Louvain-initialized Genetic Algorithms for Community Detection in Complex Networks

Bachelorarbeit

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Dominic Plein

Abgabedatum:	28. August 2022
Bearbeitungszeitraum:	06.06.2022 - 28.08.2022
Matrikelnummer, Kurs:	9664381, TINF19B2
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma:	Matthias Hauck, Umang Rawat
Gutachter der Dualen Hochschule:	Prof. Dr. Heinrich Braun

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Louvain-initialized Genetic Algorithms for Community Detection in Complex Networks*

gemäß § 5 der "Studien- und Prüfungsordnung DHBW Technik" vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Speyer, den 28. August 2022

Plein, Dominic

Abstract

- English -

The identification of community structure in complex networks is actively studied in network science. Communities are densely connected groups of vertices that are only loosely linked to other groups. These modules are present in many social and biological networks and thus their recognition is beneficial to the understanding of the overall structure. In this thesis, the well-known Louvain method for community detection is efficiently implemented for the SAP HANA Graph Engine and the delta modularity calculation generalized for usage with Genetic Algorithms (GAs). We present a GA based on the optimization of modularity Q and employ the Louvain algorithm as initialization heuristic. Mutation operators are borrowed from Lehnerer while we develop two additional mutations. Experiments on computer-generated and several real-world networks show that the GAs are not capable of improving the modularity achieved by Louvain in most cases, although slight increases were sometimes possible.

Keywords: Complex Networks, Community Detection, Louvain Method, Modularity, Genetic Algorithm.

Abstract

- Deutsch -

Der Forschungsbereich *Network Science* beschäftigt sich unter anderem ausgiebig mit dem Erkennen von Gemeinschaftsstrukturen in komplexen Netzwerken. Solche Gemeinschaften (*Communities*) sind Zusammenschlüsse von Knoten, die in sich sehr stark vernetzt sind, jedoch nur wenige Kanten zu Knoten aus anderen Gemeinschaften aufweisen. Da Communities in realen sozialen und biologischen Netzwerken allgegenwärtig sind, kann deren Identifikation zu neuen Erkenntnissen bezüglich des Aufbaus dieser Netzwerke führen. In dieser Bachelorarbeit wird die bekannte Louvain-Methode effizient für die SAP HANA Graph Engine implementiert und die Berechnung der Delta-Modularität für die Verwendung in genetischen Algorithmen (GAs) verallgemeinert. Wir stellen einen GA vor, der Modularität Q optimiert und den Louvain-Algorithmus als Initialisierungsheuristik verwendet. Die Mutationsoperatoren sind von Lehnerer übernommen und zwei weitere Mutationen werden eingeführt. Experimente auf computergenerierten und zahlreichen realen Netzwerken zeigen, dass die GAs in den meisten Fällen nicht in der Lage sind, die Modularitätswerte von Louvain zu verbessern, jedoch teilweise kleine Verbesserungen möglich sind.

Schlagwörter: Komplexe Netzwerke, Gemeinschaftsstrukturen, Louvain Methode, Modularität, Genetische Algorithmen.

Table of Contents

Mathematical symbols

$G = (V, E)$	Undirected graph consisting of a set of vertices V and edges E.
$V(G)$, $E(G)$	Vertex and edge set of a graph G.
$\lvert E(c)\rvert$ or m_c	Number of edges in community c (both ends of the edges are inside community c).
A	Adjacency matrix of a graph G.
k_v	Weighted degree of vertex v, that is, the sum of the weights of edges incident upon vertex v: $k_v = \sum_w A_{vw}$.
$k_v^{\to c}$	Sum of the weights of the edges between node u and community c: $k_v^{\to c} = \sum_{w, c_w = c} A_{vw}$.
$u \sim v$	Vertices u and v have an undirected edge between them.
$c \subseteq V(G)$	Community (also called cluster, module or circle): subset of vertices. The subgraph induced by this subset is not inherently required to be connected, although this is beneficial for a good modularity.
$\dot{c}$	Singleton community, i. e. a community comprising only one vertex.
$\mathcal{C} = \{c_1, \ldots, c_t\}$	Clustering (also called division or partition): a partition of $V(G)$ into communities such that each vertex appears in exactly one community.
$G(U, V, E)$	Bipartite graph with $\lvert U\rvert + \lvert V\rvert$ vertices, that is, V can be split into two disjoint sets, so that no edges run between the nodes *within* the same vertex set, or more formally: $\forall \{ u, v \} \in E : (u \in U \wedge v \in V) \dot{\vee} (u \in V \wedge v \in U)$.
$K_{m,n}$	Complete bipartite graph with $m + n$ vertices, i. e. isomorphic to $G = (V, E)$ with $V = A \cup B$ where $A = \{ a_1, ..., a_m \}$ and $B = \{ b_1, ..., b_n \}$ are partite sets of G and $E = \{ \{ a_u, b_v \} : a_u \in A, b_v \in B \}$. Special case of a bipartite graph, so that every node of U is connected to every other node of V.
$N_G(v)$	The set of neighbors of $v \in V(G)$ in the graph G, more formally, $N_G(v) := \{ w : vw \in E(G) \}$.
clique $H \subseteq V(G)$	Subset of vertices of the undirected graph G, such that the subgraph induced by H is a complete graph. Vertices of a graph are 1-vertex, edges are 2-vertex cliques.

List of Abbreviations

Louvain CRATER	Louvain on Crater Rim, Adoption Towards Eruption Repair
CV	Coefficient of Variation
GA	Genetic Algorithm
LIIM	Louvain-Initialized Inherit Mutation
PRNG	Pseudo-Random Number Generator
SNAP	Stanford Network Analysis Platform
TSP	Traveling Salesperson Problem

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1. Motivation

Many complex systems and real-world interactions can be represented by graphs, e. g. social interaction between humans. Social network analysis is indeed its own active research area in sociology. Many of these networks exhibit special characteristics that differ from the Erdős–Rényi model [1] used to generate random graphs. Such features include the common power-law distribution [2] of the vertex degrees, i. e. a small number of vertices is part of the majority of edges. In other words, there are many vertices with a low degree and few with a high degree. This even holds true for the topology of the world-wide web as discovered by Albert, Jeong, and Barabási in [3].

Another common property of complex networks is that of *community structure*. Communities are groups of vertices which have many edges within the groups compared to only few edges between them. In other words, vertices are densely connected *within* and only sparsely connected *between* groups (see Figure 2.1a for a simple example). Communities are cohesive groups where vertices are "similar" to each other, i. e. they have common traits. For this reason, identifying the community structure of a complex network stemming from real-world applications might open up completely new possibilities to look at a network and understand its functional properties.

Applications of community detection include recommendation systems for video platforms (recommend similar videos to the viewer) and web shops (knowing which items form a community, a web shop can present buyer similar articles or articles that other people have bought) [4]. As mentioned before, a large application area is social network analysis especially with data available from big social media networks. Methods for community detection are not limited to these areas and could in principle be applied to any problem that can be represented as graph and that displays community structure in some sense. For example, community detection has been applied to detect and prevent crime in online social networks [5] or to study genomic patterns of cancer and tumors [6].

Modularity (see section 2.1) is a common metric to measure how well the proposed communities satisfy the aforementioned condition (high concentration of within-community edges compared to between-community edges). Unfortunately, finding a graph partition that maximizes modularity is strongly NP-complete [7]. A large research community has formed around community detection to devise numerous heuristics in order to find good communities in a reasonable time even for big networks with millions of nodes. A well-known greedy optimization method is presented by Blondel et al. in [8]. Part of this thesis will explain how this method works and derive all formulas step by step as to compensate for the lack of a detailed explanation of the math behind the successful method. We will present the overall algorithm and its parts as pseudo-code and point out how Louvain was implemented for the SAP HANA Graph Engine.

While the Louvain method finds a local optimum of modularity Q, the global optimum could stay out of reach. To encounter this problem, we devise a GA (see section 2.2). These stochastic optimization methods are modeled after the evolutionary process of adaptation to an environment by means of sexual reproduction throughout multiple generations. They have demonstrated in other research areas their ability to evade local optima and have also been applied to the community detection problem as will be elaborated in the related work section.

Existing GA for community detection mostly initialize the population randomly. From [9], we learn that there are many other alternatives which could significantly improve the performance of the GA, e. g. its convergence speed or the quality of solutions. In this thesis, we will initialize a GA with the results obtained by the Louvain method. We ask ourselves if a genetic algorithm can overcome the local maximum for modularity that Louvain achieves. Results are compared to the random-initialized GA and to Louvain. For the genetic algorithm, we focus on mutation operators: two of them are taken from [10], while the other two are newly devised. We will generalize the quick delta modularity ΔQ formula found in Louvain for usage in the GA. The evaluation is performed on random computer-generated networks varying the z_{out} value (see section V.A. in [11]) and on several real-world datasets that can be downloaded from GitHub[1].

While we implemented Louvain such that edge weights can be passed along as lambda function, for this work we restrict ourselves to unweighted graphs. Furthermore, we only

[1] https://github.com/splines/bachelor-community-detection-datasets

consider undirected graphs and do not allow self-loops in the original network to simplify some formulae. We also exclude the analysis of dynamic networks and only consider non-overlapping communities (although overlapping clusters are included in our dataset, however we do not consider the ground-truth in this case or use a simple technique to adjust the graph accordingly).

1.2. Related work

A very popular algorithm for community detection is that by Newman and Girvan [11]. It employs edge betweenness-centrality in order to find edges through which the shortest paths lead. These edges are likely to be inter-community edges and are successively removed from the graph. This technique is very similar to divisive hierarchical clustering, i. e. "clustering techniques that reveal the multilevel structure of the graph" [12]. Newman and Girvan's work is considered as milestone in the research area of community detection as it introduced modularity as quality function which has quickly gained popularity and is now commonly employed in many different clustering methods.

In [12], Fortunato gives a comprehensive overview of different approaches to community detection ranging from traditional methods like spectral clustering to modularity-based strategies using greedy techniques and simulated annealing. As evolutionary algorithms only make up a small part in this survey, Khan and Niazi provide a survey for this area including single- and multi-objective optimization techniques, discussions on the fitness function and operator design, e. g. mutations to the chromosome [14]. A scientometric analysis of network community detection is provided in [13] containing a list of existing literature reviews. In the following, we briefly present a limited selection of papers concerning genetic approaches for community detection.

In [15], Tasgin, Herdagdelen, and Bingol employ a genetic algorithm that optimizes modularity Q. Vertex-community assignments are encoded as vector of size n where the element at index i encodes the community for vertex i. We will discuss this representation in section 4.1. This encoding is ambiguous as different vectors may correspond to the same partition of the network into communities. The authors therefore employ variation of information based on information entropy to measure the distance between two partitions. We will apply a simpler measure in section 7.1. With respect to their genetic algorithm,

the initial population is created randomly with a slight improvement by additionally picking random vertices and assigning their community to all neighbor vertices. A simple one point-mutation operator is used that randomly picks two vertices and assigns the community of one vertex to the other. Moreover, a cross-over is implemented with an additional "translation" step to account for the ambiguity of the community encoding. The algorithm is evaluated on synthetic and small real-world networks obtaining results comparable to Newman's agglomerative hierarchical clustering method [16] while not relying on previous knowledge about the number of communities.

In [10], Lehnerer uses the same encoding for the chromosome as [15] and described in section 4.1. He optimizes for modularity Q as the objective function. Two mutations are used: in the first mutation random vertices get assigned random communities. This mutation is discussed in detail in section 5.1. In the second method, a vertex inherits the community that is most prevalent in the respective neighborhood. We will describe this mutation operator in section 5.2. An additional clean up removes singleton communities (communities that only consist of one single node). Elitism is used as selection method, i. e. that only the fittest individual survives. Beyond that, a one-way cross-over is implemented which is comparable to the one devised in [15]. The method is evaluated on three synthesized networks and outperforms community detection methods built into Mathematica in terms of quality but not speed.

In [17], Pizzuti introduces the notion of *community score* as new objective function. It is based on the adjacency matrix of a graph, in which maximal and dense sub-matrices ought to be found. For the genetic chromosome, the locus-based adjacency representation is used where a vector of size n contains at index i a value j if $i \sim j$, i. e. vertex i is connected to j. The population is initialized randomly with a subsequent repair step to ensure safe genes. A gene i of the chromosome is considered *safe* if its value j is indeed a neighbor of i. This definition guarantees that every component an individual encodes is a *connected* subgraph of the original graph G. A simple one-point mutation is applied that randomly assigns a new value to a gene; however, to always produce safe individuals, only neighbors of node i can be selected. Additionally, a uniform cross-over with a randomly created bitmask is used as it also ensures all individuals remain safe. The approach is evaluated on synthetic networks by means of normalized mutual information (a measure introduced in [18]) and on small real-world networks yielding comparable results to Newman and Girvan.

1.3. Terminology

We want to clarify some of the ambiguous terminology used throughout this work that may lead to confusion for the inclined reader. Albeit there might exist subtle distinctions between the following terms, we will use them interchangeably as is also often done in literature. Similar words are separated by comma:

- nodes, vertices & edges, links & adjacent/neighboring vertices, neighbors
- graph, network & community, cluster, module, circle & cluster, division, partition
- within, intra- & between, inter-
- evolutionary algorithm, genetic algorithm (even though genetic algorithms are a subclass of evolutionary algorithms)
- chromosome, genotype, individual & fitness/objective/quality function
- Louvain method/algorithm, Louvain (even though Louvain is a city in Belgium)
- vertex-community assignment, vertexToCommunity (in the code)
- ground-truth communities, original vertex-community assignment/labels

1.4. Structure

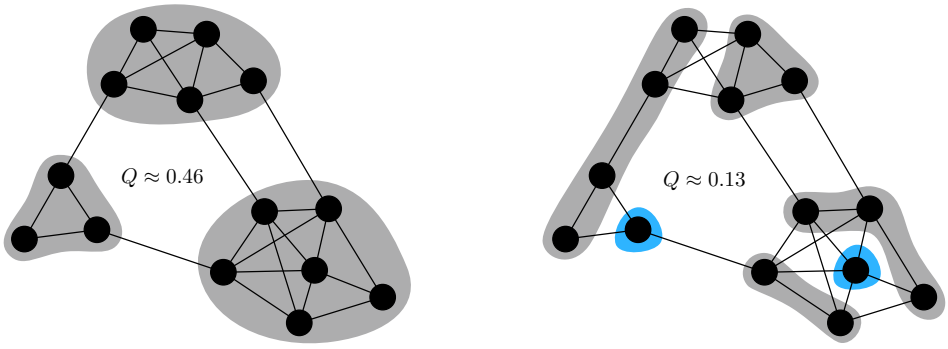
This thesis is structured as follows. Chapter 2 introduces fundamentals regarding modularity and genetic algorithms. Chapter 3 extensively addresses the Louvain algorithm including pseudo-code and the derivation of the quick delta modularity calculation. It also provides a generalized formula and presents C++ implementation details. In Chapter 4, we describe the methodology for our genetic algorithm including Louvain as initialization heuristic. Mutation operators are presented and devised in Chapter 5. The entire Chapter 6 is devoted to the time-intensive task of data preparation and introduces the datasets used for the evaluation in Chapter 7. Chapter 8 concludes this work by highlighting the main findings and discussing which problems might be of interest for future research. Appendix A includes further experimental results.

2. Preliminaries

2.1. Modularity

We are often faced with networks where we do not know the underlying community structure in advance, yet still need a measure to evaluate how good our proposed partition of the network into communities is. This is especially important for genetic algorithms requiring an objective function to maximize. Newman and Girvan proposed a measure called *modularity* in 2003 [19], which was adopted broadly and successfully in the network research community [12]. Modularity Q is maximized for divisions of a graph when many edges fall *within* the proposed communities (intra-community edges) as compared to edges *between* communities (inter-community edges).

Figure 2.1 depicts the same contrived network with different vertex-community assignments. The intuitive grouping of vertices into communities in (a) leads to the maximal achievable modularity value of $Q \approx 0.46$ for this particular graph. This grouping is characterized by only sparse connections between communities which are in themselves densely connected. In contrast, figure (b) results in a lower value of just $Q \approx 0.13$. We will discuss the possible range of modularity values later on.



(a) optimal grouping of vertices into three communities, inspired by [20]

(b) bad grouping into five communities, the blue community induces a disconnected subgraph

Fig. 2.1.: Simple network with different proposed vertex-community assignments

In the following, we assume a graph G with $|V(G)| = n$ vertices and $|E(G)| = m$ edges. The edge set $E(G)$ is determined by the adjacency matrix A with entries:

$$A_{uv} = \begin{cases} 1, & \text{if } u \sim v \\ 0, & \text{otherwise} \end{cases}$$

Of all the edges, **a high fraction of edges that connect vertices in the same community** should give a good partition. For a specific community c, this fraction is given by

$$e_c = \frac{1}{2m} \sum_{u \in c} \sum_{v \in c} A_{uv} \tag{2.1}$$

where we iterate over all edges $\{u, v\}$ with both ends in community c. We divide by m to retrieve the fraction of all edges. The factor 2 comes from the assumption that we only deal with undirected edges, so every edge occurs twice for every order of arguments (A_{uv} and A_{vu}). We call this term by e_c in accordance with [19][1]. Aiming for

$$\max \sum_{c \in \mathcal{C}} e_c \tag{2.2}$$

– with $\mathcal{C}$ being the set of all proposed communities – makes sure a high fraction of edges is within communities (and not between). However, the trivial case with all vertices in the same community yields a maximum value of 1. Therefore, we cannot use this measure alone, but instead compare this fraction of intra-community edges with the *expected* number of intra-community edges if edges were distributed at random in our graph. Note that "random" does not mean any arbitrary graph, instead, the following properties should be preserved to allow for a fair comparison:

1. the number of nodes n,
2. the number of edges m and
3. the vertex degrees k_v for all $v \in V(G)$.

We employ the configuration model as null model (as in [11]). A new, "empty" graph with n nodes is constructed and m edges are randomly inserted between vertices whilst ignoring their (potentially known) community structure.

[1] In contrast to the mentioned paper, for better clarity, we will sum over the communities instead of using the Dirac measure δ to indicate that vertices belong to the same community

In order to analyze the expected fraction of edges between any nodes u and v in a community, we cut every edge into two halves, leaving only edge stubs on the vertices as shown in Figure 2.2. Since we have m edges in our graph, we end up with $2m$ edge stubs. There are k_v ways to connect one edge stub of vertex u (marked in red) to any edge stub of v (marked in blue). Randomly, we could connect this one edge stub of u to any of the $2m-1$ edge stubs in the entire graph[2]. Hence, the probability that one edge stub of u connects to any edge stub of v in our random graph is given by $\frac{k_v}{2m-1}$. As vertex u has k_u edge stubs, the probability that vertex u connects to vertex v, i. e. any edge stub of u connects to any edge stub of v, is $\frac{k_u k_v}{2m-1}$.

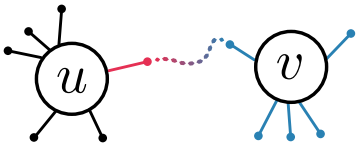


Fig. 2.2.: Configuration model with $k_u = 6$ and $k_v = 5$. One possible edge between stubs is shown as dashed line.

This yields the equation for the **expected number of edges that fall within a community c if edges were distributed randomly in the graph while preserving node degrees**:

$$a_c^2 = \frac{1}{2m} \sum_{u \in c} \sum_{v \in c} \frac{k_u k_v}{2m} \tag{2.3}$$

The same reasoning as before applies here for the term $\frac{1}{2m}$. Since we are dealing with big networks where $2m \gg 1$, it is reasonable to drop the aforementioned "-1" in the denominator, which is common in literature[3] [19, 20]. Again, to show the connection to the original definition of modularity in [11], we refer to this term as a_c^2. This is because a_c, for any community c, is given by

$$a_c = \frac{1}{2m} \sum_{v \in c} k_v \tag{2.4}$$

[2]The "-1" is introduced as an edge stub cannot connect to itself in the configuration model. However, this still allows for self-loops as one edge stub could connect to another one on the same vertex.

[3]Beyond that, Equation 2.3 is actually only true if we assume that there are not too many self-loops and multi-edges in the graph. In probability theory, the general expected value is defined as $E[X] = x_1 p_1 + x_2 p_2 + \cdots + x_n p_n$. Only if all possible outcomes $x_1, \cdots, x_n$ evaluate to 1 do we get $E[X] = p_1 + p_2 + \cdots + p_n$, which we make use of in Equation 2.3.

which is the **fraction of edge stubs** (out of all $\approx 2m$ edge stubs in the graph) **that are attached to vertices in community** c. One could also refer to this as the total vertex degree of vertices in community c divided by twice the number of edges in the graph $2m$.

By combining Equation 2.1 and 2.3 and by iterating over all communities $c \in \mathcal{C}$, we get:

$$\begin{aligned} Q(\mathcal{C}) &= \big(\text{fraction of edges that fall within the proposed communities}\big) \\ &\quad - \big(\text{expected fraction of such edges in a random graph}\big) \\ Q(\mathcal{C}) &= \sum_{c \in \mathcal{C}} (e_c - a_c^2) \\ Q(\mathcal{C}) &= \sum_{c \in \mathcal{C}} \Big(\frac{1}{2m} \sum_{u \in c} \sum_{v \in c} A_{uv} - \frac{1}{2m} \sum_{u \in c} \sum_{v \in c} \frac{k_u k_v}{2m} \Big) \end{aligned}$$

This gives us the definition of modularity that is often found in literature. We can later use this measure as quality function for genetic algorithms, rendering the problem of community detection into that of modularity optimization.

$$\boxed{Q(\mathcal{C}) = \frac{1}{2m} \sum_{c \in \mathcal{C}} \sum_{u \in c} \sum_{v \in c} \left(A_{uv} - \frac{k_u k_v}{2m} \right)} \tag{2.5}$$

Note that while we only consider undirected graphs in this thesis, modularity can be extended to directed graphs [21].

Alternative formulation

An alternative formulation of modularity is used in [7]. Both variants are equal as we can transform them into each other:

$$\begin{aligned} Q(\mathcal{C}) &= \textstyle\sum_{c \in \mathcal{C}} \Big(\sum_{u \in c} \sum_{v \in c} \frac{A_{uv}}{2m} - \sum_{u \in c} \sum_{v \in c} \frac{k_u k_v}{2m} \frac{1}{2m} \Big) \\ &= \textstyle\sum_{c \in \mathcal{C}} \Big(\sum_{u \in c} \sum_{v \in c} \frac{A_{uv}}{2m} - \big(\frac{\sum_{v \in c} k_v}{2m} \big)^2 \Big) \end{aligned} \quad \Big\downarrow \; \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} x_i x_j = \big(\sum_{k=1}^{n} x_k \big)^2$$

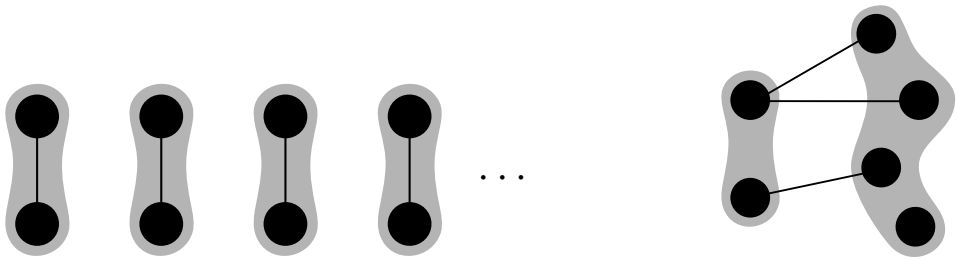
$$\boxed{Q(\mathcal{C}) = \sum_{c \in \mathcal{C}} \left(\frac{|E(c)|}{m} - \left(\frac{\sum_{v \in c} k_v}{2m} \right)^2 \right)} \tag{2.6}$$

where $|E(c)|$ or m_c denotes the number of edges with both ends in community c.

> **"This reveals an inherent trade-off: To maximize the first term, many edges should be contained in clusters, whereas the minimization of the second term is achieved by splitting the graph into many clusters with small total degrees each."** [22]

Value range

The modularity measure is zero if the fraction of edges within communities is no different from the respective fraction we would expect in a random graph. **We get a positive value if this fraction is higher than expected on the basis of chance.** In Figure 2.1b, Q was around 0.13 which is still slightly better than at random but nevertheless worse than the best result of $Q \approx 0.46$ for that particular graph.



(a) Infinite graph and community assignment yielding the upper limit of modularity: 1

(b) Bipartite graph and community assignment yielding the lower limit of modularity: −0.5

Fig. 2.3.: Upper and lower limit of modularity

In theory, the maximum possible modularity value is 1 for a graph with infinitely many communities, such that $a_c^2 \to 0$. We show this property for a graph with $t = |\mathcal{C}|$ communities, each forming a K_2 graph, i. e. two vertices sharing one edge (see Figure 2.3a). Note that this graph has t communities ($t = |\mathcal{C}|$) as well as t edges ($t = m$) in total. When t approaches infinity, we obtain for the modularity Q:

$$
\begin{aligned}
Q &= \lim_{t\to\infty} \tfrac{1}{2t} \textstyle\sum_{c=1}^{t} \sum_{u\in c} \sum_{v\in c} \Big(A_{uv} - \tfrac{k_u k_v}{2t} \Big) \\
&= \lim_{t\to\infty} \tfrac{1}{2t} \textstyle\sum_{c=1}^{t} \Big(2 \cdot 1 - 4 \cdot \tfrac{1\cdot 1}{2t} \Big) \qquad \big) \lim_{t\to\infty} \tfrac{1}{t} = 0 \\
&= \lim_{t\to\infty} \tfrac{1}{2t} \textstyle\sum_{c=1}^{t} 2 \\
&= \lim_{t\to\infty} \tfrac{2t}{2t} \\
&= 1
\end{aligned}
$$

As $\sum_{c\in\mathcal{C}}\sum_{u\in c}\sum_{v\in c} A_{uv} \leq 2m$ and $\sum_{c\in\mathcal{C}}\sum_{u\in c}\sum_{v\in c}\frac{k_u k_v}{2m} \geq 0$ for every graph partitioning, $Q = 1$ is indeed the upper limit of modularity.

The lower limit of modularity is $-1/2$ which is achieved using any bipartite graph $G(U, V, E)$ with the clustering $\mathcal{C} = \{ c_U, c_V \}$ [22], that is, each cluster encompasses all the vertices of one bipartition of G (the vertex sets U or respectively V). Figure 2.3b shows the bipartite graph $G(U, V, E)$ with $U = \{ 0, 1 \}$, $V = \{ 2, 3, 4, 5 \}$ and $E = \{ \{ 0, 2 \}, \{ 0, 3 \}, \{ 1, 4 \} \}$. The given community assignment yields a modularity of $-1/2$, while the maximum modularity for this particular graph is $Q \approx 0.44$.

A proof that $-0.5 \leq Q(\mathcal{C}) \leq 1$ holds true for any undirected and unweighted graph G and any community assignment (clustering) $\mathcal{C}$ is presented in [22]. Note however, that for a given graph the maximum achievable modularity is oftentimes not 1. Newman and Girvan claim that a typical range for modularity values is from 0.3 to 0.7 [11] with values above 0.3 indicating a "significant community structure in a network" [19].

Resolution limit

Fortunato and Barthélemy have shown that modularity suffers from a resolution limit, implying that optimizing for modularity does not necessarily reveal the actual community structure of the network [23]. Two communities might get combined into a larger one, yielding a higher modularity. This is the case for communities where $m_c < \sqrt{m/2}$, i. e. the number of edges inside community c is too small for the grouping of vertices to be recognized as one community. Note that the notion of "community" is defined in the "weak" sense as in [24]: "In a weak community the sum of all degrees within [the subgraph] is larger than the sum of all degrees toward the rest of the network". In the mathematical definition of modularity, however, what is considered a community is dependent on the size of the whole network (number of edges m) – which is not ideal.

The authors in [23] point out the importance of further investigation of the modules found by an algorithm that optimizes for modularity since one does not know if a claimed "module" is really just one community or a combination of multiple communities. Special care is required for communities with $m_c < \sqrt{2m}$ as they are likely to be composed of two or more smaller[4] ones (this number results from merging the two biggest, but

[4]The terms "small" and "big" are used here not to indicate the number of nodes, but the number of intra-edges in the respective communities. How nodes are distributed in the network does not have any impact on modularity, i. e. whether a community has more or less nodes is irrelevant.

still unrecognizable modules together). Yet, even bigger modules could suffer from this constraint of modularity if they are – as the authors call it – "fuzzy", meaning that they have an increasing number of external edges (edges to other modules), making the difference between internal and external edges smaller.

Figure 2.4 illustrates the resolution limit in an extreme example. The two cliques (complete graphs K_4) on the right are interconnected by one single edge ("bridge") and share just one edge with the rest of the network, yet they are not recognized as two individual communities; optimal modularity assigns all eight vertices to the same community. The clique K_{13} may be an arbitrary graph with ≥ 84 edges, so that the whole network consists of ≥ 98 edges. This ensures that for the K_4 cliques (composed of 6 edges each) the condition $m_c < \sqrt{m/2} = \sqrt{98/2} = 7 \Rightarrow m_c = 6 < 7$ is met. A complete graph K_n holds $\frac{n(n-1)}{2}$ edges, thus, our network consists of $\frac{13 \cdot 12}{2} + 2 \cdot \frac{4 \cdot 3}{2} + 1 + 1 = 92$ edges. $92 < 98$, yet still, $\sqrt{\frac{92}{2}} \approx 6.78$ which happens to be sufficiently close [5] to 7.0.

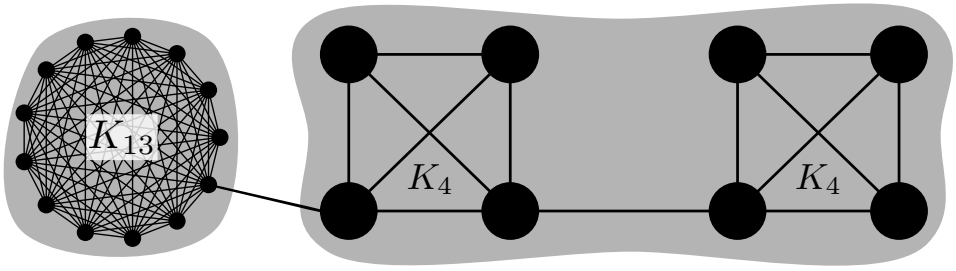


Fig. 2.4.: Network consisting of two K_4 and one K_{13} clique. The former are not correctly identified as individual communities when optimizing for modularity. $Q_{single} \approx 0.239$ (K_4 cliques identified as single communities), $Q_{pair} \approx 0.240$ (K_4 cliques grouped together as a pair).

While this example is contrived, even in real networks modularity "might miss important substructures" [23] as it exhibits an intrinsic scale dependent on the total number of edges. This may lead to counter-intuitive results. The proposed communities by an algorithm should therefore be further analyzed, e. g. by running the modularity optimization again on each individual module.

[5] Fortunato and Barthélemy point out that they calculate as if the variables were continuous, which they are not. Thus, for the given formula of the resolution limit, one has to test the closest integer values.

2.2. Genetic/Evolutionary algorithms

John Holland is remembered as the first to apply the biological process of evolution to computational problems in the late 1960s and early 1970s. His work *Adaption in Natural and Artificial Systems* [25, 26] is considered pioneering in this discipline that has been addressed by only a handful of researchers at that time. Over the time, GAs have found numerous successful applications in various disciplines ranging from circuit design [27], quickly finding good solutions to the Traveling Salesperson Problem (TSP) [28] to protein structure prediction [29] and finding the best parameters for a maximum likelihood estimation in order to better fit models describing cosmic rays to observed data [30]. In these cases, GAs unfold their full potential sampling effectively from a potentially large solution space where other methods might only find local maxima. Due to their implicit parallel nature, genetic algorithms can be implemented efficiently and therefore find optimal or near-optimal solutions very quickly where it might even be impractical to perform exhaustive search optimizations with traditional methods.

GAs are stochastic optimization methods [31] that draw on Charles Darwin's observations that organisms best adapted to their environment have a better chance to survive and breed offspring, a concept also known as "survival of the fittest/fitter". Through the process of sexual reproduction that inherently mutates the genetic material of the individuals, species adapt to their ever-changing environment. A variation in the chromosomes affects how well an individual "performs" in a given surrounding and, as a consequence, how likely it is to reproduce.

GAs borrow from these evolutionary principles as demonstrated in Algorithm 2.2.1 which outlines the basic steps of a GA. Solutions to the underlying problems are encoded in the chromosomes of individuals. Hereinafter, we will equate the words *chromosome* and *individual*. First, a **population of chromosomes** is generated (randomly in the absence of insights into the problem domain, otherwise heuristics can help) as a starting point for the search. The **fitness** of a chromosome is then evaluated based on an **objective function** modeling the environment – or in the case of a computational problem, the quality of the current solution (chromosome).

Darwin's theory of evolution involves natural selection which is imitated by a **selection** (or reproduction) function (line 5) supposed to choose the best individuals from the current population and duplicate them to form a mating pool. This can be done with

a selection proportionate to the fitness (fitter individuals have a higher chance to be selected). Another strategy is Elitism where only the best chromosome (or a few best chromosomes) is selected and copied until the parameterizable population size is reached.

We subsequently pick from the mating pool two individuals that undergo a **crossover** operator combining the genetic material of two parents to form a new offspring (child). This process may be repeated any number of times and should help to better sift through the solution space and to preserve genetic material from the parents (they encode good solutions to the problem as they were selected for reproduction). This ensures the concept of heritability [32] (next to reproduction and variation).

Darwin wrote in his book *On the Origin of species*, published in 1859: "for natural selection can act only by taking advantage of slight successive variations" [33]. Variations to the chromosome are introduced using a **mutation** operator that changes with an adjustable probability each position in a chromosome (also called *locus* or *gene* with an *allele* being an alternative form of a gene) to a new value. This in fact allows for a very efficient sampling of the solution space and is crucial for the GA to evade local optima – or speaking in metaphors: this rearrangement in the genetic material helps the child to survive in the environment it faces.

Selection (renew population with best solutions), crossover (combine solutions) and mutation (alter solutions) form one **generation** of a genetic algorithm. The selection makes sure that good solutions survive and bad solutions (according to the objective function) are eliminated. We hope for the GA to continuously improve the solutions through several generations. The process is repeated until a stopping criterion is met, e. g. an optimal or good-enough solution is found or a timer has elapsed.

It is worth mentioning that not every crossover or mutation might bear a *valid* solution in the sense that it encodes any solution (even a bad one) at all to the underlying problem. For example, for the TSP a mutation could produce a chromosome such that a city is not included in a trip. For this reason and to make the GA converge more quickly, it is necessary to develop custom operators tailored to the specific problem at hand.

Algorithm 2.2.1: High-level overview of a genetic algorithm (adapted from [34])

```
1 Function doGeneticAlgorithm():
2     population ← initializePopulation()
3     fitness ← calculateFitness(individual of population)
4     while Stopping criterion not met do
5         population ← Selection(population)
6         population ← Crossover(population)
7         population ← Mutation(population)
8         fitness ← calculateFitness(individual of population)
```

3. Louvain

The Louvain method – named after the University of Louvain where Blondel et al. developed the algorithm – finds communities by optimizing modularity (see section 2.1) locally for every node's neighborhood, then consolidating the vertices of newly found communities to super vertices and repeating the steps on the new, smaller graph [8]. This multi-step algorithm ends when modularity does not improve in one pass anymore. It has been shown to yield state-of-the-art results with very good performance which is linear in the number of edges m in the graph [35].

3.1. Algorithm

The Louvain algorithm is demonstrated by means of Figure 3.1. The shape of the eight vertices indicates one of the two communities that vertex belongs to.

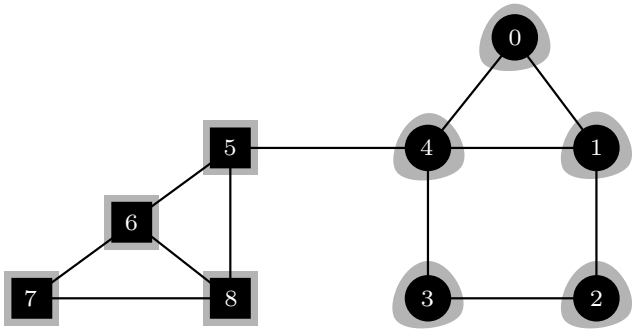
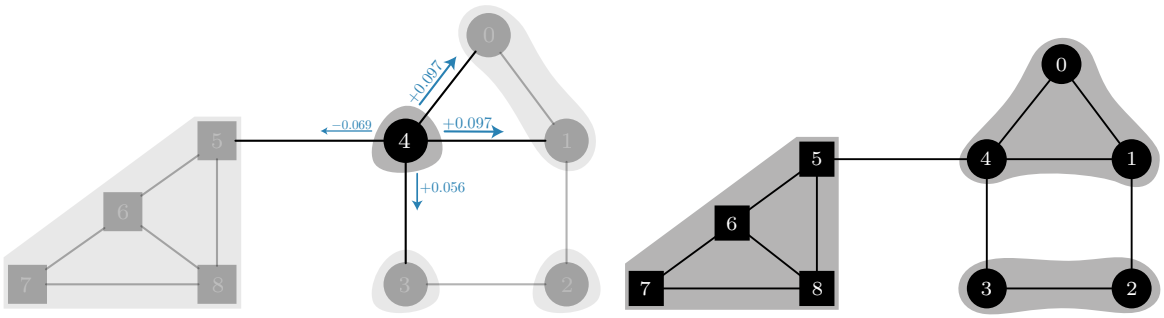


Fig. 3.1.: Initial singleton communities

I. Modularity Optimization

Initially, every vertex of the graph is assigned to its own community ("singleton" community). The first step of one Louvain pass is essentially to apply Newman's agglomerative hierarchical clustering method presented in [16] and improved by Clauset, Newman, and Moore in [19]:

> "[S]tarting with each vertex being the sole member of a community of one, we repeatedly join together the two communities whose amalgamation produces the largest increase in Q." [19]



(a) Delta modularity calculation for vertex 4 (b) Result of local optimization

Fig. 3.2.: Louvain optimization phase in first pass

In the Louvain method, this is done by iterating over all vertices (in randomized order): a vertex u is then "moved" to the community of one of its neighbors $v \in N_G(u)$, so that the modularity gain ΔQ is maximized. If no positive gain is possible, u remains in its singleton community. In Figure 3.2a, some vertices are already merged, e. g. vertices 5,6,7 and 8 on the left and vertices 0 and 1 on the right. At this point, we consider vertex $u = 4$ and its neighbors $N_G(4) = \{0, 1, 3, 5\}$. The highest increase is $\Delta Q \approx 0.097$ when the community of vertices 0 and 1 accommodates vertex 4, hence 4 is "moved" to this community. An efficient calculation of ΔQ is derived in section 3.2. We consider vertices multiple times until modularity cannot be increased anymore which corresponds to a local maximum. Figure 3.2b illustrates the community assignments as result of the first local optimization phase.

II. Community aggregation

In the second phase, the communities found are replaced by super vertices. This results in a new graph where vertices are the communities of phase I. Consider the middle layer of Figure 3.3 showing the outcome of the first pass after phase I and II. Edges now have a weight assigned to them:

- Edges *between* communities (inter-community edges) are condensed to one edge between the communities with edge weight equal to the sum of the weight of all previous edges between the communities.
- Edges *within* communities (intra-community edges) result in self-loops with a weight of value two in order to account for the undirectedness (see calculation in section 3.2).

Termination

These two phases make one pass of the Louvain algorithm. The first phase works directly on the graph that emerges from the second phase (the new super vertices are considered to lie in their own singleton communities). Passes are executed repeatedly while modularity is always calculated with respect to the original graph. The algorithm ends when modularity cannot be increased anymore (hopefully corresponding to a global maximum), which is often the case after a handful of passes, even for large networks [8].

Capturing the resulting community-vertex assignment after each pass produces a hierarchy (see Figure 3.3) since the second step creates super-vertices based on the previous vertices of a community and thus the local optimization phase essentially finds communities of communities. Blondel et al. remark that this is favorable in order to reveal hierarchical structures present in many large networks. They also point out that the resolution limit discussed in section 2.1 is mitigated as vertices are moved one after another, yielding a low probability of merging two distinct communities. Even if the super vertices of the respective communities are merged in later passes, the resulting hierarchy helps in identifying on which "organization level" this merge took place [8].

Algorithm overview

The complete algorithm as pseudo-code is shown in Algorithm 3.1.1. Inside executeLouvain(), we first call `preparePasses()` (line 2) which makes the vertex ids contiguous (0,1,2,3 instead of 0,42,142,143) and ensures we ignore isolated vertices. The while loop in line 4 constitutes one pass where we first try to optimize modularity in phase I (line 6) and subsequently construct the graph for the next level of the hierarchy in phase II (line 8). The current hierarchy H is returned in line 7 when modularity cannot be optimized anymore.

Modularity optimization is carried out in the function `tryOptimizeModularity()`. While modularity can be increased (line 26), we iterate over all vertices in a random order (line 14), remove them from their current community c_v (line 16) and insert them into the respective best neighbor community (line 22). The modularity gain is therefore calculated in line 18 by calling `calculateModularityGain()`. The following sections address the question of how to efficiently compute this delta increase ΔQ in modularity. If no best neighbor community was found ($\Delta Q \leq 0$ for all neighbor communities), the vertex stays in its current community c_v (in this case $bestCommunity = c_v$ in line 22).

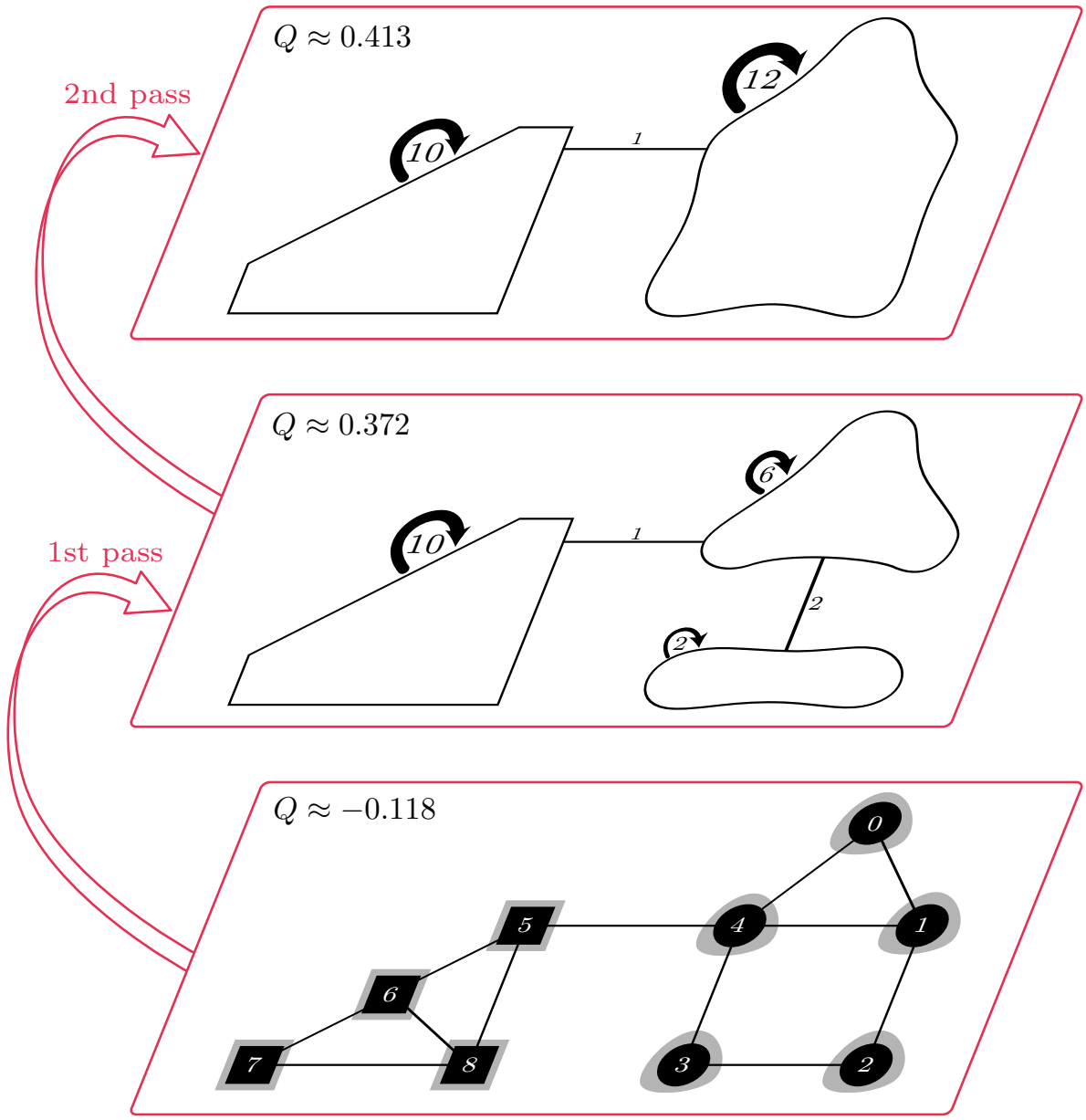


Fig. 3.3.: Resulting Louvain hierarchy for the sample graph (Figure 3.1). Communities of the previous pass become the new super vertices. Note that we shape these new vertices according to the previous community (either sharp or rounded) to make clear that they arise from previous super-vertices (communities).

Algorithm 3.1.1: Louvain algorithm

```
   Input: G: Input graph
   Output: H: Community assignment hierarchy
1  Function executeLouvain:
2      preparePasses()
3      H ← ∅
4      while True do
5          calculateModularity() ▷ We also store Q of the current pass (level)
6          improvement: bool ← tryOptimizeModularity(H.lastElement)
7          if not improvement then return H
8          H ← H ∪ getNextLevelGraph()

   Input: G: Current graph
   Output: improvementInPass: bool
9  Function tryOptimizeModularity:
10     improvementInPass: bool ← false
11     improvementInIteration: bool ← false
12     do
13         improvementInIteration ← false
14         foreach v ∈ V(G) in random order do
15             bestCommunity: int ← c_v
16             removeVertexFromCommunity(v, c_v)
17             foreach Neighboring vertex w of v: w ∈ N_G(v) do
18                 increase: double ← calculateModularityGain(v, c_w)
19                 if increase > bestIncrease then
20                     bestCommunity ← c_w
21                     bestIncrease ← increase
22             insertVertexIntoCommunity(v, bestCommunity)
23             if bestCommunity ≠ c_v then
24                 improvementInIteration ← true
25                 improvementInPass ← true
26     while improvementInIteration
27     return improvementInPass
```

3.2. Delta modularity calculation (singleton)

In the modularity optimization phase, we only rely on local information. Recalculating the global modularity value for every possible neighbor of each vertex would significantly degrade the performance of the Louvain algorithm. This is why Blondel et al. employ a delta modularity formula that can be applied to millions of nodes. In this section, we will explain and derive the formula and show how it can be simplified for usage in a program.

In Equation 2.6, we have seen that the modularity $Q(c)$ for a community c is given by:

$$Q(c) = \frac{|E(c)|}{m} - \left(\frac{\sum_{v \in c} k_v}{2m}\right)^2 \tag{3.1}$$

Next, we look closely at the process of "moving" a vertex u to the community of one of its neighbors $N_g(u)$. We assume the vertex to be located in a singleton community. The sole formula in [8] is for this case, where we remove u from its singleton community and then insert it into the neighbor's community. This is actually only applicable for the first iteration of the modularity optimization phase where in the beginning of every new pass we deal with singleton communities. The authors state that they use a similar formula to calculate the modularity change when u is removed from *any* community (also those with more than one vertex in it), but do not reveal the expression used. We will therefore derive the generalized version in section 3.3.

We consider the case where we move a vertex u from its singleton community $\dot{c}_u$ to any other community c. The modularity change is then given by

$$\Delta Q(u, \dot{c}_u, c) = Q'(c) - Q(c) - Q(\dot{c}_u) \tag{3.2}$$

where $Q(c)$ is the quality of community c *before* the merge (and hence without vertex u) and $Q'(c)$ is the quality of c *after* the merge (and hence after vertex u was integrated into community c). As the singleton community does not exist anymore after the merge, we also subtract the modularity of the singleton community $Q(\dot{c}_u)$. In the following, let the prime symbol always denote the state of a variable after the merge.

As we deal with weighted graphs in Louvain, we use $\Sigma_c/2$ instead of $|E(c)|$, where Σ_c **is twice the sum of the weights of edges inside community**[1] c:

$$\Sigma_c = \sum_{u \in c} \sum_{v \in c} A_{uv} \tag{3.3}$$

Furthermore, we define $\Sigma_{\hat{c}}$ **to be the sum of the weights of edges incident to vertices**[2] **in** c. This can also be understood as the sum of the weighted vertex degrees of vertices in c:

$$\Sigma_{\hat{c}} = \sum_{v \in c} k_v \tag{3.4}$$

We can now redefine modularity $Q(c)$ for a community c (based on Equation 3.1):

$$Q(c) = \frac{\Sigma_c}{2m} - \left(\frac{\Sigma_{\hat{c}}}{2m}\right)^2 \tag{3.5}$$

This in turn allows us to transform Equation 3.2 into:

$$\Delta Q(u, \dot{c}_u, c) = \left[\frac{\Sigma'_c}{2m} - \left(\frac{\Sigma'_{\hat{c}}}{2m}\right)^2\right] - \left[\frac{\Sigma_c}{2m} - \left(\frac{\Sigma_{\hat{c}}}{2m}\right)^2\right] - \left[\frac{\Sigma_{\dot{c}_u}}{2m} - \left(\frac{k_u}{2m}\right)^2\right] \tag{3.6}$$

which is the equation presented in the original Louvain paper [8]. If we consider that after the merging process, the **sum of the weights of the edges between node** u **and community** c, that we will denote by[3] $k_u^{\to c}$, now adds to the community c accommodating vertex u, we obtain

$$\Sigma'_c = \Sigma_c + 2k_u^{\to c} \tag{3.7}$$

Likewise, we find that the weighted vertex degree k_u adds to the total weighted vertex degree of c after the merge, thus:

$$\Sigma'_{\hat{c}} = \Sigma_{\hat{c}} + k_u \tag{3.8}$$

Next, we rewrite and simplify expression 3.6. Note that we use $\Sigma_{\dot{c}_u} = 0$, as there are no edges inside a singleton community $\dot{c}_u$ (we do not allow self-loops in the original graph).

[1] Σ_c is denoted by Σ_{in} in the Louvain paper.
[2] This choice is made in accordance with [36]. In the Louvain paper, $\Sigma_{\hat{c}}$ is denoted by Σ_{tot}.
[3] This is done in conformity with [37]. The arrow does not indicate a direction; our matter at hand is still an undirected graph.

$$
\begin{aligned}
\Delta Q(u, \dot{c}_u, c) &= \left[\frac{\Sigma_c + 2k_u^{\to c}}{2m} - \left(\frac{\Sigma_{\hat{c}} + k_u}{2m}\right)^2\right] - \left[\frac{\Sigma_c}{2m} - \left(\frac{\Sigma_{\hat{c}}}{2m}\right)^2\right] - \left[\frac{\Sigma_{\dot{c}_u}}{2m} - \left(\frac{k_u}{2m}\right)^2\right] \\
&= \left[\frac{\Sigma_c + 2k_u^{\to c}}{2m} - \left(\frac{\Sigma_{\hat{c}} + k_u}{2m}\right)^2\right] - \left[\frac{\Sigma_c}{2m} - \left(\frac{\Sigma_{\hat{c}}}{2m}\right)^2 - \left(\frac{k_u}{2m}\right)^2\right] \\
&= \frac{2k_u^{\to c}}{2m} - \left(\frac{\Sigma_{\hat{c}}}{2m}\right)^2 - 2\frac{\Sigma_{\hat{c}}}{2m}\frac{k_u}{2m} - \left(\frac{k_u}{2m}\right)^2 + \left(\frac{\Sigma_{\hat{c}}}{2m}\right)^2 + \left(\frac{k_u}{2m}\right)^2 \\
&= \frac{k_u^{\to c}}{m} - \frac{\Sigma_{\hat{c}} k_u}{2m \cdot m} \\
&= \frac{1}{m}\left(k_u^{\to c} - \frac{\Sigma_{\hat{c}} k_u}{2m}\right)
\end{aligned}
\tag{3.9}
$$

Finally, we have

$$
\boxed{\Delta Q(u, \dot{c}_u, c) \propto k_u^{\to c} - \frac{\Sigma_{\hat{c}} k_u}{2m}} \tag{3.10}
$$

where we dropped the constant $1/m$ from Equation 3.9 since we only compare different modularity increases $\Delta Q(u, \dot{c}_u, c)$ with each other and therefore merely require a relative measure. This saves us one division in the algorithm. After one complete pass, the new global modularity is calculated using Equation 2.6 (Algorithm 3.1.1, line 5).

Equation 3.10 is applied in Algorithm 3.2.1 to efficiently calculate the delta modularity gain ΔQ (line 12). To remove a vertex u from its previous community c_u or to insert it into a new community c, only Σ_c and $\Sigma_{\hat{c}}$ – that we store as class member variables – have to be updated. Σ_c is adjusted for the global modularity calculation and not for the delta modularity ΔQ. Note that we can precalculate the sum of the weights of edges between u and community c_u or c ($k_u^{\to c_u}$ and $k_u^{\to c}$) inside the for-loop starting in line 14 (Algorithm 3.1.1) before calling the remove- or insert-function. This is crucial for the speed of Louvain as ΔQ has to be computed frequently. As stated above, we also dropped the factor $1/m$ in line 12 (Algorithm 3.2.1) to save one division. The calculation of global modularity starts in line 13 and draws upon Equation 3.5. We do not omit the factor $1/2m$ here (line 17) in order to obtain the absolute global modularity Q.

In the beginning of every Louvain pass, a new modularity object is constructed and initialized according to `initializeModularity()` (line 1). We iterate over all vertices (line 2) of the current graph (that contains super-vertices starting with the second level), find out their community c_v and update Σ_{c_v} and $\Sigma_{\hat{c}_v}$.

Algorithm 3.2.1: Modularity calculation

Input: G: Graph
Function `initializeModularity()`:
  foreach $v \in V(G)$ **do**
    $\Sigma_{c_v} \leftarrow \Sigma_{c_v} + k_v^{\to c}$
    $\Sigma_{\hat{c}_v} \leftarrow \Sigma_{\hat{c}_v} + k_v$

Input: u: Vertex, c_u: Old community of u
Function `removeVertexFromCommunity()`:
  $\Sigma_{c_u} \leftarrow \Sigma_{c_u} - 2k_u^{\to c_u}$
  $\Sigma_{\hat{c}_u} \leftarrow \Sigma_{\hat{c}_u} - k_u$

Input: u: Vertex, c: New community of u
Function `insertVertexIntoCommunity()`:
  $\Sigma_c \leftarrow \Sigma_c + 2k_u^{\to c}$
  $\Sigma_{\hat{c}} \leftarrow \Sigma_{\hat{c}} + k_u$

Input: u: Vertex, c: New community of u
Output: ΔQ: double
Function `calculateModularityGain()`:
  return $k_u^{\to c} - \frac{\Sigma_{\hat{c}} k_v}{2m}$ ▷ see Equation 3.10

Output: Q: double
Function `calculateModularity()`:
  Q: double $\leftarrow 0.0$
  foreach Community $c \in \mathcal{C}$ **do**
    $Q \leftarrow Q + \Sigma_c - \frac{\Sigma_{\hat{c}}^2}{2m}$ ▷ see Equations 3.5 and 2.6
  return $Q/2m$

3.3. Delta modularity calculation (generalized)

Here we generalize the delta modularity calculation presented in section 3.2 for the case where we move a vertex u from *any* community (not just a singleton community) to any other community c. We proceed by removing u from the old community and putting it into its own singleton community. The latter is done, so that the derived Louvain modularity difference formula can be reused, which is only applicable to the case where one moves a vertex from a *singleton* community to any other community. With this more generalized equation, we fill the gap caused by the absence of this formula in the original Louvain paper [8]. The generalized expression is then used in conjunction with the genetic mutation operators (Chapter 5) as means of an efficient delta modularity calculation.

First, we calculate the modularity difference when removing a vertex u from its old community c_u (not required to be a singleton community) and inserting it into its own singleton community c_u. As before, we prohibit self-loops in the original graph, hence $\Sigma_{\dot{c}_u} = 0$. We won't be as elaborate this time due to the rationale being very similar:

$$\begin{aligned}
\Delta Q(u, c_u, \dot{c}_u) &= Q'(c_u) - Q(c_u) + Q_{\dot{c}_u} \\
&= \left[\frac{\Sigma'_{c_u}}{2m} - \left(\frac{\Sigma'_{\hat{c}_u}}{2m}\right)^2\right] - \left[\frac{\Sigma_{c_u}}{2m} - \left(\frac{\Sigma_{\hat{c}_u}}{2m}\right)^2\right] + \left[\frac{\Sigma_{\dot{c}_u}}{2m} - \left(\frac{k_u}{2m}\right)^2\right] \\
&= \left[\frac{\Sigma_{c_u} - 2k_u^{\to c_u}}{2m} - \left(\frac{\Sigma_{\hat{c}_u} - k_u}{2m}\right)^2\right] - \left[\frac{\Sigma_{c_u}}{2m} - \left(\frac{\Sigma_{\hat{c}_u}}{2m}\right)^2\right] - \left(\frac{k_u}{2m}\right)^2 \\
&= -\frac{k_u^{\to c_u}}{m} - \left(\frac{\Sigma_{\hat{c}_u}}{2m}\right)^2 + 2\frac{\Sigma_{\hat{c}_u}}{2m}\frac{k_u}{2m} - \left(\frac{k_u}{2m}\right)^2 + \left(\frac{\Sigma_{\hat{c}_u}}{2m}\right)^2 - \left(\frac{k_u}{2m}\right)^2 \\
&= -\frac{k_u^{\to c_u}}{m} + \frac{2k_u\Sigma_{\hat{c}_u}}{(2m)^2} - \frac{2k_u^2}{(2m)^2} \\
&= \frac{1}{m}\left(-k_u^{\to c_u} + \frac{k_u(\Sigma_{\hat{c}_u} - k_u)}{2m}\right)
\end{aligned} \tag{3.11}$$

We bring together Equation 3.9 and 3.11 to find the modularity **when moving a vertex u from its previous community c_u to any other community** c. This is done by first moving u from c_u into its own singleton community $\dot{c}_u$, then moving it from there

into the target community c:

$$\begin{aligned}\Delta Q(u, c_u, c) &= Q(c_u, \dot{c}_u) + Q(\dot{c}_u, c) \\ &= \frac{1}{m}\left(k_u^{\to c} - k_u^{\to c_u} + \frac{k_u(\Sigma_{\hat{c}_u} - k_u) - \Sigma_{\hat{c}} k_u}{2m}\right) \\ &= \frac{1}{m}\left(k_u^{\to c} - k_u^{\to c_u} + \frac{k_u(\Sigma_{\hat{c}_u} - \Sigma_{\hat{c}} - k_u)}{2m}\right)\end{aligned} \tag{3.12}$$

Finally, for the generalized formula, we have

$$\boxed{\Delta Q(u, c_u, c) \propto (k_u^{\to c} - k_u^{\to c_u}) + \frac{k_u(\Sigma_{\hat{c}_u} - \Sigma_{\hat{c}} - k_u)}{2m}} \tag{3.13}$$

The delta modularity calculation for removing from a singleton community (Equation 3.10) can be derived from this formula by assuming that c_u is a singleton community. As we do not allow self-loops, $k_u^{\to c_u} = 0$. Moreover, the sum of the weights of edges incident to vertices in c_u is equal to the vertex degree: $\sum_{\hat{c}_u} = k_u$, which gives us:

$$\begin{aligned}\Delta Q(u, \dot{c}_u, c) &\propto (k_u^{\to c} - 0) + \frac{k_u(k_u - \Sigma_{\hat{c}} - k_u)}{2m} \\ &= k_u^{\to c} - \frac{\Sigma_{\hat{c}} k_u}{2m} \qquad \text{see Equation 3.10}\end{aligned} \tag{3.14}$$

The implementation is identical in large parts with Algorithm 3.2.1: the remove- and insert-functions and the calculation of the global modularity stay the same, while the function to compute ΔQ changes as follows:

Algorithm 3.3.1: Delta Modularity calculation (general case)

Input: u: Vertex, c: New community of u

Output: ΔQ: double

```
1 Function calculateModularityGain():
2     return (k_u^{→c} − k_u^{→c_u}) + k_u(Σ_{ĉ_u} − Σ_ĉ − k_u)/2m      ▷ see Equation 3.13
```

Again, we could precalculate $k_u^{\to c}$ and $k_u^{\to c_u}$. Yet, we will only use the generalized delta modularity formula in the context of genetic mutations where we do not greedily check for the best neighbor community as Louvain does, but instead assign vertices to (possibly arbitrary) new communities one after another. Therefore, the precalculation is of no advantage here because the value cannot be reused anywhere.

3.4. Further implementation details

We implemented the Louvain algorithm in C++ for the Graph engine in SAP HANA and were guided by the original C++ implementation of Blondel et al. in [38]. Even though the overall method is straightforward, more than 700 lines of code (without header files) were needed to fully implement the algorithm and optimize its performance. We already covered some optimizations in the previous sections, e. g. dropping a factor or precalculating values to reuse them in subsequent calls. The C++ code is not shown here as the algorithm was already discussed in great depth in the previous sections.

The vertex-community assignments are stored inside a vector which is indexed by the vertex id and has entries representing the assigned community (see section 4.1 where this encoding is used for the representation of a genetic chromosome). The final mapping hierarchy is then a vector of these vertex-community assignment vectors, where the first vector describes the original singleton community assignment and the last vector represents the last ("upmost") hierarchy level. The size of this level (number of entries) gives the number of communities found by Louvain. To obtain the final vertex-community assignment for all vertices based on the last level, we implement a utility function that traverses the hierarchy bottom-up to find the final community of a vertex (see Algorithm 3.4.1). For every vertex $v \in V(G)$ (line 2), we assign the community of the first ("bottommost") level (line 3) and subsequently iterate through the upper levels (line 4) to follow the super-vertex community assignments up to the last level. We do so by overwriting the previous assignment c_v in the loop with the new community found in the next-level hierarchy (line 5).

Algorithm 3.4.1: Traverse the vertex-community hierarchy bottom-up to find the community of every vertex

Input: G: Graph,
hierarchy: Community Assignment Hierarchy (Vector of vectors)

```
1 Function calculateCommunityAssignmentsBasedOnHierarchy():
2     foreach v ∈ V(G) do
3         c_v ← hierarchy[0][v]
4         for level = 1; level < |hierarchy|; level ← level + 1 do
5             c_v ← hierarchy[level][c_v]
```

The code for the Louvain method is split into four different classes:

- `LouvainGraph`: Wrapper around the Graph API of the SAP HANA Graph Engine. It provides methods such as `getAdjacentEdges()`, `getEdgeWeight()` or `graphToLouvainEdges()`, with the latter being responsible to make the vertex ids contiguous and adjust the edges accordingly. This function is called inside the `preparePasses()` function (Algorithm 3.1.1, line 2).
- `Louvain`: The high-level Louvain implementation as presented in the function `executeLouvain()` in Algorithm 3.1.1. We orchestrate the execution of the two phases here. This class is used from outside by passing in a Graph object and optionally edge weights as lambda function of vertices. Note, however, that we will set the edge weight to 1 for all vertices throughout this work for reasons of simplicity.
- `LouvainIteration`: Implementation of the two phases implemented in the functions `tryOptimizeModularity()` (compare with Algorithm 3.1.1, line 6) and `getNextLevelGraph()` (see Algorithm 3.1.1, line 8). This class contains a Modularity object as member variable.
- `Modularity`: Class to use the Newman-Girvan modularity as quality function for Louvain. Implements methods such as `remove()` (to remove a vertex from its current community and update the respective parameters related to modularity), `insert()` (to insert a vertex into a new community and update the respective parameters) and `calculateModularity()` (Algorithm 3.2.1).

The implementation was tested on several small graphs using parameterized tests of Google's C++ unit test framework. We also compare the modularity Q and the assigned communities in the last hierarchy level against the NetworkX Python results [39] to ensure we are on par.

4. Methodology for Louvain & genetic approaches

In this chapter, we present several choices made for the genetic algorithms, namely how the vertex-community assignments are encoded as chromosomes, how the initial population is created and how offsprings (children) are locally optimized to improve convergence of the algorithms.

4.1. Encoding of vertex-community assignments

We will use a common encoding for the representation of vertex-community assignments in the genotype. In this encoding, a list is used where the index i indicates the vertex v_i in the graph and the element at that index denotes the corresponding community for that vertex. Although formally correct we should write c_i for the community of the vertex at position i in the list, we abbreviate this with c_v. A partition of the graph into communities can thus be represented by a list: $\mathcal{C} = [c_{v_1}, c_{v_2}, c_{v_3}, \ldots, c_{v_n}]$ with $n = |V(G)|$ and $c_v \in \{1, 2, \ldots, |\mathcal{C}|\}$. Two vertices v and w are in the same community iff $c_v = c_w$. Figure 4.1 shows the chromosome for a conceived network of eight nodes. The colors are stored as integer community labels; in the example, we could write the chromosome as the following array: $[0, 0, 1, 0, 0, 1, 2, 3]$.

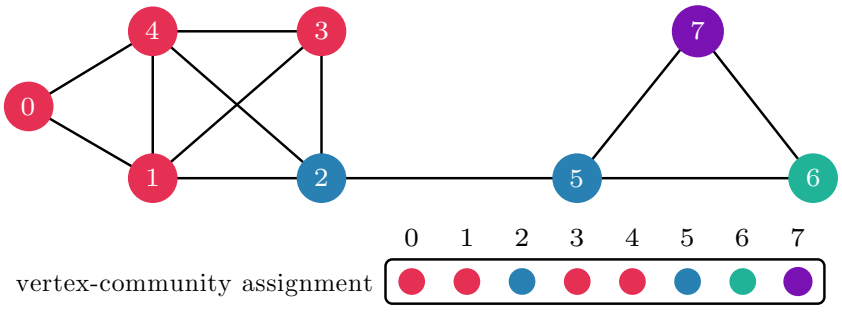


Fig. 4.1.: Example for a vertex-community encoding (chromosome)

4.2. Initialization heuristics

Genetic algorithms have in common that a population must be initialized in the beginning which is then improved (see Algorithm 2.2.1, line 4) throughout multiple generations until a stopping criterion is met. Oftentimes, the initial population is chosen randomly, especially if no prior information about the problem domain and the respective fitness landscape is available. A randomly initialized population can be advantageous in the sense that the solution space is sampled effectively in the beginning, i. e. the genetic algorithm can start from many different individuals out of which some might already be close to local or even global optima.

However, "the chance for a population to cover promising regions of the search space decreases as increasing the dimension of the search space" [9]. This is why a genetic algorithm might benefit from a good initialization heuristic that can already provide promising initial guesses and therefore increase the chance of finding the global optimum or converging to a local or global optimum more quickly. Borhan Kazimipour, Xiaodong Li, and A.K. Qin present in [9] a review of different population initialization techniques for evolutionary algorithms. They categorize these techniques based on randomness, compositionality and generality including sub-categories. We adopt this grouping and will classify the used heuristics for the genetic algorithms accordingly.

As first very simple strategy, the chromosomes are initialized (seemingly) randomly using a Pseudo-Random Number Generator (PRNG). We do this by assigning a random community in the range $[0, 2 \cdot \text{number of communities in the fittest chromosome}]$ to every vertex. This is to allow for greater variation as with this range it is less likely for two vertices to get assigned the same community. With this heuristic, every vertex could possibly end up in its own singleton community although this is very improbable. By relabeling the community numbers, they can remain in the range $[0, n)$. This random population initialization is employed in [10] and in a biased version in [15] where the authors additionally pick random vertices and assign their community to all neighbors. In terms of the categories presented in [9], this heuristic falls in the stochastic random number generator class for randomness, is non-composite (not composed of multiple steps that could be applied individually as an initializer) and generic since it does not leverage any problem-specific knowledge (it acts as if the underlying problem was a black-box puzzle).

As second initialization heuristic we will use the Louvain method [8] presented in Chapter 3. To the best of our knowledge, applying Louvain as initialization for genetic algorithms has not been attempted in literature yet. We have chosen this approach because – as can be seen from the evaluation in Chapter 7 – Louvain is fast while producing very good results even for big networks. We would like to evaluate whether the genetic mutations are capable of further improving the modularity values achieved by Louvain. In terms of [9], this initialization method falls in the *stochastic random number generator* category since Louvain generates different results for different initial seeds that determine in which order the vertices of the graph are considered. Furthermore, we would classify Louvain as composite, multi-step algorithm according to [9] as it creates a hierarchy and incrementally refines the results. Individual steps could also be used as standalone initializer, e. g. putting every vertex in its own singleton community in the beginning (without performing the rest of Louvain) is a valid initialization heuristic (although not necessarily a very good one). With regards to *generality*, we categorize Louvain as application-specific as this method is only applicable to community detection exploiting specific knowledge of that domain, e. g. how modularity is calculated and how to positively affect it by altering the vertex-community assignments.

As a result of the Louvain algorithm, we get a vertex-community assignment hierarchy (see Figure 3.3), hence the genetic algorithms could possibly operate on a smaller graph consisting of super-vertices instead of the original graph. However, we choose to still use the original graph with the assignments based on the last level of the hierarchy according to Algorithm 3.4.1. This is done in order to not be overly influenced by the initialization and still have the possibility to alter the communities found by Louvain in the original graph instead of being restricted to the communities found at a specific hierarchy level. The initialization should only serve as a good starting point but not limit the exploration of the solution space in any way. Another variant could consist of only calculating Louvain up to a specific hierarchy level and using the resulting vertex-community assignments (again applied to the original graph via Algorithm 3.4.1). Yet, we have chosen to calculate up to the last level when Louvain has found a local (or possibly global) optimum since the modularity between passes usually improves significantly and the number of total passes (hierarchy levels) is fairly low, e. g. always smaller than 5 in the Louvain paper [8].

Due to the Louvain method being stochastic in nature (the order in which vertices are considered is determined randomly), it will return different results for multiple executions.

This could be a problem when we get a – compared to other runs – worse solution and try to improve it by genetic operators. To avoid this less than ideal initial situation, we run Louvain multiple times and pick the best result as initialization for the genetic part. However, Blondel et al. state that "[p]reliminary results on several test cases seem to indicate that the ordering of the nodes does not have a significant influence on the modularity that is obtained" [8] and we can confirm this behavior in our experiments by means of a low variance obtained (see section 7.4). This is why we only execute Louvain three times to roughly separate worse solutions from good ones while keeping the overall runtime small.

4.3. Local optimization of offsprings

In order to avoid obvious flaws in the offsprings, which can arise from applying the mutation operators discussed in Chapter 5, we employ a quick local optimization resulting in a smoothing of the "fitness landscape". Figure 4.2 clarifies this point by means of an example with $y(x) = -x^2 - 100\sin(x - \frac{\pi}{2}) + 200$ as contrived objective function where we want to find the global maximum. When the fitness of an offspring evaluates to a point on the curve that is not a local maximum we can resort to local optimization techniques. In the example, the current individual's fitness is marked by the red point. With local optimization, the fitness increases to the next local maximum. If this procedure is applied after each mutation, we have essentially simplified the fitness landscape to the parabola marked by the dashed line. While the local optimization certainly requires additional computational time, we hope that faster convergence to the global maximum will compensate for this. [40] serves as an example where local optimization strategies have been successfully applied to the TSP problem.

Our local optimization heuristic for genetic community detection is shown in Algorithm 4.3.1. We essentially employ Lehnerer's strategy, which is not evident from his paper [10] but only from his code on GitHub [42]. Sensible communities consist of more than one node. This is because we do not allow self-loops in the original network and therefore a singleton community cannot contain any edge with both ends in it. This causes the minuend in Equation 2.6 to be 0, while the subtrahend is positive. Hence, singleton communities have a negative impact on modularity, and we try to circumvent them for this reason. To do so, we include an additional "Clean up" step after having

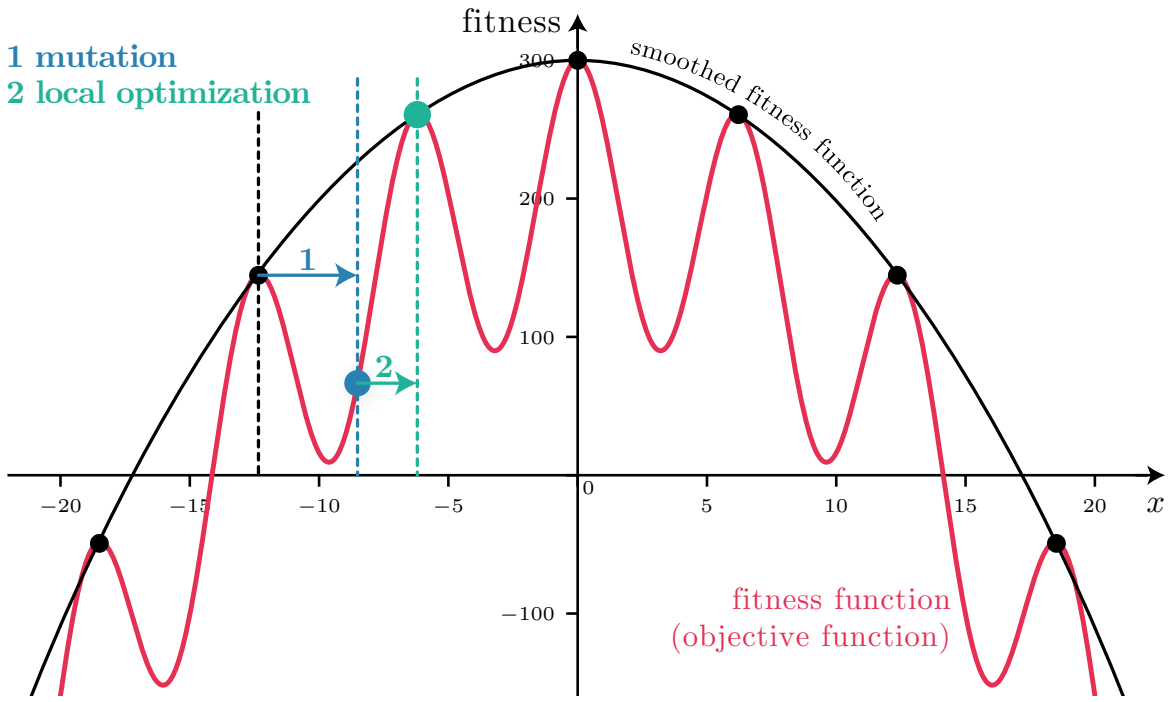


Fig. 4.2.: Example for a local optimization of an individual (adapted from [41])

applied mutation operators. We check for singleton communities (line 3) and allot to the respective vertex (line 4) the community of a random neighbor (line 6).

Algorithm 4.3.1: Local optimization of offsprings

Input: *vertices*: Set of vertices

```
1 Function localOptimization():
2     foreach Community c ∈ C do
3         if Size of community c is 1 then
4             v ← Vertex in singleton community c
5             randomNeighbor ← getRandomElement(N_G(v))
6             c_v ← c_randomNeighbor          ▷ c_v = c before this line
```

Finding singleton communities requires to search elements (communities) which only occur once in the vector representing the chromosome. This operation is therefore linear in the number of vertices of the graph: $\mathcal{O}(n)$. Additionally, we need to access the adjacent vertices of v – which is implemented with a runtime in the complexity class $\mathcal{O}(1)$. Note that Algorithm 4.3.1 does not guarantee that a local optimum of modularity is reached every time, yet it increases the probability of this event (or at least moves in the direction of the next local maximum) by eliminating singleton communities.

4.4. Selection

A commonly used notation to describe the selection strategy of a GA is (μ, λ), where μ denotes the number of parents and λ is the population size, i. e. the number of children newly created in each generation. In a (μ, λ) GA, the children replace their parents even if the best child might yield worse results than the parent generation. This is why we will avail ourselves of a $(\mu + \lambda)$ strategy where the best children together with their parents are used as pool from which the best μ candidates are chosen for the next generation.

5. Mutations

Mutation operators are key to achieve genetic variation and effectively explore the solution space in order to find a global maximum or at least a very good local one. Mutations alter the alleles of a chromosome; here we present several ways in which this can be done. The first two mutations are taken from paper [10], while the triangle chain and Louvain CRATER are novel operators we have not encountered in literature yet (to the best of our knowledge).

5.1. Random community

In this straightforward, yet important mutation operator described in Algorithm 5.1.1, a random subset of vertices $M \subseteq V(G)$ gets assigned random communities. In terms of a chromosome, this can be seen as a multi-point mutation where we select random alleles and for each of them assign a new random number representing a community. Through this approach, we avoid being caught in local maxima. Visually, it corresponds to jumping to a different point in the solution space that might be located far apart from the one previously considered. New communities can arise by applying this mutation.

[10] and [15] use this operation. In the latter, the number of vertices obtaining a new community is fixed to half the size of the graph, i. e. $|M| = 0.5n$, while in [10], Lehnerer[1] chooses this number randomly in the range $[0, 0.1n]$. In this thesis, we will adopt this approach but use the range $[1, 0.1n]$ (line 7) as to not produce mutations that keep the chromosome unaltered. For the assigned community numbers, Lehnerer chooses the range $[0, 2 \cdot \text{number of communities in the fittest chromosome}]$ (line 12). This has the same reason as for the random initialization heuristic described in section 4.2.

The function to sample random vertices in the graph starts in line 1, where we fill an initially empty set with random vertices from the graph (line 4) until its size is equal to the number of vertices (line 3) given as parameter to the function. Note that we sample from

[1] See the code and data used in [10] on GitHub: [42] (which was not linked in the original paper)

the discrete uniform distribution with probability mass function $P(i|a,b) = \frac{1}{b-a+1}, a \leq i \leq b$ [43] (see `drawUniformly`$([a,b])$ in the pseudocode). *Since all following mutations will likewise apply changes to a random subset of vertices of the graph, we include the "full" mutation here once (see the function `randomCommunityMutation()`), but restrict ourselves to the adjustments for one single vertex in the next sections. Thus, the creation of a random subset of vertices will be omitted for reasons of clarity although the range is still described in the text.*

Algorithm 5.1.1: Random community mutation

```
   Input: numVertices: int
 1 Function getRandomVertices():
 2 |   vertices ← ∅
 3 |   while |vertices| ≠ numVertices do
 4 |   |   vertices ← vertices ∪ drawUniformly([0, n))
 5 |   return vertices

 6 Function randomCommunityMutation():
 7 |   numNodesMutate ← drawUniformly([1, 0.1n])
 8 |   randomVertices ← getRandomVertices(numNodesMutate)
 9 |   foreach Vertices v ∈ randomVertices do
10 |   |   mutateRandomly(v)

   Input: v: Vertex to mutate
11 Function mutateRandomly():
12 |   c_v ← drawUniformly([0, 2 · |fittestChromosomeCommunities|] − 1)
```

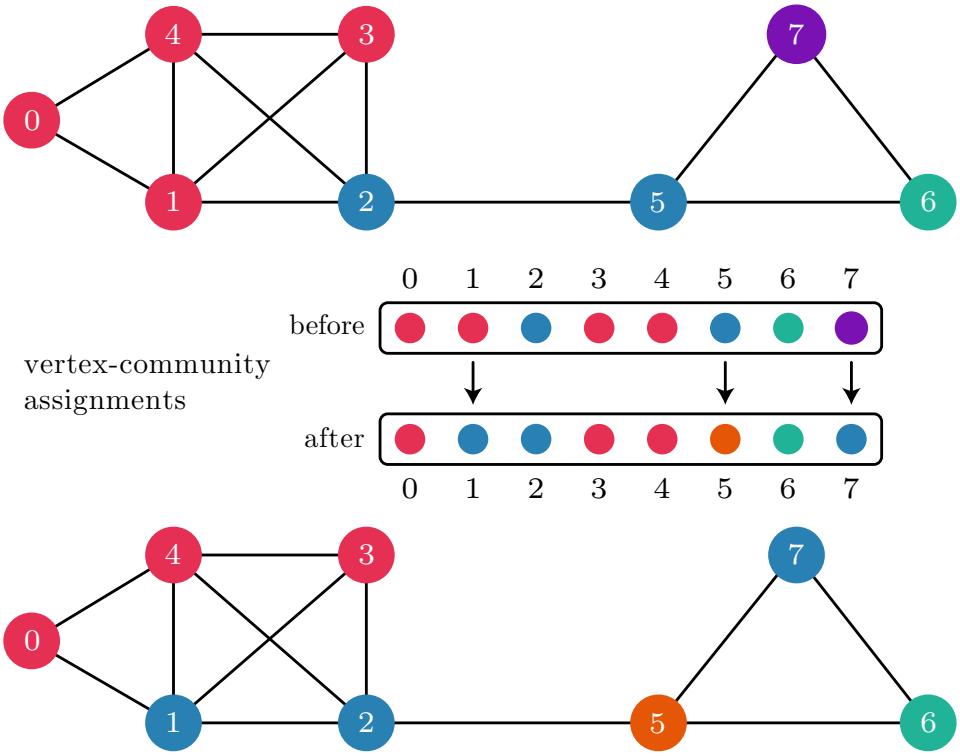


Fig. 5.1.: Random community mutation with $M = \{1, 5, 7\}$

Figure 5.1 illustrates the effect of the random community mutation. In the example, the subset of vertices $M = \{1, 5, 7\}$ is randomly chosen. Each of these vertices is allotted a new random community, e. g. vertex 7 is now in the green community while it was in the purple one beforehand. New communities might arise as can be seen on vertex 5: it is assigned the new orange community that did not exist beforehand.

5.2. Inherit community

This mutation (see Algorithm 5.2.1) picks a random subset of vertices $M \subseteq V(G)$ and assigns to them the prevalent community in their neighborhood, e. g. if a vertex v has two neighbors that are in community c_1 and only one neighbor in community c_2, the vertex "inherits" c_1 ($c_v \leftarrow c_1$). In this sense, let us define the "most prevalent community" in a subset of vertices $S \subset V(G)$ as follows:

$$c_{prevalent} := \arg\max_{c} |\{v \in S : c_v = c\}| \tag{5.1}$$

In our case, considering vertex v, we set $S = N_G(v)$, i. e. S encompasses only the direct neighbors of vertex v. This inheritance of the prevailing community in one's own neighborhood is reasonable as communities are characterized by many links inside and only few between communities. Hence, it is probable that a vertex belongs to the same community as the dominant community of its neighborhood.

Lehnerer uses this mutation operator in [10] and we adopt his algorithm and parameters in Algorithm 5.2.1. First, a random sample of vertices (alleles of the chromosome) is chosen with size in range $[0, n)$. Then, for every vertex v in that sample, we check its neighbors for the most dominant community by calling `getPrevalentCommunity()` in line 7. This function constructs an array with the index representing the community and the element itself the number of occurrences in the given *vertices* set (line 2). Note that instead of an array, we use a map (dictionary) in the C++ code. If there is a tie after having filled this data structure (more than one max element), we select randomly among the most prevalent communities. In the end, v is allotted the dominant community among its neighbors (line 7).

Figure 5.2 depicts an example for $M = \{1, 2, 7\}$. Vertex 1 stays in the red community as the majority of its neighbors (0, 3 and 4) are red too. For vertex 2, we have explicitly marked its neighbors by colored arrows. It is also assigned the red community for the same reason as vertex 1. For vertex 7, the blue and green community both occur exactly once in its neighborhood; we randomly assign the green community to it.

Algorithm 5.2.1: Inherit community mutation (for one vertex)

Input: *vertices*: Set of vertices
Function `getPrevalentCommunity()`:
  $communityCounters \leftarrow [0, 0, 0, \ldots, 0]$
  foreach Vertices $v \in vertices$ **do**
    $communityCounters[c_v] \leftarrow communityCounters[c_v] + 1$
  return index of max element in *communityCounters*

Input: v: Vertex to mutate
Function `inheritCommunityMutation()`:
  $c_v \leftarrow$ `getPrevalentCommunity(`$N_G(v)$`);`

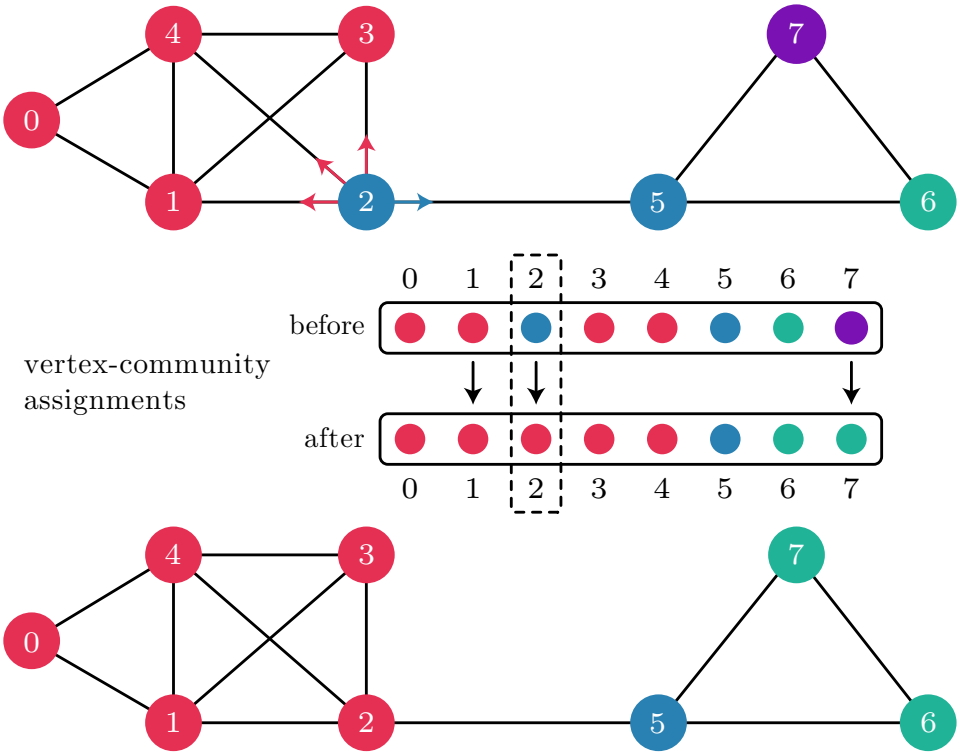


Fig. 5.2.: Inherit community mutation with $M = \{1, 2, 7\}$. The neighboring communities of vertex 2 are indicated by colored arrows.

5.3. Triangle chain

This novel mutation operator tries to construct a "triangle chain" in the graph and assigns to all vertices in that chain the most dominant community. We start with a randomly selected vertex v and check if one of its neighbors has an edge to a vertex that v also has an edge to. More formally: considering vertex v, we look for a neighbor $neighbor \in N_G(v)$, such that $\{neighbor, u\} \in V(G) \wedge \{v, u\} \in V(G) \Rightarrow u \in N_G(v) \cap N_G(neighbor)$ (line 3). In the following, this property is referred to by "Vertex v and its *neighbor* share the vertex u" and we say that these vertices are part of triangle $\triangle v\, neighbor\, u$. The $bestNeighbor$ is selected based on which *neighbor* shares most vertices with v (line 4). Afterwards, we continuously check for the most recently constructed triangle if the last two vertices (in this case: *neighbor* and u, see lines 13 and 14) share another vertex w (line 16) that we haven't visited yet (line 15). Out of the possible candidates, we choose one common neighbor randomly (line 19). The extension process is repeated exhaustively until the chain of triangles can no longer be enlarged (line 18).

As there are more edges within a community than between communities, this chain should include – in the best case – only vertices that belong together. This property is accounted for by assigning the most prevalent community (according to Equation 5.1) to all vertices in the triangle chain as a last step of this mutation (lines 7 to 9) .The whole algorithm is executed for all vertices that we chose randomly (number of nodes to mutate is in range $[0, n)$, compare to set M in the previous mutations).

In Figure 5.3, we have randomly selected vertex v as starting point. The orange *neighbor* shares u with v allowing us to construct the triangle $\triangle v\, neighbor\, u$. Note that v also shares a vertex with two other neighbors, however, with *neighbor* it shares the most vertices (here: two in number, one of them being u). Having established the first triangle, we check if u and *neighbor* share a vertex that is not v. There are two vertices to choose from, in this case w is selected. This new triangle $\triangle neighbor\, u\, w$ is followed by one more triangle $\triangle u\, w\, p$ constructed in the same way. The chain of triangles ends here as no new shared neighbors can be identified. Finally, all vertices in the chain – namely v, *neighbor*, u, w and p – are moved to the prevalent community. The red community appears twice (u and w), while all other communities are found only once. The dominant community is therefore the red one and the resulting vertex-community assignment is shown in the lower graph in the figure.

Algorithm 5.3.1: Triangle chain mutation (for one vertex)

Input: v: Vertex to mutate
Function `triangleChainMutation()`:
  foreach $neighbor \in N_G(v)$ **do**
    $commonNeighbors \leftarrow N_G(v) \cap N_G(neighbor)$ ▷ and store this
  $bestNeighbor \leftarrow neighbor$ where $|commonNeighbors|$ is max
  $commonNeighbor \leftarrow$ Random element of $N_G(v) \cap N_G(bestNeighbor)$
  $\triangle vertices \leftarrow$ `extendTriangle(`$vertex, bestNeighbor, commonNeighbor$`)`
  $prevalentCommunity \leftarrow$ `getPrevalentCommunity(`$\triangle vertices$`)`
  foreach $v \in \triangle vertices$ **do**
    $c_v \leftarrow prevalentCommunity$

Input: $vertex, bestNeighbor, commonNeighbor$
Output: $\triangle vertices$: All vertices in the triangle chain
Function `extendTriangle()`:
  $\triangle vertices \leftarrow [vertex, bestNeighbor, commonNeighbor]$
  while true **do**
    $startVertex \leftarrow \triangle vertices[-2]$ ▷ second to last element
    $otherVertex \leftarrow \triangle vertices[-1]$ ▷ last element
    $candidates \leftarrow$ Filter $N_G(startVertex)$ for vertices $v \notin \triangle vertices$
    $extensionNeighbors \leftarrow candidates \cap N_G(otherVertex)$
    if $extensionNeighbors = \varnothing$ **then**
      break
    $\triangle vertices \leftarrow \triangle vertices \cup \{$ `getRandomElement(`$extensionNeighbors$`)` $\}$
  return $\triangle vertices$

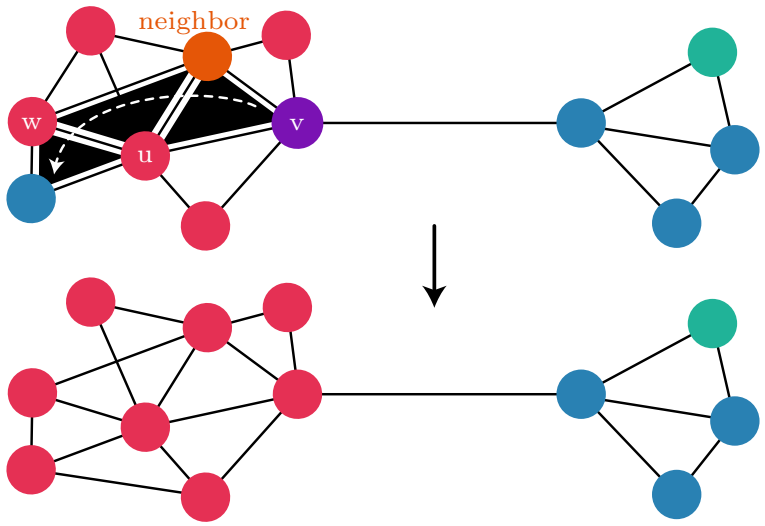


Fig. 5.3.: Triangle chain mutation starting with the random vertex u

5.4. Louvain CRATER

We introduce Louvain CRATER, short for "Louvain on Crater Rim, Adoption Towards Eruption Repair". This new mutation operator applies the "Ruin & Recreate" strategy – devised by Schrimpf et al. in [44] and successfully used to solve the TSP – to the problem of community detection. It requires to destroy parts of the solution and subsequently recreate them, so that in the best case, we obtain a better solution or can move away from a local maximum. While for the TSP, it might be beneficial to destroy large parts, so that there is more freedom in finding a new solution that is viable, we tend to destroy only smaller ones as there are no invalid community assignments and, hence, every solution produced is valid in the sense that every vertex has exactly one community assigned. This is ensured by our chromosome encoding explained in section 4.1. The underlying algorithm is presented in 5.4.1.

We will refer to the process of "ruining" parts of a solution as throwing an imaginary bomb onto a vertex, accompanied by a subsequent shockwave which spreads out and destroys the landscape of vertices in the neighborhood of the eruption center (line 4). Starting with the ground zero vertex, the shockwave spreads to its neighbors (line 11) as first layer, then their neighbors as next layer and so on. The number of layers ($numLayers$) is an adjustable parameter of this mutation. We store the vertex-layer assignment for later contraction of the shockwave in an array of arrays of vertices ($layerVertices$, see lines 2 and 12) and also keep track of which vertices have which vertices in an outer layer as neighbors ($adjVerticesInOuterLayer$: Map<Vertex, Vertices>).

Then, in the "recreate" phase, we assign new communities to these vertices starting with those on the rim of the crater (line 5). They get allocated neighboring community, so that the local modularity gain is maximized (line 18). This corresponds exactly to the first phase of Louvain presented in section 3.1, the difference being that we do not employ this greedy modularity optimization to every vertex in the graph, but only to the crater rim saving plenty of computation time compared to the Louvain method.

Afterwards, the shockwave "shrinks" (line 6) and we propagate the assigned communities on the crater rim towards the eruption center (ground zero). Starting with the second to last layer (line 20), for each vertex v (line 21), we adopt the community that is most prevalent (according to Equation 5.1) in the neighborhood of the vertex, while only considering those vertices that are in the next outer layer (line 23). If v has no vertices

in an outer layer, i. e. only edges to an inner layer, we choose a random community of its neighbors before they were unassigned (line 26). The "adoption towards eruption repair" is repeated until we've reached layer 0 – the eruption center – which gets assigned a new community in the same manner. After that, we can start with the next bomb on another random vertex in the graph. The number of bombs is an adjustable parameter of the Louvain CRATER mutation.

Figure 5.4 shows the crater of one bomb in a sample graph. Here, we set the number of layers to 3. Vertices 5, 6, 7 and 8 form the crater rim. In the upper part of the image, the community assignment has been deleted for all vertices in the crater and the Louvain modularity optimization was applied for the vertices on the crater rim, which is why they are colored now. The inner vertices have no community assigned yet at this point (they are marked in black). The lower part of the figure depicts the final vertex-community assignment after the "recreation" phase of Louvain on Crater Rim, Adoption Towards Eruption Repair (Louvain CRATER). Vertex 1 has links to two vertices in the next outer layer: 5 and 8. As there is no prevalent community, we choose one of them randomly (the blue one). Vertex 2 has relevant neighbors 5, 6 and 7 with the red community being dominant. Vertex 3 has no neighbors in the closest outer layer; in that case, we assign the community of any neighbor (chosen randomly) to this vertex assuming no bomb dropped at all (the "ruin" part is only temporary and we still have the original community-vertex assignment available at that point). For vertex 4 the same reasoning applies as to vertex 1. Afterwards, the next inner layer is considered which solely encompasses vertex 0. There is no clear prevailing community in its relevant neighborhood, so we choose out of the available ones randomly (the blue one). We've reached ground zero (vertex 0) signaling the end of Louvain CRATER.

Algorithm 5.4.1: Louvain CRATER mutation (for one eruption vertex)

Input: v: Eruption vertex (ground zero)
1 **Function** `CRATERMutation()`:
2 $layerVertices \leftarrow [[eruptionVertex]]$ ▷ array of array of vertices
3 $c_v \leftarrow$ UNASSIGNED
4 $adjVerticesInOuterLayer \leftarrow$ `expandShockwave(`$layerVertices$`)`
5 `doCraterRimLouvain(`$layerVertices[-1]$`)` ▷ -1 means last element
6 `shrinkShockwave(`$layerVertices$, $adjVerticesInOuterLayer$`)`

Input: $layerVertices$
Output: $adjVerticesInOuterLayer$
7 **Function** `expandShockwave()`:
8 $adjVerticesInOuterLayer$: Map<Vertex, Vertices>
9 **for** $layer \leftarrow 0$; $layer < numLayers - 1$; $layer \leftarrow layer + 1$ **do**
10 **foreach** $v \in layerVertices[layer]$ **do**
11 **foreach** $neighbor \in N_G(v)$ that is not already visited **do**
12 $layerVertices[layer + 1]$.append($neighbor$)
13 $adjVerticesInOuterLayer[v] \leftarrow adjVerticesInOuterLayer[v] \cup \{neighbor\}$
14 $c_{neighbor} \leftarrow$ UNASSIGNED
15 **return** $adjVerticesInOuterLayer$

Input: $rimLayerVertices$
16 **Function** `doCraterRimLouvain()`:
17 **foreach** $v \in rimLayerVertices$ **do**
18 Check for each $neighbor \in N_G(v)$ the ΔQ we would get by setting $c_v \leftarrow c_{neighbor}$. Assign community where ΔQ is maximized as done in Louvain, see Algorithm 3.1.1, line 17.

Input: $layerVertices$, $adjVerticesInOuterLayer$
19 **Function** `shrinkShockwave()`:
20 **for** $layer \leftarrow numLayers - 1$; $layer \geq 0$; $layer \leftarrow layer - 1$ **do**
21 **foreach** $v \in layerVertices[layer]$ **do**
22 **if** $v \in adjVerticesInOuterLayer$ **then**
23 $c_v \leftarrow$ `getPrevalentCommunity(`$adjVerticesInOuterLayer[v]$`)`
24 **else**
25 $randomNeighbor \leftarrow$ `getRandomElement(`$N_G(v)$`)`
26 $c_v \leftarrow c_{randomNeighbor}$ ▷ community prior to unassignment in line 14

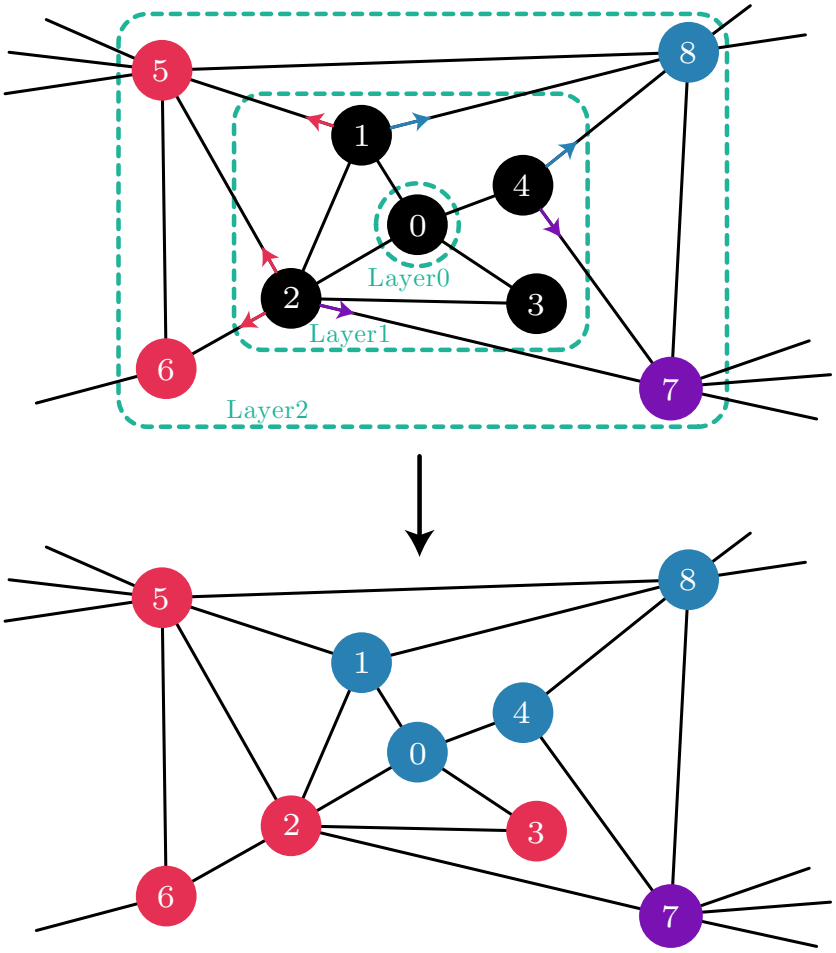


Fig. 5.4.: Louvain CRATER mutation

6. Datasets

To evaluate our heuristics presented in the previous chapters, we employ synthesized and several real-world datasets (mainly from Stanford Network Analysis Platform (SNAP) [45]) ranging from social media networks to those describing author collaboration in publications. All networks are available as .CVS files on GitHub[1]. The graphs have been plotted with the free and open-source network visualization tool Gephi [46] applying either the force-directed layout algorithm ForceAtlas2 [47] or the one devised by Yifan Hu in [48]. Both algorithms are well-integrated into Gephi offering continuous feedback over the current state of the layout process. The drawing is based on a physical model where nodes exert a repulsive force on each other (comparable to the electrostatic force between two like charges), while edges attract each other (comparable to physical springs pulling vertices together).

6.1. synthesized-z_{out}

For a first evaluation of our algorithms, we generate simple random graphs with known community structure as discussed by Girvan and Newman [11] and thereafter used in [16], [18] and [24]. The graph consists of $n = 128$ **vertices and is split into four communities with 32 vertices** each[2]. We insert an edge with probability p_{in} between vertices of the same community, and with probability p_{out} between vertices of different communities. p_{in} and p_{out} are sampled from a normal distribution and directly control the average number of edges a node has to the same community z_{in}, or to any other community z_{out}. We choose these values, such that $z_{in} + z_{out} = 16$ for every generated graph. At the same time, p_{out} and therefore z_{out} is varied, while p_{in} is adjusted to satisfy the constraint. This allows us to generate various random networks with a very clear community structure (small z_{out}) to diffuse and hard to identify community structures when the number of within- and between-community edges per vertex is the same. The

[1] https://github.com/splines/bachelor-community-detection-datasets

[2] While this is parameterizable, we will keep the settings used in literature to be able to compare results.

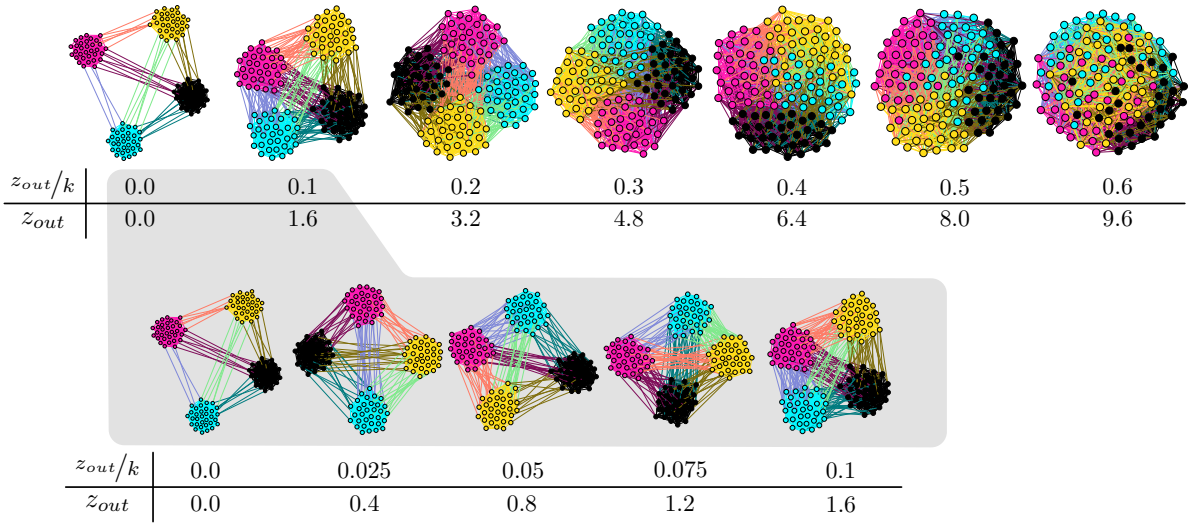


Fig. 6.1.: Synthesized networks with known-ground truth (indicated by colors). z_{out}/k (with $k = 16$) increases from left to right. The top row depicts the range $[0.0, 0.6]$, while the bottom row "zooms in" and shows the networks for the range $[0.0, 0.1]$.

latter is the case for $z_{out} = 8$ in our network. More generally, we consider the condition $z_{in} + z_{out} = k$, with k as the total average vertex degree, and generate random networks with different z_{out}/k. Here, for $z_{out}/k = 8/16 = 0.5$, the network is not distinguishable from a random one.

Figure 6.1 depicts randomly generated networks as described above. The ground-truth communities are marked in different colors. We vary z_{out} and increase it from left to right, making it more and more difficult to correctly identify the communities (if one did not know the ground-truth).

6.2. synthesized-lehnerer

We will also evaluate the algorithms on Lehnerer's third randomly generated network (Figure 9 in [10]) which consists of **1293 vertices and 4145 edges**. This network already has about $6.7 \cdot 10^{2609}$ possible community partitions. Although this network is computer-generated, no ground-truth data is available. The communities found by Gephi's built-in modularity analysis are shown in Figure 6.2.

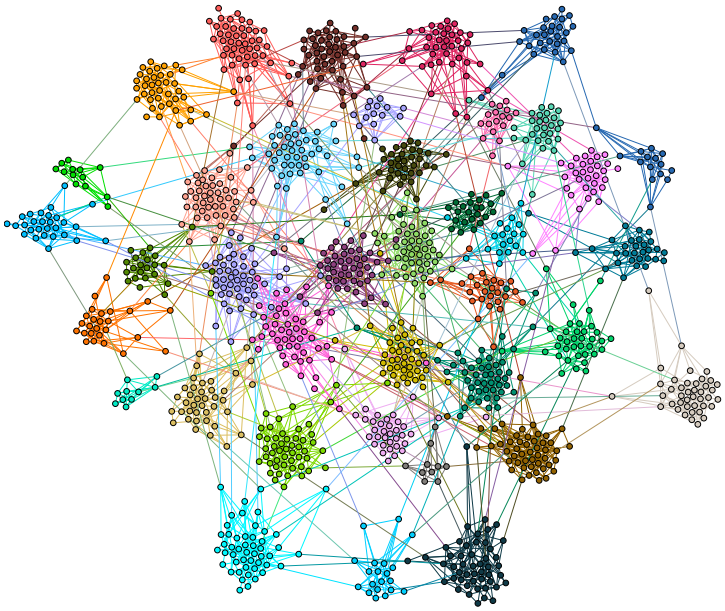


Fig. 6.2.: Lehnerer synthesized network. 36 communities colored according to Gephi's modularity analysis with $Q \approx 0.906$.

6.3. ego-Facebook

On the social media platform Facebook, users can connect to their friends and add them to their "friends list". This list is private, yet Leskovec and McAuley asked ten survey participants to "manually identify all the circles to which their friends belonged", e. g. friends working in the same company, friends who go to the same high school or friends who form a family [49]. The anonymized data is publicly available on SNAP [45] and includes the ground-truth circles (communities) of the ten combined ego-networks. The ego-Facebook network consists of **4039 vertices (users) and 88,234 edges (users' friendship)** with 46 communities and 1155 vertices that do not have a community assigned (that is 28.60 % of all vertices). The latter is the case if a user did not include a friend in any circle.

50 % of the communities overlap with another community and 25 % are even nested inside other communities. Modularity as objective function and our genetic approaches do not take this into account; in our model, a vertex (user) can only belong to one community. As an aid to still make use of the ground-truth data, we employ a simple strategy: a vertex v that belongs to communities $c_v \in C_v$ is assigned the first one and then "locked",

i. e. even if we find that v belongs to another circle iterating through the ground-truth circles, its community is not changed anymore. As a result of this approach, we identify 39 communities and observe a modularity of 0.4781. If one excludes vertices that don't have a community assigned, we achieve a modularity of 0.5591 with 38 ground-truth communities. These communities are colored in Figure 6.3, while gray nodes indicate they have no community assigned.

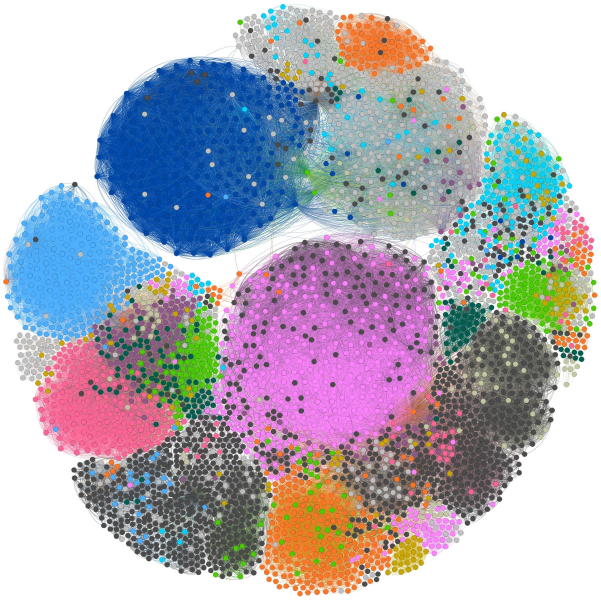


Fig. 6.3.: ego-Facebook network. Communities colored according to filtered ground truth (without overlap), which yields a modularity of $Q \approx 0.5591$ (38 communities). Gray nodes have no community. Gephi's built-in modularity analysis found 15 communities with $Q \approx 0.835$.

6.4. com-YouTube

Users of the online video platform YouTube can create groups and invite other people to join them. In this dataset provided by Mislove et al. in [50] and available on SNAP [45], groups are considered the ground-truth communities. The network consists of 1,134,890 vertices and 2,987,624 undirected edges. However, 8001 communities encompass less than three vertices. We filter them out and get 8385 communities. The filtered graph has 52,675 vertices (4.64 % of original vertices) with 17,722 vertices in multiple communities corresponding to 2135 overlapping communities. We put to use the same strategy as

described for the ego-Facebook dataset to eliminate overlap and filter out 5540 vertices. Our final network has **47,135 vertices (users), 300,717 edges (user groups)** and 6249 ground-truth communities with an average community size of 7.54.

Unfortunately, with our simple strategy to remove overlapping communities, the ground-truth communities only yield a modularity of 0.1257. That is why we do not consider the ground-truth communities and assume them to not exist anymore. We can still try to optimize modularity using Louvain and our genetic approaches. Figure 6.4 shows the filtered dataset with 428 communities found by Gephi yielding a modularity of 0.629. Unlike in Figure 6.3, gray nodes are viable communities here as well but smaller than the colored communities.

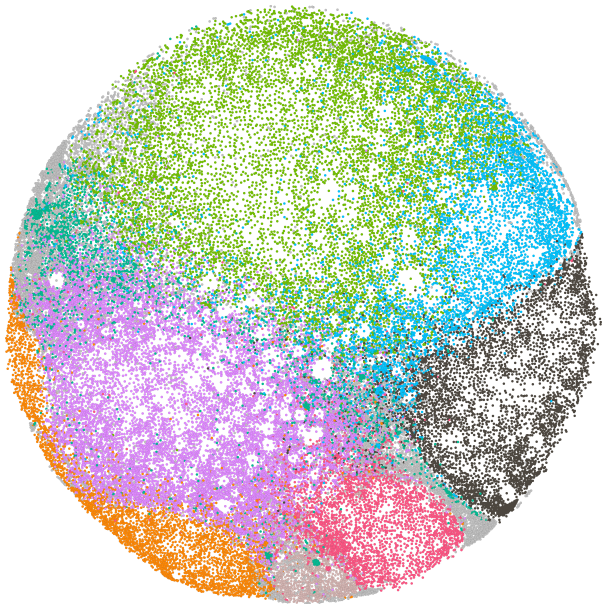


Fig. 6.4.: com-YouTube network, filtered for communities with $k_v \geq 3$ and omitted overlap. Communities colored according to Gephi with $Q \approx 0.629$ (428 communities found but small communities get assigned a gray color).

6.5. hep-ph-citations

The hep-ph-citations dataset was used in the data mining competition *KDD Cup* in 2003 [51]. It describes the citation tree (at that time) of High Energy Particle Physics (hep-ph) literature published on the e-print archive *arXiv*, labeled "Arxiv" in the Louvain

paper [8]. The directed edges specify which paper cited which other paper. The network consists of **34,546 vertices (paper ids) and 420,921 undirected edges (paper cites other paper)**. Originally, there are 421,578 directed edges, but we consider them to be undirected and filter out mutual edges.

77 communities were found by Gephi resulting in a modularity of $Q \approx 0.722$. Some of them are depicted in Figure 6.5. Filtering for vertices with a degree $k_v \geq 50$ is only done in favor of a more meaningful visualization, but we will use the whole graph for our experiments later on.

Fig. 6.5.: hep-ph-citations network, filtered for $k_v \geq 50$. Communities colored according to Gephi's modularity analysis with $Q \approx 0.722$ (77 communities found while not all are depicted here due to $k_v \geq 50$).

6.6. ca-AstroPh

The ca-AstroPh network from SNAP [45] represents co-authorship on astro physics papers from January 1993 to April 2003 on the e-print archive *arXiv*. Vertices denote authors, while an undirected edge $\{u, v\}$ exists if author u collaborated with author v on a scientific paper. A paper that p authors worked on corresponds to a complete graph of size p as subgraph in the network. We filtered out 60 self-loops on one vertex; the final graph

consists of **18,771 vertices (authors) and 198,050 edges (authors collaborated together)**. A subset of the graph is shown in Figure 6.6.

Fig. 6.6.: ca-AstroPh Astro Physics collaboration network. Gephi found 317 communities with $Q \approx 0.621$, we filtered for $k_v \geq 50$ for better visualization.

6.7. email-EU-core

The email-Eu-core network available on SNAP [45] originates from the email traffic of a European research institution, whose members are represented as vertices in the graph. When person u sent person v at least one email, there is a directed edge (u, v) in the graph. Originally, the graph consists of 1005 vertices and 25,571 edges. As we consider edges to be undirected, we filter out mutual edges reducing them to 16,706. Of these edges, we encountered 642 self-loops on 19 vertices. Our final network is composed of **986 vertices (people in the institution) and 16,064 edges (mail communication between institution members)**.

This dataset is labeled according to one of the 42 departments a member of the research institution is part of. We therefore have 42 communities as depicted in Figure 6.7. The ground-truth vertex-community assignment yields a modularity of 0.2880. Gephi found 7 communities and achieved a modularity of 0.415. In this case, the effects of the resolution limit (Figure 2.1) are clearly visible: all ground-truth communities except for one of size

107 are smaller than the resolution limit which calculates to $\sqrt{16064/2} \approx 89.6 \approx 90$, while the average community size is $23.5 \approx 24 < 90$. Thus, we except communities to be merged together when optimizing for modularity.

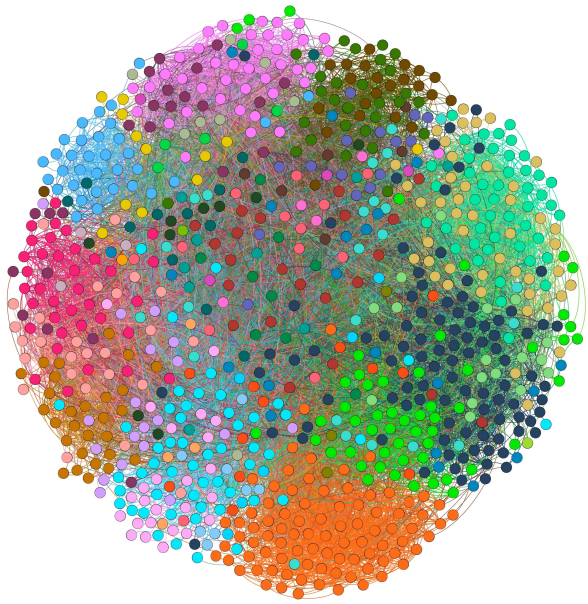


Fig. 6.7.: email-EU-core dataset from a European research institution, filtered for $k_v \geq 2$ and self-loops were omitted. 42 communities colored according to ground-truth yielding a modularity of $Q \approx 0.2880$, while Gephi found a value of $Q \approx 0.415$ (7 communities).

6.8. speyer-web

Speyer is one of the oldest cities in Germany located in the federal state Rhineland-Palatinate in the southwest of Germany. Inspired by [3], where Albert, Jeong, and Barabási scraped websites to show the power-law distribution of link distributions on the world-wide-web, we implemented a similar program starting from the website "https://speyer.de". We collect all URLs found in the HTML document of this website[3] and recursively visit them to collect new URLs until we've reached a certain depth (level). Keeping track of which website links to which other websites, we construct a graph that portrays the connectivity of the world-wide web in a small scope. We implemented

[3]To avoid parsing the whole HTML document, we search for *a*-tags using this regex: `(?<=<a href=")[^"]*` and only accept *http* and *https* as protocol omitting *tel* and the like.

the web-robot in Python leveraging all 40 CPU cores[4] through the *Pool* object of the *multiprocessing* package.

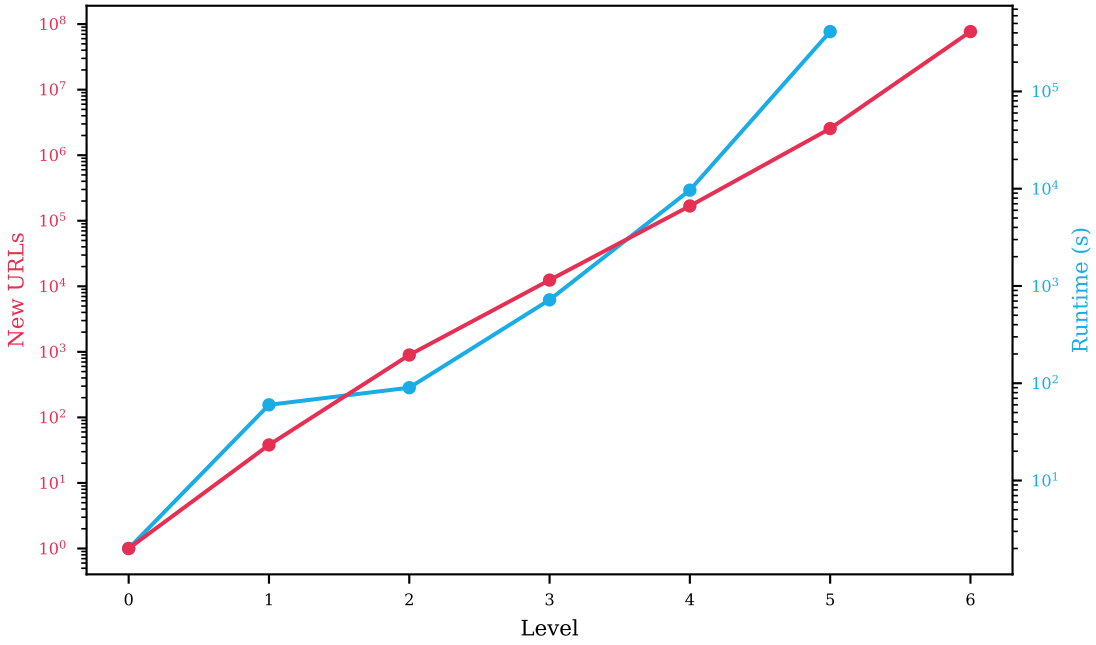


Fig. 6.8.: Number of new URLs per level and runtimes for the speyer-web network. Axes are log-scaled and both measures are *not* cumulative.

Figure 6.8 shows the number of URLs visited at each level, with level 0 containing only the starting URL. While third level, with 13,360 vertices, took 12 minutes, level 4 took almost three hours (for 181,155 URLs to visit at that level) and level 5 nearly five days (for 2,724,473 URLs). The sixth level would have implied accessing 79,618,374 URLs; we stopped the robot after successful termination of the fifth level.

Subsequently, we "flattened" the resulting graph in a way that we do not consider the full URL anymore, but instead merge together URLs (vertices) with the same network location (netloc, see [52]), i. e. the same domain and subdomain if present, however we ignored the *www* subdomain. This way, the URLs *https://speyer.de/de/impressum/* and *https://www.speyer.de/de/datenschutz/* are combined into one vertex labeled *speyer.de*. We do this since we are (1), not interested in the links referencing another page of the same domain, and (2), the graph would otherwise be too big, e. g. the edges data at level 5 contains almost 122 million edges and is 29 GB in size. Although edges are directed in

[4]Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz, see Chapter 7 for more details on the machine.

our original graph, we assume them to be undirected and remove every mutual edge as we only deal with unweighted and undirected graphs throughout this thesis.

The flattened graph is comprised of 8463 nodes and 43,133 edges including 1167 self-loops. The final speyer-web graph we will use as test data in the next chapter therefore has **8463 vertices and 41,966 edges**. It is depicted in Figure 6.9 including the vertex labels. The PDF is embedded in the electronic version of this thesis, so that the graph can be enlarged in order to better read the labels.

The big dark-blue colored community in the bottom-right of the network strongly resembles a fully-connected (sub)graph, with vertices representing subdomains for different languages of the free encyclopedia *Wikipedia*. Articles on Wikipedia are characterized by a plethora of links to other articles and usually also contain links to the same article written in different languages – properties that lead to a fully-connected graph. Even though the number of articles on Wikipedia differs among languages [53], the node sizes in this graph are the same for this community, as we stopped the web-robot after five levels starting with *speyer.de* – which is in the light-blue community in the middle above the Wikipedia community. We only scratched the surface of Wikipedia and cannot thus deduce any significant information about the language distribution of articles in the encyclopedia.

The biggest node in the graph is *dongleauth.com* situated on the left. This website compares websites with each other and therefore contains plenty of links. Even though other websites might not link to this domain, this vertex still has a high degree due to the undirectedness of edges. A high degree can indicate that (1), the website contains many links to other website, *or* (2), the website is being referenced by many other websites. The latter is the case for social media platforms such as YouTube, Facebook, Instagram, LinkedIn and Twitter (in the light-blue community). Other big nodes include developer-related websites such as *github.com*, *apache.org*, *android.com* and *cloud.google.com* (orange community in the lower left), a help page for victims of crime in Germany called *weisser-ring.de* (purple community in the upper right) and websites such as *boys-day.de*, *girls-day.de* and *staedtetag.de* (red community in the top center).

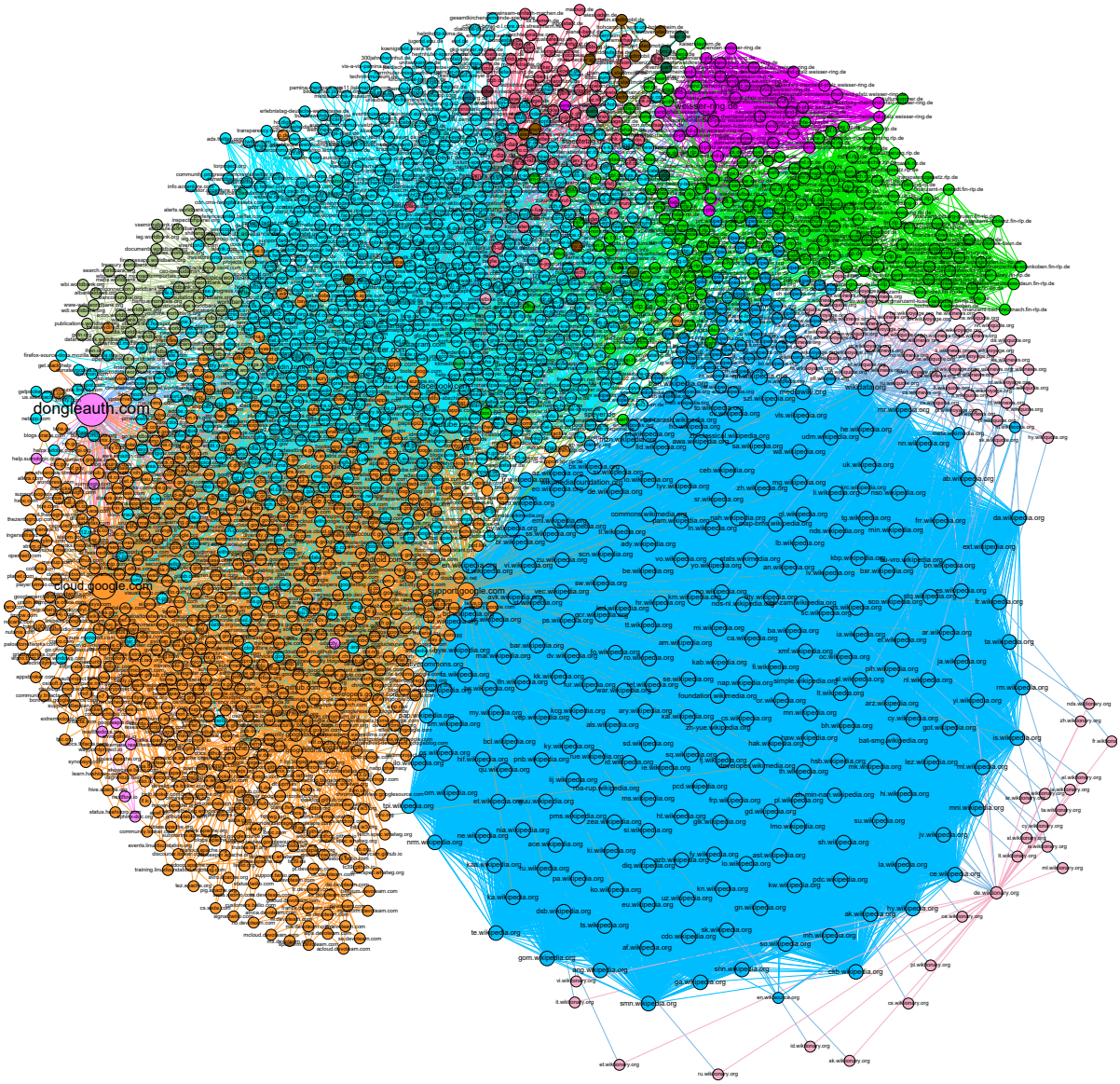


Fig. 6.9.: speyer-web network, filtered for $k_v \geq 3$. 16 communities were found with Gephi (in the graph prior to visual filtering) resulting in $Q \approx 0.485$. Nodes with a higher degree are bigger in the graph.

6.9. Overview

The following table serves as an overview of the presented datasets, including their number of vertices and edges and the modularity found by Gephi or, if present, the ground truth modularity with the number of communities.

Name	**#vertices**	**#edges**	Gephi Q (#communities)	Ground-truth Q (# communities)
synthesized-z_{out}	128	random	-	individual
synthesized-lehnerer	1293	4145	0.906 (36)	-
ego-Facebook	4039	88,234	0.835 (15)	0.5591 (38)
com-YouTube	47,135	300,717	0.629 (428)	-
hep-ph-citations	34,546	420,921	0.722 (77)	-
ca-AstroPh	18,771	198,050	0.617 (316)	-
email-EU-core	986	16,064	0.415 (7)	0.2880 (42)
speyer-web	8463	41,966	0.485 (16)	-

Table 6.1.: Overview of the datasets

7. Experimental results

In this chapter, we will discuss results obtained for different genetic approaches introduced in the previous chapter which we evaluated on the datasets described in Chapter 6. All tests were performed on the following Linux cloud system:

- 64-bit Linux with SUSE Linux Enterprise Server 15 SP3 installed
- 40 Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz processors
- 160 GB of RAM

However, we did not yet implement concurrency and thus the algorithms were executed only on one processor.

7.1. Ambiguity of community labels

One problem that arises when comparing ground-truth vertex-community assignments with those emitted from a genetic algorithm is that of ambiguous community labels. As an example, a mutation operator might not reference community 0 by this exact "name" (integer), but instead label it "3". If all vertices of the original community 0 are labeled 3 in the output of our algorithm, and if no other vertex has this community label assigned, we identified the whole community correctly despite having used a different label for it. To account for this, we employ the strategy presented in Algorithm 7.1.1 to count the fraction of correctly classified vertices.

First, we iterate through all communities $c \in \mathcal{C}$ (line 4) and collect the set of vertices in that community (line 5). For all vertices $v \in V_c$, we retrieve the label that the algorithm assigned to this vertex, namely $vertexToCommunity[v]$. We then find the element with maximum occurrence in the list (or select the first if there is a tie) (line 6). The mapping from c to *prevalent* is stored (line 8), but only if *prevalent* is not already present in the values of the map (line 7). The latter could be the case when we observe that in the next ground-truth community the prevalent label is again 3. The if-condition makes sure

we do not overwrite any mapping, although this might bias the outcome, e. g. when we lay down a specific mapping due to one small community but then later on encounter a very big community whose vertices are correctly grouped together in the output of an algorithm, yet labeled differently. In this contrived example, many vertices would be incorrectly classified compared to the correctly identified vertices in the small community. In the absence of a more suitable approach, we will accept this limitation regardless.

Having built the map, we iterate over all vertices $v \in V(G)$ (line 9) and check if there is an entry for c_v, in which case we temporarily overwrite c_v with the value returned from the map (line 11). Otherwise, c_v is used from the ground-truth. Finally, c_v is compared against the current vertex-community assignment for v (line 12) and in the case of equality the counter for correctly identified vertices is increased (line 13). It is returned as the result of the function.

Algorithm 7.1.1: Calculate number of correctly classified vertices (comparing ground-truth with current vertex-community assignment as output of an algorithm)

Input: c_v: ground-truth vertex-community assignment,
vertexToCommunity: assignment as output of an algorithm
Output: *numCorrect*: int

```
1  Function getNumberOfCorrectlyClassifiedVertices():
2      numCorrect ← 0
3      map: Map<int,int>
4      foreach Community c ∈ 𝒞 do
5          V_c ← { Vertices v ∈ V(G) : c_v = c }
6          prevalent ← Prevalent community in V_c according to vertexToCommunity
7          if prevalent ≠ map.values() then
8              map[c] ← prevalent
9      foreach v ∈ V(G) do
10         if c_v ∈ map then
11             c_v ← map[c_v]                    ▷ Temporarily overwrite
12         if c_v = vertexToCommunity[v] then
13             numCorrect ← numCorrect + 1
14     return numCorrect
```

7.2. GAs & Hyperparameters

The GAs are dependent on many parameters which we will refer to as hyperparameters resorting to the terminology of machine learning (where these parameters are used to control the learning process itself in contrast to model parameters that are adjusted in the training process). Prior to the evaluation presented in the following sections, we have conducted smaller experiments to find reasonable settings for the hyperparameters. We will enumerate them in the following.

A population size $\lambda = 20$ is chosen. We employ elitism as selection strategy and always select the best of the children as new parent, therefore $\mu = 1$. This results in a $(1, 20)$ strategy. **We always execute the random assignment operator followed by one of the other three mutations (dependent on the experiment) with a probability of** $50\,\%$**. In the end, the local optimization (see section 4.3) is always applied.** After that, the best individual is chosen and used for the next generation. Choices concerning the number of selected random vertices were already covered in Chapter 5. Apart from that, we choose three layers for the bombs in Louvain CRATER and throw as many bombs as determined by a random number between $1\,\%$ and $5\,\%$ of the vertices $(0.05n)$, but at most 10 bombs for any graph.

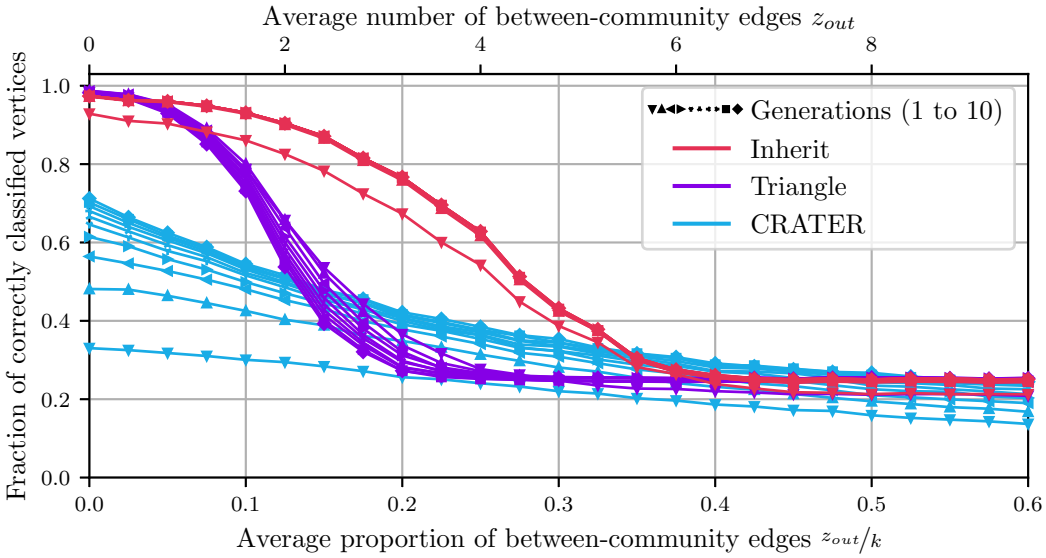
7.3. Evaluation on z_{out}-graphs

For $z_{out} \in [0.0, 0.6]$ and using a size of 0.025, we generated 1,000 graphs for each z_{out} value. This took less than two seconds for every 1,000 graphs (including writing .CSV files to disk). In total, 25,000 randomized graphs were created.

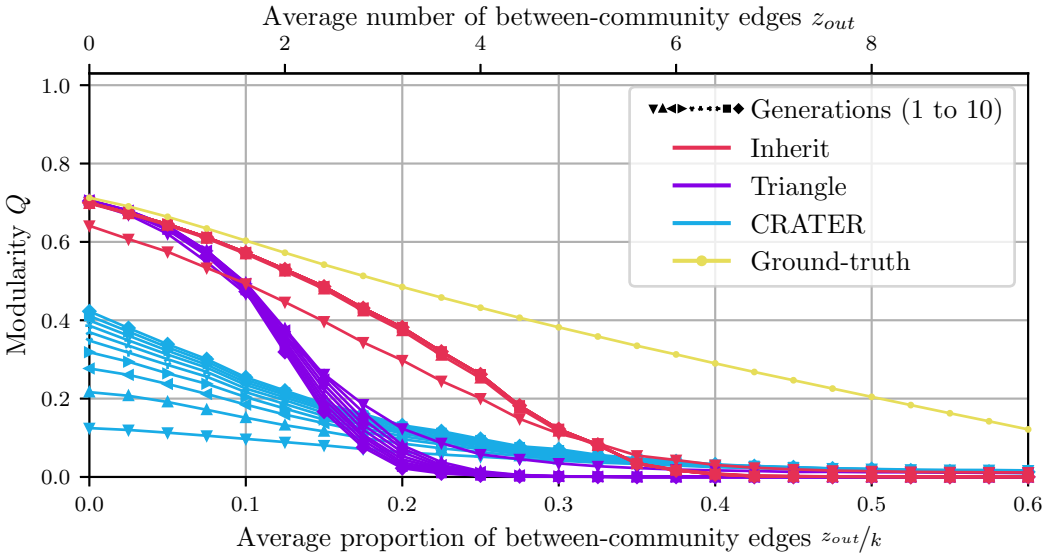
Figure 7.1a depicts the fraction of correctly classified vertices per proportion of between-community edges z_{out}/k (or alternatively per z_{out}, see the upper x-axis). We plot the results for mutation operators presented in Chapter 5 but without Louvain as initialization heuristic. Instead, a random vertex-community assignment is used as starting point. The characteristic curves illustrate the results after each generation (from 1 to 10)[1]. They are marked with different signs as shown in the legend, where the leftmost mark (triangle

[1]Note that we only ran each algorithm once for every 1,000 graphs corresponding to one z_{out} value and stopped after 10 generations. The vertex-community assignment was recorded after every generation and these intermediary results are shown as different characteristic curves in the graph

pointing downwards) represents the first generation and the rightmost mark (diamond shape) symbolizes the tenth generation. Figure 7.1b is similarly constructed but instead plots modularity Q on the y-axis and additionally shows the ground-truth modularity values as yellow line that decreases almost linearly with rising z_{out}/k.



(a) Fraction of correctly classified vertices



(b) Modularity Q

Fig. 7.1.: Evaluation of random-initialized mutation operators on various z_{out} graphs and different number of generations. Each point is an average over 1000 graphs.

Out of the three mutation operators Inherit, Triangle and CRATER, Inherit performs best in most parts, especially for $z_{out}/k \in [0.05, 0.3]$, although it only classifies more than 80 % of vertices correctly up to a value of 0.175. After that, results quickly fall under the 50 % mark as of 0.275. This is due to communities getting merged together, e. g. the algorithm detects three communities while there are four in reality as shown for Lehnerer in Figure 7.3a. This, however, is very unlikely to be related to the resolution limit (page 11) as is apparent from Figure 7.2. We plot the number of within-community edges as average over 1,000 graphs for each z_{out} value (red line) and compare with the resolution limit, calculated by $m_c \leq \sqrt{m/2}$. The critical area is below and on the black (dashed) curve, while the actual numbers are much higher. The resolution limit therefore does not play any role for the z_{out}-graphs.

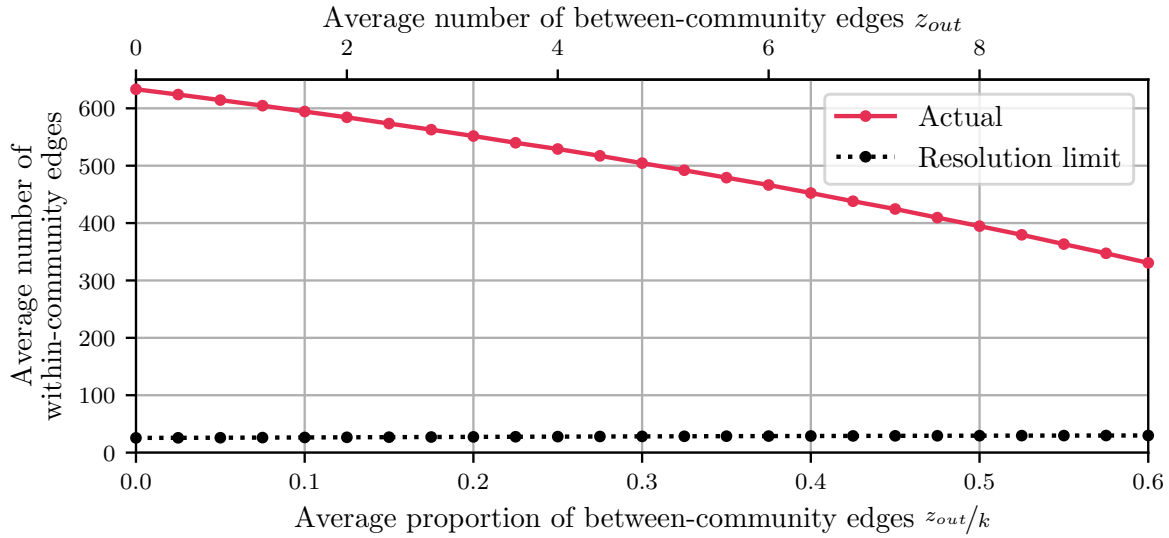
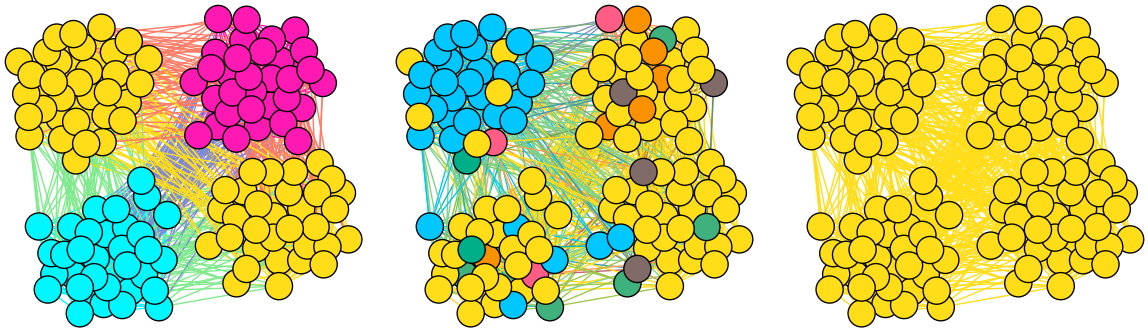


Fig. 7.2.: Within-community edges per z_{out}/k and the resolution limit

This "merging" problem is also present in the Triangle mutation. For $z_{out}/k = 0.225$, figures 7.3b and 7.3c show the resulting vertex-community assignment after the first and second generation. After two generations, the modularity has dropped to $Q = 0.0$ as all vertices have been assigned to the same community. In this case, the first term of Equation 2.6 evaluates to $\frac{|E(c)|}{m} = \frac{m}{m} = 1$ as every edge is contained within this one community. For the second term, we calculate $\frac{\sum_{v \in c} k_v}{2m} = \frac{2m}{2m} = 1$ (summing over k_v counts every undirected edge twice). The second term is subtracted from the first one resulting in a modularity of $Q = 0.0$. In comparison, even the bad assignment in Figure 7.3b yields a higher value of

$Q \approx 0.1223$. This observation explains why the first generation of the triangle operator is above the other lines in Figure 7.1a and 7.1b. Due to more edges between communities with rising z_{out}, the triangle chains start to trespass on other communities and are no longer bounded to one single community. For these synthesized graphs, the triangle operator only performs well for clearly identifiable communities (low z_{out} values).



(a) Inherit after five generations for $z_{out}/k = 0.275$. $Q \approx 0.3495$. Ground-truth: $Q \approx 0.4063$.

(b) Triangle after one generation for $z_{out}/k = 0.225$. $Q \approx 0.1223$. Ground-truth: $Q \approx 0.4580$.

(c) Triangle after two generations for $z_{out}/k = 0.225$. $Q = 0.0$. Ground-truth: $Q \approx 0.4580$.

Fig. 7.3.: Examples for failed vertex-community assignments on z_{out}-graphs

To our surprise, we also observe that the Triangle results get worse as generations go by. This is unexpected as we employ a $(\mu + \lambda)$ strategy and not the (μ, λ) version where the children might replace the parents (see section 4.4). Theoretically, results from one generation are not allowed to deteriorate anymore in subsequent generations. We have found that the delta modularity calculation returns small positive values (e. g. zeros up to five decimal places) in many instances, while the actual modularity change was slightly negative. We could not find errors in the generalized delta modularity formula presented in section 3.3 and assume the problem to be related to numerical errors. However, we were not able to pinpoint the exact cause of this problem after several attempts made. An easy fix would imply to always recalculate the global modularity instead of comparing children by their ΔQ scores. However, this would result in severe runtime performance impacts which is why we did not change the behavior and leave this issue unaddressed in this thesis.

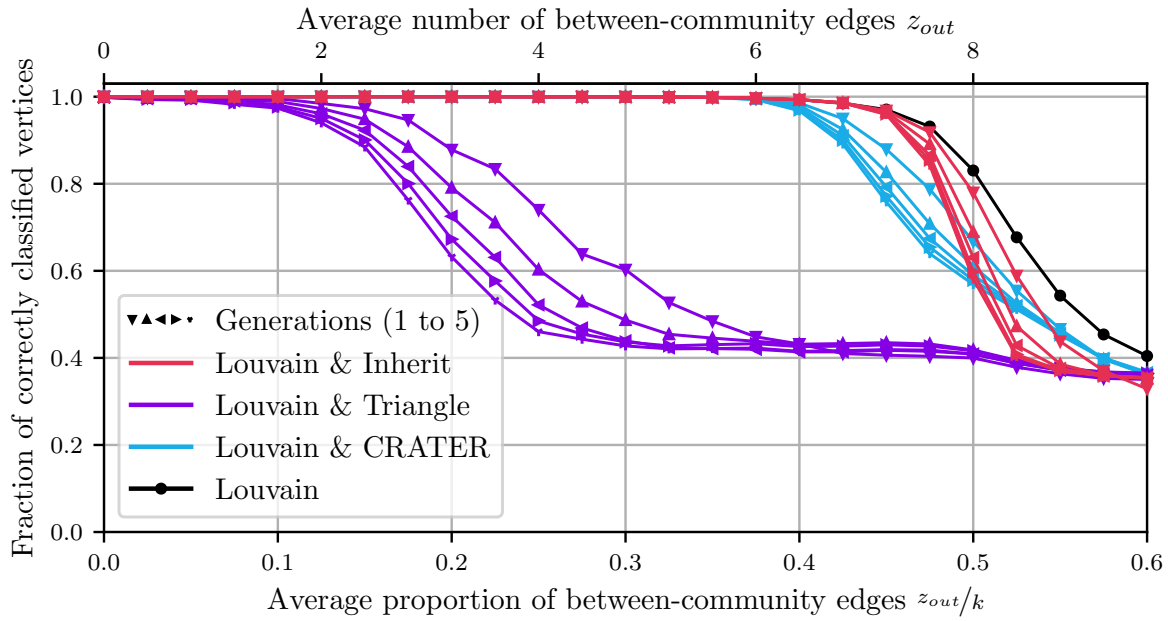
Like the Triangle operator, Louvain CRATER gives worse results than Inherit. For low z_{out} values, Louvain CRATER takes more generations to converge to an upper limit, e. g. for $z_{out} = 0$ the first generation classifies less than 40 % of vertices correctly, but 70 % in the tenth generation. For $z_{out}/k = 0.3$ this difference is smaller, that is, it starts with more than 20 % correctly classified vertices in the first generation and ends with about 35 % in the tenth. The overall low values for Louvain CRATER can be explained by its destructive nature that makes it difficult to find good communities on a large scale. While we apply parts of the Louvain algorithm to the vertices on the crater rim, vertices within the crater only pass on this information towards the inner layers of the bomb. Hence, for the inner vertices, the assigned communities might not reflect the conditions in their direct neighborhood. We found three layers of the bomb to be the "sweet spot" since we obtain worse results for values below or above this. Beyond that, vertices get frequently assigned new communities when two or more bomb craters overlap, which might also have a negative impact on efficacy.

Figure 7.4 portrays the results for different z_{out} graphs when the mutation operators were initialized with Louvain as described in section 4.2. Again, the different shapes for the same-colored lines denote the results of the algorithms after a specific generation. The Triangle mutation suffers from the same problem as beforehand: its quality significantly decreases over the generations, especially for the range $z_{out}/k \in [0.15, 0.35]$. CRATER performs better, although it does not improve the modularity obtained by Louvain. Its performance only drops for values above 0.4. Similar behavior is exhibited by Inherit. For the relevant range $z_{out}/k \in [0.0, 0.5]$, the Inherit mutation performs best. Nevertheless, all mutations deteriorate the Louvain results with the Inherit mutation proving to be the least poor of them.

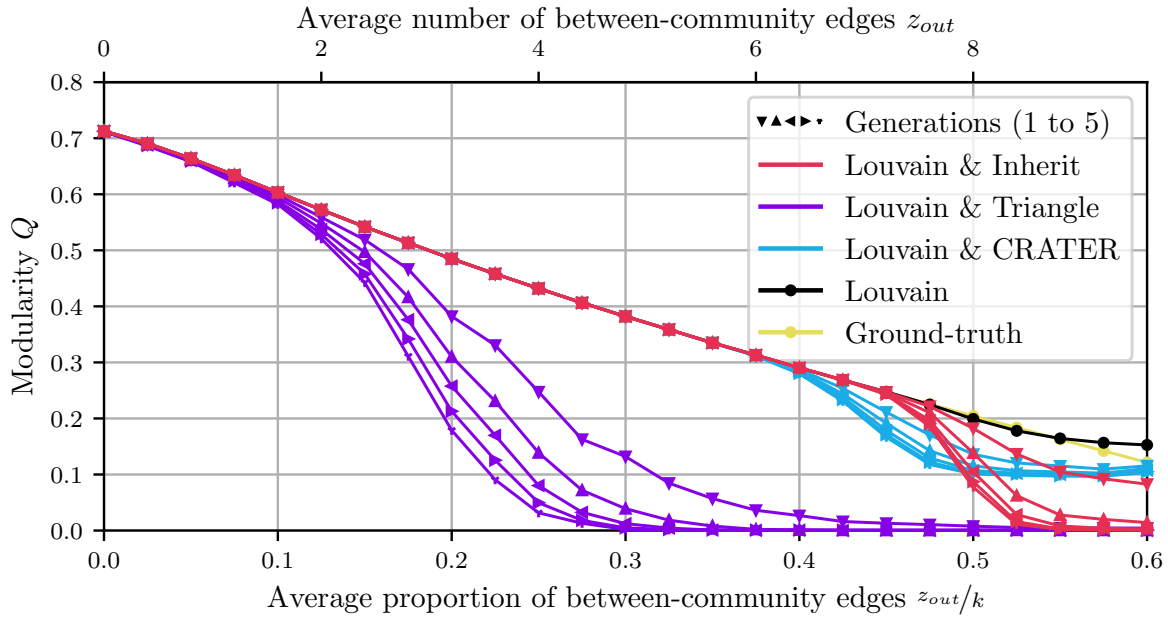
Louvain gives the correct vertex-community assignment up to a value of 0.4 and still classifies more than 80 % of the vertices correctly for $z_{out}/k = 0.5$, while coming close to ground-truth modularity. It even outperforms Newman's agglomerative method [16] where the fraction of classified vertices starts to significantly decrease with $z_{out}/k = 0.375$. Starting off $z_{out} \geq 6.75$ (corresponding to $z_{out}/k \approx 0.42$), his method classifies less than 80 % correctly, while Louvain still assigns correct communities to almost 100 % of the vertices. The fact that Louvain reaches even better values for Q than the ground truth for $z_{out}/k = 0.575$ and 0.6 is not surprising given that these graphs contain more edges

between communities than within-communities, and thus the ground-truth communities are no longer representative and not guaranteed to yield good modularity values.

We can summarize the results for the z_{out} graphs as follows: the random-initialized mutation operators do not measure up to Louvain, both in terms of the fraction of correctly classified vertices and the achieved modularity. When the operators were initialized with Louvain, they revealed better results in absolute measures but always degraded the metrics achieved with Louvain. We conclude that for this contrived setup of z_{out} graphs, the genetic algorithms failed and do not provide any benefit over the Louvain method, which is quick and bears even better results compared to literature, e. g. Newman's agglomerative method [16].



(a) Fraction of correctly classified vertices



(b) Modularity Q

Fig. 7.4.: Evaluation of Louvain-initialized mutation operators on various z_{out} graphs and different number of generations. Each point is an average over 1000 graphs.

7.4. Evaluation of Louvain

The Louvain algorithm was discussed in Chapter 3, including some C++ implementation details (section 3.1). We executed the algorithm on all real[2] graphs 1000 times each and report the results in Figure 7.5 and 7.6. Histograms visualize the distribution of number of occurrences of a certain modularity value (both figures), runtime (7.5) and number of found communities (7.6).

As is apparent from Figure 7.5, the implementation is very efficient: for lehnerer, ego-facebook, email-eu-core and speyer-web, it only takes a fraction of a second for Louvain to complete (on average and for the maximum value), e. g. around 22 ms on average for email-eu-core and around 90 ms for speyer-web. For larger graphs in our dataset – such as com-youtube, hep-ph-citations and ca-astroph, Louvain still only needs a few seconds on average. Note that the time to read in the graph data from a .CSV file is excluded as this is not indicative of the performance of the actual algorithm. We also omit the time to prepare the passes (see Algorithm 3.1.1, line 2) but this is negligible as even for the hep-ph-citations dataset the preparation of the graphs only takes around 15 ms (and for the smaller datasets only a fraction of a millisecond).

The histograms for modularity and time as well as for the number of communities strongly resemble normal distributions for a large part of the graphs. It is therefore reasonable to calculate common statistical measures such as mean μ and standard deviation σ. The Coefficient of Variation (CV) puts the two measures in relation: $CV = \sigma/\mu$. $CV = 100\,\%$ when the standard deviation is equal to the mean. While the CV for runtime is as high as around 29 % for the com-youtube dataset and as low as around 15 % for lehnerer, the CV values for modularity are in the range ≈ 0.0003 (ego-facebook) and ≈ 0.008 (email-eu-core). This implies that the variance of modularity is very low, while that of runtime is higher in general.

The third column in Figure 7.5 depicts how modularity and runtime are related to each other. Most occurrences are located on a spot in the upper left of the diagrams indicating that high modularity values are usually associated with low runtimes. This is reasonable as the algorithm takes less time if there is a high contraction of the graph throughout the hierarchy levels. In this case, the graph is smaller in the next Louvain passes and

[2]The Lehnerer graph is synthesized, but we will nevertheless include it in this category of "real-world" graphs as comparison to the z_{out} graphs.

thus less work has to be done resulting in a smaller runtime. At the same time, a higher contraction in the first passes usually goes along with higher modularity values as has been confirmed in preliminary tests, but it not formally proofed here. In sum, faster Louvain runs generally yield best modularity values.

Histograms for the number of communities found at the last hierarchy level are shown in Figure 7.6. Like runtime, this measure is dependent on the size of the network, e. g. for com-youtube we find considerably more communities than for the small email-eu-core network. The CV value is below 9 % for all graphs except for speyer-web where it reaches 29 % corresponding to a standard deviation of around 7.45. As can be seen from the 2D histogram this is because Louvain often identifies 21 and 22 communities but also 37 and 38 communities frequently. For 37/38 communities, modularity is usually higher than for the lower number of communities where a wider range of modularity values is covered.

The network visualization tool Gephi [46] provides a built-in community detection analysis that also employs the Louvain algorithm. The results in Table 6.1 are very similar to the ones reported here. Most of the time, Gephi's result are very close to the average quality we calculated in 1000 runs. Modularity values that deviate slightly more from the average are still within the ranges we found for the dataset and are due to the fact that we executed Gephi's analysis only once for every dataset.

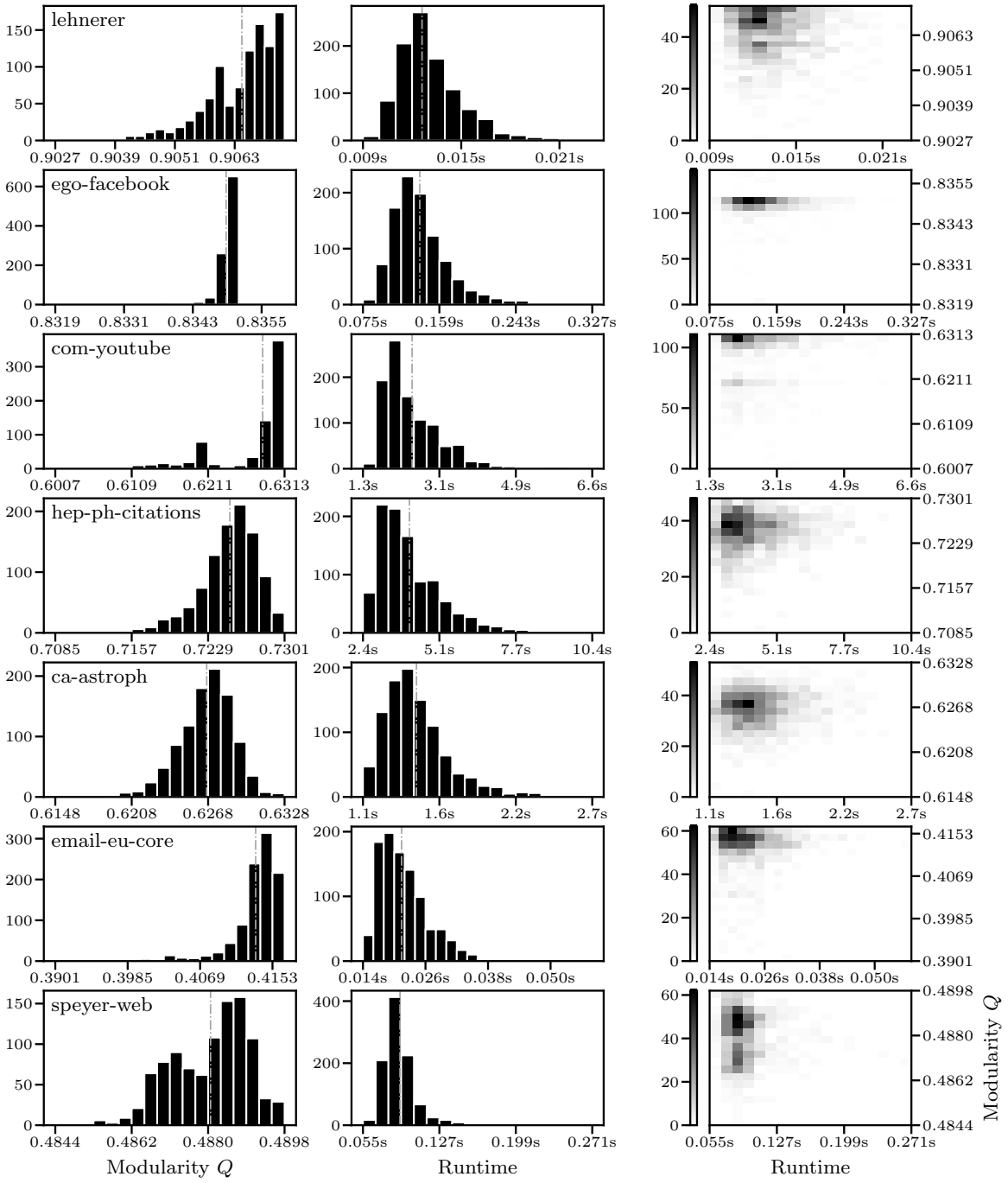


Fig. 7.5.: Louvain modularity and runtime performance for real-world datasets. The first column shows a modularity histogram, the second column a runtime histogram and the last column a 2D histogram of both measures. The number of occurrences is plotted on the y-axis (first two columns) or encoded as color (third column).

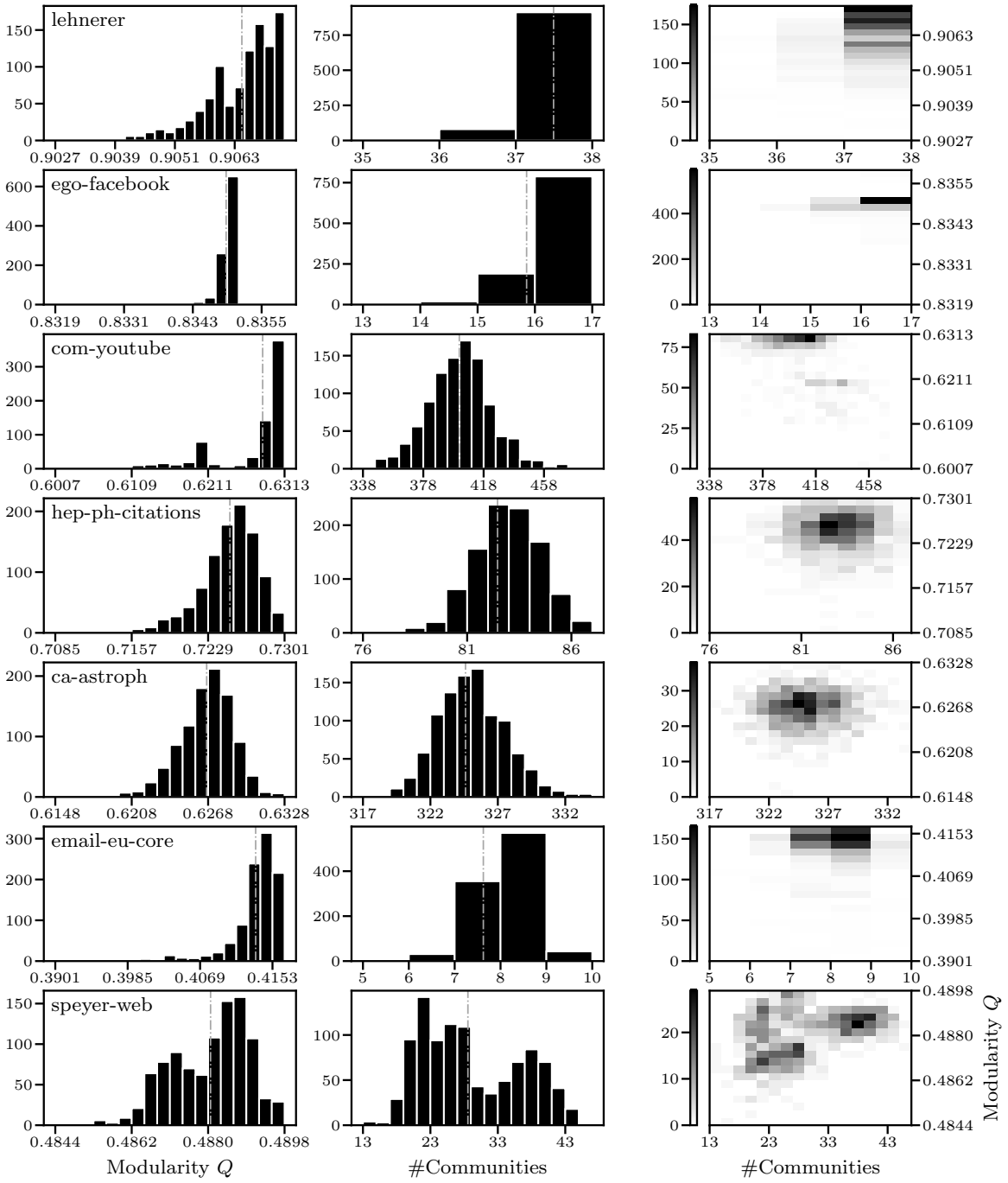


Fig. 7.6.: Louvain modularity and number of communities for real-world datasets. The first column shows a modularity histogram, the second column a histogram for the number of communities found and the last column a 2D histogram of both measures. The number of occurrences is plotted on the y-axis (first two columns) or encoded as color (third column).

7.5. Comparison of Louvain with Louvain-initialized mutations

As explained in section 4.2, we evaluate how well the mutation operators perform when initialized with the Louvain method. Figure 7.12 illustrates modularity Q, runtime and number of communities found with the Louvain-Initialized Inherit Mutation (LIIM) operator after three generations (marked by red bars). We compare with the scenario in which only the Louvain method is executed (black bars). Again, the algorithms are executed 1000 times for each dataset as described in section 7.4 to plot one row of the figure. The Louvain results from the mentioned section are reused to draw the plots but not to initialize the genetic algorithms. Instead the best out of three Louvain runs is chosen as initial vertex-community assignment for one run. Each of the 1000 runs is freshly initialized with three Louvain runs.

For lehnerer, ego-facebook, com-youtube and speyer-web, the variance of modularity Q is very small as can be seen in the histograms and also in the respective approximate CV values 1.04×10^{-7}, 9.32×10^{-7}, 1.71×10^{-6} and 4.29×10^{-5}. The reason for this is that the initial modularity achieved by Louvain could not be increased anymore (or only very slightly) by the Inherit operator. As we employ a $(\mu + \lambda)$ strategy, the modularity values do not fall below the initial one and therefore the number of communities is also not altered because the children are discarded altogether. As already discussed with regards to the z_{out} graphs (section 7.3), this is not always the case due to a bug in the code or numerical instabilities.

The small email-eu-core network is an example for this behavior. The average modularity dropped from ≈ 0.4133 to ≈ 0.3602. While Louvain found seven or eight communities most of the time, applying the Inherit mutation afterwards yields around one community more on average (note the logarithmic scale of all graphs in Figure 7.12). To illustrate this, we pick the results of one of the 1000 runs and visualize them in Figure 7.7. Louvain achieves a modularity of around 0.4137 (8 communities) whereas LIIM only scores a value of about 0.3743 (9 communities). The additional community comprises just three vertices which are shown in white in Figure 7.7. One of them is located in the bottom amid dark blue vertices, another at the top among light green vertices and the last one between the dark green and black community in the upper left of this network. This reveals an

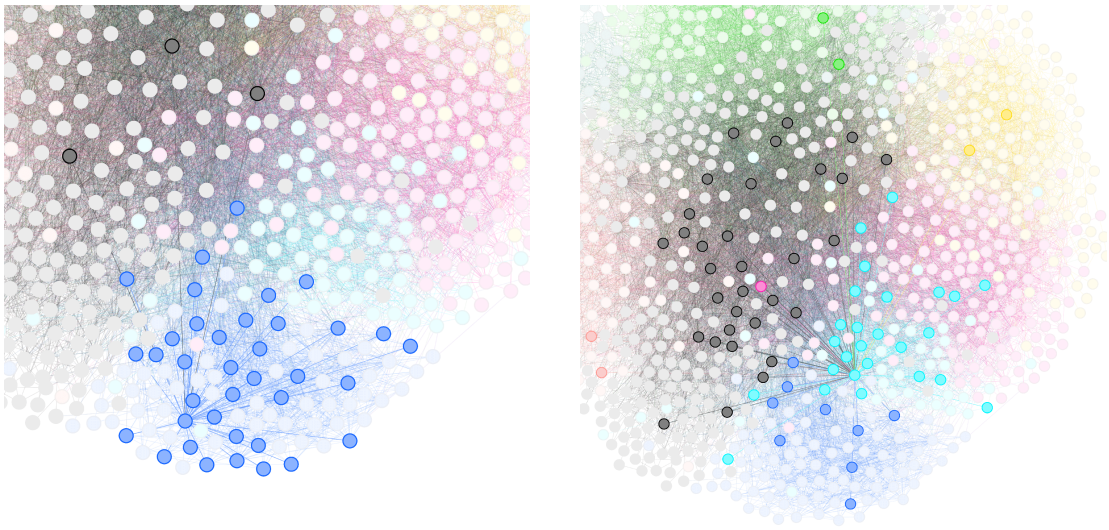
inherent flaw of the local optimization presented in section 4.3: while it is ensured that singleton communities are eliminated, we do not check whether the subgraphs induced by the vertices of a community are connected. In the example, the white community is not considered a singleton community (as it consists of more than one node), yet the vertices are not connected. This flaw was only discovered very late and we could not fix it yet. It might explain some of the less-than-ideal results.

(a) Only Louvain (b) After three generations of LIIM

Fig. 7.7.: Vertex-community assignments for the email-eu-core network

Another interesting observation is that LIIM has further enlarged the big black community, e. g. to the bottom right light blue community. Figure 7.8 attempts to explain this behavior by highlighting two vertices (one main vertex in each subfigure) and their respective neighbors while coloring them according to the Louvain results. In Figure 7.8a, a vertex in the dark blue community is highlighted. It features many edges to vertices of its own community and only few to other communities like the black one. In contrast, a vertex from the light blue community is highlighted in Figure 7.8b. It is connected to members of its own community but also to many black vertices. This scenario is comparable to a high z_{out} value for the synthesized graphs where we have already seen the poor performance of the genetic operators. In this case, it is very likely that the Inherit operation will include the light blue vertices in the black community – which might indeed slightly increase Q. It could also be that in this case, the increase is very slightly negative, but changes sign due to accumulated numerical errors. Preliminary analyses show that this could be the case, but further research is necessary. Louvain yields better results for this real-world

example; by continuously trying to achieve local maxima with respect to the whole graph, this method seems to be more far-sighted than the genetic approaches which only try to locally optimize modularity in a small area of the network.



(a) Main vertex is in same community in Louvain and LIIM – easy scenario

(b) Main vertex is in the light blue community for Louvain and gets assigned the black community with LIIM – difficult scenario

Fig. 7.8.: Highlighting vertices and their neighbors for excerpts from the email-eu-core network. Vertices are colored according to Louvain (compare Figure 7.7a). "Main vertex" refers to the vertex whose neighbors are shown.

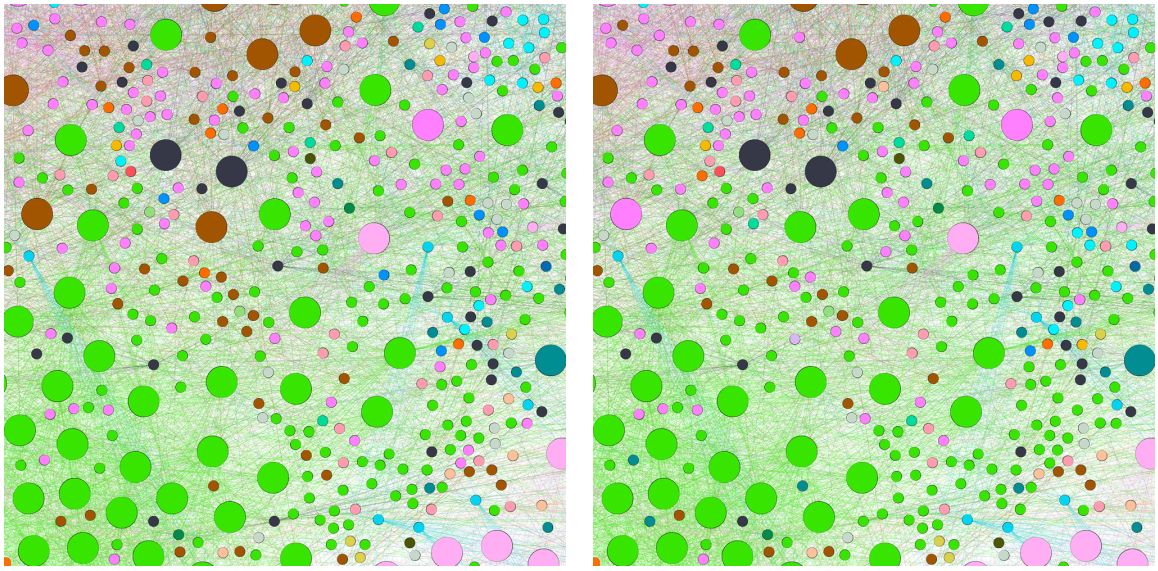
For the ca-AstroPh network, we achieve a small improvement. Not only has the average modularity increased from about 0.6266 (Louvain) to 0.6359 (LIIM), but also the maximum achieved modularity in 1000 runs is slightly higher: for Louvain $Q_{max} \approx 0.6333$ and for LIIM we get $Q_{max} \approx 0.6364$, an improvement of about 0.5 %. In Figure 7.9, subfigure 7.9a shows the vertex-community assignment from the best of the three Louvain runs used as initialization for the Inherit mutation. Louvain yields a modularity of ≈ 0.6296. In comparison: after three runs, the assignment depicted in Figure 7.9b results in a modularity of ≈ 0.6357. Both approaches have found 327 communities. From the two figures against the backdrop of the filtered ca-AstroPh network for vertices with $k_v \geq 50$, it is not immediately apparent what caused this increase in modularity. However, we observe the same tendency as beforehand: the biggest community in the graph (here the magenta-colored area slightly left from the center) enlarges and this part of the graph

appears more homogenous afterwards. Readers of the electronic version might zoom in to recognize the differences more easily. This behavior is also evident with respect to the unfiltered ca-AstroPh graph; Figure 7.10 and 7.11 also feature example where areas become more homogeneous judging by their colors. This characteristic of LIIM might explain the modularity increase we obtained. However, the gain we obtained is very small and thus it cannot be ruled out that it results from other, not yet considered phenomena. For the 1000 runs, the modularity histogram clearly shows that the normal distribution for ca-AstroPh has shifted towards the right (Figure 7.12).

(a) Only Louvain

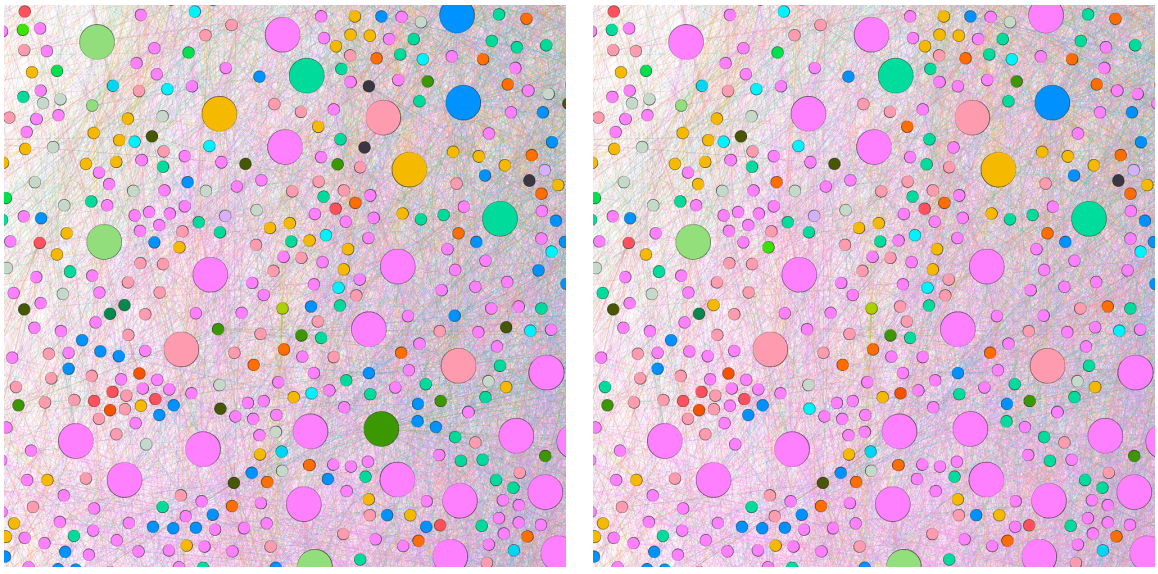
(b) After three generations of LIIM

Fig. 7.9.: Vertex-community assignments for the ca-AstroPh network



(a) Only Louvain (b) After three generations of LIIM

Fig. 7.10.: Vertex-community assignments for a section of the ca-AstroPh network (center bottom). Vertex size according to degree.



(a) Only Louvain (b) After three generations of LIIM

Fig. 7.11.: Vertex-community assignments for a section of the ca-AstroPh network (center left). Vertex size according to degree.

Beyond the previous results, Figure 7.12 also compares the runtime of Louvain with that of LIIM in the second column. To allow for a fair comparison, the runtimes of the *Only Louvain* method are doubled[3]. At the same time, the average runtime of the *Only Louvain* method is added to the runtimes of the Inherit operator in LIIM. This implies that while Louvain was executed three times in the initialization step, we show the runtime results as if Louvain was run only once. Due to the aforementioned low variance in the quality of Louvain, this should not impose a big problem in reality, i. e. it should suffice to execute Louvain only once in the beginning as the initialization step and still achieve good initial results. From the runtime histograms, it can be seen that the average total runtime increases for all networks but is still within a tolerable range, e. g. it takes only half a second longer for ca-AstroPh. Total runtimes when LIIM consists of only one generation can be found in Figure 7.12. The achieved quality values are very similar but partly a little lower and display a greater variance. This shows that the genetic algorithm works properly and is able to improve the population through multiple generations.

We also include results for the Triangle and the Louvain-initialized Louvain CRATER mutation operators. However, they perform very badly with regards to modularity and runtime. Not a single execution was able to outperform the initial Louvain modularity and oftentimes even degraded the results. The Triangle operator additionally takes very long time, sometimes up to several minutes for a single run (including Louvain and three generations) as seen in Figure A.10. As already observed for the z_{out} graphs, the random-initialized mutation operators do not come close to the Louvain results. In terms of runtime, Inherit outperforms Louvain for small graphs but not for larger ones such as com-youtube or hep-ph-citations. All results are included in the appendix.

[3]We compare two Louvain executions with the Louvain-initialized genetic algorithm where one execution of Louvain is followed by the GA

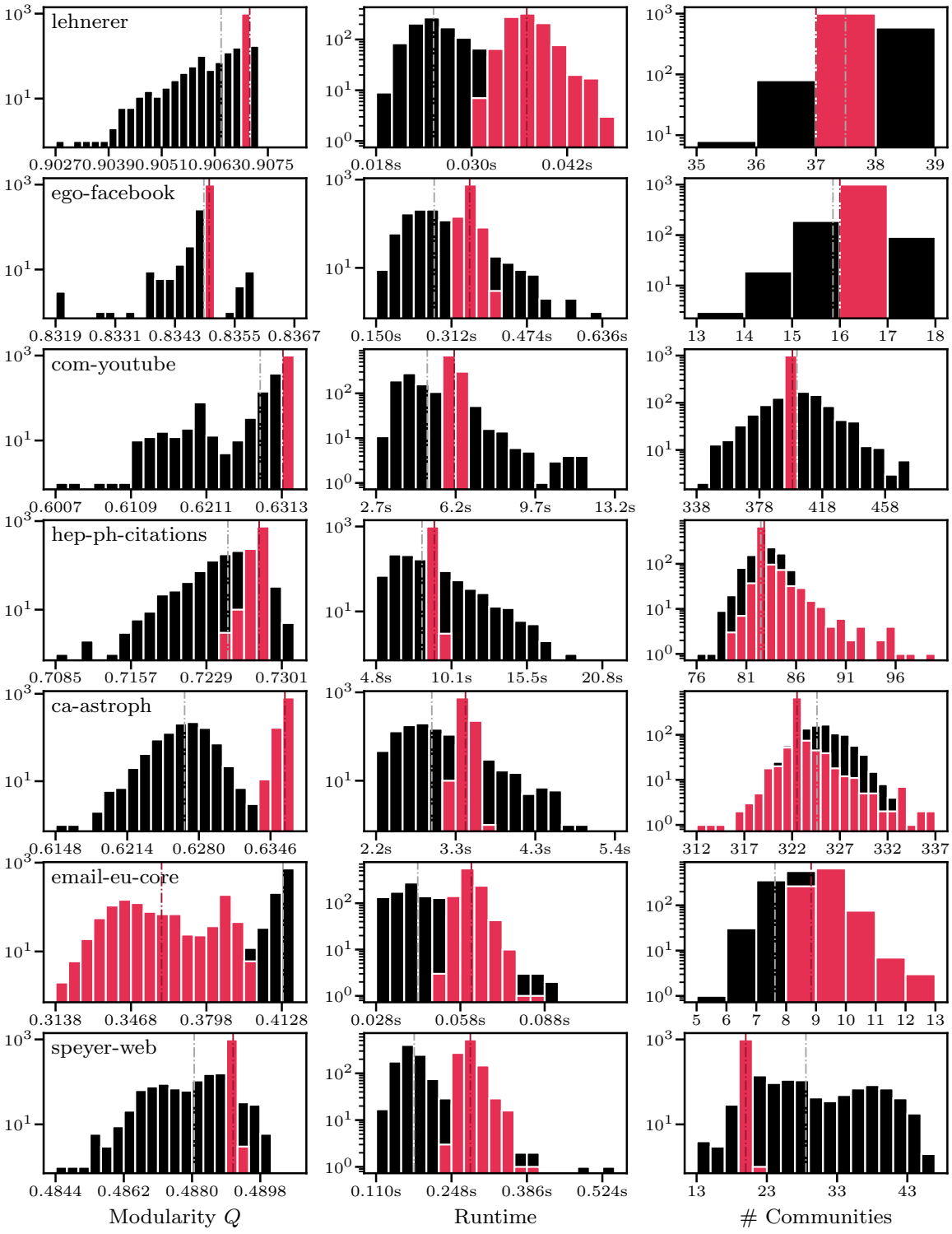


Fig. 7.12.: Comparison of Louvain with Louvain-initialized Inherit on real-world datasets after 3 generations. First column: modularity histograms. Second column: runtime histograms. Third column: number of communities histograms. The number of occurrences is plotted on the y-axis.

8. Conclusion

Let us briefly outline the particular noteworthy points of this thesis: in Chapter 3, we explained in detail how the Louvain method works. To fill a gap of the original paper [8], we derived every required formula including a quick ΔQ calculation. The latter was then generalized for application in the genetic algorithms. The Louvain method was completely implemented for the SAP HANA Graph Engine and is ready to be included in the language syntax of a graph-specific custom programming language at SAP. Benchmarks show that the optimized implementation is very efficient: even for large graphs such as com-YouTube (around $47,000$ vertices and $300,000$ edges) the average runtime is only about 2.5 s.

In Chapter 4, we presented the new idea of applying Louvain as initialization heuristic for genetic algorithms and described points worth noting when this technique is employed. The *Assign random community* and *Inherit community from neighborhood* mutation operators were taken from Lehnerer [10], while two new mutation operators *Triangle* and *Louvain CRATER* were introduced including pseudo-code. All mutations were implemented in C++.

Datasets used for a comprehensive experimental evaluation were described in Chapter 6. They include the common synthesized networks introduced by Girvan and Newman in [11], where a parameter z_{out} is varied as to allow for different degrees of difficulty in terms of identifying communities. We also evaluated results on a synthesized network from Lehnerer as well as various real-world networks from SNAP [45], which we filtered in some cases. Additionally, a custom web-graph called "speyer-web" was generated. All networks are available for download, see the link in Chapter 6.

Finally, in Chapter 7, we analyzed the results obtained for the algorithms on the various datasets. We conclude with the following observations: the randomly initialized genetic operators do not work well for the z_{out} graphs. They achieve better results when initialized with Louvain but still fall short of expectations compared to the Louvain method itself. Bad start heuristics are avoided by executing Louvain three times in the beginning and picking the best run as initializer for the GA. However, one Louvain run usually

suffices in reality as the variance of the modularity values achieved by Louvain is very small. Moreover, we show that we can improve the quality achieved by Louvain by 0.5 % for the ca-AstroPh network through the Louvain-Initialized Inherit Mutation (LIIM). However, this gain is insignificant and also only appears with this specific dataset. Louvain outperforms the genetic algorithms both in terms of achieved modularity Q and runtime. Out of the presented operators, the Inherit mutation by Lehnerer scores best, while Triangle and Louvain CRATER take longer and cannot measure up to Inherit in terms of modularity.

Further research is required for hyperparameter optimization of the GA, including variables such as the number of generations, size of the population and probability of applying individual mutation operators. Experimenting with the number of parents selected for each population λ might yield better results compared to the fixed $\lambda = 1$ that we used for all experiments. Moreover, the local optimization could be improved such that the graph induced by the vertices of one community stays connected. This can be achieved by employing articulation points (cut vertices) which can be calculated in linear time with Hopcroft and Tarjan's algorithm [54]. Furthermore, the normalized mutual information measure introduced in [18] could be implemented to classify which vertices were assigned the correct communities in the z_{out} graphs more accurately. Beyond that, numerical instabilities regarding the generalized delta modularity calculation need to be further investigated to ensure mutations cannot degrade modularity. Finally, GAs lend themselves very well to parallel processing. Future work could therefore greatly improve runtime performance by implementing concurrent techniques. However, the main focus should be on further analysis of the genetic operators and how they can be improved.

Bibliography

We highly recommend BarabásiLab's book on Network Science which is also published as website [55]. It covers basic graph theory and many advanced concepts of this vivid field that brings various scientific disciplines together. Chapter 9 in the book addresses communities and also includes the Louvain algorithm.

[1] Erdös, P./ Rényi, A. "On Random Graphs I." In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290. (Visited on 08/22/2022).

[2] Albert, R./ Barabási, A.-L. "Statistical Mechanics of Complex Networks." In: *Reviews of Modern Physics* 74.1 (2002-01), pp. 47–97. URL: https://arxiv.org/pdf/cond-mat/0106096.pdf (visited on 07/07/2022).

[3] Albert, R./ Jeong, H./ Barabási, A.-L. "Diameter of the World-Wide Web." In: *Nature* 401.6749 (1999), pp. 130–131. (Visited on 07/21/2022).

[4] D. Lalwani/ D. V. L. N. Somayajulu/ P. R. Krishna. "A community driven social recommendation system." In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, pp. 821–826. DOI: 10.1109/BigData.2015.7363828.

[5] Ogerta Elezaj/ Sule Yildirim Yayilgan/ Edlira Kalemi. "Criminal Network Community Detection in Social Media Forensics." In: *Intelligent technologies and applications*. Ed. by Yayilgan, S. Y./ Bajwa, I. S./ Sanfilippo, F. Communications in computer and information science, 1865-0937. Cham, Switzerland: Springer, 2021, pp. 371–383. DOI: 10.1007/978-3-030-71711-7_31. URL: https://www.researchgate.net/publication/350055897_Criminal_Network_Community_Detection_in_Social_Media_Forensics.

[6] N. Haq/ Z. J. Wang. "Community detection from genomic datasets across human cancers." In: *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 2016, pp. 1147–1150. DOI: 10.1109/GlobalSIP.2016.7906021. (Visited on 08/26/2022).

[7] Brandes, U. et al. "Maximizing Modularity is hard." In: (2006-08-30). URL: https://arxiv.org/pdf/physics/0608255.pdf (visited on 04/28/2022).

[8] Blondel, V. D. et al. "Fast unfolding of communities in large networks." In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (2008-04). DOI: 10.1088/1742-5468/2008/10/P10008. URL: https://perso.uclouvain.be/vincent.blondel/publications/08BG.pdf (visited on 06/02/2022).

[9] Borhan Kazimipour/ Xiaodong Li/ A.K. Qin. "A Review of Population Initialization Techniques for Evolutionary Algorithms." In: *2014 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2585–2592. DOI: 10.1109/CEC.2014.6900618. URL: https://www.researchgate.net/publication/263011604_A_Review_of_Population_Initialization_Techniques_for_Evolutionary_Algorithms (visited on 08/03/2022).

[10] Lehnerer, S. "Community Detection in Complex Networks using Genetic Algorithms." In: (). URL: https://dl.gi.de/bitstream/handle/20.500.12116/28981/SKILL2018-03.pdf?sequence=1&isAllowed=y (visited on 05/30/2022).

[11] Newman, M. E. J./ Girvan, M. "Finding and evaluating community structure in networks." In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 69.2 (2004-02). DOI: 10.1103/PhysRevE.69.026113. URL: https://doi.org/10.1103%2Fphysreve.69.026113 (visited on 04/07/2022).

[12] Fortunato, S. "Community detection in graphs." In: *Physics Reports* 486.3-5 (2010), pp. 75–174. DOI: 10.1016/j.physrep.2009.11.002. URL: https://arxiv.org/pdf/0906.0612.pdf (visited on 02/11/2022).

[13] Khan, B. S./ Niazi, M. A. *Network Community Detection: A Review and Visual Survey.* 08/03/2017. URL: https://arxiv.org/pdf/1708.00977 (visited on 08/23/2022).

[14] Cai, Q. et al. "A survey on network community detection based on evolutionary computation." In: *International Journal of Bio-Inspired Computation* 8.2 (2016), p. 84. DOI: 10.1504/IJBIC.2016.076329. URL: https://www.researchgate.net/publication/286331285_A_survey_on_network_community_detection_based_on_evolutionary_computation (visited on 07/06/2022).

[15] Tasgin, M./ Herdagdelen, A./ Bingol, H. *Community Detection in Complex Networks Using Genetic Algorithms.* DOI: Based. URL: https://arxiv.org/pdf/0711.0491.pdf (visited on 01/06/2022).

[16] Newman, M. E. J. “Fast algorithm for detecting community structure in networks.” In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 69.6 Pt 2 (2004), p. 066133. DOI: 10.1103/PhysRevE.69.066133.

[17] Pizzuti, C. “GA-Net: A Genetic Algorithm for Community Detection in Social Networks.” In: *Parallel problem solving from nature - PPSN X*. Ed. by Rudolph, G. et al. Vol. 5199. Lecture Notes in Computer Science. Berlin: Springer, 2008, pp. 1081–1090. DOI: 10.1007/978-3-540-87700-4_107. URL: https://staff.fmi.uvt.ro/~daniela.zaharie/ma2016/projects/applications/communities_detection/GA_NET.pdf (visited on 06/13/2022).

[18] Danon, L. et al. “Comparing community structure identification.” In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.09 (2005), P09008–P09008. DOI: 10.1088/1742-5468/2005/09/P09008. URL: https://iopscience.iop.org/article/10.1088/1742-5468/2005/09/P09008/pdf (visited on 07/07/2022).

[19] Clauset, A./ Newman, M. E. J./ Moore, C. “Finding community structure in very large networks.” In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 70.6 (2004), p. 066111. DOI: 10.1103/PhysRevE.70.066111. URL: https://doi.org/10.1103%2Fphysreve.70.066111 (visited on 01/07/2022).

[20] Newman, M. E. J. “Modularity and community structure in networks.” In: *Proceedings of the National Academy of Sciences* 103.23 (2006), pp. 8577–8582. DOI: 10.1073/pnas.0601602103. URL: https://doi.org/10.1073%2Fpnas.0601602103 (visited on 04/07/2022).

[21] Arenas, A. et al. “Size reduction of complex networks preserving modularity.” In: *New Journal of Physics* 9.6 (2007), p. 176. DOI: 10.1088/1367-2630/9/6/176. URL: https://arxiv.org/pdf/physics/0702015.pdf (visited on 08/25/2022).

[22] Ulrik Brandes et al. “On Modularity Clustering.” In: *Knowledge and Data Engineering, IEEE Transactions on* 20 (2008), pp. 172–188. DOI: 10.1109/TKDE.2007.190689. URL: https://kops.uni-konstanz.de/bitstream/handle/123456789/5853/modularity.pdf (visited on 06/07/2022).

[23] Fortunato, S./ Barthélemy, M. “Resolution limit in community detection.” In: *Proceedings of the National Academy of Sciences* 104.1 (2007), pp. 36–41. DOI: 10.1073/pnas.0605965104. URL: https://www.pnas.org/doi/pdf/10.1073/pnas.0605965104 (visited on 06/07/2022).

[24] "Defining and identifying communities in networks." In: (2004). URL: https://arxiv.org/pdf/cond-mat/0309488.pdf (visited on 07/07/2022).

[25] Holland, J. H./ Holland, P. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975. URL: https://books.google.de/books?id=JE5RAAAAMAAJ.

[26] Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Complex Adaptive Systems. MIT Press, 1992. URL: https://books.google.de/books?id=5EgGaBkwvWcC.

[27] Cohen, M. W./ Aga, M./ Weinberg, T. "Genetic Algorithm Software System for Analog Circuit Design." In: *Procedia CIRP* 36 (2015), pp. 17–22. DOI: 10.1016/j.procir.2015.01.033. URL: https://www.sciencedirect.com/science/article/pii/S2212827115000360 (visited on 08/20/2022).

[28] Potvin, J.-Y. "Genetic algorithms for the traveling salesman problem." In: *Annals of Operations Research* 63.3 (1996), pp. 337–370. DOI: 10.1007/BF02125403. (Visited on 08/20/2022).

[29] Saleh, S./ Olson, B./ Shehu, A. "A population-based evolutionary search approach to the multiple minima problem in de novo protein structure prediction." In: *BMC Structural Biology* 13 Suppl 1.1 (2013), S4. DOI: 10.1186/1472-6807-13-S1-S4. URL: https://bmcstructbiol.biomedcentral.com/articles/10.1186/1472-6807-13-S1-S4 (visited on 08/25/2022).

[30] Luo, X.-L./ Feng, J./ Zhang, H.-H. "A genetic algorithm for astroparticle physics studies." In: *Computer Physics Communications* 250 (2019-07-01), p. 106818. DOI: 10.1016/j.cpc.2019.06.008. URL: https://www.sciencedirect.com/science/article/pii/S001046551930195X (visited on 08/20/2022).

[31] Prajneshu. "Genetic algorithms and their applications: an overview." In: (2008-03-18). DOI: 5863. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.613.5863&rep=rep1&type=pdf (visited on 08/20/2022).

[32] Downes, S. M./ Matthews, L. "Heritability." In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Metaphysics Research Lab, Stanford University, 2020.

[33] Darwin, C. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life.* New York, D. Appleton and Co, 1871, 1859. URL: http://darwin-online.org.uk/content/frameset?itemID=F373&viewtype=text&pageseq=1 (visited on 08/20/2022).

[34] Simoncini, D./ Zhang, K. Y. "Population-Based Sampling and Fragment-Based De Novo Protein Structure Prediction." In: *Encyclopedia of bioinformatics and computational biology.* Ed. by Ranganathan, S. Amsterdam, Netherlands: Elsevier, 2019, pp. 774–784. DOI: 10.1016/B978-0-12-809633-8.20507-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128096338205074 (visited on 08/17/2022).

[35] Lancichinetti, A./ Fortunato, S. "Community detection algorithms: a comparative analysis." In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 80.5 Pt 2 (2009), p. 056117. DOI: 10.1103/PhysRevE.80.056117. URL: https://arxiv.org/pdf/0908.1062.pdf (visited on 07/12/2022).

[36] Meo, P. de et al. "Generalized Louvain method for community detection in large networks." In: *2164-7143* (2011), pp. 88–93. DOI: 10.1109/ISDA.2011.6121636. URL: https://www.researchgate.net/publication/51929918_Generalized_Louvain_Method_for_Community_Detection_in_Large_Networks (visited on 07/11/2022).

[37] "Notes on Modularity." In: (2010-01-07). URL: https://www.hongliangjie.com/notes/modularity.pdf (visited on 05/07/2022).

[38] Blondel, V. D. et al. *Louvain method: Finding communities in large networks: A Multi-Level Aggregation Method for optimizing modularity.* 04/20/2021. URL: https://sites.google.com/site/findcommunities/ (visited on 06/09/2022).

[39] *louvain_communities — NetworkX 2.8.5 documentation.* 06/23/2022. URL: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.louvain.louvain_communities.html (visited on 06/23/2022).

[40] Borna, K./ Hashemi, V. H. "An Improved Genetic Algorithm with a Local Optimization Strategy and an Extra Mutation Level for Solving Traveling Salesman Problem." In: *International Journal of Computer Science, Engineering and Information Technology* 4.4 (2014), pp. 47–53. DOI: 10.5121/ijcseit.2014.4405. URL: https://arxiv.org/ftp/arxiv/papers/1409/1409.3078.pdf (visited on 08/25/2022).

[41] Braun, H. *Evolutionäre Algorithmen und ihre Anwendung: Vorlesung an der Dualen Hochschule Baden-Württemberg Karlsruhe.*

[42] Lehnerer, S. *Genetic algorithm community detection: Data and code.* GitHub, 06/28/2022. URL: https://github.com/SimonNick/genetic-algorithm-community-detection (visited on 06/29/2022).

[43] *uniform_int_distribution - C++ Reference.* 12/04/2013. URL: https://cplusplus.com/reference/random/uniform_int_distribution/ (visited on 07/08/2022).

[44] Schrimpf, G. et al. *Record Breaking Optimization Results Using the Ruin and Recreate Principle.* Vol. 159. 2000. DOI: 10.1006/jcph.1999.6413. URL: https://www.sciencedirect.com/science/article/pii/S0021999199964136 (visited on 08/11/2022).

[45] Leskovec, J./ Krevl, A. *SNAP Datasets: Stanford Large Network Dataset Collection.* 2014.

[46] Bastian, M./ Heymann, S./ Jacomy, M. *Gephi: An Open Source Software for Exploring and Manipulating Networks.* 2009. URL: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

[47] Jacomy, M. et al. "ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software." In: *PloS one* 9.6 (2014), e98679. DOI: 10.1371/journal.pone.0098679. URL: https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0098679&type=printable (visited on 10/08/2022).

[48] Hu, Y. "Efficient and High Quality Force-Directed Graph Drawing." In: *Mathematica Journal* 10 (2005), pp. 37–71. URL: http://yifanhu.net/PUB/graph_draw_small.pdf (visited on 08/10/2022).

[49] Leskovec, J./ McAuley, J. "Learning to Discover Social Circles in Ego Networks." In: *Advances in Neural Information Processing Systems.* Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc, 2012. URL: https://proceedings.neurips.cc/paper/2012/file/7a614fd06c325499f1680b9896beedeb-Paper.pdf.

[50] Mislove, A. et al. "Measurement and Analysis of Online Social Networks." In: *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07).* San Diego, CA, 2007.

[51] *KDD Cup 2003 - Datasets*. 09/04/2003. URL: https://www.cs.cornell.edu/projects/kddcup/datasets.html (visited on 06/21/2022).

[52] Fielding, R. *Relative Uniform Resource Locators*. 1995. DOI: 10.17487/RFC1808. URL: https://www.rfc-editor.org/rfc/rfc1808.html#section-2.1.

[53] Wikipedia, ed. *List of Wikipedias*. 2022. URL: https://en.wikipedia.org/w/index.php?title=List_of_Wikipedias&oldid=1104208867 (visited on 08/15/2022).

[54] Hopcroft, J./ Tarjan, R. "Algorithm 447: efficient algorithms for graph manipulation." In: *Communications of the ACM* 16.6 (1973), pp. 372–378. DOI: 10.1145/362248.362272.

[55] BarabásiLab. *Network Science by Albert-László Barabási*. 07/26/2021. URL: http://networksciencebook.com/ (visited on 07/09/2022).

A. Appendix

A.1. ca-AstroPh

The context for this figure is given in section 7.5.

(a) Only Louvain (b) After three generations of LIIM

Fig. A.1.: Vertex-community assignments for the unfiltered ca-AstroPh network. Vertex size according to degree.

A.2. Random-initialized mutations

We refer the reader to section 7.5 for background information on the following diagrams. The first column always depicts modularity histograms, and the second column shows either runtime histograms or histograms for the number of identified communities. The last column presents 2D histograms of both measures to quickly recognize correlations. The number of occurrences is plotted on the y-axis (first two columns) or encoded as color (third column). One dataset corresponds to one row in the grid of plots. Note that we chose a logarithmic scale for some y-axes, but never for an x-axis. Furthermore, the plots are merely arranged as grid to save space; when directly comparing histograms, the reader should note the different minimum and maximum values on the axes, e. g. the same color in the 2D histogram may indicate a different number of occurrences for the various datasets.

Due to a runtime limit, we could not finish 1000 runs for every method-dataset combination. We therefore list the number of runs we collected before the script terminated:

- Random-initialized Inherit: 1000 runs for all datasets
- Random-initialized Triangle: lehnerer 1000, ego-facebook 376, com-youtube 96, hep-ph-citations 134, ca-astroph 208, email-eu-core 1000, speyer-web 600
- Random-initialized Louvain CRATER: 1000 runs for all datasets other than only 324 for com-youtube and 879 for hep-ph-citations

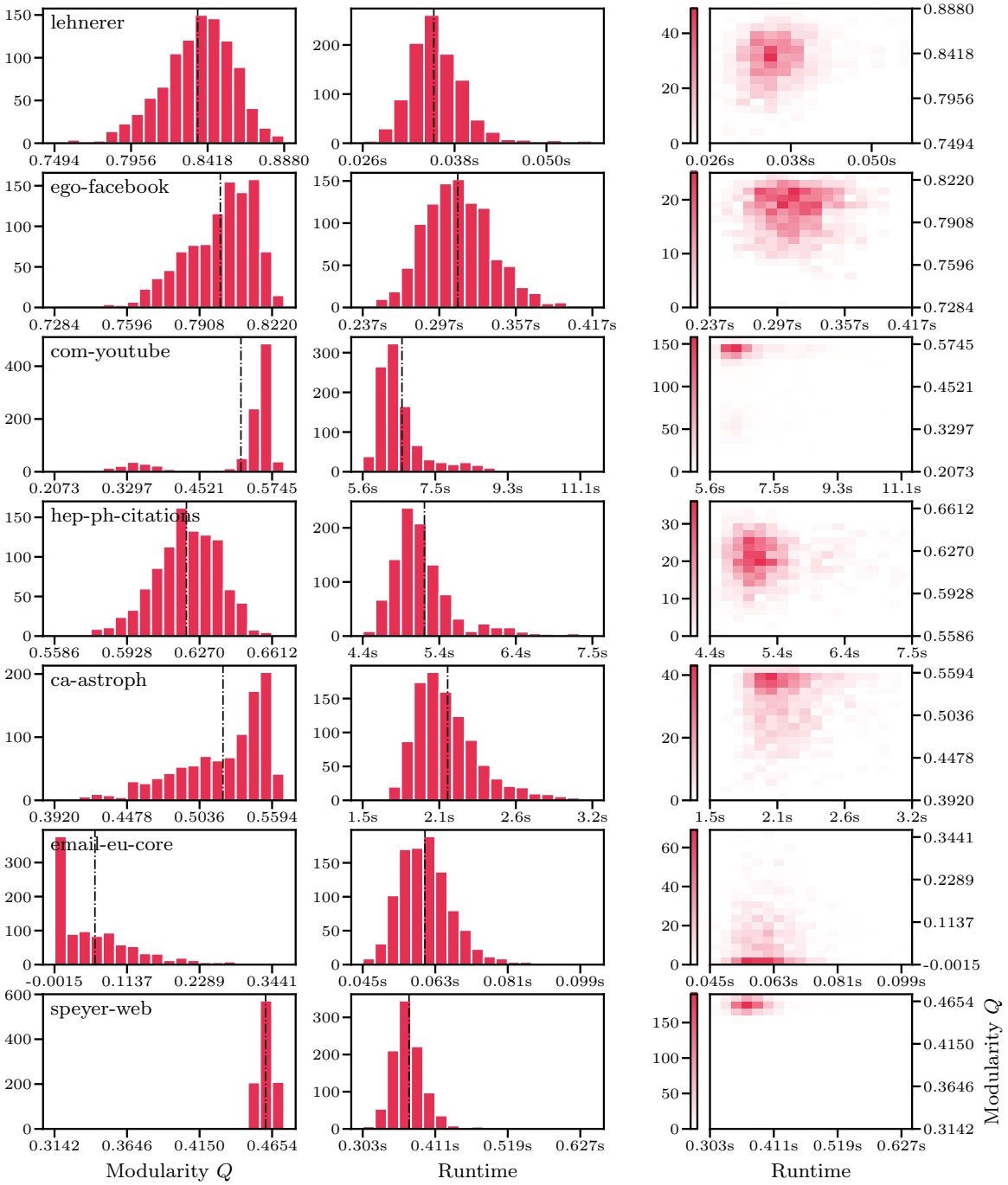


Fig. A.2.: Random-initialized Inherit modularity and runtime performance for real-world datasets. Results after 5 generations.

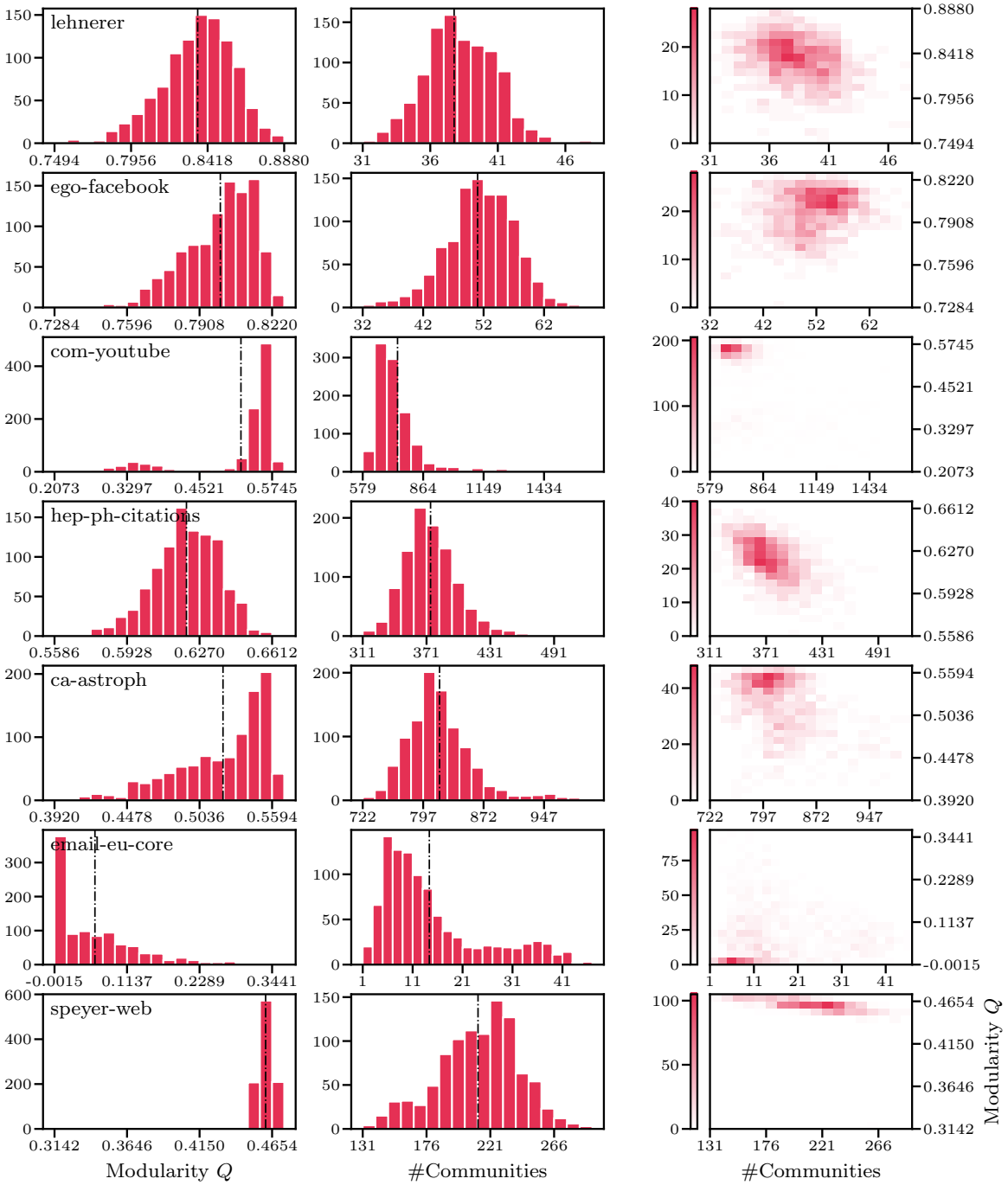


Fig. A.3.: Random-initialized Inherit modularity and number of communities for real-world datasets. Results after 5 generations.

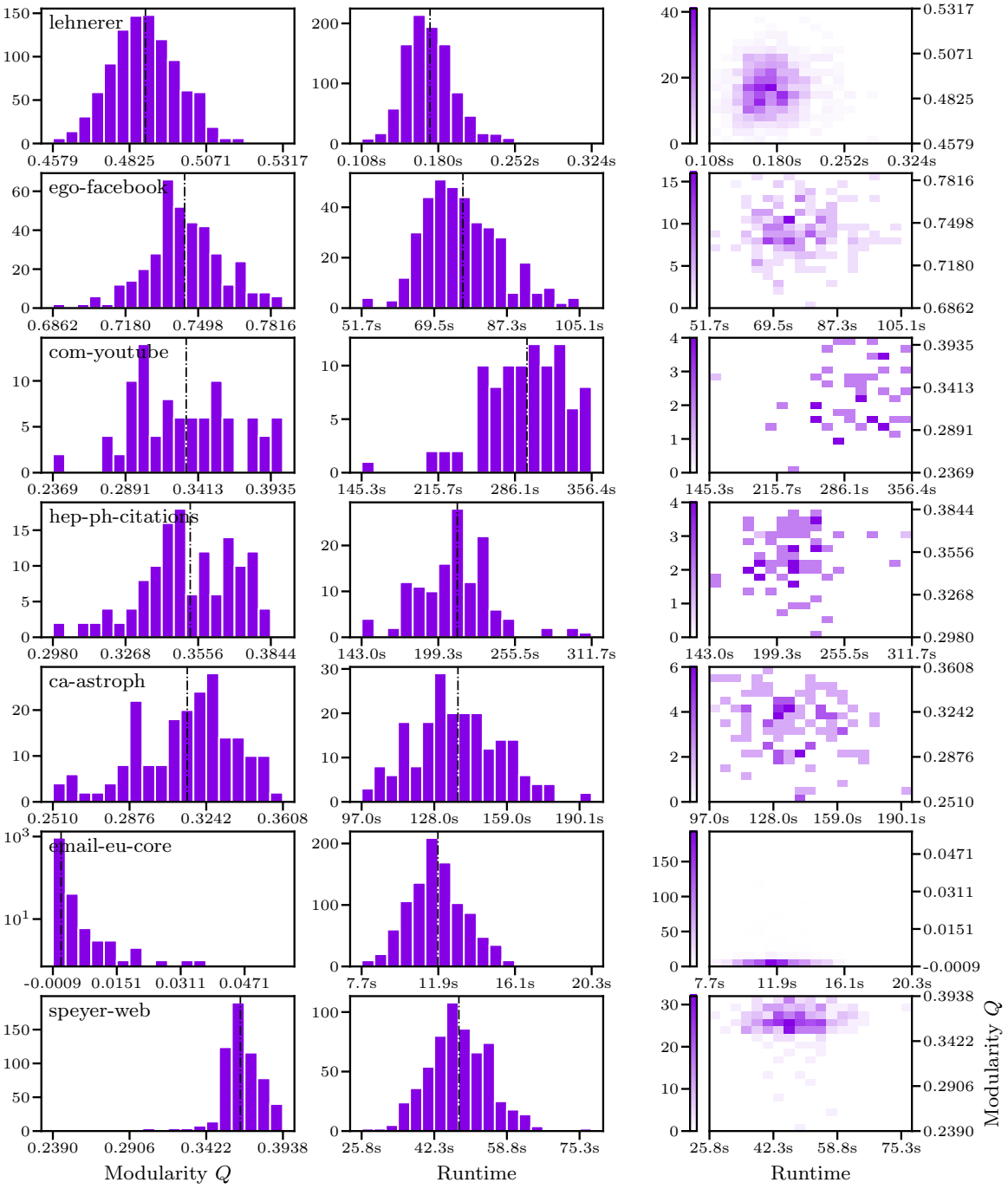


Fig. A.4.: Random-initialized Triangle modularity and runtime performance for real-world datasets. Results after 5 generations.

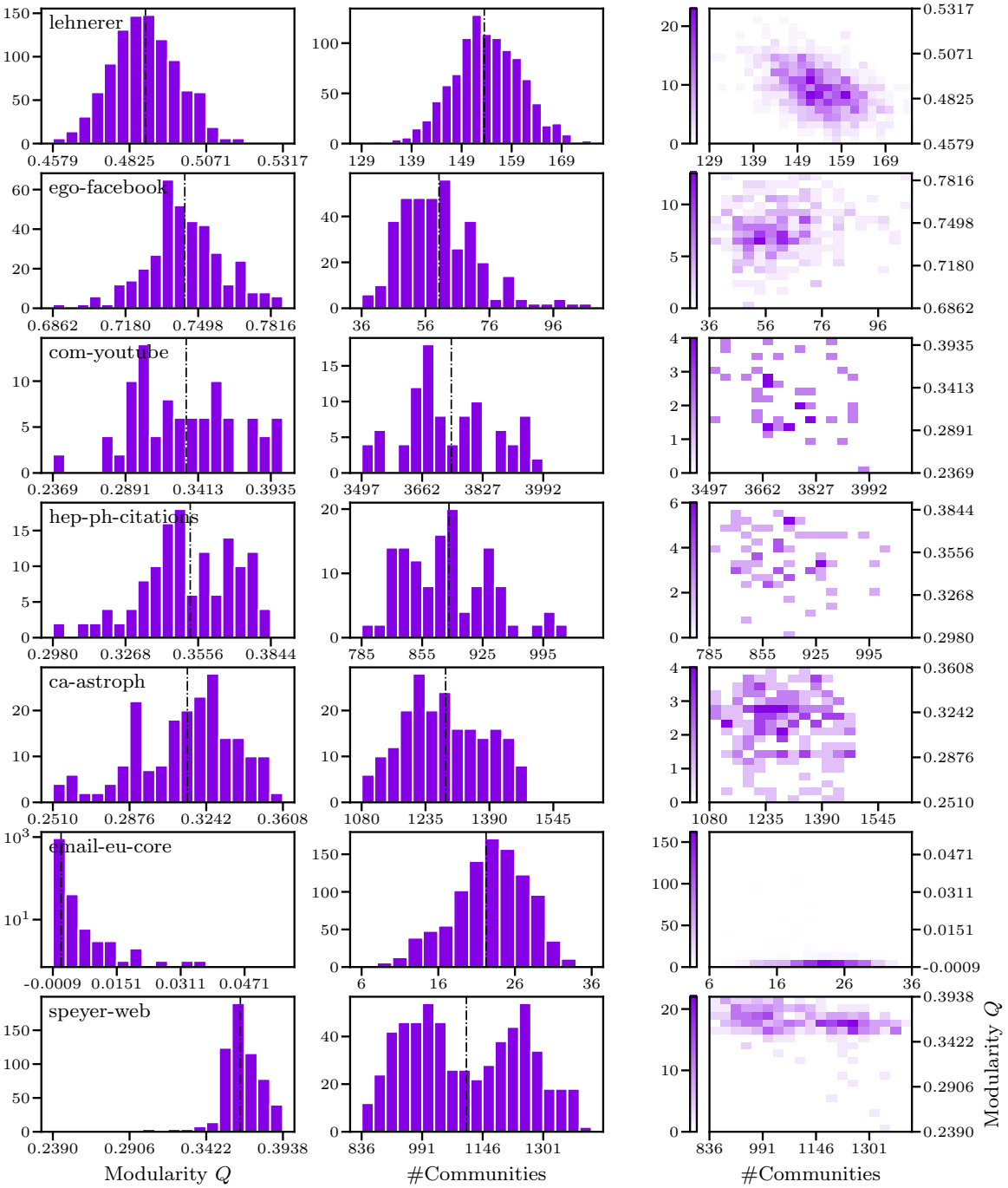


Fig. A.5.: Random-initialized Triangle modularity and number of communities for real-world datasets. Results after 5 generations.

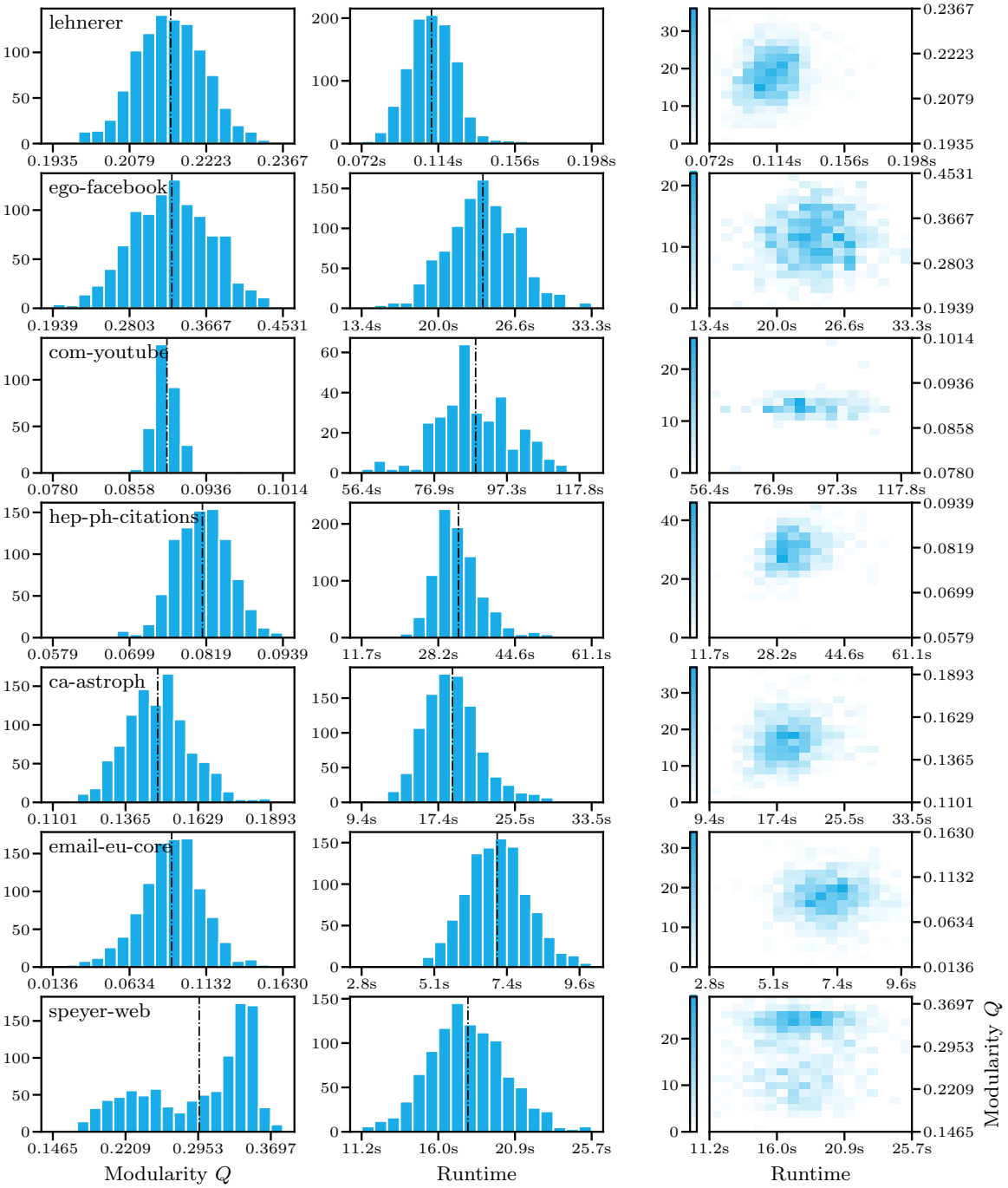


Fig. A.6.: Louvain-initialized Louvain CRATER modularity and runtime performance for real-world datasets. Results after 5 generations.

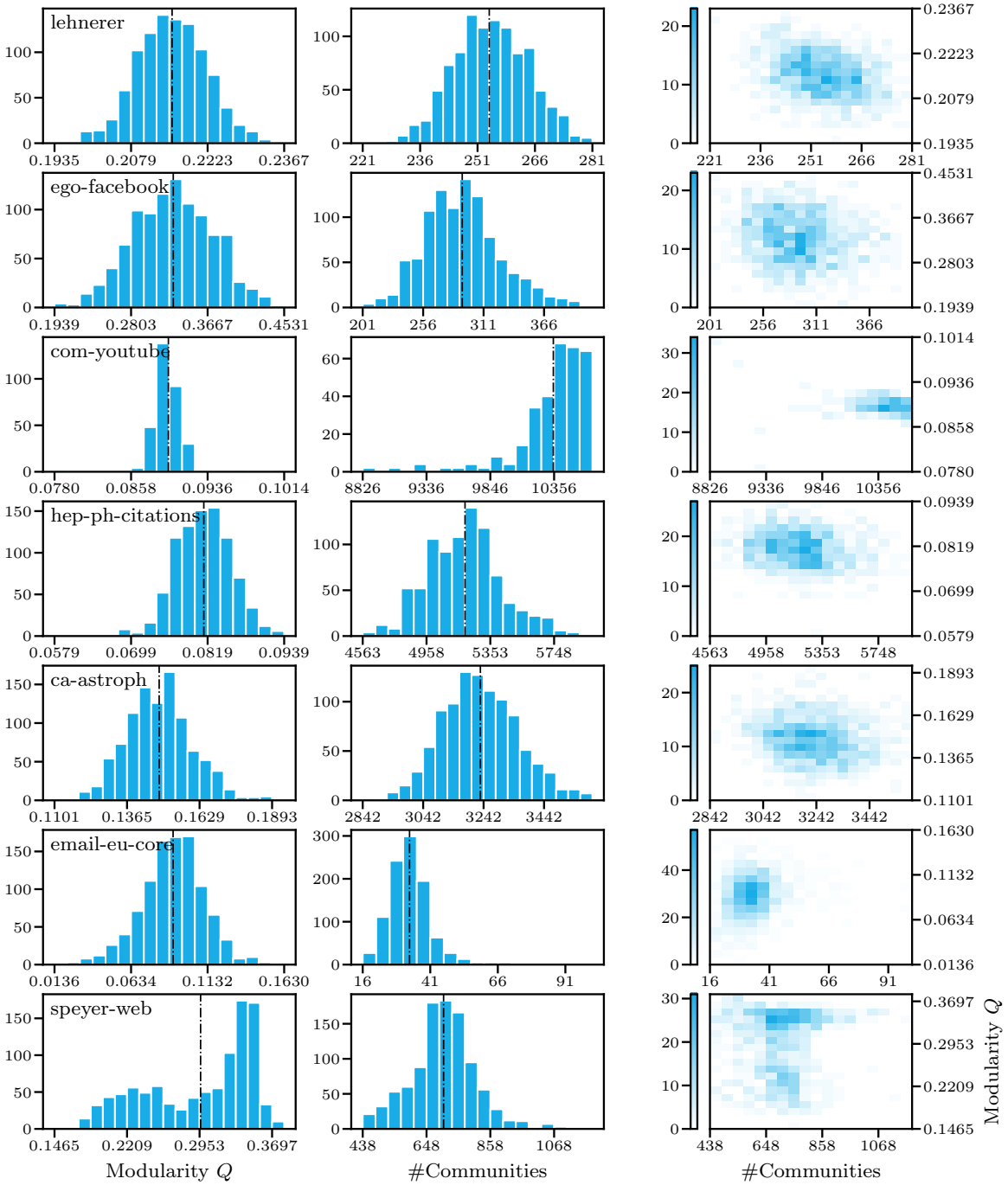


Fig. A.7.: Random-initialized Louvain CRATER modularity and number of communities for real-world datasets. Results after 5 generations.

A.3. Louvain-initialized mutations

The following diagrams are structured as described in section A.2. The reader is referred to section 4.2 and section 7.5 for background information. Due to a runtime limit, we could not finish 1000 runs for every method-dataset combination. We therefore list the number of runs we collected before the script terminated:

- LIIM: 1000 runs for all datasets
- Louvain-initialized Triangle: lehnerer 1000, ego-facebook 649, com-youtube 161, hep-ph-citations 234, ca-astroph 357, email-eu-core 1000, speyer-web 990
- Louvain-initialized Louvain CRATER: 1000 runs for all datasets other than only 499 for com-youtube

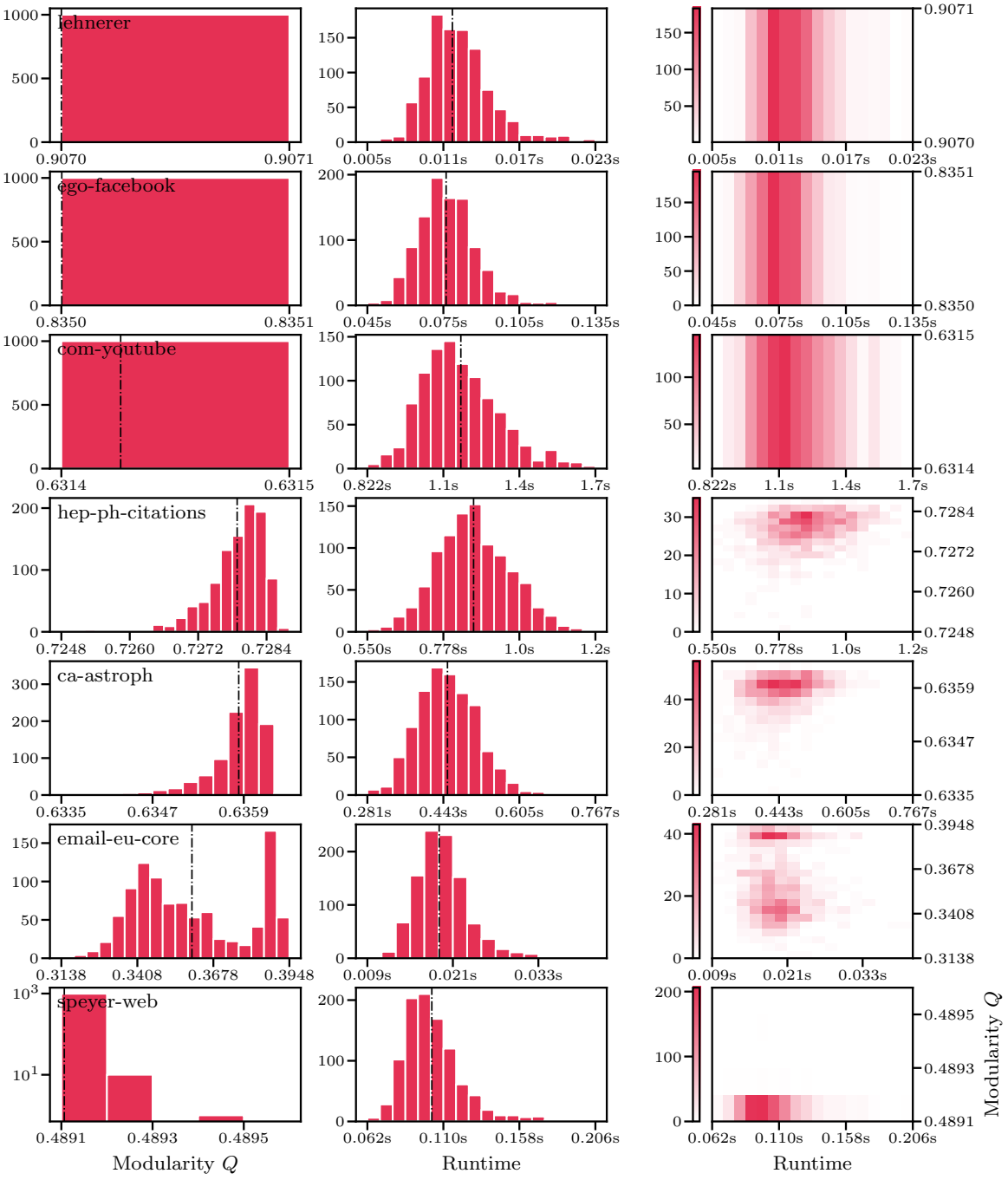


Fig. A.8.: LIIM modularity and runtime performance for real-world datasets. Results after 3 generations.

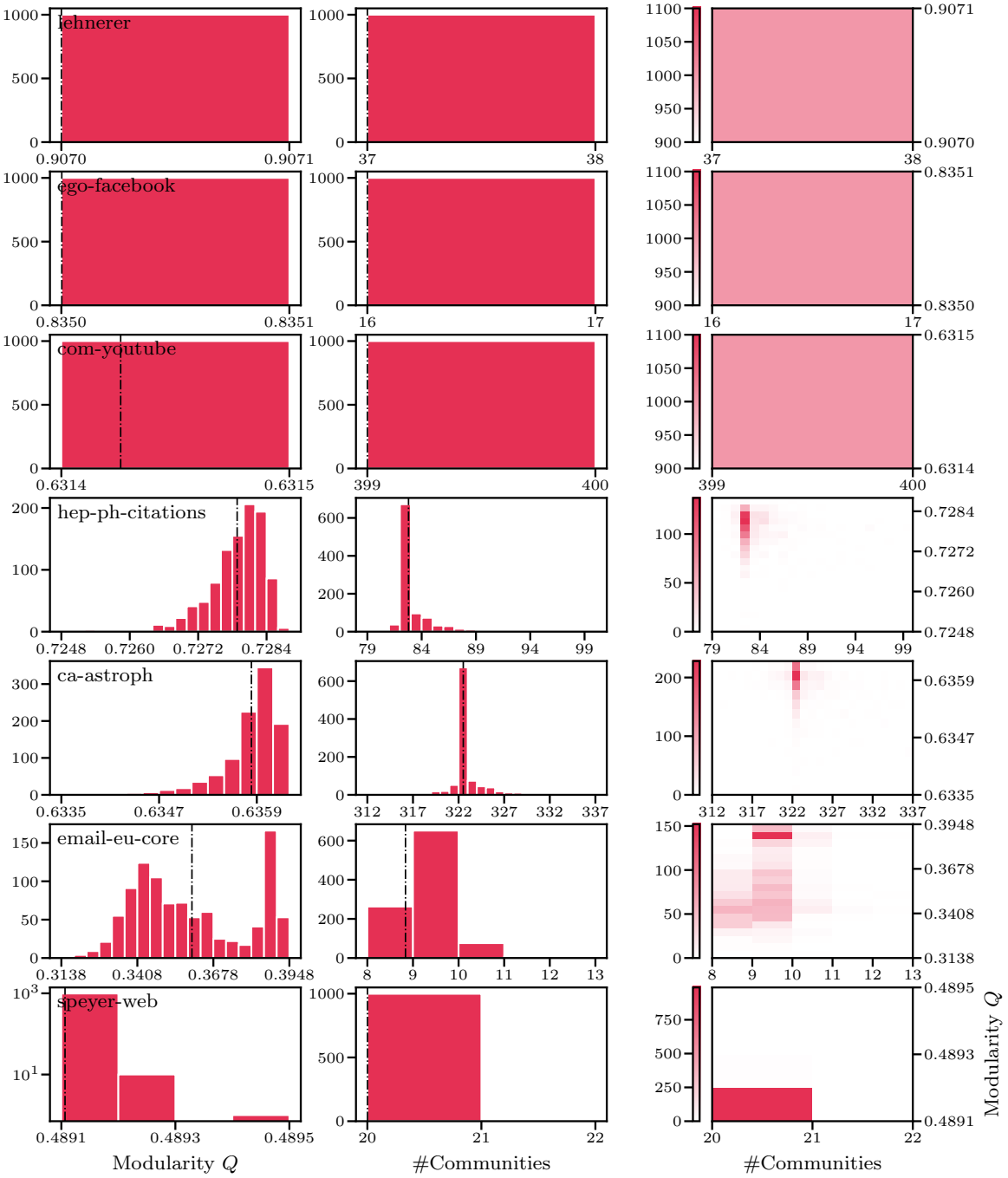


Fig. A.9.: LIIM modularity and number of communities for real-world datasets. Results after 3 generations.

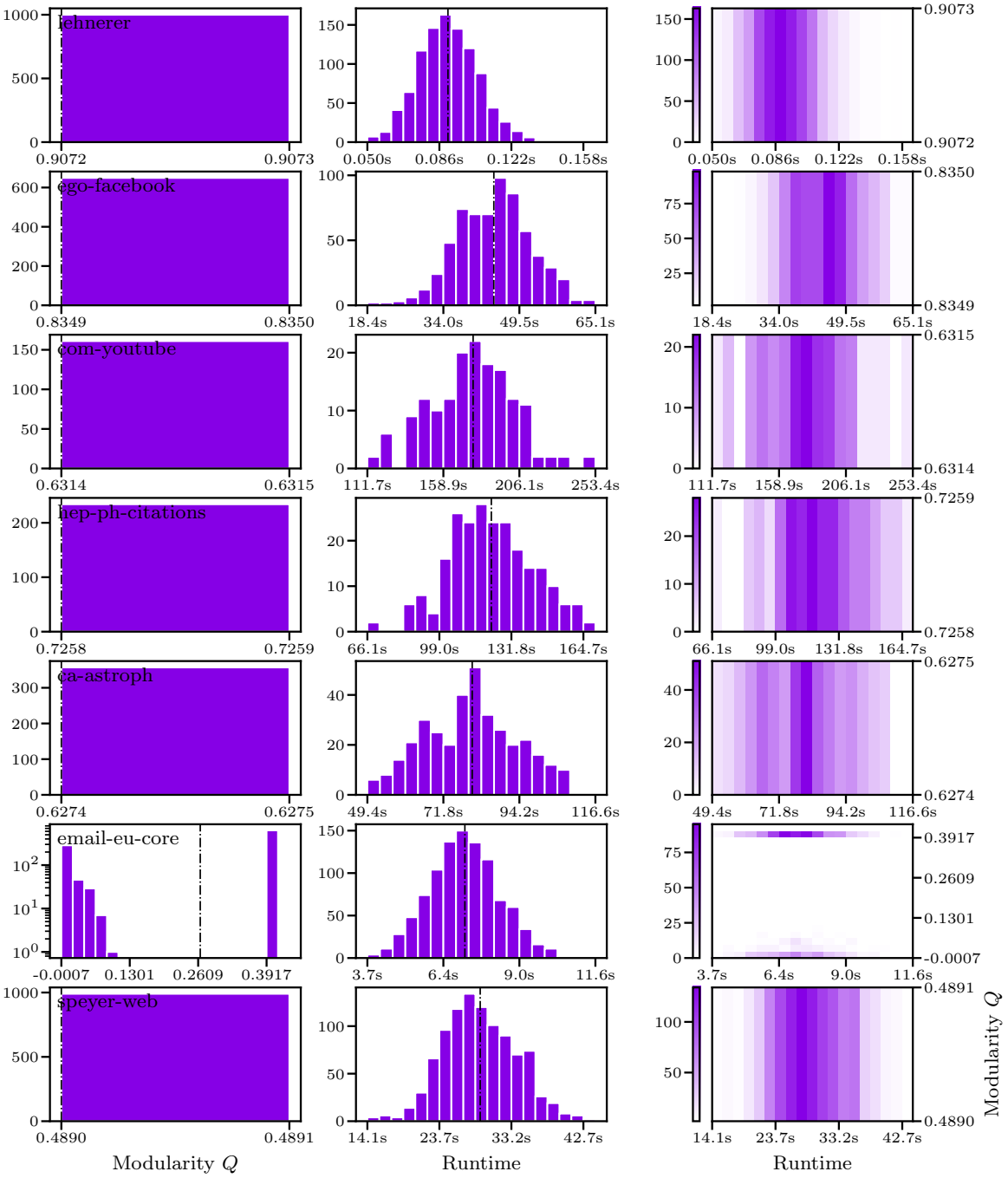


Fig. A.10.: Louvain-initialized Triangle modularity and runtime performance for real-world datasets. Results after 3 generations.

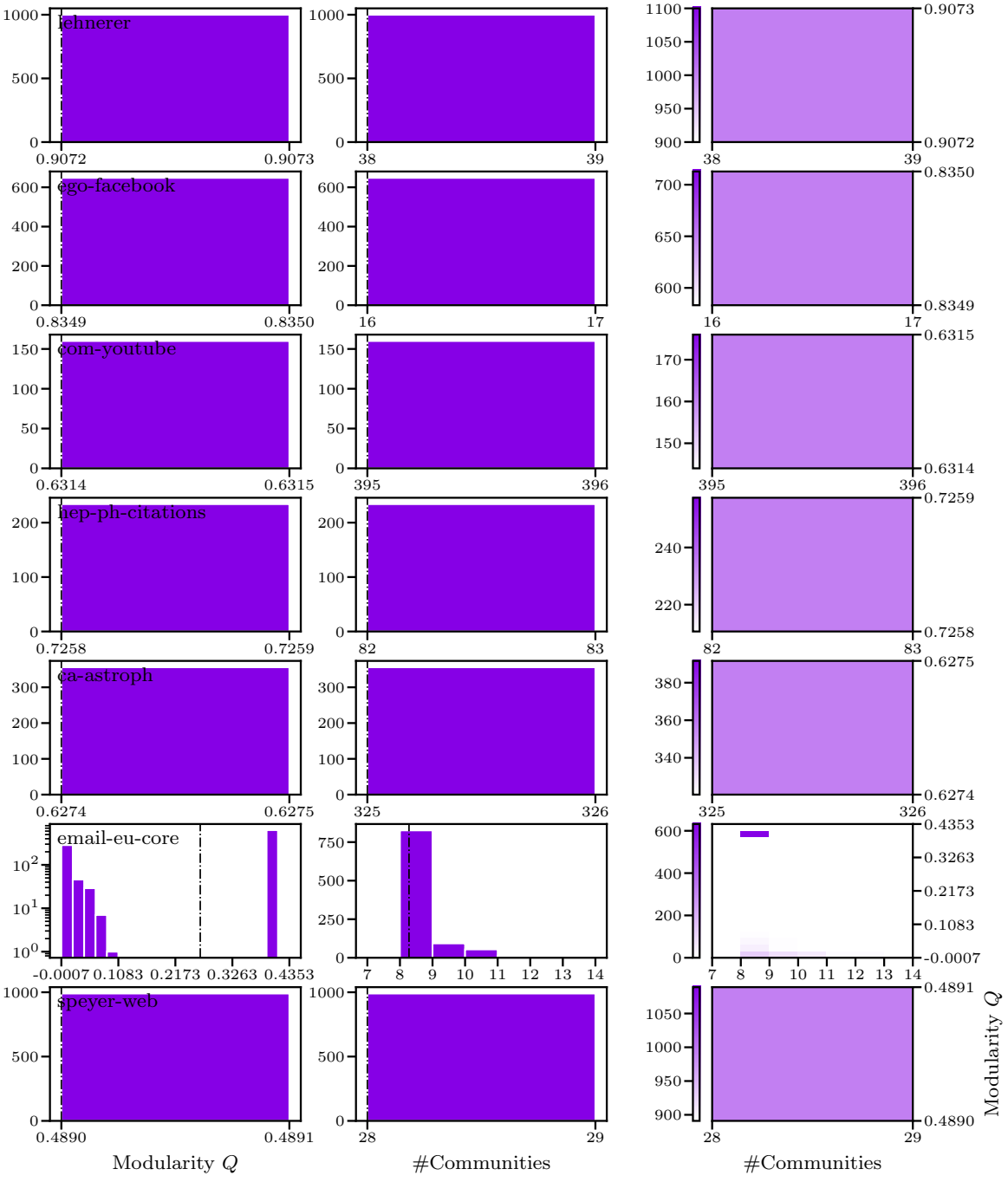


Fig. A.11.: [Louvain-initialized Triangle modularity and number of communities for real-world datasets. Results after 3 generations.

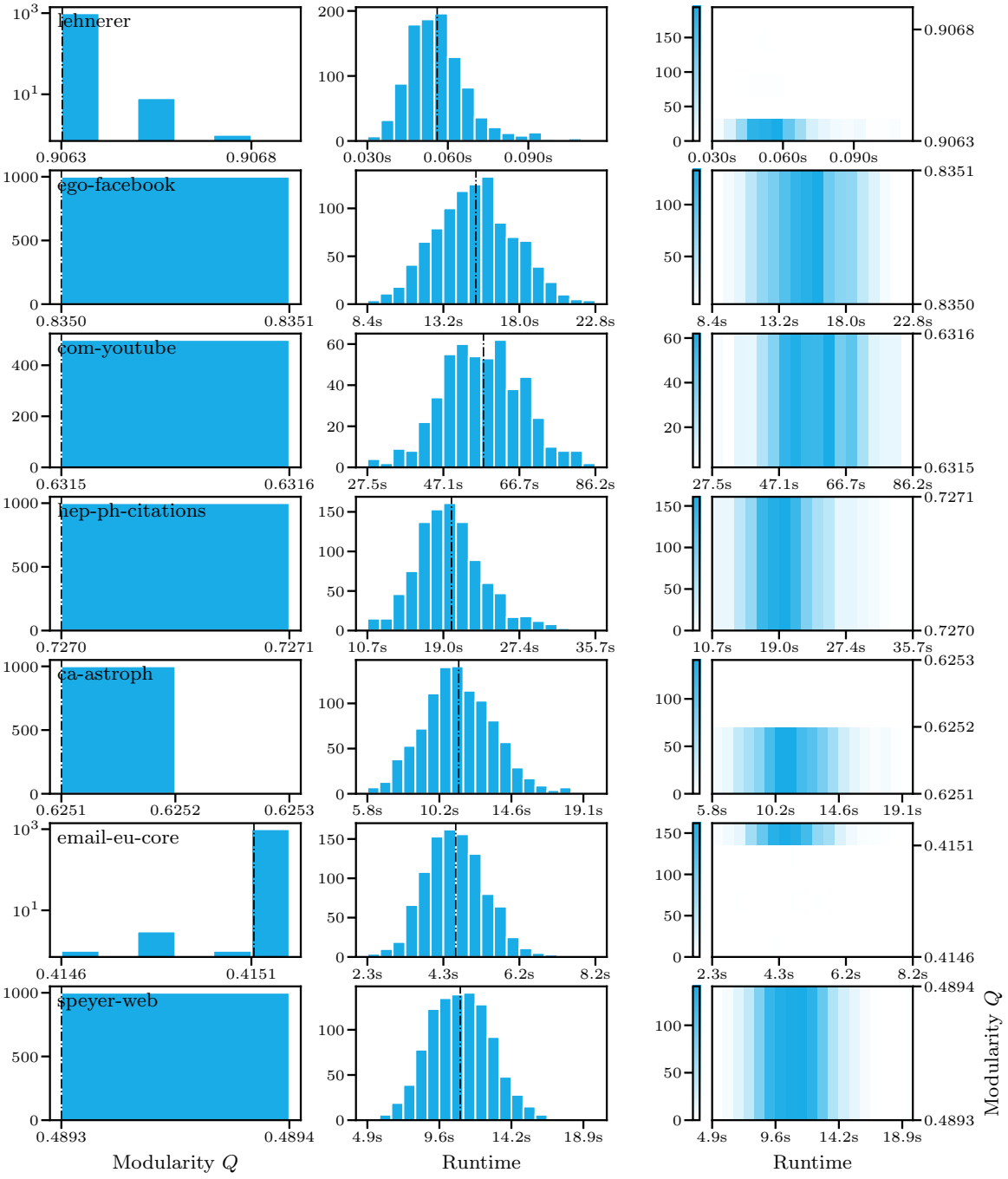


Fig. A.12.: Louvain-initialized Louvain CRATER modularity and runtime performance for real-world datasets. Results after 3 generations.

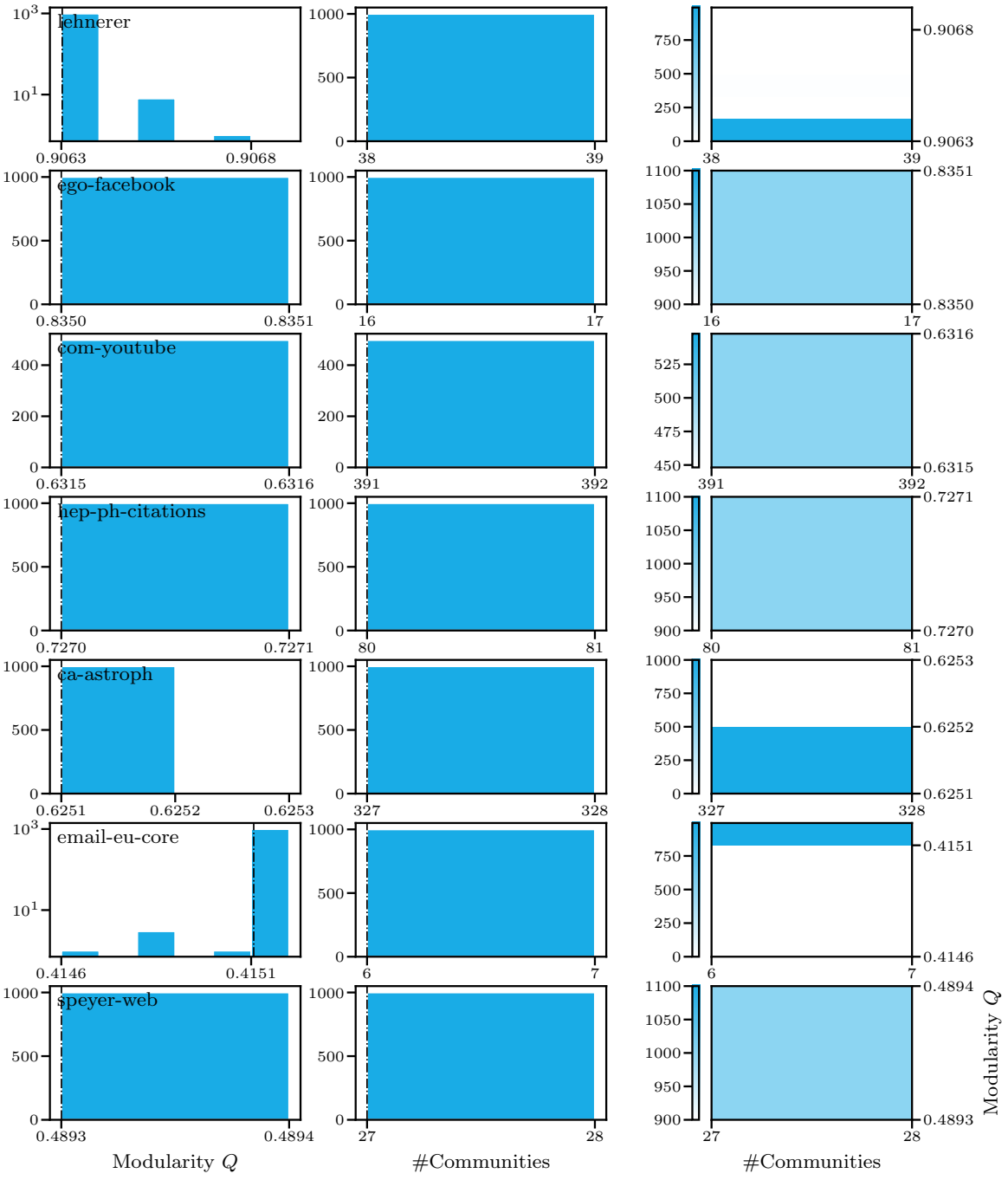


Fig. A.13.: Louvain-initialized Louvain CRATER modularity and number of communities for real-world datasets. Results after 3 generations.

A.4. Comparison of Louvain with Louvain-initialized mutations

In the following diagrams, the first column always depicts modularity histograms, the second column shows runtime histograms, and the last column presents histograms for the number of identified communities. The number of occurrences is plotted on the y-axis. Again, results of one dataset can be found in one row. We compare different strategies with Louvain, therefore always two methods are included in each cell of the grid-like figure. The reader is referred to section 4.2 and section 7.5 for further information regarding the experiments that led to these graphs.

Due to a runtime limit, we could not finish 1000 runs for every method-dataset combination. The respective number of runs that we collected before the script terminated are listed in the two previous sections. Louvain (results marked by black bars) always executed 1000 runs without exception.

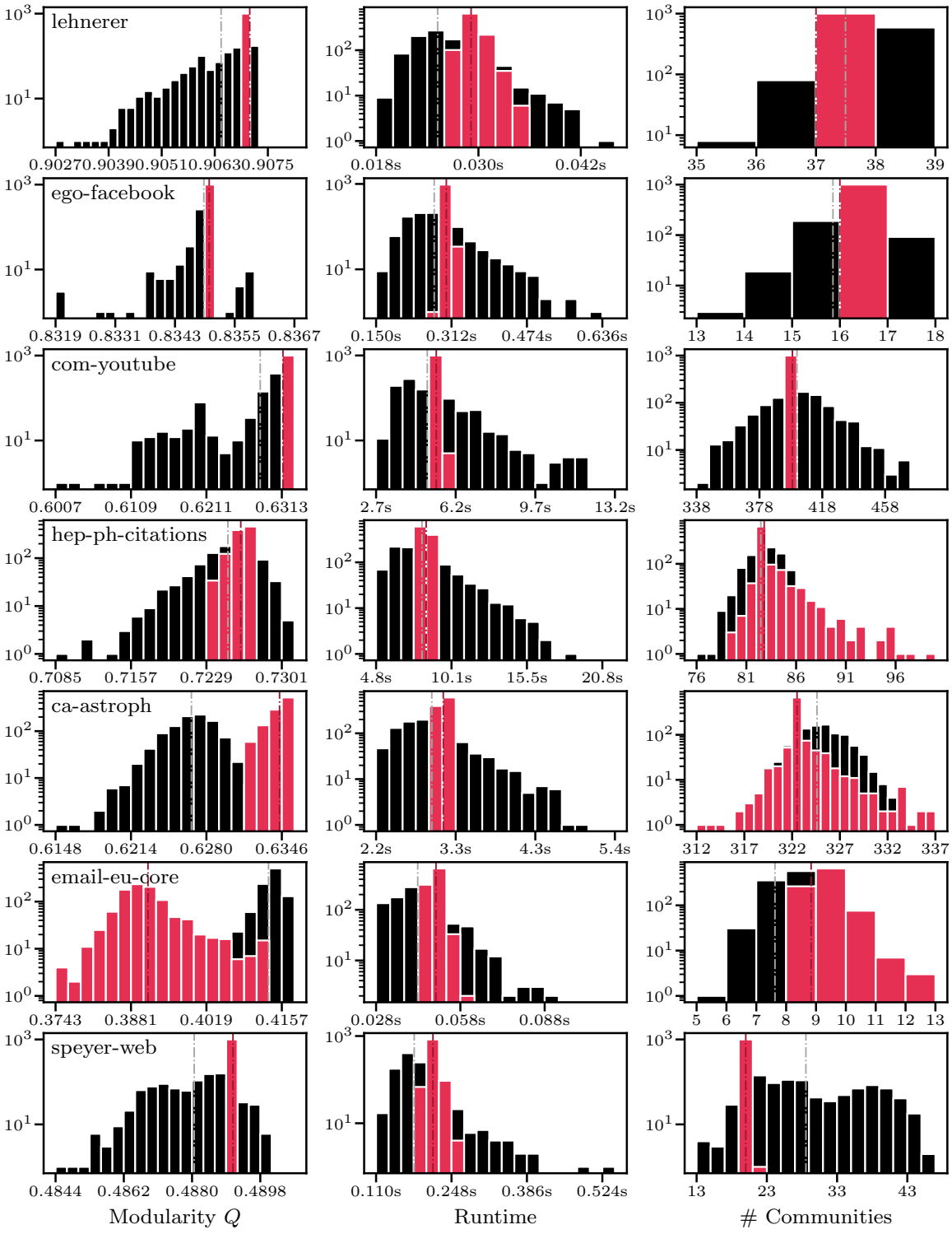


Fig. A.14.: Comparison of Louvain with Louvain-initialized Inherit on real-world datasets after 1 generation.

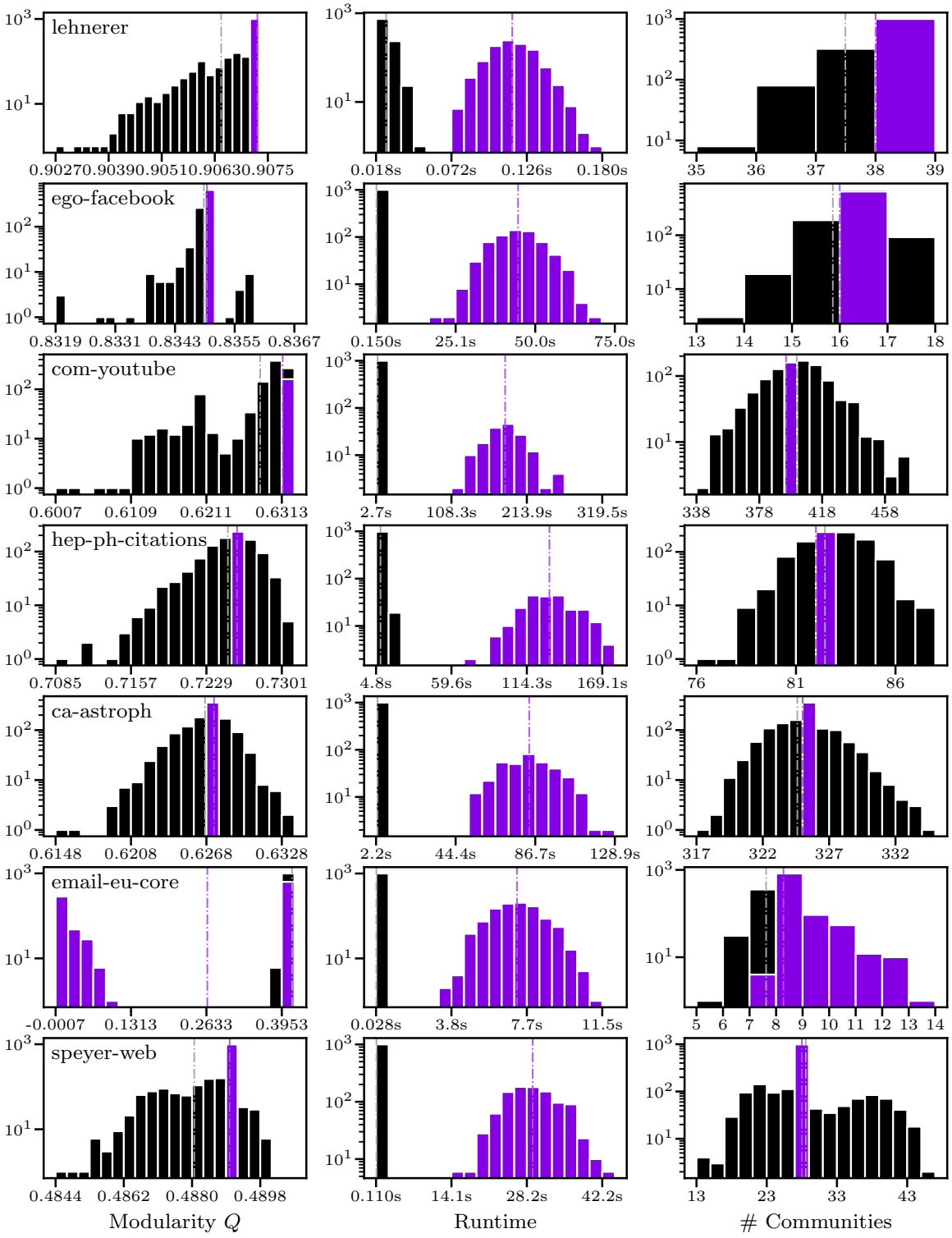


Fig. A.15.: Comparison of Louvain with Louvain-initialized Triangle on real-world datasets after 3 generations.

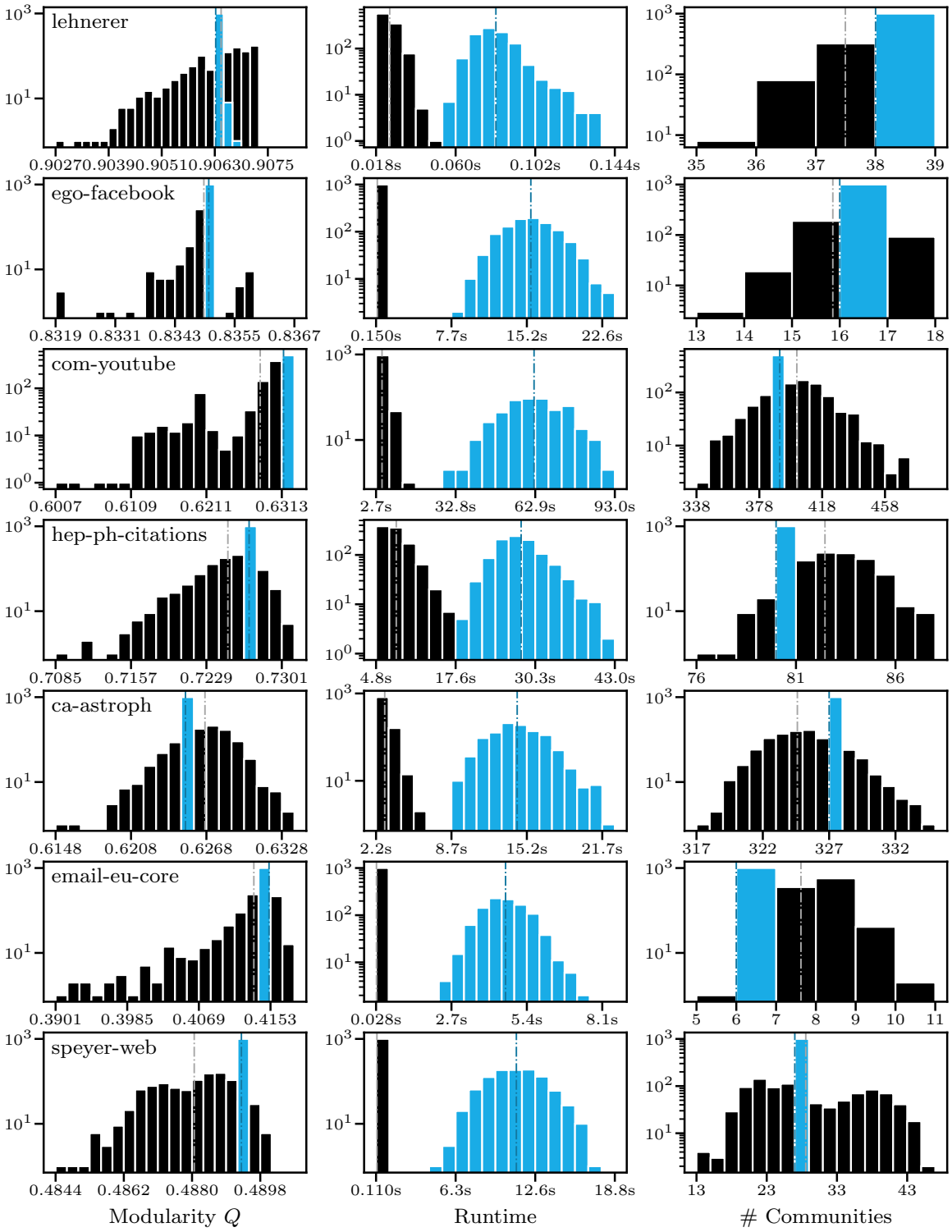


Fig. A.16.: Comparison of Louvain with Louvain CRATER on real-world datasets after 3 generations.