University of Hawai'i

Community Detection on Graphs with Multiple Layered Community Structures

Author
Selim Karaoglu

Professor
Daniel D. Suthers

December 20, 2019

Abstract

This project focuses on the community structure analysis on graphs with multiple layers of community structures embedded in it. To conduct experiments on multiple layers of community structures in the graph, graphs that are based on the same nodes but wired with different edges are merged. Then the merged graph is analyzed by different techniques - such as community detection algorithms, overlapping community detection algorithms, nested community detection algorithms - to recover information about community structures set in the merged graph.

Introduction

In network science, we call a community a group of nodes that have a higher likelihood of connecting to each other than to nodes from other communities[3]. Community detection methods are primary tools to uncover the community structures in graphs. In the network analysis literature, "community detection" refers to finding subgraphs of mutually associated vertices under graph-theoretic definitions[30]. Social networks are full of easy to spot communities. Indeed, the employees of a company are more likely to interact with their coworkers than with employees of other companies[3, 12]. In a small isolated group, different relations had been built with different social interactions; therefore, the same actors in the group possibly are members of several different communities. Also, since the group is small and isolated, the network that contains multiple different layers of communities casts a problematic structure to analyze. This project focuses on the analysis of graphs with multiple layers of community structures by experimenting with numerous community detection techniques.

Most real networks are characterized by well-denied statistics of overlapping and nested communities[19]. A node participating in multiple communities can be detected by overlapping community detection algorithms. A community embedded in another (larger) community can be detected with nested community detection. But there are graphs that contain multiple relational structures in a small isolated group, and there is no method in the literature tailored to analyze these types of graphs yet. Social networks in small scale form is an excellent example of this type of networks. Each node in the network participates in different communities like family relations, colleagues, classmates, etc. groups. Especially on a small scale, it's expected that different communities formed on the same set of nodes to create multiple layers of communities. This project focuses on obtaining information about community structures from graphs with multiple layered structures.

The main objective of this project is the analysis of community structures on networks with multiple layers of communities. These layers of community structures are constructed by combining graphs that are devised on the same set of nodes with dierent edges. Each graph has their own disjoint community structures, and these structures are the ground truth for this experiment. The graph structures are then merged by unifying the edges of the graphs. The merged graph now contains multiple layered community structures for the nodes that are assigned to a community in the original graphs. Then the

merged graph is analyzed by using overlapping - disjoint and nested community detection methods to recover information about the original graphs from the merged graph. The results of each community detection algorithm are compared to the original graph's ground-truth, then scored by the sequential matching ratio of the detected communities and ground-truth. Naturally, disjoint community detection algorithms are unable to discover any overlapping on the communities, therefore they will never completely uncover the multiple layers of community structures. Still, these algorithms will detect a community for each node and provide information about at least one of the original graph structures. On the other hand, overlapping and nested community detection algorithms can discover multiple community memberships for a single node and expected to reveal about the multiple layered community structure more accurate than disjoint community detection algorithms. The results of each method are presented and explained in detail to show how much information can be gathered with these methods from the graphs with multiple layers of communities.

Background

This project has several steps conducted on different environments. The graph creation with the ground truth communities are provided by extended LFR benchmark[17]. Then in igraph-R, these graphs' edges are combined and the merged graph is created. Experiments on the merged graphs are conducted with python-igraph[18], networkx[11], python-louvain[2], igraph-R and igraph-linkcomm[14] libraries. To compare the results of the community detection algorithms, sequential matching by difflib in python is employed. The results are analyzed to collect information about multiple layers of community structures.

LFR benchmark

Lancichinetti–Fortunato–Radicchi benchmark is an algorithm that generates benchmark networks (artificial networks that resemble real-world networks). They have a priori known communities and are used to compare different community detection methods.[13] The advantage of the benchmark over other methods is that it accounts for the heterogeneity in the distributions of node degrees and of community sizes[16]. This advantage of the LFR benchmark is provided with defining the degree distribution and community sizes following power-law exponent. To create the graphs with ground truth communities, extended LFR benchmark algorithm[17] is uti-

lized. Graphs created with LFR shares the number of nodes (N), but diers in other parameters such as average degree (k), maximum degree (maxk), minimum community size (minc), maximum community size (maxc), μ (mixing parameter for communities), power-law exponent for degree and community size distribution (t1 & t2), etc. Resulting graphs' edges are combined onto the same set of nodes to create multiple layers of communities.

An essential aspect of the extended LFR benchmark is that it can create directed or undirected, weighted or unweighted graphs. For this project, every single graph is created as a directed and weighted graph, however for some algorithms to work, the graphs are converted to undirected graphs owing to the fact that numerous disjoint community algorithms can only be applied to undirected graphs.

Igraph-R

The original graphs' edges are merged on igraph-R environment. After the merged graph is created, igraph-R library tools for community detection have experimented and the results are assigned to the merged graph as a vertex attribute. These community detection algorithms are; Fastreedy[5], Infomap[28], Label Propagation[24], Louvain[4], Spinglass[27] and Walktrap[22] algorithms. In addition to these algorithms, Linkcomm[1] and Clique Percolation Method[20] is also experimented on igraph-R environment.

Python-igraph

Python-igraph library is the implementation of igraph-R library to python. This environment mainly took place in seamlessly connecting the networkx and igraph-R libraries' experiments and using the advantage of the python platform that provides third-party libraries such as python-louvain library.

Networkx

Networkx provides several community detection algorithms alongside useful network analysis tools. These algorithms include k-Clique[20], Fast-Greedy[5], Label Propagation[24], Asynchronous Label Propagation[24], Asynchronous Fluid[21] and Girvan-Newman[10] community detection algorithms.

Literature Review

There are several concepts that are forming a basis for this project. In this section, a literature review is provided to help readers to comprehend the project. This review introduces the communities in the graph and explains each community detection method employed in this project. Understanding how each different community detection method works is crucial to explaining the results.

Communities are locally dense connected subgraphs in a network. All members of a community must be reached through other members of the same community. At the same time, we expect that nodes that belong to a community have a higher probability of linking to the other members of thatcommunity than to nodes that do not belong to the same community[3]. There are multiple definitions of communities that satisfy this rule. The most common definitions are cliques, strong communities and weak communities. A clique in the social sciences is a group of individuals who interact with one another and share similar interests[29]. In terms of network science, this refers to a complete subgraph. A community is defined as a strong community if each node within C has more links within the community than with the rest of the graph or defined as a weak community if the total internal degree of a subgraph exceeds its total external degree[23]. As can be interpreted from the denition; each clique is a strong community, and each strong community is a weak community at the same time[3].

Community Detection Algorithms

The community detection algorithms utilized in this project are mentioned in the previous section. The background information about these methods and a detailed explanation are provided in this section. Each method is delivered for a particular type of problem, hence the analysis results dier from each other.

Disjoint Community Detection Algorithms

There are numerous disjoint community detection algorithm exerted to analyze the graph structures. The algorithms are; Fast-Greedy, Infomap, Label Propagation, Louvain, Spinglass, Walktrap, Asynchronous Fluid and Girvan-Newman community detection algorithms.

Fast-Greedy and Louvain community detection methods are modularity based algorithms[4, 5]. Modularity in network science is defined as a value between -1 and 1 that measures the density of links inside communities compared to links between communities[4] Fast-Greedy method works by greedily optimizing the modularity, runs in time O($mdlogn$) and works best on hierarchical networks with balanced dendrogram[5].

The Louvain method for community detection is a method to extract communities from large networks. The method is a greedy optimization method that runs in time O($nlogn$). In this method, the value to be optimized is modularity. Both Fast-Greedy and Louvain methods are based on maximizing modularity, but Louvain method iteratively optimizes local communities during this process.

Modularity optimization methods suer from resolution limit, as it has been shown that modularity optimization often fails to detect clusters smaller than some scale, depending on the size of the network[9].

Infomap is a flow optimization-based community detection algorithm that runs on O($nlogn$) time. Unlike modularity based methods implemented in this experiment, Infomap also uses the edge weights and directions. The flow optimization in the algorithm is calculated by mapping the system-wide flow induced by local interactions between nodes and to retain the information about the directions and the weights of the links[28].

Label Propagation method is a fast, nearly linear time algorithm for detecting community structure in networks. The method works by labeling the vertices with unique labels and then updating the labels by majority voting in the neighborhood of the vertex[24]. In this project, the extended implementation of Label Propagation method is utilized with igraph-R library. This extension implements edge weights to the method.

Spinglass method tries to nd communities in graph by employing simulated annealing and spin-glass model. The community structure of the network is interpreted as the spin configuration that minimizes the energy of the spin-glass, with the spin states being the community indices. This method applies to weighted and directed networks[27].

WWalktrap method is based on the following intuition: random walks on a graph tend to get trapped into densely connected parts corresponding to

communities. This algorithm's runtime is O(nm^2) in worst case scenario, in sparse networks the runtime is O(mnH) where H is the height of the hierarchical community structure that may be represented as a tree called dendrogram[22].

Asynchronous Fluid community detection algorithm is based on the idea of introducing the number of fluids (i.e., communities) within a non- homogeneous environment (i.e., a non-complete graph), where fluids will expand and push each other influenced by the topology of the environment until a stable state is reached[21]. It's important to note that this algorithm is not implemented for weighted graphs.

Girvan-Newman method is a hierarchical divisive method that runs in O(n^2) time and uses betweenness centrality. Divisive procedures systematically remove the links connecting nodes that belong to different communities, eventually breaking a network into isolated communities[10]. This method can use link betweenness and random walk betweenness to dene the centrality. This project implemented the link betweenness centrality to capture the role of each edge in the graph.

Overlapping Community Detection Algorithms

The Clique Percolation algorithm, often called CFinder, views a community as the union of overlapping cliques. Two k-cliques are considered adjacent if they share k 1 nodes. A k-clique community is the largest connected subgraph obtained by the union of all adjacent k-cliques. Each k-clique community has its own threshold. For $k = 2$, the k-cliques are links which is the condition for the emergence of a giant connected component in Erdős-Rényi networks. For $k = 3$ the cliques are triangles[7, 20]. To detect every clique in the graph, the runtime grows exponentially with N. CFinder community definition is based on cliques instead of maximal cliques, which can be identied in polynomial time[15]. If, however; there are large cliques in the network, it is more efficient to identify all cliques using an algorithm with O(e^N) complexity[20]. k-clique communities naturally emerge in sufficiently dense networks; therefore, they can be emerged to interpret the overlapping community structure of a network[3].

Link communities method is designed with the idea of links being more community specific and capturing the particular relationship that decides the membership of the node. Link clustering algorithm, as referred as Linkcomm

consists of two steps. The first step is to define the link similarity and the second step is to apply hierarchical clustering. The link similarity measures the relative number of common neighbors each node pair has. Then the similarity matrix created with link similarities is exerted to calculate hierarchical clustering[1]. The link clustering algorithm involves two time-limiting steps: similarity calculation and hierarchical clustering. Calculating the similarity for a linked pair with degrees k_i and k_j requires $\max(k_i, k_j)$ steps. For a scale-free network with degree exponent γ the calculation of similarity has complexity $O(N^{2/(\gamma-1)})$, determined by the size of the largest node, k_{max}. Hierarchical clustering requires $O(L^2)$ time steps. Hence the algorithm's total computational complexity is $O(N^{2/(\gamma-1)}) + O(L^2)$. For sparse graphs the latter term dominates, leading to $O(N^2)$[3]. Linkcomm method supports directed and weighted graphs.

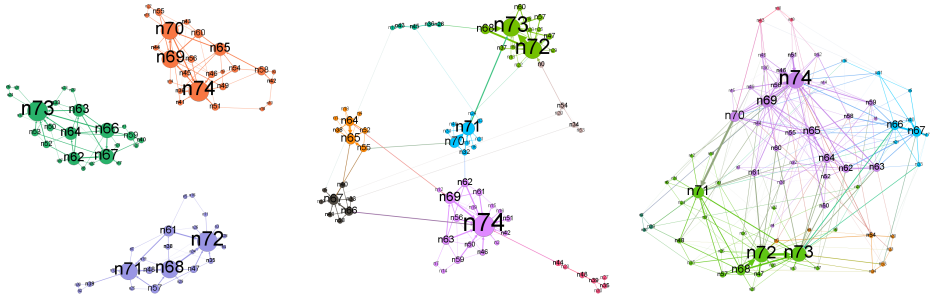
Methodology

Several combined graphs are created and experimented on in this project. All these combined graphs are based on two original graphs built on the same set of nodes. This set of nodes represents the small isolated group of actors and the two original graphs are created to resemble two different real network structures. The number of nodes and the fabric is designed on original graphs' creation with LFR benchmark in order to design the network structure to represent the real network structure accordingly. By creating different community structures for original graphs and combining the edges of both graphs, the combined graphs are created. This project's focus is on small scale networks just like Zachary's Karate Club[10] and the graphs are allbased on an isolated space with small number of nodes. The primary motivation for this project was to experiment with a specific graph of elementary school students in a small town. However, the idea of merging multiple graphs with different community structures sharing the same actors led this project's perspective to be broadened. Several different real network types are implemented to compare how different community structures affect the community detection methods' results.

Although no common definition has been agreed upon, it is widely accepted that a community should have more internal than external connections[19, 26]. Community structures for the first graph is designed to have no external connection. This disjoint large cluster structure is discovered in a paper by Blondel et al. Similar to the communication network in Belgium graph, this graph contains large cluster structure[4]. The first graph contains 3 equal

sized and completely isolated clusters (fig 1.a) and the second graph has 8 different clusters that are connected to each other forming a single weakly connected component (fig 1.b). The second graph's structure is similar to Karate Club's; community sizes are following a power-law distribution and there are external connections between communities[10]. Two different structures are merged to form the first merged graph (fig 1.c).

First merged graph contains the fewest number of nodes with 75. This graph is built by merging the edges of two different LFR benchmark graphs. One of the original graphs holds 142 edges and the other one holds 144. There are no overlapping nodes in any of the source graphs.



(a) 1st graph (b) 2nd graph (c) test75 merged graph

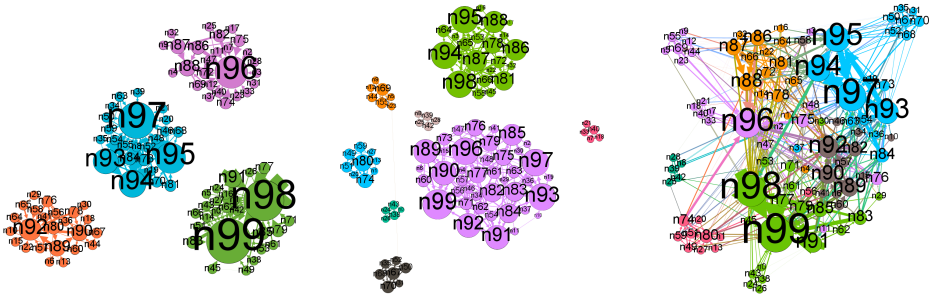
Figure 1: Original graphs' node colors represent the ground truth communities, the merged graph's node colors represent the Walktrap algorithm results.

The second test graph is based on two graphs with different community structures. The first network structure is built the same way with the previous graph's primary structure, four equal-sized disjoint communities that are completely isolated, hence forming four weakly connected components in the graph. Second original graph is also built with disjoint communities, but unlike the first one, community sizes are different and number of communities are designed similar to the test75 graph's second network structure with 8 disjoint communities.

The merged graph with 100 nodes is built by combining the edges of two different LFR benchmark graphs detailed above. First, the original graph is wired with 227 edges and the second graph is wired with 279 edges. The first graph is not as densely connected to the second graph. Different from the test75 graph, the original graphs' community structures for test100 graph-does not have edges between communities. This means that both original

graphs include multiple weakly connected components[31]. After merging the graphs, test100 graph appears as a completely connected graph.

Figure 2: Test100 original graphs and the merged graph. Original graphs' node colors represent the ground truth communities, the merged graph's node colors represent the Louvain algorithm results.



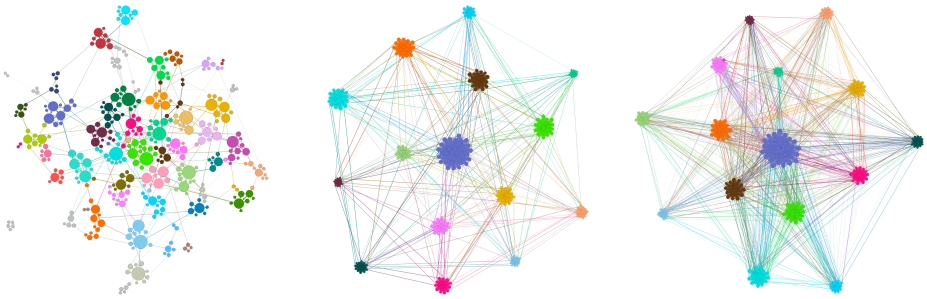
(a) test100 1st graph (b) test100 2nd graph (c) test100 merged graph

This graph represents the very fundamental idea behind this project. The isolated small group represents a small village's students and the network structures are designed to show the different types of relations between this group. First graph represents the family relationship of the group(fig 3.a) and the second graph corresponds to the weekend activity course relationship (fig 3.b). As shown in figures, both graphs' communities have lots of external edges connecting the different communities. The first graphs is one weakly connected component and the second graph is two loosely connected components. In addition, the second graph's cluster sizes are similar to each other to map the classroom structure of weekend courses; on the other hand, first graph's cluster sizes vary to interpret the different family sizes.

Test400 graph is created by merging the wiring of two different LFR benchmark graphs built with 400 nodes. First of the original graphs are built with few numbers of edges, 726. Even though this graph has few edges, number of weakly connected components are 2, and that is caused by an isolated cluster. Other graph is created with 4955 edges with strongly connected communities and numerous edges connecting the different communities.

The last graph is similar to the second graph, the test100. The underlying graphs are designed with isolated communities. The community sizes within original graphs are similar. The contrast is the number of communities for

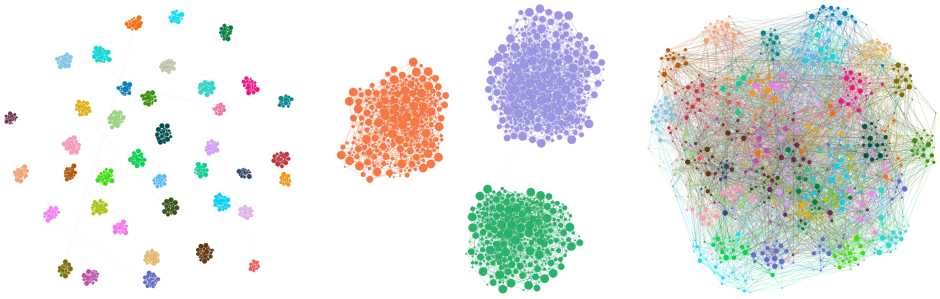
Figure 3: Test400's original graphs' node colors represent the ground truth, the merged graph's colors represent the Spinglass algorithm.



(a) test400 1st graph (b) test400 2nd graph (c) test400 merged graph

each graph; the first graph has 39 communities while the second graph carries 3 large communities. To emphasize the difference between the community sizes, this group created with more significant number of nodes which were able to form 39 small clusters in the first graph.

Figure 4: Test1000's original graphs' node colors represent the ground truth, the merged graph's colors represent the Infomap algorithm results.



(a) test1000 1st graph (b) test1000 2nd graph (c) merged graph

This graph is built with 1000 nodes. Two LFR benchmark graphs with different connections are merged to form the combined graph. The first graph and the second graph are made with 2662 and 2651 edges, respectively.

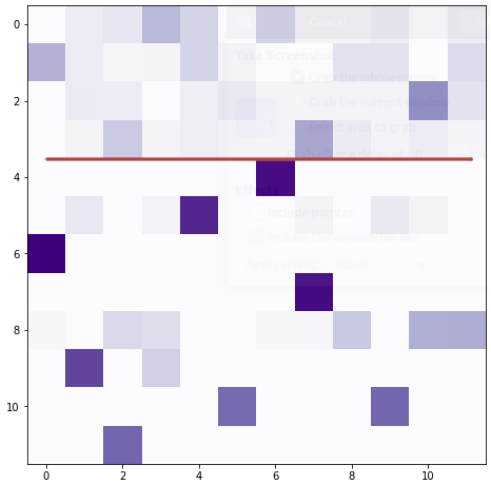
Experiment

To analyze the merged graphs and gather information about the original graphs' community structures, several community detection algorithms are used. These community detection algorithms form the basis of this experiment to reveal the community structure information. The algorithms are developed using different methods to uncover the communities in the graph.

Figure 5: Ratios between detected communities and ground truth.

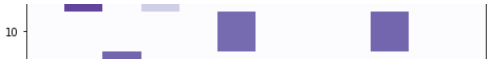
```
----------------------
Sequental matching ratio: 1.000000
Index: 36 - 12
[16, 17, 39, 83, 201, 202, 298, 312, 330, 384, 441, 473, 528, 551, 567, 590, 605, 655, 669, 754, 916, 919, 972, 988]
[16, 17, 39, 83, 201, 202, 298, 312, 330, 384, 441, 473, 528, 551, 567, 590, 605, 655, 669, 754, 916, 919, 972, 988]
----------------------
Sequental matching ratio: 0.983607
Index: 18 - 11
[15, 100, 136, 184, 187, 191, 210, 220, 221, 285, 295, 317, 367, 370, 445, 474, 489, 549, 561, 587, 588, 672, 695, 707, 721, 730, 836, 942, 958, 960, 961]
[15, 100, 136, 184, 187, 191, 210, 220, 221, 285, 295, 317, 370, 445, 474, 489, 549, 561, 587, 588, 672, 695, 707, 721, 730, 836, 942, 958, 960, 961]
```

Each graph is inspected with the same disjoint, overlapping, and nestedcommunity detection methods. The results of community detection algorithms are then compared to the ground truth communities. To compare the results with ground truth values, the sequential matching algorithm is employed. 10 This algorithm is provided by python's built-in library named difflib[6]. Sequential matching algorithm determines the similarity of two hashables by calculating twice the number of matching characters K_m divided by the total number of characters of both strings[25]. Where T is the total number of elements in both sequences, and M is the number of matches, the ratio is calculated as $2.0 * M/T$. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common[6]. By comparing the results and ground truth with sequential matching, the matching ratio be-



(a) Heatmap seperated by source graph structure

(b) A perfect match



(c) A divided match

Figure 6: Heatmap of ratios

tween each detected community and ground truth are observed to nd the best match amongst all detected communities. As shown in Figure 5, when the sequential matching ratio is 1, this inclines that the two values compared are a perfect match. The success of the community detection algorithms are measured with this technique. The execution time of this particular Python implementation was stated as as $O(n^2)$ in worst case and $O(n)$ in best case[6].

For the graphs that contain large numbers of communities, observing each possible match is not an easy task. Therefore the heatmaps are plotted to visualize the ratios between the detected communities and ground truth values. Figure 6-a shows the heatmap of sequential matching ratios between communities detected by Label propagation method on y-axis and ground truth communities of the test100 graph on x-axis. As discussed in the previous section, since the source graphs contain 4 and 8 communities in their structures respectively, the rows above the red line represent the first original graph's communities and the rows below the red line represent the second source graph's communities. This chart shows the detected community with index #4 by the Labelprop has a high sequential matching ratio with the first graph's ground-truth community with index #6. So a perfect detection of a community would look like one dark point (that points out the matching community) with the rest of the row with white (or lighter) color (fig 6.b). If there are two similarly dark-colored points on the same row (fig 6.c), this implies that this community is detected as two different communities by the community detection method (also see figure 14). The scenario of all points in the same row have different colors means the community detection algorithm was not successful in recognizing the community structure presented by that row. In this figure, the first four rows that represent the first graph's structure is not revealed by the community detection algorithm, as seen in the heatmap. Higher resolution versions of the heatmaps and graphs are provided in the appendix.

Results

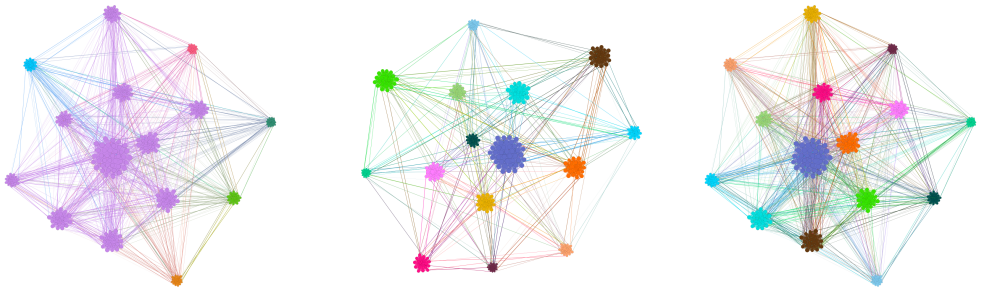
In this section, each community detection method experimented on the merged graphs - graphs with multiple layers of community structure. The community detection methods mentioned above are provided with different libraries on various platforms. Some of the algorithms are devised with extensions or modifications, some available community detection techniques could not get implemented in this project owing to the fact that they are only applicable for very small graphs (with number of edges fewer than 100)[24, 8].

Fast-Greedy algorithm is one of the modularity based algorithms. It is already stated that modularity based methods suffer from resolution limit. The effects of this resolution limit are highly visible in this experiment's results. All the merged graphs are densely connected, which is not suitable for modularity based algorithms. Even weakly interconnected complete graphs, which have the highest possible density of internal edges, and represent the best identifiable communities, would be merged by modularity optimization if the network were sufficiently large[9].

Figure 7

Fast-Greedy algorithm; for 3 of the merged graphs, detected only 1 community, which is the complete graph, but for test400 graph the algorithm detected six communities. The result is not useful for this experiment when the number of detected communities is 1. This means the algorithm, somehow did not discover any community, therefore no information about community structure is retrievable. The method detected six communities in the test400 graph and 5 of these detected communities are matching perfectly with the ground truth communities of the graph. Other detected community has a sequential matching ratio of 0.58 with one of the ground truth communities. The matching algorithms are clear in figure 7, the dark-colored points are the 5 perfectly matching results. Detected communities are all from the first layer of the community structures.

Results of the algorithm can be observed in Figure 8. There are 5 communities created on the first original graph from test400, successfully detected by the Fast-Greedy method. The rest of the nodes are assigned to the same community.



(a) Fast-Greedy community detection on test400

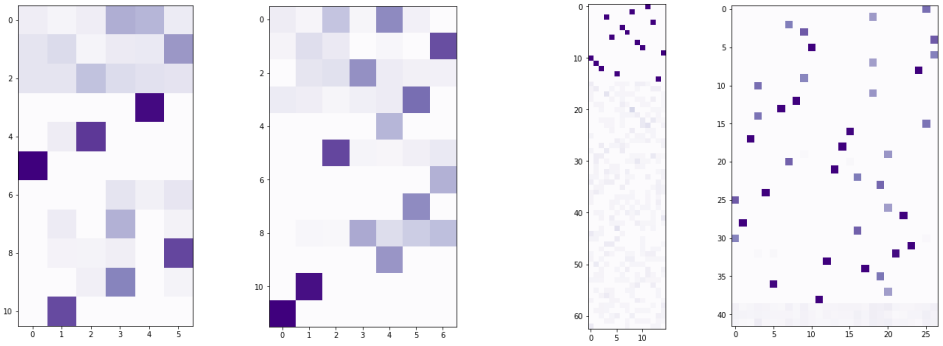
(b) test400 first original graph

(c) test400 Louvain detection results

Figure 8: test400 graph with different detection results

Louvain community detection technique is the second modularity based method. Unlike the Fast-Greedy algorithm, Louvain method discovered several communities for each test graph. For the test75 graph there are six communities detected. Louvain method resulted with 1 perfect detection result. 4 of the remaining five detected communities also have high ratios (between 0.78-0.96) and the lowest matching ratio for Louvain method is 0.59. All of these detected communities belong to the second original graph of test75. The first graph's community structure is not recognized (fig 9.a). The experiment with test100 graph resulted with seven communities, however only 2 ofthese detected communities have high sequential matching ratio; one detected community is a perfect match and the other detected community has a 0.95 matching ratio. Other detected communities follow with ratios 0.78, 0.76 and 0.67 (fig 9.b). For the test400 graph, Louvain method's results are surprising; the algorithm recovered the complete structure information from the first original graph. The 15 detected communities are a perfect match of first graphs community structure (fig 9.c). This also means the second graphs structure is completely invisible in this situation. Louvain algorithm results are compared to the test400's first original graph with 15 communities defined in it. Figure 8.b shows the community structure of the first original graph deployed in test400. The Louvain community detection results are displayed on Figure 8.c, which is completely equal to the community structure in Figure 8.b. The test1000 graph divided to 27 communities by Louvain method. 14 of these communities are perfect sequential matches with test1000's first original graph's structure. Beside of these 14 perfect matches, there are 3 detected communities with sequential matching ratio between 0.96 and 0.98. These communities are detected with a slight difference (assigned a node from a different community or not assigned a node that is in the community). The second original graph's structure, again, is not detected by Louvain method (fig 9.d).

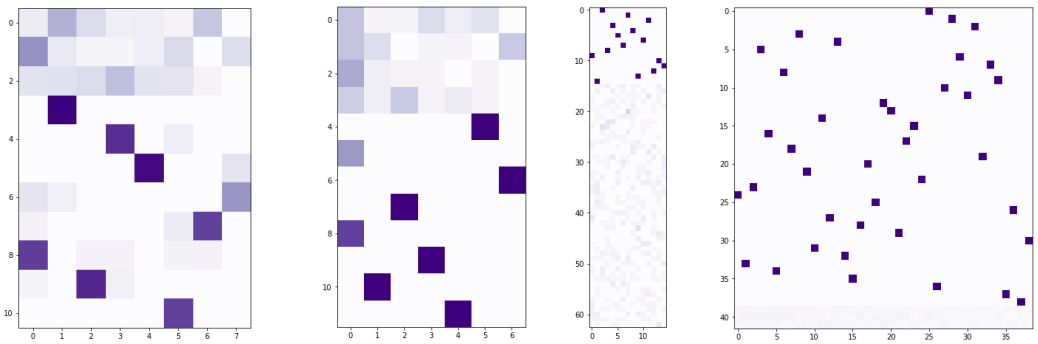
Figure 9: Heatmaps of Louvain method for test graphs



(a) test75 (b) test100 (c) test400 (d) test1000

Infomap community detection algorithm is based on flow optimization. This method can analyze directed and weighted graphs, however; through the experiment process, igraph-R library could not employ the directed graph's modularity, therefore the algorithm utilized on the undirected version of the graph. Even though information flow relies heavily on the edge directions [28, 9] and this experiment is conducted without edge directions; Infomap method still resulted with accurate community detection results. Interestingly, for the test400 and test1000 graphs, the algorithm recovered first graph's community structure with perfect ratios (fig 10.c - 10.d) but the second graph's community structure is completely ignored. For test75 and test100 graphs, recovered community structures belong to the second graphs' community structures and the first graph structures are not detected (fig 10.a - 10.b).

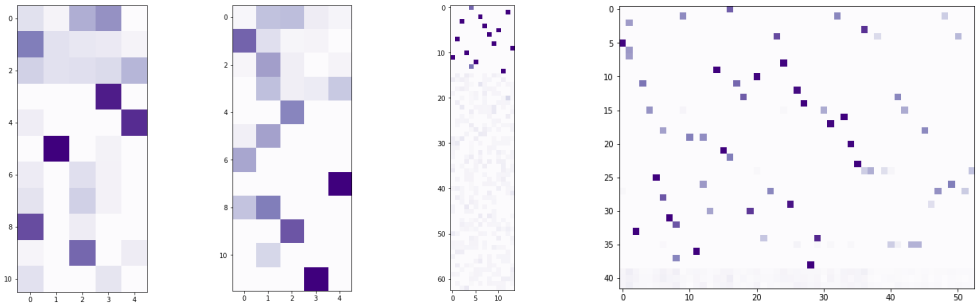
Figure 10: Heatmaps of Infomap method for test graphs



(a) test75 (b) test100 (c) test400 (d) test1000

The discovered communities are all from one of the original graphs. While Infomap algorithm successfully recovered one of the community structures embedded in the graph, the other graph's structure is not even partly recognized. This is also visible in heatmaps, for the test75 and test100 graphs, the first 3 and 4 rows of the heatmap (responds to the number of communities from the first original graph of each merged graph) does not have a high sequential match with any of the detected communities (fig 10.a - 10.b). The light and similar colored points in the same row explains this. For the test400 the situation is the same but upside down, the first few rows each are presented with a high ratio shown as dark point, and the rest of the communities (the community structure of second original graph) does not have any high matching ratio (fig 10.c). Last graph is also similar to test400, but since the number of communities in the first original graph (39) is way higher than the first one's (3), the last few rows are only detected with very low ratios.

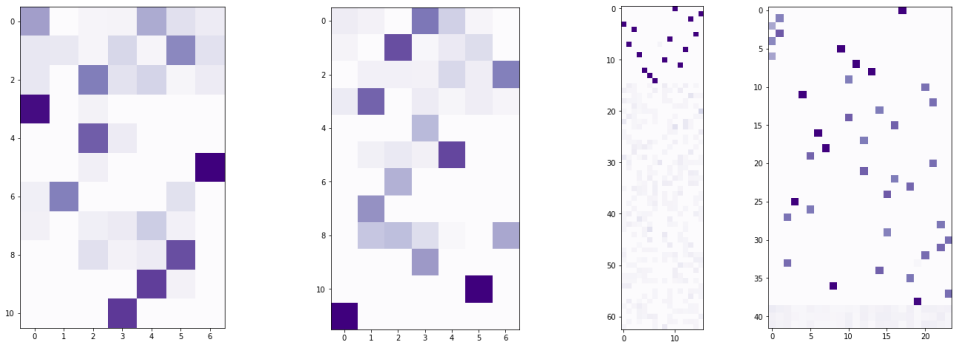
Figure 11: Heatmaps of Labelprop method for test graphs



(a) test75 (b) test100 (c) test400 (d) test1000

Label propagation algorithm is non-deterministic, there is no unique solution to this algorithm. Therefore we ran the algorithm numerous times and selected the median results for each graph experiment. In label propagation process, when many such dense (consensus) groups are created throughout the network, they continue to expand outwards until it is impossible to do so[24]. The graphs employed in this project are a good example of this and label propagation suffers from the dense community structures in the graphs. For test75 graphs, the best matching community has a 0.93 matching ratio. Rest of the matching ratios are between 0.5 and 0.84 (fig 11.a). The graphs employed in this project are a good example to this and label propagation suffers from the dense community structures in the graphs. For test75 graphs, the best matching community has a 0.93 matching ratio. Rest of the matching ratios are between 0.5 and 0.84 (fig 11.b). The label propagation algorithm generated the best results on the test400 graph by detecting 13 perfect matches out of 15 communities created on the first original graph. Similar to test400, label propagation algorithm detected ten perfect matches and 7 matches with ratios above 0.9 out of 39 communities originated in the first original graph's structure.

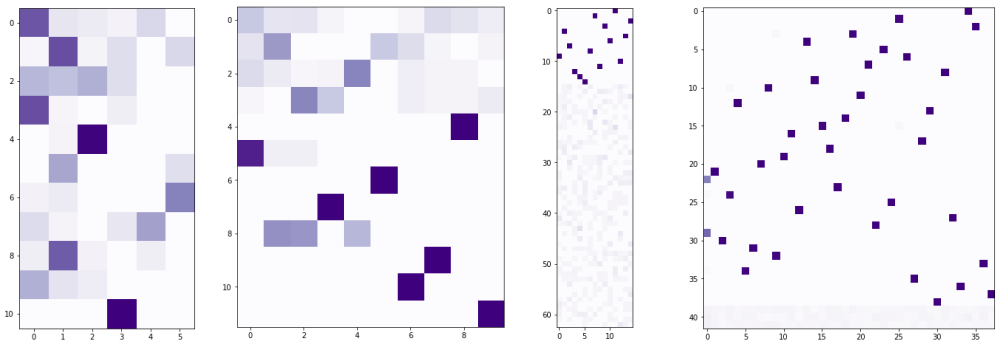
Figure 12: Heatmaps of Spinglass method for test graphs



(a) test75 (b) test100 (c) test400 (d) test1000

Spinglass algorithm deploys directed and weighted edges. Very similar to Labelprop, Spinglass algorithm did not discover any ground truth communities on test75 graph with a perfect match. The highest match is 0.93 (fig 12.a). Also for test100 ground there are two perfectly discovered community, but the rest of the detections don't have high sequential matching ratios. But the recovered information in this graph does not belong to only one of the original graphs' structure, the algorithm recovered information partly from both of the underlying community structures. Figure 12.b shows that the highest ratio for each column (detected communities) belong to the different structure. The first 4 rows are the first original graph's communities and the following 7 rows are the second graph's communities. While the 1st and 6nd communities detected by the algorithm match with the communities from the second original graph, the 2nd and 3rd communities match with the first graph's communities. Test400 and test1000 graph results are very similar with the Louvain algorithm results. The dominating structure is the first original graph's structure for both test400 and test1000 graphs. The results for test400 graph matches almost perfectly with all ground truth communities from first graph and 9 out of 39 communities are detected perfectly in test1000 graph with Spinglass method (fig 12.c - 12.d).

Figure 13: Heatmaps of Walktrap method for test graphs



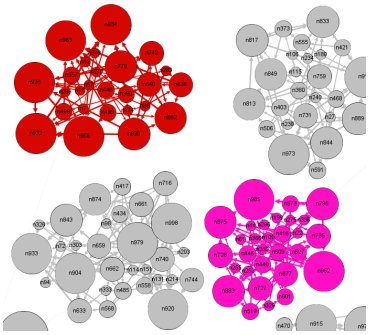
(a) test75 (b) test100 (c) test400 (d) test1000

Walktrap algorithm results show that this method only recovers one of the underlying structures on all merged graphs. The sequential matching ratios between detected communities and ground truth is not high on the test75 graph. Two detections have matching ratios roughly equal to 0.83 (fig 13.a). On test100 graph analysis results, 6 of the 10 detected communities are a perfect match for second original graph's structure in test100 (fig 13.b). There are 15 communities detected in test400 graph and all of the detected communities match perfectly with the first underlying community structure

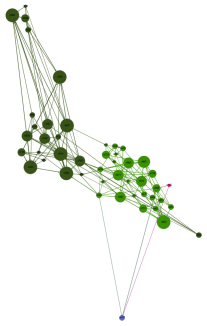
of the graph (fig 13.c). There are 38 communities detected by the Walktrap algorithm in the test1000 graph. As stated previously, the first original graph of the test1000 graph has a structure with 39 communities. 37 of the detected communities match with the ground truth structure perfectly.

There is an interesting result in the detection here; the first detected community of the walktrap algorithm (fig 14.b), actually is the combination of 23th and 30th communities from test1000 graph (fig 14.a). The two lightly colored ratios in the first column of the Figure 13.d shows the detection.

Figure 14: Walktrap detection and ground truth



(a) 23th and 30th communities



(b) Walktrap 1st community

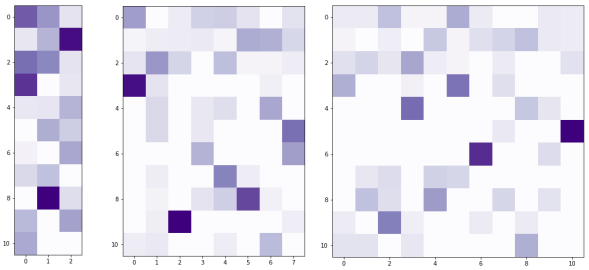
Girvan-Newman method is the only hierarchical divisive method employed in this project. This method is implemented with networkx library. The method results are achieved for test75 and test100 graphs, but for test400 and test1000 methods algorithm was not able to produce results. Since the results produced by the algorithm is not complete, the comparison with other community detection algorithms would not be meaningful.

Asynchronous Fluid community detection takes the number of communities to be found as an argument. Since the graphs are created with multiple layers of community structures, the experiment with this method conducted on several steps. First the number of communities to be found as assigned as the number first original graph's communities. On the second step, the number of communities to find is assigned to second source graph's ground-truth communities. Finally, the sum of the number of communities for both source graphs is assigned to the argument of the method.

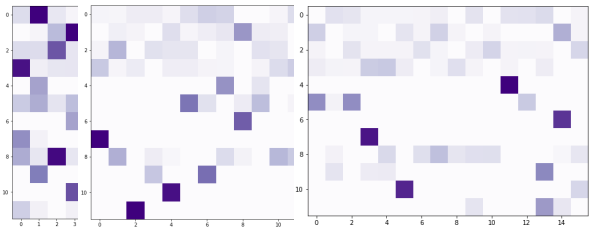
Asynchronous Fluid comment resulted with a 0.68 ratio at it's best, the second and third variations resulted in slightly higher ratios of 0.84 and 0.93. Even though these ratios show detected communities are close results to the ground truth community structure, the rest of the ratios are low to deliver any healthy information about the community structure from the merged graph (fig 15.a).

Figure 15: Asynchronous Fluid heatmaps

(a) AsynFluid heatmap - test75



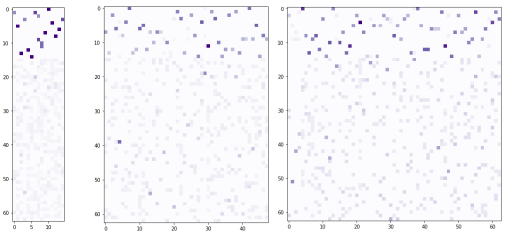
(b) AsynFluid heatmap - test100



The analysis of test100 graph is not too different from than test75 graph, but this time when the Asynchronous Fluid method employed with the second underlying structure, 3 of the 12 communities detected are perfect matches with the ground-truth communities, when the algorithm tried to find the total number of communities, 2 of the 16 communities matched perfectly(fig 15.b).

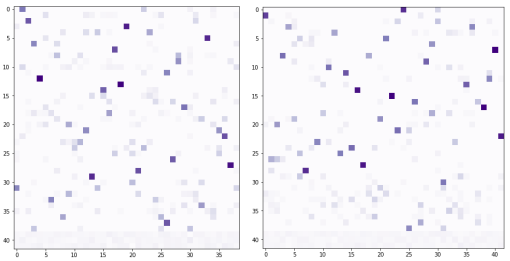
The analysis of test100 graph is not too different from than test75 graph, but this time when the Asynchronous Fluid method employed with the second underlying structure, 3 of the 12 communities detected are perfect matches with the ground-truth communities, when the algorithm tried to find the total number of communities, 2 of the 16 communities matched perfectly (fig 16).

Figure 16: Heatmaps of Asynchronous Fluid methods for test400 graph



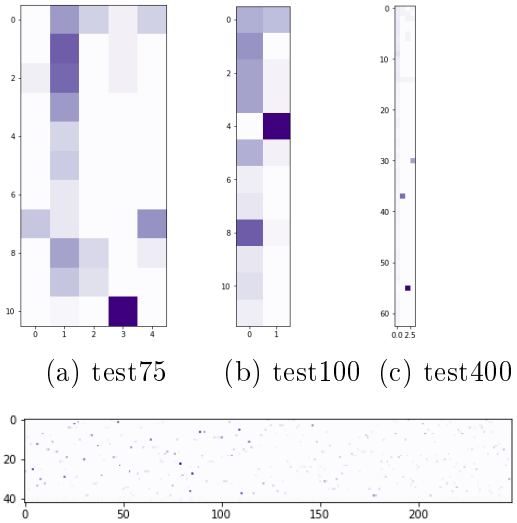
For the test1000 graph, the algorithm did not find any community that matches perfectly with a ground truth community. Especially on the detection where the number of communities to be found is equal to second original graph's communities and the highest sequential matching ratio is 0.47. First and the last implementations return some almost perfect matching ratios like 0.97, but the number of matches with high ratios are rather few (fig 17).

Figure 17: Heatmaps of Asynchronous Fluid methods for test1000 graph



The Clique Percolation method is able to detect overlapping communities. This method theoretically can detect nodes in the graph that is a member of multiple communities[20]. In this experiment, each node in all of the merged graphs is assigned to 2 different communities. This is provided by the process of embedding two graphs' edges on the same set of nodes. But this embedding process results with a dense, complete graph for each experiment. When that is the case, the Clique Percolation method suffers from this structure, owing to the fact that the merged graphs contain lots of cliques.

Figure 18: Heatmaps of Clique Percolation method (k=3) for test graphs



(a) test75 (b) test100 (c) test400

(d) test1000

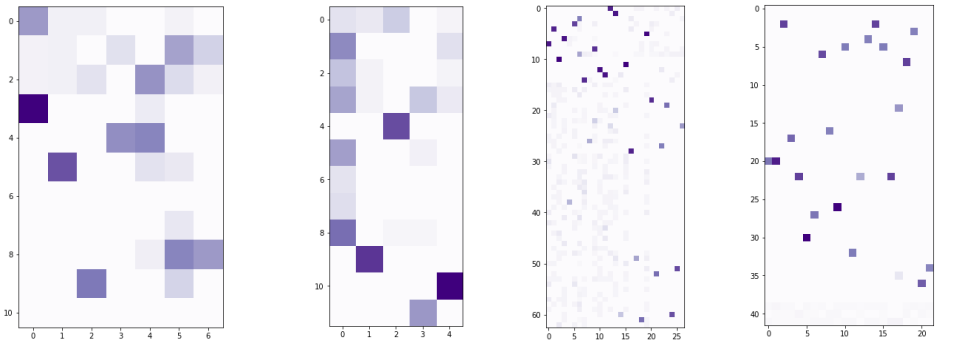
The Clique Percolation algorithm requires an argument, the clique size. If the clique size is 2, the algorithm will always result as a complete graph in densely connected graphs. The first experiment is conducted with clique size as 3. The algorithm finds every triangle connection in the graph to detect communities.

The Clique Percolation method on inspecting triangles did not yield high matching ratios for any of the merged graphs. For test75 graph, there are

only 3 communities detected and the highest ratio is 0.75 on a 5 member community (fig 18.a). The experiment for test100 graph resulted similarly; the number of detected communities is 2 and the highest ratio is 0.83 on a 5 member community (fig 18.b). There are 4 communities detected in the test400 graph; the highest ratio is 0.91, and the rest of the ratios are lower than 0.6 (fig 18.c). Surprisingly, there are 248 communities detected on the test1000 graph. Only one of the detected graphs has a high ratio with 0.93 on an almost perfect detection; however, the rest of the matching ratios are lower than 0.7 (fig 18.d).

The Clique Percolation method, when clique size is defined as 4, has a higher chance of detecting the ground-truth communities. This is a result of the merged graphs' densely connected design. In the merged graphs there are lots of triangles or 3 cliques, due to the high connectedness in the nature of graphs. But 4 cliques generally does not occur as much as 3 cliques if the graph is not a complete graph.

Figure 19: Heatmaps of Clique Percolation method (k=4) for test graphs



(a) test75 (b) test100 (c) test400 (d) test1000

The Clique Percolation algorithm detected 7 communities on test75 graph with the highest matching ratio calculated as 0.72 (fig 19.a). Similarly, on test100 graph; highest ratio of 5 detected communities is 0.87 (fig 19.b). In contrast to these two graphs, Clique Percolation method detected 2 communities perfectly and 12 communities with high matching ratios between 0.85 and 0.9 on test400 graph. Interestingly, this method recovered information from both structures with high matching ratios. As can be seen from the Figure 19-c, there are 2 perfectly detected communities and one of these communities (1st community) is from the first original graph's structure, while the other one (16th community) is from the second original graph. The

rest of the high ratio matches are also shared between both structures. The matching ratios for test1000 graph are very low, and the highest matching ratio is 0.4 (fig 19.d). The method does not produce any high ratio matches for larger clique sizes.

Link communities method focuses on link similarities of the edges that connect the nodes in the graph to discover the community structure of the network. Linkcomm is able to detect overlapping nodes, nested communities and overlapping clusters. As mentioned before, disjoint community detection algorithms are not able to recover information from multiple layers of communities. Clique percolation method is able to detect overlapping community structures. However, the algorithm only focuses on complete k- cliques on the graph and does not have the capability of detecting nested communities. Linkcomm is the only method in this project that can detect nested communities and overlapping clusters.

Linkcomm method is utilized with unweighted edges for test75. Due to the sparsity of the graph, Linkcomm method, when directed edges included, detected a single community, which is the complete graph. For this reason, the results of the unweighted graph's analysis is presented. Linkcomm detected 45 communities on test75 graph. The sequential matching ratio between detected communities and ground truth values are not too high. There are only 8 detected communities with ratios higher than 0.67 (fig 20). The best matching detected community with a matching ratio of 0.8 contains another community inside it. This is an interesting find because Linkcomm partly recovers the underlying community membership on a group of nodes. While the group of nodes discovered by Linkcomm belong to the same community in the second original graph, the nested community nodes are a part of a different community in the first original graph. Figure 21-b shows the ego network of node named 71, and the nodes are colored with their second graph's community memberships. Figure 21-a shows the nested community detection by Linkcomm, in this case, detected the green community with 0.8 matching ratio. The linkcomm detection with pie charts indicating possible community memberships for each node is shown on Figure 21-c.

Figure 20: Linkcomm on test75 graph

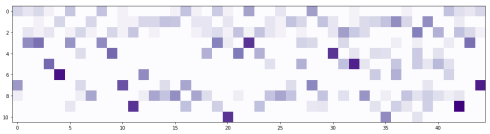
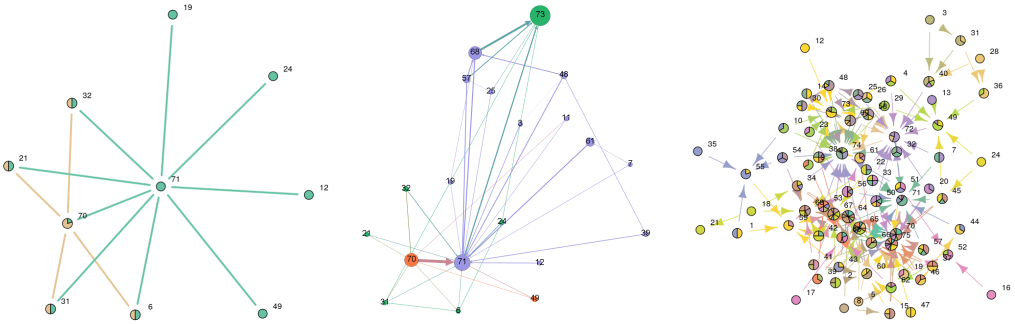


Figure 21: Linkcomm nested community detection on test 75 graph



(a) Nested community (b) Ego graph of node 71 (c) Linkcomm on test75

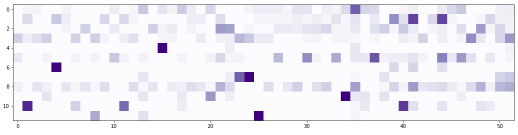
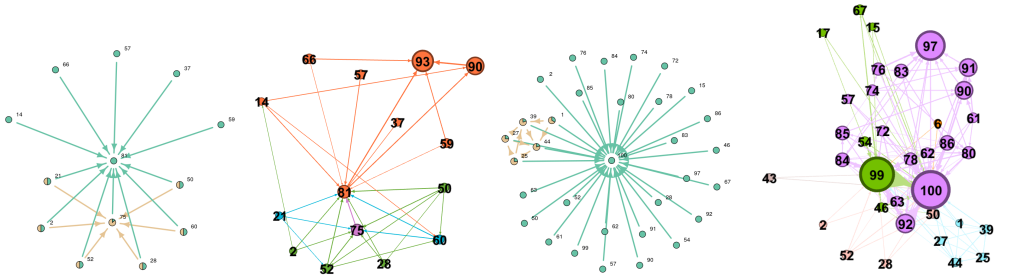


Figure 22: Linkcomm on test100 graph

Link community detection method results are more accurate on test100 graph. This test graph is implemented with directed and weighted edges; therefore, the detection was more accurate. There are 52 detected communities, and 5 of them are perfect matches. The following 9 matches also have high ratios between 0.67 and 0.87 (fig 22). There are 13 nested communities detected in this network. This also supports the expectation of recovering information from both layers of communities. Figure 23 displays 2 of the nested communities detected on the test100 graph. The community shown on figure 23- detected the green community shown in figure 23-b. Similarly, figure 23-c is another detected nested community structure from test 100 graph and figure 23-d is the ego network of the central node in the detection, 100th node in this case. The detected nested community in figure 23-c with yellow color is the blue-colored community in figure 23-d with an exact matching.

Figure 23: Linkcomm nested community detection on test 100 graph



(a) Nested community (b) Ego graph of node 81 (c) Nested community (d) Ego graph of node 100

Linkcomm method performed the best with test400 graph. The algorithm detected 13 communities with a perfect match out of 154 identified communities on the graph. As the sequential matching ratio heatmap in figure 24 suggests, the matches recover valuable information from both original graphs' structures. While the first 22 detected communities provide information from the first original graph, the rest of the detection focused on the second original graph's community structure. The ratios related to the second graph's structure are lower than the first graph's since the algorithm detected the structure partially. Communities detected by this method are generally small in size. Hence the algorithm rather discovers several small communities instead of larger communities. This resulted in low ratios, but still, the discovered structure information is useful. As mentioned before, the algorithm creates a dendrogram from edge similarity matrix, and calculates the cut for hierarchical clustering. By changing the level of cut in the dendrogram, bigger communities are still possible to find.

Figure 24: Linkcomm on test400 graph

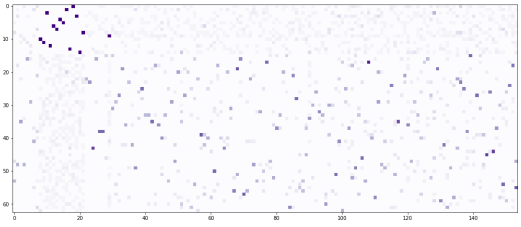
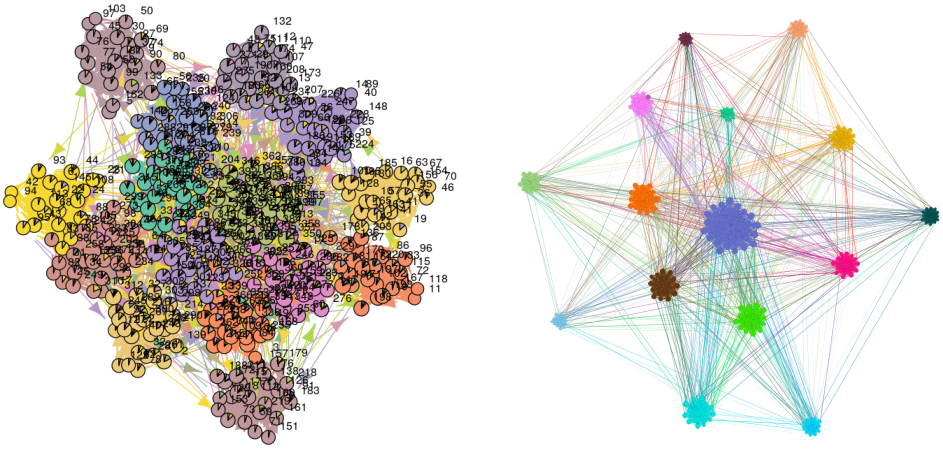


Figure 25: Linkcomm nested community detection on test 400 graph



(a) Linkcomm on test400

(b) test400 graph

In contrast to other graphs, the algorithm did not nd any nested communities on test400 graph. There are numerous nodes that are assigned to multiple communities by the algorithm. However, since the connectivity is higher in this graph and this is caused by the first original graph's structure,

link similarities of the nodes that are in the same community on the first graph have higher chances to form a community (fig 25.a - 25.b). The pie charts of nodes in the figure 26-a show most of the nodes are assigned to multiple communities with very low edge similarities.

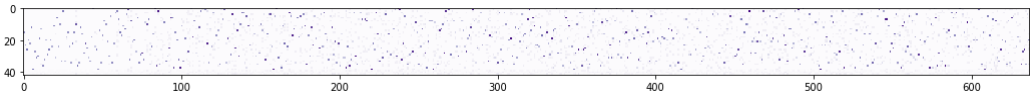


Figure 26: Heatmap of Linkcomm on test1000 graph

Link community method resulted poorly on test1000 graph. The highest matching ratio amongst 637 detected communities is 0.53 (fig 26). This is due to graph and the community sizes being larger than other merged graphs in scale. There are 11 nested communities detected by the algorithm. These nested communities does not have good matching ratios.

inkcomm algorithm can detect multiple community structures in the graph. However, the graph structure tested in this project is different. While the Linkcomm technique's main focus is on the nested communities, the graphs employed in this project have multiple layers of community structures. Even though the algorithm is not designed for such a task, it is still reliable to recover information from graphs with multiple layered community structures, especially on a small scale.

Conclusion

In this project, several community detection methods are utilized to analyze the community structure of graphs that are created by combining multiple different community structures based on the same set of nodes. To achieve this goal, two graphs build on same vertices with different community structures are combined by merging the edges of the two graphs. These graphs are created with extended LFR benchmark and the ground truth community membership values are already defined in the creation of the graphs. The community detection algorithms are employed to analyze the community structure of the merged graphs, and then results are compared to the ground-truth community membership values. To compare the detected communities and the ground-truth, the sequential matching method is employed.

Disjoint community detection methods are not able to assign a single vertice to multiple communities, therefore can not recover complete structure

information from merged graphs. However, the results are still interesting to see which structures are easier to detect by disjoint community detection techniques. All methods except Spinglass recovered information from only one of the multiple community structures. Spinglass algorithm was able to recover information from two different structures on one of the merged graphs. In addition, the methods recover the information about smaller communities better than the large ones. For all of the merged graphs, the structure with smaller community sizes are favored by disjoint community detection methods. The results of experiments conducted on test graph based on 400 vertices also proved the structures with higher average degrees or higher connectivity is still more apparent for the algorithms even the number of links between communities are higher. The most accurate detection amongst disjoint community detection methods is provided by Infomap algorithm. Following the Infomap, the Walktrap method has the second-highest sequential matching ratios, but the method suffers from bad results on the smallest merged graph. Spinglass, Label Propagation andLouvain methods also recover some useful community information; however, overall matching rates are not high as other methods owing to the fact that these methods suffered on graphs with smaller scale. Asynchronous Fluid community detection resulted with overall low matching ratios. Fast-Greedy method suffers from a resolution limit and thus performed poorly. The communities detected by Asynchronous Fluid and Fast-Greedy methods do not match the ground truth information in high rates for any graph experimented in this project.

The Clique Percolation method searches for the complete k-cliques in the graph to form communities; however, the LFR benchmark community memberships are not that strict. This resulted in the CPM method performing badly on small isolated network structures. The 3-clique detection results are far from matching the ground truth results. The 4-cliques on CPM was not very successful on the smallest two graphs. In contrast, the structure from multiple community structures is detected with high ratios on merged graph with 400 vertices. The largest graph detection was only successful on one of the multiple community structures. Linkcomm method detects the highest number of communities for each graph, but this also caused the community detections to have higher sequential matching ratios. This method is able to recover nested community structures from the graph. These overlapping and nested community detections recovered the overlapping community structure. Therefore, Linkcomm was the most accurate method to recover community structure information from the merged graphs experimented on this project.

Future Work

This project focuses on the community structure information detection on graphs with multiple layers of community structures. The experiments are conducted by employing several community detection algorithms. Since these algorithms dene community in several different ways and utilizes different methods to assign community membership for each node, some other community detection algorithms can also be utilized and tested on the graphs with multiple layers of community structures. In this project, the space defined for each graph is small-scaled and the group of actors is considered isolated. Future experiments can be conducted on larger scales with similar structures. The merged graphs experimented in this project are created to resemble the social network structures that occur in real life. In the future, this experiment can be conducted on a real network.

References

[1] Ahn, Y.-Y., Bagrow, J. P., and Lehmann, S. *Link communities reveal multiscale complexity in networks* Nature, 466, 761-765, 2010.

[2] Aynaud T. *https://github.com/taynaud/python-louvain* 2009.

[3] Barabasi A.-L. *Network Science* http://networksciencebook.com/chapter/9.

[4] Blondel V., Guillaume J., Lambiotte R., Lefebvre E. *Fast unfolding of communities in large networks* J. Stat. Mech. P10008, 2008.

[5] Clauset A., Newman M.E.J., Moore C. *Finding community structure in very large networks* 2004.

[6] *difflib* Helpers for computing deltas, inside the Python documentation. https://docs.python.org/3/library/difflib.html

[7] Erdős, P., Rényi, A. *On Random Graphs. I* (PDF). Publicationes Mathematicae. 6: 290–297, 1959.

[8] Fagiolo G. *Clustering in complex directed networks* Physical Review E 76(2 Pt 2):026107, 2007.

[9] Fortunato S., Barthelemy M. *Resolution limit in community detection* Proceedings of the National Academy of Sciences of the United States of America. 104 (1): 36–41, 2007.

[10] Girvan M. and Newman M. *Community structure in social and biological networks* Proc. Natl. Acad. Sci. USA 99, 7821–7826, 2002.

[11] Hagberg A., Schult D. and Swart P. *Exploring network structure, dynamics, and function using NetworkX* Proceedings of the 7th Python in Science Conference, 2008.

[12] Homans G.C. *The Human Groups* Harcourt, Brace & Co, New York, 1950.

[13] Hua-Wei Shen *Community Structure of Complex Networks* Springer Science & Business Media. 11–12, 2013

[14] Kalinka A. *Tools for Generating, Visualizing, and Analysing LinkCommunities in Networks* 2015.

[15] Kumpula J.M., Kivelä M., Kaski K., and Saramäki J. *A sequential algorithm for fast clique percolation* Physical Review E, 78:026109, 2008.

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

[17] Lutov A. *https://github.com/eXascaleInfolab/LFR-Benchmark_ UndirWeightOvp* 2009.

[18] Nepusz T., Péter P. *IGraph library* Version: 0.7.1, 2006-2012.

[19] Newman, M. and Girvan, M. *Finding and evaluating community structure in networks* Physical Review E69, 026113, 2004.

[20] Palla, G., Der'enyi, I., Farkas, I. and Vicsek, T. *Uncovering the overlapping community structure of complex networks in nature and society* Nature 435, 814, 2005.

[21] Parés F., Garcia-Gasulla D. et al. *Fluid Communities: A Competitive and Highly Scalable Community Detection Algorithm* https://arxiv.org/pdf/1703.09307.pdf, 2017.

[22] Pons P. and Latapy M. *Computing communities in large networks using random walks* http://arxiv.org/abs/physics/0512106, 2006.

[23] Radicchi F., Castellano C., Cecconi F., Loreto V., and Parisi D. *Defining and identifying communities in networks* PNAS, 101:2658–2663, 2004.

[24] Raghavan, U.N., Albert, R., Kumara, S. *Near linear time algorithm to detect community structures in large-scale networks* Phys Rev E 76, 036106, 2007.

[25] Ratcliff J. and Metzener D. *Pattern Matching: The Gestalt Approach* Dr. Dobb's Journal, 1988.

[26] Reichardt, J. and Bornholdt, S. *Detecting fuzzy community structures in complex networks with a potts model* Phys. Rev. Lett.93, 218701, 2004.

[27] Reichardt J. and Bornholdt S. *Statistical Mechanics of Community Detection* Phys. Rev. E, 74, 016110, http://arxiv.org/abs/cond-mat/0603718, 2006.

[28] Rosvall M. and Bergstrom C., *Maps of information flow reveal community structure in complex networks* , PNAS 105, 1118 http://dx.doi.org/10.1073/pnas.0706851105, http://arxiv.org/abs/0707.0609, 2008.

[29] Salkind N. *Cliques* Encyclopedia of educational psychology. Sage Publications, 2008.

[30] Suthers, D. D., Fusco, J., Schank, P., Chu, K.-H., & Schlager, M. *Discovery of community structures in a heterogeneous professional online network* In Proceedings of the Hawaii International Conference on the System Sciences (HICSS-46), January 7-10, 2013.

[31] Tarjan R. *Depth-First Search and Linear Graph Algorithms* in SIAM Journal on Computing 1 (2): 146–160, 1972.

Appendix

The Code

igraph-R

This block of code is the merged graph creation where two LFR benchmark graphs' edges are combined to form a single merged graph. Then several community detection methods are tested and the results are assigned as a vertex attribute.

```
library(igraph)
library(tibble)
lfr_data_dir <- "... directory ..."
graphml_dir  <- "... directory ..."
read_nse_as_edges <- function (nse_file) {
  df <- read.delim(nse_file, header = FALSE, comment.char = "#")
  colnames(df) <- c("Source", "Target", "weight")
  return(as_tibble(df))
}
read_nmc_as_nodes <- function (nmc_file, name=NA) {
  df <- read.delim(nmc_file, header = FALSE, comment.char = "#")
  colnames(df) <- c("Id",
                    if (is.na(name)) {"Cluster_ID"}
                    else {paste0("Cluster_ID_", name)})
  return(as_tibble(df))
}
###First original graph from LFR benchmark
A.edges <- read_nse_as_edges(paste0(lfr_data_dir, "/", "first.nse"))
A.nodes <- read_nmc_as_nodes(paste0(lfr_data_dir, "/", "first.nmc"), name="A")
structure <- graph_from_data_frame(A.edges, vertices=A.nodes)
summary(structure)
write_graph(structure, paste0(graphml_dir, "/first.graphml"), format="graphml")
###Second original graph from LFR benchmark
A.edges <- read_nse_as_edges(paste0(lfr_data_dir, "/", "second.nse"))
A.nodes <- read_nmc_as_nodes(paste0(lfr_data_dir, "/", "second.nmc"), name="A")
structure <- graph_from_data_frame(A.edges, vertices=A.nodes)
summary(structure)
write_graph(structure, paste0(graphml_dir, "/second.graphml"), format="graphml")
A.edges <- read_nse_as_edges(paste0(lfr_data_dir, "/", "first.nse"))
A.nodes <- read_nmc_as_nodes(paste0(lfr_data_dir, "/", "first.nmc"), name="A")
```

```
B.edges <- read_nse_as_edges(paste0(lfr_data_dir, "/", "second.nse"))
B.nodes <- read_nmc_as_nodes(paste0(lfr_data_dir, "/", "second.nmc"), name="B")
edges <- rbind(A.edges, B.edges)
nodes <- cbind(A.nodes, B.nodes[,2])
structure_A <- graph_from_data_frame(A.edges, vertices=A.nodes)
summary(structure_A)
structure_B <- graph_from_data_frame(B.edges, vertices=B.nodes)
summary(structure_B)
network <- graph_from_data_frame(edges, vertices=nodes)
summary(network)
is_simple(network)
network <- simplify(network)
network$name <- "Merged Graph"
network$transitivity <- transitivity(network, type="global")
network$mean_distance <- mean_distance(network)
network$assortativity_degree <- assortativity_degree(network)
d <- degree(network)
network$k_min <- min(d)
network$k_avg <- mean(d)
network$k_max <- max(d)
undirected <- as.undirected(network)
greedy <- cluster_fast_greedy(undirected, merges=FALSE)
V(network)$comm_greedy <- membership(greedy)
(network$modularity_greedy <- modularity(greedy))
infomap <- cluster_infomap(undirected)
V(network)$comm_infomap <- membership(infomap)
(network$modularity_infomap <- modularity(infomap))
labelprop <- cluster_label_prop(undirected)
V(network)$comm_labelprop <- membership(labelprop)
(network$modularity_labelprop <- modularity(labelprop))
louvain <- cluster_louvain(undirected)
V(network)$comm_louvain <- membership(louvain)
(network$modularity_louvain <- modularity(louvain))
spinglass <- cluster_spinglass(network)
V(network)$comm_spinglass <- membership(spinglass)
(network$modularity_spinglass <- modularity(spinglass))
walktrap <- cluster_walktrap(network)
V(network)$comm_walktrap <- membership(walktrap)
(network$modularity_walktrap <- modularity(walktrap))
summary(network)
write_graph(network, paste0(graphml_dir, "/merged.graphml"), format="graphml")
```

This part utilizes the Linkcomm method.

```
merged <- read.graph("Graphml_Networks/merged.gml", format="gml")
summary(merged)
```

```
merged.edges <- as_edgelist(merged)
merged.wedges <- cbind(merged.edges, E(merged)$weight)
merged.dwlc <- getLinkCommunities(merged.wedges, hcmethod="
    average", directed=TRUE)
for (i in seq(1, length(merged.dwlc$clusters), by=1)){
     print(getNodesIn(merged.dwlc, clusterids=i))
     }
plot(merged.dwlc, type = "graph", layout=layout_with_kk, node.
    pies=TRUE, ewidth=1)
head(merged.dwlc$numclusters, 5)
getNodeCommunities <- function(lc, id) {
  lc$nodeclusters$cluster[which(lc$nodeclusters$node == id)]
}
getNodeCommunities(merged.dwlc, 6)
merged.nestedcomm <- getAllNestedComm(merged.dwlc, plot=TRUE)
```

python

The libraries used in this project:

```
import igraph as ig
import itertools
import networkx as nx
import difflib
import matplotlib.pyplot as plt #to plot data
import math
import numpy as np
#K-clique community detection
from networkx.algorithms.community.kclique import
    k_clique_communities
#Fast greedy modulartiy algorithm
from networkx.algorithms.community.modularity_max import
    greedy_modularity_communities
#Label Propagation algorithms
from networkx.algorithms.community.label_propagation import
    label_propagation_communities
from networkx.algorithms.community.label_propagation import
    asyn_lpa_communities
#The asynchronous fluid communities algorithm
from networkx.algorithms.community.asyn_fluid import asyn_fluidc
#Girvan-Newman
from networkx.algorithms.community.centrality import
    girvan_newman
```

Some functions utilized in this project:

```
#This function takes the membership list and attribute as a
    parameter
```

```
#Adds the membership values to each vertex with given name
def attributer(benchmark, com_list, attr_name):
    z = 1
    for i in com_list:
        a = sorted(i)
        for j in a:
            benchmark.vs[(int(j[1:]))][attr_name] = z
        z = z + 1
    return;

#This function brings the vertex names from corresponding community
def get_comm(benchmark, attr_name, index):
    vs, comms, ccom, j = [],[],[],1
    for i in range(len(benchmark.vs)):
        if benchmark.vs[i][attr_name]:
            if benchmark.vs[i][attr_name] is not None:
                if math.isnan(benchmark.vs[i][attr_name]) == False:
                    vs.append(int(benchmark.vs[i][attr_name]))
    for j in range(1,max(vs)+1):
        for i in range(len(benchmark.vs)):
            if benchmark.vs[i][attr_name] == j: ccom.append(int(benchmark.vs[i]["name"]))
        comms.append(sorted(ccom))
        ccom = []
    return(comms[index])

def get_comms(benchmark, attr_name):
    vs, comms, ccom, j = [],[],[],1
    for i in range(len(benchmark.vs)):
        if benchmark.vs[i][attr_name]:
            if benchmark.vs[i][attr_name] is not None:
                if math.isnan(benchmark.vs[i][attr_name]) == False:
                    vs.append(int(benchmark.vs[i][attr_name]))
    for j in range(1,max(vs)+1):
        for i in range(len(benchmark.vs)):
            if benchmark.vs[i][attr_name] == j: ccom.append(int(benchmark.vs[i]["name"]))
        if ccom != []:
            comms.append(sorted(ccom))
            ccom = []
    return(comms)

#This function returns the average of non-zero values in the matrix
def nonzeroavg(m):
    total, index, lmax = 0, 0, 0
```

```
45      for i in m:
46          for j in i:
47              if j != 0:
48                  if j > float(lmax):
49                      lmax = j
50                      total = total + j
51                      index = index + 1
52          lmax = 0
53      if index != 0:
54          avg = total/index
55      else: avg = 0
56      return avg;
57
58  #This function returns the sequence matching ratio matrix
59  '''ratio() returns a float in [0, 1], measuring the similarity of
        the sequences.
60  As a rule of thumb, a ratio() value over 0.6 means the sequences
        are close matches:'''
61  def seq_match(a, b):
62      res, r = [], []
63      for i in a:
64          for j in b:
65              s = difflib.SequenceMatcher(None, i, j)
66              r.append(float("{0:.3f}".format(s.ratio())))
67          res.append(r)
68          r = []
69      return res
70
71  #Clique Percolation method
72  #This method is taken from a github project
73  #https://gist.github.com/conradlee/1341933#file-
        clique_percolation-py
74  def get_percolated_cliques(G, k):
75      perc_graph = nx.Graph()
76      cliques = list(frozenset(c) for c in nx.find_cliques(G) if
        len(c) >= k)
77      perc_graph.add_nodes_from(cliques)
78
79      # Add an edge in the clique graph for each pair of cliques
        that percolate
80      for c1, c2 in itertools.combinations(cliques, 2):
81          if len(c1.intersection(c2)) >= (k - 1):
82              perc_graph.add_edge(c1, c2)
83
84      for component in nx.connected_components(perc_graph):
85          yield(frozenset.union(*component))
86
87  #Maximum ratio is the first element in the output
88  #Printing the first n elements of the list - highest n ratios.
```

```
89 #Print the indexes of the n highest ratio values - from highest
       to lowest
90 """Params:
91 @ratio: the ratio matrix for first and second clustering to
       compare
92 @n    : number of values to print
93 @f    : first clustering
94 @s    : second clustering
95 """
96 def getHighestRatios(ratio, n, f, s, lmtr):
97     m_index = []
98     lastval = []
99     for i in sorted(np.partition(np.asarray(ratio).flatten(), -2)
       [-n:], reverse = True):
100         m_ = [(index, row.index(i)) for index, row in enumerate(
       ratio) if i in row]
101         if m_ != lastval:
102             m_index = m_index + m_
103         lastval = m_
104     rList = sorted(np.partition(np.asarray(ratio).flatten(), -2)
       [-n:], reverse = True)
105     printStuff(m_index, rList, f, s, lmtr)
106     return;
107
108 def getHighestRatioList(ratio, n):
109     return sorted(np.partition(np.asarray(ratio).flatten(), -2)[-
       n:], reverse = True);
110
111 def printStuff(m_index, rList, f, s, lmtr):
112     for i,j in zip(rList, m_index):
113         if i > lmtr:
114             print("Sequental matching ratio: %f" % i)
115             print("Index: %d - %d" % j)
116             print(s[j[1]])
117             print(f[j[0]])
118             print("----------------------")
119     return;
120
121 sprtr = "-----------------------------------------------"
122
123 def convertStringList(comms):
124     a = []
125     for i in range(len(comms)):
126         b = []
127         for j in range(len(comms[i])):
128             c = []
129             for z in range(len(comms[i][j])):
130                 c.append(int(comms[i][j][z]))
131             b.append(c)
```

```
                c = []
            a.append(b)
            b = []
        return comms;
```

Graph analysis and community detection results' comparison takes place in this part of the code:

```
#Read the graph with networkX
merged = nx.read_graphml("merged.graphml")
#Graph summary
print(nx.info(merged))
#Community structure information
cia, cib = nx.get_node_attributes(merged, "Cluster_ID_A"), nx.
    get_node_attributes(merged, "Cluster_ID_B")
cia, cib = list(cia.values()), list(cib.values())
print("Number of communities found in %s com A: %s"% ("merged",
    max(cia)))
print("Number of communities found in %s com B: %s"% ("merged",
    max(cib)))
#Clique Percolation Method
##k = size of the clique
k = 3
n_kcliq_3 = list(k_clique_communities(nx.DiGraph.to_undirected(
    merged), k))
print("Number of communities found in %s: %s"% ("merged",len(
    n_kcliq_3)))
for j in n_kcliq_3:
    print(sorted(j))
k = 4
n_kcliq_4 = list(k_clique_communities(nx.DiGraph.to_undirected(
    merged), k))
print("Number of communities found in %s: %s"% ("merged",len(
    n_kcliq_4)))
for j in n_kcliq_4:
    print(sorted(j))
#Fast-Greedy Method
n_fastgreedy = list(greedy_modularity_communities(nx.DiGraph.
    to_undirected(merged)))
print("Number of communities found in %s: %s"% ("merged", len(
    n_fastgreedy)))
for j in n_fastgreedy:
    print(sorted(j))
#Label Propagation Method
n_labelprop_gen = label_propagation_communities(nx.DiGraph.
    to_undirected(merged))
for j in n_labelprop_gen:
    n_labelprop.append(list(j))
```

```
31 print("Number of communities found in %s: %s"% ("merged", len(
       n_labelprop)))
32 print(n_labelprop)
33 #Asyndchronous Fluid Community Detection Method
34 #First run on #communities_A
35 n_asynfluid_a = list(asyn_fluidc(nx.DiGraph.to_undirected(merged)
       , 3))
36 print("Number of communities found in %s: %s"% ("merged", len(
       n_asynfluid_a)))
37 for j in n_a:
38     print(n_asynfluid_a(j))
39 #Second run on #communities_B
40 n_asynfluid_b = list(asyn_fluidc(nx.DiGraph.to_undirected(merged)
       , 8))
41 print("Number of communities found in %s: %s"% ("merged", len(
       n_asynfluid_b)))
42 for j in n_asynfluid_b:
43     print(sorted(j))
44 #Last run on #communities_A + #communities_B
45 n_asynfluid_ab = list(asyn_fluidc(nx.DiGraph.to_undirected(merged
       ), 11))
46 print("Number of communities found in %s: %s"% ("merged", len(
       n_asynfluid_ab)))
47 for j in n_asynfluid_ab:
48     print(sorted(j))
49 #Asynchronous Clique Percolation, same as CPM
50 a_perccliq_gen = get_percolated_cliques(nx.DiGraph.to_undirected(
       merged), 3)
51 for j in a_perccliq_gen:
52     a_perccliq_3 = []
53     a_perccliq_3.append(list(j))
54 print("Number of communities found in %s: %s"% (merged, len(
       sorted(a_perccliq_3))))
55 print(a_perccliq_4)
56 a_perccliq_gen = get_percolated_cliques(nx.DiGraph.to_undirected(
       merged), 3)
57 for j in a_perccliq_gen:
58     a_perccliq_4 = []
59     a_perccliq_4.append(list(j))
60 print("Number of communities found in %s: %s"% (merged, len(
       sorted(a_perccliq_4))))
61 print(a_perccliq_4)
62 #Girvan Newman Method
63 k = 2
64 comp = girvan_newman(merged)
65 comms = []
66 for communities in itertools.islice(comp, k):
67     comms.append(list(communities))
```

```
68 print("Number of communities found in %s: %s"% ("merged", len(
       comms)))
69 print(comms)
70 #Read igraph format
71 merged = ig.read("merged.graphml", format="graphml")
72 #Results are added to vertices of the graph as attributes
73 attributer(merged, n_kcliq_3, "nkcliq3")
74 attributer(merged, n_kcliq_4, "nkcliq4")
75 attributer(merged, n_fastgreedy, "nfastgreedy")
76 attributer(merged, n_labelprop, "nlabelprop")
77 attributer(merged, n_asynfluid_a, "nasynfluida")
78 attributer(merged, n_asynfluid_b, "nasynfluidb")
79 attributer(merged, n_asynfluid_ab, "nasynfluidab")
80 attributer(merged, a_percclig_3, "apercclig3")
81 attributer(merged, a_percclig_4, "apercclig4")
82 #Save graphs
83 merged.write_graphml("merged.graphml")
84 merged.write_gml("merged.gml")
85 #Print membership values
86 print("%s graph community memberships" % "merged")
87 vs_A = []
88 for j in range(len(merged.vs)):
89     vs_A.append(int(merged.vs[j]['Cluster_ID_A']))
90 print(vs_A)
91 vs_B = []
92 for j in range(len(merged.vs)):
93     vs_B.append(int(merged.vs[j]['Cluster_ID_B']))
94 print(vs_B)
95 #Merge two community structures to one
96 c_A, c_B = get_comms(gs[i], "Cluster_ID_A"), get_comms(gs[i], "
       Cluster_ID_B")
97 c_A = c_A + c_B
98 print("Communities:")
99 print(c_A)
100 com_A.append(c_A)
101 #Print ground truth communities
102 print("%s graph communities" % merged)
103 for j in range(len(com_A)):
104     print("Nodes in community membership %d:" % (j + 1))
105     print(com_A[j])
106 #Labelprop community detection ratios
107 com_labelprop = get_comms(merged, "comm_labelprop")
108 print("Number of communities in %s graph: %s" % ("merged", len(
       com_labelprop)))
109 print("Labelprop Communities:")
110 print(sorted(com_labelprop))
111 r_A_labelprop = seq_match(com_A, com_labelprop)
112 print(r_A_labelprop)
113 plt.figure(figsize=(8,8))
```

```
114 plt.imshow(r_A_labelprop, cmap=plt.cm.Purples, interpolation='
    nearest')
115 plt.show()
116 a_A_labelprop = nonzeroavg(r_A_labelprop)
117 print("Non-zero values average: %f" % a_A_labelprop)
118 print("graph: %s" % gnames[i])
119 getHighestRatios(r_A_labelprop, 25, com_A, get_comms(merged, "
    comm_labelprop"), 0.4)
120 #Spinglass community detection ratios
121 com_spinglass = get_comms(merged, "comm_spinglass")
122 print("Number of communities in %s graph: %s" % ("merged", len(
    com_spinglass)))
123 print("Spinglass Communities:")
124 print(sorted(com_spinglass)
125 r_A_spinglass = seq_match(com_A, com_spinglass)
126 print(r_A_spinglass)
127 plt.figure(figsize=(8,8))
128 plt.imshow(r_A_spinglass, cmap=plt.cm.Purples, interpolation='
    nearest')
129 plt.show()
130 a_A_spinglass = nonzeroavg(r_A_spinglass)
131 print("Non-zero values average: %f" % a_A_spinglass)
132 print("graph: %s" % merged)
133 getHighestRatios(r_A_spinglass, 25, com_A, get_comms(merged, "
    comm_spinglass"), 0.4)
134 #Spinglass community detection ratios
135 com_spinglass = get_comms(merged, "comm_spinglass")
136 print("Number of communities in %s graph: %s" % ("merged", len(
    com_spinglass)))
137 print("Spinglass Communities:")
138 print(sorted(com_spinglass)
139 r_A_spinglass = seq_match(com_A, com_spinglass)
140 print(r_A_spinglass)
141 plt.figure(figsize=(8,8))
142 plt.imshow(r_A_spinglass, cmap=plt.cm.Purples, interpolation='
    nearest')
143 plt.show()
144 a_A_spinglass = nonzeroavg(r_A_spinglass)
145 print("Non-zero values average: %f" % a_A_spinglass)
146 print("graph: %s" % merged)
147 getHighestRatios(r_A_spinglass, 25, com_A, get_comms(merged, "
    comm_spinglass"), 0.4)
148 #Walktrap community detection ratios
149 com_walktrap = get_comms(merged, "comm_walktrap")
150 print("Number of communities in %s graph: %s" % ("merged", len(
    com_walktrap)))
151 print("Walktrap Communities:")
152 print(sorted(com_walktrap)
153 r_A_walktrap = seq_match(com_A, com_walktrap)
```

```
154 print(r_A_walktrap)
155 plt.figure(figsize=(8,8))
156 plt.imshow(r_A_walktrap, cmap=plt.cm.Purples, interpolation='
    nearest')
157 plt.show()
158 a_A_walktrap = nonzeroavg(r_A_walktrap)
159 print("Non-zero values average: %f" % a_A_walktrap)
160 print("graph: %s" % merged)
161 getHighestRatios(r_A_walktrap, 25, com_A, get_comms(merged, "
    comm_walktrap"), 0.4)
162 #Louvain community detection ratios
163 com_louvain = get_comms(merged, "comm_louvain")
164 print("Number of communities in %s graph: %s" % ("merged", len(
    com_louvain)))
165 print("Louvain Communities:")
166 print(sorted(com_louvain)
167 r_A_louvain = seq_match(com_A, com_louvain)
168 print(r_A_louvain)
169 plt.figure(figsize=(8,8))
170 plt.imshow(r_A_louvain, cmap=plt.cm.Purples, interpolation='
    nearest')
171 plt.show()
172 a_A_louvain = nonzeroavg(r_A_louvain)
173 print("Non-zero values average: %f" % a_A_louvain)
174 print("graph: %s" % merged)
175 getHighestRatios(r_A_louvain, 25, com_A, get_comms(merged, "
    comm_louvain"), 0.4)
176 #Infomap community detection ratios
177 com_infomap = get_comms(merged, "comm_infomap")
178 print("Number of communities in %s graph: %s" % ("merged", len(
    com_infomap)))
179 print("Infomap Communities:")
180 print(sorted(com_infomap)
181 r_A_infomap = seq_match(com_A, com_infomap)
182 print(r_A_infomap)
183 plt.figure(figsize=(8,8))
184 plt.imshow(r_A_infomap, cmap=plt.cm.Purples, interpolation='
    nearest')
185 plt.show()
186 a_A_infomap = nonzeroavg(r_A_infomap)
187 print("Non-zero values average: %f" % a_A_infomap)
188 print("graph: %s" % merged)
189 getHighestRatios(r_A_infomap, 25, com_A, get_comms(merged, "
    comm_infomap"), 0.4)
190 #Asynfluid community detection ratios on com_A
191 com_asynfluid_a = get_comms(merged, "comm_asynfluid_a")
192 print("Number of communities in %s graph: %s" % ("merged", len(
    com_asynfluid_a)))
193 print("Asynchronous Fluid com_A Communities:")
```

```
194 print(sorted(com_asynfluid_a)
195 r_A_asynfluid_a = seq_match(com_A, com_asynfluid_a)
196 print(r_A_asynfluid_a)
197 plt.figure(figsize=(8,8))
198 plt.imshow(r_A_asynfluid_a, cmap=plt.cm.Purples, interpolation='
    nearest')
199 plt.show()
200 a_A_asynfluid_a = nonzeroavg(r_A_asynfluid_a)
201 print("Non-zero values average: %f" % a_A_asynfluid_a)
202 print("graph: %s" % merged)
203 getHighestRatios(r_A_asynfluid_a, 25, com_A, get_comms(merged, "
    comm_asynfluid_a"), 0.4)
204 #Asynfluid community detection ratios on com_B
205 com_asynfluid_b = get_comms(merged, "comm_asynfluid_b")
206 print("Number of communities in %s graph: %s" % ("merged", len(
    com_asynfluid_b)))
207 print("Asynchronous Fluid com_B Communities:")
208 print(sorted(com_asynfluid_b)
209 r_A_asynfluid_b = seq_match(com_A, com_asynfluid_b)
210 print(r_A_asynfluid_b)
211 plt.figure(figsize=(8,8))
212 plt.imshow(r_A_asynfluid_b, cmap=plt.cm.Purples, interpolation='
    nearest')
213 plt.show()
214 a_A_asynfluid_b = nonzeroavg(r_A_asynfluid_b)
215 print("Non-zero values average: %f" % a_A_asynfluid_b)
216 print("graph: %s" % merged)
217 getHighestRatios(r_A_asynfluid_b, 25, com_A, get_comms(merged, "
    comm_asynfluid_b"), 0.4)
218 #Asynfluid community detection ratios on com_A + com_B
219 com_asynfluid_ab = get_comms(merged, "comm_asynfluid_ab")
220 print("Number of communities in %s graph: %s" % ("merged", len(
    com_asynfluid_ab)))
221 print("Asynchronous Fluid com_A + com_B Communities:")
222 print(sorted(com_asynfluid_ab)
223 r_A_asynfluid_ab = seq_match(com_A, com_asynfluid_ab)
224 print(r_A_asynfluid_ab)
225 plt.figure(figsize=(8,8))
226 plt.imshow(r_A_asynfluid_ab, cmap=plt.cm.Purples, interpolation='
    nearest')
227 plt.show()
228 a_A_asynfluid_ab = nonzeroavg(r_A_asynfluid_ab)
229 print("Non-zero values average: %f" % a_A_asynfluid_ab)
230 print("graph: %s" % merged)
231 getHighestRatios(r_A_asynfluid_ab, 25, com_A, get_comms(merged, "
    comm_asynfluid_ab"), 0.4)
232 #Clique Percolation method detection ratios with k=3
233 com_n_kcliq3 = get_comms(merged, "comm_n_kcliq3")
```

```
234 print("Number of communities in %s graph: %s" % ("merged", len(
    com_n_kcliq3)))
235 print("Clique Percolation k=3 Communities:")
236 print(sorted(com_n_kcliq3)
237 r_A_n_kcliq3 = seq_match(com_A, com_n_kcliq3)
238 print(r_A_n_kcliq3)
239 plt.figure(figsize=(8,8))
240 plt.imshow(r_A_n_kcliq3, cmap=plt.cm.Purples, interpolation='
    nearest')
241 plt.show()
242 a_A_n_kcliq3 = nonzeroavg(r_A_n_kcliq3)
243 print("Non-zero values average: %f" % a_A_n_kcliq3)
244 print("graph: %s" % merged)
245 getHighestRatios(r_A_n_kcliq3, 25, com_A, get_comms(merged, "
    comm_n_kcliq3"), 0.4)
246 #Clique Percolation method detection ratios with k=4
247 com_n_kcliq4 = get_comms(merged, "comm_n_kcliq4")
248 print("Number of communities in %s graph: %s" % ("merged", len(
    com_n_kcliq4)))
249 print("Clique Percolation k=4 Communities:")
250 print(sorted(com_n_kcliq4)
251 r_A_n_kcliq4 = seq_match(com_A, com_n_kcliq4)
252 print(r_A_n_kcliq4)
253 plt.figure(figsize=(8,8))
254 plt.imshow(r_A_n_kcliq4, cmap=plt.cm.Purples, interpolation='
    nearest')
255 plt.show()
256 a_A_n_kcliq4 = nonzeroavg(r_A_n_kcliq4)
257 print("Non-zero values average: %f" % a_A_n_kcliq4)
258 print("graph: %s" % merged)
259 getHighestRatios(r_A_n_kcliq4, 25, com_A, get_comms(merged, "
    comm_n_kcliq4"), 0.4)
260 #Fast-Greedy method detection ratios
261 com_fastgreedy = get_comms(merged, "comm_fastgreedy")
262 print("Number of communities in %s graph: %s" % ("merged", len(
    com_fastgreedy)))
263 print("Fast-Greedy Communities:")
264 print(sorted(com_fastgreedy)
265 r_A_fastgreedy = seq_match(com_A, com_fastgreedy)
266 print(r_A_fastgreedy)
267 plt.figure(figsize=(8,8))
268 plt.imshow(r_A_fastgreedy, cmap=plt.cm.Purples, interpolation='
    nearest')
269 plt.show()
270 a_A_fastgreedy = nonzeroavg(r_A_fastgreedy)
271 print("Non-zero values average: %f" % a_A_fastgreedy)
272 print("graph: %s" % merged)
273 getHighestRatios(r_A_fastgreedy, 25, com_A, get_comms(merged, "
    comm_fastgreedy"), 0.4)
```

```
#(NetworkX) Label Propagation method detection ratios
com_n_labelprop = get_comms(merged, "comm_n_labelprop")
print("Number of communities in %s graph: %s" % ("merged", len(
    com_n_labelprop)))
print("Networkx Labelprop Communities:")
print(sorted(com_n_labelprop)
r_A_n_labelprop = seq_match(com_A, com_n_labelprop)
print(r_A_n_labelprop)
plt.figure(figsize=(8,8))
plt.imshow(r_A_n_labelprop, cmap=plt.cm.Purples, interpolation='
    nearest')
plt.show()
a_A_n_labelprop = nonzeroavg(r_A_n_labelprop)
print("Non-zero values average: %f" % a_A_n_labelprop)
print("graph: %s" % merged)
getHighestRatios(r_A_n_labelprop, 25, com_A, get_comms(merged, "
    comm_n_labelprop"), 0.4)
#Asynchronous Clique Percolation method detection ratios k=3
com_perccliq_3 = get_comms(merged, "comm_perccliq_3")
print("Number of communities in %s graph: %s" % ("merged", len(
    com_perccliq_3)))
print("Asynperqcliq k=3 Communities:")
print(sorted(com_perccliq_3)
r_A_perccliq_3 = seq_match(com_A, com_perccliq_3)
print(r_A_perccliq_3)
plt.figure(figsize=(8,8))
plt.imshow(r_A_perccliq_3, cmap=plt.cm.Purples, interpolation='
    nearest')
plt.show()
a_A_perccliq_3 = nonzeroavg(r_A_perccliq_3)
print("Non-zero values average: %f" % a_A_perccliq_3)
print("graph: %s" % merged)
getHighestRatios(r_A_perccliq_3, 25, com_A, get_comms(merged, "
    comm_perccliq_3"), 0.4)
#Asynchronous Clique Percolation method detection ratios k=4
com_perccliq_4 = get_comms(merged, "comm_perccliq_4")
print("Number of communities in %s graph: %s" % ("merged", len(
    com_perccliq_4)))
print("Asynperqcliq k=4 Communities:")
print(sorted(com_perccliq_4)
r_A_perccliq_4 = seq_match(com_A, com_perccliq_4)
print(r_A_perccliq_4)
plt.figure(figsize=(8,8))
plt.imshow(r_A_perccliq_4, cmap=plt.cm.Purples, interpolation='
    nearest')
plt.show()
a_A_perccliq_4 = nonzeroavg(r_A_perccliq_4)
print("Non-zero values average: %f" % a_A_perccliq_4)
print("graph: %s" % merged)
```

```
315 getHighestRatios(r_A_perccliq_4, 25, com_A, get_comms(merged, "
    comm_perccliq_4"), 0.4)
```

Graphs

The graph structures can be found in this section. Test graphs are presented in order; first original graph, second original graph and lastly the merged graph.

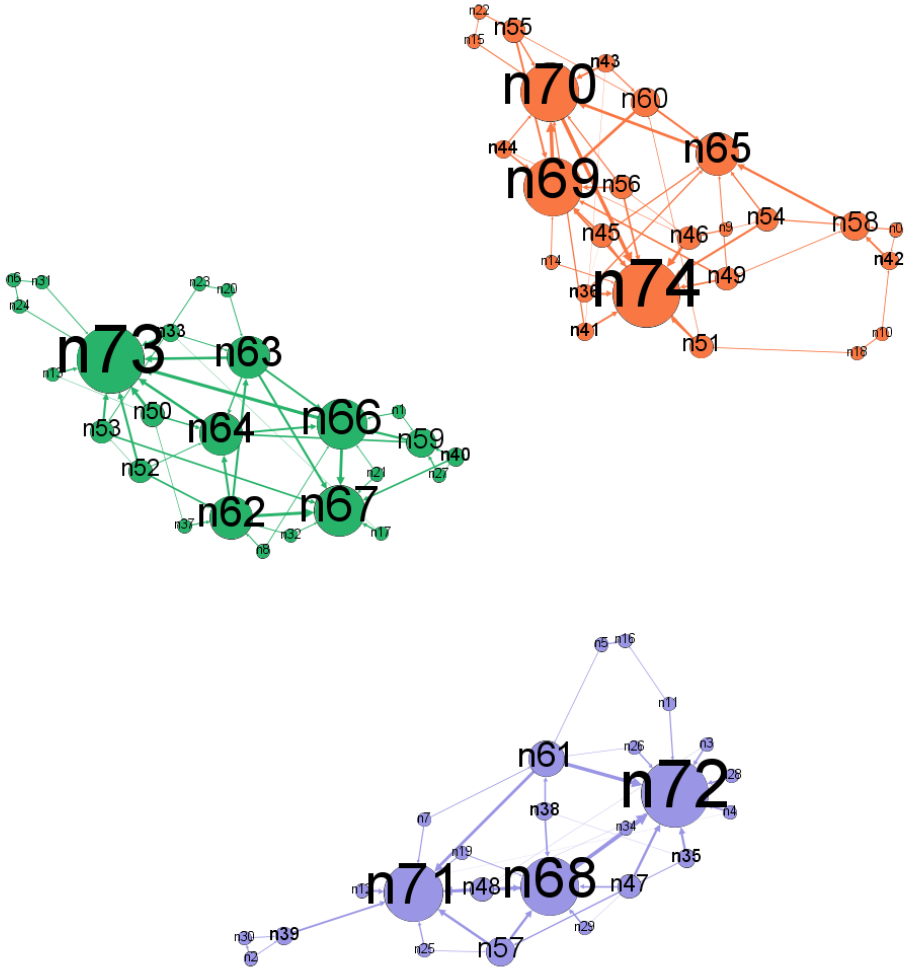


Figure 27: 1st original graph from test75

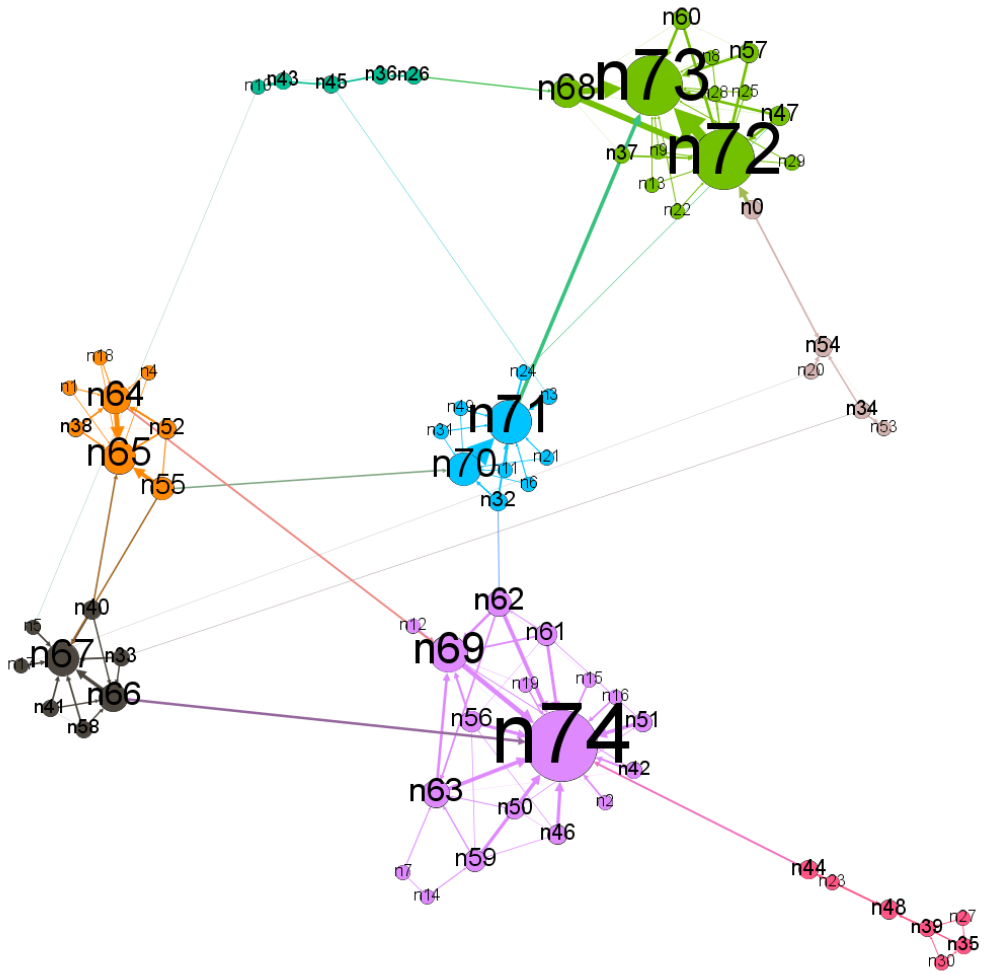


Figure 28: 2nd original graph from test75

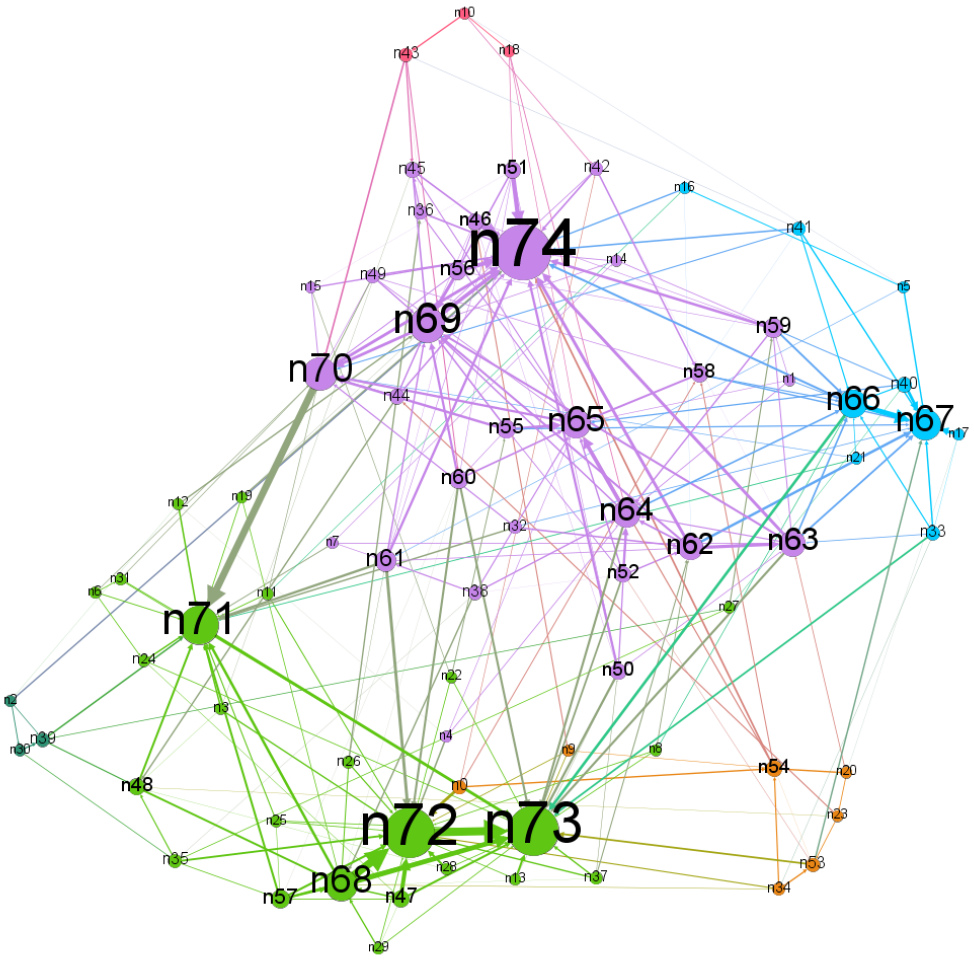


Figure 29: test75 merged graph with Walktrap results

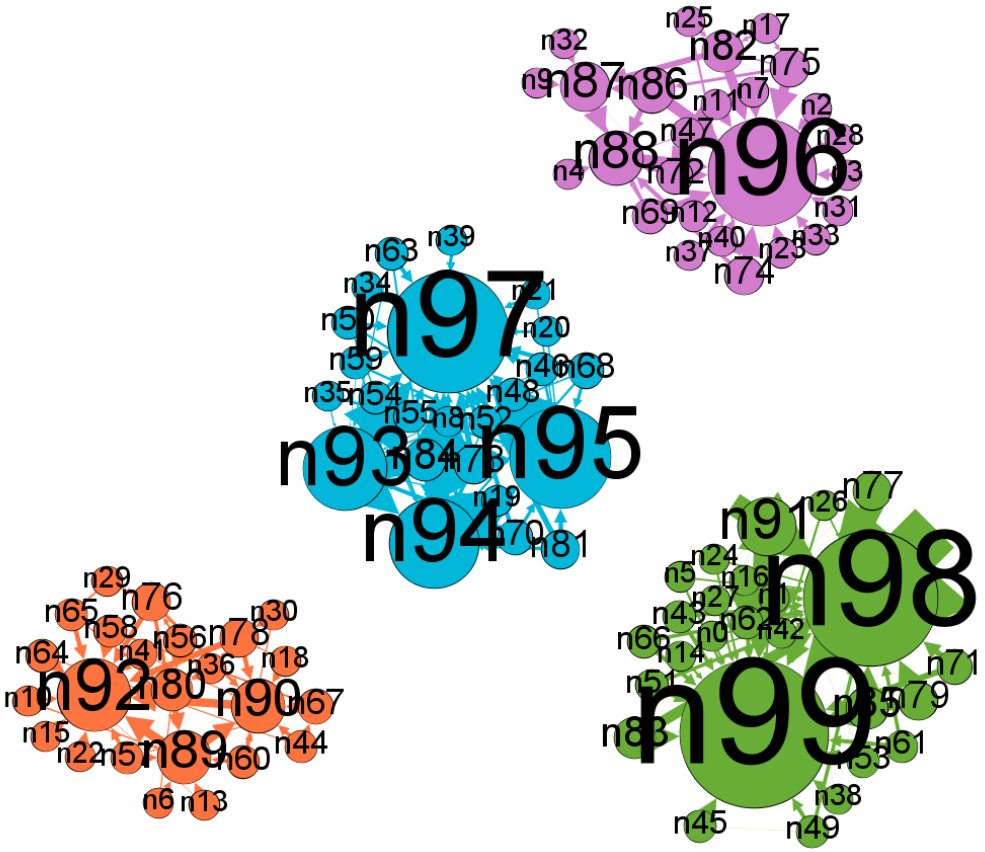


Figure 30: 1st original graph from test100

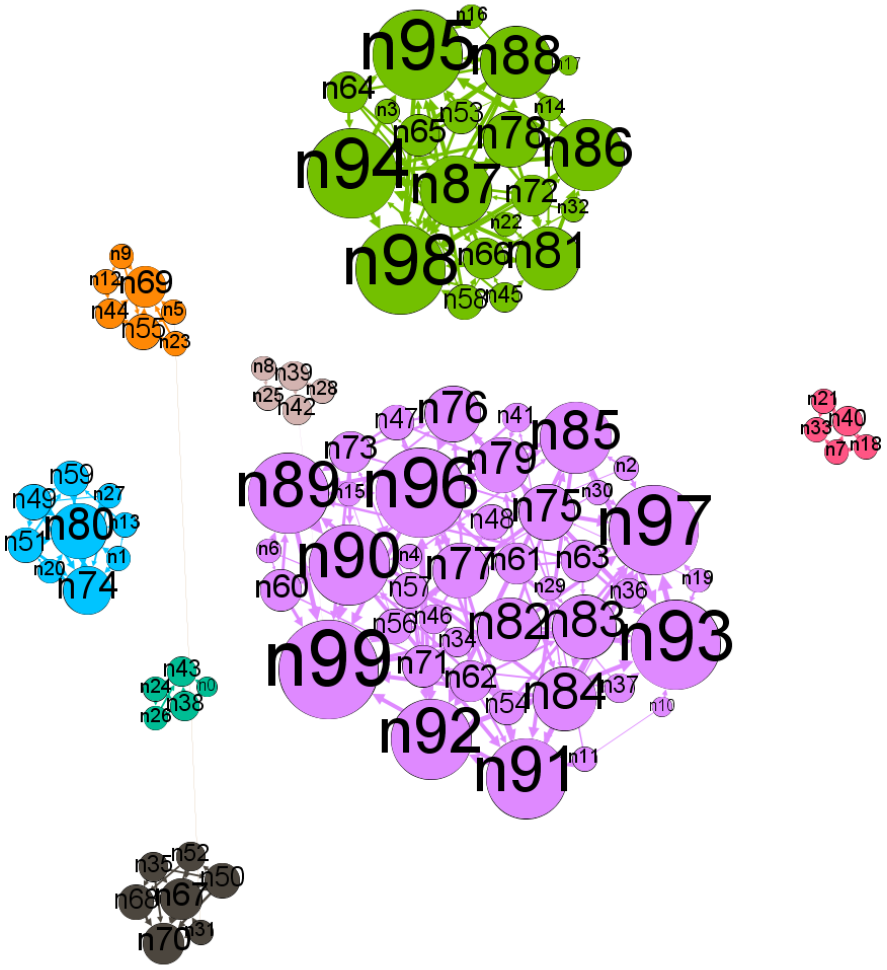


Figure 31: 2nd original graph from test100

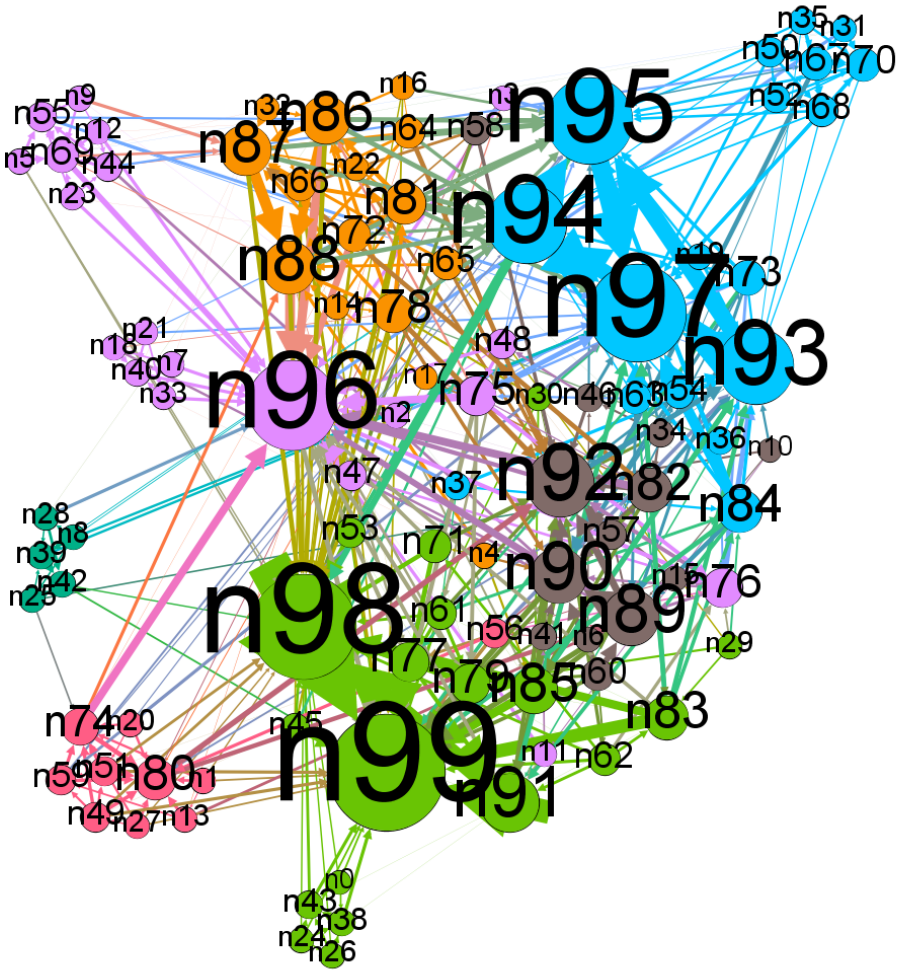


Figure 32: test100 merged graph with Louvain results

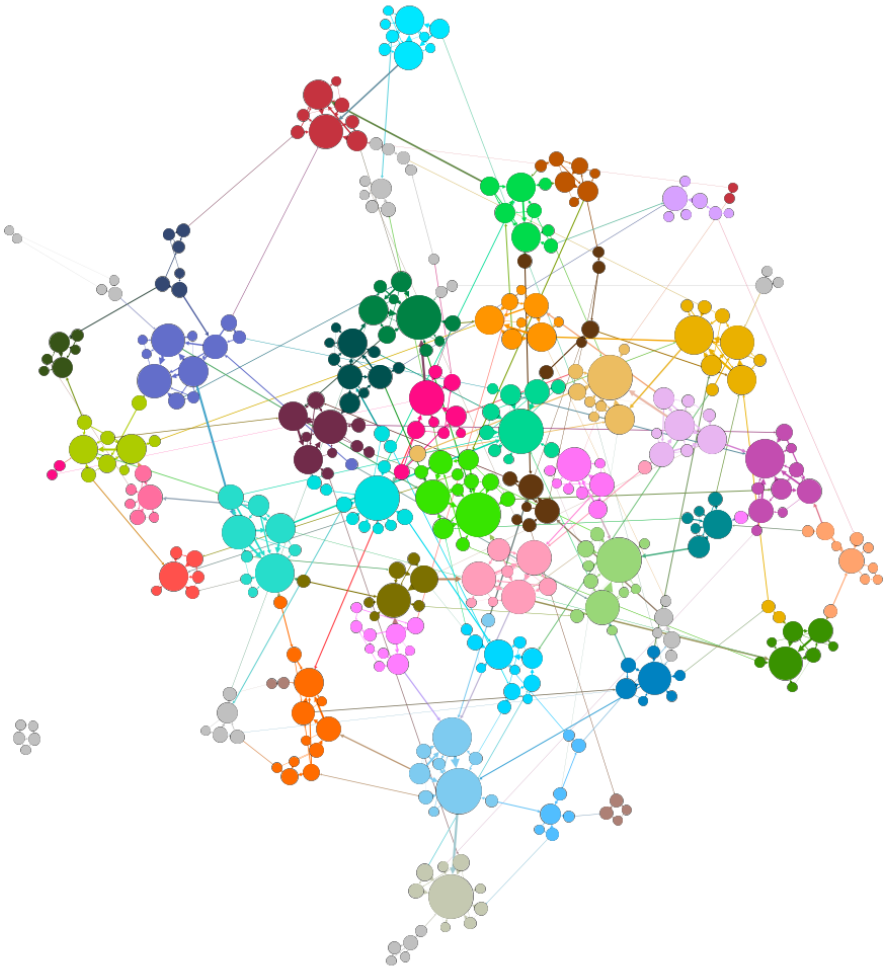


Figure 33: 1st original graph from test400

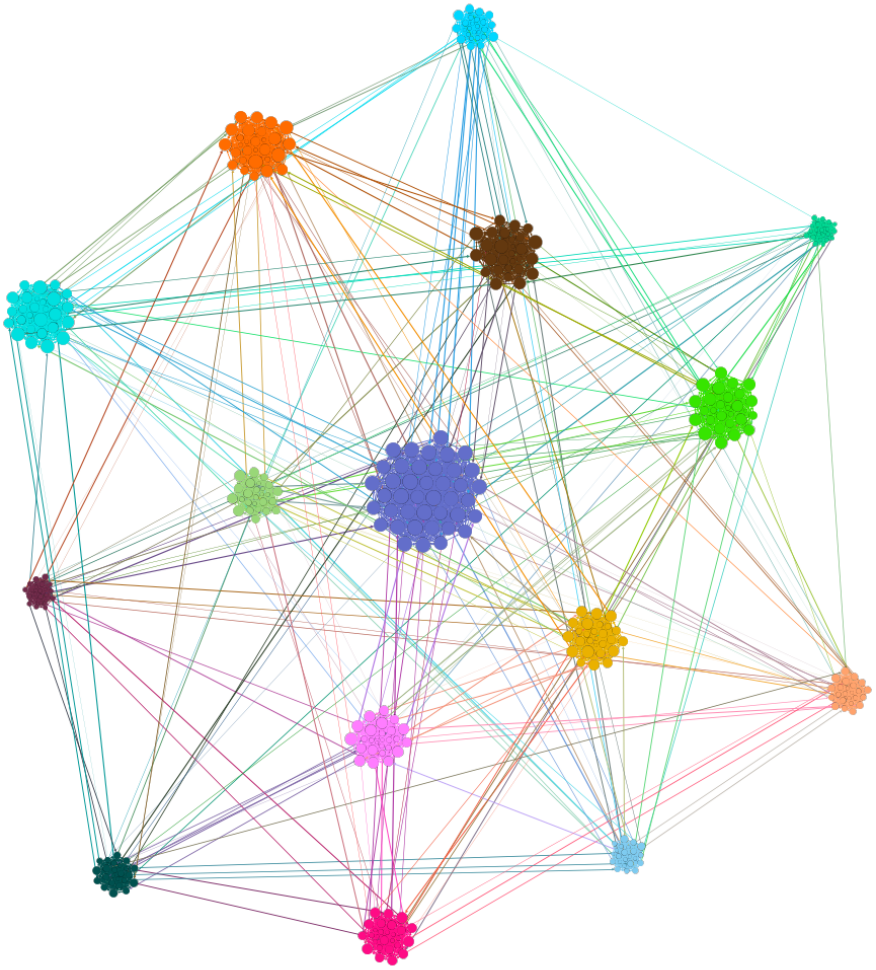


Figure 34: 2nd original graph from test400

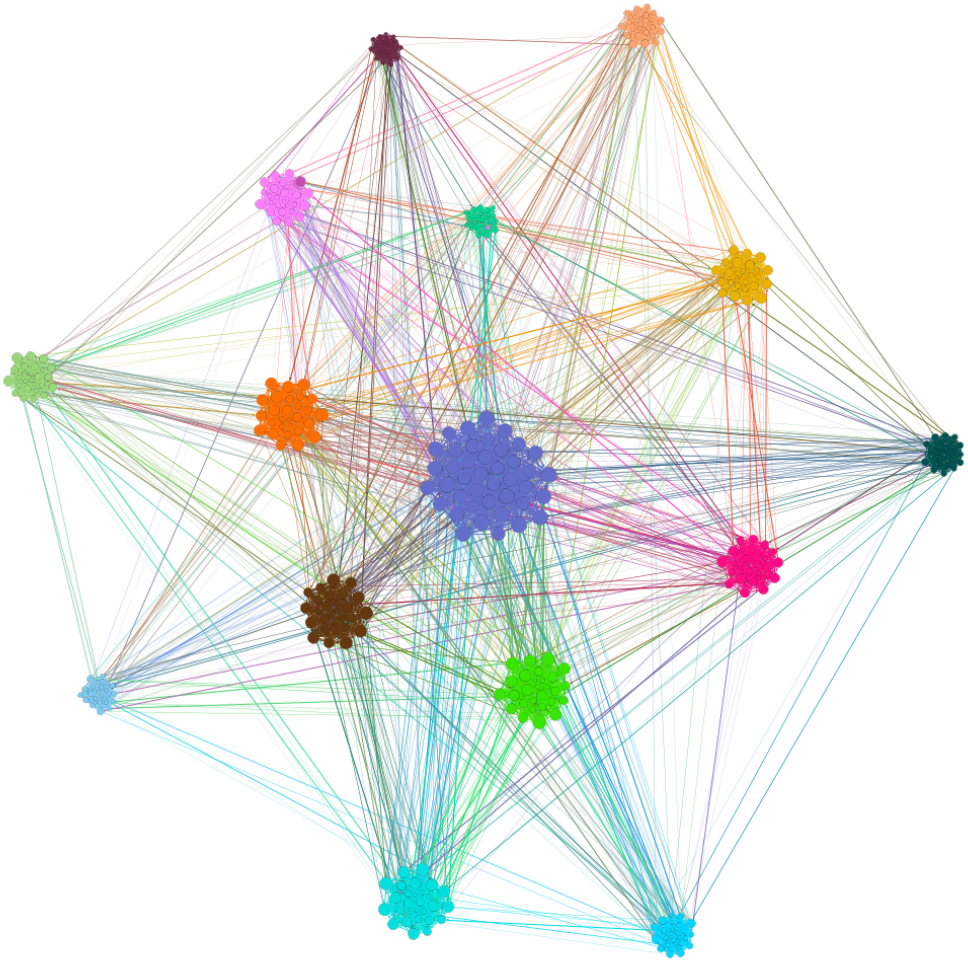


Figure 35: test400 merged graph with Spinglass results

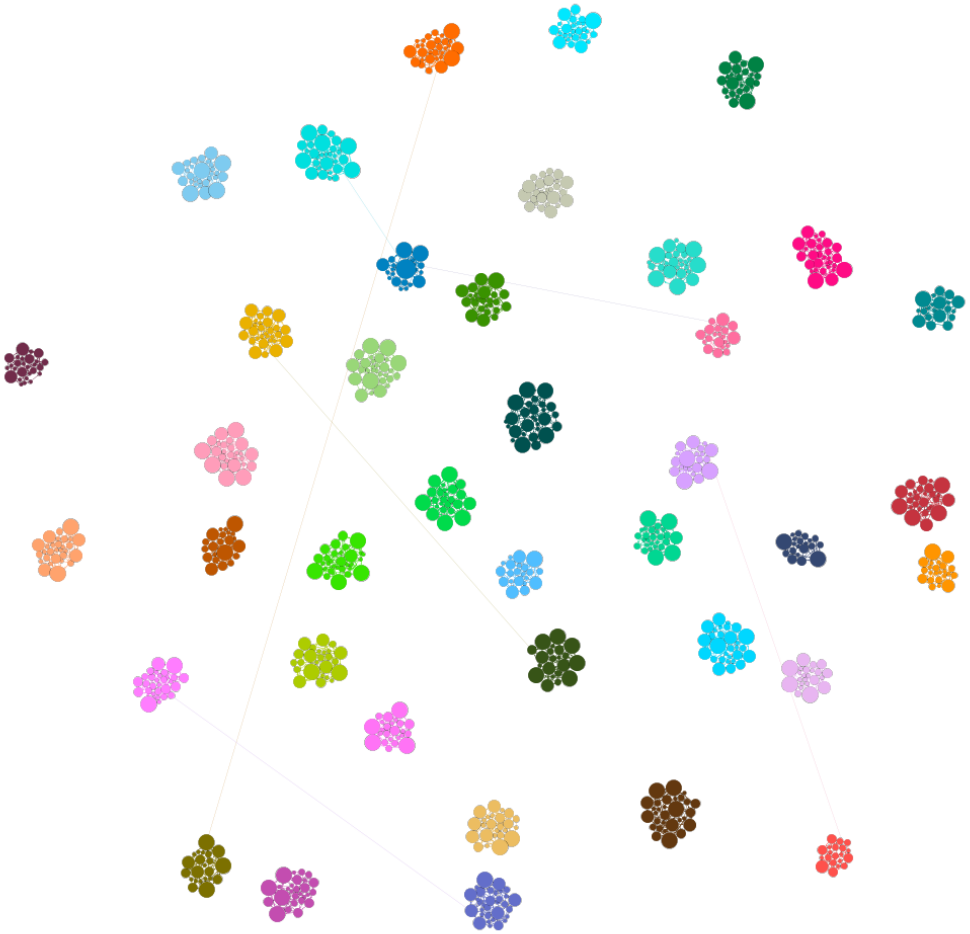


Figure 36: 1st original graph from test1000

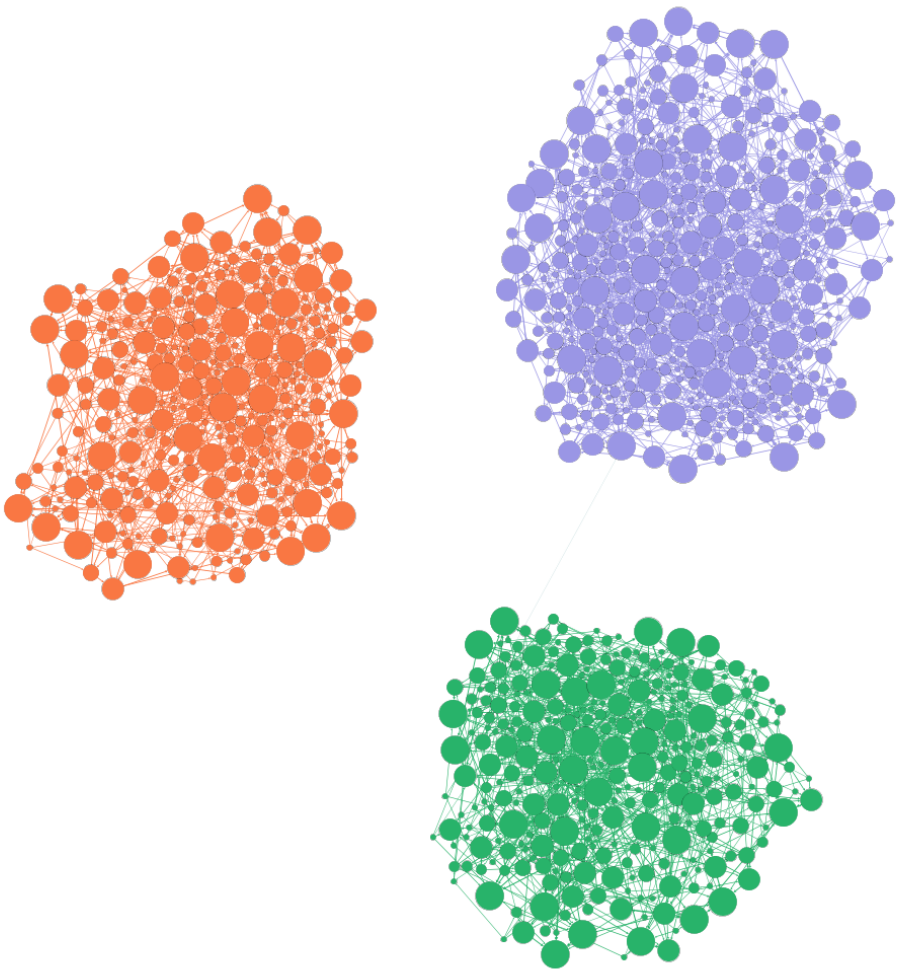


Figure 37: 2nd original graph from test1000

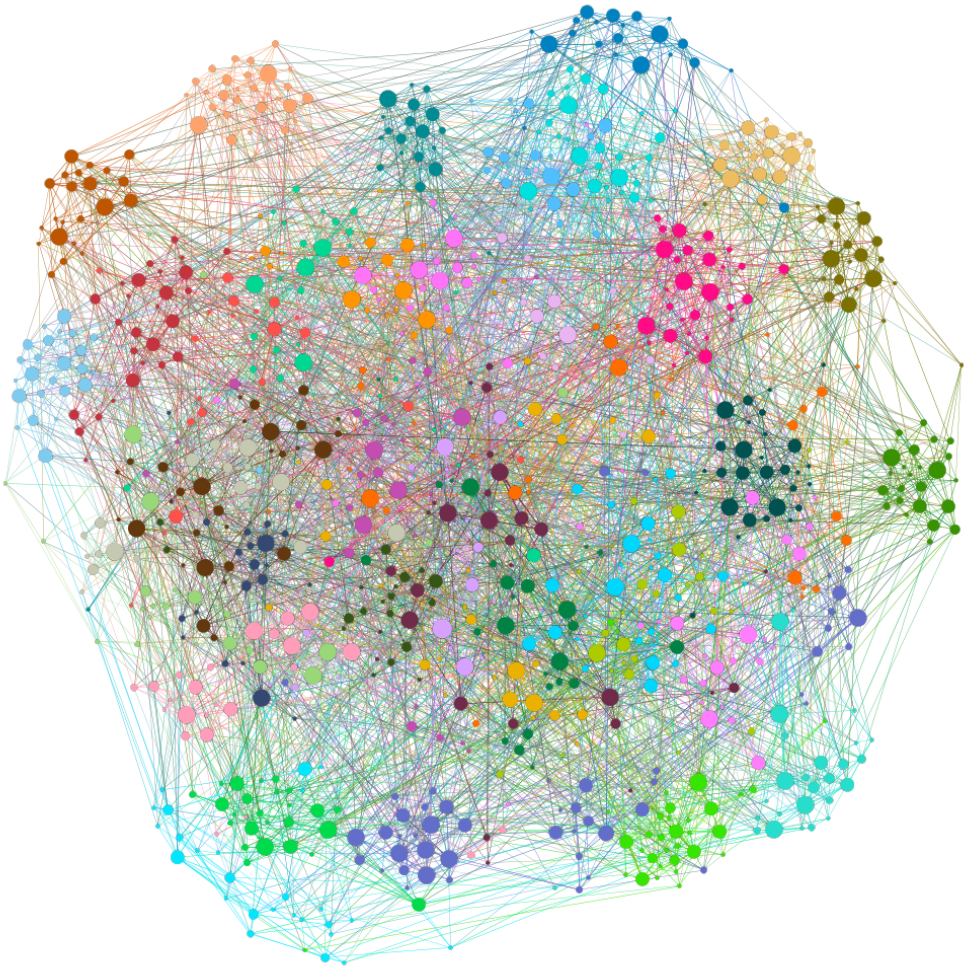


Figure 38: test1000 merged graph with Infomap results

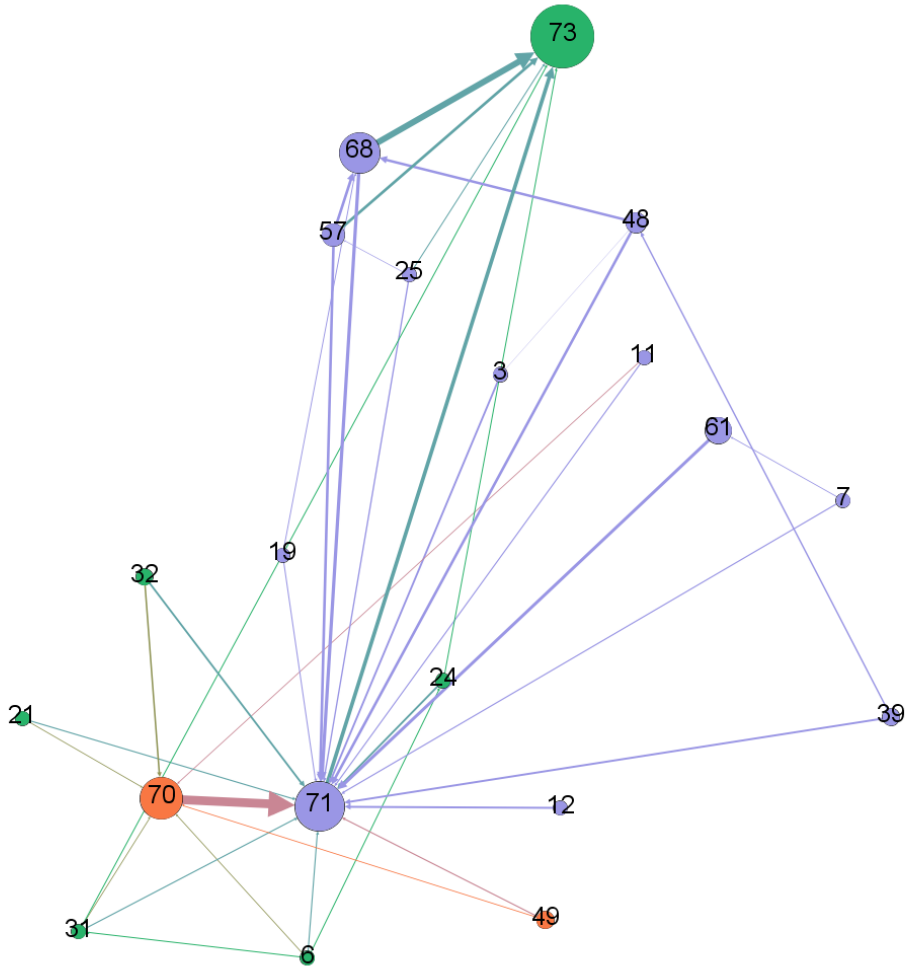


Figure 39: test75 ego graph of node with id 71

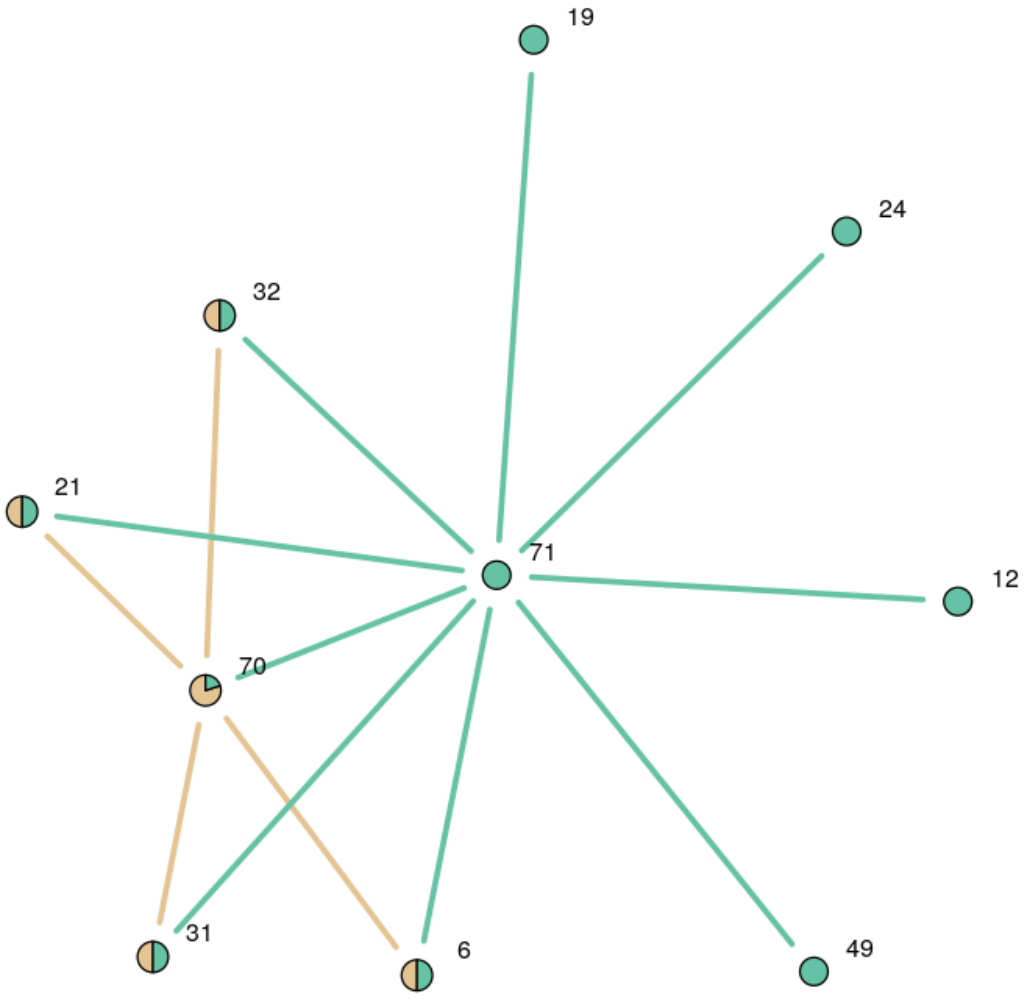


Figure 40: Linkcomm nested community on test75

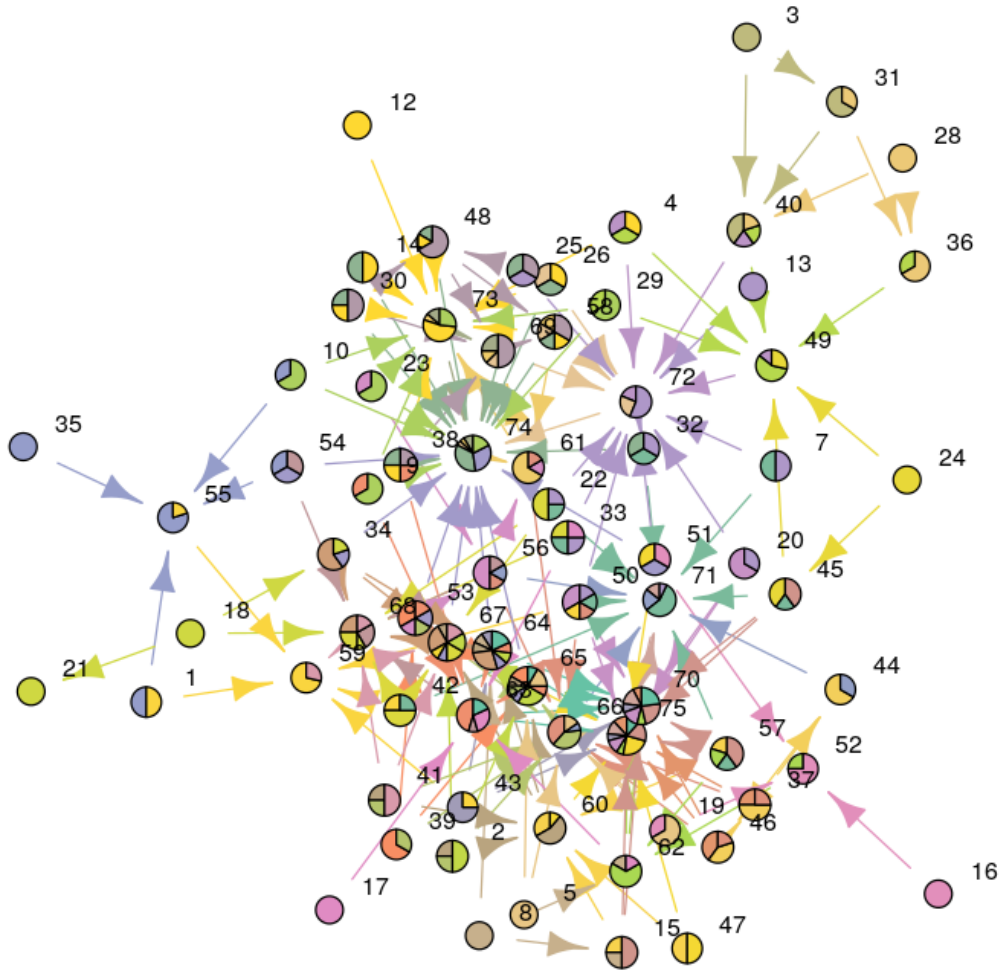


Figure 41: Linkcomm community detection on test75

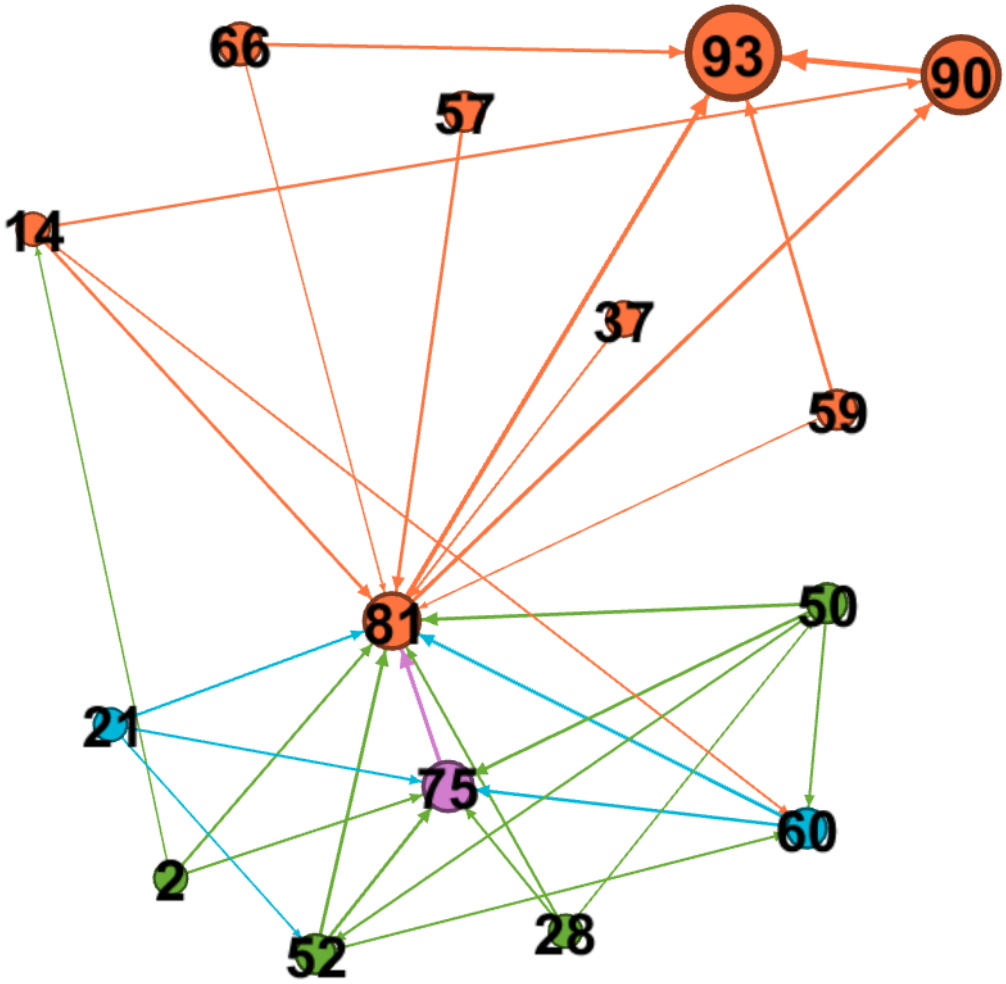


Figure 42: test100 ego graph of node with id 81

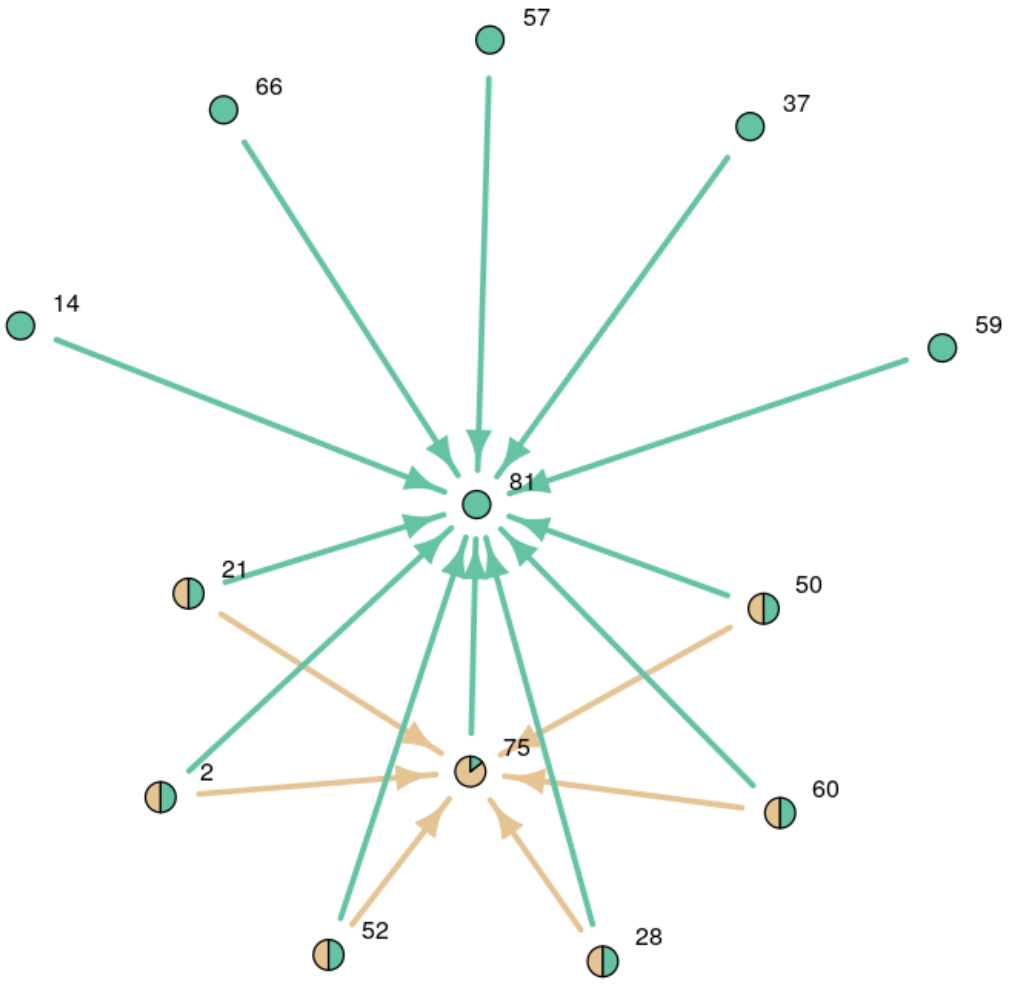


Figure 43: Linkcomm nested community on test100

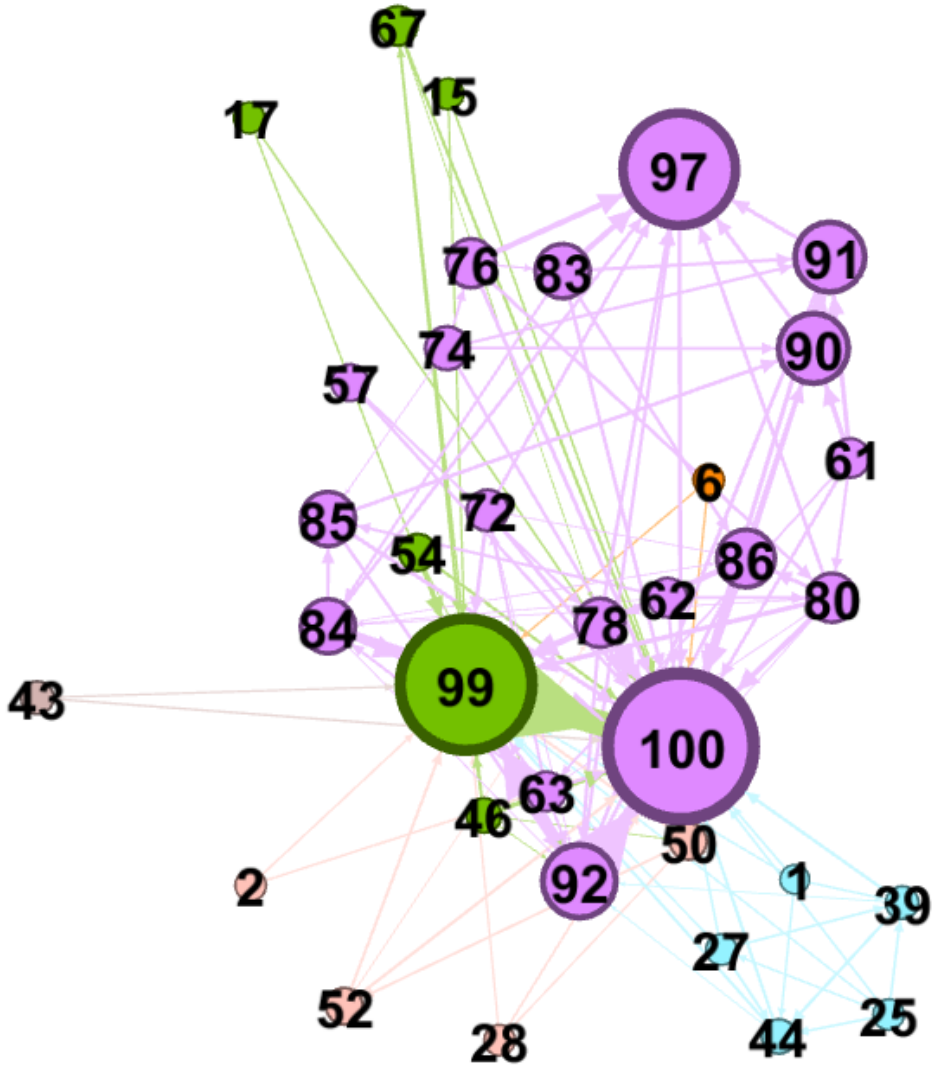


Figure 44: test100 ego graph of node with id 100

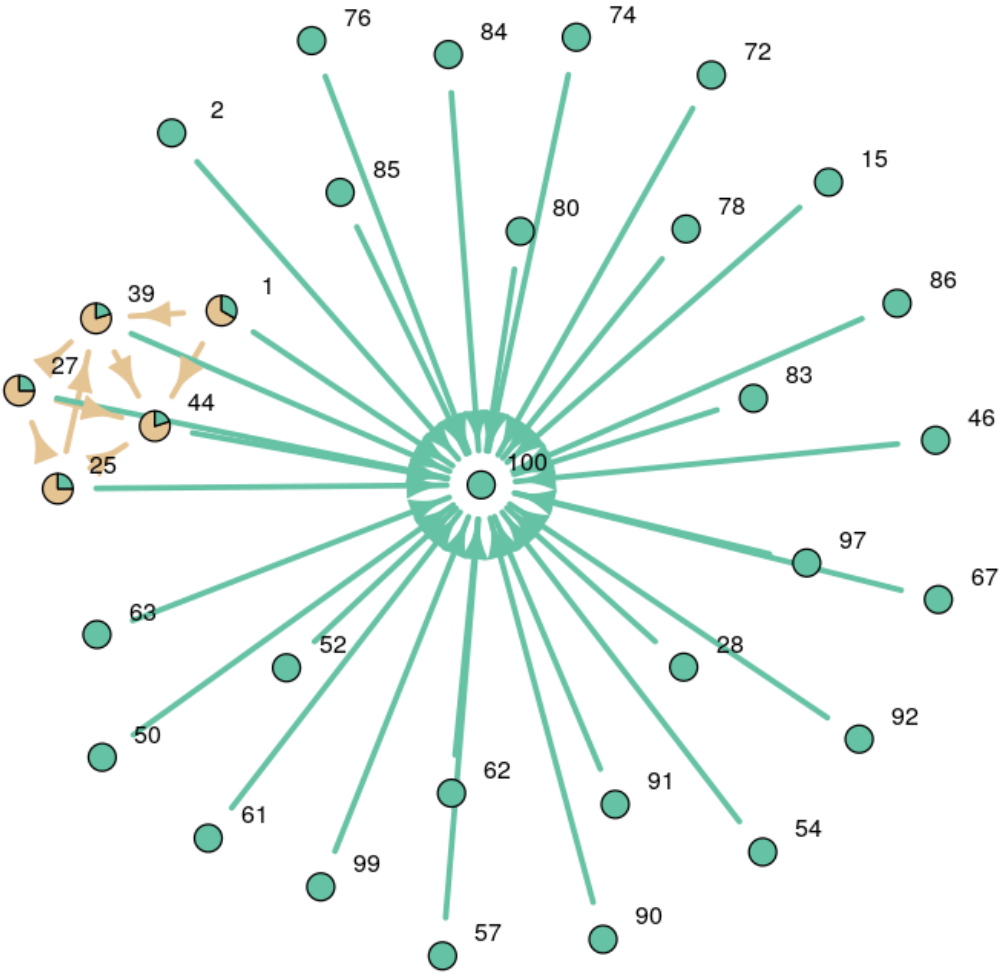


Figure 45: Linkcomm nested community on test100

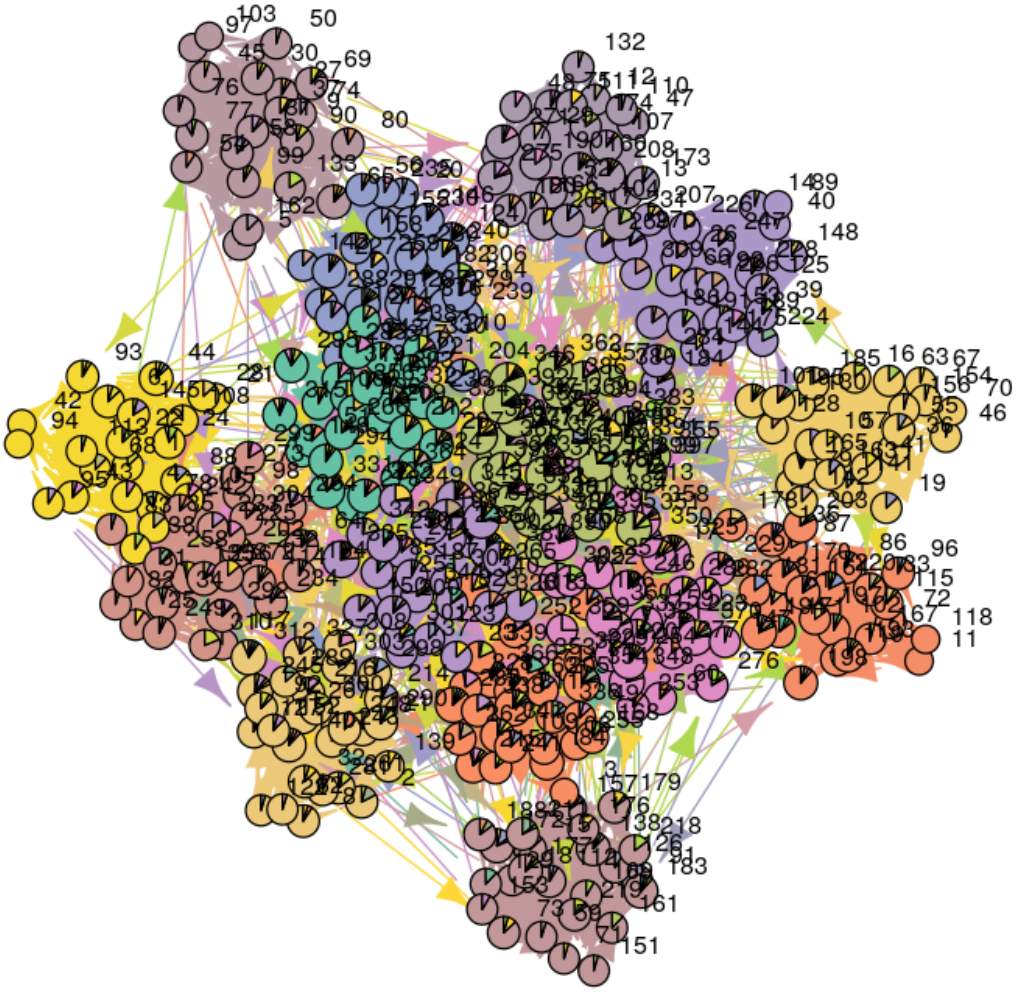


Figure 46: Linkcomm community detection on test400

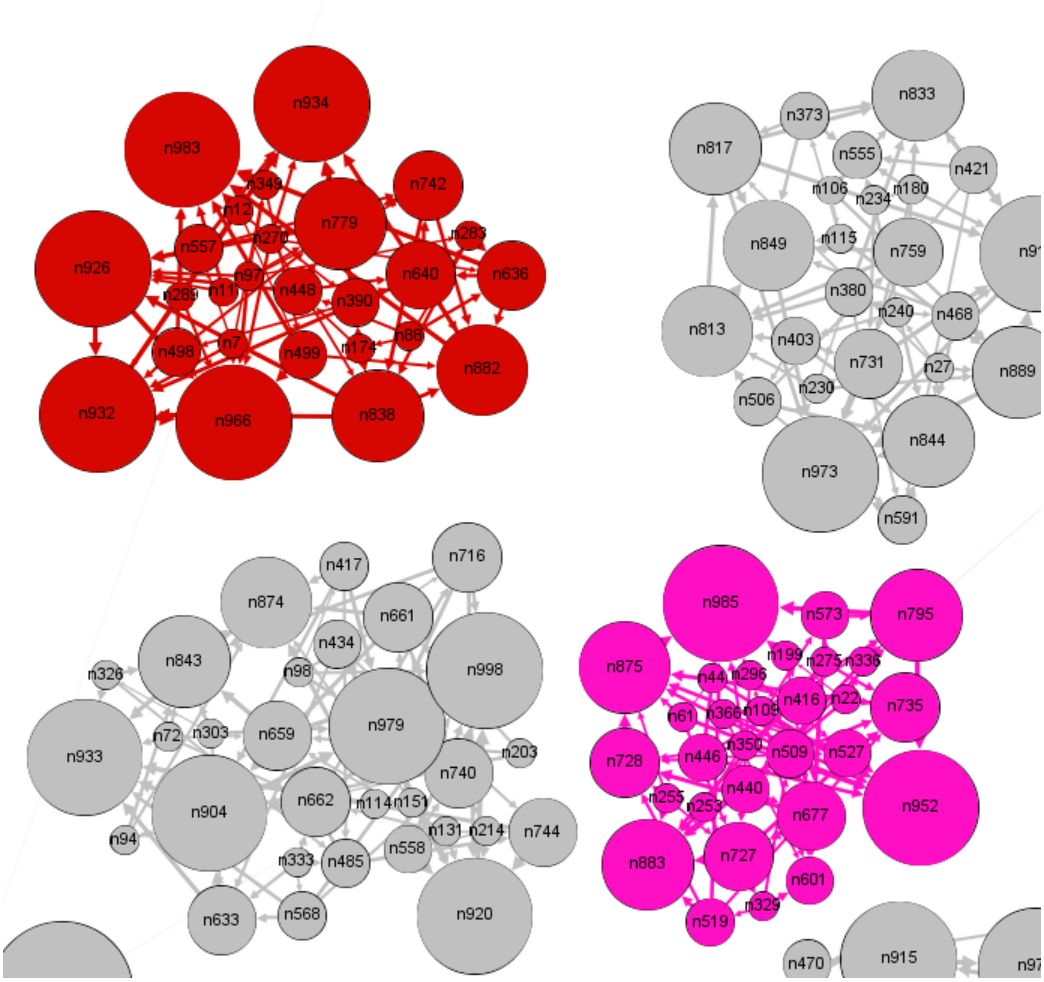


Figure 47: test1000 with 23th and 30th communities colored

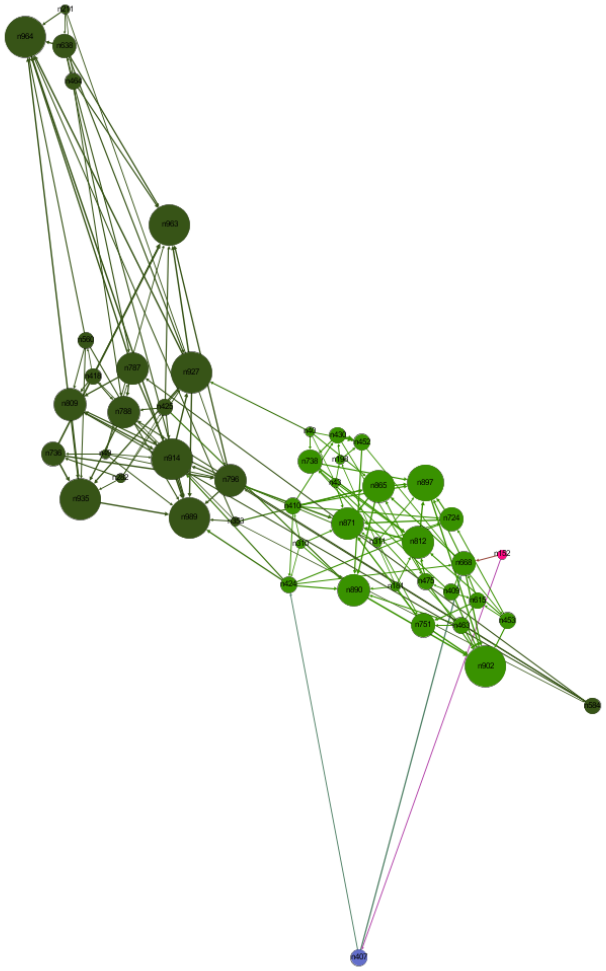
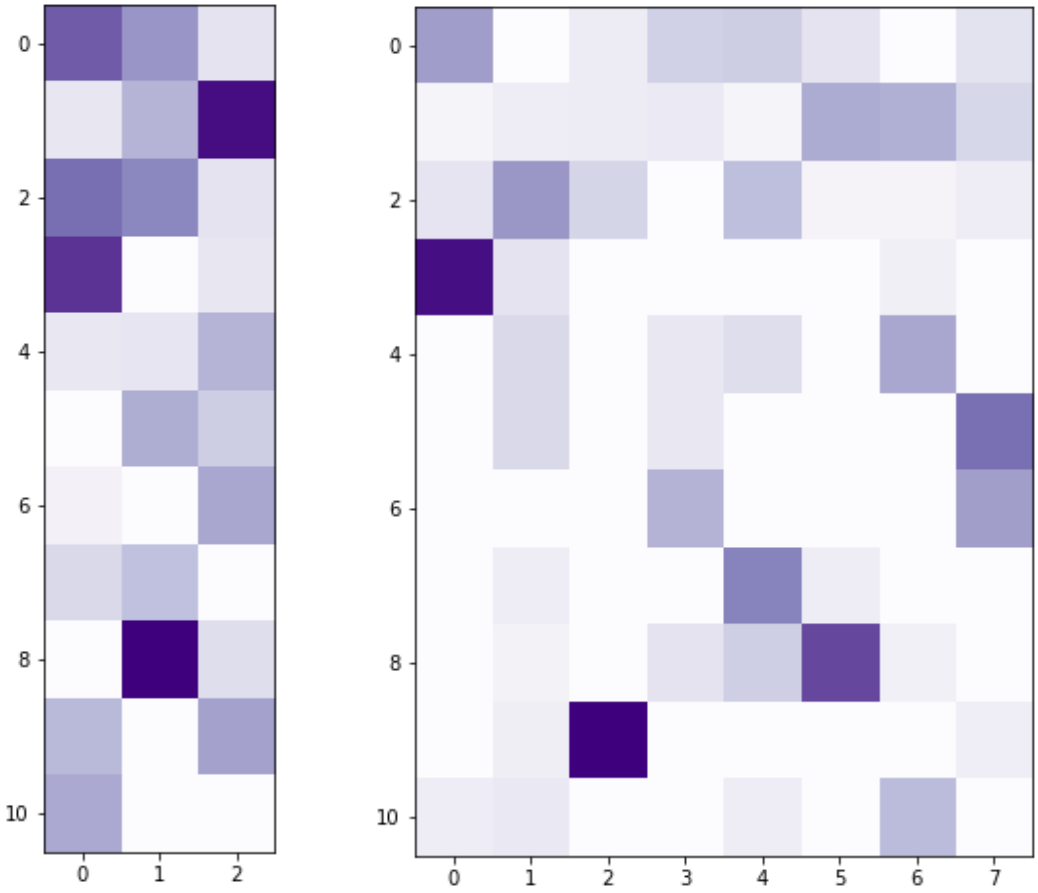


Figure 48: Walktrap detected community on test1000



(a) Heatmap of Async Fluid (k=a) on test75

(b) Heatmap of Asynchronous Fluid (k=b) on test75

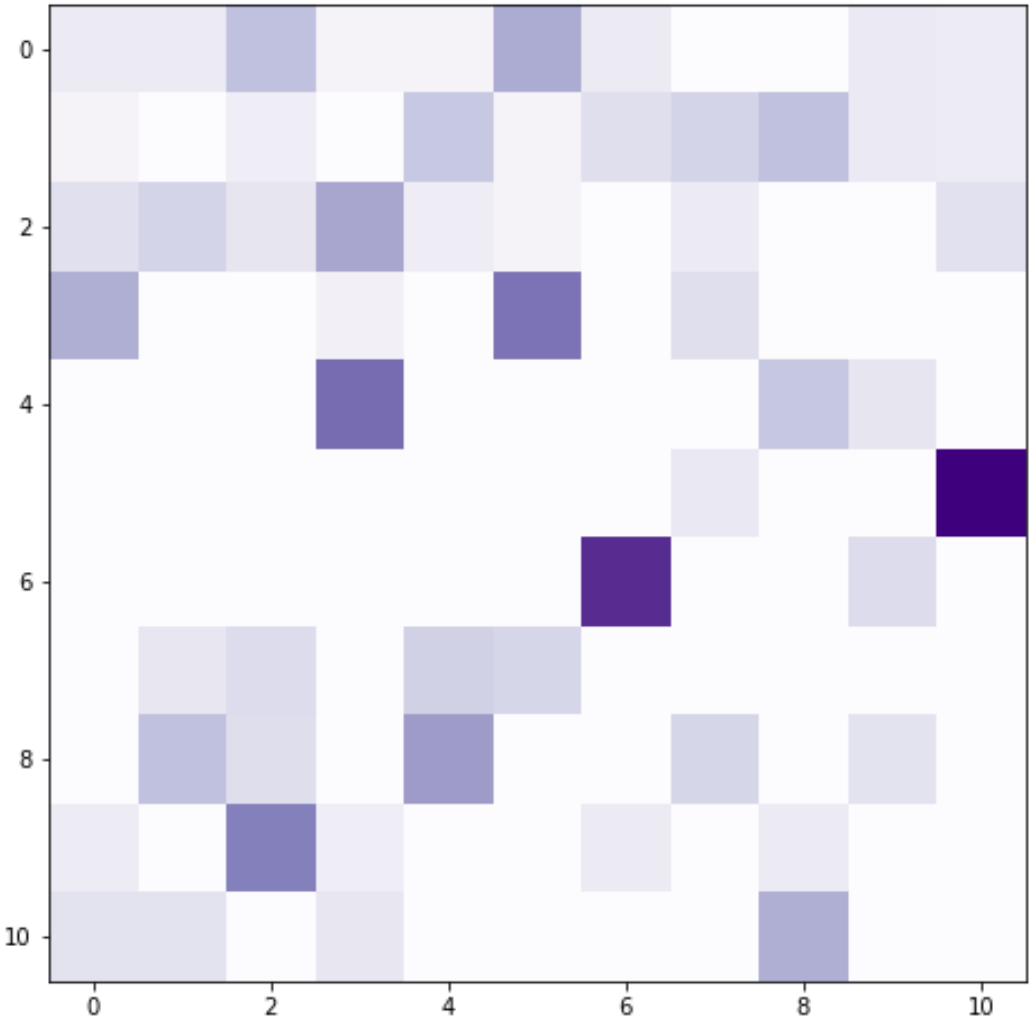
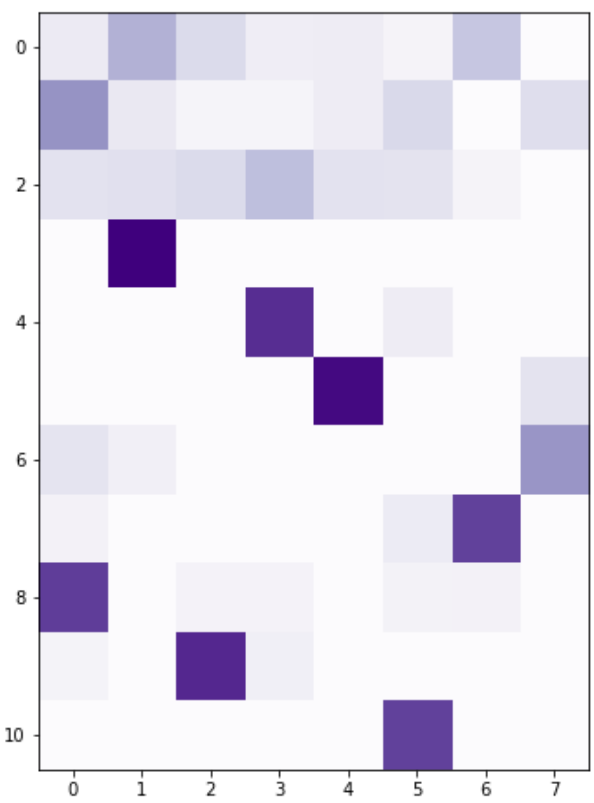
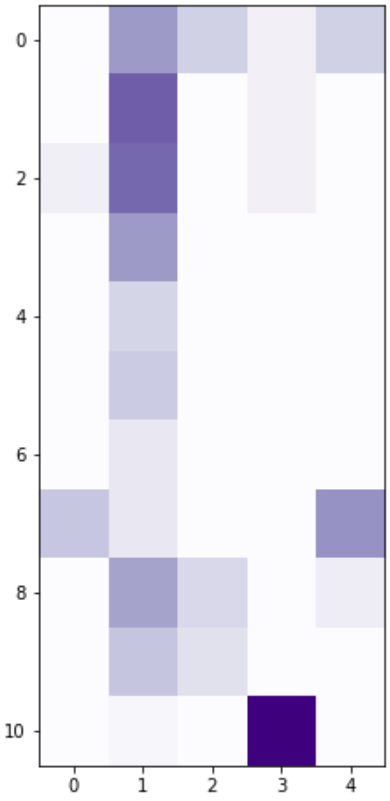


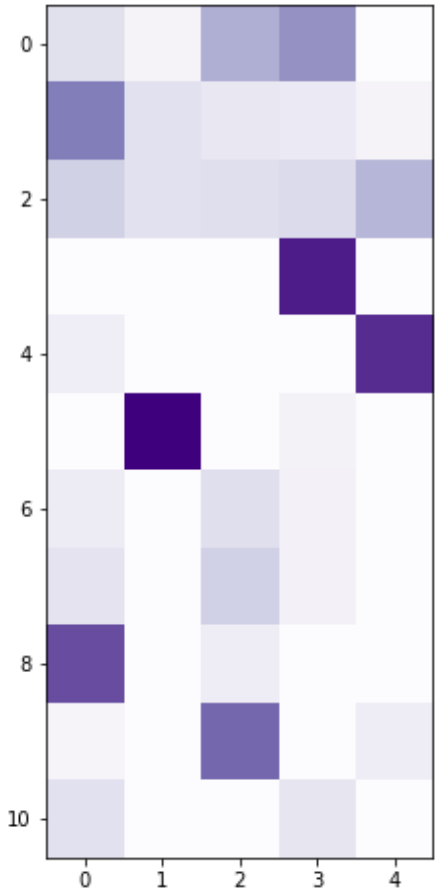
Figure 50: Heatmap of Asynchronous Fluid (k=a+b) on test75



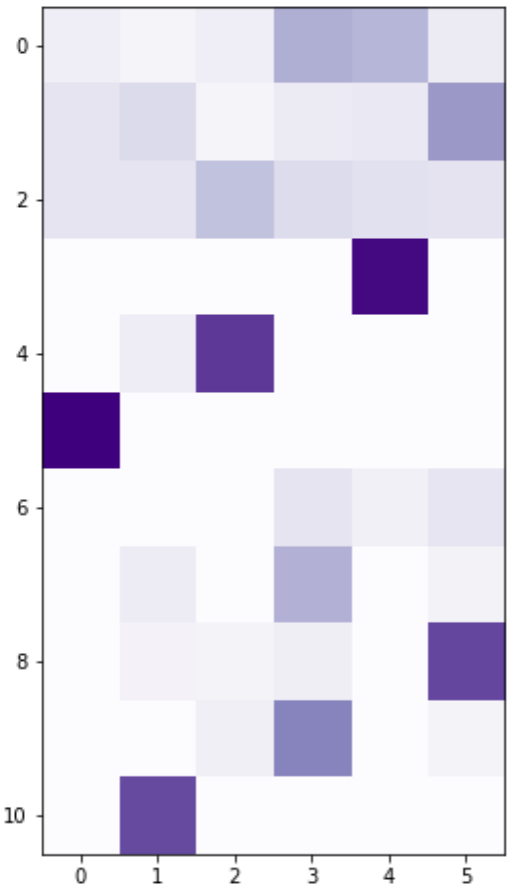
(a) Heatmap of Infomap on test75



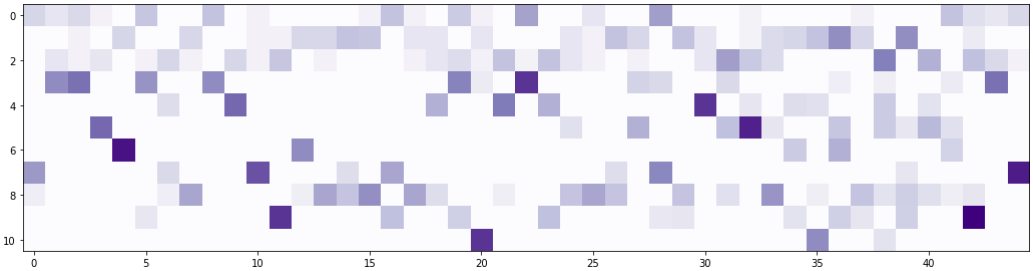
(b) CPM (k=3) on test75



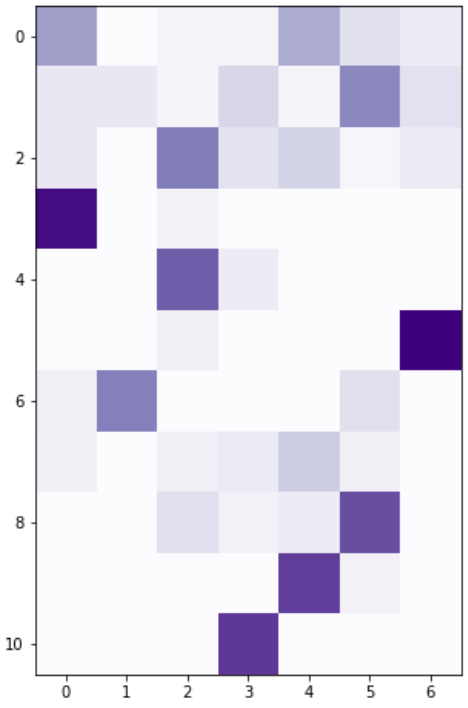
(a) Heatmap of Labelprop on test75



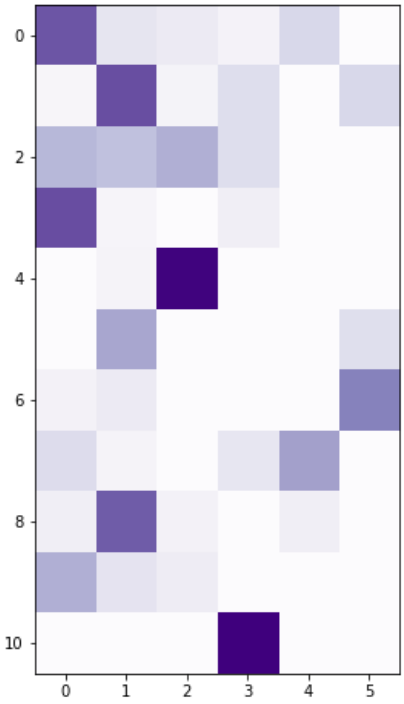
(b) Heatmap of Louvain on test75



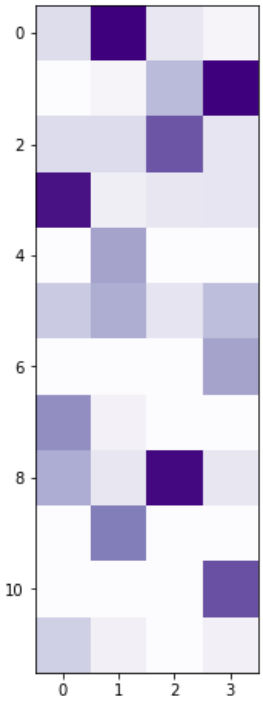
(c) Heatmap of Linkcomm on test75



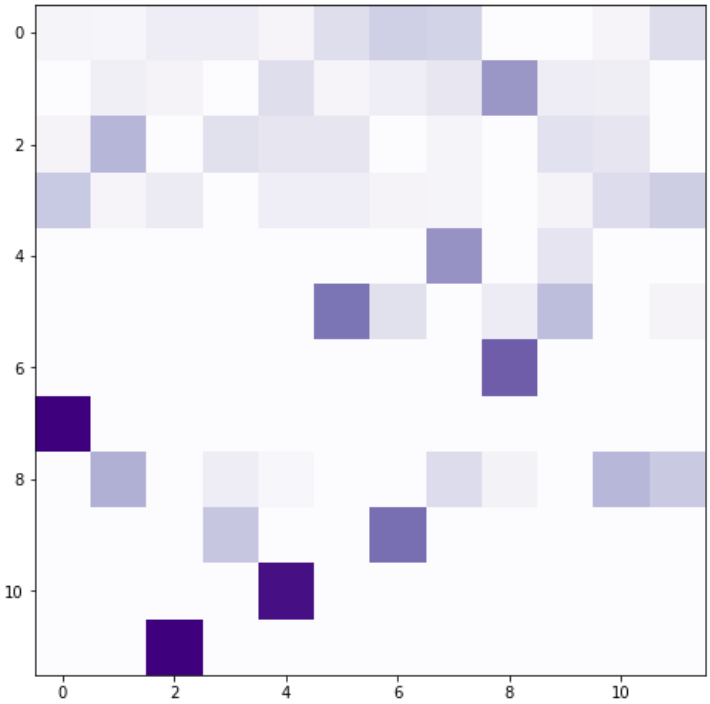
(a) Heatmap of Spinglass on test75



(b) Heatmap of Walktrap on test75



(a) Async Fluid (k=a)



(b) Heatmap of Asynchronous Fluid (k=b) on test100

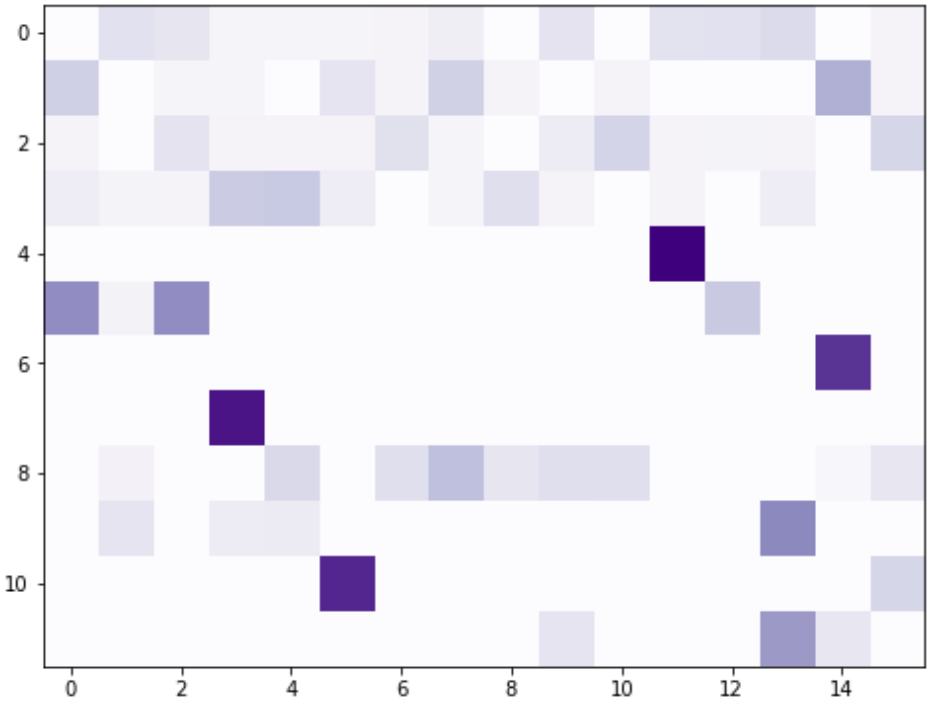
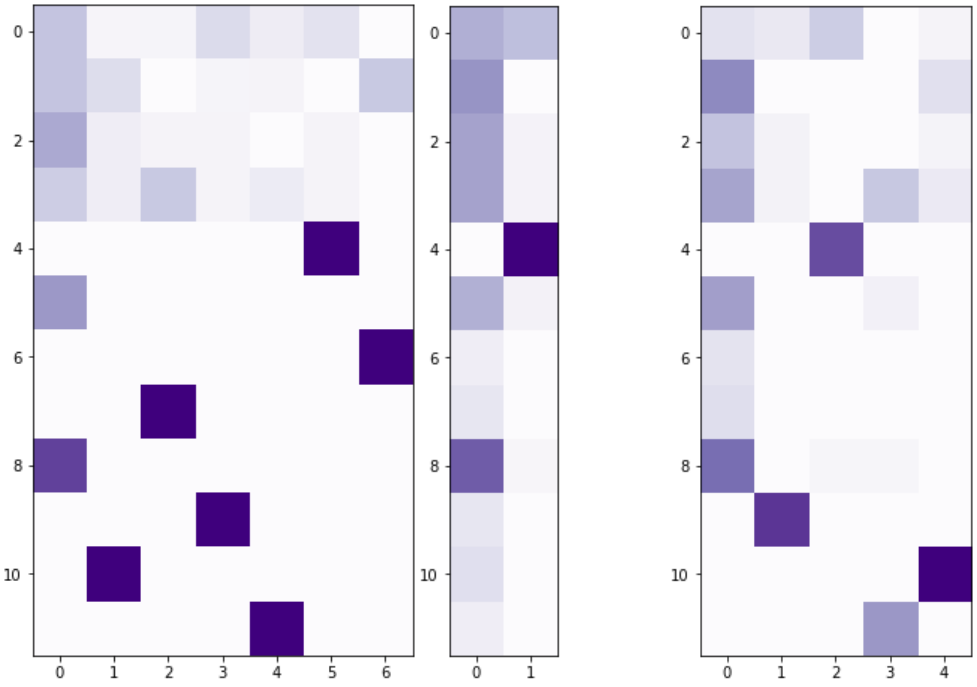


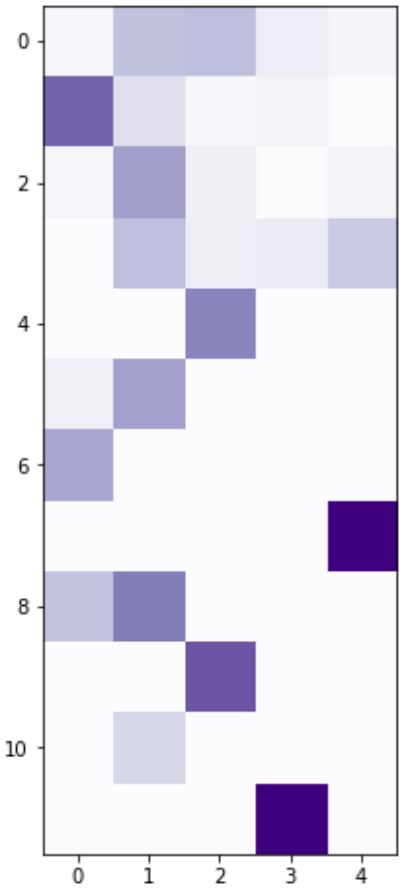
Figure 55: Heatmap of Asynchronous Fluid (k=a+b) on test100



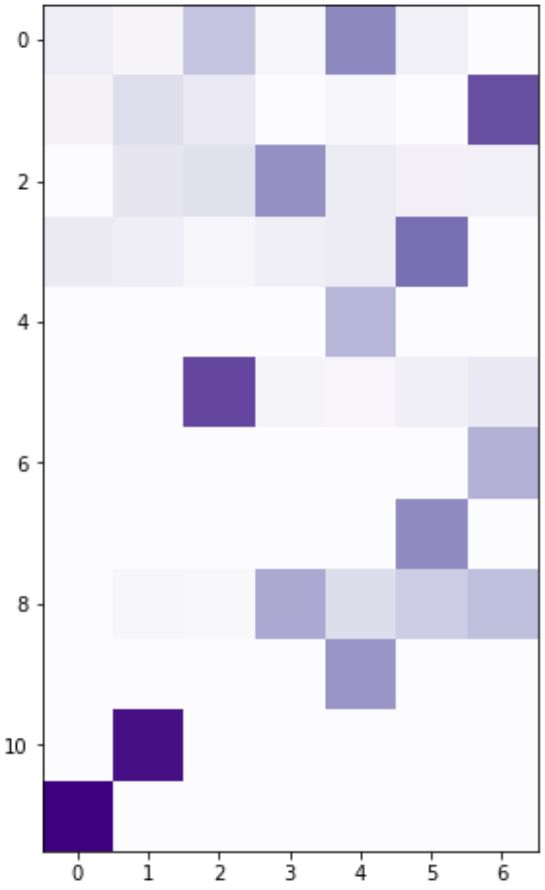
(a) Heatmap of Infomap on test100

(b) CPM (k=3) on test100

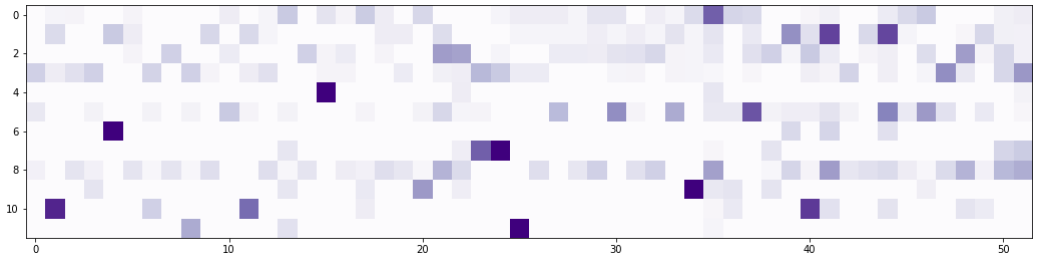
(c) Heatmap of CPM (k=4) on test100



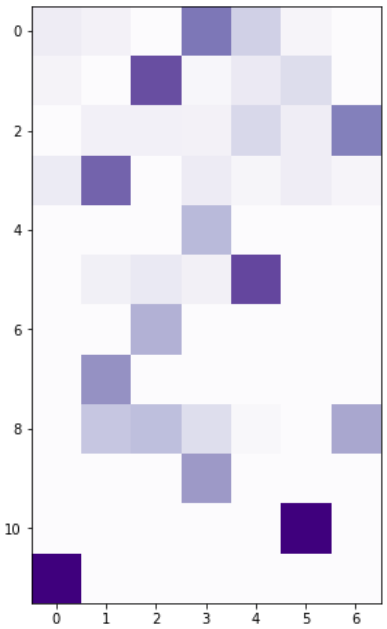
(a) Heatmap of Labelprop on test100



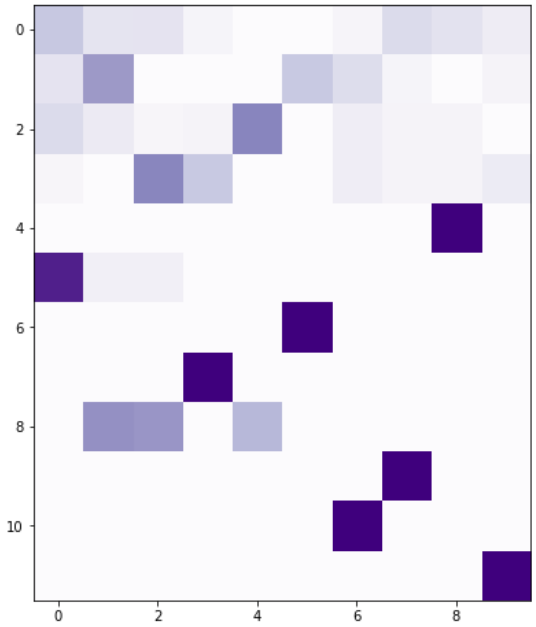
(b) Heatmap of Louvain on test100



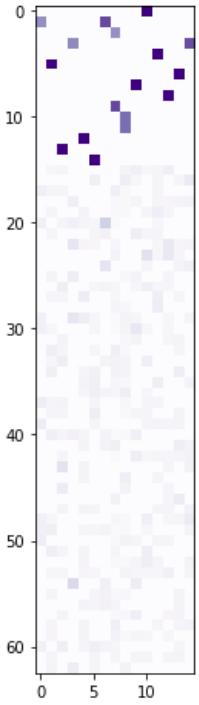
(c) Heatmap of Linkcomm on test100



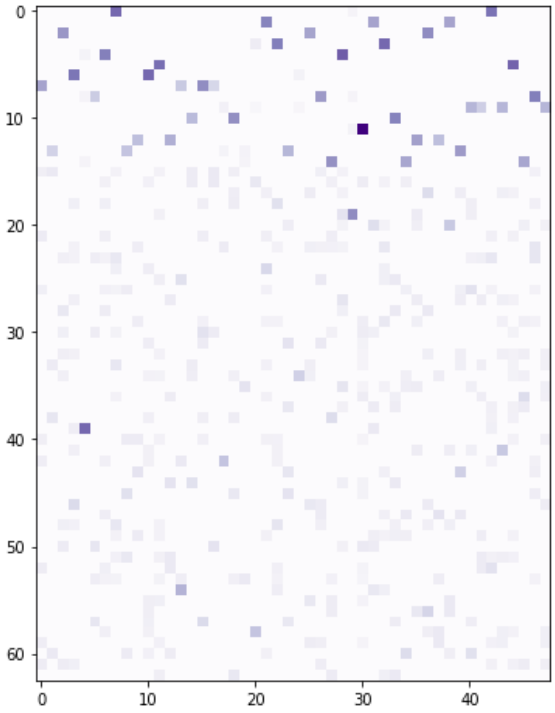
(a) Heatmap of Spinglass on test100



(b) Heatmap of Walktrap on test100



(a) Async Fluid (k=a)



(b) Heatmap of Asynchronous Fluid (k=b) on test400

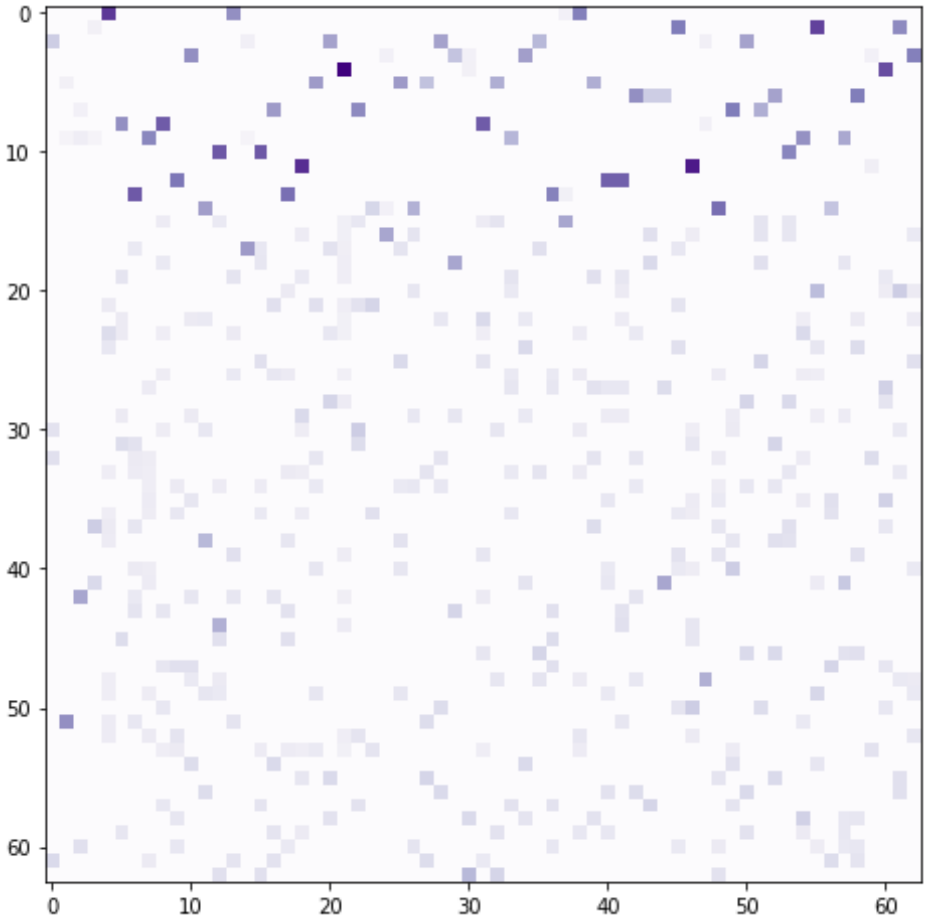
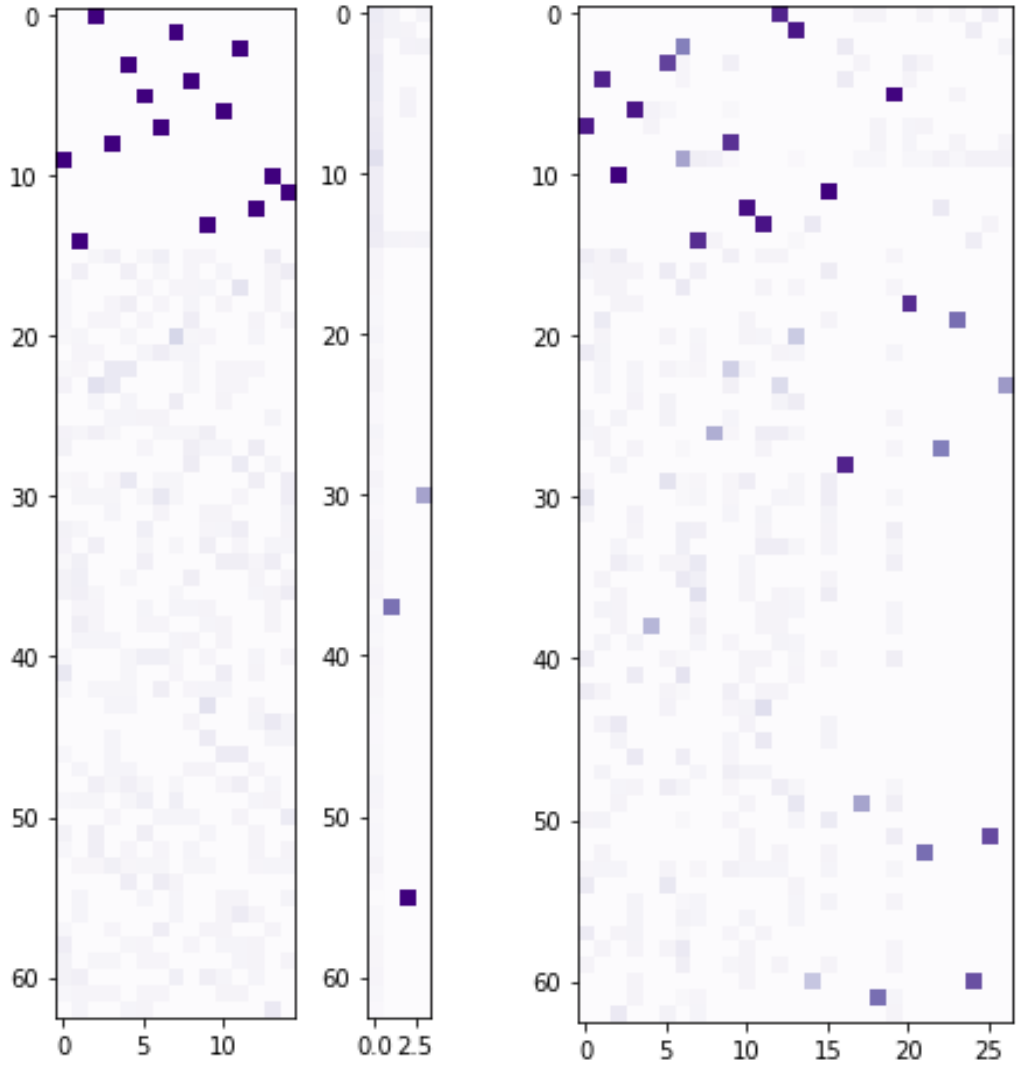
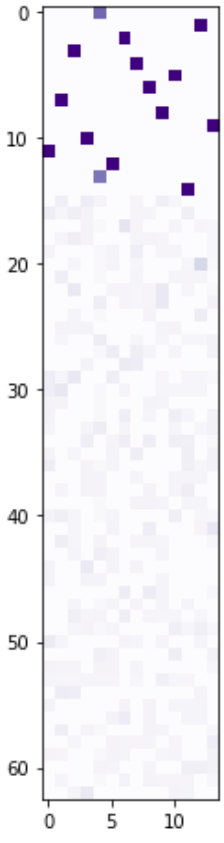


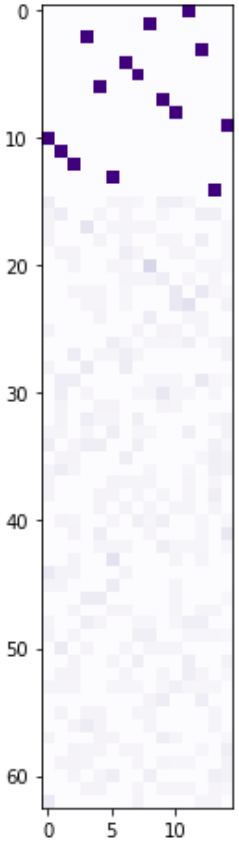
Figure 60: Heatmap of Asynchronous Fluid (k=a+b) on test400



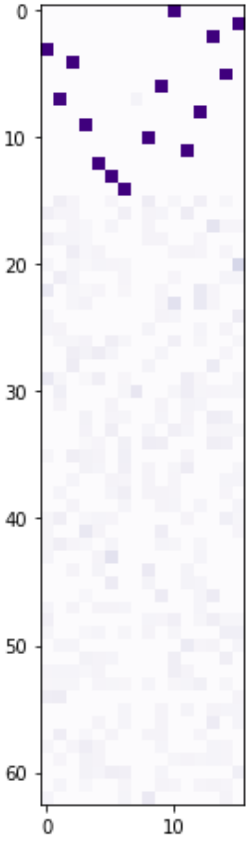
(a) Heatmap of Infomap on test400 (b) CPM (k=3) on test400 (c) Heatmap of CPM (k=4) on test400



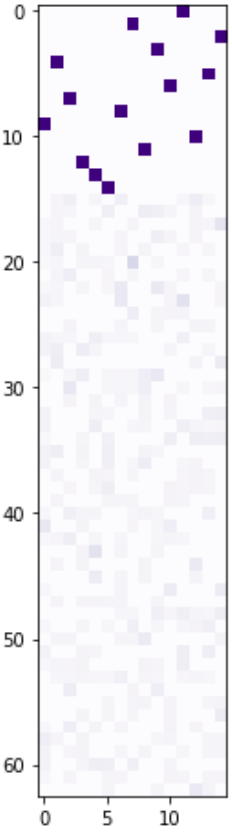
(a) Heatmap of Labelprop on test400



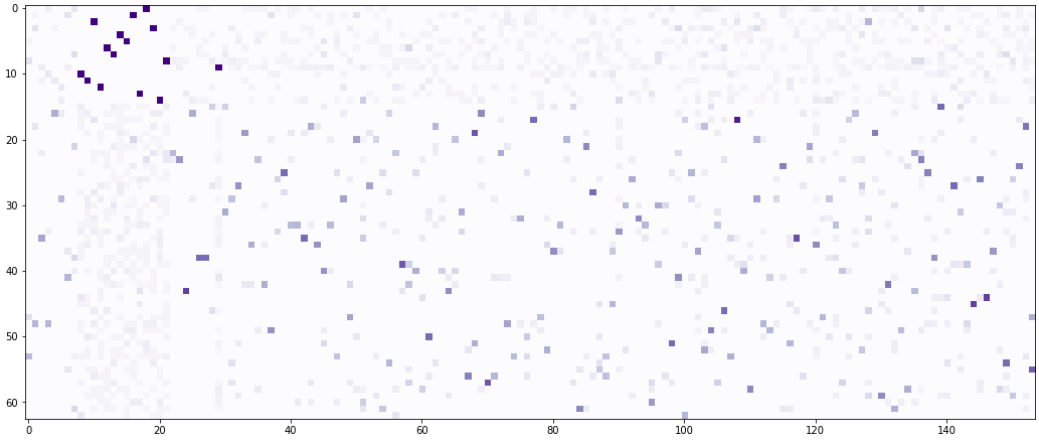
(b) Heatmap of Louvain on test400



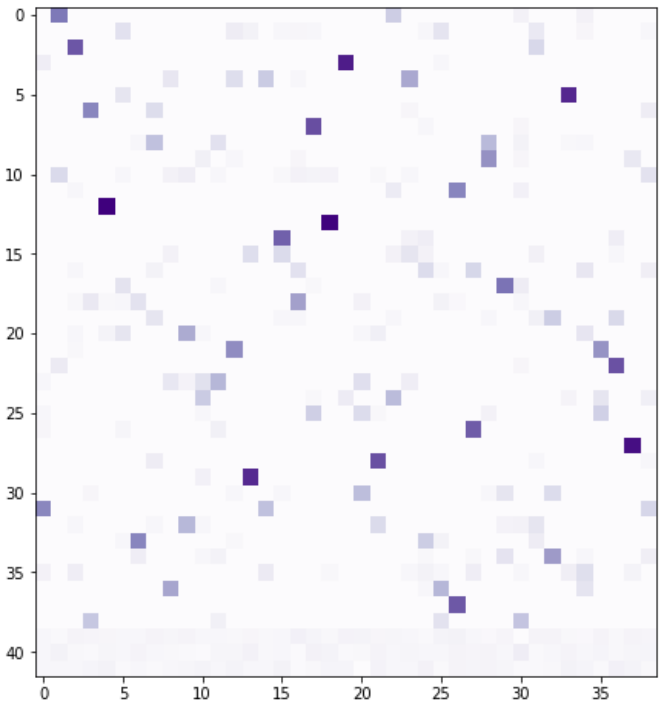
(c) Heatmap of Spinglass on test400



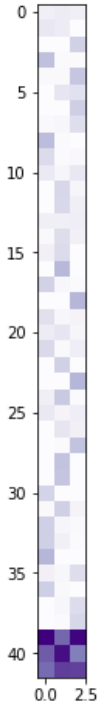
(d) Walktrap on test400



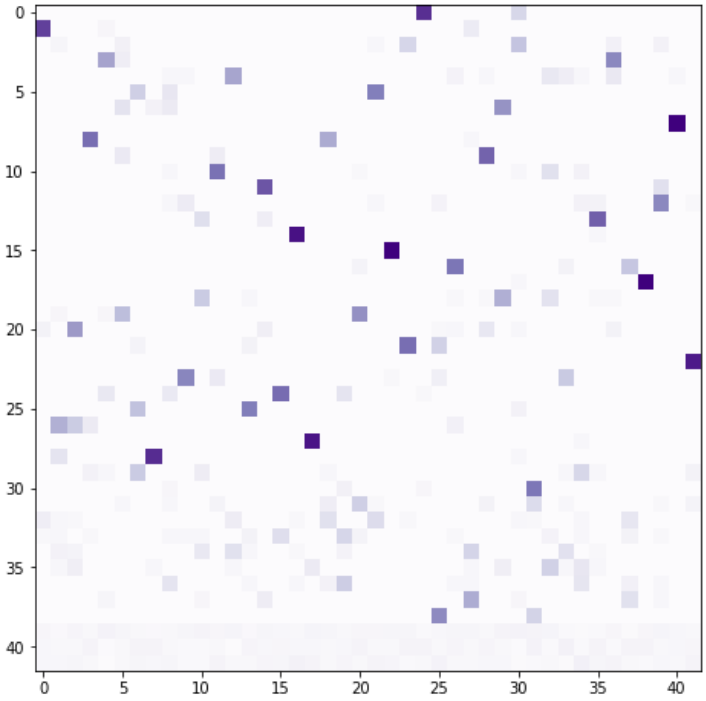
(e) Heatmap of Linkcomm on test400



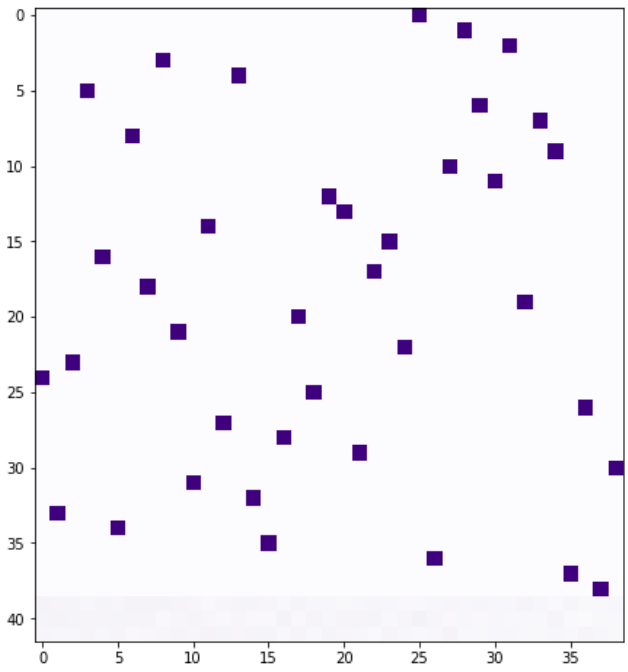
(a) Heatmap of Async Fluid (k=a) on test1000



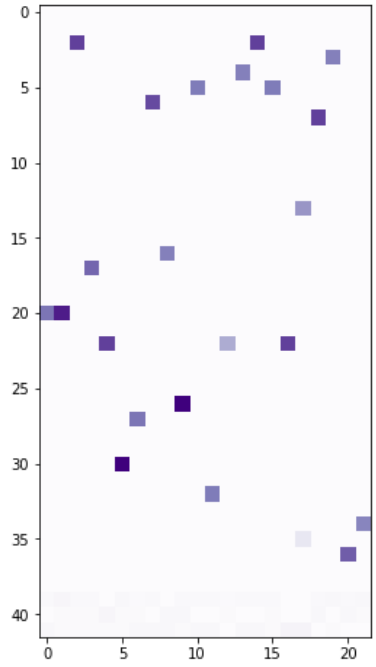
(b) Asynfluid (k=b)



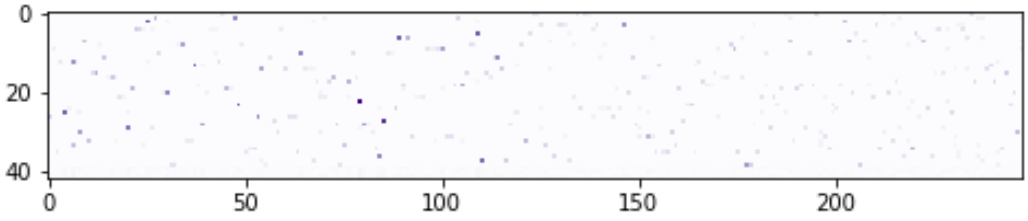
(c) Heatmap of Async Fluid (k=a+b) on test1000



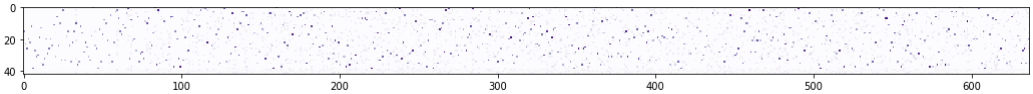
(a) Heatmap of Infomap on test1000



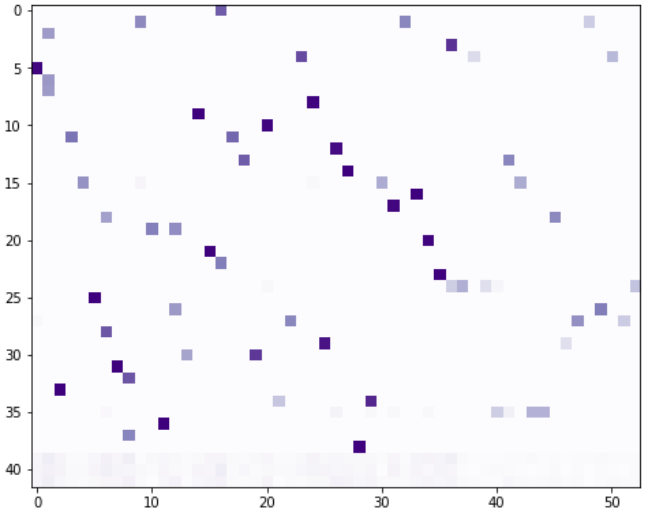
(b) CPM with k=4 on test1000



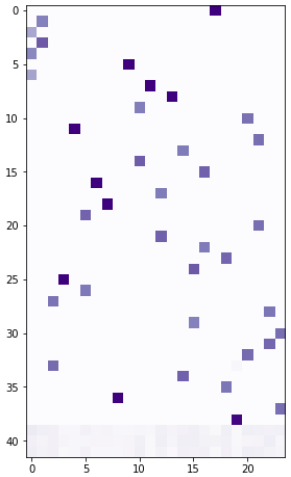
(c) CPM with k=3 on test1000



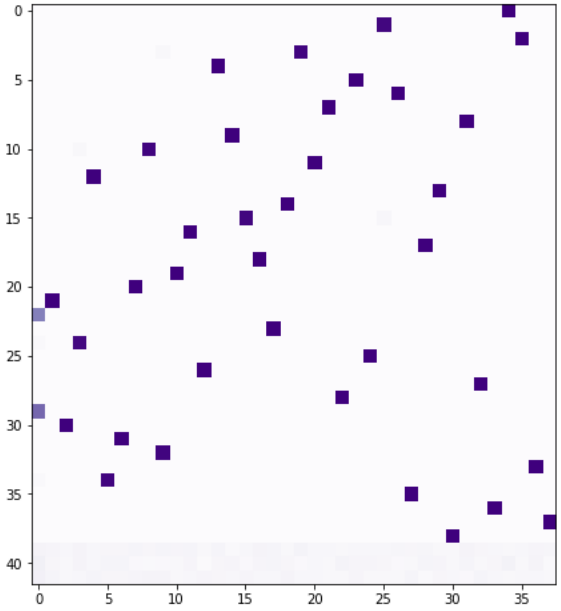
(d) Linkcomm on test1000



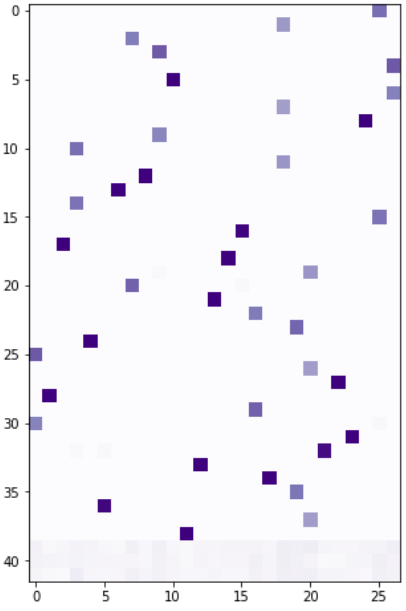
(a) Heatmap of Labelprop on test1000



(b) Spinglass on test1000



(c) Heatmap of Walktrap on test1000



(d) Louvain on test1000