our hypergraph community detection algorithms into the parallel shared-memory hypergraph partitioner `Mt-KaHyPar`. Finally, we evaluate different optimizations and heuristics we used to speed up the hypergraph modularity method and compare our different approaches as well as the current graph-based approach in extensive experiments.
In our experiments involving 488 hypergraphs from its different applications, our approach produced better partitions than the current configuration `Mt-KaHyPar-D` on 61% of the instances. However, our approach is slower by a factor of 5.4 on average. Additionally, we evaluate our algorithm on instances that are an order of magnitude larger than used in other publications.

Deutsche Zusammenfassung

Die Hypergraph-Partitionierung ist ein wichtiges Werkzeug zur Optimierung von elektrischen Schaltungen und Verbesserung der Parallelisierung von dünnbesetzten Matrix-Vektor-Multiplikationen im wissenschaftlichen Rechnen. Das Ziel der Hypergraph-Partitionierung ist die Knoten auf eine feste Anzahl an Blöcken zu verteilen, sodass die Blöcke möglichst gleich groß sind. Die wichtigste Heuristik für die Hypergraphen-Partitionierung ist das Multi-Level-Paradigma. Es besteht aus drei Phasen: der Vergröberungs-, der Partitionierungs- und Verbesserungsphase. Die Vergröberungsphase zielt darauf ab, sukzessiv kleinere und strukturell ähnliche Hypergraphen zu konstruieren, indem Knotenpaare, die wahrscheinlich in der gleichen Partition enden werden, kontrahiert werden. Diese Kontraktionen werden üblicherweise mit Bewertungsfunktionen durchgeführt, die nur die direkte Nachbarschaft des Knotens einbeziehen. Daher beschränkt der kürzlich eingeführte Community-Aware Coarsening-Algorithmus die Kontraktionen auf dicht gekoppelte Regionen des Hypergraphen. Der aktuelle Ansatz transformiert zunächst den Hypergraphen

in einen bipartiten Graphen und wertet dann die Community-Struktur auf diesem Graphen aus. Bei einer solchen Projektion auf eine dyadische Graphendarstellung geht ein Teil der Struktur höherer Ordnung, die wir einbeziehen wollen, verloren. Daher stellen wir in dieser Arbeit verschiedene Ansätze zur Community-Erkennung auf Hypergraphen vor und evaluieren sie als Vorverarbeitungsschritt für die Vergröberungs Phase. Wir stellen einen Ansatz vor, der eine modularity Formulierung optimiert, die wir durch Einbeziehung der Konnektivität von Hyperkanten abgeleitet haben. Wir optimieren diese Hypergraphen-modulairity mit Hilfe der parallelen Louvain-Methode. Zusätzlich präsentieren wir Formulierungen für die map equation auf Hypergraphen, basierend auf verschiedenen Random-Walk-Modellen. Wir integrieren unsere Community-Erkennungs-Algorithmen in den parallelen Shared-Memory-Hypergraphen-Partitionierer `Mt-KaHyPar`. Schließlich evaluieren wir verschiedene Optimierungen und Heuristiken, die wir zur Beschleunigung der Hypergraphen-modularity eingeführt haben und vergleichen unsere verschiedenen Ansätze, sowie den aktuellen graphbasierten Ansatz in umfangreichen Experimenten.
In unseren Experimenten mit 488 Hypergraphen aus den verschiedenen Anwendungsbereichen, produzierte unser Ansatz bessere Partitionen als die aktuelle Konfiguration `Mt-KaHyPar-D` für 61% der Instanzen. Allerdings ist unser Ansatz im Durchschnitt um den Faktor 5,4 langsamer. Zusätzlich evaluieren wir unseren Algorithmus auf Instanzen, die um eine Größenordnung größer sind als in anderen Publikationen.

Contents

1. Introduction

Hypergraphs are a generalization of graphs, where each (hyper)edge can connect an arbitrary number of nodes. The k-way hypergraph partitioning problem is a direct generalization of the graph partitioning problem. Here we try to find a partition of the node-set of the hypergraph into k disjoint, non-empty blocks of roughly equal size while minimizing an objective function defined on the hyperedges. The partition must satisfy the balance constraint that each block is smaller than $(1 + \epsilon)$ times the average block size.

Hypergraph partitioning is used in a wide range of problem domains. In VLSI design, it is used to partition a circuit into smaller modules such that the length of connecting wires is minimized [KAKS99]. Since a wire can connect more than two gates, hypergraphs are a more suitable model than graphs. In SAT Solving, hypergraph partitioning is used as a preprocessing step to decompose formulas into smaller sub-formulas [DK04]. Another area of application is scientific computing, where it is used to reduce the communication volume for sparse matrix-vector multiplications [VB05].

Since hypergraph partitioning is NP-hard [Len90, Fel19], heuristics are used in practice to keep up with the growing problem sizes from its many different applications. The most popular heuristic used in state-of-the-art partitioners is the *multilevel-paradigm.* It consists of three phases: First, the hypergraph is coarsened by contracting clusters of vertices while maintaining its underlying structure (*coarsening phase*). In the second phase, an initial partition is computed on the coarsest hypergraph. This partition is improved, during uncoarsening of the hypergraph, using local search heuristics.

The goal of the coarsening phase is to reduce the problem size while maintaining the structure of the hypergraph. To achieve this goal, coarsening schemes try to merge naturally existing clusters of vertices. This is done using rating functions which often only incorporate the direct neighborhood of a node. As Heuer and Schlag showed [HS17], incorporating the structure found by community detection algorithms is highly beneficial to the quality of partitions. They introduce community aware coarsening, where the coarsening algorithm only contracts nodes within the same community, previously found by a community detection algorithm.

Many of the current approaches for community detection on hypergraphs include projecting it to its bipartite- or clique-expansion and then using graph-based community detection algorithms to find community structure [EERR21, HS17, AJZM$^+$05, BGL16, LM18, KVA$^+$18]. While this is a viable approach, using a graph-based representation of a hypergraph loses some of its higher-order structure [IWW93]. Especially using the clique-expansion is not feasible in practice, since projecting each hyperedge e onto $|e| \cdot (|e| - 1)$ edges causes a combinatorial explosion. Others are limited to k-uniform hypergraphs or are more theoretic in nature and do not scale well to large instances. There is a lack of approaches working on arbitrary hypergraphs that scale well to large instances.

1.1. Problem Statement

This thesis aims to exploit the community structure of hypergraphs by using community detection directly on hypergraphs as a preprocessing step in the coarsening phase of the multilevel paradigm. Therefore different approaches to graph-based community detection should be transferred to the more general case of hypergraphs. Parallel algorithms should be developed to optimize the generalized metrics. These algorithms should be integrated into the shared-memory hypergraph partitioner `Mt-KaHyPar`. The goal is to improve the solution quality of the partitioner while not compromising too much on the running time.

1.2. Contribution

In this work, we present, to the best of our knowledge, the first parallel shared- memory community detection algorithms for hypergraphs: First, we derive a hypergraph modularity formulation based on the connectivity of hyperedges, which is the number of different communities contained in a hyperedge. We optimize this objective function by implementing the parallel Louvain method and presenting multiple optimizations, such as pruning rules and a data structure to efficiently look up connected communities of hyperedges. Second, we derive different formulations for the map equation on hypergraphs, which we optimize using our implementation of the parallel RelaxMap algorithm. We integrate our hypergraph community detection algorithms in the parallel shared memory hypergraph partitioner `Mt-KaHyPar` and use them as a preprocessing step to the community aware coarsening algorithm. The results show that our modularity approach is better suited for this application than the map equation approaches. Compared to the current configuration `Mt-KaHyPar-D`, the new configuration using hypergraph modularity produces better partitions on 61% of the benchmark instances. Unfortunately, our preprocessing step is slower by a factor of 5.4 on average.

1.3. Outline

We begin by introducing the basic concepts and notation in Section 2. We take a look at hypergraphs and different methods for community detection we will transfer to hypergraphs. We also take a brief look at hypergraph partitioning. After that, we summarize related work concerning community detection and hypergraph partitioning in Section 3. In the following Section 4 we derive a modularity metric directly on hypergraphs. The algorithm and optimizations used to optimize this metric are described in Section 5. In Section 6 we transfer the map equation to hypergraphs and describe its implementation. We present the experimental evaluation in Section 7 and conclude our work in Section 8.

2. Preliminaries

In this chapter, we will introduce the basic concepts and notation used in this work. We will start by introducing hypergraphs and graphs in Section 2.1. After that, we will briefly define random walks in 2.2 and take a look at two community detection methods on graphs (see Section 2.3): modularity and map equation. Finally, we will give a short introduction to hypergraph partitioning in Section 2.4.

2.1. Hypergraphs and Graphs

An *undirected weighted* hypergraph $H = (V, E, c, \omega)$ consists of a set of n *hypernodes* V and m *hyperedges/nets* E, as well as the *vertex weights* $c : V \to \mathbb{R}_{\geq 0}$, and the *edge weights* $\omega : E \to \mathbb{R}_{>0}$. We extend these weights to sets, i.e. for $V' \subseteq V$ and $E' \subseteq E$ $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. We dedicate $W = \omega(E)$ to the sum of all edge weights and $W_d = \omega(E_d)$ to the accumulated weight of all edges of size d. A hyperedge $e \in E$ is defined as a set of nodes $v \in V$. We call a single occurrence of a node $v \in e$ a *pin*. In this work we will also allow multipins, i.e. a node can appear more than once in an edge. Therefore we add multiplicities $m_v \in \mathbb{N}$ to the definition of a hyperedge, i.e. $e = \{(v, m_v) | v \in V\}$. The size of e is its cardinality $|e|$ or in other words the sum of multiplicities $\sum_{v \in e} m_v$. E_d denotes the set of hyperedges of size d and $\mathcal{D} = \{d : |E_d| > 0\}$ represents the set of different edge sizes occurring in the hypergraph. We define $\Phi(e, v) = m_v$ for $(v, m_v) \in e$ for net $e \in E$ and node $v \in V$ as the pin count of $v \in e$. We extend this definition to sets of nodes. For the subset $C \subseteq V$ the cumulative pin count of C in e is defined as $\Phi(e, C) = \sum_{v \in C} \Phi(e, v)$.
A vertex $v \in V$ is *incident* to a hyperedge e if $v \in e$ and $m_v > 0$ respectively. $I(v)$ denotes the set of incident nets of v, which we extend to sets of nodes as the set of unique edges with $\Phi(e, C) > 0$, in other words $I(C) = \{e \in E | \Phi(e, C) > 0\}$. With these definitions in mind we define the *volume* (or weighted degree) of a node v as $\text{vol}(v) = \sum_{e \in I(v)} \Phi(e, v) \cdot \omega(e)$, we naturally extend this definition to sets as $\text{vol}(C) = \sum_{e \in I(C)} \Phi(e, C) \cdot \omega(e)$. A hypergraph $H' = (V', E')$ is called a *subgraph* of $H = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. We call a subgraph *node-induced* if it contains the nodes V' and only the edges, for which all pins are in V', or more formally $E' = \{e \in E | e \subseteq V'\}$. Just like we call a subgraph *edge-induced* if it consists of the edges E' and all nodes which are incident to at least one of the edges in E', more formally $V' = \{v \in V | \exists e \in E' : \Phi(e, v) > 0\}$. A graph, also called *dyadic* graph, is a special case of a hypergraph, where all edges are exactly of size two. Therefore all definitions from above also apply to them.

A technique often employed, is to represent a hypergraph as a dyadic graph. The *clique/2-section* representation $G_x = (V, E_x)$ of the hypergraph $H = (V, E)$, replaces each hyperedge with a clique between its pins. In the bipartite graph representation $G_* = (V \cup E, E_*)$, the nodes and hyperedges make up the vertex set. E_* contains edge (e, v) if e is incident to v in H. Figure 2.1 illustrates these projections.

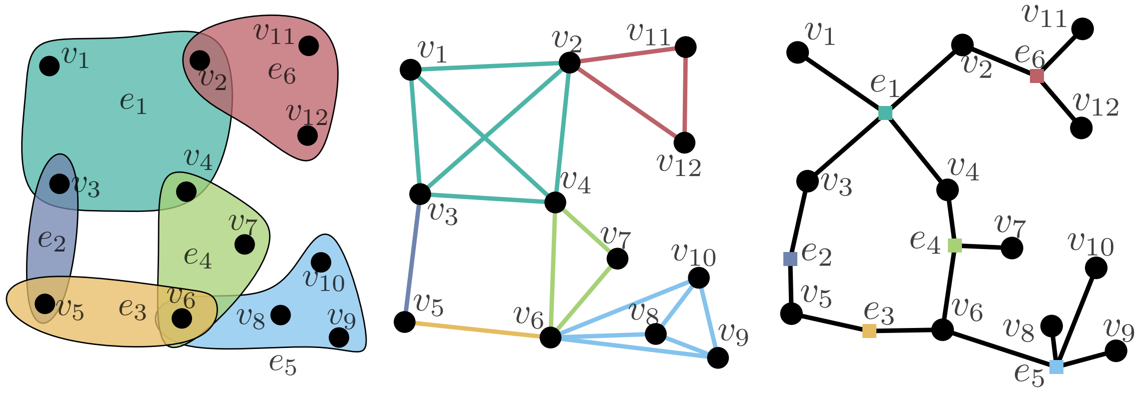

Figure 2.1.: An example of a hypergraph with 12 nodes, 6 hyperedges and 18 pins (left) along with its clique (center) and bipartite (right) graph representations.

Clusterings A *clustering* $\mathcal{C} = \{C_1, \ldots, C_k\}$ of a hypergraph $H = (V, E)$ is a partition of its nodes V, where the individual *clusters/communities* C_i are non-empty and disjoint. The set $\mathrm{NC}(v) = \{C_i \in \mathcal{C} | \exists e \in E\{u, v\} \subseteq e \wedge u \in C_i \wedge v \notin C_i\}$ denotes the community neighbors of v, excluding its own community. Given a cluster C_i and its node-induced subgraph $G(C_i) = (C_i, E(C_i), \omega)$, where $E(C_i) = \{\{u, v\} \in E | v, w \in C_i\}$ is the set of *intra-cluster* edges of C_i. We can then define the set of intra-cluster edges of the whole clustering as $E(\mathcal{C}) = \bigcup_{i=1}^{k} E(C_i)$ and $E \setminus E(\mathcal{C})$ denotes the set of *inter-cluster* edges, where $|E(\mathcal{C}) = m(\mathcal{C})|$ and $|E \setminus E(\mathcal{C})| = \overline{m}(\mathcal{C})$. We call a clustering a *singleton clustering* if all communities contain only one element, i.e., every node is in its own community.
These definitions naturally generalize to weighted graphs by using the edge weights $\omega(e)$ instead of 1. $\omega(\mathcal{C})$ is the sum of the weights of all intra-cluster edges, while $\overline{\omega}(\mathcal{C})$ denotes the sum of the weights of all inter-cluster edges.

2.2. Random Walks

The mathematical model for random walks we will use to formulate the map equation are *Markov Chains*. A Markov Chain is a stochastic model which describes a sequence of possible events. It consists of a (in this work always finite) set of states S and a transition matrix P of size $|S| \times |S|$. An entry P_{ij} is the probability to transition from state i to state j in one discrete time step. Note that this probability is solely dependent on the states i and j. A state i is called *recurrent* if an infinite random walk that starts in i visits i infinitely often. The *period* d_i of a state i is the greatest common divisor of the number of transitions by which i can be reached, starting from i. If d_i is equal to 1 it is called *aperiodic*. A state is called *ergodic* if it is aperiodic and recurrent. If all states of a Markov Chain are ergodic it is called ergodic. The stationary distribution π of an ergodic Markov Chain is given by $\pi = \pi P$.

2.3. Community Detection on Graphs

The goal of community detection on a graph G is to find the underlying structure of G by dividing the set of nodes V into disjoint subsets, so-called communities, so that nodes within a community are *densely*, but different communities are *sparsely* connected [Sch07, For10]. Since there is no universally agreed-upon definition on how to judge the quality of such a clustering, there are multiple different quality functions one can optimize, such as modularity [NG04] and map equation [RAB09]. Which we are going to look into in the following sections.

Coverage. The simplest way to model the goal of community detection is the so called *coverage*. The coverage $cov(\mathcal{C})$ of a clustering $\mathcal{C}$ is defined as the fraction of intra-cluster edges $m(\mathcal{C})$ (or their weight $\omega(\mathcal{C})$) within a Graph:

$$cov(\mathcal{C}) := \frac{m(\mathcal{C})}{|E|} = \frac{m(\mathcal{C})}{m} \qquad\qquad cov_\omega := \frac{\omega(\mathcal{C})}{W} \tag{2.1}$$

Since the goal is to have dense intra-cluster connections, large coverage values correspond to a clustering of good quality. The problem with coverage is that this is not necessarily true because if there is only one cluster, which contains all nodes, then coverage takes on its maximum value of 1.

Modularity. Modularity, as proposed by Newman and Girvan [NG04], tries to tackle the problem of coverage by subtracting the expected fraction of intra-cluster edges in a graph with the same clustering but random connections between its nodes. This translates to: Given a Graph $G = (V, E)$ and a clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$

$$mod(\mathcal{C}) := \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{deg(i) \cdot deg(v)}{2m} \right) \delta(C(i), C(j)) \tag{2.2}$$

where A is the adjacency matrix of G and A_{ij} represents the edge (i, j). Therefore $m = \frac{1}{2} \sum_{ij} A_{ij}$ is the number of edges $|E|$ and $\mathcal{C}(i)$ is the community of node i. $\delta(\mathcal{C}(i), \mathcal{C}(j))$ is the Kronecker delta, which is one exactly when $\mathcal{C}(i) = \mathcal{C}(j)$ and zero otherwise. We can reformulate this equation to emphasize the presence of coverage:

$$mod(\mathcal{C}) := \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C_i \in C} \left(\sum_{v \in C_i} deg(v) \right)^2 \tag{2.3}$$

In the latter formulation we can see the inherent trade off of modularity. On the one hand we want to have as many edges as possible inside of communities to maximize coverage, on the other hand we want to minimize the probabilistic term and therefore splitting the graph into equally sized communities of rather balanced total degree. We can also generalize the formula to the weighted case as proposed by Newman [New04]:

$$mod_\omega(\mathcal{C}) = \frac{\omega(\mathcal{C})}{W} - \frac{1}{4W^2} \sum_{C_i \in \mathcal{C}} \left(\sum_{v \in C_i} \omega(v) \right)^2 \tag{2.4}$$

The values of modularity reside in $[-0.5, 1]$, while the smallest values are obtained by bipartite graphs, the largest are obtained by disjoint cliques [Gö10].
We can quickly take a look at the previously discussed problem of coverage. If there is only one cluster containing all nodes, the modularity value is 0 since the coverage and expected coverage are both 1. While this issue has been solved, there are still known issues of modularity, such as the resolution limit [FB07]. Nevertheless, modularity is one of the most widely used quality functions for partitioning large graph data.

Map Equation. The *map equation* is an information-theoretic quality function for community detection first proposed by Roswall et al. in [RAB09]. The map equation describes the shortest possible encoding of a random walker's path on a given graph with a given community structure.

The encoding process works as follows: There are two codebooks, which work like lookup tables for codes. The first one, the index codebook, specifies the codes for the community codebooks. The second one, the community codebook, specifies the codes for each node in this community. We use an exit code word every time the random walker exits a community. The lengths of the codes in the index codebook are derived by looking at the relative rate at which the random walker visits each community. Thus short codewords are used for frequently visited communities and longer ones for rarely visited communities. The lengths of code words in the community codebook are derived from the relative probability of visiting each node of the community accordingly.

The map equation now uses the duality between encoding this random walk and finding regularities in the structure of a graph. Since by minimizing the encoding of the random walk, we simultaneously tackle the problem of partitioning the graph into communities with respect to the relative visit frequencies.

To condense this all into one formula we use Shannon's source coding theorem [Sha48], it implies that the average length of code words to describe the n states of a random variable X, occurring with frequency p_i, can't be less than the entropy of X, which is $H(X) = -\sum_1^n p_i \log(p_i)$. The map equation is the following:

$$L(M) = q_{\curvearrowright} H(Q) + \sum_i^m p_{\circlearrowright}^i H(P^i) \tag{2.5}$$

where $H(Q)$ is the average length of the index codebook weighted by its frequency of use. It is used every time the random walker exits a community. The length of a codeword is derived from the probability to leave the community $q_{i\curvearrowright}$ in comparison to the probability to leave any community at all. Therefore

$$H(Q) = -\sum_{i=1}^m \frac{q_i \curvearrowright}{\sum_{j=1}^m q_{j\curvearrowright}} \log\left(\frac{q_i \curvearrowright}{\sum_{j=1}^m q_{j\curvearrowright}}\right)$$

.

The second part of the sum models the code lengths inside a community. Where $H(P^i)$ is the frequency weighted average length of a code word in the community codebook. A codebook of community C_i is used when visiting any node of this community or leaving it. Let p_v be the probability to visit node $v \in V$ then this translates to $p_{\circlearrowright}^i = \sum_{v \in C_i} p_v + q_{i\curvearrowright}$. The resulting entropy for the community codebooks is:

$$H(P^i) = -\frac{q_{i\curvearrowright}}{q_{i\curvearrowright} + \sum_{u \in C_i} p_u} \log\left(\frac{q_{i\curvearrowright}}{q_{i\curvearrowright} + \sum_{u \in C_i} p_u}\right) \tag{2.6}$$

$$-\sum_{v \in C_i} \frac{p_v}{q_{i\curvearrowright} + \sum_{u \in C_i} p_u} \log\left(\frac{p_v}{q_{i\curvearrowright} + \sum_{u \in C_i} p_u}\right) \tag{2.7}$$

2.4. Hypergraph Partitioning

The practical results of this work will be tested and evaluated in the hypergraph partitioning framework `Mt-KaHyPar`, therefore we first define what a *k-way partition* is and then introduce the *k-way hypergraph partitioning problem* which the framework is trying to solve.

k-way Partition. A *k-way partition* of a hypergraph H is a partition of its vertex set into k *disjoint blocks* $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset \ \forall i \in \{1, \ldots, k\}$. A *k-way* partition is ϵ-*balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq (1+\epsilon)\lceil \frac{c(V)}{k} \rceil$. We define the *connectivity set* $\Lambda(e) := \{V_i | \Phi(e, V_i) > 0\}$ for each edge $e \in E$ and its cardinality $\lambda(e) := |\Lambda(e)|$ is called the *connectivity* of a hyperedge e. A net e, whose nodes are in more than one block and therefore with connectivity $\lambda(e) > 1$, is called *cut net* and $E(\Pi) := \{e \in E | \lambda(e) > 1\}$ denotes the set of all cut nets.

k-way Hypergraph Partitioning Problem. The *k-way hypergraph partitioning problem* is to find an ϵ-balanced k- way partition of a hypergraph H(V,E,c,ω) that minimizes an objective function for a given $\epsilon \geq 0$. The two most popular objective functions are the *cut-net* metric, which sums the weights of all cut nets: $cut(\Pi) := \sum_{e \in E(\Pi)} \omega(e)$ and the *connectivity*(also km1- or ($\lambda - 1$)-) metric: $(\lambda - 1)(\Pi) := \sum_{e \in E(\Pi)} (\lambda(e) - 1)\omega(e)$. The latter one additionally takes into account the number of different blocks cut by each edge. While both metrics revert to the edge-cut metric on dyadic graphs, we will, in this work, focus on the connectivity metric since our definition of modularity is also based on it.

3. Related Work

In this chapter we give a concise overview over the work most relevant to ours. In Section 3.1, we present different results regarding community detection, including community detection using different modularity approaches in Section 3.1.1 and random walks on hypergraphs related to the map equation in Section 3.1.2. Finally, in Section 3.2, we introduce the *multilevel-paradigm* as well as the hypergraph partitioning framework `Mt-KaHyPar`.

3.1. Community Detection

An approach for community detection aside from modularity maximization (see Section 3.1.1) and map equation is spectral clustering. Here clustering is performed based on the eigenvalues of a hypergraph Laplacian. One approach of generalizing the graph Laplacian to hypergraphs, is to project the hypergraph onto a dyadic graph [AJZM$^+$05, BGL16, LM17]. Other approaches include p-Laplacian operators [LM18] or different tensor-based methods, like spectral clustering using Laplacian tensors [CCQY20] or regularized tensor power iteration [KSX20]. Although many possibilities have been tried, there is a lack of approaches that can perform community detection on arbitrary hypergraphs and also scale well to large data sets. Many previous approaches are limited to k-uniform hypergraphs or are more theoretic in nature and do not scale well to large instances.

Louvain Method. The *Louvain* algorithm was first introduced by Blondel et al. [BGLL08]. It is a community detection algorithm on graphs maximizing the modularity metric. Initially, every node is placed into its own singleton community. The algorithm then improves this initial assignment by alternating between the following two phases:

- *local moving.* In this phase, the algorithm iterates over all nodes of the graph, moving the node into the neighboring community that increases modularity the most. This is repeated until no single move improves the objective.
- *contraction.* In this phase, the communities are contracted into single nodes. Emerging parallel edges are also contracted by adding up their weights. Therefore decreasing the size of the graph significantly. This has the effect that the next local moving phase is significantly faster and also executes moves on a bigger scale, i.e., each node move corresponds to moving a group of nodes into a neighboring one.

A significant part of the efficiency of the Louvain method lies in the computation of the change in modularity for each neighboring community of a node. This modularity delta can be calculated in $\mathcal{O}(deg(v))$ time using appropriate data structures. The difference in modularity by moving node v from community C to D is given by [SM16]:

$$\Delta mod(u, C \to D) = \frac{\omega(u, D \setminus \{v\}) - \omega(u, C \setminus \{v\})}{W} + \frac{(\mathrm{vol}(C \setminus \{v\}) - \mathrm{vol}(D \setminus \{v\})) \cdot \mathrm{vol}(u)}{2 \cdot W^2}$$

Parallel Louvain Method (PLM). The *parallel Louvain method* first presented by Staudt and Meyerhenke [SM16], speeds this approach up by parallelizing both the local moving phase, as well as the coarsening phase. In the first phase, the moves are evaluated and executed in parallel. Therefore one has to pay attention to keep data structures consistent, e.g., by updating stored values atomically. A significant difference to the sequential Louvain method is that, since local moving occurs in parallel, an executed move might not be the best local decision. While evaluating a move for one node, other threads might have performed a move that affects the change in modularity Δmod for this very node. This can result in moves decreasing modularity. However, it seems to only slightly impact solution quality because these non-optimal local moves may be corrected in later iterations. In particular, randomizing the order in which nodes are accessed helps to mitigate the effect.

Parallel Louvain Methods for Map Equation. When executing a move in the modularity approach, we have to update the volumes of the corresponding communities. This is possible since the change is solely dependent on the moved node, which is only looked at by one thread per iteration. For the map equation on the other hand, we have to update the exit probabilities, which are also dependent on the number of pins a community has in each edge. Because of this, a calculated delta can be wrong if another move has been executed in the meantime. This can lead to cyclical movements of nodes and premature convergence. As Bae et al. [BHW+17] showed, taking the same approach for parallelization as for modularity is not competitive with its sequential implementation.
Instead, the authors suggest the *RelaxMap* algorithm. Here a thread has to acquire a global lock to execute a move and update the information. Because of this, when a subsequent thread tries to perform a move, it sees all changes corresponding to previous moves. Although one might think waiting for the lock may become a major bottleneck, it has been shown to work well in practice. Another approach by Hamann et al. [HSWZ18] involves a procedure they call *synchronous local moving*. The idea is to split one local moving round into multiple sub-rounds, where each node is assigned a random sub-round. Then for all nodes of a sub-round, the best move is calculated based on the data of the previous sub-round and stored in parallel. After all threads finish, all moves are executed synchronously. The idea is that by splitting the nodes into sub-rounds, the probability of cyclical moves is reduced. The synchronous movement then ensures that all updates are visible at once. Therefore no wrong moves can be made based on partially updated data.

3.1.1. Hypergraph Modularity

There are multiple ways to translate the modularity definition on graphs to hypergraphs. For example, hyperedges can count towards the coverage if either all pins belong to the same community, the majority belong to the same community, or we simply count the number of different communities spanned by the hyperedge. Additionally, different null models can lead to different probabilistic terms for the expected coverage. Depending on the specific application, certain translations may be more suitable than others. In this section, we will provide a concise overview of recent work on hypergraph modularity.

Modularity directly on Hypergraphs. Modularity on graphs compares the intra-cluster edges of a community structure with the expected number of intra-cluster edges on a null model. For graphs, this is the Chung-Lu model [CL04], which describes how edges are independently sampled while preserving the degree distribution of the original graph. Kaminski et al. generalized this model in [KPP$^+$19] to hypergraphs:

Let F_d be the family of multisets of size d, representing all possible edges of size d in the hypergraph H; that is,

$$F_d = \left\{ \{(v_i, m_i) : 1 \leq i \leq n\} : \sum_{i=1}^{n} m_i = d \right\}$$

We can now calculate the probability to generate edge $e \in F_d$ by independently drawing d nodes, with the probability to draw a node $v \in V$ of $deg(v_i)/\operatorname{vol}(V)$, and then repeating this process $|E_d|$ times.

$$P_{\mathrm{H}}(e) = |E_d| \cdot \binom{d}{m_1, \ldots, m_n} \prod_{i=1}^{n} \left(\frac{deg(v_i)}{\operatorname{vol}(V)} \right)^{m_i}$$

The authors note that this process does not only result in the expected node degrees matching the node degrees in H, but also preserves the number of edges of a particular size $|E_d|$.

Let $\left(X_1^{(d)}, \ldots, X_n^{(d)}\right)$ be the random vector following a multinomial distribution with parameters $d, p_{\mathrm{H}}(1), \ldots, p_{\mathrm{H}}(n)$, where $p_{\mathrm{H}}(i) = deg(v_i)/\operatorname{vol}(V)$ and $\sum_{i=1}^{n} p_{\mathrm{H}}(i) = 1$; that is,

$$s_{\mathrm{H}}(e) = \mathbb{P}\left(\left(X_1^{(d)}, \ldots, X_n^{(d)}\right) = (m_1, \ldots, m_n) \right) = \binom{d}{m_1, \ldots, m_n} \prod_{i=1}^{n} \left(p_{\mathrm{H}}(i)\right)^{m_i}$$

Using their generalized Chung-Lu model, Kaminski et al. [KPP$^+$19] define the edge contribution as the (weighted) number of edges entirely contained in a community. This very strict approach leads to the following definition of the edge contribution for a given community $C_i \subseteq V$:

$$e(C_i) = |\{e \in E; e \subseteq C_i\}|$$

which results in the definition of *strict modularity*:

$$mod(\mathcal{C}) = \frac{1}{|E|} \left(\sum_{C_i \in \mathcal{C}} e(C_i) - \sum_{d \geq 2} |E_d| \sum_{C_i \in \mathcal{C}} \left(\frac{\operatorname{vol}(C_i)}{\operatorname{vol}(V)} \right)^d \right)$$

Additionally, they define a *majority modularity*, where edges contribute to the edge contribution of a community if the majority (more than half) of its pins belong to the community. Which looks like this:

$$mod(\mathcal{C}) = \frac{1}{|E|} \left(\sum_{C_i \in \mathcal{C}} e(C_i) - \sum_{d \geq 2} |E_d| \sum_{C_i \in \mathcal{C}} \mathbb{P}\left(Bin\left(d, \frac{\operatorname{vol}(C_i)}{\operatorname{vol}(V)}\right) > d/2 \right) \right)$$

The authors propose a heuristic greedy algorithm based on the algorithm by Clauset et al. [CNM04]. It iterates over all edges connecting at least two communities and selects the edge, which, when merging all connected communities, increases modularity the most. In their most recent work [KPT21], they propose an algorithm consisting of three phases.

First, they cluster the graph using graph-based modularity. After that, they merge the clusters found in step 1 based on their hypergraph modularity measure and finally, they refine this clustering by running a single Louvain pass.

Another method of receiving a modularity objective apart from comparing against a null model, like Kaminski et al. [KPP+19] do, is statistical inference. Chodrow et al. [CVB21] generalize the degree corrected stochastic block model (DCSBM) from graphs to hypergraphs and then use maximum-likelihood inference to derive a parameterized modularity objective. In their model, the probability that a given hyperedge e exists not only depends on the size of the edge and the degree of its nodes but additionally on an affinity function Ω. Such an affinity function influences the probability based on the clusters of the pins of e. The authors present four different classes of affinity functions. They focus on the All-Or-Nothing affinity function, which only differentiates between edges fully contained in one community and all others. The strict-modularity objective by Kaminski et al., for example, is obtained by choosing a special case of the All-Or-Nothing function. They also mention that by choosing the affinity function to be $exp(\lambda(e) - 1)$, the first part of their modularity objective corresponds to the connectivity metric.
The authors propose a very general sequential *symmetric hypergraph maximum-likelihood Louvain* algorithm to optimize the modularity metric. In principle, it works just like the Louvain algorithm by Blondel et al. [BGLL08] with the major drawback, as mentioned by the authors that clusters of nodes cannot be collapsed into super-nodes, and therefore it is slow even on relatively small hypergraphs. Although they also propose the *all-or-nothing hypergraph maximum likelihood Louvain*, which is specially designed for their all-or-nothing approach and, due to the much simpler affinity function is able to properly execute the contraction phase of the original Louvain algorithm.

Modularity on Graph Representations. Another common approach for community detection, which is used in the `KaHyPar` [AHSS17, HS17, HSS19] and `Mt-KaHyPar` [GHSS20] frameworks, is to use a graph-based representation of the hypergraph and then perform (graph-based) community detection on it. The two common models are the *clique* and *bipartite* representations.
This technique may have several problems. The first one being that the hypergraph has to be projected onto the graph first, which may cause additional work and increased memory requirements. Additionally, approaches projecting the hypergraph onto its clique-graph representation may result in a combinatorial explosion since every hyperedge e is projected onto $|e| \cdot (|e| - 1)$ edges in the corresponding graph.
Let $G_* = (V \cup E, E_*)$ be the bipartite graph representation of the hypergraph $H = (V, E)$. We will call the nodes representing edges *E-nodes* and the ones representing nodes *V-nodes*. Performing community detection on this representation of the hypergraph results in a partition of the V- and E-nodes, since we are only interested in the partition of the V-Nodes, Heuer and Schlag model the weights of the edges in E_* according to the *density* $d := \frac{m}{n}$ of the hypergraph [HS17, Sch20]. If $d \approx 1$, the degree of V- and E-nodes is roughly the same and uniform edge weights are used. If $d \gg 1$ then there are a lot more E- than V-nodes. To prevent the community structure from being dominated by the E-Nodes the edge weights are chosen as $\frac{d(v)}{|e|}$, to promote the formation of communities around high degree nodes $v \in V$ by strengthening the connection between E-nodes and high degree V-nodes. If the density is low, i.e., $d \ll 1$, the authors choose the edge weights as $1/|e|$ to ensure that high-degree E-nodes do not attract too many V-nodes and therefore dominate the community structure.
Kumar et al. [KVA+18, KVA+20] expand upon the approach of projecting a hypergraph to its clique graph representation with their *Iteratively Reweighted Modularity Maximization (IRMM)* algorithm. It takes turns between running the Louvain algorithm on the clique-

graph representation of the hypergraph and reweighing the edges of the graph by applying a cut penalty, with the aim of favoring balanced cut edges.

3.1.2. Random Walks on Hypergraphs

Chitra and Raphael [CR19] formulate a two step process for random walks on hypergraphs: The random walker chooses an incident hyperedge e of node v with probability $\frac{\omega(e)}{\omega(I(v))}$ and then chooses a node u from this hyperedge with probability $\frac{\gamma_e(u)}{\delta(e)}$. Where $\gamma_e(u)$ is the edge dependent vertex weight of u and $\delta(e)$ the sum of all edge dependent vertex weights of that edge. In the recent work of Eriksson et al. [EERR21], the authors model random walks on hypergraphs in a couple of different ways. They project the hypergraph onto its bipartite and clique-representation as well as a multi-layer representation and then define a random walk using hyperedge-dependent node weights. The authors use the infomap [EBR17] algorithm to optimize the community structure on each of these dyadic representations. They additionally suggest an extension to the random walk on hypergraphs, where the probability of picking a hyperedge is not only proportional to its weight but also its similarity to the previously picked hyperedge.

3.2. Hypergraph Partitioning

In practice, heuristic algorithms are used for hypergraph partitioning since the problem is known to be NP-hard [Len90, Fel19]. We will first present the *multilevel paradigm*, which is the most successful approach used in many state-of-the-art-partitioners [KAKS99, Sch20, CA99], including the `Mt-KaHyPar` framework which we introduce after.

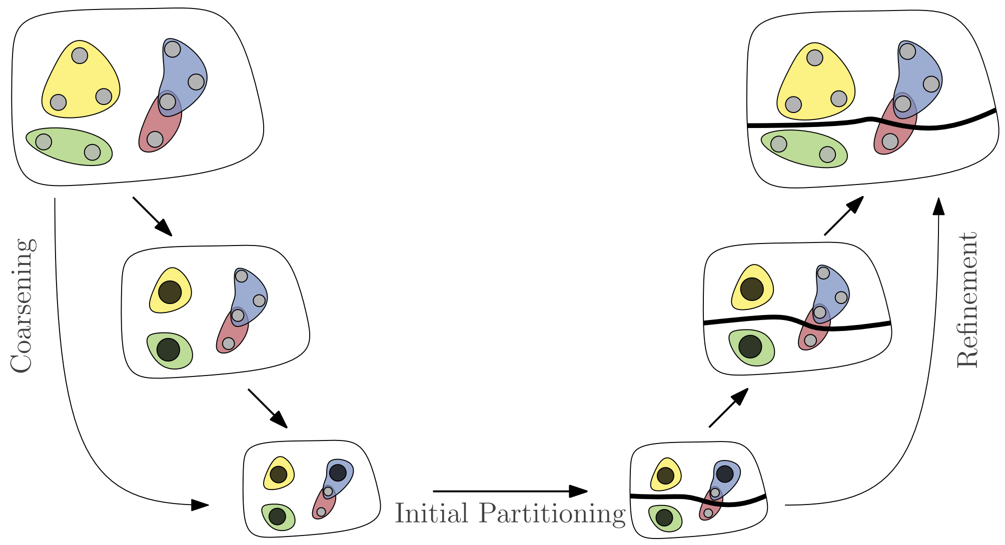

Figure 3.1.: Partitioning a Hypergraph via the multilevel paradigm

Multilevel Paradigm. The multilevel paradigm (see Figure 3.1) consists of three phases, the *coarsening* phase, the *initial partitioning* phase and the *uncoarsening/refinement* phase. The goal of the first phase is to provide a sequence of successively smaller hypergraphs while maintaining its structural similarity. This is achieved by contracting clusters of vertices of the hypergraph, which is done until the size of the coarsest hypergraph reaches a predefined contraction limit. The second phase, the initial partitioning phase, computes a k-way partition of the smallest hypergraph. This can be done using more sophisticated algorithms, which provide better quality since the hypergraph is small. The final phase, the uncoarsening or refinement phase, successively projects the partition onto the next finer hypergraph in the hierarchy. Moreover, at each uncoarsening step, the solution is refined, using local search heuristics.

Mt-KaHyPar. The Mt-KaHyPar framework is a shared-memory multilevel hypergraph partitioning system [GHSS20]. Mt-KaHyPar, uses an additional *preprocessing* phase which was originally proposed in the sequential hypergraph partitioner KaHyPar. In this preprocessing phase, community detection is used to identify *densely connected* regions of the hypergraph, which are then used in the coarsening phase to restrict contractions to vertices within the same community. The community detection algorithm currently used is the parallel Louvain method (PLM) proposed by Staudt and Meyerhenke [SM16] for modularity maximization on the bipartite graph representation of the input hypergraph. The coarsening algorithm contracts the hypergraph with respect to the community structure obtained in the preprocessing phase. It preserves the global community structure of the input hypergraph by prohibiting contractions between nodes of different communities and therefore only coarsening by executing intra-community contractions based on rating functions developed for the hypergraph partitioning problem. The initial partitioning is performed via parallel multilevel recursive bipartitioning with work-stealing. To compute an initial bipartition of the coarsest hypergraph, Mt-KaHyPar uses a portfolio of initial bipartitioning algorithms for a detailed overview we refer to the original papers [GHSS20, SHH$^+$15]. For the third phase, the *refinement* phase, a parallel label propagation algorithm called *size-constrained label propagation* in ParHIP [MSS15] and Mt-KaHIP [ASS18] is used, as well as a parallel version of the classical FM local search [FM82].

4. Hypergraph Modularity

Most modularity methods for community detection on hypergraphs transform the hypergraph into a dyadic graph to then use graph-based modularity maximization approaches to find a community structure of the hypergraph. This approach has several problems: The first one is that, as Ihler et al. [IWW93] showed, projecting a hypergraph onto a dyadic graph loses some information about the higher-order structure of the hypergraph. Therefore we expect to find more accurate clusterings by employing modularity optimization on hypergraphs directly. Additionally, projecting the hypergraph onto a dyadic graph causes additional work and memory overheads. E.g., approaches projecting the hypergraph onto its clique-graph representation may result in a combinatorial explosion, since every hyperedge of size d is projected onto $d \cdot (d-1)$ edges in the corresponding graph.
To define modularity on hypergraphs, we need an appropriate null model. We use the proposed model by Kaminski et al. [KPP$^+$19]. The purpose of the null model is to simulate the hypergraph with edges randomly rewired while preserving the general structure of the hypergraph, in particular the vertex degrees. It promotes balanced communities and thus prevents that the optimal solution is just a single community.
The general procedure of deriving the modularity objectives is as follows: Firstly we formulate the edge contribution for a community C, which is the contribution of C to the overall coverage. We then use this initial definition of the edge contribution to define the coverage and then continue to derive the expected coverage according to the null model. We do this by defining the family of multisets $F_d(C) \subseteq F_d$, which contains all edges of size d contributing to the edge contribution of C. We can then calculate the expected coverage by summing over the expected edge contribution of each community, which is the chance to randomly generate the edge multiplied by its contribution, of each edge $e \in F_d(C)$ for all $d \in \mathcal{D}$. We derive our modularity objectives for the weighted case, since that is the formulation needed for our implementation using the parallel Louvain method, but will point out the differences to the unweighted case.
In Section 4.1 we aim to tackle the problem that when using a graph representation of a hypergraph, some higher-order information may be lost. Therefore we will define the modularity objective directly on the hypergraph based on the connectivity of hyperedges. We will use this definition of modularity throughout the rest of the work and for our experiments.

4.1. Coverage defined by Connectivity

The connectivity $\lambda(e)$ of a hyperedge e, is the number of communities contained in e. We use $\lambda(e)$ to model the basic idea of coverage, that communities are sparsely connected and connections inside a community are dense, by defining the edge contribution of a community C as the cumulative weight of the edges connected with C. Note, that from now on we will refer to the edge contribution as connectivity.

$$\forall C \in \mathcal{C} : e_H(C) = \sum_{\substack{e \in E \\ \Phi(e,C)>0}} \omega(e) \tag{4.1}$$

Based on that, we define the coverage of a cluster $\mathcal{C}$ as follows:

$$cov(\mathcal{C}) = \frac{1}{W} \sum_{C \in \mathcal{C}} e_H(C) = \frac{1}{W} \sum_{e \in E} \lambda(e)\omega(e) \tag{4.2}$$

A hyperedge e contributes to the coverage function of a community C, if at least one pin of e belongs to C. In order to define $F_d(C) \subseteq F_d$ as the family of multisets of size d with at least one member in C, recall that an edge is a set of tuples (v, m_v), where $v \in V$ is a node and m_v is its multiplicity in e.

$$F_d(C) = \left\{ \{(v_i, m_i) : 1 \leq i \leq n\} \mid \sum_{1 \leq i \leq n} m_i = d \wedge \sum_{i: v_i \in C} m_i \geq 1 \right\}$$

We calculate the expected connectivity as follows:

$$\mathbb{E}_{H' \sim \mathrm{H}} \left[e_{H'}(C) \right] = \sum_{d \in \mathcal{D}} W_d \left(1 - \left(1 - \frac{\mathrm{vol}(C)}{\mathrm{vol}(V)} \right)^d \right) \tag{4.3}$$

The full derivation is given in Appendix A. More generally, we draw W_d edges of size d at random. An edge e contributes to the connectivity of C, only if $e \in F_d(C)$ and therefore only if at least one pin belongs to C. The probability to draw such an edge is the same as one minus the probability to draw an edge which does not connect to C, which is $(\mathrm{vol}(C)/\mathrm{vol}(V))^d$. We repeat this process for each $d \in \mathcal{D}$ and count the number of edges $e \in F_d(C)$ for a $d \in \mathcal{D}$.

In summary, the modularity function defined by connectivity can be formulated as follows:

$$mod(\mathcal{C}) = \frac{1}{W} \cdot \left(\sum_{C \in \mathcal{C}} e_H(C) - \sum_{d \in \mathcal{D}} W_d \sum_{C \in \mathcal{C}} 1 - \left(1 - \frac{\mathrm{vol}(C)}{\mathrm{vol}(V)} \right)^d \right) \tag{4.4}$$

The corresponding optimization problem tries to minimize $mod(\mathcal{C})^*$. The rationale behind this is that we want edges to connect as few communities as possible, which corresponds to a low connectivity, therefore a low coverage score, and consequently a high expected connectivity.
We can directly transfer the above formulas to the unweighted case, since edges in an unweighted hypergraph have weight 1. This translates to counting the number of edges instead of accumulating their weight, replacing W with $|E|$ and instead of using the volumes $\mathrm{vol}(C)$, $\mathrm{vol}(V)$ using the pin counts $\sum_{e \in I(C)} \Phi(e, C)$ and $\sum_{e \in E} \Phi(e, V) = \sum_{e \in E} |e|$.

Change in Modularity. The Louvain method moves a vertex to an adjacent cluster C that maximizes the modularity function. Recalculating modularity after each move from scratch would not be feasible for large graphs. Therefore, the algorithm computes the local change in modularity based on the neighborhood of each vertex. When moving a vertex v from Community C to D, the change in modularity $\Delta_{mod}(v, C \to D)$ consists of two parts. The change in connectivity Δ_{e_H} of C and D and the change in the expected connectivity $\Delta_{\mathbb{E}}$ of C and D.

$$\Delta_{mod}(v, C \to D) = \frac{1}{W}\left(\Delta_{e_H} + \Delta_{\mathbb{E}}\right)$$

Let $C^- := C \setminus v$ and $D^+ := D \cup v$. We can calculate the change in connectivity by subtracting the old values, i.e. before moving the node ($e_H(C)$, $e_H(D)$), from the new ones ($e_H(C^-)$, $e_H(D^+)$).

$$\begin{aligned}\Delta_{e_H} &= e_H(C^-) - e_H(C) + e_H(D^+) - e_H(D)\\ &= \omega\left(\{e \in E \mid \Phi(e, C^-) > 0\}\right) - \omega\left(\{e \in E \mid \Phi(e, C) > 0\}\right)\\ &+ \omega\left(\left\{e \in E \mid \Phi(e, D^+) > 0\right\}\right) - \omega\left(\{e \in E \mid \Phi(e, D) > 0\}\right)\\ &= \omega\left(\{e \in I(v) \mid \Phi(e, D) = 0\}\right) - \omega\left(\{e \in I(v) \mid \Phi(e, C^-) = 0\}\right)\end{aligned} \tag{4.5}$$

We do the same for the change in expected connectivity:

$$\begin{aligned}\Delta_{\mathbb{E}} &= -\mathbb{E}_{H' \sim \mathrm{H}}\left[e_{H'}(C^-)\right] + \mathbb{E}_{H' \sim \mathrm{H}}\left[e_{H'}(C)\right] - \mathbb{E}_{H' \sim \mathrm{H}}\left[e_{H'}(D^+)\right] + \mathbb{E}_{H' \sim \mathrm{H}}\left[e_{H'}(D)\right]\\ &= \sum_{d \in \mathcal{D}} W_d \cdot \left[-1 + \left(1 - \frac{\mathrm{vol}(C) - \mathrm{vol}(v)}{\mathrm{vol}(V)}\right)^d + 1 - \left(1 - \frac{\mathrm{vol}(C)}{\mathrm{vol}(V)}\right)^d\right.\\ &\left. -1 + \left(1 - \frac{\mathrm{vol}(D) + \mathrm{vol}(v)}{\mathrm{vol}(V)}\right)^d + 1 - \left(1 - \frac{\mathrm{vol}(D)}{\mathrm{vol}(V)}\right)^d\right]\\ &= \sum_{d \in \mathcal{D}} \frac{W_d}{\mathrm{vol}(V)^d}\left[(\mathrm{vol}(V) - \mathrm{vol}(C) + \mathrm{vol}(v))^d - (\mathrm{vol}(V) - \mathrm{vol}(C))^d\right.\\ &\left. + (\mathrm{vol}(V) - \mathrm{vol}(D) - \mathrm{vol}(v))^d - (\mathrm{vol}(V) - \mathrm{vol}(D))^d\right]\end{aligned} \tag{4.6}$$

As we can see in Equation (4.5) and Figure 4.1 only edges incident to v, for which v was the only node belonging to one of the communities C and D, change the connectivity. More specifically, there are two distinct cases:

1. If $v \in e$ is the last pin of e part of cluster C, then the connectivity $\lambda(e)$ of that edge is reduced by one.
2. If e does not contain any pin of cluster D before, then its connectivity is increased by one.

In the unweighted case, instead of summing the weights of edges in the respective sets, we simply count them.

Contraction. The main goal of this procedure is to reduce the number of structures (vertexes and edges) we have to keep track of while preserving the structure of the hypergraph and the current modularity value. To achieve this, we have to preserve the overall weight of all edges (W), the weight of edges of size d (W_d), as well as the volume of each community ($\mathrm{vol}(C)$). We achieve this by doing the following: First, we collapse all

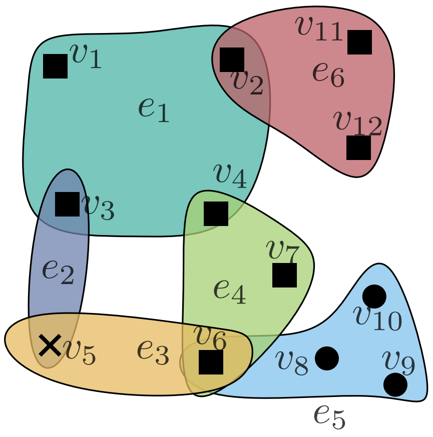

Figure 4.1.: A hypergraph where the shape of the node denotes the community it is currently part of. If we try to move node v_6 from the square-community to the circle-community, we can observe all three cases for the change in connectivity. e_5 currently connects to the square-community only via v_6 (case 1). e_4 currently doesn't connect to the circle-community at all, but will after moving v_4 (case 2). For e_3 the connectivity doesn't change at all, since it will connect two communities before and after the move is executed.

vertexes $v \in C$ into a single super-vertex v_C. This also means replacing all occurrences of $v \in C$ in all edges $e \in E$ by v_C. Both is done without changing the values we have an eye on. After that we can reduce the number of single-node edges, which only contain v_C. Let $E_C = \{e \mid e \in E, \forall v_i \in V : m_i = 0, m_{v_C} \neq 0\}$ be the set of these edges. By merging all $e \in E_C$ of size d into a single edge e_d with weight $\omega(e_d) = \sum_{e \in E_C \wedge |e|=d} \omega(e)$ we reduce the amount of edges while preserving the volume as well as the overall weight W, since no edge weight is lost. Finally, we can also remove parallel edges, i.e. edges which contain the same pairwise vertexes as well as multiplicities. More formally: two edges $e, f \in E$ are mergeable if and only if $\forall v \in V : (v, m_v) \in e, (v, w_v) \in f : m_v = w_v$. Figure 4.2 illustrates the contraction of an example hypergraph.

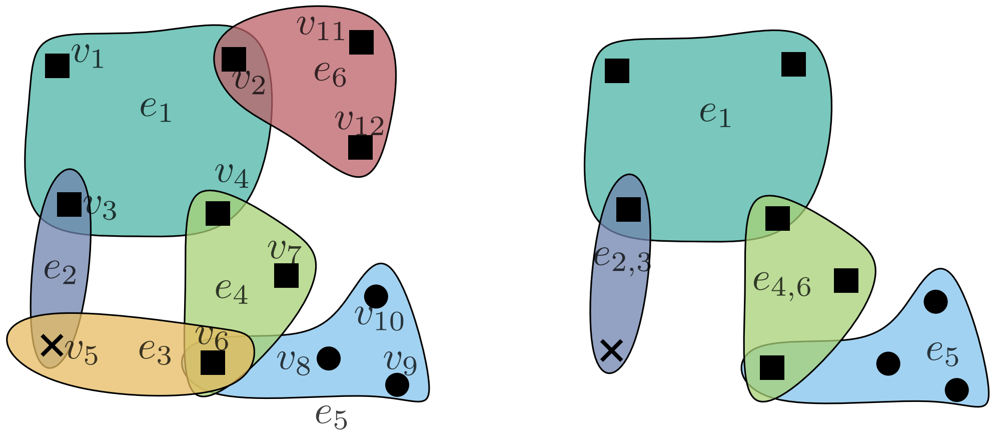

Figure 4.2.: An example of a hypergraph before (left) and after (right) contraction. Note, that in the right hypergraph there are only three nodes left $V = \{●, \times, ■\}$. The multiple occurrences of the different nodes just depict the multiplicities in the corresponding edge. Additionally $\omega(e_{2,3}) = \omega(e_2) + \omega(e_3)$ and $\omega(e_{4,6}) = \omega(e_4) + \omega(e_6)$.

5. Modularity Optimization on Hypergraphs

In Section 4.1, we presented a modularity formulation for hypergraph based on the connectivity of hyperedges. In this section, we explain how we adapt the well-known Louvain algorithm to optimize the hypergraph modularity function. The Louvain algorithm proceeds in rounds. Each round, it iterates over the set of active vertices and moves them to the community that maximizes the change in our objective function. This makes the computation of the deltas the most running time intensive part of the algorithm. Compared to modularity optimization on graphs, the computation of the deltas is significantly more complex on hypergraphs. We will present different techniques to speed up this process in Sections 5.2 and 5.3.

5.1. Algorithm Outline

Algorithm 5.1 gives an overview over the structure of our HYPERGRAPHLOUVAIN implementation. It is the same as for the standard Louvain algorithm (see Section 3.1). Initially, every node belongs to its own singleton community. We then improve upon this initial assignment by executing the LOCALMOVING subroutine (see Algorithm 5.2). Here we iterate over all nodes of H in random order and move them to an adjacent community which yields the largest improvement of the modularity metric. We do this by evaluating the change in connectivity and expected connectivity for each adjacent community of a node and choosing the community with the lowest score < 0. If there is no local move that improves the modularity objective, the node remains in its community. We repeat this process as long as at least 1% of the nodes were moved in the previous round or the maximum number of iterations $l_{\max}$ is reached. We call one complete run of the LOCALMOVING function a *Louvain pass*. After a pass – if the clustering changed – the hypergraph is contracted (see Section 5.4) and we recursively call the HYPERGRAPHLOUVAIN algorithm. Afterward, we project the clustering computed on the coarsest hypergraph to the input hypergraph.

5.2. Tuning the Connectivity Delta Calculation

To calculate the connectivity deltas for a node v, we need to know the weights of hyperedges incident to v, for which v was the only node belonging to the source or the destination community (see Equation (4.5)). A naive approach to compute Δ_{e_H} would be to iterate

Algorithm 5.1: hypergraphLouvain

```
   Input: Hypergraph $H = (V, E, \omega)$
   Output: Clustering $\mathcal{C}$
   // Initializations
1  C ← singleton clustering
   // Local Moving Contract Recurse
2  C' ← localMoving($H$, $C$)                    // See Algorithm 5.2
3  if $C \neq C'$ then
4      $H'$ ←contract($H$, $C'$)                 // See Section 5.4
5      C' ←hypergraphLouvain($H'$)
6      C ←prolong($H$,$C'$)
7  return C
```

Algorithm 5.2: localMoving (Louvain pass)

```
   Input: Hypergraph $H = (V, E, \omega)$, Clustering $\mathcal{C}$
   Output: Clustering $\mathcal{C}$
1  while $\#nodes\ moved \geq 0.01 \cdot n \wedge \#iterations < l_{max}$ do
2      forall $v \in V$ in random order do                // $\Delta_{e_H}$ in Section 5.2
3          $D \leftarrow argmin\{\Delta_{e_H} + \Delta_{\mathbb{E}} | D \in \mathrm{NC}(v)\}$     // $\Delta_{\mathbb{E}}$ in Section 5.3
4          if $D \neq \mathcal{C}(v)$ then
5              moveNode($v, D$)
6  return C
```

over all pins for each of its incident nets to find all adjacent communities and hence, the corresponding change in connectivity. For our connectivity-based modularity metric, we are only interested in whether a hyperedge connects to a community or not. Therefore iterating over all pins is especially problematic in later iterations of a Louvain pass since they might be part of the same community and we gain no additional information by iterating over each of them. This results in an overall running time of $\sum_{e \in I(v)} |e|$, which is in $\mathcal{O}(deg(v) \cdot \max_e |e|)$. The easiest way to improve the running time would be to store each adjacent community for every node. Since this would take $\mathcal{O}(n^2)$ memory, it is not feasible even for medium-sized hypergraphs. Instead, we look for a data structure that allows us to find all connected communities of an edge e in time $\mathcal{O}(\lambda(e))$, i.e. all communities $C \in \mathcal{C}$ with $C \cap e \neq \emptyset$. Additionally, this data structure must also store the number of different nodes it contains for each of these communities, since we are interested in whether v is the last pin of a community C in a hyperedge $e \in I(v)$ to compute Δ_{e_H}.

Community Count Data Structure. To store the communities contained in each hyperedge, we propose the *community count* data structure. It allows us to iterate over all its connected communities in $\mathcal{O}(\lambda(e))$. It realizes updates in $\mathcal{O}(|e|)$ and requires $\mathcal{O}(|e|)$ space.
The data structure consists of an array of size $|e|$, which contains all communities C with $|C \cap e| = 1$. Additionally, we use a hash map to store communities with $|C \cap e| > 1$. We use a combination of an array and a hash map because initially, there are $|e|$ entries in the community count data structure. If we were to insert all of these into the hash map, we would need significantly more space to reduce hash collisions.
We can now iterate over all communities of a hyperedge e by enumerating all communities contained in the vector and hash map. Therefore we achieve our desired running time of

Algorithm 5.3: CONNECTIVITY Δ_{e_H}(*naive*)

```
Input: Hypergraph $H = (V, E, \omega)$, Clustering $\mathcal{C}$, Node $v \in V$
Output: $\Delta_{e_H}$ for every $D \in \mathrm{NC}(v)$
1  $\mathrm{NC}(e) \leftarrow \emptyset$; $\mathrm{NC}(v) \leftarrow \emptyset$; $\Delta_{e_H} \leftarrow \langle 0, \dots, 0 \rangle$
2  forall $e \in I(v)$ do                                        // $\Theta(d(v))$
3      forall pins $u \in e, u \neq v$ do                          // $\mathcal{O}(n)$
4          $\mathrm{NC}(e) \leftarrow \mathcal{C}(u)$
5      if $\mathcal{C}(v) \notin \mathrm{NC}(e)$ then
6          $\Delta_{e_H}(C(v)) \leftarrow \Delta_{e_H}(C(v)) - \omega(e)$
7      forall $D \in \mathrm{NC}(e)$ do
8          $\mathrm{NC}(v) \leftarrow D$
9          $\Delta_{e_H}(D) \leftarrow \Delta_{e_H}(D) - \omega(e)$
10     $\mathrm{NC}(e) \leftarrow \emptyset$
11 $\forall D \in \mathrm{NC}(v) : \Delta_{e_H}(D) \leftarrow \Delta_{e_H}(D) - \Delta_{e_H}(C) + \omega(I(v))$
12 return $\Delta_{e_H}$
```

$\Theta(\lambda(e))$.
We update the community count data structure of an edge $e \in I(v)$ when moving a node v from community C to D, as follows: First, we have to check whether C is stored in the array or the hash map. We can check if it is in the hash map in expected constant time. If it is contained, we simply update the value. If the corresponding value decreases to 0, we remove the element from the hash map, which can also be done in constant time. Note that if the value decreases to 1, we do not remove it and reinsert it in the array.
If the community C is not contained in the hash map, it must be contained in the array. We search for community C by iterating over all its entries. Note that we do not keep the array sorted and use binary search to find elements since an update would also require maintaining the sorting order, which would also require $\mathcal{O}(|e|)$. We can then remove the entry from the array by swapping it with the last element and decrement the size of the array. If the count increases (in the case of D), we additionally have to add it to the hash map, which we can do in expected constant time.
We use this data structure to improve the performance of the connectivity delta calculation as follows. We store a pointer to a community count data structure for each edge e. Since initially all nodes belong to their own community, the array contains all the pins of the edge. We initialize the hash map to be of size $|e|$ since we expect up to $|e|/2$ elements to be inserted, and we do not want the linear probing to become too slow. This is initially as slow as the naive variant but will become significantly faster when the community structure starts to form, as we expect $\lambda(e) \ll |e|$. To update it after a move, we have to iterate over each incident edge of the node and make two calls to the update function.

Caching Threshold While the previously presented data structure significantly speeds up the calculation for larger edges; it is not worth using for edges of size smaller than 16. Therefore we use the naive approach as a fallback strategy for small edges. This also avoids the additional memory required for the community count data structure. We evaluate this threshold in Section 7.2.

5.3. Tuning the Expected Connectivity Delta Calculation

The second part of the change in modularity $\Delta_{mod}(v, C \to D)$ is the change in the probabilistic term. As we can see in Equation (4.6), we have to recalculate the expected

Algorithm 5.4: CONNECTIVITY Δ_{e_H}(*improved*)

```
    Input: Hypergraph $H = (V, E, \omega)$, Clustering $\mathcal{C}$, Node $v \in V$
    Data: Sparse map $\Delta_{e_H}$
    Data: Community count for every edge L
    Output: $\Delta_{e_H}$ for every $D \in \mathrm{NC}(v)$
1 $c \leftarrow 0$                          // $\omega(\{e \in I(v) | \Phi(e, \mathcal{C} \setminus v = 0\})$
2 forall $e \in I(v)$ do                                          // $\Theta(d(v))$
3     forall $D \in L(e)$ do                                      // $\Theta(\mathrm{NC}(v))$
4         if $D = \mathcal{C}(v) \wedge L(e)(D) = 1$ then
5             $c \leftarrow c + \omega(e)$
6         $\Delta_{e_H}(D) \leftarrow \Delta_{e_H}(D) - \omega(e)$
7 $\forall D \in \Delta_{e_H} : \Delta_{e_H}(D) \leftarrow \Delta_{e_H}(D) - c + \omega(I(v))$
8 return $\Delta_{e_H}$
```

Algorithm 5.5: EXPECTEDCONNECTIVITY $\Delta_{\mathbb{E}}$ (naive)

```
    Input: Hypergraph $H = (V, E, \omega)$
    // Initializations
1 $\Delta_{best} \leftarrow 0$
2 $D_{best} \leftarrow C$
    // Main Loop
3 forall communities $D \in \mathrm{NC}(v)$ do
4     forall edge sizes $d \in \mathcal{D}$ do
5         $\Delta_{\mathbb{E}} \leftarrow \Delta_{\mathbb{E}} + W_d \cdot$
            $\left[\left(1 - \frac{\mathrm{vol}(C^-)}{\mathrm{vol}(V)}\right)^d - \left(1 - \frac{\mathrm{vol}(C)}{\mathrm{vol}(V)}\right)^d + \left(1 - \frac{\mathrm{vol}(D^+)}{\mathrm{vol}(V)}\right)^d - \left(1 - \frac{\mathrm{vol}(D)}{\mathrm{vol}(V)}\right)^d\right]$
6     $\Delta_{mod} \leftarrow \Delta_{e_H} + \Delta_{\mathbb{E}}$
7     if $\Delta_{mod} < \Delta_{best}$ then
8         $\Delta_{best} \leftarrow \Delta_{mod}$
9         $D_{best} \leftarrow D$
```

connectivity for both the source and the destination community.
To determine the best neighboring community, we have to calculate the change in expected connectivity. Therefore, for each community in the neighborhood and each unique edge size $d \in \mathcal{D}$ we have to raise four terms to the d-th power, which takes $\mathcal{O}(\log(d))$ for arbitrary edge sizes d. The overall running time of this procedure results in $\mathcal{O}(|\mathrm{NC}| \cdot |\mathcal{D}| \cdot \log(d_{max}))$, where d_{max} is the largest edge size occurring in the hypergraph. We illustrate this naive algorithm in Algorithm 5.5. In this chapter, we present the techniques which we introduce to improve the running time in practice.

Pruning One significant factor in the running time of the algorithm is the number of communities for which we have to compute the expected connectivity. Therefore reducing this number is crucial to improve the performance. Given a neighboring community D of v, the idea is to determine whether the resulting change in modularity $\Delta_{mod}(v, C \to D)$ will be better than the value for the currently best destination community. We will introduce the following three rules for pruning that allow us to efficiently skip neighboring communities without calculating the expected connectivity:

- if $\mathrm{vol}(C) = \mathrm{vol}(D^+)$ we know that $\Delta_{\mathbb{E}}$ will be 0, therefore we can skip its calculation

- if the change in connectivity $\Delta_{e_H} \geq 0$ and $\mathrm{vol}(C) \leq \mathrm{vol}(D^+)$ and therefore $\Delta_{\mathbb{E}} \geq 0$ we can skip the calculation of $\Delta_{\mathbb{E}}$ since the overall change in modularity will not be negative and therefore will not result in an improvement.
- let Δ_{best} be the best change in modularity found so far for a given community C and a node v. If $\Delta_{e_H} \geq \Delta_{best}$ and $\mathrm{vol}(C) \leq \mathrm{vol}(D^+)$ we can skip calculating $\Delta_{\mathbb{E}}$.

We will prove the correctness of these rules in the Lemma 5.1. Let $C \in \mathcal{C}$ be a community in the clustering $\mathcal{C}$ and $v \in C_i$ then we define $\mathbb{E}_d(C^-) = (\mathrm{vol}(V) - \mathrm{vol}(C) + \mathrm{vol}(v))^d > 0$ and $\mathbb{E}_d(C) = (\mathrm{vol}(V) - \mathrm{vol}(C))^d \geq 0$.

We make the following observation, which will be used in the proof of Lemma 5.1: For a fixed d

$$\forall C \in \mathcal{C} : \mathrm{f}(C_i) = \mathbb{E}_d(C^-) - \mathbb{E}_d(C) < 0$$

is strictly monotonically decreasing for $\mathrm{vol}(C) \in [\mathrm{vol}(v), \mathrm{vol}(V)]$. Note that we can also formulate this for the destination community D:

$$\forall D \in \mathcal{C} : \mathbb{E}_d(D^+) - \mathbb{E}_d(D) = \mathbb{E}_d(D^+) - \mathbb{E}_d(D^+ \setminus v) = -\,\mathrm{f}(D^+)$$

This leads to the observation that the bigger a community, the bigger the absolute change in expected connectivity for this community when moving a node from or to it.

Lemma 5.1. *Sign of the change in expected connectivity*

1. $\mathrm{vol}(C) = \mathrm{vol}(D^+) \Rightarrow \Delta_{\mathbb{E}} = 0$

2. $\mathrm{vol}(C) > \mathrm{vol}(D^+) \Rightarrow \Delta_{\mathbb{E}} < 0$

3. $\mathrm{vol}(C) < \mathrm{vol}(D^+) \Rightarrow \Delta_{\mathbb{E}} > 0$

Proof. We can write the formula for the expected connectivity from Equation (4.6) as follows:

$$\Delta_{\mathbb{E}} = \sum_{d \in \mathcal{D}} \frac{W_d}{\mathrm{vol}(V)^d} \left[\mathbb{E}_d(C^-) - \mathbb{E}_d(C) + \mathbb{E}_d(D^+) - \mathbb{E}_d(D) \right]$$

Since W_d, $\mathrm{vol}(V) > 0$ for all edge sizes $d \in \mathcal{D}$ we can ignore these factors of $\Delta_{\mathbb{E}}$ because then don't change the sign of the result. If we can now prove that the partial sums

$$\mathbb{E}_d(C^-) - \mathbb{E}_d(C) + \mathbb{E}_d(D^+) - \mathbb{E}_d(D) = \mathrm{f}_d(C) - \mathrm{f}_d(D^+)$$

all have the same sign or, in the first case, are all zero, we are done.

1. Let $\mathrm{vol}(C) = \mathrm{vol}(D^+)$, then

$$\forall d : \mathrm{f}_d(C) - \mathrm{f}_d(D^+) = \mathrm{f}_d(C) - \mathrm{f}_d(C) = 0$$

2. Let $\mathrm{vol}(C) > \mathrm{vol}(D^+)$, then

$$\forall d : \mathrm{f}_d(C) - \mathrm{f}_d(D^+) < \mathrm{f}_d(C) - \mathrm{f}_d(C) = 0$$

3. Let $\mathrm{vol}(C) < \mathrm{vol}(D^+)$, then

$$\forall d : \mathrm{f}_d(C) - \mathrm{f}_d(D^+) > \mathrm{f}_d(C) - \mathrm{f}_d(C) = 0$$

□

Speedup Heuristics The calculation of the probabilistic term takes up a significant part of the running time of our algorithm. To counteract this, we introduce the heuristic to only evaluate the expected connectivity for the s best neighboring communities based on their connectivity. We conduct an experiment on what to choose for s in Section 7.2, which will show that $s = 100$ the algorithm is significantly faster while maintaining the quality. In the following, we will take a look at why this is the case.
During early experiments, we investigated the impact of the probabilistic term on the choice of the neighboring community. More precisely, we calculated the connectivity for all communities $D \in \mathrm{NC}(v)$ and then sorted these in increasing order (a value < 0 corresponds to an improved modularity score) according to their connectivity. Then we calculated the probabilistic term for each of them and sorted the communities according to their final change in modularity. As a result, we could look at the initial ranking of the community corresponding to the best modularity delta, based on connectivity. We can then use this information to determine the number of neighboring communities in $NC(v)$ sorted by connectivity, we would have to calculate the probabilistic term for, in order to find the move with the biggest improvement. To illustrate this, we collected the initial rankings of all best moves for each instance. The x-axis in Figure 5.1 shows the ranking per instance, while the y-axis shows the fraction of instances for which the 92nd percentile was equal or less than the x-value.

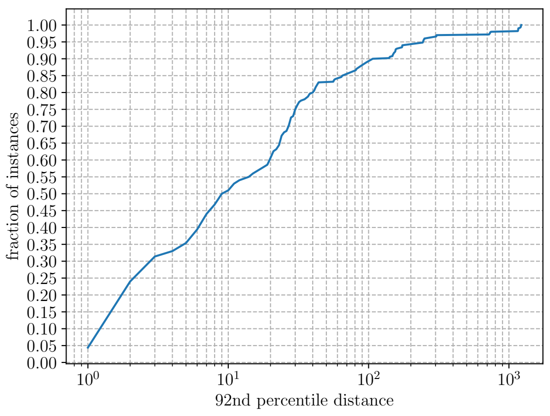

Figure 5.1.: 92nd percentile of the ranking of the best move based on the connectivity

As we can see in Figure 5.1 when only choosing the best 100 communities based on their connectivity, we also select the best possible community in 92% of the cases for 90% of the instances. This does not mean that if the actual best adjacent community is not among the 100 selected, the calculated move is significantly worse. We will see in Section 7 that this indeed does not impact the performance negatively. Another reason for this is that we only calculate locally optimal moves. Therefore another move might lead to a series of moves that yield a bigger improvement of the modularity objective.
Additionally, preliminary experiments showed that only a tiny part of the best moves worsens the coverage score, i.e., the change in connectivity Δ_{e_H} is greater than zero. Therefore we also skip the calculation of $\Delta_{\mathbb{E}}$ for communities that increase the connectivity.

Reducing the cost of Exponentiation. We can reduce the cost of exponentiation in line 5 of Algorithm 5.5 in two ways. Firstly, since the first two summands $\left(1 - \frac{\mathrm{vol}(C^-)}{\mathrm{vol}(V)}\right)^d - \left(1 - \frac{\mathrm{vol}(C)}{\mathrm{vol}(V)}\right)^d$ only incorporate the source community C, we can precalculate these terms. Note that we don't have to calculate the terms including C at all, if all neighboring communities $D \in \mathrm{NC}(v)$ are sorted out by the previously discussed pruning methods. We iterate through $\mathcal{D}$ in ascending order. By storing the result of $\mathbb{E}_{d_{prev}}(x) = \left(1 - \frac{\mathrm{vol}(x)}{\mathrm{vol}(V)}\right)^{d_{prev}}$

Algorithm 5.6: expectedConnectivity $\Delta_{\mathbb{E}}$ (improved)

```
   Input: Hypergraph $H = (V, E, \omega)$
   // Initializations
 1 $\Delta_{best} \leftarrow 0$
 2 $D_{best} \leftarrow C$
   // Main Loop
 3 forall $D \in \mathrm{NC}(v)$ do
 4     $\Delta_{\mathbb{E}} \leftarrow 0$
 5     if $\Delta_{e_H} \geq \Delta_{best} \wedge \mathrm{vol}(C) \leq \mathrm{vol}(D^+)$ then
 6         continue
 7     if $\mathrm{vol}(C) \neq \mathrm{vol}(D^+)$ then
 8         forall $d \in \mathcal{D}$ do                                  // Initialize only once
 9             $\mathbb{E}_d \leftarrow \left(\frac{\mathrm{vol}(C^-)}{\mathrm{vol}(V)}\right)^d - \left(\frac{\mathrm{vol}(C)}{\mathrm{vol}(V)}\right)^d$
10         forall $d \in \mathcal{D}$ do
11             $\Delta_{\mathbb{E}} \leftarrow \Delta_{\mathbb{E}} + W_d \cdot \left[\mathbb{E}_d + \left(1 - \frac{\mathrm{vol}(D^+)}{\mathrm{vol}(V)}\right)^d - \left(1 - \frac{\mathrm{vol}(D)}{\mathrm{vol}(V)}\right)^d\right]$
12     $\Delta_{mod} \leftarrow \Delta_{e_H} + \Delta_{\mathbb{E}}$
13     if $\Delta_{mod} < \Delta_{best}$ then                   // if equal $\rightarrow$ tie-breaking strategy
14         $\Delta_{best} \leftarrow \Delta_{mod}$
15         $D_{best} \leftarrow D$
```

for the previous edge size d_{prev} and for each $x \in \{C, C^-, D, D^+\}$, we can instead calculate $\mathbb{E}_d(x) = \mathbb{E}_{d_{prev}} \cdot \mathbb{E}_{d-d_{prev}}(x)$.

5.4. Contraction

To contract a clustering, we use the parallel contraction algorithm already implemented in the `Mt-KaHyPar` framework [GHSS20]. It internally uses a parallelization of the InrSrt algorithm by Aykanat et al. [ACU08, DKC13] for identical net detection. In the following, we will only highlight the features important to our work here and refer to the corresponding paper for more details. The algorithm remaps the contracted communities to a consecutive range in $[1, |\mathcal{C}|]$. We explicitly store the volumes $\mathrm{vol}(C)$ for every community $C \in \mathcal{C}$ and the accumulated edge weights W_d for each $d \in \mathcal{D}$. This allows us, in contrast to the contraction described in Section 4.1, to remove duplicate pins in edges as well as edges only containing a single pin. We illustrate this difference in Figure 5.2.
Additionally to contracting the hypergraph, we have to update our data data structures accordingly. We remap the volumes according to the mapping calculated in the contraction algorithm. Instead of remapping each entry of every community count data structure we construct a new one for each edge in the coarse hypergraph.

5.5. Parallelization

To parallelize the presented algorithm, we iterate over all nodes in random order in parallel (see Line 2 in Algorithm 5.2). We use two different kinds of thread-local hash maps to store the connectivity Δ_{e_H} for each neighboring community of a node. We roughly estimate the number of neighbors as $\sum_{e \in I(v)} |e|$. If this estimate is small enough, we use a cache efficient fixed-size hash map, which is roughly 1 MB, and otherwise, we use a sparse map [BT93] using the number of nodes n as size. To update the data structures when moving a

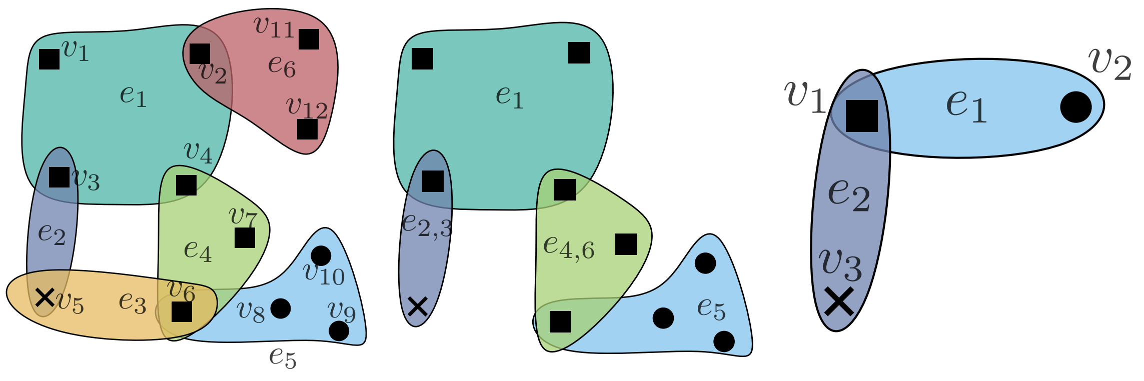

Figure 5.2.: An example of a hypergraph before (left), after theoretical contraction (middle) and optimized contraction (right).

node in parallel, we have to pay special attention to keep them consistent. We do this by atomically updating the volumes and using spinlocks whenever we update a community count data structure. Additionally, we parallelize the projection of the communities to the finer hypergraph, as well as the volume remapping and construction of the community count data structures in the contraction algorithm.

5.6. Implementation Details

In this section, we will briefly go over some relevant implementation details. Before running the community detection algorithm, we remove all large hyperedges e with $|e| > 0.01 \cdot n$. We do not expect them to contain any information about the community structure and they can impact the running time significantly, since both the number of different edge sizes $|\mathcal{D}|$ and the maximum edge size d_{max} are a factor in the running time of the delta calculation.
We implement the speedup heuristic, to only select the best s neighboring communities based on their connectivity, using the `std::nth_element` procedure. It partitions the neighboring communities so that the change in connectivity of the first s communities is smaller than of the rest of the neighboring communities while not sorting it.
The hash map we use in the community count data structure needs up to $3 \cdot |e|$ space. It consists of a dense vector where we store the positions of corresponding entries in the sparse part. The sparse part is accessed via a hash function, and hash collisions are resolved using linear probing. It is an array of size $2 \cdot |e|$ and stores a tuple (D, c, p) consisting of community D, the number of nodes c in e belonging to D, and the position of its corresponding dense element p. This allows us to update, insert and delete any element from the hash map in expected constant time. Additionally, we can iterate over all entries of the hash map by simply iterating over the dense vector.

6. Hypergraph Map Equation

In contrast to the topological approach of modularity, we will take a look at an information-theoretic approach to community detection in this chapter. The map equation, which was originally presented by Rosvall et al. [RAB09], aims to use the duality between the minimum description length of a random walk through a graph and its community structure. Therefore when optimizing the map equation, we minimize the encoding of the random walk by partitioning the graph into communities using their relative visit frequencies. Additionally, its resolution limit is less restrictive in practice [KR15], and in contrast to modularity, the detected communities do not tend to favor balance as much. Therefore the community structure found by the map equation approach may differ significantly from the one found by a modularity approach.
In this chapter, we will first take a brief look at the definition of the map equation to then formulate a random walk model on hypergraphs. We will use this model to define the map equation on hypergraphs in Section 6.1. In Section 6.2 we will calculate the change in the objective function when moving a node from one community to another. This change, called delta, is needed to optimize the map equation using the Louvain algorithm, which repeatedly evaluates this delta for each neighboring community of a node. Finally, we will briefly highlight the differences in the implementation to the hypergraph modularity approach in Section 6.3.

6.1. Formulating the Hypergraph Map Equation

We can use the same definition for the map equation as originally proposed by Rosvall et al. [RAB09] since its general formulation is solely based on the average description length to encode a random walk. We will quickly refresh its definition:
Given a community clustering $\mathcal{C}$, the map equation is defined as

$$L(\mathcal{C}) = q_{\curvearrowright} H(Q) + \sum_{C_i \in \mathcal{C}} p^i_{\circlearrowright} H(C_i)$$

where $q_{\curvearrowright}$ is the probability to leave any community and $p^i_{\circlearrowright}$ is the probability to be in community i plus the probability to leave it.

$$q_{\curvearrowright} = \sum_{C_i \in C} q_{i\curvearrowright} \text{ and } p^i_{\circlearrowright} = \sum_{u \in C_i} p_u + q_{i\curvearrowright}$$

$$H(Q) = -\sum_{C_i \in C} \frac{q_{i\curvearrowright}}{q_{\curvearrowright}} \log\left(\frac{q_{i\curvearrowright}}{q_{\curvearrowright}}\right)$$

$$H(C_i) = -\frac{q_{i\curvearrowright}}{p^i_{\circlearrowright}} \log\left(\frac{q_{i\curvearrowright}}{p^i_{\circlearrowright}}\right) - \sum_{u \in C_i} \frac{p_u}{p^i_{\circlearrowright}} \log\left(\frac{p_u}{p^i_{\circlearrowright}}\right)$$

We will use the following formulation, which can be obtained by plugging in the definitions and some algebra in the rest of this section. Let $\operatorname{plogp}(x) = x \cdot \log(x)$:

$$\begin{aligned} L(C) = &\operatorname{plogp}\left(\sum_{C_i \in C} q_{i\curvearrowright}\right) - 2\sum_{C_i \in C} \operatorname{plogp}\left(q_{i\curvearrowright}\right) \\ &+ \sum_{C_i \in C} \operatorname{plogp}\left(q_{i\curvearrowright} + \sum_{u \in C_i} p_u\right) - \sum_{u \in V} \operatorname{plogp}\left(p_u\right) \end{aligned} \tag{6.1}$$

We omit the last part of the formula since it is constant throughout the whole optimization process. In Equation (6.1), we can see which parts we will have to define for hypergraphs in order to use the map equation for community detection: The exit probabilities $q_{i\curvearrowright}$ for each community C_i and stationary distribution p_u, i.e the average node visit frequency for every node $u \in V$.

Random walks on Hypergraphs To define a random walk, we first have to define the transition probabilities. That is, the probability for the random walker to move from one node to another in a single time step. In the dyadic case, one simply chooses an incident edge of the node the random walker is currently on, uniformly at random. Since, in this case, edges are only of size two, choosing an edge directly determines the next node for the random walker. In the hypergraph case, there are edges of arbitrary sizes, and nodes can be connected by more than one hyperedge. Additionally, a node can be contained in an edge multiple times, which we also need to consider. Therefore we first have to choose an edge with respect to the number of occurrences of our current node in it and then choose the node for the next time step among the pins of said hyperedge. We can formalize this in the following two-step process: If the random walker currently resides on node $u \in V$:

1. Choose an edge e from the incident edges of u $I(u)$, with probability $\frac{\Phi(e,u)\cdot\omega(e)}{\operatorname{vol}(u)}$.
2. Choose a node from all pins of edge e, with probability $\frac{1}{|e|}$. In the case that there are multiple pins of a node $v \in V$, the probability would be relative to its pin count in e, like $\frac{\Phi(e,v)}{|e|}$. Note that this means a walker can stand still.

In this formulation, the same node u that the random walker is currently on can be chosen as the node for the next time step. This is called a lazy walk. Note that this process is similar to the random walk process described by Chitra and Raphael [CR19] when choosing the edge dependent vertex weights $\gamma_e(u)$ as the relative number of pins of the node in the edge $\frac{\Phi(e,v)}{|e|}$. The difference is in the way we choose an edge: While Chitra and Raphael choose the edge only dependent on its relative weight ($\frac{\omega(e)}{\omega(I(u))}$), we also take the pin count of the node in edge e into account ($\frac{\Phi(e,u)\cdot\omega(e)}{\operatorname{vol}(u)}$).

We will also take a look at a *less-lazy* random walk, where the random walker must not stand still on the same pin but can move to another pin of the same node. In this case the probability to choose a node is $\frac{\Phi(e,v)}{|e|-1}$. The resulting formulas are identical to the lazy-random walk when replacing $|e|$ with $|e-1|$, except for the deltas, where the last equality is not true. See Appendix C for the complete calculations.

Another possibility is to look at non-lazy random walks. Here a random walker can only move to a different node and cannot stand still at all. in this case the probability for the second step ends up as $\frac{\Phi(e,v)}{|e|-\Phi(e,u)}$. We leave this formulation for future work.

Stationary Distribution Additionally, we want to know the probability for the random walker to be on any given edge at any given time. This is the average frequency a random walker visits a node in a random walk of infinite length. A random walk on a hypergraph is a Markov chain on V with the transition probabilities between nodes u and v:

$$P_{uv} = \sum_{e \in I(u)} \frac{\Phi(e,u)\omega(e)}{\text{vol}(u)} \cdot \frac{\Phi(e,v)}{|e|}$$

We can now calculate the transition probability between two communities C_i and C_j, as the sum of probabilities to be on any node $u \in C_i$ and move to any node $v \in C_j$:

$$\begin{aligned} P_{ij} &= \sum_{e \in I(C_i)} \sum_{u \in C_i} \frac{\text{vol}(u)}{\text{vol}(C_i)} \sum_{v \in C_j} \frac{\Phi(e,u)\omega(e)}{\text{vol}(u)} \cdot \frac{\Phi(e,v)}{|e|} \\ &= \sum_{e \in I(C_i)} \sum_{u \in C_i} \frac{\Phi(e,u)\omega(e)}{\text{vol}(C_i)} \cdot \frac{\Phi(e,C_j)}{|e|} \\ &= \sum_{e \in I(C_i)} \frac{\Phi(e,C_i)\omega(e)}{\text{vol}(C_i)} \cdot \frac{\Phi(e,C_j)}{|e|} \end{aligned}$$

If the Markov chain starts with a distribution of average node visit frequencies and this distribution stays the same at any given time step, it is called a stationary distribution. In community detection, we are only interested in strongly connected graphs since we can otherwise perform community detection on each of the strongly connected components. This makes the Markov chain M representing our random walk recurrent. Since we need M to be aperiodic, we also have to exclude bipartite graphs because a Markov Chain representing such a graph would have period 2. With this in mind, our Markov chain is aperiodic and recurrent and thus ergodic. Therefore p is the stationary distribution exactly when

$$(pP)_v = p_v \; \forall v \in V$$

Lemma 6.1. *For every node $v \in V$ the stationary distribution p of M is given by*

$$p_v = \frac{\text{vol}(v)}{\text{vol}(V)}$$

Proof.

$$\begin{aligned} (pP)_v &= \sum_{u \in V} p_u P_{uv} \\ &= \sum_{u \in V} \frac{\text{vol}(u)}{\text{vol}(V)} \sum_{e \in I(u)} \frac{\Phi(e,u)\omega(e)}{\text{vol}(u)} \cdot \frac{\Phi(e,v)}{|e|} \end{aligned} \tag{6.2}$$

Taking a look at the inner sum, we can see that a summand is zero exactly when either u or v aren't in the edge e, which corresponds to m_u or $m_v = 0$. Therefore only edges $e \in E$ with $\Phi(e,u) > 0 \wedge \Phi(e,v) > 0$ contribute to the sum. We can now use that $I(u) \cap I(v) = \{e \in E | \Phi(e,u) > 0 \wedge \Phi(e,v) > 0\}$, so every edge that contributes to the sum has to be in both $I(u)$ and $I(v)$. Therefore it is equal to iterate over all incident edges of v instead of u.

$$\begin{aligned}(pP)_v &= \frac{1}{\mathrm{vol}(V)} \sum_{u\in V} \sum_{e\in I(v)} \Phi(e,u)\cdot\omega(e)\cdot\frac{\Phi(e,v)}{|e|} \\ &= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} \Phi(e,v)\omega(e)\frac{1}{|e|}\sum_{u\in V}\Phi(e,u) \\ &= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} \Phi(e,v)\omega(e) \\ &= \frac{\mathrm{vol}(v)}{\mathrm{vol}(V)}\end{aligned} \tag{6.3}$$

□

Finally we define the exit probability $q_{i\curvearrowright}$, which is the probability, that the next move of the random walker is to leave community C_i. In other words, it is the probability for the random walker to be in community C_i and moving to any other community C_j in the next step. Again, we give the full derivation in Appendix B.

$$\begin{aligned}q_{i\curvearrowright} &= \left(\sum_{u\in C_i} p_u\right)\cdot\sum_{C_j\in C} P_{ij} \\ &= \frac{1}{\mathrm{vol}(V)}\sum_{e\in I(C_i)}\frac{\Phi(e,C_i)\cdot\omega(e)}{|e|}\cdot(|e|-\Phi(e,C_i))\end{aligned} \tag{6.4}$$

6.2. Change in Map Equation

We have now fully defined all parts of the hypergraph map equation. Since we will use the Louvain algorithm, which uses local moving, we want to know how much the map equation changes if we move a node v from its current community C_i to another neighboring community C_j. To calculate this, we have to look at the components of the map equation that change. The only ones that change are the exit probabilities $q_{i\curvearrowright}$ and $q_{j\curvearrowright}$ for the participating communities.

To calculate the change in exit probability for a community C_i we first subtract the previous contributions of the incident edges of v and then add the contributions of these edges if v was removed from C_i.

$$\begin{aligned}\Delta q_{i\curvearrowright} &= \frac{1}{\mathrm{vol}(V)}\sum_{e\in I(v)} \overbrace{-\,\Phi(e,C_i)\omega(e)\frac{|e|-\Phi(e,C_i)}{|e|}}^{\text{old contribution}} + \overbrace{\Phi(e,C_i\setminus v)\omega(e)\frac{|e|-\Phi(e,C_i\setminus v)}{|e|}}^{\text{new contribution}} \\ &= \frac{1}{\mathrm{vol}(V)}\left(\left(\sum_{e\in I(v)}\omega(e)\Phi(e,v)\frac{2\Phi(e,C_i)-\Phi(e,v)}{|e|}\right)-\mathrm{vol}(v)\right)\end{aligned} \tag{6.5}$$

For the destination community C_j, this results in:

$$\begin{aligned}\Delta q_{j\curvearrowright} &= \frac{1}{\mathrm{vol}(V)}\sum_{e\in I(v)} \overbrace{-\,\Phi(e,C_j)\omega(e)\frac{|e|-\Phi(e,C_j)}{|e|}}^{\text{old contribution}} + \overbrace{\Phi(e,C_j\cup v)\omega(e)\frac{|e|-\Phi(e,C_j\cup v)}{|e|}}^{\text{new contribution}} \\ &= \frac{1}{\mathrm{vol}(V)}\left(\left(\sum_{e\in I(v)}\omega(e)\Phi(e,v)\frac{-2\Phi(e,C_j)-\Phi(e,v)}{|e|}\right)+\mathrm{vol}(v)\right)\end{aligned} \tag{6.6}$$

For more details on the derivations, see Appendix B. To calculate the change in map equation, we then evaluate all parts of the formula affected by the particular communities, once with the old exit probabilities $q_{i\curvearrowright}$ and once with the updated probabilities and subtract them from each other. Let

$$\begin{aligned} L_{old}(C,i,j,v) = \operatorname{plogp}\left(\sum_{C_k \in C} q_{k\curvearrowright}\right) - 2\cdot\left(\operatorname{plogp}\left(q_{i\curvearrowright}\right) + \operatorname{plogp}\left(q_{j\curvearrowright}\right)\right) \\ + \operatorname{plogp}\left(q_{i\curvearrowright} + \sum_{u\in C_i} p_u\right) + \operatorname{plogp}\left(q_{j\curvearrowright} + \sum_{u\in C_j} p_u\right) \end{aligned} \tag{6.7}$$

be the parts of $L(C)$ affected by moving node v from community C_i to community C_j before executing the move and

$$\begin{aligned} & L_{new}(C,i,j,v) = \\ & -\operatorname{plogp}\left(\Delta q_{i\curvearrowright} + \Delta q_{j\curvearrowright} \sum_{C_k\in C} q_{k\curvearrowright}\right) - 2\cdot\left(\operatorname{plogp}\left(q_{i\curvearrowright}\Delta q_{i\curvearrowright}\right) + \operatorname{plogp}\left(q_{j\curvearrowright}\Delta q_{j\curvearrowright}\right)\right) \\ & + \operatorname{plogp}\left(q_{i\curvearrowright} + \Delta q_{i\curvearrowright} - p_v + \sum_{u\in C_i} p_u\right) + \operatorname{plogp}\left(q_{j\curvearrowright} + \Delta q_{j\curvearrowright} + p_u + \sum_{u\in C_j} p_u\right) \end{aligned} \tag{6.8}$$

the same parts calculated with the updated values.
We can then calculate the final change in Map Equation by subtracting $L_{new}(C,i,j,v)$ from $L_{old}(C,i,j,v)$.

$$\Delta L(C,i,j,v) = L_{old}(C,i,j,v) - L_{new}(C,i,j,v) \tag{6.9}$$

Since we want to minimize the map equation and we subtract the new (hopefully smaller value) from the old value, we choose the biggest change in Map Equation $\Delta L(C,i,j,v) > 0$ as the best possible move.

6.3. Implementation

To optimize the hypergraph map equation, we implement the RelaxMap algorithm by Bae et al. [BHW⁺17], which is a parallelization of the Louvain algorithm with some adaptation to fit the map equation. Therefore the structure of the algorithm stays the same: We alternate the local moving and contraction algorithms until the clustering can not be improved and then map the clustering of the coarse hypergraph to the original hypergraph. We will briefly highlight the differences to the modularity approach, including the delta calculation, the updates needed when making a move and the difference in the contraction. Note that our implementation is not optimized. We use it as a comparison to our modularity approach when integrating them into the `Mt-KaHyPar` framework as a preprocessing step.

Delta Calculation To calculate the change in map equation, we store the sum of exit probabilities $\sum_{C_i\in\mathcal{C}} q_{i\curvearrowright}$, the exit probability for each community $q_{i\curvearrowright}$ in an array of size n as well as the volume for each community $\operatorname{vol}(C_i)$ and node $\operatorname{vol}(v)$. In contrast to the modularity approach, we need to know the exact number of pins $\Phi(e,v)$ for each node v in an edge e. Therefore we store pins of an edge as tuples (v, m_v), we call multipins, where v is a node contained in e and m_v is its multiplicity in e. We calculate the deltas for the source community and each neighboring communities from Equations (6.5) and (6.6), by iterating over each incident edge $e \in I(v)$ and each multipin of that edge. Therefore the running time of this procedure is in $\sum_{e \in I(v)} |e|$. The evaluation of Equation (6.9) is straight forward. We break ties by always choosing the community with the smaller ID.

Moving a Node When moving a node v from community C_i to community C_j, we have to update our data structures. Before making any updates, we acquire a lock for community C_i and C_j each. We then recompute the change in the exit probabilities $\Delta q_{i\curvearrowright}$ and $\Delta q_{j\curvearrowright}$ and reevaluate the change in map equation to make sure this move is actually an improvement. After that, we update the $\mathrm{vol}(C_i)$ and $\mathrm{vol}(C_j)$ by subtracting or adding the volume of v respectively as well as the exit probabilities $q_{i\curvearrowright}, q_{j\curvearrowright}$ and the sum of exit probabilities via the calculated deltas. This introduces an error produced by the limited precision of floating-point numbers. We try to compensate for this by recalculating all exit probabilities in parallel after iterating over all nodes once.

Contraction While we still use the same algorithm to contract the hypergraph as in the modularity approach, we have to make some restrictions on what to contract. Since we need the multiplicities of pins for each edge, we can not remove duplicate pins from edges. As a result, the number of edges decreases significantly less than in the modularity approach.

7. Experimental Results

First, we present the experimental setup in Section 7.1, which we used to conduct our experiments. After that, we will evaluate different configurations and optimizations of our algorithm using hypergraph modularity based on connectivity in Section 7.2, as well as its scalability in Section 7.3. In Section 7.4 we will compare our algorithm to the current implementation of `Mt-KaHyPar-D`, which uses modularity based on the bipartite graph representation. Finally, we briefly compare the modularity approach with the map equation approach in Section 7.5.

7.1. Setup and Methodology

We implemented the algorithms described in Section 5 and 6.3 in the shared memory partitioning framework `Mt-KaHyPar`. The code is written in C++ and compiled using g++-9.2 with the flags -O3 -mtune=native -march=native. We parallelized our implementation with work-stealing using the TBB library [Phe08]. We refer to the original algorithm as `D`, our hypergraph modularity version as `HM` and the hypergraph map equation versions as `HME` and `HME-1` respectively.

Instances. To evaluate the performance of our algorithm, we use two different benchmark sets. Table 7.1 provides an overview. Set A consists of 488 hypergraphs, with unit net and vertex weights, by Heuer and Schlag [HS17]. It consists of instances from the ISPD98 VLSI Circuit Benchmark Suite (ISPD98) [Alp98], the DAC2012 Routability-Driven Placement Contest (DAC)[VAS$^+$12], sparse matrix instances from the SuiteSparse Matrix Collection (SPM) [DH11] and SAT formulas from the 2014 SAT Competition (Literal, Primal and Dual) [BDHJ]. We interpret the sparse matrices as hypergraphs using the row-net model [CA99], where each row corresponds to a net, and each column corresponds to a node in the hypergraph. We derive three different hypergraphs from each SAT instance. In the *literal* model proposed by Papa and Markov [PM07], each boolean literal is represented by a node and each clause by a hyperedge. Papp and Mann [MP14] proposed two more representations: primal and dual. For the *primal* model, we interpret each variable as a node and each clause as a hyperedge. For the *dual* model, it is the other way around, i.e., each variable is represented by a hyperedge, and clauses translate to nodes. We will use this benchmark set to compare our algorithm to the current implementation of `Mt-KaHyPar-D`.

We use a subset of 100 instances as a parameter tuning subset, containing instances from all different types, chosen by Schlag [Sch20] to produce the same qualitative result as the

Table 7.1.: Composition of the used Sets

Class	Set A	Set B	Set C
DAC2012	10	10	4
ISPD98	18	0	10
SAT14 - Primal	92	14	18
SAT14 - Dual	92	14	18
SAT14 - Literal	92	14	18
SPM	184	42	32
$\sum$	488	94	100

whole set A. We will call it set C in the following. Additionally we use a set of 94 large hypergraphs by Gottesbüren et al. [GHSS20] to conduct scalability experiments. We call this set of larger instances set B. It consists of a random sample of 42 instances from the SuiteSparse Matrix Collection with more than 15 million non-zero entries, as well as all DAC instances, the 24 largest SAT instances from set A and an additional 18 larger SAT instances from the 2014 SAT Competition.

Systems. Due to availability issues, experiments involving parameter tuning and evaluation of the optimizations are performed on a machine consisting of two Intel Xeon E5-2683 16 core processors clocked at 2.1 GHz. The machine runs Ubuntu 18.04, has 512 GB of main memory and 40 MB of Cache.
We perform the experiments involving set B on a different machine. It consists of an AMD EPYC Rome 7702P with 64 cores clocked at 2.0 - 3.35 GHz. The machine runs Ubuntu 20.04, has 1024 GB of main memory and 256 MB of L3-Cache.
To compare our hypergraph modularity approach with the current graph-based approach, we use a machine consisting of 2 Intel Xeon Gold 6230 processors. A single processor has 20 cores and runs at 2.1 GHz with 96GB RAM.

Performance Profiles. To visualize the performance of different algorithms, we use a measure called *performance profiles*, which was first introduced by Dolan and Moré [DM02]. We compare the quality of the results produced by the different algorithms using the $(\lambda - 1)$-metric. In the plots, each colored line represents a different algorithm or configuration. The x-axis displays the quality τ relative to the best result of all algorithms in the graph, which we call *performance ratio*. The y-axis displays the fraction of instances for which the quality, compared to the best partition, is worse by a factor of τ or less. For example, an algorithm with the value of 0.75 on the y-axis for a value of 2 on the x- axis, i.e. the performance ratio $\tau = 2$. This means that the quality of the resulting partitions of the algorithm are worse than the best by a factor of up to 2 on 75% of the instances. The y-axis for $\tau = 1$ shows the fraction of instances the algorithm produced the best results out of all the algorithms in the plot. Therefore, when comparing two algorithms, if the fraction of instances is 0.5 for $\tau = 1$, both algorithms produce the best partition on an equal amount of instances. Generally speaking, an algorithm performs better than others if its line in the plot is above the others. If needed, we divide the x-axis into three parts. The first part goes up to 1.1, showing the fraction of instances where the algorithm is within 10% of the best algorithm. The second one goes up to 2, showing instances of medium quality in comparison. The last section, displaying the worst instances, uses a logarithmic scale for all remaining instances.

7.2. Configuring the Algorithm

In this section, we evaluate different configurations of our hypergraph modularity algorithm on our parameter tuning benchmark subset. We need to evaluate the following configurations and tuning parameters:

- The maximum number of iterations per Louvain pass l_{max}
- Caching threshold for edges c
- Number of selected neighboring communities s
- Threshold t on the number of neighboring communities, when to start selecting neighboring communities
- (P)runing
- (P)runing communities with (p)ositive connectivity
- Different tie-breaking rules (random, smallest ID, hybrid)

We evaluate each configuration for $k \in \{2, 8, 16, 64\}$, 5 different seeds and an imbalance parameter $\epsilon = 3\%$. When comparing different configurations, we first calculate the arithmetic mean over all seeds to summarize the results.

To tune the different parameters we run the experiments with $p = 16$ threads and the following configuration: ($l_{max} = 5$, $c = 16$, $s = 100$, $t = 100$, +P, +Pp, hybrid). Where a (+) denotes that the technique is enabled and a (-) that it is disabled. The abbreviations are explained in the listing above; see the letters in parentheses. We vary each parameter in a separate experiment, where the remaining parameters are set as in the baseline configuration.

Maximum Number of Iterations per Louvain Pass We set an upper limit l_{max} to the number of iterations per Louvain pass. The reasoning behind this is that iterating over all nodes of the fine hypergraph is very expensive. Therefore we want to contract the hypergraph as soon as possible to reduce the number of nodes. Our default value of 5 is very low in comparison to other Louvain implementations where this limit is usually between 16 and 32. This works for our specific application because, in later iterations, fewer nodes are moved, which only affects the community structure on a microscopic level. We are more interested in the macroscopic structure of the hypergraph. Therefore we prefer to merge too many nodes into a community rather than too few. If we merge too many nodes into a community, the coarsening algorithm can still decide not to contract them into a single node. On the other hand, since we restrict contraction to nodes of the same community, the contraction algorithm cannot correct the mistake if we merge too few nodes into a community. By allowing the algorithm to contract early, i.e. after five iterations, we promote node moves on the macroscopic level since a single move after a contraction corresponds to moving a whole set of nodes at a time. As we can see in Figure 7.1 the quality of the results is very similar for all l_{max} except for $l_{max} = 1$. We can also see on the right that increasing l_{max} to 16 does not improve the performance of the resulting partitions over $l_{max} = 5$. In Figure 7.2, we can see that the running time saved by reducing the number of passes is not proportional to the number of passes. This is because the first iteration takes the longest, and each subsequent iteration over all nodes is faster due to the decreased number of communities and improving moves.

Caching Threshold for Edges We presented a data structure in Section 5.2 to speed up the calculation of the connectivity. The overhead introduced by the data structure outweighs its benefits for smaller edges. We will refer to using the data structure for an

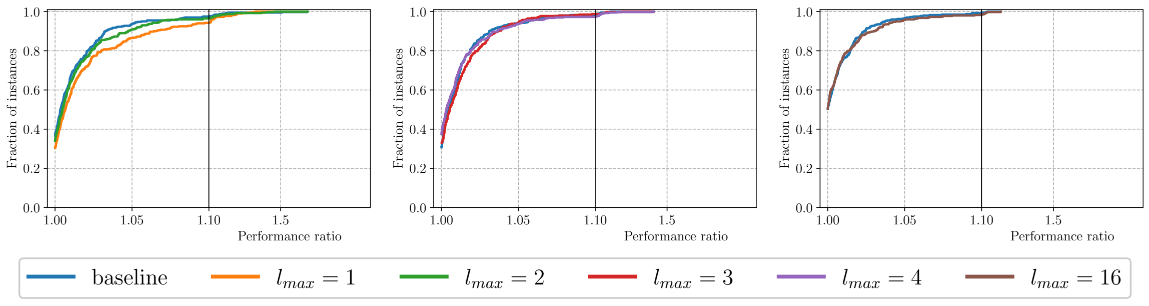

Figure 7.1.: Comparing the quality for $l_{max} = \{1, 2, 3, 4, 16\}$ and the baseline algorithm

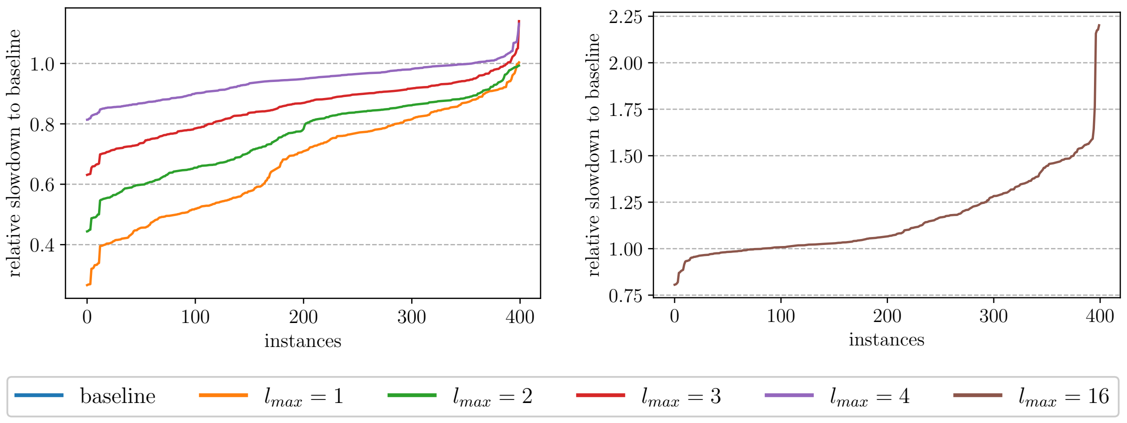

Figure 7.2.: Comparing the relative running times in comparison to the baseline configuration for $l_{max} = \{1, 2, 3, 4, 16\}$

edge as *caching* the edge in the following. Therefore we introduce a threshold c, which decides whether to cache the edge or calculate the connectivity naively. Figure 7.3 shows the results, in comparison to the baseline configuration, which uses $c = 16$. As we can see on the left, caching every edge is significantly slower because of the overhead the data structure introduces. We can also see that it is worth caching edges of size < 128 since not caching at all and only caching edges greater than 128 or 256 show a significant slowdown in comparison to the baseline configuration. On the right, we can observe that $c = 64$ is also too large for a threshold, and $c = 8$ is too low. Therefore we decide between 16 and 32. We can see that $c = 32$ and the baseline configuration are very close in the right plot, but the baseline configuration is faster for slightly more than half the instances. Therefore we conclude that the optimal value is closer to 16 than 32. Although the speedup by using the data structure is not as high as hoped, we expect it to be more efficient on larger hypergraph instances since most instances in the parameter tuning benchmark set contain relatively small hyperedges. One instance that demonstrates the use of the data structure is "opt1.mtx.hgr", which has a minimum edge size of 44 and a maximum edge size of 243. Therefore the baseline configuration caches all edges. When comparing the baseline configuration to not caching any edges at all, we observed a speedup of almost 50%.

Number of selected neighboring Communities. This parameter is based on the observation we made in Section 5.3, that the average distance between the ranking before and after the expected connectivity has been calculated is much lower than the actual number of neighboring communities $|\text{NC}(v)|$. Therefore we introduce this parameter to select the best s communities based on their connectivity for which we will choose the best community based on the fully calculated modularity delta. The results are illustrated in Figure 7.4. As we can see, selecting only 16 or 32 instances slightly impacts the solution

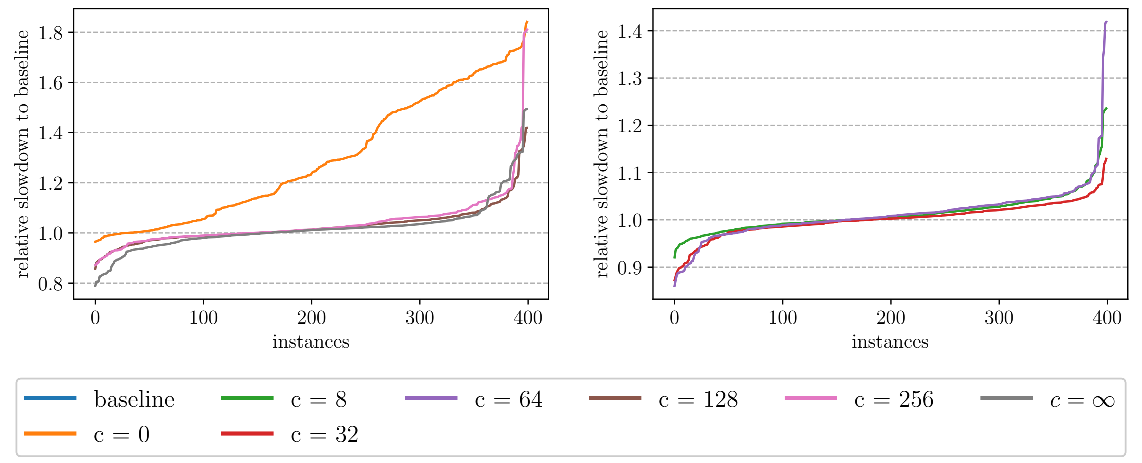

Figure 7.3.: Slowdown for caching thresholds $c = \{0, 8, 16, 32, 64, 128, 256, \infty\}$. Note that $c = 0$ corresponds to caching every edge and $c = \infty$ to not caching any edge.

quality. Since the quality for $s = 128$ is also slightly better than for $s = 64$ but slower we chose a compromise of $s = 100$. As we can see in the right graph, choosing $s = 100$ does not negatively impact the performance compared to calculating the whole modularity delta for all neighboring communities ($s = \infty$). Finally, as we can see in Figure 7.5 this yields a speedup of up to 4.5.

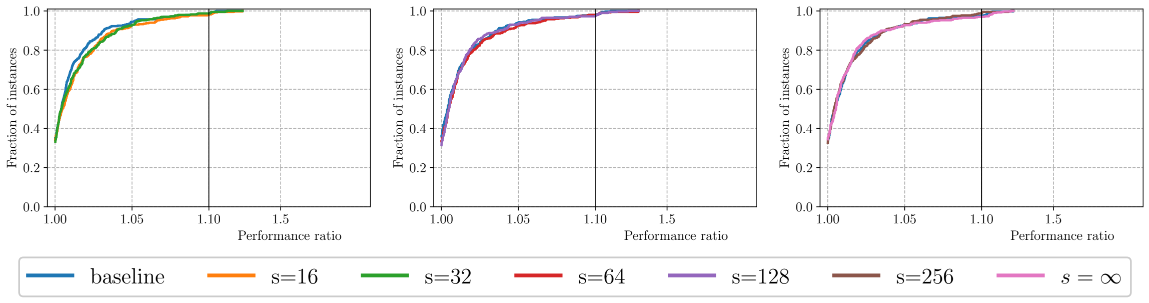

Figure 7.4.: Comparing the quality for $s = \{16, 32, 64, 128, 256, \infty\}$ and the baseline algorithm. Note that $s = \infty$ corresponds always considering all neighboring communities.

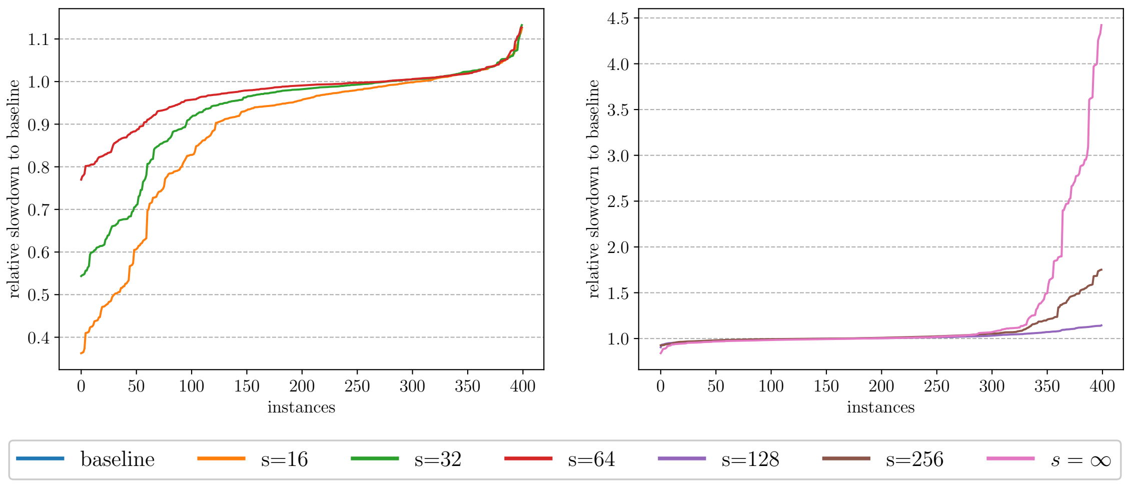

Figure 7.5.: Running times relative to the baseline configuration

When to start selecting neighboring Communities. Since we are choosing the 100 best neighboring communities based on their connectivity, we need to partition them. This introduces overhead, which we would like to prevent when it is not necessary, i.e., when there are less than 100 neighboring communities or only slightly more than 100. Figure 7.6 illustrates the results. As we can see, it is always worth partitioning the neighboring communities when there are at least 100 neighbors and selecting the best 100. For $t = 110, 120$ the slowdown on 50% of the instances is only slightly worse than the speed up for the other half. For all other t, it is slower. Therefore the overhead of sorting does not outweigh the time needed to compute the expected connectivity for additional communities.

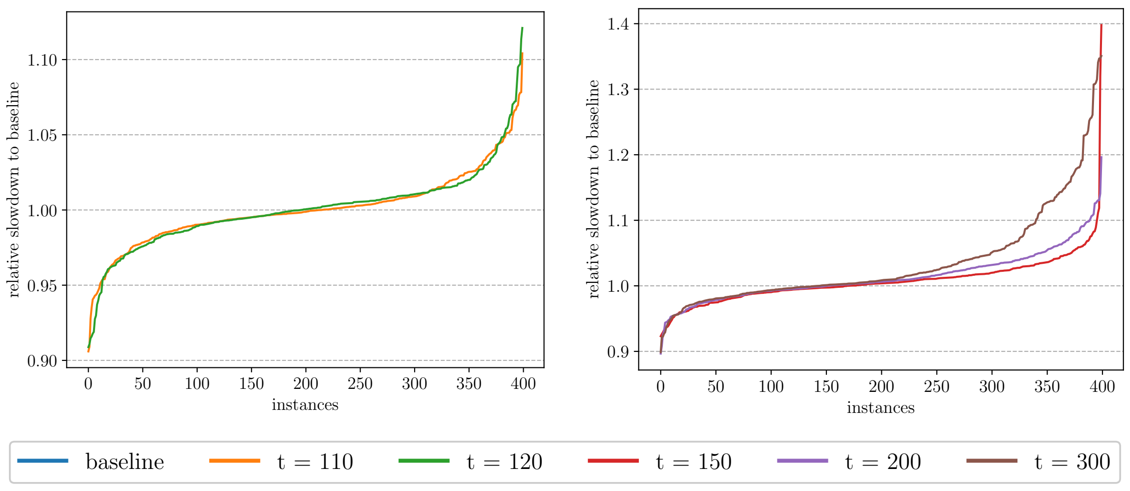

Figure 7.6.: Relative slowdown to the baseline configuration of the algorithm when only selecting 100 neighbors if there are at least $t = \{110, 120, 150, 200, 300\}$ neighbors to choose from.

Pruning We introduced three rules for pruning in Section 5.3. As we can see in Figure 7.7, implementing the three rules for pruning is faster on virtually all instances. Applying them yields a speedup of up to 6. This also illustrates the significant impact of the calculation of the probabilistic term on the overall running time.

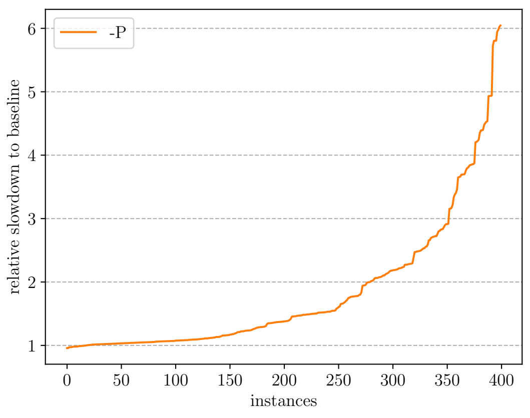

Figure 7.7.: Slowdown of the algorithm when turning off the pruning

Pruning positive connectivity. The main assumption is that the algorithm almost exclusively performs moves that reduce the connectivity, i.e., reduces the overall number of connections between communities. By sorting out all other moves, we can greatly reduce the number of communities for which we have to calculate the second part of the modularity delta. Figure 7.8 shows the results. As we can see, it does not have a negative effect on the quality while yielding a significant speedup of up to 2.25. This works in our specific case for the same reason as the limit on iterations per Louvain pass l_{max}. Since we generally prefer merging too many nodes into communities over too few, restricting node moves to only reduce the number of connections between communities promotes this behavior.

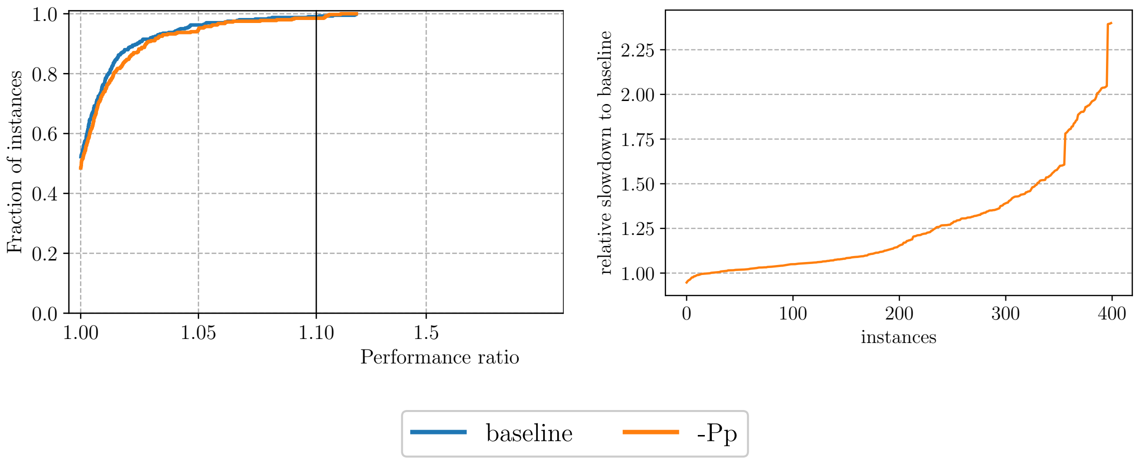

Figure 7.8.: Comparing the quality without pruning communities with positive connectivity(left) and the running time relative to the baseline configuration (right).

Tie-Breaking Rules While using random tie-breaking, we noticed that our algorithm produces substantially more communities than the algorithm based on the bipartite representation. This difference is especially noticeable on the dual-instances, which have significantly more nodes than edges. A possible reason for this is that initially, there are many neighboring communities for which the modularity delta is equal. By randomly choosing a neighbor, the algorithm produces many small but very similar communities. Instead, if there is a choice, we want to always move the node to the same communities and therefore reduce the number of communities. We do this by breaking ties by choosing the community with the smallest ID. Since we want to only apply this rule for dual - instances or similar ones, we decide which rule we apply based on the density of the initial hypergraph since the dual-instances stand out by having a low density < 1. The density of a hypergraph is the ratio of hypernodes to hyperedges $\frac{m}{n}$. We choose the threshold as 1, i.e., if there are fewer edges than nodes, we choose the community with the smaller ID. Otherwise, we apply random tie-breaking. As we can see in Figure 7.9 this slightly improves the quality overall, especially in comparison to the random tie-breaking rule. If we look at the different instance classes in Figure 7.10, we can see the improvement of the smaller ID tie-breaking rule over the random tie-breaking rule on the dual-instances. While the quality stays very close for most other instance classes, the hybrid tie-breaking performs the worst on the ISPD98 instances. This is due to these instances being close to 1 in density, which results in the hybrid rule to use random tie-breaking for some instances and the smaller ID one for others – though the difference in quality stays within 4%.

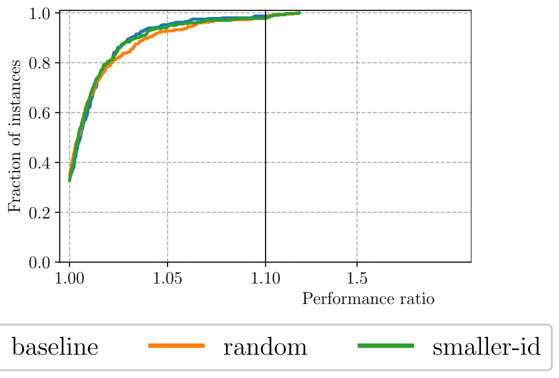

Figure 7.9.: Comparing the overall quality for different tie-breaking rules

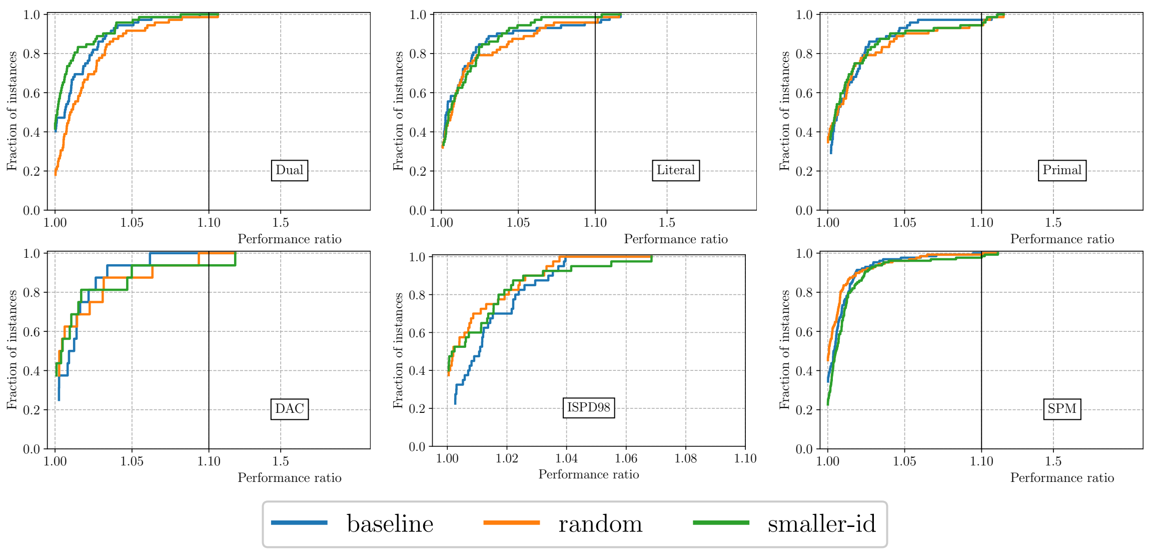

Figure 7.10.: Comparing the quality of different tie-breaking rules per instance class

7.3. Scalability

We ran the following scalability experiments with $p \in \{4, 16, 64\}$ threads on the benchmark set B. Due to time constraints, we had to exclude the five hypergraphs in Table D.1. We illustrate the speedups of preprocessing step in Figure 7.11. In the plot, points represent the speedup for a single instance and lines the cumulative harmonic mean speedup over all instances with a single-threaded running time $\geq x$. Our approach achieves a harmonic mean speedup of 3.4 for $p = 4$, 11.1 for $p = 16$ and 23.8 for $p = 64$. If we only consider instances with a running time $\geq 100s$, the harmonic mean speedup is 3.5 for $p = 4$, 12.5 for $p = 16$ and 32.6 for $p = 64$. For $p = 4$ we achieve a speedup of at least 3 for more than 84% of the instances.

As we can see in comparison to the graph-based approach (on the right), our algorithm exhibits a slightly better scaling behavior, especially for $p = 64$. The graph-based approach achieves a harmonic mean speedup of 3.3 for $p = 4$, 9.6 for $p = 16$ and 18.2 for $p = 64$. If we again, only consider instances with a running time over $\geq 100s$, the harmonic mean speedup is 2.9 for $p = 4$, 10.4 for $p = 16$ and 27.5 for $p = 64$. For $p = 4$ it achieves a speedup of at least 3 for only 75% of the instances. Although the reason for this might be the overall lower running time of `Mt-KaHyPar-D`, therefore sequential parts and the overhead of multiple threads take up a bigger fraction of the running time than in the case of `Mt-KaHyPar-HM`.

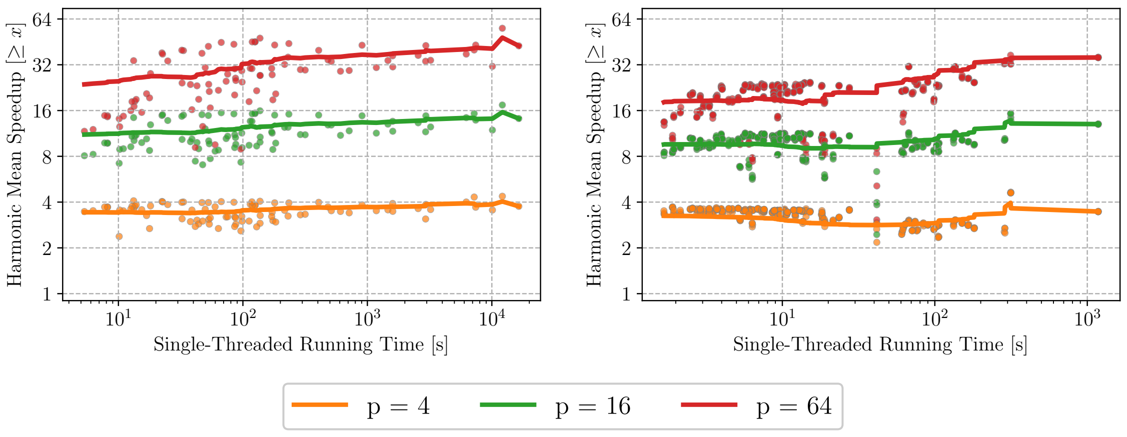

Figure 7.11.: Arithmetic mean speedup (points) and cumulative harmonic mean speedup (lines) of the preprocessing step of HM (left) and D (right) with $p \in \{4, 16, 64\}$ for each instance of set B.

7.4. Comparison to Community Detection via Bipartite Graph Representation

We compare our hypergraph modularity (HM) to the current approach based on the bipartite graph representation (D) on the whole benchmark set A. We use $k \in \{2, 4, 8, 16, 32, 64, 128\}$, an imbalance parameter of $\epsilon = 0.03$, ten different random seeds as well as $p = 10$ threads. Additionally, we compare both approaches on the whole benchmark set B. We use $k \in \{2, 8, 16, 64\}$, an imbalance parameter of $\epsilon = 0.03$, five different random seeds as well as $p = 64$ threads.

Figure 7.12 shows the results for benchmark set A. Our hypergraph modularity approach produces the best results for 61% of the instances. Looking at the different instance classes in Figure 7.13, we can see that our approach significantly improves the quality on the sparse matrix instances (bottom right). On these instances, we produce the best partitions on 77.6% of the instances. Additionally, we notice that for over 0.8% of the instances, we produce partitions better by a factor of 100 or more, which comes down to the hypergraphs "Trec14", "Chebyshev4" and "us04". Our approach also performs better on the literal SAT instances, producing the best partition for 58% of the instances. On the primal SAT instances, the two approaches perform equally. The graph-based approach performs better on the ISPD98 and the dual SAT instances, computing the best partition on 55.3% and 54.8% of the instances, respectively. Although we would like to mention that there is one of the dual instances where our approach performs better by a factor of over 326. The most significant difference between the two approaches is visible in the bottom left for the DAC VLSI instances. Here our approach is worse since the graph-based approach produces the best partition for 90% of the instances. We also append more detailed insight for the different k, as well as a comparison of the absolute running times in Appendix E.

As we can see in Figure 7.14, our approach is unfortunately significantly slower, especially on the dual SAT instances. Comparing the geometric mean running times in Appendix E, we can see that the preprocessing step using our hypergraph modularity is slower by a factor of 5.4. Overall, integrating our hypergraph based approach into the partitioner results in a slowdown by a factor of 1.5.

Figure 7.15 shows the overall results on benchmark set B. Both approaches compute the best quality partition on 50% of the hypergraphs, although our approach performs slightly worse overall. Looking at the different instance classes in Figure 7.16, we see that

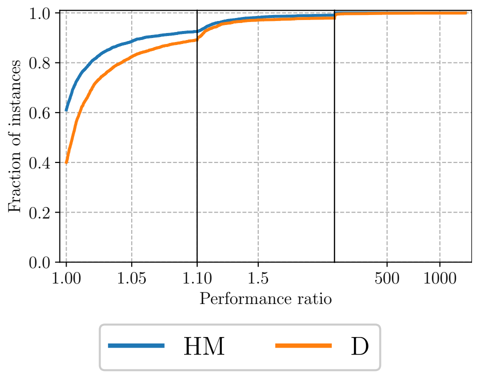

Figure 7.12.: Comparing the overall quality to the current graph based approach

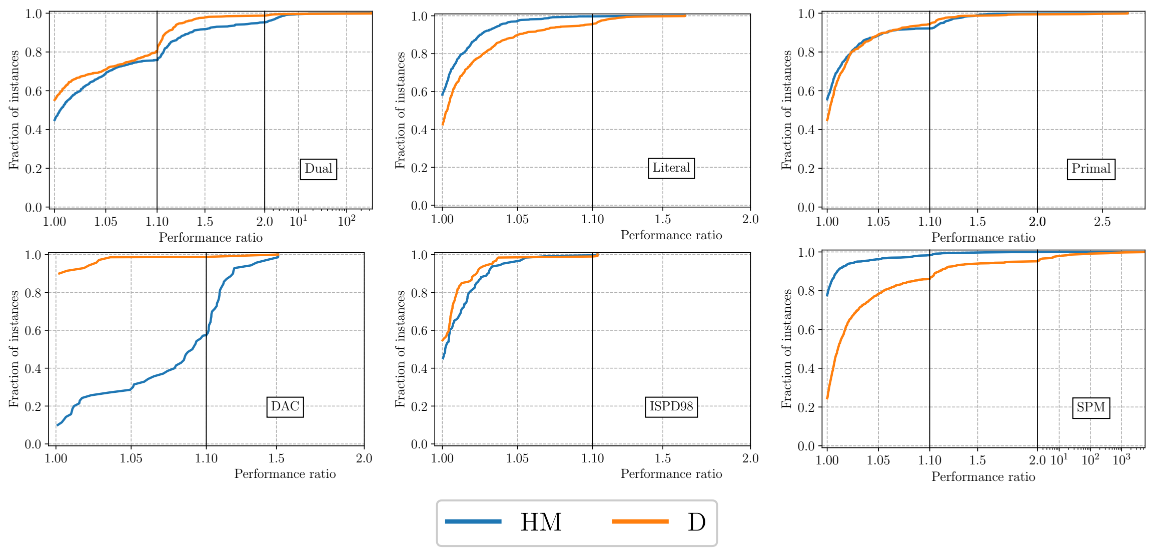

Figure 7.13.: Comparing the quality of the hypergraph modularity approach with the current graph based approach per instance class

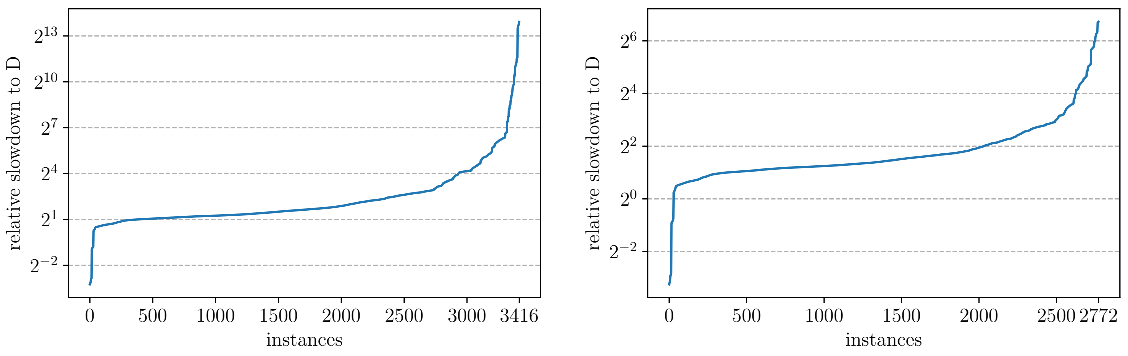

Figure 7.14.: Comparing the relative slowdown of the community detection step for both configurations. On the right dual SAT instances are excluded.

our approach slightly improves the results on the literal and primal SAT instances by computing the best quality partition on 53.6% and 58.9% of the instances, respectively. Additionally, `HM` produces the best quality partition on 62.5% of the SPM instances. On the dual SAT instances, we were not able to improve the quality of partitions. Here our approach produces the best quality partitions on only 35.8%. The DAC instances are the same as in set A. Therefore there is no change in comparison to set A. The geometric mean running times of the preprocessing step are 0.82s for the graph-based approach and 6.1s for our hypergraph-based approach (see Appendix E). Therefore our approach is slower by a factor of 7.5. This slows down the overall running time of the partitioner by a factor of 1.84. Further optimizations are needed to make the running time competitive.

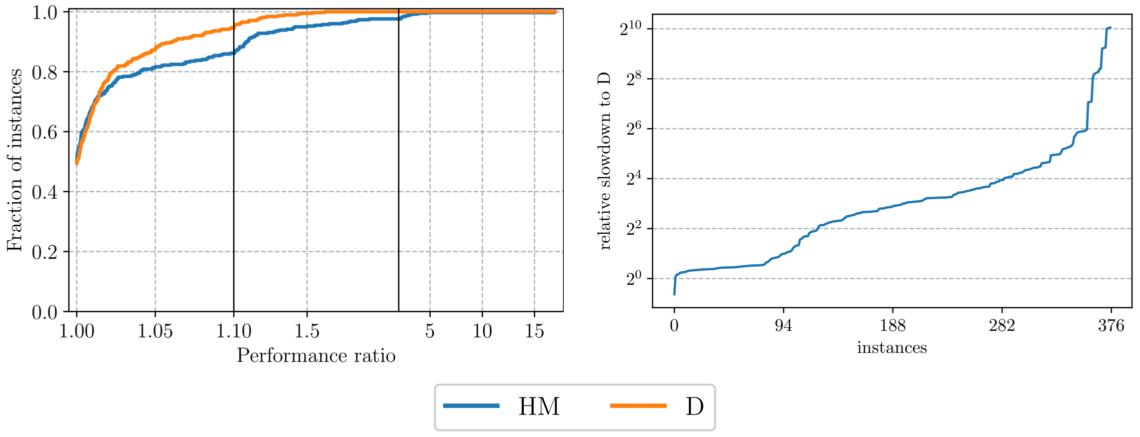

Figure 7.15.: Comparing the overall quality to the current graph based approach on set B (left) and the running time of the preprocessing step (right)

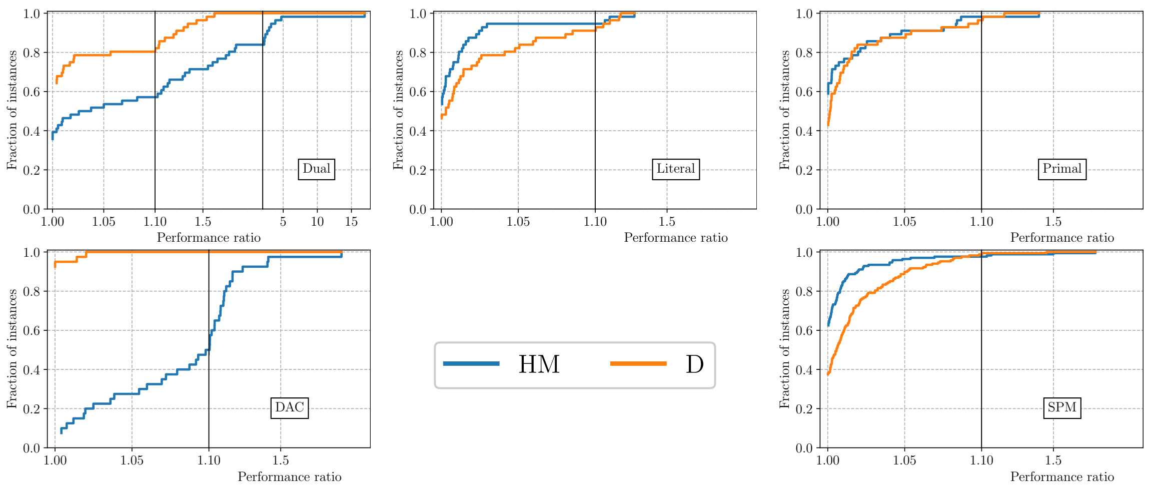

Figure 7.16.: Comparing the quality of the hypergraph modularity approach with the current graph based approach per instance class on set B

7.5. Comparing Hypergraph Map Equation and Modularity

We tried out two different random walk models for the map equation. The lazy random walk model (`HME`), which allows the random walker to stand still, and the model where the random walker cannot stand still on the same pin but can move to different pins of the same node (`HME-1`). We compare the quality of both approaches on the left in Figure 7.17. The right plot shows the quality in comparison to the modularity approach. As we can see, the less-lazy random walk model produces better results on 58% than its lazy counterpart. Therefore we use it to compare the map equation approach to the modularity approach on the right. The modularity approach produces the best result on 63% of the instances and therefore significantly outperforms our RelaxMap implementation as a preprocessing step to the coarsening phase.

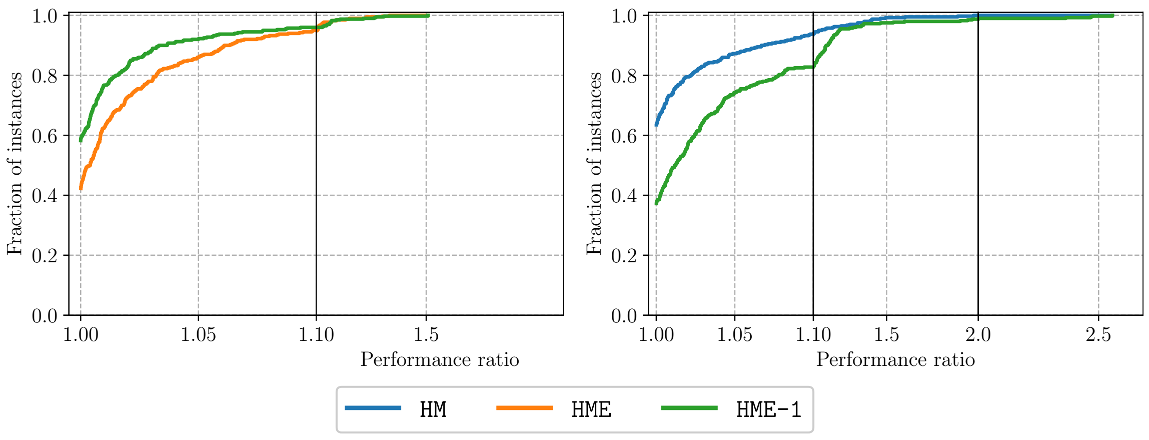

Figure 7.17.: Comparing the quality of the map equation approaches to the modularity approach

8. Conclusion

In this work, we derived a modularity definition directly on hypergraphs using the generalized Chung-Lu model [KPP$^+$19]. The modularity formulation based on the connectivity compares the number of communities connected by hyperedges to the expected number of communities connected by hyperedges if edges were randomly rewired. To optimize this modularity function, we implemented the parallel Louvain method (PLM) [SM16] and introduced many optimizations to reduce the running time while maintaining solution quality. We presented the community count data structure to determine connected communities of edges more efficiently for large hyperedges. Additionally, we presented three pruning rules, which allow us to skip the calculation of the probabilistic term solely based on the connectivity and volume of a community. We introduced multiple heuristics, such as ignoring moves that increase the number of connected communities and only considering the 100 best neighboring communities of a node based on their change in connectivity, which we validated in extensive experiments on a large real-world benchmark set.
In addition to the modularity approach, we derived formulations for the map equation on hypergraphs based on lazy random walks and less-lazy random walks. We optimize these approaches using our implementation of the RelaxMap algorithm [BHW$^+$17].
All approaches are integrated into the parallel shared-memory partitioner `Mt-KaHyPar` and used as a preprocessing step to the community aware coarsening algorithm. Our experiments showed that our modularity approach is significantly better suited for this application than the map equation approaches. It produces better results on 63% of the benchmark instances.
The new configuration `HM` using hypergraph modularity produces significantly better results on the sparse matrix instances and the literal SAT instances, producing the best partition in 77.6% and 58% of the instances, respectively. Since the sparse matrix instances have many large hyperedges, it shows that our approach, incorporating the higher-order structure of the hypergraph, can significantly improve the solution quality. On the other hand, VLSI, primal- and literal- SAT instances are often similar to graphs, and therefore we expect less improvement on hypergraphs from these application domains. Our approach performs significantly worse on instances from the DAC2012 Routability-Driven Placement Contest, where it only produces the best partition for 10% of the instances. Although it produces the best partition for 61% of the instances overall. However, the preprocessing step using hypergraph modularity is slower by a factor of 5.4 on average.

8.1. Future Work

While our hypergraph modularity approach was able to improve the quality of partitions for the SPM and literal SAT instances, it computes significantly worse results for the DAC instances. Hypergraphs from the DAC2012 Routability-Driven Placement Contest are similar to graphs except for a small subset of large hyperedges. It is unclear why our hypergraph approach performs significantly worse on them. Further experiments are needed to examine these differences in quality. However, our algorithm still is significantly slower due to the more complex calculation of the probabilistic term. Therefore further optimizations are needed to make our approach feasible. First of all, we are calculating the change in connectivity for all neighboring communities of a node and then only select 100 of them to evaluate the whole modularity delta. The running time could be significantly improved by not calculating this change for each neighboring community. Additionally, as we already mentioned in 5.2 instead of using our community count data structure, we could store each adjacent community for every node in later iterations. The required $\mathcal{O}(n^2)$ memory is feasible due to the hypergraph being sufficiently small.

Although we could significantly improve the running time of the delta calculation for the probabilistic term, it is still roughly as slow as the delta calculation for the connectivity. One approach that could improve the running time is to calculate a lower bound on $\Delta_{\mathbb{E}}$, which can then be used as a pruning rule to skip the actual calculation of the delta. We, for example, tried to approximate it using the geometric series, which unfortunately was not tight enough to be useful. This approach would also have the advantage of eliminating some of the inaccuracy introduced by the floating-point operations.

On the other hand, there are still plenty of different modularity approaches to explore: One possible addition to the null model by Kaminski et al. [KPP+19] could be to make sure only nodes which are in an edge of size d in the hypergraph $H = (V, E, \omega)$ are considered when randomly choosing nodes to add to an edge of size d in H'. This can be modeled by using $d_{vol}(u)$ instead of $vol(u)$, where $d_{vol(u)} = \sum_{e \in I(u) || e| = d} \omega(e)$ and $d_{vol}(V) = \sum_{u \in V} d_{vol}(u)$. This results in a model even closer to the original hypergraph since it preserves even more information about the structure of H.
As we have already seen, there are plenty of definitions for modularity on hypergraphs, which could be integrated into `Mt-KaHyPar` and evaluated as a preprocessing step to the coarsening phase. We also mentioned a formulation for the map equation in Section 6, which uses non-lazy random walks. This is also an approach worth exploring in the future.

Although we introduced methods for community detection, we did not evaluate them as such since our primary goal was to improve the preprocessing step of the partitioner. Additionally, current approaches to synthetic benchmark hypergraphs for community detection are limited to small edges and k-uniform hypergraphs. There remains a need for synthetic benchmark instances for community detection on hypergraphs, which resemble arbitrary real-world hypergraphs.

Bibliography

[ACU08] Cevdet Aykanat, B. Barla Cambazoglu, and Bora Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, May 2008.

[AHSS17] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k -way Hypergraph Partitioning Algorithm. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 28–42. Society for Industrial and Applied Mathematics, January 2017.

[AJZM$^+$05] 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, San Diego, CA, USA, 2005. IEEE.

[Alp98] Charles J. Alpert. The ISPD98 circuit benchmark suite. In *Proceedings of the 1998 international symposium on Physical design - ISPD '98*, pages 80–85, Monterey, California, United States, 1998. ACM Press.

[ASS18] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-Quality Shared-Memory Graph Partitioning. *arXiv:1710.08231 [cs]*, October 2018. arXiv: 1710.08231.

[BDHJ] A Belov, D Diepold, M Heule, and M Järvisalo. The SAT Competition 2014.

[BGL16] Austin R. Benson, David F. Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, July 2016. arXiv: 1612.08447.

[BGLL08] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008. arXiv: 0803.0476.

[BHW$^+$17] Seung-Hee Bae, Daniel Halperin, Jevin D. West, Martin Rosvall, and Bill Howe. Scalable and Efficient Flow-Based Community Detection for Large-Scale Graph Analysis. *ACM Transactions on Knowledge Discovery from Data*, 11(3):1–30, April 2017.

[BT93] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, March 1993.

[CA99] U.V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999.

[CCQY20] Jingya Chang, Yannan Chen, Liqun Qi, and Hong Yan. Hypergraph Clustering Using a New Laplacian Tensor with Applications in Image Processing. *SIAM Journal on Imaging Sciences*, 13(3):1157–1178, January 2020.

[CL04] Fan Chung and Linyuan Lu. Complex graphs and networks. *American Mathematical Society, Providence*, January 2004.

[CNM04] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, December 2004. arXiv: cond-mat/0408187.

[CR19] Uthsav Chitra and Benjamin J. Raphael. Random Walks on Hypergraphs with Edge-Dependent Vertex Weights. *arXiv:1905.08287 [cs, stat]*, May 2019. arXiv: 1905.08287.

[CVB21] Philip S. Chodrow, Nate Veldt, and Austin R. Benson. Generative hypergraph clustering: from blockmodels to modularity. *arXiv:2101.09611 [physics, stat]*, January 2021. arXiv: 2101.09611.

[DH11] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, November 2011.

[DK04] V. Durairaj and P. Kalla. Guiding CNF-SAT search via efficient constraint partitioning. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 498–501, San Jose, CA, USA, 2004. IEEE.

[DKC13] Mehmet Deveci, Kamer Kaya, and Umit V. Catalyurek. Hypergraph Sparsification and Its Application to Partitioning. In *2013 42nd International Conference on Parallel Processing*, pages 200–209, Lyon, France, October 2013. IEEE.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, January 2002.

[EBR17] Daniel Edler, Ludvig Bohlin, and Martin Rosvall. Mapping higher-order network flows in memory and multilayer networks with Infomap. *arXiv:1706.04792 [physics, stat]*, October 2017. arXiv: 1706.04792.

[EERR21] Anton Eriksson, Daniel Edler, Alexis Rojas, and Martin Rosvall. Mapping flows on hypergraphs. *arXiv:2101.00656 [physics]*, January 2021. arXiv: 2101.00656.

[FB07] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, January 2007. arXiv: physics/0607100.

[Fel19] Andreas Emil Feldmann. Fast Balanced Partitioning is Hard, Even on Grids and Trees. *arXiv:1111.6745 [cs]*, April 2019. arXiv: 1111.6745.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181, Las Vegas, NV, USA, 1982. IEEE.

[For10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, February 2010. arXiv: 0906.0612.

[GHSS20] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable Shared-Memory Hypergraph Partitioning. *arXiv:2010.10272 [cs]*, October 2020. arXiv: 2010.10272.

[Gö10] Robert Görke. *An Algorithmic Walk from Static to Dynamic Graph Clustering.* PhD Thesis, Karlsruher Institut für Technologie (KIT), 2010.

[HS17] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 1868-8969.

[HSS19] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics*, 24:1–36, December 2019.

[HSWZ18] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. Distributed Graph Clustering using Modularity and Map Equation. *arXiv:1710.09605 [physics]*, 11014:688–702, 2018. arXiv: 1710.09605.

[IWW93] Edmund Ihler, Dorothea Wagner, and Frank Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45(4):171–175, March 1993.

[KAKS99] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, March 1999.

[KPP+19] Bogumił Kamiński, Valérie Poulin, Paweł Prałat, Przemysław Szufel, and François Théberge. Clustering via hypergraph modularity. *PLOS ONE*, 14(11):e0224307, November 2019.

[KPT21] Bogumi\l Kamiński, Pawe\l Pra\lat, and François Théberge. Community Detection Algorithm Using Hypergraph Modularity. In Rosa M. Benito, Chantal Cherifi, Hocine Cherifi, Esteban Moro, Luis Mateus Rocha, and Marta Sales-Pardo, editors, *Complex Networks & Their Applications IX*, pages 152–163, Cham, 2021. Springer International Publishing.

[KR15] Tatsuro Kawamoto and Martin Rosvall. Estimating the resolution limit of the map equation in community detection. *Physical Review E*, 91(1):012809, January 2015.

[KSX20] Zheng Tracy Ke, Feng Shi, and Dong Xia. Community Detection for Hypergraph Networks via Regularized Tensor Power Iteration. *arXiv:1909.06503 [math, stat]*, January 2020. arXiv: 1909.06503.

[KVA+18] Tarun Kumar, Sankaran Vaidyanathan, Harini Ananthapadmanabhan, Srinivasan Parthasarathy, and Balaraman Ravindran. Hypergraph Clustering: A Modularity Maximization Approach. *arXiv:1812.10869 [cs, stat]*, December 2018. arXiv: 1812.10869.

[KVA+20] Tarun Kumar, Sankaran Vaidyanathan, Harini Ananthapadmanabhan, Srinivasan Parthasarathy, and Balaraman Ravindran. A New Measure of Modularity in Hypergraphs: Theoretical Insights and Implications for Effective Clustering. In Hocine Cherifi, Sabrina Gaito, José Fernendo Mendes, Esteban Moro, and Luis Mateus Rocha, editors, *Complex Networks and Their Applications VIII*, volume 881, pages 286–297. Springer International Publishing, Cham, 2020. Series Title: Studies in Computational Intelligence.

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Vieweg+Teubner Verlag, Wiesbaden, 1990.

[LM17] Pan Li and Olgica Milenkovic. Inhomogeneous Hypergraph Clustering with Applications. *arXiv:1709.01249 [cs, stat]*, November 2017. arXiv: 1709.01249.

[LM18] Pan Li and Olgica Milenkovic. Submodular Hypergraphs: p-Laplacians, Cheeger Inequalities and Spectral Clustering. *arXiv:1803.03833 [cs]*, October 2018. arXiv: 1803.03833.

[MP14] Zoltan Mann and Pal Papp. Formula partitioning revisited. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 41–56. EasyChair, 2014. ISSN: 2398-7340.

[MSS15] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel Graph Partitioning for Complex Networks. *arXiv:1404.4797 [physics]*, January 2015. arXiv: 1404.4797.

[New04] M. E. J. Newman. Analysis of weighted networks. *Physical Review E*, 70(5):056131, November 2004. arXiv: cond-mat/0407503.

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, February 2004. arXiv: cond-mat/0308217.

[Phe08] Chuck Pheatt. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298, April 2008. Place: Evansville, IN, USA Publisher: Consortium for Computing Sciences in Colleges.

[PM07] David Papa and Igor Markov. Hypergraph Partitioning and Clustering. In Teofilo Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*, volume 20073547, pages 61–1–61–19. Chapman and Hall/CRC, May 2007. Series Title: Chapman & Hall/CRC Computer & Information Science Series.

[RAB09] M. Rosvall, D. Axelsson, and C. T. Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, November 2009.

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[Sch20] Sebastian Schlag. *High-Quality Hypergraph Partitioning.* PhD Thesis, Karlsruher Institut für Technologie (KIT), 2020.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, July 1948.

[SHH$^+$15] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. *arXiv:1511.03137 [cs]*, November 2015. arXiv: 1511.03137.

[SM16] Christian L. Staudt and Henning Meyerhenke. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):171–184, January 2016.

[VAS$^+$12] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 routability-driven placement contest and benchmark suite. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 774, San Francisco, California, 2012. ACM Press.

[VB05] Brendan Vastenhouw and Rob H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, January 2005.

Appendix

A. Derivation of the Expected Connectivity

We calculate the expected connectivity as follows:

$$\begin{aligned}
\mathbb{E}_{H'\sim \mathrm{H}}\left[e_{H'}(C)\right] &= \sum_{d\in\mathcal{D}} \sum_{e\in F_d(C)} P_{\mathrm{H}}(e) \\
&= \sum_{d\in\mathcal{D}} W_d \sum_{e\in F_d(C)} s_{\mathrm{H}}(e) \\
&= \sum_{d\in\mathcal{D}} W_d \cdot \mathbb{P}\left(\sum_{i:v_i\in C} X_i^{(d)} \geq 1\right) \\
&= \sum_{d\in\mathcal{D}} W_d \left(1 - \mathbb{P}\left(\sum_{i:v_i\in C} X_i^{(d)} = 0\right)\right) \\
&= \sum_{d\in\mathcal{D}} W_d \left(1 - \left(1 - \frac{\mathrm{vol}(C)}{\mathrm{vol}(V)}\right)^d\right)
\end{aligned} \tag{8.1}$$

B. Calculations for Hypergraph Map Equation

Derivation of Exit Probabilities Given the transition probabilities

$$P_{ij} = \sum_{e\in I(C_i)} \frac{\Phi(e,C_i)\omega(e)}{\mathrm{vol}(C_i)} \cdot \frac{\Phi(e,C_j)}{|e|}$$

and the stationary distribution

$$p_v = \frac{\mathrm{vol}(v)}{\mathrm{vol}(V)}$$

We derive the exit probability $q_{i\curvearrowright}$ from Equation (6.4), which is the probability for the random walker to be in community C_i and moving to any other community C_j in the following step:

$$
\begin{aligned}
q_{i\curvearrowright} &= \left(\sum_{u\in C_i} p_u\right)\cdot \sum_{C_j\in C} P_{ij} \\
&= \frac{\mathrm{vol}(C_i)}{\mathrm{vol}(V)} \sum_{C_j\in C}\sum_{e\in I(C_i)} \frac{\Phi(e,C_i)\omega(e)}{\mathrm{vol}(C_i)}\cdot\frac{\Phi(e,C_j)}{|e|} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{C_j\in C}\sum_{e\in I(C_i)} \Phi(e,C_i)\cdot\omega(e)\cdot\frac{\Phi(e,C_j)}{|e|} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(C_i)}\sum_{C_j\in C} \Phi(e,C_i)\cdot\omega(e)\cdot\frac{\Phi(e,C_j)}{|e|} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(C_i)} \frac{\Phi(e,C_i)\cdot\omega(e)}{|e|}\sum_{C_j\in C}\Phi(e,C_j) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(C_i)} \frac{\Phi(e,C_i)\cdot\omega(e)}{|e|}\cdot(|e|-\Phi(e,C_i))
\end{aligned}
\tag{8.2}
$$

Derivation of Change in Exit Probabilities To calculate the change in exit probability for a community C_i, we first subtract the previous contributions of the incident edges of v and then add the contributions of these edges if v was removed from C_i.

$$
\begin{aligned}
\Delta q_{i\curvearrowright} &= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} -\overbrace{\Phi(e,C_i)\omega(e)\frac{|e|-\Phi(e,C_i)}{|e|}}^{\text{old contribution}} + \overbrace{\Phi(e,C_i\setminus v)\omega(e)\frac{|e|-\Phi(e,C_i\setminus v)}{|e|}}^{\text{new contribution}} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} \omega(e) \\
&\cdot\left(-\Phi(e,C_i)\frac{|e|-\Phi(e,C_i)}{|e|} + (\Phi(e,C_i)-\Phi(e,v))\cdot\frac{|e|-(\Phi(e,C_i)-\Phi(e,v))}{|e|}\right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} \omega(e)\cdot\left(-\Phi(e,C_j)\frac{|e|-\Phi(e,C_j)}{|e|} + \Phi(e,C_j)\frac{|e|-\Phi(e,C_j)}{|e|}\right. \\
&\left. + \frac{\Phi(e,C_j)\Phi(e,v)}{|e|} - \Phi(e,v)\frac{|e|-\Phi(e,C_j)+\Phi(e,v)}{|e|}\right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} \omega(e)\cdot\left(\frac{\Phi(e,C_i)\Phi(e,v)}{|e|} - \Phi(e,v)\frac{|e|-\Phi(e,C_i)+\Phi(e,v)}{|e|}\right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e\in I(v)} \omega(e)\Phi(e,v)\frac{2\Phi(e,C_i)-\Phi(e,v)-|e|}{|e|} \\
&= \frac{1}{\mathrm{vol}(V)} \left(\left(\sum_{e\in I(v)} \omega(e)\Phi(e,v)\frac{2\Phi(e,C_i)-\Phi(e,v)}{|e|}\right) - \mathrm{vol}(v)\right)
\end{aligned}
\tag{8.3}
$$

For the destination community C_j, this results in:

$$
\begin{aligned}
\Delta q_{j\curvearrowright} &= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \overbrace{- \Phi(e, C_j)\omega(e)\frac{|e| - \Phi(e, C_j)}{|e|}}^{\text{old contribution}} \overbrace{+ \Phi(e, C_j \cup v)\omega(e)\frac{|e| - \Phi(e, C_j \cup v)}{|e|}}^{\text{new contribution}} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \\
&\cdot \left(-\Phi(e, C_j)\frac{|e| - \Phi(e, C_j)}{|e|} + (\Phi(e, C_j) + \Phi(e, v)) \cdot \frac{|e| - (\Phi(e, C_j) + \Phi(e, v))}{|e|}\right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \cdot \left(-\frac{\Phi(e, C_j)\Phi(e, v)}{|e|} + \Phi(e, v)\frac{|e| - \Phi(e, C_j) - \Phi(e, v)}{|e|}\right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e)\Phi(e, v)\frac{-2\Phi(e, C_j) - \Phi(e, v) + |e|}{|e|} \\
&= \frac{1}{\mathrm{vol}(V)} \left(\left(\sum_{e \in I(v)} \omega(e)\Phi(e, v)\frac{-2\Phi(e, C_j) - \Phi(e, v)}{|e|}\right) + \mathrm{vol}(v)\right)
\end{aligned}
\tag{8.4}
$$

C. Calculations for the less-lazy Hypergraph Map Equation

We will take a look at a *less-lazy* random walk, where the random walker must not stand still on the same pin but can move to another pin of the same node. In this case the probability to choose a node is $\frac{\Phi(e,v)}{|e|-1}$.

Therefore the transition probability between nodes becomes:

$$P_{uv} = \sum_{e \in I(u)} \frac{\Phi(e, u)\omega(e)}{\mathrm{vol}(u)} \cdot \frac{\Phi(e, v)}{|e| - 1}$$

With this in mind, we can also formulate the transition probabilities between communities by replacing $|e|$ with $|e| - 1$ in the formulation for the lazy random walk:

$$P_{ij} = \sum_{e \in I(C_i)} \frac{\Phi(e, C_i)\omega(e)}{\mathrm{vol}(C_i)} \cdot \frac{\Phi(e, C_j)}{|e| - 1}$$

The stationary distribution stays the same as for the lazy-random walk:

Lemma 8.1. *For every node $v \in V$ the stationary distribution p of M is given by*

$$p_v = \frac{\mathrm{vol}(v)}{\mathrm{vol}(V)}$$

Proof.

$$
\begin{aligned}
(pP)_v &= \sum_{u \in V} p_u P_{uv} \\
&= \sum_{\substack{u \in V \\ u \neq v}} \frac{\mathrm{vol}(u)}{\mathrm{vol}(V)} \sum_{e \in I(u)} \frac{\Phi(e,u)\omega(e)}{\mathrm{vol}(u)} \cdot \frac{\Phi(e,v)}{|e|-1} + \frac{\mathrm{vol}(v)}{\mathrm{vol}(V)} \sum_{e \in I(v)} \frac{\Phi(e,v)\omega(e)}{\mathrm{vol}(v)} \cdot \frac{\Phi(e,v)-1}{|e|-1} \\
&= \frac{1}{\mathrm{vol}(V)} \left(\sum_{e \in I(v)} \frac{\Phi(e,v)\omega(e)}{|e|-1} \left(\Phi(e,v) - 1 + \sum_{\substack{u \in V \\ u \neq v}} \Phi(e,u) \right) \right) \\
&= \frac{1}{\mathrm{vol}(V)} \left(\sum_{e \in I(v)} \frac{\Phi(e,v)\omega(e)}{|e|-1} (|e|-1) \right) \\
&= \frac{\mathrm{vol}(v)}{\mathrm{vol}(V)}
\end{aligned}
\tag{8.5}
$$

From line 2 to 3, we use the observation that we can sum over the incident edges of v instead of u, since if either of the two nodes is not contained in the edge, the summand is zero. Therefore, summing over the incident edges of u or v is equivalent. □

The exit probability also only changes in a minor way, in comparison to the lazy random walk version. We can again just exchange $|e|$ for $|e| - 1$:

$$
\begin{aligned}
q_{i\curvearrowright} &= \left(\sum_{u \in C_i} p_u \right) \cdot \sum_{C_j \in C} P_{ij} \\
&= \frac{\mathrm{vol}(C_i)}{\mathrm{vol}(V)} \sum_{C_j \in C} \sum_{e \in I(C_i)} \frac{\Phi(e,C_i)\omega(e)}{\mathrm{vol}(C_i)} \cdot \frac{\Phi(e,C_j)}{|e|-1} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{C_j \in C} \sum_{e \in I(C_i)} \Phi(e,C_i) \cdot \omega(e) \cdot \frac{\Phi(e,C_j)}{|e|-1} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(C_i)} \sum_{C_j \in C} \Phi(e,C_i) \cdot \omega(e) \cdot \frac{\Phi(e,C_j)}{|e|-1} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(C_i)} \frac{\Phi(e,C_i) \cdot \omega(e)}{|e|-1} \sum_{C_j \in C} \Phi(e,C_j) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(C_i)} \frac{\Phi(e,C_i) \cdot \omega(e)}{|e|-1} \cdot (|e| - \Phi(e,C_i))
\end{aligned}
\tag{8.6}
$$

To calculate the change in exit probability for a community C_i, we first subtract the previous contributions of the incident edges of v and then add the contributions of these edges if v was removed from C_i.

$$\begin{aligned}
\Delta q_{i\curvearrowright} &= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \overbrace{- \Phi(e,C_i)\omega(e)\frac{|e| - \Phi(e,C_i)}{|e|-1}}^{\text{old contribution}} + \overbrace{\Phi(e, C_i \setminus v)\omega(e)\frac{|e| - \Phi(e, C_i \setminus v)}{|e|-1}}^{\text{new contribution}} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \\
&\cdot \left(-\Phi(e,C_i)\frac{|e| - \Phi(e,C_i)}{|e|-1} + (\Phi(e,C_i) - \Phi(e,v)) \cdot \frac{|e| - (\Phi(e,C_i) - \Phi(e,v))}{|e|-1} \right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \cdot \left(-\Phi(e,C_j)\frac{|e| - \Phi(e,C_j)}{|e|-1} + \Phi(e,C_j)\frac{|e| - \Phi(e,C_j)}{|e|-1} \right. \\
&\left. + \frac{\Phi(e,C_j)\Phi(e,v)}{|e|-1} - \Phi(e,v)\frac{|e| - \Phi(e,C_j) + \Phi(e,v)}{|e|-1} \right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \cdot \left(\frac{\Phi(e,C_i)\Phi(e,v)}{|e|-1} - \Phi(e,v)\frac{|e| - \Phi(e,C_i) + \Phi(e,v)}{|e|-1} \right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e)\Phi(e,v)\frac{2\Phi(e,C_i) - \Phi(e,v) - |e|}{|e|-1}
\end{aligned} \tag{8.7}$$

For the destination community C_j, this results in:

$$\begin{aligned}
\Delta q_{j\curvearrowright} &= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \overbrace{- \Phi(e,C_j)\omega(e)\frac{|e| - \Phi(e,C_j)}{|e|-1}}^{\text{old contribution}} + \overbrace{\Phi(e, C_j \cup v)\omega(e)\frac{|e| - \Phi(e, C_j \cup v)}{|e|-1}}^{\text{new contribution}} \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \\
&\cdot \left(-\Phi(e,C_j)\frac{|e| - \Phi(e,C_j)}{|e|-1} + (\Phi(e,C_j) + \Phi(e,v)) \cdot \frac{|e| - (\Phi(e,C_j) + \Phi(e,v))}{|e|-1} \right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e) \cdot \left(-\frac{\Phi(e,C_j)\Phi(e,v)}{|e|-1} + \Phi(e,v)\frac{|e| - \Phi(e,C_j) - \Phi(e,v)}{|e|-1} \right) \\
&= \frac{1}{\mathrm{vol}(V)} \sum_{e \in I(v)} \omega(e)\Phi(e,v)\frac{-2\Phi(e,C_j) - \Phi(e,v) + |e|}{|e|-1}
\end{aligned} \tag{8.8}$$

D. Excluded hypergraphs from set B

Due to time constrains, we had to exclude the slowest five hypergraphs out of the 94 total hypergraphs for the scaling experiments, which we listed in Table D.1.

Table D.1.: Excluded hypergraphs from set B

Class	Hypergraph
SAT14 - Dual	sv-comp19_prop-reachsafety.barrier_3t_true-unreach-call.i-witness.dual
	zfcp-2.8-u2-nh.dual
	velev-npe-1.0-9dlx-b71.dual
SPM	it-2004
	sk-2005

E. Detailed Comparison to Mt-KaHyPar-D

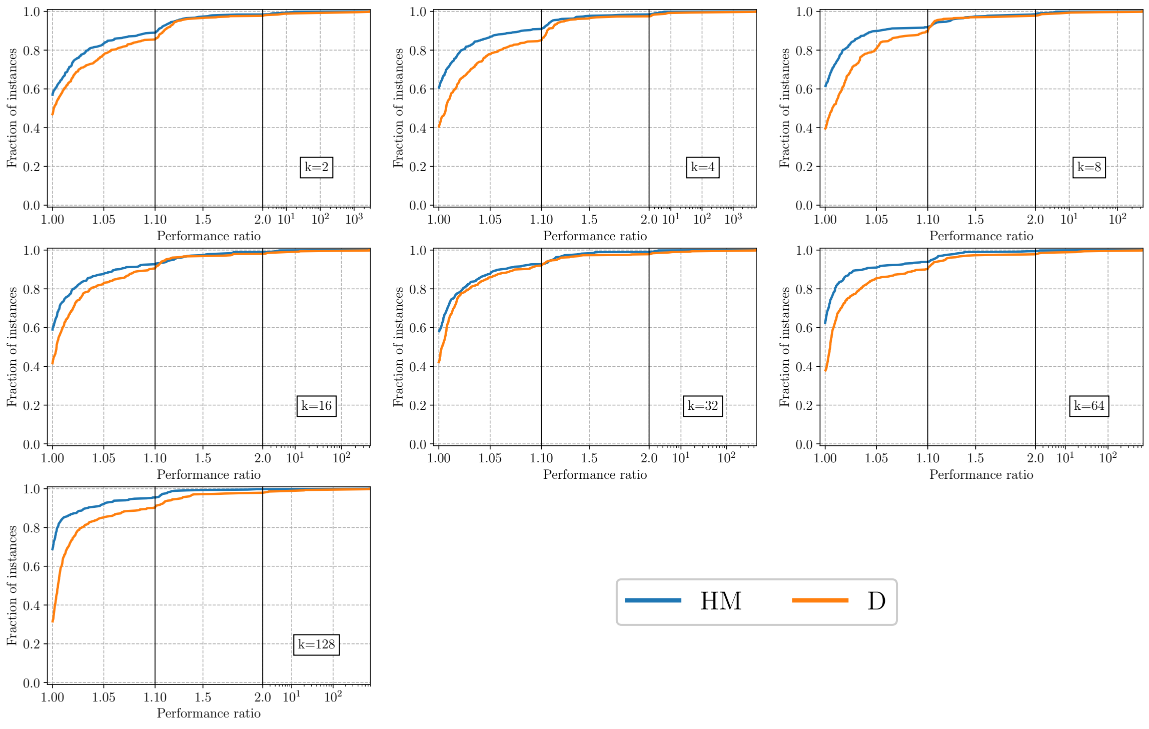

Figure E.1.: Comparing the quality of the hypergraph modularity approach with the current graph based approach for $k = \{2, 4, 8, 16, 32, 64, 128\}$ on set A.

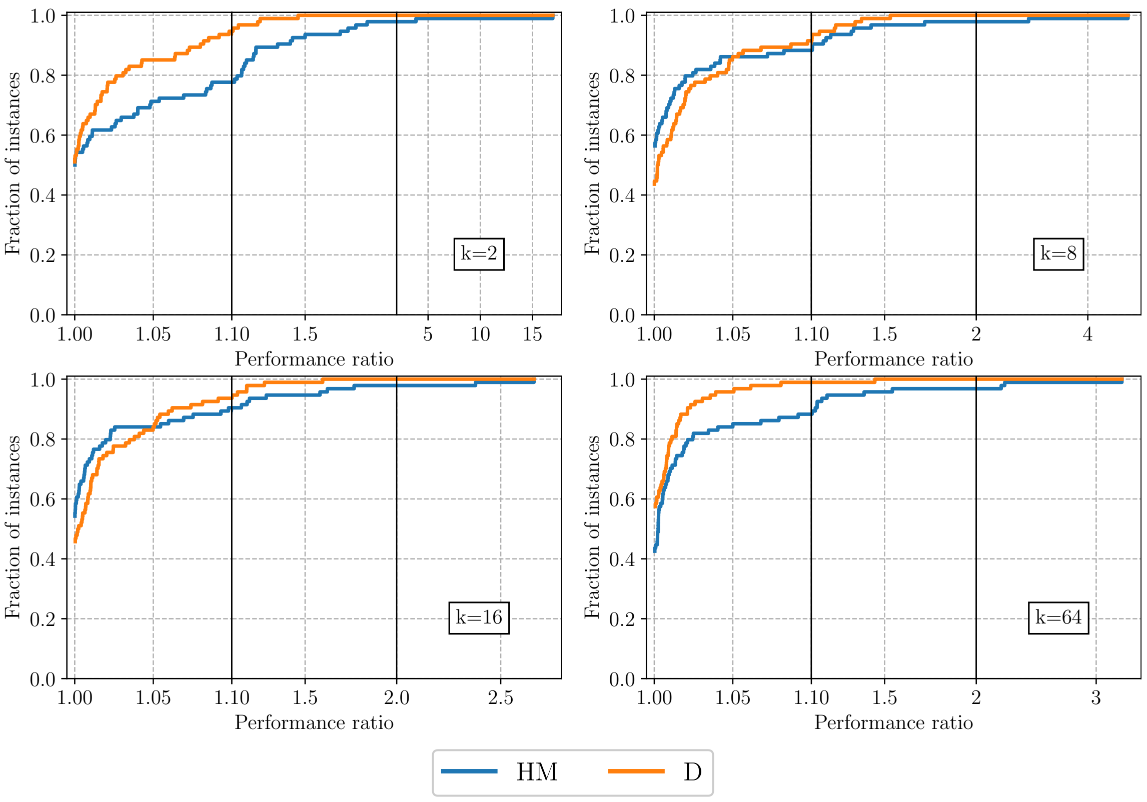

Figure E.2.: Comparing the quality of the hypergraph modularity approach with the current graph based approach for $k = \{2, 8, 16, 64, \}$ on set B.

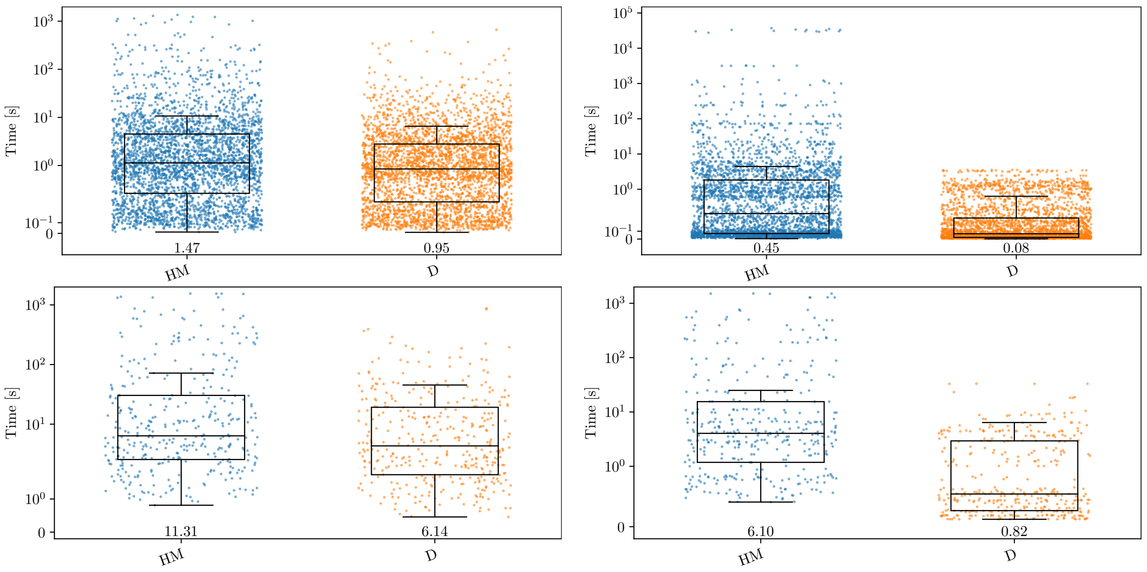

Figure E.3.: Comparing the absolute running time of the current configuration of `Mt-KaHyPar` (`D`) to the version with hypergraph modularity as a preprocessing step (`HM`) on benchmark set B. The top plots show the running times on set A and the bottom plots show the running times on set B. The left plot displays the total running time of the algorithm, while the right plots display the running time of the preprocessing step. The number below the box plot is the geometric mean running time.