Modularity Based Community Detection in Hypergraphs

Bogumił Kamiński[*] Paweł Misiorek[†] Paweł Prałat[‡] François Théberge[§]

August 22, 2024

Abstract

In this paper, we propose a scalable community detection algorithm using hypergraph modularity function, **h–Louvain**. It is an adaptation of the classical **Louvain** algorithm in the context of hypergraphs. We observe that a direct application of the **Louvain** algorithm to optimize the hypergraph modularity function often fails to find meaningful communities. We propose a solution to this issue by adjusting the initial stage of the algorithm via carefully and dynamically tuned linear combination of the graph modularity function of the corresponding two-section graph and the desired hypergraph modularity function. The process is guided by Bayesian optimization of the hyper-parameters of the proposed procedure. Various experiments on synthetic as well as real-world networks are performed showing that this process yields improved results in various regimes.

1 Introduction

Many networks that are currently modelled as graphs would be more accurately modelled as hypergraphs. This includes the collaboration network [5] in which nodes correspond to researchers and hyperedges correspond to papers that consist of nodes associated with researchers that co-author a given paper. Social events may include more than two people which is not equivalent to social interactions among all pairs of people participating in the event. Hypergraphs have shown promise in modeling systems such as protein complexes and metabolic reactions [18]. Another natural examples are co-purchases hypergraphs but there are plenty of other real-world hypergraphs.

After many years of intense research using graph theory in modelling and mining complex networks [17, 23, 30, 45], hypergraphs start gaining considerable traction [4, 5, 6, 8]. Many higher-order network data is being collected in recent years (see, for example, [5]). It has became clear to both researchers and practitioners that dyadic relationships are insufficient in many real-world scenarios. Higher-order network analysis, using the ideas of hypergraphs, simplicial complexes, multilinear and tensor algebra, and more, is needed to study complex systems and to make an impact across many important applications [6, 36, 52, 40]. Indeed, the inherent expressiveness of hypergraphs has led to their applications across a diverse range of fields such as recommendation

[*]Decision Analysis and Support Unit, SGH Warsaw School of Economics, Warsaw, Poland; e-mail: bogumil.kaminski@sgh.waw.pl

[†]Institute of Computer Sciences, Poznan University of Technology, Poznan, Poland; e-mail: pawel.misiorek@put.poznan.pl

[‡]Department of Mathematics, Toronto Metropolitan University, Toronto, ON, Canada; e-mail: pralat@torontomu.ca

[§]Tutte Institute for Mathematics and Computing, Ottawa, ON, Canada; email: theberge@ieee.org

systems [54], computer vision [42], natural language processing [16], social network analysis [44], financial analysis [56], bioinformatics [18], and circuit design [22]. Standard but important questions in network science are currently being revisited in the context of hypergraphs. However, hypergraphs also create brand new questions which did not have their counterparts for graphs. For example, how hyperedges overlap in empirical hypergraphs [41]? Or how the existing patters in a hypergraph affect the formation of new hyperedges [24]?

In this paper we concentrate on the classical problem of *community detection* in networks that can be represented using hypergraphs [1, 7, 11, 12, 27, 28, 34, 35, 57, 58]. Community detection is a challenging, NP-hard problem even for graphs [10, 19, 47] so obtaining an optimal solution becomes computationally infeasible, even for small networks represented as graphs. Dealing with hypergraphs is clearly much more difficult so, despite the fact that currently there is a vivid discussion around hypergraphs, the theory and tools are still not sufficiently developed to tackle this problem directly within this context. Indeed, researchers and practitioners, due to lack of proper solutions for hypergraphs, often create the 2-section graph of a hypergraph of interest (that is, replace each hyperedge with a clique, a process known also as clique expansion). Given the 2-section graph representation, we can directly apply some graph clustering algorithm such as **Louvain** [9] and **Leiden** [53]. Another approach is to perform agglomerative clustering via some definition of distance between nodes, such as the derivative graph defined in Contreras-Aso et al. [15], and then select the partition that maximizes the 2-section graph modularity. However, with the 2-section graph, one clearly loses some information about hyperedges of size greater than two. In the experiments presented in Section 5, we use the **Louvain** algorithm on the 2-section graph representations as our basis of comparison for hypergraph-based algorithms.

As mentioned earlier, there are some recent attempts to deal with hypergraphs in the context of clustering. For example, Kumar et al. [34, 35] still reduce the problem to graphs but use original hypergraphs to iteratively adjust weights to encourage some hyperedges to be included in some cluster but discourage other ones (this process can be viewed as separating signal from noise). In Chodrow et al. [12], a hypergraph stochastic block model is defined, leading to a Louvain-type clustering algorithm, in particular, for the "all or nothing" regime (**AON**), where edges must have all nodes from the same community to improve the objective function. We provide more details about these two algorithms at the beginning of Section 5.

Many of the successful graph clustering algorithms use the modularity function to benchmark partitions to guide the associated optimization heuristics. Two widely used algorithms from this family are the **Louvain** and **Leiden** algorithms mentioned earlier. Based on its spectacular success, a number of extensions of the classical graph modularity function to hypergraphs are proposed [27, 28] that can potentially be used by true hypergraph algorithms. In this paper, we concentrate on this approach.

Unfortunately, there are many ways such extension of the modularity function to hypergraphs can be done, depending on how often nodes in one community share hyperedges with nodes from other communities. We believe that the underlying process that governs *pureness* of community hyperedge is something that varies between networks at hand and also potentially depends on the hyperedge sizes. Let us come back to the collaboration network we discussed earlier. Hyperedges associated with papers written by mathematicians might be more homogeneous and smaller in comparison with those written by medical doctors who tend to work in large and multidisciplinary teams. Moreover, in general, papers with a large number of co-authors tend to be less homogeneous, and other patterns can be identified [24]. The algorithm we propose in this paper, **h–Louvain**, is flexible and can use any of such hypergraph modularity function. In other words, there is no unique way of extending the concept of modularity from graphs to hypergraphs. For this reason we consider a family of such extensions parametrized by the user's preference over homogeneity of within-community hyperedges. At the same time we recognize that there can be situations in which it might not be clear for a user what homogeneity level is desired. Therefore, in Section 5.1 we provide some suggestions to help the user to make the right choice.

A significant challenge in optimizing modularity functions is that these objective functions have their domains defined over all partitions of the set of nodes and they are known to be extremely difficult to optimize. As already mentioned, one of the most popular and efficient heuristic methods for modularity optimization for graphs is the **Louvain** algorithm [9]. In this paper, we show how this algorithm can be adapted to optimize hypergraph modularity. One of the main challenges is the fact that, when hyperedges of size two (edges) or three are not present in the hypergraph, then the **Louvain** algorithm immediately gets stuck in its local minimum. Moreover, even if there are a few hyperedges of size two or three, the algorithm may still get stuck almost immediately, and yield a solution that is heavily biased toward small edges. Hence, in such situations, one cannot simply start optimizing the hypergraph modularity right from the beginning. More importantly, we observe that even if hyperedges of size two are present in the hypergraph, the algorithm often converges to a local optimum that is of low quality. In order to address these two problems, we propose a method that works reasonably well in practice in which we optimize a weighted average of the 2-section graph modularity function and the hypergraph modularity function. For that we adjust the **Louvain** algorithm in such a way that the weight of the hypergraph modularity function increases during the optimization process. The pace of this weight change is governed by two hyperparameters of the procedure, which we tune using Bayesian optimization.

The paper is structured as follows. We first introduce the necessary notation; in particular, we state the definitions of graph and hypergraph modularity functions (Section 2). Synthetic as well as real-world hypergraphs that are used in our experiments are introduced in Section 3. Section 4 is devoted to explain details behind the proposed algorithm, **h–Louvain**. First, we discuss the classical **Louvain** algorithm for graphs (Subsection 4.1) and explain why it is difficult to adjust it to directly optimize hypergraph modularity (Subsection 4.2). Following this, we describe our solution that is considering a linear combination of the 2-section graph modularity and the hypergraph modularity as objective function (Subsection 4.3), and explain its implementation challenges (Subection 4.4). In particular, the main challenge is to tune the two hyperparameters responsible for the speed of convergence to the hypergraph modularity function. To find a "sweet spot" in an unsupervised way, Bayesian optimization is used (Subsection 4.5). Section 5 highlights the results of numerical experiments of using the proposed algorithm on synthetic hypergraphs (Subsections 5.2 and 5.3) as well as real-world hypergraphs (Subsection 5.4). We also highlight important implications of the choice of the modularity function to optimize (Subsection 5.1). The paper is concluded with a summary of outlooks for further research in this area (Section 6).

Finally, let us mention that this paper is an extended, journal version of the short, proceeding paper [25] that contained some preliminary experiments with a much simpler algorithm. The algorithm as well as notebooks containing all experiments included in this paper can be found on GitHub repository[*].

[*] `https://github.com/pawelwm/h-louvain`

2 Modularity Functions

Let us start with some basic definitions. In the hypergraph $H = (V, E)$, each hyperedge $e \in E$ is a multiset of V of any cardinality $d \in \mathbb{N}$ called its size. Multisets in the context of hypergraphs are natural generalization of loops in the context of graphs. Hypergraphs are natural generalization of graphs in which edge is a multiset of size two. Even though H does not always contain multisets, it is convenient to allow them as they may appear in the random hypergraph that will be used as the null model to "benchmark" the edge contribution component of the modularity function. It will be convenient to partition the hyperedge set E into $\{E_1, E_2, \ldots\}$, where E_d consists of hyperedges of size d. As a result, hypergraph H can be expressed as the disjoint union of *d-uniform hypergraphs* $H = \bigcup H_d$, where $H_d = (V, E_d)$. As for graphs, $\deg_H(v)$ is the degree of node v, that is, the number of hyperedges v is a part of (taking into account the fact that hyperedges are multisets). Finally, the volume of a subset of nodes $A \subseteq V$ is $\mathrm{vol}_H(A) = \sum_{v \in A} \deg_H(v)$.

Graph Modularity

The definition of modularity for graphs was first introduced by Newman and Girvan in [48]. Despite some known issues with this function such as the "resolution limit" reported in [20], many popular algorithms for partitioning nodes of large graphs use it [14, 38, 46] and perform very well. The two prominent ones from this family are **Louvain** [9] and **Leiden** [53]. The modularity function favours partitions of the set of nodes of a graph G in which a large proportion of the edges fall entirely within the parts (often called clusters), but benchmarks it against the expected number of edges one would see in those parts in the corresponding Chung-Lu random graph model [13] which generates random graphs with the expected degree sequence following exactly the degree sequence in G.

Formally, for a graph $G = (V, E)$ and a given partition $\mathbf{A} = \{A_1, A_2, \ldots, A_k\}$ of V, the *modularity function* is defined as follows:

$$q_G(\mathbf{A}) \quad = \quad \sum_{A_i \in \mathbf{A}} \frac{e_G(A_i)}{|E|} - \sum_{A_i \in \mathbf{A}} \left(\frac{\mathrm{vol}_G(A_i)}{\mathrm{vol}_G(V)} \right)^2 , \tag{1}$$

where $e_G(A_i)$ is the number of edges in the subgraph of G *induced by* set A_i. The first term in (1), $\sum_{A_i \in \mathbf{A}} e_G(A_i)/|E|$, is called the *edge contribution* and it computes the fraction of edges that fall within one of the parts. The second one, $\sum_{A_i \in \mathbf{A}} (\mathrm{vol}_G(A_i)/\mathrm{vol}_G(V))^2$, is called the *degree tax* and it computes the expected fraction of edges that do the same in the corresponding random graph (the null model). The modularity measures the deviation between the two.

The maximum *modularity* $q^*(G)$ is defined as the maximum of $q_G(\mathbf{A})$ over all possible partitions $\mathbf{A}$ of V; that is, $q^*(G) = \max_{\mathbf{A}} q_G(\mathbf{A})$. In order to maximize $q_G(\mathbf{A})$ one wants to find a partition with large edge contribution subject to small degree tax. If $q^*(G)$ approaches 1 (which is the trivial upper bound), we observe a strong community structure; conversely, if $q^*(G)$ is close to zero (which is the trivial lower bound), there is no community structure. The definition in (1) can be generalized to weighted edges (with weight function $w : E \to \mathbb{R}_+$), by replacing edge counts with sums of the corresponding edge weights.

Using Graph Modularity for Hypergraphs

Given a hypergraph $H = (V, E)$, it is common to transform its hyperedges into complete graphs (cliques), the process known as forming the 2-section of H or clique expansion, the graph $H_{[2]}$, on the same set of nodes as H. For each hyperedge $e \in E$ with $|e| \geq 2$ having weight $w(e)$, $\binom{|e|}{2}$ edges are formed, each of them with weight of $w(e)/\binom{|e|}{2}$. This choice preserves the total weight. There are other natural choices for the weight, for example the weighting scheme where $w(e)/(|e|-1)$ that ensures that the degree distribution of the created graph matches the one of the original hypergraph H [35, 34]. As hyperedges in H usually overlap, this process creates a multigraph. In order for $H_{[2]}$ to be a simple graph, if the same pair of vertices appear in multiple hyperedges, the corresponding edge weights are summed.

One of the approaches for finding communities in hypergraphs that practitioners use is to apply one of the algorithms that aim to maximize the original, graph modularity function (such as **Louvain**, **Leiden**, or **ECG**) to graph $H_{[2]}$. Despite the fact that this procedure is simple, it has a drawback that the 2-section graph looses some potentially useful information. Therefore, it is desired to define modularity function that is tailored explicitly for hypergraphs and aim to optimize it directly.

Hypergraph Modularity

For edges of size greater than 2, several definitions can be used to quantify the edge contribution for a given partition $\mathbf{A}$ of the set of nodes. As a result, the choice of hypergraph modularity function is not unique. It depends on how strongly one believes that a hyperedge is an indicator that some of its vertices fall into one community. The fraction of nodes of a given hyperedge that belong to one community is called its *homogeneity* (provided it is more than 50%). In one extreme case, all vertices of a hyperedge have to belong to one of the parts in order to contribute to the modularity function; this is the *strict* variant assuming that only homogeneous hyperedges provide information about underlying community structure. In the other natural extreme variant, the *majority* one, one assumes that edges are not necessarily homogeneous and so a hyperedge contributes to one of the parts if more than 50% of its vertices belong to it; in this case being over 50% is the only information that is considered relevant for community detection. All variants in between guarantee that hyperedges contribute to at most one part. This is an important difference from the modularity on $H_{[2]}$, where a single original hyperedge is split into multiple graph edges that could be considered as contributing to multiple different parts (communities). Once the variant is fixed, one needs to benchmark the corresponding edge contribution using the degree tax computed for the generalization of the Chung-Lu model to hypergraphs proposed in [27].

The hypergraph modularity function is controlled by *hyper-parameters* $\eta_{c,d} \in [0,1]$ ($d \geq 2$, $\lfloor d/2 \rfloor + 1 \leq c \leq d$). For a fixed set of hyper-parameters and a given partition $\mathbf{A} = \{A_1, A_2, \ldots, A_k\}$ of V, we define

$$q_H(\mathbf{A}) = \sum_{d \geq 2} \sum_{c=\lfloor d/2 \rfloor + 1}^{d} \eta_{c,d} \; q_H^{c,d}(\mathbf{A}), \tag{2}$$

where

$$q_H^{c,d}(\mathbf{A}) = \frac{1}{|E|} \sum_{A_i \in \mathbf{A}} \left(e_H^{c,d}(A_i) - |E_d| \cdot \Pr\left(\mathrm{Bin}\left(d, \frac{\mathrm{vol}(A_i)}{\mathrm{vol}(V)}\right) = c \right) \right);$$

$e_H^{c,d}(A_i)$ is the number of hyperedges of size d that have exactly c members in A_i, and $\mathrm{Bin}(d,p)$ is the binomial random variable, that is,

$$\Pr\left(\mathrm{Bin}(d,p) = c\right) = \binom{d}{c} p^c (1-p)^{d-c}.$$

Hyper-parameters $\eta_{c,d}$ give us a lot of flexibility and allow to value some hyperedges more than other ones depending on their size and homogeneity. However, there is a natural family of hyper-parameters that one might consider, namely, $\eta_{c,d} = (c/d)^\tau$ for some constant $\tau \in [0, \infty)$. We will refer to the corresponding modularity function as *τ-modularity function.* This family has only one parameter to tune, τ, but it still covers a wide range of possible scenarios. For example, one might want to value all hyperedges equally ($\tau = 0$) or value more homogeneous hyperedges more ($\tau > 0$), including the extreme situation in which only fully homogeneous hyperedges are counted ($\tau \to \infty$). In particular, we get the following four natural parameterizations of the modularity function to optimize:

- *strict modularity* ($\tau \to \infty$): $\eta_{d,d} = 1$ and $\eta_{c,d} = 0$ for $\lfloor d/2 \rfloor + 1 \le c < d$,
- *quadratic modularity* ($\tau = 2$): $\eta_{c,d} = (c/d)^2$ for $\lfloor d/2 \rfloor + 1 \le c \le d$,
- *linear modularity* ($\tau = 1$): $\eta_{c,d} = c/d$ for $\lfloor d/2 \rfloor + 1 \le c \le d$,
- *majority modularity* ($\tau = 0$): $\eta_{c,d} = 1$ for $\lfloor d/2 \rfloor + 1 \le c \le d$.

Note that regardless of the parameter τ, the weights are normalized so that $\max_c \eta_{c,d} = 1$ for all d. This ensures that the modularity function is normalized to be between 0 and 1.

As already mentioned above, the choice of the parameter τ should be made depending on how much more homogeneous hyperedges are valued compared to inhomogeneous ones. However, in an absence of any external intuition about the nature of the ground-truth communities, our suggestion is to use $\tau = 2$. This choice is justified based on the connection to $H_{[2]}$, the corresponding 2-section graph of H. Indeed, hyperedges of size d in H that have exactly c members in one of the communities contribute $\binom{c}{2}/\binom{d}{2} = \frac{c(c-1)}{d(d-1)} \approx (c/d)^2$ fraction of their original weight to the graph modularity function of $H_{[2]}$.

Having said that, let us stress the fact that optimizing the hypergraph 2-modularity function of H is *not* equivalent to optimizing the graph modularity function of $H_{[2]}$ since hyperedges with $c \le d/2$ members in one of the communities do not contribute to the hypergraph modularity whereas they still do in the graph counterpart.

Indeed, this observation highlights the key difference between our approach to extracting communities from the hypergraph and doing it via the corresponding 2-section graph $H_{[2]}$ that we already indicated when introducing hypergraph modularity. Our assumption is that there exists an underlying set of latent communities in the hypergraph (commonly referred to as the ground-truth). A given set of nodes appears as a hyperedge with the probability that depends on whether the majority of them are from one of the communities or not. As a result, hyperedges of size d in H that have at most $d/2$ members in one of the communities are considered as noise, that unnecessarily influences the modularity function of $H_{[2]}$. Indeed, the hypergraph modularity is guaranteed to count a single hyperedge at most for one community (as we require $c > d/2$). On the other hand, the graph modularity of $H_{[2]}$ potentially treats a single hyperedge as a positive signal contributing to multiple communities.

It is well known that optimizing modularity function in large networks might fail to resolve small communities, even when they are well defined. This well-known potential problem of applying a global null-models and is often referred to as the resolution limit [20]. A standard approach which tries to solve the resolution limit is to multiply the degree tax in the definition of the modularity function by a parameter $\gamma > 0$. This additional parameter controls the relative importance between the edge contribution and the degree tax. The hypergraph modularity function may be tuned the same way, if needed.

Finally, let us mention that for a given partition $\mathbf{A}$, the values of different modularity functions should not be compared, as they are scaled differently; rather the same modularity function should be used to rank various partitions for a given graph.

3 Hypergraphs Used in Our Experiments

In this section, we introduce the hypergraphs we use in our experiments, both synthetic and real-world ones. Despite the fact that there are several real-world hypergraphs with large hyperedges, we restrict ourselves to hypergraphs with a relatively small hyperedges. Large hyperedges, spanning multiple communities, are usually rare and do not provide any strong signal about the underlying communities. (Such hyperedges could be useful for other tasks such as node classification.) From that perspective, even if present, they are typically removed before running a clustering algorithm as part of the exploratory data analysis (EDA).

3.1 Synthetic Hypergraph Model: h–ABCD

There are very few hypergraph datasets with ground-truth identified and labelled. Synthetic networks are extremely useful to test various scenarios, such as the level of noise, via tuneable and interpretable parameters. As a result, there is need for synthetic random graph models with community structure that resemble real-world networks in order to benchmark and tune clustering algorithms that are unsupervised by nature.

It is worth mentioning that the family of clustering algorithms we are interested in aims to find partitions that maximize given modularity function, not to find the ground-truth partition. Those are often very similar partitions (but not always). Note that ground truth partitions typically influence the creation of a hypergraph in a noisy way, which means that just as a consequence of this randomness a good community in a graph (after the randomness is resolved) does not have to match exactly the ground truth community.

In particular, algorithm A would be considered better than algorithm B if it finds a partition yielding larger modularity. Selecting the right modularity function to optimize is crucial for making sure that the outcome of the algorithm is close to the ground-truth (or some specific requirements of the user), but once the function is selected the algorithm should aim to maximize it. We propose a simple, unsupervised method for making such selection (see Section 5.1) but this paper focuses on the optimization algorithm.

The standard for the generation of synthetic graphs is rather clear. The **LFR** (**L**ancichinetti, **F**ortunato, **R**adicchi) model [39, 37] generates networks with communities and at the same time it allows for the heterogeneity in the distributions of both node degrees and of community sizes. It became a standard and extensively used method for generating artificial networks. The **A**rtificial **B**enchmark for **C**ommunity **D**etection (**ABCD**) [29] was recently introduced and implemented[†], including a fast implementation[‡] that uses multiple threads (**ABCDe**) [33]. Undirected variant of **LFR** and **ABCD** produce graphs with comparable properties but **ABCD**/**ABCDe** is faster than **LFR** and can be easily tuned to allow the user to make a smooth transition between the two

[†]`https://github.com/bkamins/ABCDGraphGenerator.jl/`

[‡]`https://github.com/tolcz/ABCDeGraphGenerator.jl/`

extremes: pure (disjoint) communities and random graph with no community structure. Moreover, it is easier to analyze theoretically—for example, in [26, 2] various theoretical asymptotic properties of the **ABCD** model are investigated including the modularity function and self-similarities of the ground-truth communities.

The situation for hypergraphs is not as clear as for graphs. There are not only few real-world datasets (with ground-truth) available, but also there are not so many synthetic hypergraph models. Fortunately, the building blocks in the **ABCD** model are flexible and may be adjusted to satisfy different needs. For example, the model was adjusted to include potential outliers in [31] resulting in **ABCD+o** model. Adjusting the model to hypergraphs is more complex but it was also done recently [32] resulting in **h–ABCD** model. We will use this model for our experiments.

The **h–ABCD** model generates a hypergraph on n nodes. The degree distribution follows power-law with exponent γ, minimum and maximum value equal to δ and, respectively, D. Community sizes are between s and S, and also follow power-law distribution, but this time with exponent β. Parameter ξ is responsible for the level of noise. If $\xi = 0$, then each hyperedge is a community hyperedge meaning that majority of its nodes belong to one community. On the other extreme, if $\xi = 1$, then communities do not play any roles and hyperedges are simply "sprinkled" across the entire hypergraph that we will refer to as background hypergraph. Vector $(q_1, \ldots, q_L)$ determines the distribution of the number of hyperedges of a given size, where L is the size of largest hyperedges.

Finally, parameters $w_{c,d}$ specify how many nodes from its own community a given community hyperedge should have. We call a community hyperedge to be of type (c, d) if it has size d and exactly c of its nodes belong to one of the communities. Note that, in light of the discussion we had at the end of the previous section, we require that a community hyperedge must have more than a half of its nodes from the community. Therefore, $w_{c,d}$ is defined for $d/2 < c \le d$, where $d \in [L]$.

The model is flexible and may accept any family of parameters $w_{c,d}$ satisfying specific needs of the users, but here is a list of three standard options implemented in the code:

- *majority* model: $w_{c,d}$ is uniform for all admissible values of c, that is, for any $d/2 < c \le d$, $w_{c,d} = \frac{1}{(d-\lfloor d/2 \rfloor)} = \frac{1}{\lceil d/2 \rceil}$,
- *linear* model: $w_{c,d}$ is proportional to c for all admissible values of c, that is, for any $d/2 < c \le d$, $w_{c,d} = \frac{2c}{(d+\lfloor d/2 \rfloor+1)(d-\lfloor d/2 \rfloor)} = \frac{2c}{(d+\lfloor d/2 \rfloor+1)\lceil d/2 \rceil}$,
- *strict* model: only "pure" hyperedges are allowed, that is $w_{d,d} = 1$ and $w_{c,d} = 0$ for $d/2 < c < d$.

Let us note that the parameterizations of $w_{c,d}$ in **h–ABCD** and $\eta_{c,d}$ in the definition of the hypergraph modularity function have the same (due to matching functional form) name but they are not equivalent. Parameters $w_{c,d}$ determine the composition of hyperedges in the generated synthetic graph whereas parameters $\eta_{c,d}$ specify the objective function that the analyst decided to optimize against while looking for communities in the hypergraph at hand.

Specifically, we used the following parameters for our experiments:

- $n = 300$ nodes,
- power-law degree exponent $\alpha = 2.5$, in the range $[5, 30]$,
- power-law community size exponent $\beta = 1.5$, in the range $[80, 120]$.

We generated 6 families of **h–ABCD** hypergraphs, namely:

- `linear_2to5`: linear model for $w_{c,d}$, with edge sizes 2 to 5 (with respective probabilities $0.1, 0.4, 0.4, 0.1$),
- `majority_2to5`: majority model for $w_{c,d}$, with edge sizes 2 to 5 (with respective probabilities $0.1, 0.4, 0.4, 0.1$),
- `strict_2to5`: strict model for $w_{c,d}$, with edge sizes 2 to 5 (with respective probabilities $0.1, 0.4, 0.4, 0.1$),
- `linear_5`: linear model for $w_{c,d}$, with all edge of size 5,
- `majority_5`: majority model for $w_{c,d}$, with all edge of size 5, and
- `strict_5`: strict model for $w_{c,d}$, with all edge of size 5.

3.2 Real-world Hypergraphs

To illustrate various aspects of hypergraph modularity-based clustering, we analyze a few real-world hypergraphs. The first is a contact hypergraphs in which nodes correspond to primary school children or teachers and hyperedges represent close physical proximity between individuals within a prescribed time period (see [12, 51, 43] for more details). There are 242 nodes labelled with respect to their class (there are 10 classes), plus another label for the teachers. There are 12,704 hyperedges of size up to 5. In Table 1 (left), we show the distribution of edge composition for this dataset with respect to the ground-true communities. We see that there are many edges between communities, in particular edges of size 2. Community edges (with $c > d/2$) are mostly "pure" edges (with $c = d$), but there is a significant number of edges of type $(c, d) = (2, 3)$, thus it is unclear if hypergraph τ-modularity functions with large parameter τ would do well, or if a small value for τ should be used.

`primary-school`

d	c	purity	frequency
2	1	50%	5202
2	2	100%	2546
3	3	100%	2434
3	2	67%	1751
3	1	33%	415
4	4	100%	158
4	2	50%	93
4	3	75%	84
4	1	25%	12
5	3	60%	6

`cora`

d	c	purity	frequency
2	2	100%	472
3	3	100%	307
4	4	100%	175
2	1	50%	151
3	2	67%	118
4	3	75%	91
5	5	100%	83
5	4	80%	55
4	2	50%	42
3	1	33%	39

Table 1: The number of hyperedges of type (c, d) (the top-10 most frequent ones) for the `primary-school` dataset (left) and the `cora` co-citation dataset (right). (Combinations contributing to hypergraph modularity are highlighted in grey.)

Another dataset we use for our experiments in the co-reference dataset between scientific publication which belong to one of seven classes (`cora`); see [55] for more details. Hyperedges consist of co-cited scientific publications, and we only keep hyperedges of size 2 or more. There are 1,434 nodes appearing in at least one hyperedge (cited publications), and 1,579 hyperedges. In Table 1 (right), we show the top-10 distribution of edge composition with respect to the true communities. We see that there are many pure community edges (with $c = d$), so we can expect that hypergraph τ-modularity functions with large parameter τ would do well.

Having node attributes that represent a proxy of the corresponding ground-truth communities is convenient, as one can easily test the quality of various algorithms by measuring the similarity between partitions returned by them and the "ground-truth" partition. However, it is important to keep in mind that, in general, the ground-truth is unknown and one can only make the hypothesis that the attributes used are correlated to it. Indeed, it is often impossible to ensure that these observed discrete-valued node attributes used as labels are good representation of an underlying mechanism responsible for creation of communities [50].

4 Hypergraph Modularity Optimization Algorithm: h–Louvain

Let us fix the hypergraph modularity function $q_H(\mathbf{A})$, either by restricting ourselves to τ-modularity function with some specific value of τ (such as $\tau = 2$ that is recommended as the default value) or by specifying the more general hyper-parameters $\eta_{c,d}$. The goal of this section is to highlight challenges in designing a heuristic algorithm aiming to optimize $q_H(\mathbf{A})$ and to describe our solution that overcame these challenges, producing an algorithm that we will refer to as **h–Louvain**.

4.1 Louvain Algorithm

Let us start by introducing one of the most popular algorithms for detecting communities in graphs, namely, the **Louvain** algorithm [9]. It is a hierarchical clustering algorithm that tries to optimize the modularity function we described in Section 2.

In the first pass of this algorithm, small communities are found by optimizing the graph modularity function locally on all nodes. Then, each small community is grouped together into a single node that we will refer to as super-node. This process is repeated recursively on those smaller graphs consisting of super-nodes (the subsequent passes) until no improvement on the modularity function can be further achieved.

One pass of the algorithm consists of two phases that are repeated iteratively. In the first phase, each node in the network is assigned to its own community. For each node v, we consider all neighbours u of v and compute the change in the modularity function if v is removed from its own community and moved into the community of u. It is important to mention that this value can be easily and efficiently calculated without the need to recompute the modularity function from scratch. Once all the communities that v could belong to are considered, v is placed into the community that resulted in the largest increase of the modularity function. If no increase is possible, v remains in its original community. The process is repeated for the remaining nodes following a given (typically random) permutation of nodes, possibly multiple times, until a local maximum value is achieved and the first phase ends.

During the second phase, the algorithm contracts all nodes that belong to one community into a single super-node. All edges within that community are replaced by a single weighted loop. Similarly, all edges between two communities are replaced by a single weighted edge. Once the new network is created, the second phase ends. The resulting graph is typically much smaller than the original graph. As a result, the first pass is typically the most time consuming part of the algorithm.

4.2 Challenges with Adjusting the Algorithm to Hypergraphs

One could try to directly apply the **Louvain** algorithm to optimize hypergraph modularity, since in both cases the goal is to find a partition of the set nodes. However, as the algorithm moves only one node at a time, it creates a problem in the case of hypergraphs.

Consider, for example, a hypergraph in which all hyperedges have size at least four. In this case, regardless which two nodes u and v are considered for possible merging into one community, the edge contribution would not change (that is, it would stay equal to zero), even if u and v are part of some hyperedge. (Recall that only hyperedges with majority of nodes from the same community may affect the edge contribution). On the other hand, the degree tax would increase after such a move and, as a result, the modularity function would decrease. Therefore, no move would be made and the algorithm would get immediately stuck. We will refer to this issue as a *lift off from the ground* problem.

The above, extreme, situation is not the only problem one should be aware of. This time consider a hypergraph that consists of a mixture of hyperedges of various sizes, including edges of size two. In this scenario there is no problem with lifting off from the ground but small hyperedges clearly play a much more important role than large ones during the initial merging in the first phase of the algorithm. On the other hand, very large hyperedges would be mostly ignored. This behaviour is not desirable either. In order to illustrate a potential danger, consider a hypergraph representing interactions between researchers at some institution. Nodes in this hypergraph correspond to researchers and hyperedges correspond to meetings of some groups of people. For simplicity, assume that there are two communities, say, faculty of science and faculty of engineering. Many hyperedges within the two communities are large (e.g. hyperedges associated with departmental meetings) whereas hyperedges between the two communities are mostly of size two (e.g. two members of different teams meet individually from time to time). In this scenario, the algorithm would start merging people from different communities during the first phase.

Finally, let us note that one could alternatively consider modifying the algorithm and allow for not only merging two nodes into one community in a single move but entire hyperedges. Again, this does not seem to be desirable as hyperedges might consist of members from different communities and so such operations would generate many incorrect merges too fast.

4.3 Our Approach to Hypergraph Modularity Optimization

In order to overcome the above mentioned challenges, we want to design an algorithm that, as in the classical **Louvain** algorithm, merges single pairs of nodes while, at the same time, takes into account information stored in hyperedges of all sizes. To that end we propose to optimize a linear combination of the hypergraph modularity $q_H(\mathbf{A})$ and the graph modularity of the corresponding 2-section graph $H_{[2]}$, that is, optimize function

$$q(\mathbf{A},\alpha) := \alpha \cdot q_H(\mathbf{A}) + (1-\alpha) \cdot q_{H_{[2]}}(\mathbf{A}), \tag{3}$$

where $\alpha \in [0,1]$. For simplicity, we will refer to our algorithm as **h–Louvain**.

To understand the motivation behind this approach, let us observe the following. The hypergraph modularity, equation (2), is flexible and may approximate well the graph modularity for the corresponding 2-section graph $H_{[2]}$. Indeed, if c vertices of a hyperedge e of size d and weight

$w(e)$ fall into one part of the partition $\mathbf{A}$, then the contribution to the graph modularity is $w(e)\binom{c}{2}/\binom{|e|}{2} \approx w(e)(c/|e|)^2$ (in the variant of the 2-section where the total weight is preserved) or $w(e)\binom{c}{2}/(|e|-1)$ (if the degrees are preserved). Hence, the hyper-parameters of the hypergraph modularity can be adjusted to approximate $H_{[2]}$ modularity. The only difference is that (2) does not allow to include contributions from parts that contain at most $d/2$ vertices which still contributes to the graph modularity of $H_{[2]}$.

The observation justifies using $q(\mathbf{A}, \alpha)$ for optimizing the hypergraph modularity. It is a linear combination of the actual hypergraph modularity we want to optimize, $q_H(\mathbf{A})$, and an approximation of the hypergraph modularity for special value of hyper-parameter ($\tau = 2$) and without the restriction of hyperedge contribution, $q_{H_{[2]}}(\mathbf{A})$. The benefit of the second part is that it is sensitive to merging two nodes and so it always gives some indication of how nodes should be merged (even if the first part $q_H(\mathbf{A})$ does not give such an indication). In short, it resolves the *lifting off from the ground* problem. If α is close to zero, then we concentrate mostly on the approximation part, while if α is close to one, then we mostly concentrate on the actual hypergraph modularity we aim to optimize.

The above discussion leads us to the conclusion that the parameter $\alpha \in [0, 1]$ should be appropriately tuned during the algorithm. The main questions are: a) when the change should be made, and b) what values of this parameter should be used? In [25], we performed various experiments and made the following observations. The optimization process should start with low values of the parameter α (to let the process *lift off from the ground*) and then it should be gradually increased till it reaches one by the end of the process. The algorithm should start increasing parameter α when the communities induce enough edges so that merging additional nodes makes a difference in the edge contribution of the q_H function value; this, in particular, means that since the strict hypergraph modularity pays attention to only pure hyperedges (all members belong to one community), in this case, the algorithm needs to start with lower values of α and increase it slower than for the majority or the linear counterparts of the hypergraph modularity for which it is enough that over 50% of nodes in some hyperedge are captured in one community.

Based on these observations, we propose the following schema for setting the successive values of α used in the objective function (3), which leads to monotonic (non-decreasing) sequences $(\alpha_1, \alpha_2, \dots)$. The schema is guided by the following two parameters: $p_b \in [0, 1]$ and $p_c \in (0, 1)$. The parameter p_b is used to determine the values of α_i, while p_c governs when the algorithm switches from α_{i-1} to α_i (for $i \geq 2$) as the optimization progresses.

For a given pair of parameters (p_b, p_c), the values of α_i are determined as follows: for any $i \in \mathbb{N}$,

$$\alpha_i = 1 - (1 - p_b)^{i-1}.$$

(We use the convention that $0^0 = 1$.) Note that $\alpha_1 = 0$ and $\alpha_i \to 1$ as $i \to \infty$, unless $p_b = 0$. In the degenerate case, if $p_b = 0$, then $\alpha_i = 0$ for all i. The algorithm switches from α_{i-1} to α_i's (for $i \geq 2$) when the number of communities drops to np_c^{i-1} or below for the first time (note that the number of communities typically decreases but it is not always the case; as usual, n denotes the number of nodes). In summary, the two parameters have the following interpretation: p_b controls the rate of change of α (values close to zero make α_i converge to one slowly, values close to one make convergence fast); p_c controls the speed of change of α.

There are two possible endings once the algorithm reaches a partition in which no improvement of the modularity function is possible via local changes. (Note that it might happen when the value of α_i is still away from one.) By default, we fix $\alpha_i = 1$ and continue optimizing the hypergraph modularity function on the small graph consisting of super-nodes until no further improvement can be achieved. Alternatively, the local optimization can be performed on the original graph consisting of nodes. The pseudo-code of **h–Louvain** can be found in the Appendix (see Section A). As it is discussed in Section 4.5, we use the default ending when doing Bayesian optimization to select a good pair (p_b, p_c) of parameters because it is faster. Once the final pair is chosen, we do an additional tuning process with local-optimization enabled which typically yields better values of the objective function.

4.4 Parameters of the h–Louvain Algorithm

In this subsection, we aim to investigate the quality of the **h–Louvain** algorithm for different pairs of parameters (p_b, p_c). To that end, we analyzed the performance of the algorithm using 9 different **h–ABCD** graphs on 1,000 nodes. For each of the three options for community hyperedges in the **h–ABCD** model (namely, strict, linear, and majority), we used the following three settings with respect of different levels of noise and sizes of hyperedges:

1. small level of noise ($\xi = 0.15$, $\xi_{emp} = 0.29$), hyperedges of size between 2 and 5, the degree distribution following power-law with exponent $\gamma = 2.5$, minimum and maximum degree 5 and 20,

2. large level of noise ($\xi = 0.6$, $\xi_{emp} = 0.62$), hyperedges of size between 2 and 5, the degree distribution following power-law with exponent $\gamma = 2.5$, minimum and maximum degree 5 and 20,

3. large hyperedges (sizes between 5 and 8), large level of noise ($\xi = 0.3$, $\xi_{emp} = 0.63$), the degree distribution following power-law with exponent $\gamma = 2.5$, minimum and maximum degree 5 and 60.

(In the above description, ξ_{emp} refers to the actual level of noise in the produced hypergraph. The model ensures that $\xi_{emp} \approx \xi$ for graphs without small communities. In our scenario, it is not the case but the generated hypergraphs still have drastically different levels of noise.) In all three settings, the distribution of community sizes follows power-law with exponent $\beta = 1.5$, minimum and maximum size 10 and 30. The distribution of hyperedges of different sizes is $(0.4, 0.3, 0.2, 0.1)$, that is, there are slightly more hyperedges of smaller size.

Figure 1 presents the performance of the algorithm for three selected hypergraphs out of the 9 we experimented with. (Results for the remaining six hypergraphs can be found in the associated GitHub repository.) For each hypergraph, we present the quality of the algorithm optimizing the corresponding modularity function (that is, for example, for strict hypergraph we optimize the strict modularity function) as a function of (p_b, p_c). The parameters were tested from the 2-dimensional grid (p_b, p_c), where $p_b \in \{0.05, 0.1, 0.3, 0.5, 0.7, 0.9, 0.95\}$ and $p_c \in \{0.1, 0.2, \ldots, 0.9\}$. For each pair of parameters, the average modularity function is reported over 10 independent runs with different random seeds. (Recall that **h–Louvain** is a randomized algorithm.)

The general conclusion is that the optimal choice of parameters depends on many factors: property of the hypergraph (such as the composition of community hyperedges, the level of noise, sizes of hyperedges) as well as the modularity function that one aims to maximize. However, not surprisingly, it is not recommended to set both parameters to be close to zero (the case of slow and small increases of the alpha parameter, so optimizing mainly the graph modularity of the corresponding 2-section graph $H_{[2]}$), or to be close to one (the case of fast and significant changes of the alpha parameter, so

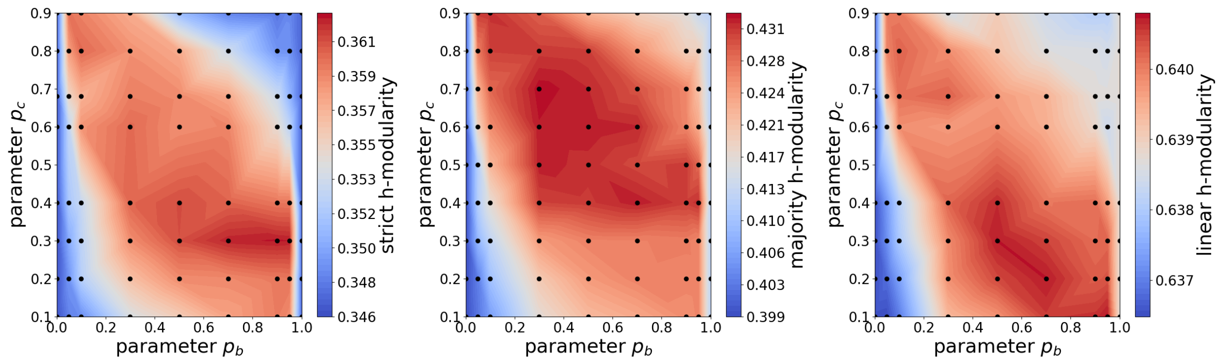


Figure 1: Quality of **h–Louvain** on **h–ABCD** as a function of parameters p_b, p_c. Optimal combinations of the two parameters depend on the choice of **h–ABCD** variant and hypergraph modularity function: strict, large level of noise (left), majority, large level of noise (middle), or linear, small level of noise (right).

optimizing the hypergraph modularity q_H almost from the beginning of algorithm execution). The optimal values are often obtained for settings with balanced values of both parameters, namely, with $p_b + p_c \approx 1$. In order to find a "sweet spot" in an unsupervised way, we use Bayesian optimization that we discuss next. Let us remark that one can expect to see large regions of the parameter space yielding low quality solutions (blue and light red zones on Figure 1). Therefore, using a simpler hyperparameter tuning technique such as the grid search would be less efficient. The algorithm would spend a lot of time investigating regions of the hyperparameter space that are not promising. On the other hand, Bayesian optimization avoids scanning such regions in detail and performs more dense search in more promising regions instead.

4.5 Bayesian Optimization: Selecting the Parameters

In order to find a good pair of the two parameters (p_b, p_c) guiding the **h–Louvain** algorithm that yield large hypergraph modularity function, we use the Bayesian optimization approach [21]. We chose this tool for our problem as this approach is best suited for optimizing objective functions that take a long time to evaluate over continuous domains of less than 20 dimensions, and tolerates well non-negligible local variability of the evaluation of the function. It builds a surrogate for the objective function and quantifies the uncertainty in that surrogate using a Bayesian machine learning technique, Gaussian process regression, and then uses an acquisition function defined from this surrogate to decide where to sample the domain in an on-line fashion. Specifically, we use the [49] package providing the Bayesian optimization procedure. All general properties of time complexity and performance follow this implementation. Below we describe in detail how we specifically prepare an input for this optimization in our specific optimization problem.

Specifically, in our case the Bayesian optimization aims to explore the two dimensional space (p_b, p_c) with $p_b \in [0, 1]$ and $p_c \in (0, 1)$. The target function is defined as the average modularity function of the outcome partition for 10 independent executions of the **h–Louvain** algorithm with different (but fixed across runs) random number generator seeds. Note that in this setting we maximize a deterministic function (since the seeds are fixed). We take the average over 10 different seeds because we aim to identify the region of the (p_b, p_c) domain that leads to good values of the obtained evaluations of modularity and taking the average reduces the noise that is present in modularity values observed in single runs of the algorithm.

Because tuning hyper-parameters is computationally intensive, we initially use the default ending of the algorithm, that is, without the local-optimization approach for the last phase (see Section 4.3 for an explanation how the optional ending works). The reason for this choice is that in this phase of the process we are mostly interested in capturing the shape of the response surface (recall that we take the average of 10 runs of the algorithm for the very same reason, namely, to smooth-out the results and to better capture the shape of the response surface). The default ending is sufficient for this purpose and it is substantially faster. After finding an approximation of the optimal (p_b, p_c) combination, the algorithm comes back to the partition obtained with these parameters but this time the local-optimization approach is used during the last phase. It is more computationally expensive, but at this stage of the procedure we are interested in finding the maximum value of the hypergraph modularity, and so this additional computational cost is justified.

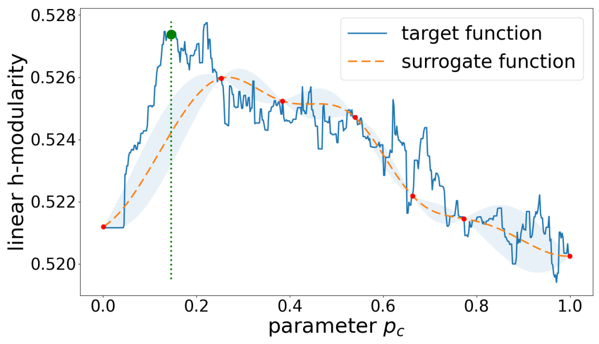
We configured the Bayesian optimization procedure so that it starts with evaluation of 5 initial pairs of parameters selected randomly from the domain and at least 10 pairs are tested in total. Once the Bayesian optimization converges, the algorithm returns the partition of the largest modularity from all partitions generated during the entire process. Note that this partition might not be one of the 10 partitions that contributed to the largest value of the target function; these partitions only have the best average modularity.

In order to visualize the Bayesian optimization procedure, we performed the following experiment. We selected one of the nine **h–ABCD** hypergraphs we experimented with (namely, the linear hypergraph with small level of noise, but this time with only $n = 300$ nodes) and one of the three modularity functions (namely, the linear one) to be our target function. For cleaner visualization, we fixed $p_b = 0.9$ and used the procedure to find the optimum value of p_c that maximizes the selected modularity function. Figure 2 presents situation at step k of the algorithm for $k \in \{8, 9, 10, 11\}$. The blue curve is the target function that we independently computed but it is not available to the Bayesian optimization. The orange curve is a surrogate for the target function based on $k - 1$ sampled points that are marked on this curve. The level of uncertainty is represented by the shaded area around this curve. Based on this information, the Bayesian optimization selects the next point to sample at this step which is depicted as a green point that lies outside of the orange curve and a green vertical line. Note that the blue curve is deterministic (as we use fixed seeding of random number generator), as discussed above. It still has visible local variability, although it is possible to identify the region of good values of the parameter p_c (in this case around 0.2). This variability is expected and is the reason why when computing it we take the average of 10 independent evaluations of the algorithm (this approach significantly reduces the level of noise).

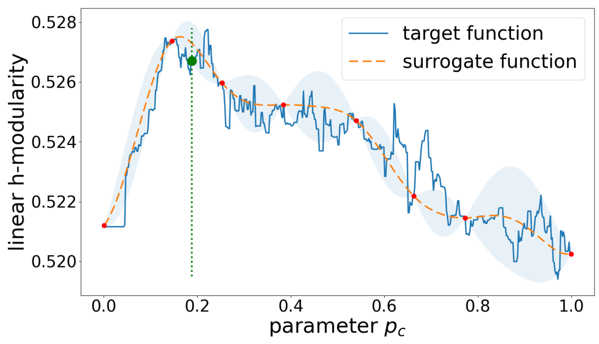
4.6 Computational Complexity

The computational complexity of the proposed algorithm is the same as for **Louvain** algorithm (both in terms of the number of nodes $|V|$ and in terms of the number of hyperedges $|E|$), as we follow the same 2-phase process. The difference is that for computing the hypergraph modularity function we have additional parameters that affect the runtime of our algorithm.

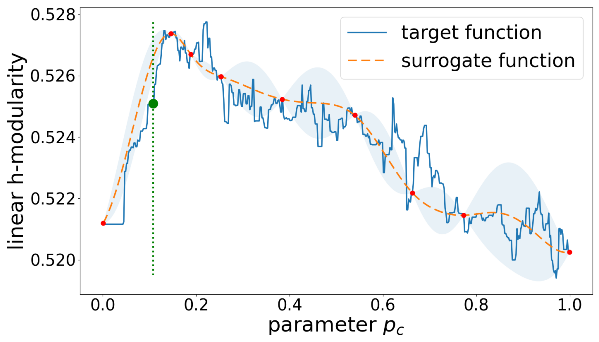
The first one is the maximum hyperedge size $d_{\max} = \max_{e \in E}\{|e|\}$, where the worst-case complexity of the algorithm is quadratic, i.e., $O(d_{\max}^2)$; recall that the hypergraph modularity function consists of $O(d_{\max}^2)$ "slices" (see (2)). However, in practice, this cost is lower as in the implementation we use caching of the intermediate results of the computations. This approach has proven empirically to significantly lower the run-time of the algorithm.



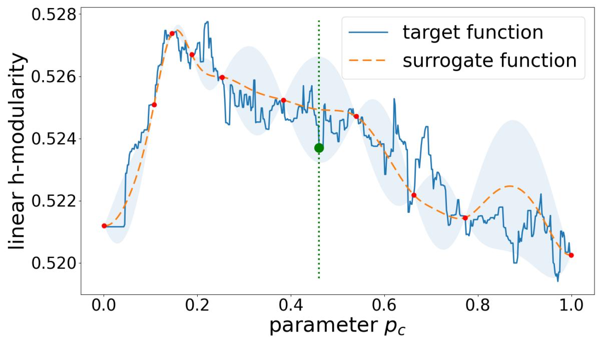
(a) Step 8



(b) Step 9



(c) Step 10



(d) Step 11

Figure 2: Visualization of the Bayesian optimization approach optimizing the modularity function returned by the **h–Louvain** algorithm.

The second one is associated with the selection of the hyperparameters of the hypergraph optimization procedure. This part only adds a constant multiplicative term to the total cost of computations. This constant term depends on the decision of the user when to stop Bayesian optimization and, in general, it can be potentially significant.

5 Experiments

In this section, we present several experiments aimed at testing our **h–Louvain** clustering algorithm as well as comparing outcomes of selecting various hypergraph modularity functions. One general observation is that the choice of the objective function to optimize, here the hypergraph τ-modularity, typically has an enormous impact on the quality of the results (see Subsection 5.1). Fortunately, one should be able to make a reasonable selection of a good value of τ in an unsupervised way.

In general, in most of our experiments, we compare results obtained with our **h–Louvain** algorithm with results obtained by classical **Louvain** algorithm on the corresponding (weighted) 2-section graphs, as well as results using **Kumar**'s algorithm; see [35]. **Kumar**'s algorithm is a modification of the **Louvain** algorithm on 2-section graphs in which edges are re-weighted by taking into account the underlying hypergraph structure, and the hyperedge composition (with respect to the communities).

We consider synthetic hypergraphs with community structures, obtained via the **h–ABCD** benchmark, as well as two real-life hypergraphs, all of them described earlier in Section 3. Synthetic hypergraphs allow us to investigate the performance of algorithms in various scenarios (see Subsection 5.2), from hypergraphs with low level of noise (ξ close to 0) that are easy to deal with to noisy hypergraphs (ξ close to 1) that are challenging to find communities in. We also investigate a challenging case in which there are many hyperedges between two small communities (see Subsection 5.3). It is known that many networks exhibit self-similar, "fractal-type" structure (see [2, 3] and references therein) so such example aims to reproduce typical scenarios. This example highlights the power of our **h–Louvain** algorithm that in this particular setting clearly outperforms its competitors.

For the real hypergraphs, we additionally consider the "all of nothing" (**AON**) variant of the **Louvain** algorithm. Specifically, we consider the version aiming to optimize the strict modularity, referred to as **AON**, as described in the associated GitHub repository[§]. Note that, unless we start from some non-trivial partition such as the one obtained from the 2-section graph with **Louvain**, this algorithm requires 2-edges to be present. This is the case for the real hypergraphs we considered (but not so for several **h–ABCD** benchmark hypergraphs).

In general, the experiments on real-world hypergraphs (see Subsection 5.4) show that for appropriate value of τ affecting the choice of the objective function to optimize, one can improve the quality of the clusters measured by the **AMI** score with respect to the ground-truth.

5.1 Selecting the Modularity Function to Optimize

Selecting an appropriate hypergraph modularity function to optimize is an important part of the process. The choice depends on how strongly one believes that a hyperedge is an indicator that at least some fraction of its nodes fall into one of the communities. In some situations, a reasonable assumption could be that not necessarily all members of that hyperedge must be in a single community but majority should (in such situations, quadratic modularity function might work well). On the other hand, some situations might have some underlying physical constraints that make one believe that all members should belong to one community unless such hyperedge is simply a noise (this time, strict modularity might be the one to optimize). If an analyst has some reasonable assumptions about the underlying process that created a hypergraph, then the decision which modularity function to use should be made based on this expert knowledge. In this section, we provide a general strategy for selecting a modularity function if such expert knowledge is not available, based on the structure of the hypergraph that can be detected in an unsupervised way.

Let us first start with highlighting important implications of the choice of the modularity function one decides to optimize. Recall that a community hyperedge (hyperedge with more than 50% of members from one of the communities) of size d that have exactly c members from one of the communities is said to be of type (c, d). In the absence of having the ground-truth available, one way to compare partitions returned by the algorithm aiming to optimize different modularity functions is to look at the distribution of edges of certain types. In our first experiment, we consider a synthetic **h–ABCD** graphs with only edges of size 5 and generated with the strict or the majority model, `strict_5` and `majority_5` with noise parameters $\xi = 0.3$ and $\xi = 0.2$. We run this experiment to

[§]`https://github.com/nveldt/HyperModularity.jl` show how the (c, d) hyperedge composition changes under various modularity functions optimized. We expect that there should be visible difference in this distribution between `strict_5` and `majority_5` graphs independent of the choice of the objective function for community detection.

In Table 2, we compare the distribution of edge types for 5 different partitions, namely, (i) the ground-truth partition, and partitioned returned by (ii) 2-section **Louvain**, (iii) **h–Louvain** with $\tau = 2$ (quadratic modularity), (iv) **h–Louvain** with $\tau = 3$ (cubic modularity), and (v) **h–Louvain** with $\tau \to \infty$ (strict modularity). We count the number of edges with 5, 4 or, respectively, 3 nodes in the most frequent community; the remaining edges are considered to be noise.

Majority class size	Ground-truth communities	**Louvain** 2-section	**h–Louvain** $\tau = 2$	$\tau = 3$	$\tau \to \infty$ (strict)
Strict with noise parameter $\xi = 0.2$					
5	352	352	349	352	352
4	36	36	41	36	36
3	92	92	91	92	92
≤ 2	42	42	41	42	42
Strict with noise parameter $\xi = 0.3$					
5	314	314	311	314	314
4	30	30	35	30	30
3	123	123	122	123	123
≤ 2	55	55	54	55	55
Majority with noise parameter $\xi = 0.2$					
5	169	170	137	169	264
4	175	165	171	174	154
3	137	138	161	137	104
≤ 2	41	49	53	42	0
Majority with noise parameter $\xi = 0.3$					
5	158	129	88	158	206
4	145	140	148	144	147
3	158	151	196	161	169
≤ 2	61	102	90	59	0

Table 2: The number of hyperedges of each type for **h–ABCD** hypergraphs with 5-edges generated with strict (`strict_5`) or majority (`majority_5`) assignment rule.

In the second column of in Table 2 we show hyperedge composition of the ground truth. As expected, for hypergraph with the strict model used the $(5, 5)$ hyperedges are most common, and for hypergraph with the majority model used $(5, 5)$, $(4, 5)$, and $(3, 5)$ hyperedges have similar frequencies (this holds both for $\xi = 0.2$ and $\xi = 0.3$). The crucial observation is that regardless of which of the modularity function is optimized (**Louvain** 2-section or **h–Louvain** with varying τ), the recovered hyperedge composition is similar to the ground truth. This observation suggests using the following approach in cases where the user does not have a prior preference for the τ parameter in the **h–Louvain** algorithm.

As a rule of thumb, running a quick clustering (for example with 2-section **Louvain**) as a part of Exploratory Data Analysis (EDA), and looking at the composition of edge types is a recommended first step that can be used to decide on the value(s) of τ one wants to use as the objective τ-modularity function for **h–Louvain**. In general, there are two major possible scenarios that the user could consider. Seeing mostly "'pure" edges suggests using large value of τ (or strict modularity), while the opposite suggests using a smaller values of τ such as $\tau = 2$ or $\tau = 3$.

5.2 Synthetic h–ABCD Hypergraphs

We ran a series of experiments using the synthetic **h–ABCD** benchmark hypergraphs. For each family of hypergraphs, we considered a wide range of values for the noise parameter ξ, and for each ξ, we generated 30 independent copies of **h–ABCD** hypergraphs. For each hypergraph, we obtained clusterings in various ways:

- taking the 2-section (weighted) graph and applying the **Louvain** algorithm several times, keeping the partition with the largest (graph) modularity;
- running **Kumar**'s algorithm;
- running our **h–Louvain** algorithm with Bayesian optimization for $\tau = 2$ and $\tau = 3$, and
- running our **h–Louvain** algorithm with Bayesian optimization using the strict modularity ($\tau \to \infty$) as the objective function.

In the analysis of the results, we computed the **AMI** of each partition with respect to the ground truth communities. The plots in Figures 3–5 show the difference of the **AMI** of a given algorithm and the **AMI** of 2-section result. In other words, we measure how much gain/loss is obtained by switching from a standard 2-section approach to finding communities in a hypergraph to our algorithm designed specifically for hypergraphs.

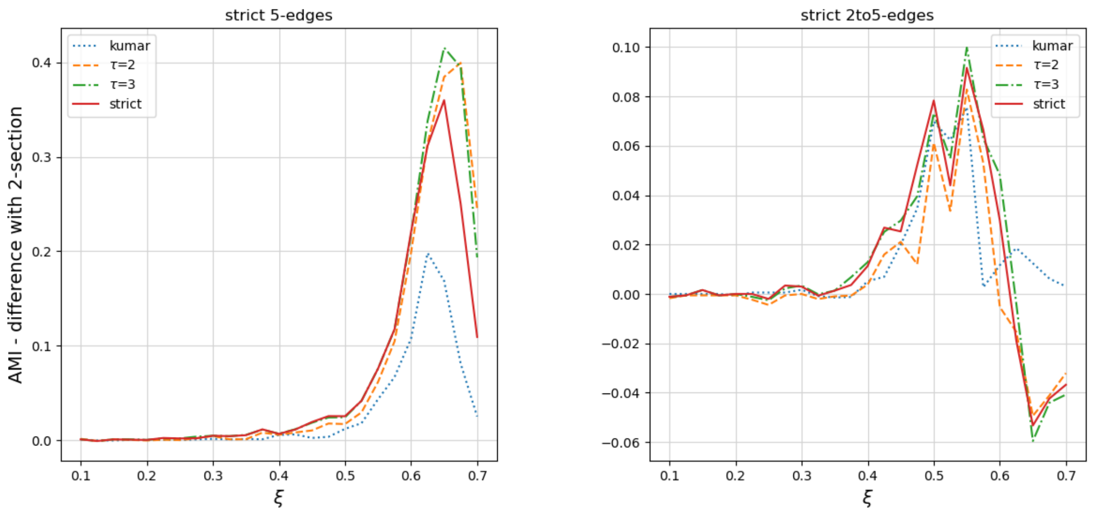


Figure 3: Results with **h–ABCD** hypergraphs with strict model for the community edge composition, (`strict_5` and `strict_2to5`) showing **AMI** difference between 2-section communities and the considered algorithms. Positive values indicate increase of **AMI** for a given algorithm.

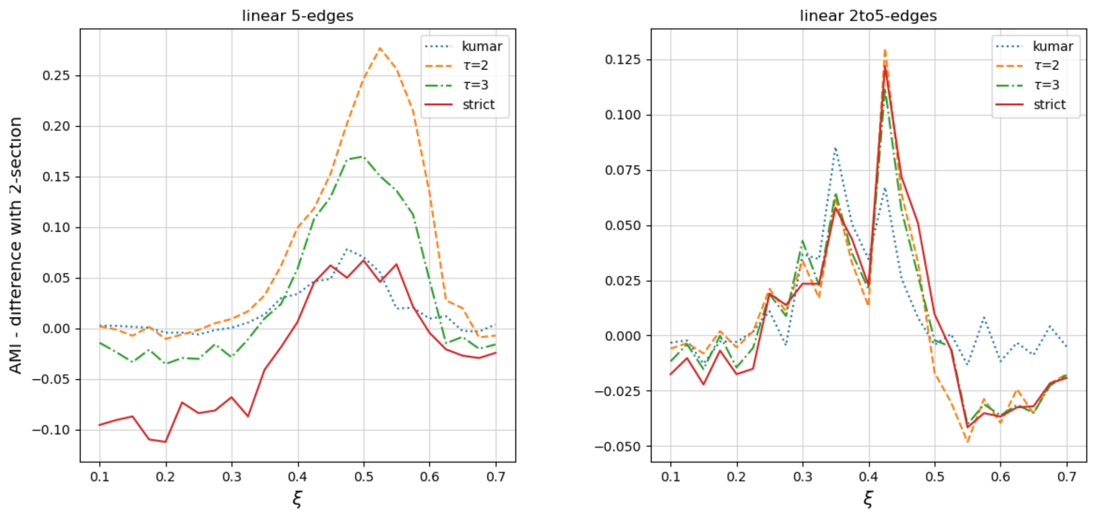


Figure 4: Results with **h–ABCD** hypergraphs with linear model for the community edge composition, (`linear_5` and `linear_2to5`) showing **AMI** difference between 2-section communities and the considered algorithms. Positive values indicate increase of **AMI** for a given algorithm.

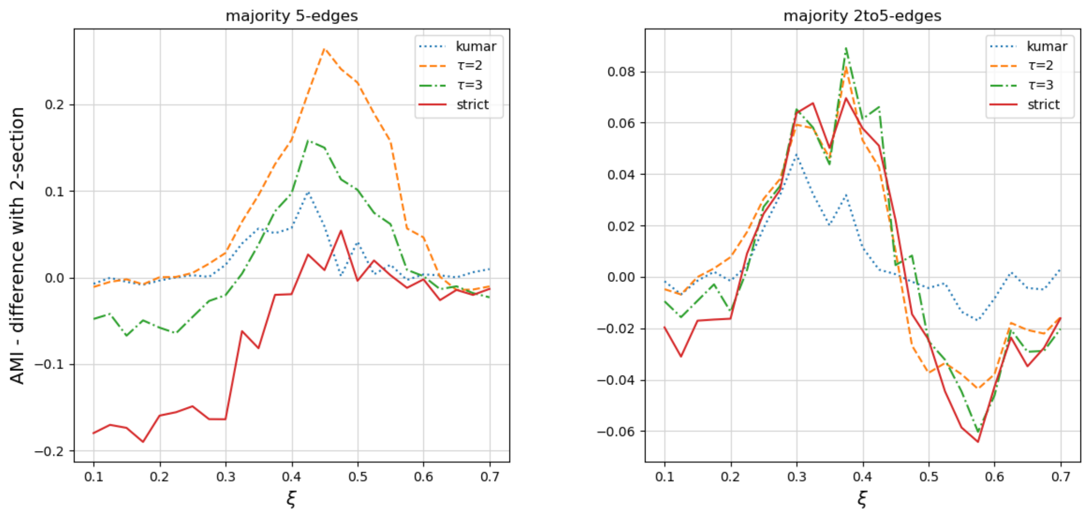


Figure 5: Results with **h–ABCD** hypergraphs with majority model for the community edge composition, (`majority_5` and `majority_2to5`) showing **AMI** difference between 2-section communities and the considered algorithms. Positive values indicate increase of **AMI** for a given algorithm.

Here are some general remarks from those experiments:

- The hypergraph specialized (**h-Louvain** and **Kumar**) algorithms give the most substantial benefits for moderately noisy hypergraphs. The reason is that for hypergraphs with very low level of noise (values of ξ close to zero) all algorithms produce similar results, as the community-finding problem is simple, and for very noisy graph (values of ξ close to one) the noise itself creates spurious communities that the algorithm start to recover (this effect has been previously studied and analytically analyzed for the **ABCD** graphs in [26]).

- Our **h–Louvain** algorithm with $\tau = 2$ and $\tau = 3$ outperforms **Kumar**'s algorithm most of the time. The exceptions are a few cases with large amount of noise in the hypergraph (values of ξ close to one).

- Strict **h–Louvain** modularity function ($\tau \to \infty$) may work poorly for hypergraphs that have many non-pure community hyperedges. For this reason, in the case of absence of a prior preference, users should follow the initial verification procedure described in Section 5.1 before using this parameterization. The reason is that for $\tau \to \infty$, all hyperedges of type (c, d) with $c < d$ are not counted as community hyperedges, which would loose potentially valuable information in cases when they are indeed informative.

5.3 More Challenging Case—Synthetic Hypergraphs with Localized Noise

In this experiment, we simulate an example in which the difficulty in recovering communities is due to the fact that several "noise" edges touch a small number of communities instead of being sprinkled over several communities. To that end, we generated a **h–ABCD** hypergraph with $n = 300$ nodes, degree exponent $\alpha = 2.5$ in the range $[5, 30]$, community size exponent $\beta = 1.5$ in the range $[40, 60]$, edges of size 5 with purity distribution for community hyperedges set to (0.7, 0.2, 0.1), i.e. 70% of them have 3 community nodes (type $(3, 5)$), 20% have 4 (type $(4, 5)$), and 10% have 5 (pure hyperedges, type $(5, 5)$). The overall noise is set to $\xi = 0.2$. To this hypergraph, we added 35 5-edges where the nodes are randomly sampled only from the two smallest communities. This simulates "localized" noise which should make the community detection more challenging.

First, simulating a real-life application of the procedure we proposed in Subsection 5.1, we look at the edge composition when running a 2-section clustering, which is reported in Table 3 along with the edge composition for the ground-truth communities. Running this quick experiment is indicative that smaller values for τ are likely to be a better choice than using the strict modularity version, since there are not that many "pure" edges.

		frequency	
d	c	ground-truth	Louvain
5	5	58	73
5	4	158	150
5	3	247	215
5	2	120	146
5	1	7	6

Table 3: Number of edges of each type for **h–ABCD** hypergraphs with 5-edges and localized noise added respectively for the ground-truth partition, and for a partition obtained by running **Louvain** on the weighted 2-section graph.

We did 100 runs for each choice of the τ-modularity for our **h–Louvain**: strict ($\tau \to \infty$) and $\tau \in \{0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4\}$. We also did 100 runs using the 2-section modularity with **Louvain**, and **Kumar**'s algorithm. The results are presented in Figure 6. From those results, we see that clustering the 2-section graph or using **Kumar**'s algorithm yield good results, but we can improve those results when optimizing the hypergraph τ-modularity when choosing $\tau \approx 2$. As expected from the preliminary EDA analysis we performed earlier, using small values for τ (close to zero) or large values for τ (including strict modularity, $\tau \to \infty$) are bad choices in this case.

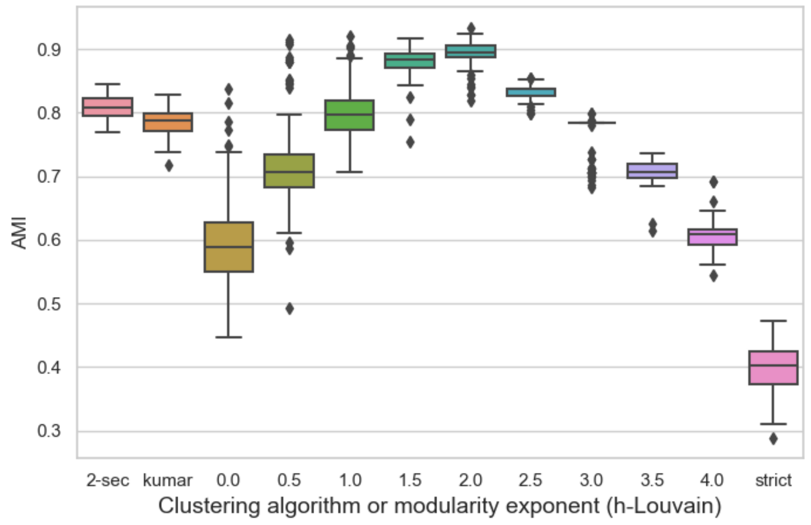


Figure 6: Results of 100 runs for several choices of hypergraph τ-modularity for **h–Louvain** (strict, and with $0 \le \tau \le 4$) as well as using **Louvain** on 2-section modularity and **Kumar**.

5.4 Real-world Hypergraphs

We consider two real-world hypergraphs the `primary-school` contact and the `cora`. We first run the EDA procedure, suggested in Subsection 5.1, on both graphs by looking at the hyperedge composition associated with the corresponding 2-section clusterings. The results are presented in Table 4. We can see that the `primary-school` hypergraph (left) has relatively more non-pure hyperedges than the `cora` hypergraph. This indicates that one should expect that for the `primary-school` case the optimal τ is smaller than the one for the `cora` hypergraph.

5.4.1 Contact Hypergraph

Let us first consider the `primary-school` contact hypergraph described in Section 3.2. The results are shown in Figure 7, where we compare 2-section (graph) clustering with **Louvain**, **Kumar**, and **AON** clustering as well as our **h–Louvain** using different values of τ for the modularity function. The **AMI** scores are averaged over 30 runs. The variance is negligible and is not shown. From this experiment, we see that one can get some improvement over 2-section and **Louvain** or **Kumar**'s algorithm when using small values for τ in our **h–Louvain**.

primary-school			
d	c	purity	frequency
2	1	50%	4051
2	2	100%	3697
3	3	100%	3385
3	2	67%	1054
3	1	33%	161
4	4	100%	240
4	3	75%	58
4	2	50%	47
5	4	80%	3
5	3	60%	3

cora			
d	c	purity	frequency
2	2	100%	512
3	3	100%	354
4	4	100%	212
5	5	100%	108
3	2	67%	85
4	3	75%	62
2	1	50%	60
5	4	80%	41
4	2	50%	32
5	3	60%	21

Table 4: The number of edges of each type (top-10 most frequent) for the `primary-school` contact (left) and `cora` (right) hypergraphs. The corresponding partitions were obtained by running **Louvain** on the weighted 2-section graph.

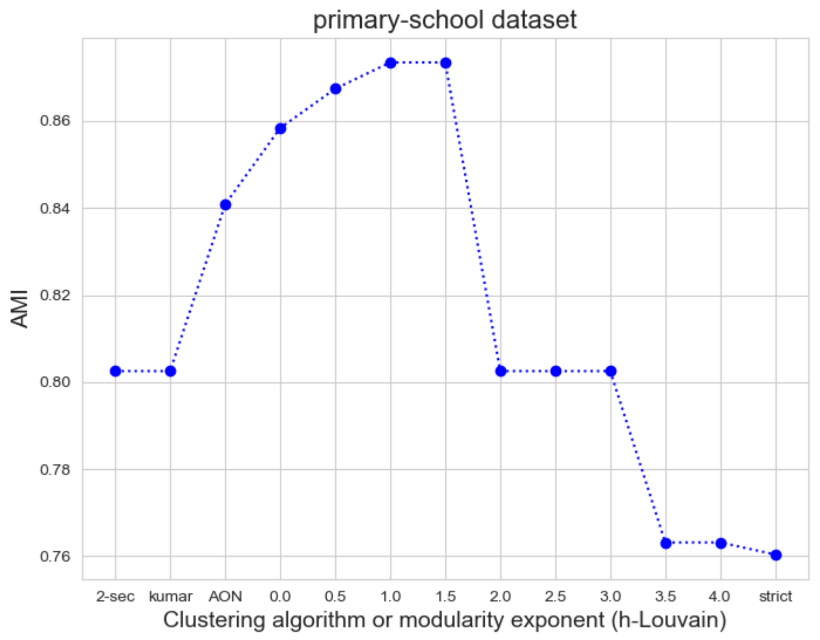


Figure 7: Results for several choices of hypergraph τ-modularity for **h–Louvain** (strict and with $0 \leq \tau \leq 4$) as well as using 2-section modularity and **Louvain**, **Kumar**'s, and **AON** algorithms.

5.4.2 Co-citation Hypergraphs

Next, we consider the `cora` co-citation hypergraph described in Subsection 3.2 in which nodes are publications which belong to 7 categories, and hyperedges represent co-citations. Since the graph has several small disconnected components, we restrict ourselves to the giant component which has

1,330 nodes and 1,503 hyperedges.

We ran each clustering algorithm 50 times, with the results reported in Figure 8. The results with **AON** were worse in this case (with **AMI** around 0.21, not reported in Figure 8). Instead, we report the results when starting from a partition returned by the 2-section graph clustering before running **AON**, which give better results. As expected from the EDA analysis comparing `primary-school` and `cora` hypergraphs, for the `cora` hypergraph, we get good results running **h–Louvain** with values of τ larger than for the `primary-school` hypergraph, slightly improving on the results with 2-section graph clustering with **Louvain**, **Kumar**'s algorithm, or **AON**.

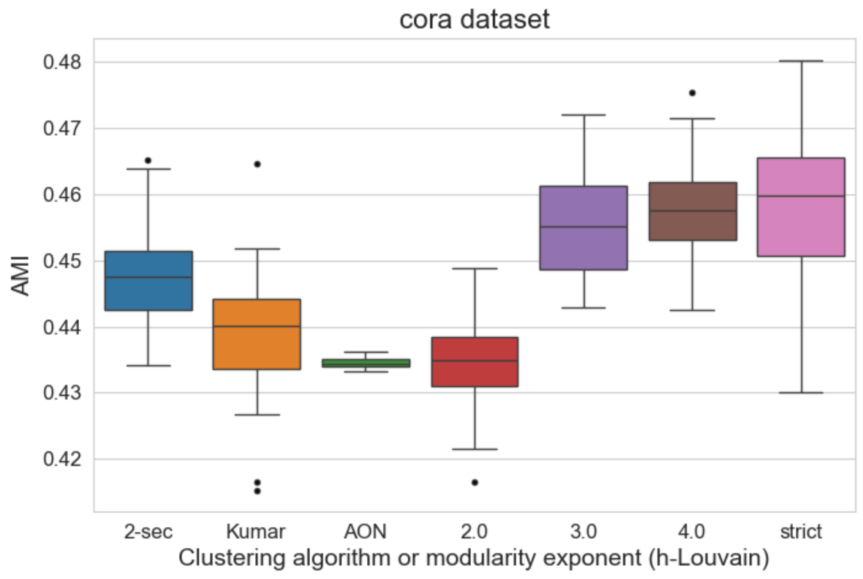


Figure 8: Clustering the `cora` co-citation hypergraph.

6 Conclusions

In this paper, we proposed a modification of the classical **Louvain** algorithm that allows us to optimize the hypergraph modularity, **h-Louvain**. Our approach is to optimize a weighted average of the 2-section graph modularity and the hypergraph modularity, with an increasing weight of hypergraph modularity component as the optimization process progresses. We presented both theoretical arguments as well as empirical evidence that the approach of increasing the weight of the hypergraph modularity component is efficient. Since there are several ways to update this weight, we developed a method allowing for automatic selection of hyperparameters of this process using Bayesian optimization. We have shown that the **h–Louvain** algorithm is competitive and, in particular, that it can outperform both **Louvain** on 2-section graph and **Kumar**'s algorithms in terms of recovering ground truth communities both for synthetic and real networks.

Additionally, let us mention about another important and interesting aspect. Since in **h–Louvain** the optimization process is stochastic by nature, the results of a single optimization pass can be easily improved by running many such optimizations in parallel. Therefore, an important extension to the algorithm is for allowing it to learn how to dynamically set the tuneable parameters when multiple optimization processes are executed.

References

[1] Ahn, K., Lee, K., Suh, C.: Hypergraph spectral clustering in the weighted stochastic block model. IEEE Journal of Selected Topics in Signal Processing **12**(5), 959–974 (2018)

[2] Barrett, J., Kamiński, B., Prałat, P., Théberge, F.: Self-similarity of communities of the abcd model. preprint, arXiv (2023)

[3] Barrett, J., Kamiński, B., Prałat, P., Théberge, F.: Self-similarity of communities of the abcd model. In: International Workshop on Algorithms and Models for the Web-Graph. pp. 17–31. Springer (2024)

[4] Battiston, F., Cencetti, G., Iacopini, I., Latora, V., Lucas, M., Patania, A., Young, J.G., Petri, G.: Networks beyond pairwise interactions: structure and dynamics. Physics Reports **874**, 1–92 (2020)

[5] Benson, A.R., Abebe, R., Schaub, M.T., Jadbabaie, A., Kleinberg, J.: Simplicial closure and higher-order link prediction. Proceedings of the National Academy of Sciences **115**(48), E11221–E11230 (2018)

[6] Benson, A.R., Gleich, D.F., Higham, D.J.: Higher-order network analysis takes off, fueled by classical ideas and new data. arXiv preprint arXiv:2103.05031 (2021)

[7] Benson, A.R., Gleich, D.F., Leskovec, J.: Tensor spectral clustering for partitioning higher-order network structures. In: Proceedings of the 2015 SIAM International Conference on Data Mining. pp. 118–126. SIAM (2015)

[8] Benson, A.R., Gleich, D.F., Leskovec, J.: Higher-order organization of complex networks. Science **353**(6295), 163–166 (2016)

[9] Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment **2008**(10), P10008 (2008)

[10] Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., Wagner, D.: On modularity clustering. IEEE transactions on knowledge and data engineering **20**(2), 172–188 (2007)

[11] Chien, I., Lin, C.Y., Wang, I.H.: Community detection in hypergraphs: Optimal statistical limit and efficient algorithms. In: International Conference on Artificial Intelligence and Statistics. pp. 871–879. PMLR (2018)

[12] Chodrow, P.S., Veldt, N., Benson, A.R.: Generative hypergraph clustering: From blockmodels to modularity. Science Advances **7**(28), eabh1303 (2021)

[13] Chung Graham, F., Lu, L.: Complex graphs and networks. No. 107, American Mathematical Soc. (2006)

[14] Clauset, A., Newman, M.E., Moore, C.: Finding community structure in very large networks. Physical review E **70**(6), 066111 (2004)

[15] Contreras-Aso, G., Criado, R., Vera de Salas, G., Yang, J.: Detecting communities in higher-order networks by using their derivative graphs. Chaos, Solitons & Fractals **177**, 114200 (2023)

[16] Ding, K., Wang, J., Li, J., Li, D., Liu, H.: Be more with less: Hypergraph attention networks for inductive text classification. arXiv preprint arXiv:2011.00387 (2020)

[17] Easley, D., Kleinberg, J.: Networks, crowds, and markets: Reasoning about a highly connected world. Cambridge university press (2010)

[18] Feng, S., Heath, E., Jefferson, B., Joslyn, C., Kvinge, H., Mitchell, H.D., Praggastis, B., Eisfeld, A.J., Sims, A.C., Thackray, L.B., et al.: Hypergraph models of biological networks to identify genes critical to pathogenic viral response. BMC bioinformatics **22**(1), 287 (2021)

[19] Fortunato, S.: Community detection in graphs. Physics reports **486**(3-5), 75–174 (2010)

[20] Fortunato, S., Barthelemy, M.: Resolution limit in community detection. Proceedings of the national academy of sciences **104**(1), 36–41 (2007)

[21] Frazier, P.I.: A tutorial on bayesian optimization. arXiv preprint arXiv:1807.02811 (2018)

[22] Grzesiak-Kopeć, K., Oramus, P., Ogorzałek, M.: Hypergraphs and extremal optimization in 3d integrated circuit design automation. Advanced Engineering Informatics **33**, 491–501 (2017)

[23] Jackson, M.O.: Social and economic networks. Princeton university press (2010)

[24] Juul, J.L., Benson, A.R., Kleinberg, J.: Hypergraph patterns and collaboration structure. arXiv preprint arXiv:2210.02163 (2022)

[25] Kamiński, B., Misiorek, P., Prałat, P., Théberge, F.: Modularity based community detection in hypergraphs. In: International Workshop on Algorithms and Models for the Web-Graph. pp. 52–67. Springer (2023)

[26] Kamiński, B., Pankratz, B., Prałat, P., Théberge, F.: Modularity of the abcd random graph model with community structure. Journal of Complex Networks **10**(6), cnac050 (2022)

[27] Kamiński, B., Poulin, V., Prałat, P., Szufel, P., Théberge, F.: Clustering via hypergraph modularity. PloS one **14**(11), e0224307 (2019)

[28] Kamiński, B., Prałat, P., Théberge, F.: Community detection algorithm using hypergraph modularity. In: International Conference on Complex Networks and Their Applications. pp. 152–163. Springer (2020)

[29] Kamiński, B., Prałat, P., Théberge, F.: Artificial benchmark for community detection (abcd)—fast random graph model with community structure. Network Science pp. 1–26 (2021)

[30] Kamiński, B., Prałat, P., Théberge, F.: Mining Complex Networks. Chapman and Hall/CRC (2021)

[31] Kamiński, B., Prałat, P., Théberge, F.: Artificial benchmark for community detection with outliers (abcd+o). Applied Network Science **8**(1), 25 (2023)

[32] Kamiński, B., Prałat, P., Théberge, F.: Hypergraph artificial benchmark for community detection (h–abcd). Journal of Complex Networks **11**(4), cnad028 (2023)

[33] Kamiński, B., Olczak, T., Pankratz, B., Prałat, P., Théberge, F.: Properties and performance of the abcde random graph model with community structure. Big Data Research **30**, 100348 (2022)

[34] Kumar, T., Vaidyanathan, S., Ananthapadmanabhan, H., Parthasarathy, S., Ravindran, B.: Hypergraph clustering by iteratively reweighted modularity maximization. Applied Network Science **5**(52) (2020)

[35] Kumar, T., Vaidyanathan, S., Ananthapadmanabhan, H., Parthasarathy, S., Ravindran, B.: A new measure of modularity in hypergraphs: Theoretical insights and implications for effective clustering. In: Cherifi, H., Gaito, S., Mendes, J.F., Moro, E., Rocha, L.M. (eds.) Complex Networks and Their Applications VIII. pp. 286–297. Springer International Publishing, Cham (2020)

[36] Lambiotte, R., Rosvall, M., Scholtes, I.: Understanding complex systems: From networks to optimal higher-order models. arXiv preprint arXiv:1806.05977 (2018)

[37] Lancichinetti, A., Fortunato, S.: Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. Physical Review E **80**(1), 016118 (2009)

[38] Lancichinetti, A., Fortunato, S.: Limits of modularity maximization in community detection. Physical review E **84**(6), 066122 (2011)

[39] Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Physical review E **78**(4), 046110 (2008)

[40] Lee, G., Bu, F., Eliassi-Rad, T., Shin, K.: A survey on hypergraph mining: Patterns, tools, and generators. arXiv preprint arXiv:2401.08878 (2024)

[41] Lee, G., Choe, M., Shin, K.: How do hyperedges overlap in real-world hypergraphs?-patterns, measures, and generators. In: Proceedings of the Web Conference 2021. pp. 3396–3407 (2021)

[42] Liao, X., Xu, Y., Ling, H.: Hypergraph neural networks for hypergraph matching. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 1266–1275 (2021)

[43] Mastrandrea, R., Fournet, J., Barrat, A.: Contact patterns in a high school: A comparison between data collected using wearable sensors, contact diaries and friendship surveys. PLoS ONE (2015)

[44] Matwin, S., Milios, A., Prałat, P., Soares, A., Théberge, F.: Generative Methods for Social Media Analysis. Springer Nature (2023)

[45] Newman, M.: Networks. Oxford university press (2018)

[46] Newman, M.E.: Fast algorithm for detecting community structure in networks. Physical review E **69**(6), 066133 (2004)

[47] Newman, M.E.: Modularity and community structure in networks. Proceedings of the national academy of sciences **103**(23), 8577–8582 (2006)

[48] Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Physical review E **69**(2), 026113 (2004)

[49] Nogueira, F.: Bayesian Optimization: Open source constrained global optimization tool for Python (2014–), `https://github.com/bayesian-optimization/BayesianOptimization`

[50] Peel, L., Larremore, D.B., Clauset, A.: The ground truth about metadata and community detection in networks. Science advances **3**(5), e1602548 (2017)

[51] Stehlé, J., et al.: High-resolution measurements of face-to-face contact patterns in a primary school. PLoS ONE (2011)

[52] Tian, H., Zafarani, R.: Higher-order networks representation and learning: A survey. arXiv preprint arXiv:2402.19414 (2024)

[53] Traag, V.A., Waltman, L., Van Eck, N.J.: From louvain to leiden: guaranteeing well-connected communities. Scientific reports **9**(1), 5233 (2019)

[54] Xia, X., Yin, H., Yu, J., Wang, Q., Cui, L., Zhang, X.: Self-supervised hypergraph convolutional networks for session-based recommendation. In: Proceedings of the AAAI conference on artificial intelligence. vol. 35, pp. 4503–4511 (2021)

[55] Yadati, N., Nimishakavi, M., Yadav, P., Nitin, V., Louis, A., Talukdar, P.: Hypergcn: A new method for training graph convolutional networks on hypergraphs. In: Advances in Neural Information Processing Systems (NeurIPS) 32, pp. 1509–1520. Curran Associates, Inc. (2019)

[56] Yi, S., Lee, D.S.: Structure of international trade hypergraphs. Journal of Statistical Mechanics: Theory and Experiment **2022**(10), 103402 (2022)

[57] Yin, H., Benson, A.R., Leskovec, J.: Higher-order clustering in networks. Physical Review E **97**(5), 052306 (2018)

[58] Yin, H., Benson, A.R., Leskovec, J., Gleich, D.F.: Local higher-order graph clustering. In: Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 555–564 (2017)

A Appendix—Pseudo-code of the h-Louvain Algorithm

Algorithm 1 h-Louvain(H, Γ)

```
Input: $H = (V, E)$ – input hypergraph; $\Gamma$ – policy to control $\alpha \in [0, 1]$ defined using parameters $p_b$ and $p_c$
Output: $\mathbf{A}$ – partition of $V$; $q_H(\mathbf{A})$ - hypergraph modularity
 1: Initialize: Build $G = (V, E_G)$ (2-section), and set partition $\mathbf{A}$ with all vertices $v \in V$ in their own cluster
 2: modified ← True
 3: $\alpha \leftarrow 0$
 4: $n \leftarrow |V|$
 5: while modified do
 6:     modified ← False
 7:     improved ← True
 8:     while improved do
 9:         improved ← False
10:         randomize the order of vertices in $V$
11:         for $v \in V$ do
12:             bestCommunity ← $A_i$ (current community of $v$)
13:             bestDelta ← 0
14:             for all neighbouring communities $A_j$ of $v$ do
15:                 deltaModularity ← $(1 - \alpha)\Delta q_G(A') + \alpha \Delta q_H(A')$ ($A'$: $v$ moved from $A_i$ to $A_j$)
16:                 if deltaModularity > bestDelta then
17:                     bestDelta ← deltaModularity
18:                     bestCommunity ← $A_j$
19:                     improved ← True
20:                 end if
21:             end for
22:             if improved then
23:                 change $\mathbf{A}$ by moving $v$ to bestCommunity
24:                 modified ← True
25:                 $\alpha \leftarrow$ UpdateAlpha($\mathbf{A}$,$n$,$\alpha$,$\Gamma$)
26:             end if
27:         end for
28:     end while
29:     if modified then
30:         update $H = (V, E)$ (merge current communities into supernodes and update edges)
31:     else if $\alpha < 1$ then
32:         $\alpha \leftarrow 1$
33:         revert the last merging communities step
34:         modified ← True
35:     end if
36: end while
37: return $\mathbf{A}$, $q_H(\mathbf{A})$
```

Algorithm 2 UpdateAlpha($\mathbf{A}$, n, α, Γ)

```
Input: current partition $\mathbf{A}$, total number of nodes $n$, previous value of $\alpha$, policy $\Gamma$ to control $\alpha \in [0,1]$
    defined using parameters $p_b$ and $p_c$
Output: new value of $\alpha$
1: if $\alpha < 1$ then
2:     $|\mathbf{A}| \leftarrow$ number of communities in current partition $\mathbf{A}$
3:     $j \leftarrow \arg\max_{k \in \mathbb{N}}(k : \frac{|\mathbf{A}|}{n} \leq p_c^k)$
4:     $\alpha \leftarrow 1 - (1 - p_b)^j$
5:     return $\alpha$
6: else
7:     return 1
8: end if
```