# Modularity Based Community Detection in Hypergraphs

Bogumił Kamiński[1], Paweł Misiorek[2], Paweł Prałat[3], and François Théberge[4]

[1] Decision Analysis and Support Unit, SGH Warsaw School of Economics, Warsaw, Poland bogumil.kaminski@sgh.waw.pl
[2] Institute of Computer Sciences, Poznan University of Technology, Poznan, Poland pawel.misiorek@put.poznan.pl
[3] Department of Mathematics, Toronto Metropolitan University, Toronto, ON, Canada pralat@torontomu.ca
[4] Tutte Institute for Mathematics and Computing, Ottawa, ON, Canada theberge@ieee.org

**Abstract.** In this paper, we make a significant step toward designing a scalable community detection algorithm using hypergraph modularity function. The main obstacle with adjusting the initial stage of the classical **Louvain** algorithm is dealt via carefully adjusted linear combination of the graph modularity function of the corresponding two-section graph and the desired hypergraph modularity function. It remains to properly tune the algorithm and design a mechanism to adjust the weights in the modularity function (in an unsupervised way), depending on how often nodes in one community share hyperedges with nodes from other communities. It will be done in the journal version of this paper.

**Keywords:** Community Detection Algorithm · Hypergraphs · Modularity Function.

## 1 Introduction

Many networks that are currently modelled as graphs would be more accurately modelled as hypergraphs. This includes the collaboration network in which nodes correspond to researchers and hyperedges correspond to papers that consist of nodes associated with researchers that co-author a given paper.

After many years of intense research using graph theory in modelling and mining complex networks [13, 15, 21, 31], hypergraphs start gaining considerable traction [2–4, 6]. Standard but important questions in network science are 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 [30]? Or how the existing patters in a hypergraph affect the formation of new hyperedges [16]?

In this paper we concentrate on the classical problem of *community detection* in networks that can be represented using hypergraphs [1, 5, 8, 9, 18, 19, 25, 26, 34, 35]. 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 often create the 2-section graph of a hypergraph of interest (that is, replace each hyperedge with a clique) and apply classical tools designed for graphs. After moving to the 2-section graph, one clearly loses some information about hyperedges of size greater than two and so there is a common belief that one can do better by using the knowledge of the original hypergraph.

As mentioned earlier, there are some recent attempts to deal with hypergraphs in the context of clustering. For example, Kumar et al. [25, 26] 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). Moreover, in [18, 19] a number of extensions of the classic null model for graphs are proposed that can potentially be used by true hypergraph algorithms.

Unfortunately, there are many ways such extensions 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 [16]. In this paper, we assume that the user has a knowledge which of the null models should be chosen to analyze a given hypergraph at hand and, as a result, wants to use the appropriate modularity function to identify communities in a hypergraph. Eventually, our clustering algorithm will be able to automatically decide which extension should be used but details will be provided in the journal version of this paper.

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. One of the most popular and efficient heuristic methods for modularity optimization for graphs is the **Louvain** algorithm. 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, then the algorithm may get stuck almost immediately. 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 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). Next, we discuss the classical **Louvain** algorithm and explain why it is difficult to adjust it to directly optimize hypergraph modularity. Following this, we describe our algorithm that is considering a linear combination of the 2-section graph modularity and the hypergraph modularity as objective function, and explain its implementation challenges (Section 3). Then, we present the results of numerical experiments of using the proposed algorithm on synthetic hypergraphs (Section 4). The paper is concluded with a summary of outlooks for further research in this area that will be addressed in the journal version of this paper (Section 5).

## 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 $d \in \mathrm{N}$. 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 [33]. Despite some known issues with this function such as the "resolution limit" reported in [14], many popular algorithms for partitioning nodes of large graphs use it [11, 28, 32] and perform very well. 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 [10] which generates 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:

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

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 by replacing edge counts with sums of 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 $H$, graph $H_{[2]}$, on the same set of nodes as $H$. For each hyperedge $e \in E$ with $|e| \geq 2$ and weight $w(e)$, $\binom{|e|}{2}$ edges are formed, each of them with weight of $w(e)/(|e| - 1)$. While there are other natural choices for the weights (such as the original weighting scheme $w(e)/\binom{|e|}{2}$ that preserves the total weight), this choice ensures that the degree distribution of the created graph matches the one of the original hypergraph $H$ [26, 25]. Moreover, let us also mention that it also nicely translates a natural random walk on $H$ into a random walk on the corresponding $H_{[2]}$. 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 edge weights are added together.

One of the approaches for finding communities in hypergraphs that practitioners use is to apply **Louvain** algorithm 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 for a hypergraph 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. 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 [18].

The hypergraph modularity function is controlled by *hyper-parameters* $w_{c,d} \in [0,1]$ ($d \geq 2$, $\lfloor d/2 \rfloor + 1 \leq c \leq d$). For a fixed set of hyper-parameters, we define

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

where

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

$e_H^{d,c}(A_i)$ 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.

Hyper-parameters $w_{c,d}$ give us a lot of flexibility and allow to value some edges more than others depending on their size and homogeneity. However, there are three natural hyper-parameters that one might consider, yielding three modularity functions to optimize:

- *strict modularity*: $w_{d,d} = 1$ and $w_{c,d} = 0$ for $\lfloor d/2 \rfloor + 1 \leq c < d$,
- *linear modularity*: $w_{c,d} = c/d$ for $\lfloor d/2 \rfloor + 1 \leq c \leq d$,
- *majority modularity*: $w_{c,d} = 1$ for $\lfloor d/2 \rfloor + 1 \leq c \leq d$.

## 3 Hypergraph Modularity Optimization Algorithm

### 3.1 Louvain Algorithm

Let us start by introducing one of the mostly used unsupervised algorithms for detecting communities in graphs, namely, the **Louvain** algorithm [7]. It is a hierarchical clustering algorithm that tries to optimize the modularity function we described in Section 2.

In this algorithm, small communities are first found by optimizing modularity locally on all nodes. Then, each small community is grouped into one node and the original step is repeated on a smaller graph. The process stops when no improvement on the modularity function can be further achieved.

One pass of the algorithm consists of two phases that are repeated iteratively. Initially, 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. If no increase is possible after considering all nodes, 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 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.

### 3.2 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. This problem can be referred to as a *lift off from the ground* problem.

The above, extreme, situation is not the only problem one should be aware of. Consider this time 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.

### 3.3 Our Approach to Hypergraph Modularity Optimization: h-Louvain

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 $w(e)\binom{c}{2}/(|e| - 1)$ (in the variant of the 2-section where the degrees are preserved) or $w(e)\binom{c}{2}/\binom{|e|}{2} \approx w(e)(c/|e|)^2$ (if the total weight is 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 values of hyper-parameters 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. In short, it resolves the *lifting off from the ground* problem. If $\alpha$ is close to zero, then we concentrate mostly on the approximation part, while if $\alpha$ 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? The main goal of this paper is to answer these two questions.

Given the theoretical derivation we presented above, the following hypotheses (that will be verified in Section 4) can be formulated:

- the optimization process should be started with low values of the parameter $\alpha$ (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 (since for this value, we reduce the problem to optimizing the function we actually want to optimize);
- the algorithm should start increasing parameter $\alpha$ 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 $\alpha$ 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.

## 4 Results

### 4.1 Synthetic Hypergraph Model: h-ABCD

There are very few datasets with ground-truth identified and labelled. 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. Since we are at the initial stage of developing our algorithm, we concentrate on experiments on synthetic networks but real-world ones will be investigated in the journal version of this paper.

The situation for graphs is rather clear. The **LFR** (**L**ancichinetti, **F**ortunato, **R**adicchi) model [29, 27] 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**) [20] was recently introduced and implemented[5], including a fast implementation[6] that uses multiple threads (**ABCDe**) [24]. 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 extremes: pure (disjoint) communities and random graph with no community structure. Moreover, it is easier to analyze theoretically—for example, in [17] various theoretical asymptotic properties of the **ABCD** model are investigated including the modularity function,

[5] https://github.com/bkamins/ABCDGraphGenerator.jl/
[6] https://github.com/tolcz/ABCDeGraphGenerator.jl/

arguably, the most important graph property of networks in the context of community detection.

Situation for hypergraphs is not as clear as for graphs. There are not only very few real-world datasets available to use but also there are not so many synthetic hypergraphs one can use. 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 [23] resulting in **ABCD+o** model. Adjusting the model to hypergraphs is more complex but it was also done recently [22] resulting in **h-ABCD** model. We will use this model for our experiments.

### 4.2 Exhaustive Search for the Best Strategy

As discussed in Section 3, one hypothesis that was identified for our **h-Louvain** algorithm is that one should avoid decreasing the values of $\alpha$ in $q(\mathbf{A}, \alpha)$ (see equation (3)) as the algorithm progresses. In this first set of experiments, we consider **h-ABCD** synthetic hypergraphs on 1,000 nodes of degrees in the range $[5, 20]$ (with average around 8) and community sizes in the range $[10, 30]$ (with average around 18), where both distributions follow a power law. The hyperedges are of size between 2 and 5, inclusively, and the linear option for community hyperedges in the **h-ABCD** generator. The noise level is set at $\xi = 0.3$, the proportion of hyperedges which are sampled randomly from the set of all nodes, regardless of community memberships.

We ran an exhaustive search for every sequence of length 5 for the values of $\alpha$, namely $(\alpha_1, \ldots, \alpha_5)$ which are chosen from the set $\{0, 0.25, 0.5, 0.75, 1\}^5$. For each sequence, we run a test with 5 different random seeds, for a total of $5 \cdot 5^5 = 15{,}625$ distinct runs of the **h-Louvain** algorithm.

One key question is when the value of parameter $\alpha$ should be changed, that is, when to move from $\alpha_i$ to $\alpha_{i+1}$ for $1 \leq i \leq 4$. After running several empirical tests, we reached the conclusion that the best results are achieved when the change is made when the number of communities reaches $n/z^i$, where $n$ equals the initial number of nodes and $z \in \mathbb{R}$, $z > 1$, is a tuneable parameter. In our experiments, parameter $z$ is set experimentally to $z = 2.3$ but in the final version of the algorithm its value will be appropriately tuned and, in particular, it will depend on the hyperedge size distribution in a given hypergraph at hand. One instance of running the **h-Louvain** algorithm is shown in Figure 1, where we show the evolution of two quantities: the number of communities and the modularity functions. More examples can be found in the associated appendix available online[7].

In Figures 2–4, we show the results of the exhaustive search using, respectively, three different hypergraph modularity functions: strict, linear, and majority. In the top plots, we compare the resulting modularity values for monotonic (that is, non-decreasing) sequences of $\alpha$'s versus all remaining sequences, while in the bottom plots, we restrict the non-decreasing sequences to the ones where

[7] https://math.torontomu.ca/~pralat/research.html

$\alpha_3 < 1$ and $\alpha_5 = 1$, thus forcing sequences of non-decreasing values spanning a wider range. For example, this avoids the non-decreasing sequence in which $\alpha_i = 1$ for all $i$. Based on those figures, we see that the non-decreasing sequences generally yield better results, and forcing the extra conditions greatly reduces the variability of the results. We also notice that the gain in choosing such a strategy is more visible in the case of the strict modularity, and less so for the linear modularity. This is to be expected as the linear hypergraph modularity bears more similarity to the 2-section graph modularity function.

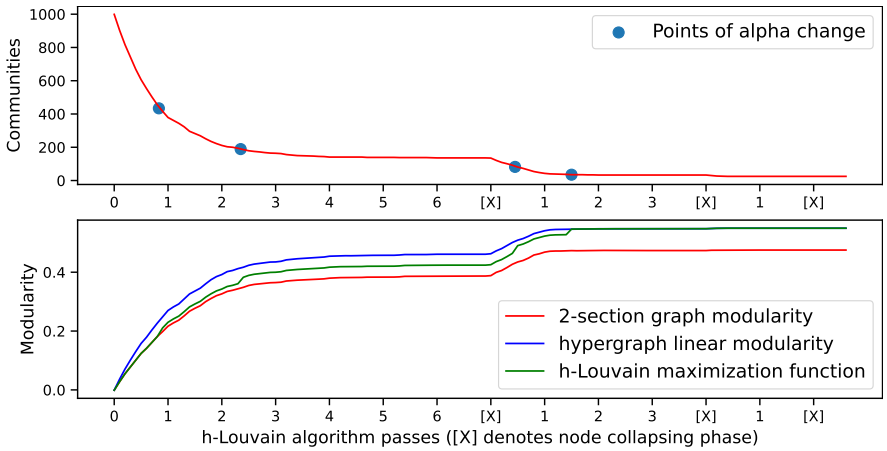


**Fig. 1.** Visualisation of the **h-Louvain** process for the sequence $(0.0, 0.25, 0.5, 0.75, 1)$ of $\alpha$ changes and maximization of the **linear** hypergraph modularity. The $x$-axis corresponds to the passes of the algorithm. For each pass, the iterations (based on checking all nodes in random order) of the modularity optimization phase are denoted by consecutive numbers, whereas the phase of node collapsing (community aggregation) is marked by [X].

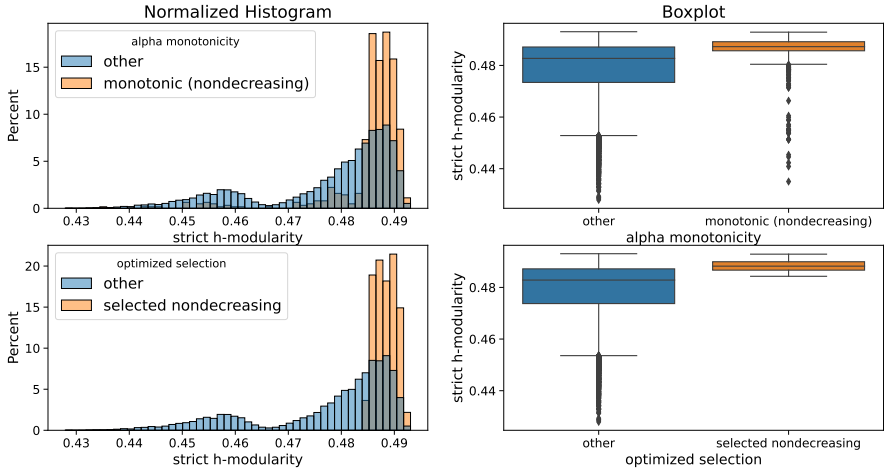


**Fig. 2.** Results of the exhaustive search for the case of **strict** hypergraph modularity.

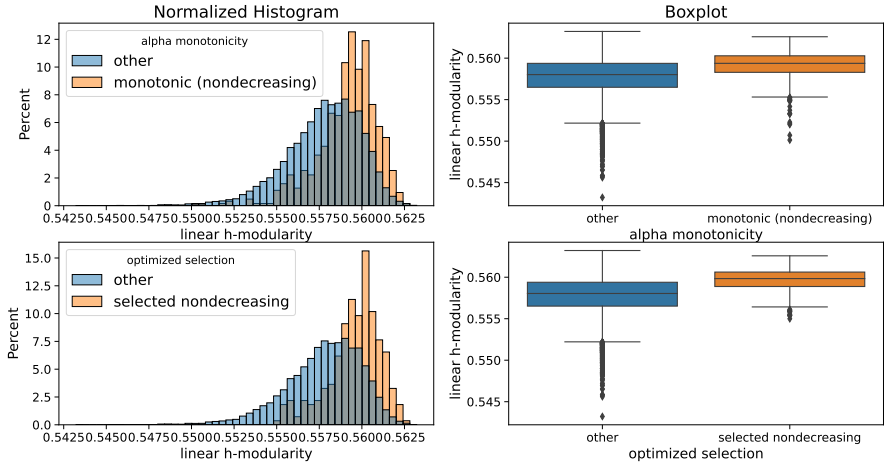


**Fig. 3.** Results of the exhaustive search for the case of **linear** hypergraph modularity.

### 4.3 Comparing Basic Policies for Different Modularity Functions

For the following experiments, we consider the **h-ABCD** model with the same set of parameters used in Section 4.2 and with three different ways to generate community edges: (i) strict, (ii) linear, and (iii) majority. In each case, we generated 10 different hypergraphs (starting with different random seeds).

We consider simple policies where we use some value $\alpha_1$ for the first pass of the **h-Louvain** algorithm, $\alpha_2$ for the second pass, and $\alpha_3$ for all subsequent passes. Note that in all runs, there were never more than 5 passes. Thus, for example, policy $(\alpha_1, \alpha_2, \alpha_3) = (0, 0, 0)$ amounts to optimizing the 2-section graph modularity during the entire process, while policy $(1, 1, 1)$ amounts to optimizing

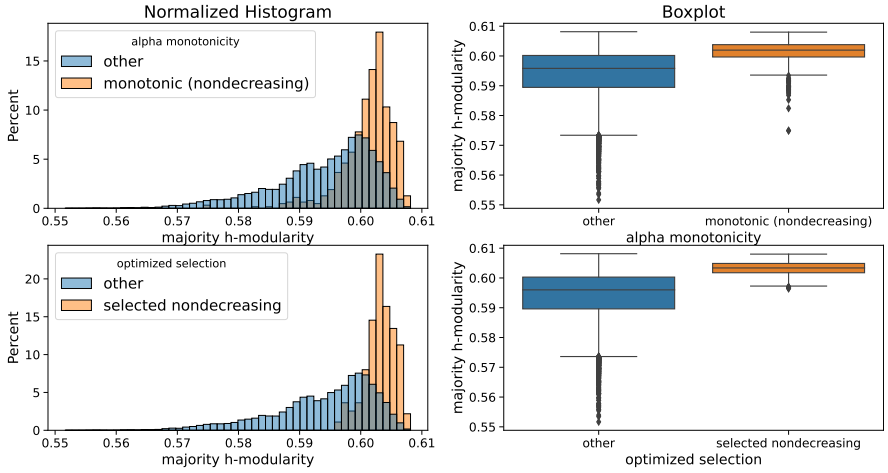


**Fig. 4.** Results of the exhaustive search for the case of **majority** hypergraph modularity.

the hypergraph modularity throughout. Each policy is tested 5 times with different random seeds for each of the 10 hypergraphs. We considered 19 different policies either with constant $\alpha_i$, or non-decreasing sequences with $\alpha_3 = 1$.

We use Friedman-Nemenyi statistical test described in detail in [12] to validate whether the performance difference between compared policies is statistically significant. We follow the same procedure for the three investigated cases, i.e., strict, linear, and majority modularity optimization. First, for every generated hypergraph, we rank the policies based on the average modularity obtained in 5 runs so that the policy with the highest score is ranked 1 while the one with the worst score is ranked 19. Then, the Friedman test based on the average ranks is used to analyze if the differences between all compared policies on multiple hypergraphs are statistically significant. For each investigated case, the calculated Friedman rank sum test statistics was greater than the critical value for the assumed level of confidence $\alpha = 0.05$, so the conclusion is that the null hypothesis that there is no difference between the policies could be rejected, and the post-hoc Nemenyi test can be conducted. The Nemenyi test aims to investigate the difference between each pair of individual policies. According to the test assumptions, the difference between a given pair of policies is regarded as statistically significant if it is bigger than the critical difference `CD` (for an assumed confidence level 0.05) calculated based on formulas presented in [12] (in our case equal to 2.03). We used the most popular way to visualize the Nemenyi test results proposed originally in [12]. The three diagrams presented in Figures 5–7 illustrate the ranked performances of the compared policies along with the critical difference `CD` when optimizing strict, linear, and majority modularity functions. The central axis presented in the diagrams is used to plot the average ranks of compared policies sorted in decreasing order (with the best policy at the

right). Moreover, in order to support the results' interpretation, the horizontal bolt lines are added below the main axis to connect the groups of policies that are not significantly different.

In general, when the strict modularity is optimized, it is important to start with low values of the parameter $\alpha$, while this is not as important when optimizing linear or majority modularity functions. Other general observations are that it is best to increase $\alpha_3$ to its maximum value of 1, and never to start with $\alpha_1 = 1$ right away.

Further results using the other **h-ABCD** hypergraphs as well as more numerical details can be found in the associated appendix available online[8]. The conclusions remain the same.

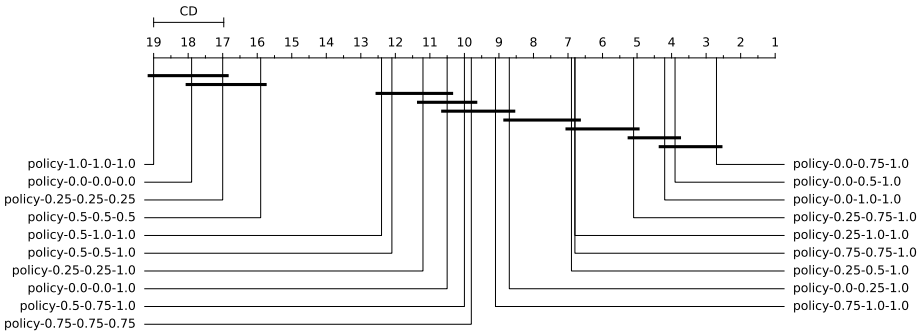


**Fig. 5.** Visualization of Friedman and Nemenyi test results for **strict modularity** with **linear h-ABCD**.

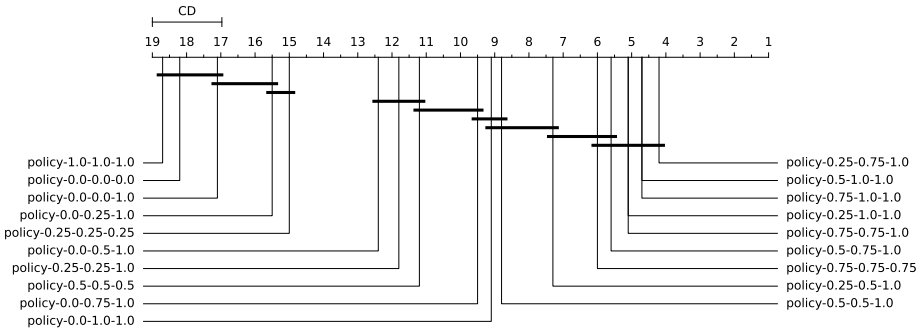


**Fig. 6.** Visualization of Friedman and Nemenyi test results for **linear modularity** with **linear h-ABCD**.

[8] https://math.torontomu.ca/~pralat/research.html

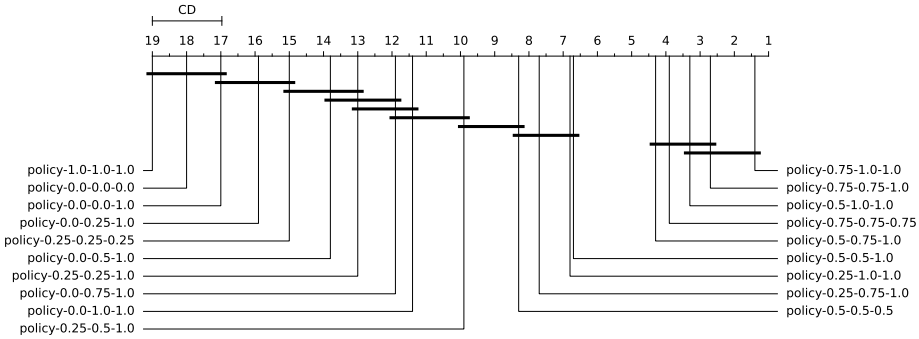


**Fig. 7.** Visualization of Friedman and Nemenyi test results for **majority modularity** with **linear h-ABCD**.

## 5 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 have presented both theoretical arguments as well as empirical evidence that the approach of increasing the weight of hypergraph modularity component improves the results of the optimization process in comparison to trying to optimize hypergraph modularity directly using the **Louvain** algorithm, or using a different weight change policy.

In this proceeding version of the paper, we concentrate on presenting the main ideas behind the proposed algorithm and the results showing that, indeed, it gives performance improvements. However, the key element of the algorithm is the schedule how the weight of the hypergraph modularity component should be changed (that is, what value should be taken and when to make the change). In the initial experiments we selected parameter $\alpha$ (the weight) from a fixed set of possible values and change it in discrete time steps governed by parameter $z$ (fixed in our experiments). In the extended, journal version of this paper, more sophisticated algorithm will be presented and experimented with that will perform auto-tuning of these parameters based on various hypergraph characteristics and the type of hypergraph modularity that is optimized.

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.

One final extension that we plan to do is to allow for auto-discovery of which version of modularity function best fits the analyzed hypergraph. In the current version of the algorithm the user has to specify this information explicitly.

## 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. 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)
3. 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)
4. 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)
5. 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)
6. Benson, A.R., Gleich, D.F., Leskovec, J.: Higher-order organization of complex networks. Science **353**(6295), 163–166 (2016)
7. 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)
8. 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)
9. Chodrow, P.S., Veldt, N., Benson, A.R.: Generative hypergraph clustering: From blockmodels to modularity. Science Advances **7**(28), eabh1303 (2021)
10. Chung Graham, F., Lu, L.: Complex graphs and networks. No. 107, American Mathematical Soc. (2006)
11. Clauset, A., Newman, M.E., Moore, C.: Finding community structure in very large networks. Physical review E **70**(6), 066111 (2004)
12. Demšar, J.: Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. **7**, 1–30 (2006)
13. Easley, D., Kleinberg, J.: Networks, crowds, and markets: Reasoning about a highly connected world. Cambridge university press (2010)
14. Fortunato, S., Barthelemy, M.: Resolution limit in community detection. Proceedings of the national academy of sciences **104**(1), 36–41 (2007)
15. Jackson, M.O.: Social and economic networks. Princeton university press (2010)
16. Juul, J.L., Benson, A.R., Kleinberg, J.: Hypergraph patterns and collaboration structure. arXiv preprint arXiv:2210.02163 (2022)
17. Kamiński, B., Pankratz, B., Prałat, P., Théberge, F.: Modularity of the abcd random graph model with community structure. preprint arXiv:2203.01480 (2022)
18. Kamiński, B., Poulin, V., Prałat, P., Szufel, P., Théberge, F.: Clustering via hypergraph modularity. PloS one **14**(11), e0224307 (2019)
19. 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)

20. 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)
21. Kamiński, B., Prałat, P., Théberge, F.: Mining Complex Networks. Chapman and Hall/CRC (2021)
22. Kamiński, B., Prałat, P., Théberge, F.: Hypergraph artificial benchmark for community detection (h-abcd). arXiv preprint arXiv:2210.15009 (2022)
23. Kamiński, B., Prałat, P., Théberge, F.: Outliers in the abcd random graph model with community structure (abcd+o). In: Proceedings of the 11th International Conference on Complex Networks and their Applications (2022 (in press))
24. 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)
25. Kumar, T., Vaidyanathan, S., Ananthapadmanabhan, H., Parthasarathy, S., Ravindran, B.: Hypergraph clustering by iteratively reweighted modularity maximization. Applied Network Science **5**(52) (2020)
26. 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)
27. 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)
28. Lancichinetti, A., Fortunato, S.: Limits of modularity maximization in community detection. Physical review E **84**(6), 066122 (2011)
29. Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Physical review E **78**(4), 046110 (2008)
30. 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)
31. Newman, M.: Networks. Oxford university press (2018)
32. Newman, M.E.: Fast algorithm for detecting community structure in networks. Physical review E **69**(6), 066133 (2004)
33. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Physical review E **69**(2), 026113 (2004)
34. Yin, H., Benson, A.R., Leskovec, J.: Higher-order clustering in networks. Physical Review E **97**(5), 052306 (2018)
35. 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 Appending – Additional Experiments – Not to be Included in the Proceeding Paper

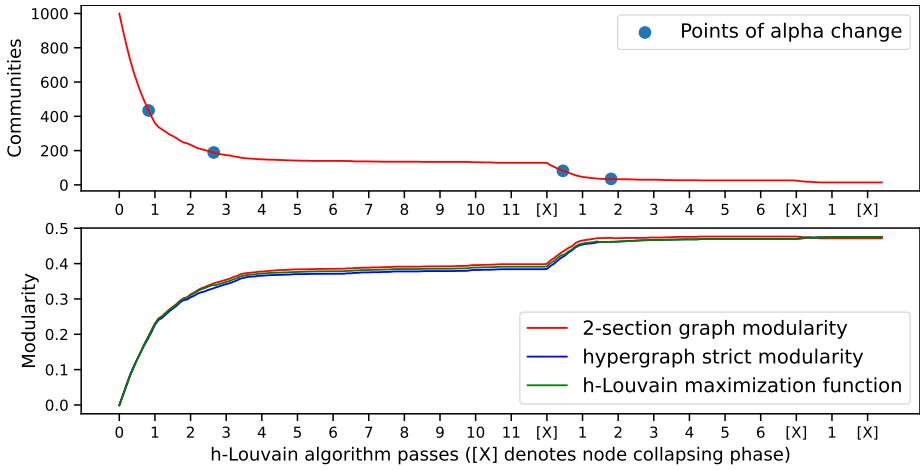


**Fig. 8.** Visualisation of the Louvain process for the cases of $\alpha$ changes correspoding to sequence $(0.0, 0.25, 0.5, 0.75, 1)$ and maximization of **strict** hypergraph modularity.

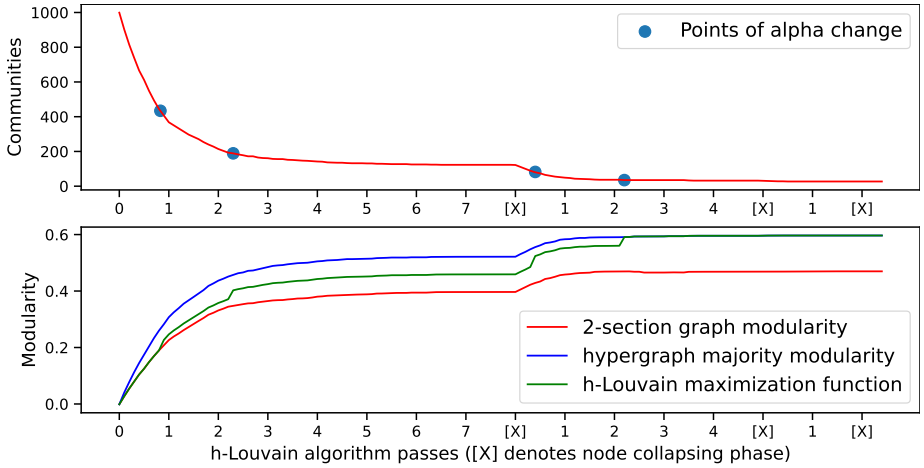


**Fig. 9.** Visualisation of the Louvain process for the cases of $\alpha$ changes correspoding to sequence $(0.0, 0.25, 0.5, 0.75, 1)$ and maximization of **majority** hypergraph modularity.

**Table 1.** Detailed results of experiments from Sec. 4.3 for the case of **strict modularity** with **strict h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.0-1.0-1.0 | 0.550054 | 0.000843 |
| policy-0.0-0.5-1.0 | 0.549909 | 0.000458 |
| policy-0.0-0.25-1.0 | 0.549798 | 0.001588 |
| policy-0.0-0.75-1.0 | 0.549738 | 0.000780 |
| policy-0.25-0.5-1.0 | 0.549566 | 0.001710 |
| policy-0.25-0.75-1.0 | 0.549514 | 0.001933 |
| policy-0.25-0.25-1.0 | 0.549067 | 0.001750 |
| policy-0.0-0.0-1.0 | 0.548944 | 0.001170 |
| policy-0.25-1.0-1.0 | 0.548707 | 0.001583 |
| policy-0.75-0.75-1.0 | 0.547741 | 0.000542 |
| policy-0.5-1.0-1.0 | 0.547677 | 0.001303 |
| policy-0.5-0.5-1.0 | 0.547676 | 0.000701 |
| policy-0.75-1.0-1.0 | 0.547637 | 0.000635 |
| policy-0.75-0.75-0.75 | 0.547615 | 0.000640 |
| policy-0.5-0.75-1.0 | 0.547405 | 0.000906 |
| policy-0.25-0.25-0.25 | 0.546696 | 0.001704 |
| policy-0.5-0.5-0.5 | 0.545957 | 0.001298 |
| policy-0.0-0.0-0.0 | 0.544068 | 0.000984 |
| policy-1.0-1.0-1.0 | 0.473480 | 0.005054 |

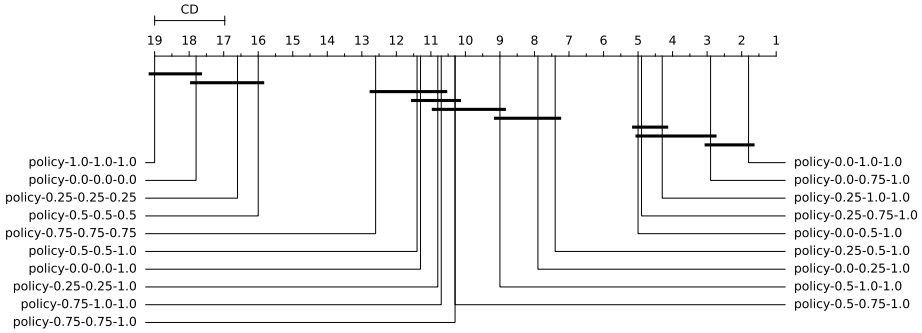


**Fig. 10.** Visualization of Friedman and Nemenyi test results for **strict modularity** with **strict h-ABCD**.

**Table 2.** Detailed results of experiments from Sec. 4.3 for the case of **strict modularity** with **linear h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.0-0.75-1.0 | 0.481523 | 0.001433 |
| policy-0.5-0.75-1.0 | 0.481497 | 0.001208 |
| policy-0.0-0.5-1.0 | 0.481055 | 0.001018 |
| policy-0.0-0.0-1.0 | 0.480840 | 0.001109 |
| policy-0.75-0.75-1.0 | 0.480819 | 0.000937 |
| policy-0.25-0.5-1.0 | 0.480813 | 0.000532 |
| policy-0.0-1.0-1.0 | 0.480773 | 0.001592 |
| policy-0.25-0.75-1.0 | 0.480625 | 0.000862 |
| policy-0.5-1.0-1.0 | 0.480434 | 0.001432 |
| policy-0.25-1.0-1.0 | 0.480343 | 0.001297 |
| policy-0.5-0.5-1.0 | 0.480257 | 0.001892 |
| policy-0.75-1.0-1.0 | 0.480220 | 0.001290 |
| policy-0.75-0.75-0.75 | 0.480127 | 0.000894 |
| policy-0.0-0.25-1.0 | 0.479862 | 0.001072 |
| policy-0.25-0.25-1.0 | 0.479861 | 0.001039 |
| policy-0.5-0.5-0.5 | 0.478779 | 0.001717 |
| policy-0.25-0.25-0.25 | 0.476036 | 0.001234 |
| policy-0.0-0.0-0.0 | 0.474350 | 0.001066 |
| policy-1.0-1.0-1.0 | 0.430973 | 0.003849 |

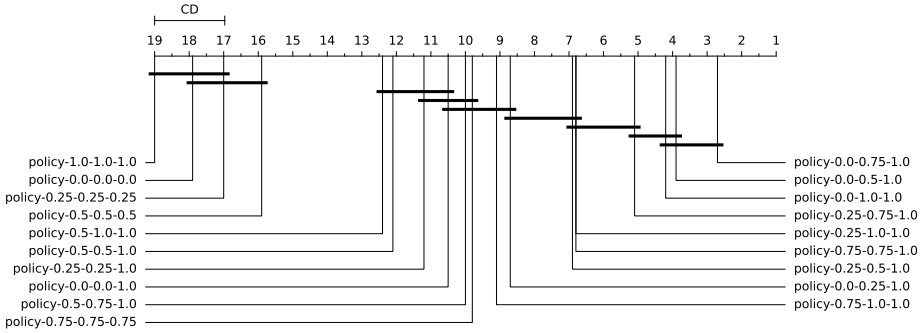


**Fig. 11.** Visualization of Friedman and Nemenyi test results for **strict modularity** with **linear h-ABCD**.

**Table 3.** Detailed results of experiments from Sec. 4.3 for the case of **strict modularity** with **majority h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.0-0.75-1.0 | 0.462403 | 0.001284 |
| policy-0.0-1.0-1.0 | 0.461853 | 0.001658 |
| policy-0.0-0.5-1.0 | 0.461712 | 0.002091 |
| policy-0.25-1.0-1.0 | 0.461485 | 0.001839 |
| policy-0.25-0.5-1.0 | 0.461008 | 0.001429 |
| policy-0.0-0.25-1.0 | 0.460910 | 0.000614 |
| policy-0.25-0.75-1.0 | 0.460675 | 0.001485 |
| policy-0.0-0.0-1.0 | 0.460318 | 0.001109 |
| policy-0.5-0.5-1.0 | 0.459948 | 0.001773 |
| policy-0.25-0.25-1.0 | 0.459884 | 0.001718 |
| policy-0.5-0.75-1.0 | 0.459498 | 0.000945 |
| policy-0.5-1.0-1.0 | 0.459138 | 0.000507 |
| policy-0.75-0.75-1.0 | 0.458213 | 0.000909 |
| policy-0.5-0.5-0.5 | 0.457937 | 0.001059 |
| policy-0.75-1.0-1.0 | 0.457812 | 0.001357 |
| policy-0.75-0.75-0.75 | 0.457552 | 0.001113 |
| policy-0.25-0.25-0.25 | 0.456586 | 0.001603 |
| policy-0.0-0.0-0.0 | 0.454287 | 0.002027 |
| policy-1.0-1.0-1.0 | 0.423735 | 0.002701 |

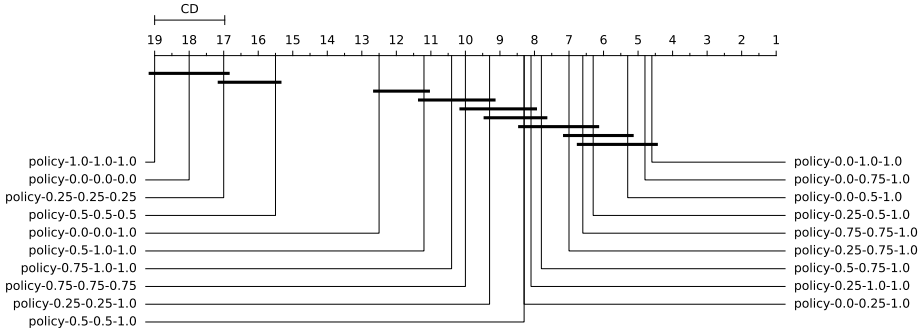


**Fig. 12.** Visualization of Friedman and Nemenyi test results for **strict modularity** with **majority h-ABCD**.

**Table 4.** Detailed results of experiments from Sec. 4.3 for the case of **linear modularity** with **strict h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.75-0.75-1.0 | 0.583035 | 0.001914 |
| policy-0.75-0.75-0.75 | 0.582899 | 0.001873 |
| policy-0.75-1.0-1.0 | 0.582889 | 0.001530 |
| policy-0.25-0.75-1.0 | 0.582140 | 0.001025 |
| policy-0.25-1.0-1.0 | 0.582111 | 0.001596 |
| policy-0.25-0.5-1.0 | 0.581496 | 0.000684 |
| policy-0.5-0.75-1.0 | 0.581150 | 0.001163 |
| policy-0.5-1.0-1.0 | 0.580949 | 0.001418 |
| policy-0.0-0.75-1.0 | 0.580588 | 0.000930 |
| policy-0.0-0.5-1.0 | 0.580580 | 0.000528 |
| policy-0.0-1.0-1.0 | 0.580555 | 0.000855 |
| policy-0.5-0.5-1.0 | 0.580505 | 0.001403 |
| policy-0.5-0.5-0.5 | 0.580387 | 0.001384 |
| policy-0.25-0.25-1.0 | 0.580097 | 0.000839 |
| policy-0.25-0.25-0.25 | 0.579939 | 0.000922 |
| policy-0.0-0.25-1.0 | 0.579437 | 0.000648 |
| policy-1.0-1.0-1.0 | 0.577242 | 0.001350 |
| policy-0.0-0.0-1.0 | 0.576973 | 0.000936 |
| policy-0.0-0.0-0.0 | 0.576508 | 0.000877 |

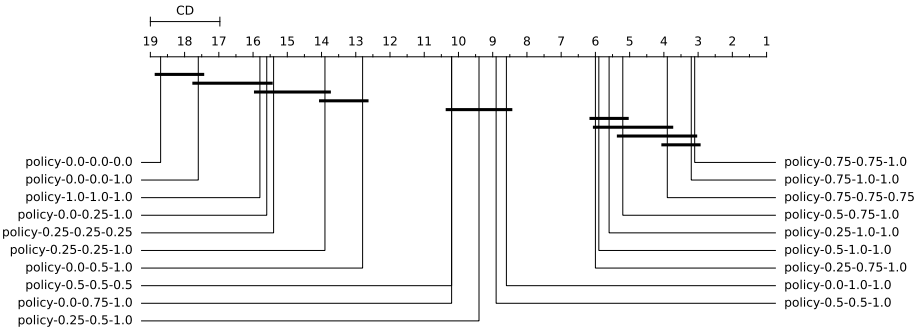


**Fig. 13.** Visualization of Friedman and Nemenyi test results for **linear modularity** with **strict h-ABCD**.

**Table 5.** Detailed results of experiments from Sec. 4.3 for the case of **linear modularity** with **linear h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.75-1.0-1.0 | 0.553803 | 0.001448 |
| policy-0.75-0.75-1.0 | 0.553197 | 0.001006 |
| policy-0.75-0.75-0.75 | 0.553059 | 0.001106 |
| policy-0.25-1.0-1.0 | 0.552988 | 0.001151 |
| policy-0.5-1.0-1.0 | 0.552743 | 0.000897 |
| policy-0.25-0.75-1.0 | 0.552738 | 0.000709 |
| policy-0.5-0.75-1.0 | 0.552715 | 0.000576 |
| policy-0.25-0.5-1.0 | 0.552109 | 0.000612 |
| policy-0.5-0.5-1.0 | 0.551876 | 0.000768 |
| policy-0.0-1.0-1.0 | 0.551628 | 0.000996 |
| policy-0.5-0.5-0.5 | 0.551426 | 0.000847 |
| policy-0.25-0.25-1.0 | 0.551390 | 0.000797 |
| policy-0.0-0.75-1.0 | 0.551355 | 0.000865 |
| policy-0.0-0.5-1.0 | 0.550627 | 0.000697 |
| policy-0.25-0.25-0.25 | 0.550586 | 0.000901 |
| policy-0.0-0.25-1.0 | 0.549692 | 0.001195 |
| policy-0.0-0.0-1.0 | 0.548950 | 0.000909 |
| policy-0.0-0.0-0.0 | 0.548078 | 0.000855 |
| policy-1.0-1.0-1.0 | 0.544506 | 0.001861 |

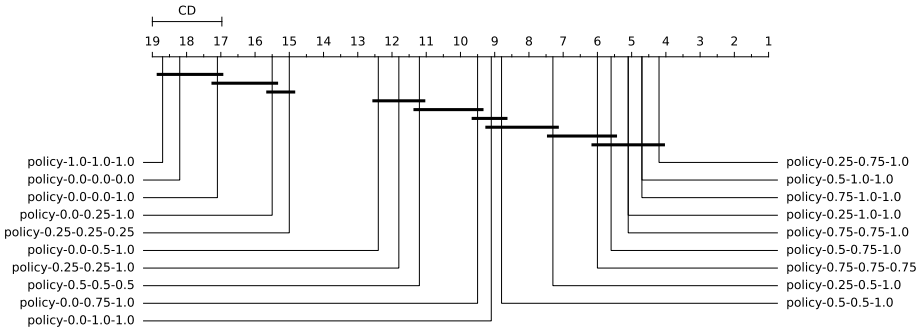


**Fig. 14.** Visualization of Friedman and Nemenyi test results for **linear modularity** with **linear h-ABCD**.

**Table 6.** Detailed results of experiments from Sec. 4.3 for the case of **linear modularity** with **majority h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.25-1.0-1.0 | 0.547888 | 0.001088 |
| policy-0.25-0.75-1.0 | 0.547571 | 0.000680 |
| policy-0.25-0.5-1.0 | 0.547143 | 0.001239 |
| policy-0.5-1.0-1.0 | 0.546262 | 0.001120 |
| policy-0.25-0.25-1.0 | 0.545856 | 0.001329 |
| policy-0.0-0.75-1.0 | 0.545830 | 0.001385 |
| policy-0.5-0.5-1.0 | 0.545795 | 0.001643 |
| policy-0.0-1.0-1.0 | 0.545691 | 0.001544 |
| policy-0.5-0.75-1.0 | 0.545676 | 0.001328 |
| policy-0.5-0.5-0.5 | 0.545565 | 0.001642 |
| policy-0.25-0.25-0.25 | 0.545422 | 0.001330 |
| policy-0.75-1.0-1.0 | 0.545278 | 0.000913 |
| policy-0.0-0.5-1.0 | 0.544920 | 0.002246 |
| policy-0.75-0.75-1.0 | 0.544874 | 0.000948 |
| policy-0.75-0.75-0.75 | 0.544769 | 0.000977 |
| policy-0.0-0.25-1.0 | 0.543846 | 0.001442 |
| policy-0.0-0.0-1.0 | 0.542411 | 0.001098 |
| policy-0.0-0.0-0.0 | 0.540734 | 0.001448 |
| policy-1.0-1.0-1.0 | 0.536671 | 0.002150 |

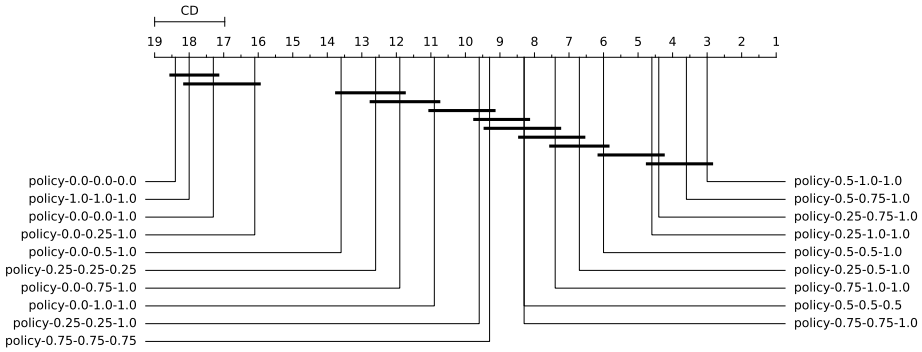


**Fig. 15.** Visualization of Friedman and Nemenyi test results for **linear modularity** with **majority h-ABCD**.

**Table 7.** Detailed results of experiments from Sec. 4.3 for the case of **majority modularity** with **strict h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.75-1.0-1.0 | 0.612747 | 0.001802 |
| policy-0.75-0.75-1.0 | 0.612150 | 0.001969 |
| policy-0.75-0.75-0.75 | 0.612096 | 0.001919 |
| policy-0.5-1.0-1.0 | 0.608267 | 0.001878 |
| policy-0.5-0.75-1.0 | 0.607718 | 0.002146 |
| policy-0.5-0.5-1.0 | 0.60506 | 0.001352 |
| policy-0.5-0.5-0.5 | 0.604750 | 0.001497 |
| policy-0.25-1.0-1.0 | 0.604107 | 0.000963 |
| policy-0.25-0.75-1.0 | 0.603918 | 0.001421 |
| policy-0.25-0.5-1.0 | 0.601904 | 0.001243 |
| policy-0.0-1.0-1.0 | 0.600812 | 0.001374 |
| policy-0.25-0.25-1.0 | 0.600524 | 0.001663 |
| policy-0.25-0.25-0.25 | 0.600475 | 0.001647 |
| policy-0.0-0.75-1.0 | 0.600407 | 0.001374 |
| policy-0.0-0.5-1.0 | 0.598773 | 0.001388 |
| policy-0.0-0.25-1.0 | 0.597274 | 0.001621 |
| policy-1.0-1.0-1.0 | 0.594255 | 0.002331 |
| policy-0.0-0.0-1.0 | 0.592029 | 0.001623 |
| policy-0.0-0.0-0.0 | 0.591359 | 0.001798 |

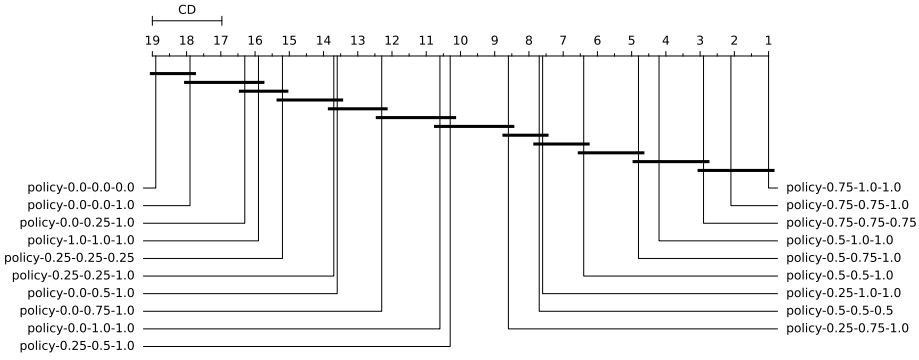


**Fig. 16.** Visualization of Friedman and Nemenyi test results for **majority modularity** with **strict h-ABCD**.

**Table 8.** Detailed results of experiments from Sec. 4.3 for the case of **majority modularity** with **linear h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.75-1.0-1.0 | 0.597977 | 0.000477 |
| policy-0.75-0.75-1.0 | 0.597941 | 0.000506 |
| policy-0.75-0.75-0.75 | 0.597785 | 0.000530 |
| policy-0.5-1.0-1.0 | 0.596106 | 0.000527 |
| policy-0.5-0.75-1.0 | 0.595675 | 0.001127 |
| policy-0.5-0.5-1.0 | 0.595044 | 0.001336 |
| policy-0.5-0.5-0.5 | 0.594313 | 0.001562 |
| policy-0.25-1.0-1.0 | 0.593421 | 0.001284 |
| policy-0.25-0.75-1.0 | 0.593265 | 0.001347 |
| policy-0.25-0.5-1.0 | 0.592274 | 0.000883 |
| policy-0.0-1.0-1.0 | 0.590636 | 0.001040 |
| policy-0.0-0.75-1.0 | 0.590026 | 0.000994 |
| policy-0.25-0.25-1.0 | 0.588817 | 0.001188 |
| policy-0.0-0.5-1.0 | 0.588693 | 0.001420 |
| policy-0.25-0.25-0.25 | 0.587252 | 0.001363 |
| policy-0.0-0.25-1.0 | 0.585226 | 0.000935 |
| policy-0.0-0.0-1.0 | 0.581413 | 0.000949 |
| policy-0.0-0.0-0.0 | 0.579814 | 0.001368 |
| policy-1.0-1.0-1.0 | 0.569114 | 0.004407 |

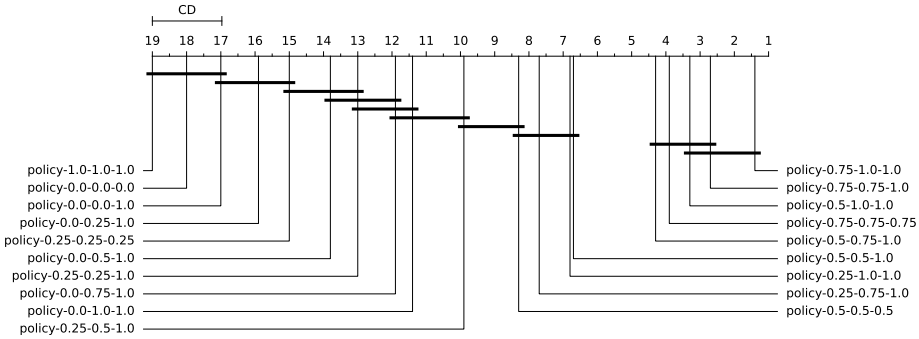


**Fig. 17.** Visualization of Friedman and Nemenyi test results for **majority modularity** with **linear h-ABCD**.

**Table 9.** Detailed results of experiments from Sec. 4.3 for the case of **majority modularity** with **majority h-ABCD** (means and standard deviations of 5 executions).

| **Policy name** | **Mean** | **Std** |
|---|---|---|
| policy-0.75-0.75-1.0 | 0.597348 | 0.001625 |
| policy-0.75-0.75-0.75 | 0.597231 | 0.001678 |
| policy-0.75-1.0-1.0 | 0.596774 | 0.002238 |
| policy-0.5-1.0-1.0 | 0.595842 | 0.001810 |
| policy-0.5-0.5-1.0 | 0.594842 | 0.002028 |
| policy-0.5-0.75-1.0 | 0.594772 | 0.001488 |
| policy-0.5-0.5-0.5 | 0.594548 | 0.001998 |
| policy-0.25-0.75-1.0 | 0.593974 | 0.001597 |
| policy-0.25-1.0-1.0 | 0.593305 | 0.001469 |
| policy-0.25-0.5-1.0 | 0.593166 | 0.002182 |
| policy-0.25-0.25-1.0 | 0.589855 | 0.001012 |
| policy-0.0-0.75-1.0 | 0.588934 | 0.003178 |
| policy-0.0-1.0-1.0 | 0.588051 | 0.002486 |
| policy-0.25-0.25-0.25 | 0.587908 | 0.001176 |
| policy-0.0-0.5-1.0 | 0.587540 | 0.003597 |
| policy-0.0-0.25-1.0 | 0.585785 | 0.002777 |
| policy-0.0-0.0-1.0 | 0.580150 | 0.004308 |
| policy-0.0-0.0-0.0 | 0.576749 | 0.004342 |
| policy-1.0-1.0-1.0 | 0.561960 | 0.005419 |

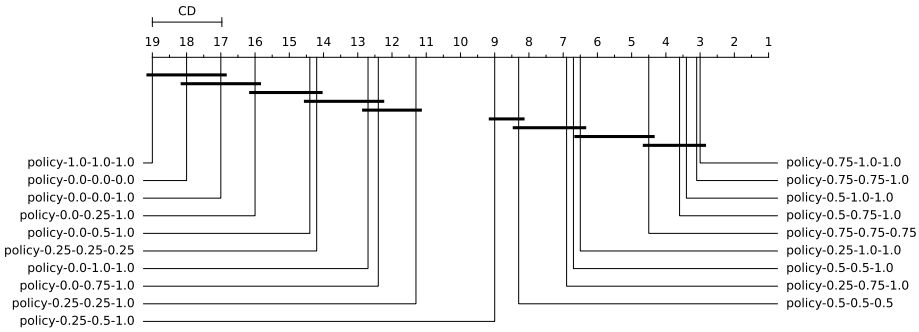


**Fig. 18.** Visualization of Friedman and Nemenyi test results for **majority modularity** with **majority h-ABCD**.