Chapter 2
Community Discovery: Simple and Scalable Approaches

Yiye Ruan, David Fuhry, Jiongqian Liang, Yu Wang and Srinivasan Parthasarathy

Abstract The increasing size and complexity of online social networks have brought distinct challenges to the task of community discovery. A community discovery algorithm needs to be *efficient*, not taking a prohibitive amount of time to finish. The algorithm should also be *scalable*, capable of handling large networks containing billions of edges or even more. Furthermore, a community discovery algorithm should be *effective* in that it produces community assignments of high quality. In this chapter, we present a selection of algorithms that follow simple design principles, and have proven highly effective and efficient according to extensive empirical evaluations. We start by discussing a generic approach of community discovery by combining multilevel graph contraction with core clustering algorithms. Next we describe the usage of network sampling in community discovery, where the goal is to reduce the number of nodes and/or edges while retaining the network's underlying community structure. Finally, we review research efforts that leverage various parallel and distributed computing paradigms in community discovery, which can facilitate finding communities in tera- and peta-scale networks.

2.1 Introduction

Community discovery has long served as an important primitive operator in the field of network science, and the ability of identifying user communities in social networks has lead to a plethora of applications including, among others, churn prediction [42], political analytics [1], and human behavior study [56]. While it is relatively easy to directly spot community structures embedded in the smallest networks, for other networks the task quickly becomes challenging for human beings. To illustrate, there

Y. Ruan (✉) · D. Fuhry · J. Liang · Y. Wang · S. Parthasarathy
Department of Computer Science and Engineering, The Ohio State University,
2015 Neil Avenue, 395 Dreese Lab, Columbus, OH 43210, USA
e-mail: ruan.17@buckeyemail.osu.edu

G. Paliouras et al. (eds.), *User Community Discovery*, Human–Computer Interaction Series, DOI 10.1007/978-3-319-23835-7_2

are 1.35 billion monthly active users on Facebook as of September, 2014.[1] If we are to plot all of those users on the computer screen, assuming each user only occupies one single pixel, we will need more than 1700 monitors of the typical 1024-by-768 resolution to just show all of them.

The rapidly-growing size of available user network data has multiple implications, all underlining the acute need for automatic community discovery algorithms that are efficient, scalable, and effective. First of all, algorithm complexity becomes a practical concern, as the improvement (or degradation) in running time is pronounced and easily perceivable. The difference between log-linear and quadratic complexities is now seconds versus years. Secondly, the RAM capacity of a single machine may be too low to fully store the underlying network in memory, let alone any auxiliary data required for the algorithm itself. Lastly, as network size grows, more noise is introduced inevitably, and this is especially true for online social networks where spammers and bots can create fake links with trivial cost, for instance. To produce results of high quality, a community discovery algorithm has to be robust and capable of differentiating between signal and noise.

In order to address these issues, many algorithms have been proposed in the literature of community discovery. In this chapter, we outline a collection of existing methods that have proven simple yet effective and scalable. Furthermore, we discuss some promising directions that deserve further investigation. In the next three sections, we focus on three respective categories:

- **Section 2.2: Multilevel community discovery methods**, which recursively *contract* a larger network into a smaller version on which a core community discovery algorithm will be run. The procedure of contraction (also known as coarsening) can follow different strategies, and they also differ on whether the core discovery algorithm is run only once or for multiple times. As a byproduct, the multilevel paradigm also produces a natural hierarchy of communities in the network.
- **Section 2.3: Sampling-based preprocessing algorithms**, which selectively keep a subset of vertices and/or edges to reduce the network size while attempting to preserve the community structure of the underlying network. The sampling process is typically guided by some quality measures, and can leverage additional information, such as vertex attributes or content.
- **Section 2.4: Parallel and distributed community discovery approaches**, which perform community identification in parallel to speed up the process.[2] Given the fact that many discovery algorithms are based on matrix/vector operations, and that many computation kernels specially designed for such operations have been developed in the parallel computing community, there is a great potential of performing community discovery in an efficient parallel environment. Furthermore, some of such algorithms also store data in a distributed fashion (e.g. HDFS), making it possible to overcome the capacity limit of one single machine.

[1]http://newsroom.fb.com/company-info/. Accessed in December 2014.

[2]Here, we will discuss methods based on both shared-memory and distributed-memory architectures.

Table 2.1 Table of notations used in this chapter

$G(V, E)$	An undirected graph with the vertex set V and edge set E.
Γ_u	The neighborhood of a node u, that is, $(u, v) \in E, \forall v \in \Gamma_u$ and $\nexists v \in V - \Gamma_u$ such that $(u, v) \in E$.
Γ_S	The neighborhood of a set of nodes S, that is, $\bigcup_{u \in S} \Gamma_u$.
$A \in \{0, 1\}^{\lvert V\rvert \times \lvert V\rvert}$	The symmetric adjacency matrix of G, i.e. $A(u, v) = A(v, u) = 1$ if $(u, v) \in E$, and 0 if $(u, v) \notin E$.
$D \in \mathbb{N}^{\lvert V\rvert \times \lvert V\rvert}$	The diagonal matrix of node degrees, i.e. $D(u, u) = \lvert\Gamma_u\rvert$ and $D(u, v) = 0, \forall u \neq v$.
N_c	The number of communities to detect.

Among existing literature, the definition of "community" itself is still open-ended and highly context-dependent, leading to various input and output specifications across different methods. Here we summarize the premise of the algorithms described in this chapter. All these algorithms can operate on unweighted, undirected networks, and some readily accept weighted/directed networks as inputs, or can be easily adapted to do so. With the exception of Louvain algorithm in Sect. 2.2, all algorithms take the desired number of communities as an additional input, either directly or indirectly (via other control parameters). In terms of output, communities generated by those algorithms are disjoint, in that no two communities will overlap with each other. Few algorithms, such as METIS (Sect. 2.2), produce communities of equal size, while most allow variation in community size. With all the distinct properties of each method in mind, one key observation from this chapter is: *Most approaches introduced here can also be viewed as meta-algorithms, and one can reuse and enhance many existing community discovery algorithms by plugging them into these frameworks.*

Before proceeding, we note that this chapter is not intended as an exhaustive survey of community discovery algorithms, and we have provided references for further reading in Sect. 2.5. Rather, we aim at simple and scalable approaches that fall into one of the three categories mentioned above. Apart from technical descriptions, we also hope to provide a systematic view of these three generic design principles, and to inspire new algorithms that follow them. Moreover, it is feasible for one to devise methods that leverage more than one of these approaches simultaneously, as they are complementary to each other.

Notations in Table 2.1 will be used throughout this chapter. Note that, we will use the terms "node" and "vertex" interchangeably, as well as "network" and "graph".

2.2 Multilevel Approach for Community Discovery

We begin with the introduction of multilevel approach, an effective scheme that has been applied in conjunction with different community detection algorithms. The key motivation behind all multilevel algorithms is to efficiently obtain a partitioning of nodes at a coarse-grained level, and then recover fine-grained communities from high-level clusters in a recursive manner. A key point to the efficacy of the multilevel paradigm is, therefore, retaining the graph's backbone structure when reducing its size, and we will discuss several principled and heuristic-driven approaches to achieve this requirement.

Aside from being efficient, the multilevel scheme also brings the benefit of flexibility since it is largely orthogonal to the underlying community detection algorithms. This enables practitioners to experiment with various community detection "kernels" with minimal change in the implementation. Moreover, multilevel approaches naturally produce a hierarchy of communities, making it easy to explore the community structure of a graph at different scales.

Here, we will showcase concrete implementations of the multilevel approach by presenting four representative algorithms: METIS, MLR-MCL, Graclus and Louvain. There are three main phases in all multilevel community discovery algorithms:

- **Contraction (Sect. 2.2.2)**: From the original network, a series of networks of decreasing size are generated. Each small network is created by contracting multiple nodes in the parent-level network into a multinode, and their edges to other nodes in the network are retained. The selections of nodes to contract and their ordering can be decided by various strategies, as we shall see below.
- **Partitioning (Sect. 2.2.3)**: This refers to running a core partitioning algorithm on the aforementioned contracted networks. The key motivation is that such an algorithm will run more efficiently on the contracted networks because their sizes are smaller. While some approaches only call the core algorithm on the smallest contracted network, others execute the core algorithm on the contracted network at each level.
- **Refinement (Sect. 2.2.4)**: To obtain community assignments on the original network, partitions on a smaller network will be projected back to the higher level by decomposing multinodes. A fast refinement procedure often follows the projection, so that the community assignments are further improved. Typically, the refinement subroutine is lightweight and only performs simple local operations. In some cases such as MLR-MCL and Graclus, however, refinement can involve the core partitioning algorithm itself, too.

2.2.1 Overview

METIS METIS [19] is a graph clustering algorithm for undirected graphs with optionally both edge weights and vertex weights. It recursively contracts the graph and partitions the smallest graph. Then the partitioning is projected to the original graph. METIS performs recursive contraction, recursive partitioning and recursive projection separately.

MLR-MCL Multilevel Regularized Markov Clustering (MLR-MCL) is another multilevel approach for graph clustering, introduced in [45]. Unlike METIS, MLR-MCL is based on Markov Clustering [49], which is a flow-based graph clustering algorithm. MCL follows the nature of flow and transition probability in graphs to conduct clustering and is able to generate clustering results in different granularities by tuning parameters. MLR-MCL goes further to improve clustering quality and the scalability of the algorithm through additional regularization and multilevel mechanism.

Graclus Graclus [9] is quite similar to MLR-MCL, except it utilizes weighted kernel k-means in the partitioning and refinement phases. Many variants of weighted kernel k-means can be derived depending on different categories of objective functions, e.g. Kernighan-Lin objective. The multilevel mechanism guarantees a desirable initial clustering at each refinement level, making the refinement step converge very fast.

Louvain Louvain [3] also adopts the multilevel framework but aims at optimizing modularity, which is a commonly-used criterion to evaluate community detection [32]. Louvain recursively optimizes modularity at each level and deals with graphs with edge weights. It alternates between contraction and partitioning while the refinement phase is mostly trivial.

2.2.2 Contraction

A series of graphs, called contracted graphs, are generated in this phase. We denote them as $G_0, G_1, \ldots, G_m$, where $G_0 = G$ is the original graph. Each $G_{i+1} = (V_{i+1}, E_{i+1})$ is obtained by contracting $G_i = (V_i, E_i)$, and hence is smaller than G_i. Correspondingly, G_i is referred to as the parent graph of G_{i+1}. If a vertex $v \in V_{i+1}$ corresponds to a group of nodes in V_i, we call v a multinode for G_i. There are two types of contraction: edge contraction and vertex contraction [39]. The former is to contract a pair of adjacent vertices into one multinode, whereas the latter is to contract a vertex and its neighbors into one. In general, vertex contraction shrinks the graph more significantly than edge contraction because the latter shrinks the graph at most by half each time.

METIS, MLR-MCL, and Graclus In the contraction phase, METIS, MLR-MCL and Graclus all follow the same procedure. Edge contraction can be done by finding a maximal matching [39]. When a maximal matching is identified, the graph is contracted in this way: Replace a matched node pair by a multinode whose weight is the sum of those of the two absorbed nodes; If parallel edges are created between the multinode and other nodes, they are replaced by one single edge whose weight is the sum of those of all parallel edges. G_{i+1} is constructed when the whole matching in G_i are replaced. An heuristic called *Heavy Edge Matching* is used to construct the maximal matching. The idea is to maximize the difference of edge weights between two consecutive contracted graphs. The motivation is to minimize the edge weight of the contracted graph and thus the upper bound of edge cut, a commonly-used partitioning criterion (Sect. 2.2.3), in the contracted graph. The contraction process recursively contracts the graph until $|V_i|/|V_{i+1}|$ is smaller than a predefined threshold, that is, the decrease in graph size is no longer significant. An alternative strategy is to terminate the process when $|V_i|$ itself, instead of the ratio, is below a threshold.

Louvain Louvain algorithm alternates between contraction and partitioning. The contraction phase itself is simple: Contract all nodes in the same community (from the parent graph) into a single multinode, and replace each group of parallel edges with one single edge of the aggregated weights. When the contraction phase ends, a new graph with fewer nodes, each of which representing a community, is constructed.

2.2.3 Partitioning

Partitioning is to partition the graph into multiple communities such that vertices within the same community are more densely connected than vertices across communities. The partitioning algorithms of all four methods here produce disjoint communities.[3] Partitioning of METIS only occurs on the coarsest graph, whereas MLR-MCL and Graclus perform partitioning in the whole process of refinement from G_m to G_0 incrementally. On the other hand, Louvain performs partitioning in the process of contraction.

METIS METIS tries to find balanced communities, i.e. communities have similar number of vertices in an unweighted graph while have similar sum of weights in a weighted graph. A bisection procedure [15, 16] is explained here while a more sophisticated k-way partition can be found in [19]. The bisection procedure works to recursively partition the graph/sub-graph into two parts until a desired number of communities are obtained. METIS can be coupled with three bisection algorithms, one using spectral bisection while the others adapting a strategy of minimizing edge-cut.

Spectral Bisection Fiedler vector [13, 36] is used for partitioning the graph. The Fiedler vector f of a network is the eigenvector corresponding to the second smallest eigenvalue of its Laplacian matrix. After computing f, let $f(v)$ be the vth element of f, the graph is bisected by a threshold t, which minimizes $|\sum_{f(v)\leq t} vw(v) - \sum_{f(v)>t} vw(v)|$, where $vw(v)$ stands for the weight of vertex v.

KL Algorithm The Kernighan-Lin (KL) algorithm [12, 21] is an iterative algorithm starting from an initial bipartition. Nodes are ranked based on their potential of edge-cut reduction. After selecting proper nodes, edge-cut is decreased in each iteration by swapping two subsets of the nodes from the two parts. KL algorithm relies on good initial partition [5]. If a good initial partition is not available, one can run KL algorithm on multiple random initial partitions and select the one with the smallest edge-cut.

[3] Overlapping community detection has also attracted considerable research attention [51, 53], yet existing studies have not adapted the multilevel framework discussed here. Combining the multilevel paradigm with overlapping community discovery will be an exciting future direction.

An easier implementation is to move one vertex at a time rather than swapping two subsets. The iteration terminates when no vertex can be moved or edge-cut decreases little in some consecutive moves.

Greedy Graph Growing Partitioning Algorithm This algorithm performs heuristic-guided breadth-first search from a vertex until half of the graph, in terms of number of vertices (for unweighted graphs) or the sum of vertex weights (for weighted graphs), are included. It works by growing a community from an individual vertex. The heuristic is to order the vertices on the frontier of the growing community based on their potential to decrease edge-cut and move the vertex with the highest ranking. The frontier and the ranking should be maintained dynamically. Reference [19] shows that this algorithm requires the same data structure as KL algorithm. Reaching the desired number of communities is a termination condition for this algorithm.

MLR-MCL The partitioning phase of MLR-MCL follows the Regularized Markov Clustering (R-MCL). R-MCL itself is a clustering algorithm based on stochastic matrices and flows. In graph G, let A be the adjacency matrix, where $A(i,j)$ is the weight of the edge between vertex v_i and vertex v_j. Then a column stochastic matrix M_G is the matrix of the transition probabilities of a random walk in the graph. Specifically, $M_G(j,i)$ represents the probability of transiting from vertex v_i to v_j. The column-stochastic transition matrix M_G is usually derived by normalizing the columns of the adjacency matrix A such that each column sums up to 1, i.e. $M_G(i,j) = \frac{A(i,j)}{\sum_{k=1}^{n} A(k,j)}$. This is usually obtained from $M_G = AD^{-1}$, where D is the diagonal degree matrix of graph G and $D(i,i) = \sum_{j=1}^{n} A(j,i)$. We call this matrix the *canonical transition matrix*.

R-MCL, similar to MCL [49], follows three steps. The first step is expansion, which allows flow to go to different parts of the graph. It can be done by using $M_{exp} = Expand(M) = M * M_G$, where M is the flow distribution matrix from the previous iteration (initially it is M_G). The second step is inflation, which both strengthens what is strong and weakens what is weak of the flow by using $M_{inf}(i,j) = \frac{M_{exp}(i,j)^r}{\sum_{k=1}^{n} M_{exp}(k,j)^r}$. The purpose of inflation is to expedite the convergence of partitioning. As long as $r > 1$, this operator will exaggerate the imbalanced distribution of each column of M_{exp}. A higher r means a more aggressive inflation and by default, it is set as 2. The third steps is pruning, which aims to eliminate those very small values in M_{inf}. Pruning will make the matrix sparser and help save memory and computation. The threshold used to prune the small values can be computed based on the average and maximum in each column (see [49] for more details) and then all the entries below the threshold will be pruned.

At the partitioning phase for a contracted graph G_i, the canonical transition matrix is built, and the expansion-inflation-pruning cycle is performed for a small number of iterations. The corresponding flow distribution M_{inf} will be projected back to the finer graph (G_{i-1}) and be treated as the initial flow distribution for R-MCL on G_{i-1}. Details of the refinement process will be discussed in the next subsection.

Graclus For Graclus, we apply a base clustering algorithm on the coarsest graph. We can use the region-growing algorithm of METIS, which is usually very efficient. Spectral clustering and bisection methods are other alternative approaches for base clustering.

However, from G_{m-1} to G_0, Graclus adopts weighted kernel k-means to perform partitioning. Weighted kernel k-means is a variant of k-means which adopts a non-linear mapping kernel function and adds a weight to each cluster. With certain objective functions (either based on association or cuts), we can use weighted kernel k-means for graph clustering. One example will be clustering a graph based on association, where we aim to partition the graph into k disjoint parts $\pi_1, \pi_2, \ldots, \pi_k$, whose union is V, such that $WAssoc(G) = \sum_{c=1}^{k} \frac{links(\pi_c, \pi_c)}{w(\pi_c)}$ is maximized. Here, $links(\pi_c, \pi_c)$ is the sum of the edge weights inside the cluster π_c and $w(\pi_c)$ is the sum of node weights inside π_c.

As [9] proved, the trace maximization formula of objectives in weighted kernel k-means and graph clustering are actually equivalent. By setting kernel matrix K as $W^{-1}AW^{-1}$, where W is the diagonal matrix of weights and A is adjacency matrix, we can run weighted kernel k-means for graph clustering without calculating its eigenvectors. In other words, graph clustering can be done by using a simple iterative algorithm.

Similar to MLR-MCL, we do not directly run weighted kernel k-means on the original graph. Instead, we run this clustering algorithm from G_{m-1} to G_0. The partitioning result from G_i is used as the initial community assignment of G_{i-1} after projection. More details of refinement will be introduced in next subsection.

Louvain Partitioning and contraction are alternately performed in the same recursion in Louvain algorithm. Each node of the current graph is treated as a community. For each node v_i, the algorithm considers each neighbor v_j of v_i and evaluates the gain of modularity by moving v_i from its community to the community of v_j. A node v_i will be moved only if there is positive gain, and it is moved to the community that yields the maximum modularity gain. Otherwise, v_i remains in the original community. This process is applied repeatedly and sequentially for all nodes until reaching a Pareto frontier, i.e. no further improvement can be achieved. Then the partitioning phase terminates. It is shown that the order of node selection may affect computation time but not the convergence of modularity.

2.2.4 Refinement

Since our goal is to perform community detection on the original graph, we need to refine the partitioning results starting from G_m to G_0. Given the communities of multinodes on the contracted graph, we need to project the communities back to the parent graph. We define P_i to be a partition on graph G_i projected from P_{i+1}. For METIS, the main task of refinement phase is projection, while MLR-MCL and Graclus also need to run partitioning techniques on the projected result. The refinement phase runs from G_m to G_0 and the final community detection results are obtained when it reaches the original graph.

METIS Let V_i^v be the set of nodes in G_i that are contracted to the node v in G_{i+1}. The recursion of P_i is defined as $P_i(u) = P_{i+1}(v), \forall u \in V_i^v$. Since uncoarsening makes graphs finer, the local optimum P_{i+1} of G_{i+1} does not necessarily lead to a local optimum of G_i. Thus, it is possible for refinement to reach a local optimum on a finer graph. Recall that the KL algorithm starts with an initial partitioning and iteratively reduces edge-cut by swapping subsets. Similar ideas work in refining and uncoarsening the graph. The strategy derived from the KL algorithm is discussed below.

Boundary KL Refinement This strategy takes P_i, which is projected directly from the partitioning on G_{i+1}, as an initial partitioning and applies KL algorithm on G_i to obtain a locally optimal partitioning. One difference from the KL algorithm is that only the boundary nodes are included in the ranking to reduce redundant computation. This refinement generates a high quality partitioning due to the high quality of the initial partitioning. It is also efficient since few nodes need to be swapped given the initial partitioning is close to a local optimum. By considering the marginal decrease of edge-cut, [19] proposes a termination condition: stop after the first iteration, which reduces time cost by a factor of two to four.

MLR-MCL The refinement phase of MLR-MCL starts from the coarsest graph to the original one, i.e. $G_m, \ldots, G_2, G_1$. In each refinement from G_i to G_{i-1}, R-MCL is first run for 4 to 5 iterations. Then the flow from the coarser graph G_i is projected onto the refined graph G_{i-1}. Given the flow distribution of a node in the coarser graph, the flow projection can be done by choosing one of its constituent nodes (in the larger graph) and assigning all the flow into it to the chosen constituent node. After the flow projection, we get a new transition matrix M and can move on to next refinement from G_{i-1} to G_{i-2}. We keep doing this until we reach the original graph.

When we reach the original graph after refinement, R-MCL is performed until convergence. Finally, we interpret the transition matrix M we obtain to figure out the clustering result.

Graclus The refinement of Graclus is quite similar to MLR-MCL. Starting from the coarsest graph G_m, we perform the refinement until we reach the original graph G_0 (namely G). Suppose we are refining the graph from G_i to G_{i-1}. We get the initial clustering for G_{i-1} from G_i by simply following this: nodes in the same cluster on G_i cause their constituent nodes on G_{i-1} to be in the same cluster. We then improve this initial clustering by using weighted kernel k-means. Given the adjacency matrix of the current graph and the objective function for graph clustering, we can set the weights and kernel matrix and run weighted kernel k-means. Note the initial clustering above is usually quite desirable so the k-means can converge very fast. Eventually, we run weighted kernel k-means on the original graph, which generates the final result of communities. Besides, weighted kernel k-means can be greatly sped up by focusing only on the boundary nodes, i.e. nodes that contain an edge to a node in another cluster. This optimization can bring much efficiency with little loss in cluster quality.

Table 2.2 Characteristics of four multilevel community discovery algorithms

	Require # of communities as input?	Produce communities of equal size?	Core partitioning algorithm	Partitioning algorithm usage
METIS	Yes	Yes	Kernighan-Lin algorithm	Only on the most contracted network
MLR-MCL	Yes	No	Markov clustering	On contracted network at each level, during refinement
Graclus	Yes	No	Weighted kernel k-means	On contracted network at each level, during refinement
Louvain	No	No	Modularity optimization	On contracted network at each level, during contraction

Louvain Because partitioning has already been performed at each level during the contraction phase, the Louvain method has a trivial projection phase: just release all the original nodes from the multinode in the smallest contracted graph and assign nodes absorbed into the same multinode the same community label.

2.2.5 Empirical Results and Summary

In this section, we have introduced four community discovery algorithms that share the same pattern of following a three-phase workflow: contraction, partitioning, and refinement. These methods differ in their core partitioning methods, as well as in the stage at which their partitioning subroutine is invoked. Table 2.2 outlines several key characteristics that make each algorithm distinct from others.

In the literature where these methods were originally proposed, it is typically found that the multilevel paradigm significantly speeds up the computation, often leading to implementations that are two to three orders faster than their counterparts without the multilevel approach. However, few studies exist that benchmark these algorithms and directly compare their performances. To provide a more comprehensive picture on the effectiveness and efficiency of these algorithms, we perform a series of experiments on multiple real-world networks. Specifically, we downloaded six distinct networks from the Stanford large network dataset repository,[4] including both social networks and web networks. Network sizes are listed in the first two

[4]http://snap.stanford.edu/data/index.html.

Table 2.3 Information on network size, and F-1 scores of communities identified by four multilevel algorithms

	$\vert V\vert$	$\vert E\vert$	METIS	MLR-MCL	Graclus	Louvain
Facebook	4039	88234	0.2356	0.2701	0.3026	**0.3868**
Twitter	81306	1342303	0.1628	0.1146	**0.2147**	0.1086
Google+	107614	12238285	0.1664	0.0100	**0.1789**	0.0549
Youtube	1134890	2987624	**0.0441**	0.0068	N/A	0.0100
LiveJournal	3997962	34681189	**0.1864**	0.1497	N/A	0.1527
Amazon	334863	925872	0.4465	**0.5001**	N/A	0.2759

The best-performing algorithm on each network is boldfaced. If an algorithm does not finish on a particular network, the corresponding cell is marked "N/A"

columns in Table 2.3, and more detailed description of the datasets can be found on the repository webpage. These networks are then transformed to unweighted and undirected networks. For METIS, MLR-MCL and Graclus, we also provide the number of ground truth communities as input.

Table 2.3 also reports the quality of communities that are identified by each algorithm, using F-1 scores. The formulation of F-1 score follows the one in [53], which assesses both the quality of discovered communities as well as the coverage of ground truth communities. Specifically, given $\hat{C}$, the set of detected communities, and C^*, the set of ground truth communities, the F-1 score reported in the table is calculated by:

$$\frac{1}{2} \cdot \left[\frac{1}{|\hat{C}|} \cdot \sum_{\hat{c} \in \hat{C}} \max_{c^* \in C^*} f(\hat{c}, c^*) + \frac{1}{|C^*|} \cdot \sum_{c^* \in C^*} \max_{\hat{c} \in \hat{C}} f(\hat{c}, c^*) \right]$$

where $f(\hat{c}, c^*)$ is the typical harmonic mean of $\hat{c}$'s precision and recall, with regard to c^*. That is, F-1 scores of best matchings for all detected communities as well as ground truth communities are averaged. Note that, since overlapping communities are present in the ground truth information, F-1 scores are biased against the four algorithms under discussion. The amounts of time (in seconds) for each algorithm to run are listed in Table 2.4.

Table 2.4 Running time (in seconds) by four multilevel algorithms

	METIS	MLR-MCL	Graclus	Louvain
Facebook	0	0	0	0
Twitter	7	14	42	**1**
Google+	31	122	35	**2**
Youtube	68	334	N/A	**7**
LiveJournal	682	2957	N/A	**123**
Amazon	22	10	N/A	**2**

The fastest algorithm on each network is boldfaced. If an algorithm does not finish on a particular network, the corresponding cell is marked "N/A"

As seen from both tables, the scalability of Graclus is limited when compared with others. It fails to finish on the three largest networks (Youtube, LiveJournal, Amazon), and those cells are marked "N/A" accordingly. The Louvain algorithm boasts a pronounced advantage in terms of efficiency, as it is uniformly faster than others. However, in terms of output quality, there is no clear winner across the board. Each algorithm is the best-performer on at least one network, and faster algorithms do not necessarily produce results of a higher quality. For practitioners of community discovery, this leaves an open choice of the underlying algorithm. If the number of communities in the network is specified in advance, or if there is a relatively small range for that value, it is highly suggested to experiment with various algorithms. Otherwise, the Louvain algorithm may be preferred because it does not require the number of communities as input.

2.3 Speeding up Community Discovery by Network Sampling

In the previous section, we have introduced a multilevel paradigm in community detection, based on the rationale that a community detection algorithm runs faster on a graph with fewer nodes and/or edges. Here, we discuss another commonly-used approach to generate small graphs: sampling. By selecting a subset of nodes and/or edges from the original graph, we also obtain a smaller graph to operate on, hence speeding up the computation process. Since network sampling is used for pre-processing, and independent from the subsequent community detection algorithm, it has the potential of providing performance improvement to all community discovery algorithms.

Although the concept of sampling is straightforward and easy to understand, its implementation will have direct implications on the results of community detection, in terms of both effectiveness and efficiency. Specifically, it has been shown that naively sampling nodes or edges by uniform randomness will introduce bias into the resultant graph, which will affect the results negatively. In this section, we will review a series of sampling methods that have been proposed, and describe the intuition, strategy and performance behind each of them.

Sampling methods can be divided into two broad categories: those that only select a subset of nodes and those that preserve all nodes. In the following discussion, we refer to the first type as *node sampling*, and the second type as *edge sampling*.[5] Table 2.5 lists the algorithms to be discussed in each category. For node sampling, a relevant problem is how to obtain community assignments for *all* nodes after a community detection algorithm has been run on the sampled network. We will discuss the solution to this problem as well, in Sect. 2.3.1.

[5]Note that node sampling can also be achieved by creating an edge-induced subgraph from a subset of edges, therefore the node selection process is not always explicitly performed. The key distinction here is whether all nodes from the original graph are kept in the resultant sample graph.

Table 2.5 Classification of network sampling algorithms

Node sampling (Sect. 2.3.1)	Random node sampling
	Random walk sampling with restart
	Random walk sampling with jump
	Forest fire sampling
	Expansion-based snowball sampling
Edge sampling (Sect. 2.3.2)	Random edge sampling
	Structural similarity-based sampling
	Structure- and content-based sampling

2.3.1 Node Sampling for Community Discovery

Walk-Based Sampling The first category of node sampling approaches to be discussed here are based on random walk and its variants, and they often mimic the exploratory behavior in real-world activities. Given a network, a node starts by exploring its vicinity according to a probabilistic way. The exploration continues recursively on node(s) that has just been explored, until n ($n < |V|$), a specified number, nodes have been visited. At that point, all nodes and edges that have been accessed during the walk are extracted and constitute the sampled graph. We introduce three walk-based methods: random walk with restart, random walk with random jump, and forest fire model [25].

Random Walk with Restart To perform a random walk with restart, we randomly select a starting node u^0 in the graph. Let u^i be the node that the random walk process is visiting at step i. At the i+1st step, with a probability of p, we will select one node $v \in \Gamma_{u^i}$ uniformly and *walk* to v, hence the method's name. Otherwise, we will move to u^0 (referred to as "restart"), even if u^0 is not connected with u^i. Random walk terminates when n nodes are reached, or the number of steps exceeds a threshold.

If the input graph G is not connected, the sampled graph will be confined to the connected component that u^0 belongs to, because by design the random walk will never reach other components. As a result, the number of sampled nodes is upper-bounded by the number of nodes in the connected component, which can be less than n.

A common practice to address this problem is to start another random walk if not enough nodes have been sampled after a number of steps. This can be repeated multiple times, until n distinct nodes have been accessed in total. Then the union of sampled graphs from all walks are returned as the sample.

Random Walk with Jump Random walk with jump also performs walking at each step with a probability p. However, with probability $1 - p$ the walk will randomly select any node in the graph to jump to. This is the key distinction from random walk with restart, which only allows jumping back to u^0, and it helps to let the walk out when it is stuck in a local neighborhood or connected component, alleviating the problem of a disconnected graph.

By definition, p in both types of random walk can be any value between 0 and 1. Typically the value of p is much greater than $1 - p$, so that walks are less likely to be terminated by restarting/jumping and the sampled graph can retain more community structure from the original graph. In practice, p is often set to 0.85.

Forest Fire Model The forest fire model is analogous to the action of fire spreading in woods. Unlike random walk where only one node is selected at a time by either walking, jumping or restarting, forest fire can possibly select multiple nodes and visit all of them simultaneously in the next step. As a result, the sequence of nodes sampled is no longer linear, but rather like a tree.

The forest fire model accepts one parameter $p \in [0, 1)$, which controls the number of neighbors to visit at each step.[6] To start, a seed node u^0 is randomly chosen. Then an integer x is drawn from a geometric distribution with mean $\frac{p}{1-p}$. Such geometric distribution models the number of binary trials it takes before the first success appears, and p is the probability of failure. Naturally, as p increases, the value of x is expected to be greater, too. Existing literature has recommended setting p to 0.7 [24].

After the value x is decided, u^0 will randomly select x of its unvisited neighbors and "burn" them. It is possible that less than x nodes can be found to satisfy the requirement, because the degree of u^0 is less than x or less than x neighbors are still unvisited. In that case, the forest fire model will burn all of them.

The process continues until n nodes have been sampled or no more nodes can be burned, at which time nodes and edges that have been burned are returned as the sampled graph. At each step, one or more nodes are burning neighbors simultaneously, and this distinguishes forest fire from random walk. Also note that a node can be burned at most once, in order to prevent cycling. If the fire dies out before a sufficient number of nodes have been sampled, additional runs of forest fire can be done.

Evaluation and comparison of those walk-based methods are discussed in [24], where the authors compare the distribution of various graph statistics between the original graph and the sampled graph. The statistics include degree, size of connected component, hop-plot and clustering coefficient. Random walk with restart and forest fire generally outperform other approaches, including random node selection and random edge selection. Furthermore, those methods are able to match properties of the original graph with as few as 15 % of nodes.

Expansion-Based Sampling Another type of node sampling is expansion-based methods. Before describing the algorithm, we first define the meaning of "expansion". Given a set of nodes S on graph G, its *expansion factor* is calculated as

$$X(S) \equiv \frac{|\Gamma_S|}{|S|}$$

[6]The forest fire model described here is slightly different from that originally proposed in [25], which operates on directed graphs and thus has two parameters to control the "burning" of in- and out-links, respectively.

Correspondingly, the *maximum expander set* of size k, as defined in [31], is a set of k nodes with the maximal expansion factor:

$$\arg \max_{S:|S|=k} X(S)$$

Expansion-based sampling methods hinge on the conjecture that samples with high expansion factors are more representative of the original graph's community structure, compared with samples with low expansion factors. Since nodes with high expansion factors are usually bridges nodes leading to new communities, a sampled graph that includes those nodes is likely to contain most communities in the original network.

Based on this intuition, the problem of identifying a good sample of the network becomes finding the maximal expansion set of a specific size. However, solving the maximal expansion set problem exactly has high complexity due to its combinatorial nature, and two approximate sampling methods are proposed in [31].

Snowball Sampling The first approach, snowball sampling, is a greedy algorithm which adds the node that maximally improves the expansion at each step, given the current set of sampled nodes. Let $S^0 = \emptyset$ be the initial set of sampled nodes, then at the ith step, we update the new sample set as $S^i = S^{i-1} \cup \{u\}$, where $u = \arg\max_{u \in V - S^{i-1}} |\Gamma_u - (\Gamma_{S^{i-1}} \cup S^{i-1})|$. That is, adding u to the current sample set S introduces more new neighbors than any other node outside of S does. Alternatively, one can adopt a stochastic approach and select a node with a probability proportional to its improvement of expansion. This can account for occasional cases where nodes with near-highest expansion improvement actually lead to more communities.

Markov Chain Monte Carlo (MCMC) The second approach for expansion sampling is to use Markov Chain Monte Carlo (MCMC), where each of G's subgraphs of size n (the desired sample size) is considered as one state in the Markov chain. Recall that the definition of expansion factor of a sample set S is $\frac{|\Gamma_S|}{|S|}$. The maximal possible expansion factor for any set S is $\frac{|V-S|}{|S|}$, i.e. S can reach every node in G with one hop. The normalized expansion factor is therefore

$$\frac{|\Gamma_S|}{|V - S|}$$

and this value is used as the quality score for a sample S in the Markov chain.

The Markov chain consists of each subset of n nodes as a state, and it starts by randomly choosing a state S^0 that has n nodes. At the ith step, we find a candidate state S^i_{can} by randomly replacing one node in S^{i-1} with a node not in $V - S^{i-1}$. Depending on the quality scores (that is, the normalized expansion factor defined above) of S^{i-1} and S^i_{can}, there are two outcomes. If the quality score of S^i_{can} is greater than that of S^{i-1}, we accept it as the next state, letting $S^i = S^i_{can}$. Otherwise, we only accept it with a probability of

$$\left[\frac{quality(S^{i-1})}{quality(S^i_{can})}\right]^p$$

where $p = 10\frac{|E|}{|V|}\log_{10}|V|$, a function of both network size and the edge-to-node ratio [17]. If S^i_{can} fails to be accepted, we keep $S^i = S^{i-1}$.

When reaching the stationary distribution, sampling from the Markov chain becomes equivalent to sampling from all subgraphs of size n according to their quality scores. If S is the node set sampled from the Markov chain, we return the node-induced subgraph of S on G as the sampled graph.

Inferring Community Assignments for All Nodes Given a sampled graph from any aforementioned node sampling technique, we can run community detection algorithms on it efficiently because of the much smaller scale. One question unresolved, however, is how community assignments of nodes in the sampled graph can be translated back to other nodes in the original network, since the goal is to perform community detection on *all* nodes in the graph.

To that end, one can resort to relational learning techniques, where community label is the attribute of interest. This is a univariate collective inferencing problem [30], whose input are $G = (V, E)$ and $x_u, \forall u \in S$, i.e. the community labels for nodes in the sampled graph. The goal is to learn the exact community label or the distribution over possible labels for each unsampled node $u \in V - S$. A number of collective inference methods have been proposed and evaluated, and in [30] the authors find that relaxation labeling with a weighted majority relation model yields the best overall performance. In [31], the authors compare the quality of communities identified by several sampling methods. Their observation is that expansion-based sampling combined with the aforementioned inference technique produces community labels that are closest to running the community detection algorithm on the entire graph directly.

2.3.2 Edge Sampling for Community Discovery

Contrary to node sampling, where only a subset of nodes is retained, edge sampling aims at selecting edges with certain criterion while preserving all nodes. One advantage of edge sampling, therefore, is that community assignments can be learned on *all* nodes at once without the need of further inference.

Since the purpose of sampling is to create a representative subnetwork that captures the graph's community structure, naive techniques do not fare well. For example, selecting edges by equal probability is known to be biased towards high-degree nodes. Therefore, the edge selection criterion needs to consider the importance of each edge in the community structure. Here, we introduce two methods which utilize the notion of *similarity* in performing edge sampling.

Sampling Using Structural Similarity Intuition The intuition behind sampling based on structural similarity [46] can be stated as: An edge is likely to (not) connect two distinct clusters if the adjacency lists of its two incident nodes have high (low) overlap. High overlap of adjacency lists indicates a large number of triangles that the edge belongs to, hence a high likelihood of the edge residing in a dense region, i.e. a cluster. We view those edges as having high structural similarity.

By preferentially retaining edges with high structural similarity which are likely to be intra-cluster, we expect to preserve the community structure in the sampled graph. Note that in order to keep the sampled graph significantly smaller than the original graph, it is inevitable that some high-similarity edges will be discarded during the sampling process. However, provided that a great fraction of low-similarity noise edges are removed as well, we will still be able to recover the community assignments of nodes from the sampled graph.

Calculating structural similarity The next question is to identify high-similarity edges efficiently. While edge centrality measures, such as betweenness centrality [32], have been proposed before to find "bottleneck" edges, i.e. low-similarity edges, they are prohibitively expensive to calculate with a complexity of $O(|V| \cdot |E|)$. To address this issue, one can calculate the Jaccard similarity of two incident nodes' adjacency lists and use it as the edge's structural similarity. For an edge (u, v), we view the neighborhoods of u and v as two sets, and the Jaccard similarity between the two sets is defined as:

$$sim(u, v) = \frac{|\Gamma_u \cap \Gamma_v|}{|\Gamma_u \cup \Gamma_v|}$$

More importantly, established methods like min-wise hashing [4] can be leveraged to estimate the Jaccard similarity of two sets in constant time. To generate one min-wise hash of Γ_u, we apply π, a permutation of V, on it and take the minimal value after the permutation. Formally, the hash value $h_\pi(\Gamma_u)$ (or $h_\pi(u)$ for short) is:

$$h_\pi(u) \equiv h_\pi(\Gamma_u) = \min_{v \in \Gamma_u} (\pi(v))$$

A min-wise hash signature of length k for u is generated by randomly drawing k permutations $\pi_1 \ldots \pi_k$ and concatenating the resultant hash values $h_{\pi_1}(u) \ldots h_{\pi_k}(u)$. The same set of permutations are applied to all adjacency lists to generate the corresponding length-k signature for each node in V. The following statistic is an unbiased estimator of the Jaccard similarity between Γ_u and Γ_v:

$$\hat{sim}(u, v) \equiv \frac{1}{k} \sum_{n=1}^{k} I[h_{\pi_n}(u) = h_{\pi_n}(v)]$$
$$E[\hat{sim}(u, v)] = sim(u, v)$$

where $I[\bullet]$ is the identity function, and it only takes $O(k)$ time to compute, a constant to the size of G.

Global and local sampling Given the structural similarity value of each edge in G, sampling can be done in one of two ways. In *global sampling*, all edges are sorted in decreasing order of their structural similarity values, and a number of top-ranked edges are retained. While easy to implement, this scheme suffers from a key drawback that it treats all edges in the graph equally and does not distinguish between clusters of varying densities. In practice, it will be able to preserve the structure of denser communities but not relatively sparse communities, since edges in the latter will have lower structural similarity and be pruned away. Global sampling also risks disconnecting the graph, since it is possible that all edges incident to a node are removed.

An alternative strategy is *local sampling*, where sampling is performed within each node's neighborhood separately. The number of incident edges that each node can keep is a function of the node's degree, for example, a power function d^e where d is degree and e is a user-specified sampling exponent ($e < 1$). This eliminates the need for a global threshold. Operating on each node separately also keeps the graph connected, as each node will be connected to at least one neighbor. Furthermore, the strict concavity of the power function ensures that sampling on high-degree nodes is more aggressive, i.e. a larger proportion of edges are removed from them. This is desirable, since high-degree nodes tend to straddle multiple clusters, and thus contain a higher fraction of inter-cluster edges.

Sampling using structural and content similarities Many real-world graphs are associated with abundant content and/or attribute information,[7] apart from the inherent network topology. The existence of content information can be leveraged to eliminate the impact of noise in the network and strengthen the community signal. This is studied in [44] where the authors investigate edge sampling based on both structural and content similarities.

Calculating content information similarity Given any two nodes of interest, the first step is to calculate their similarity based on the corresponding content information. One approach is to measure the cosine similarity between two nodes' content vectors, since many types of information can be represented as feature vectors, such as TF-IDF values for documents, SIFT features for images, as well as discrete and continuous attributes.

Furthermore, hashing via random projection method [7] can be used to efficiently estimate the cosine similarity. Let $\mathbf{c}_u \in \mathbb{R}^d_{\geq 0}$ be the content vector of node u, we can draw a random vector $\mathbf{r} \in \{0, 1\}^d$ and use $h_{\mathbf{r}}(u) \equiv sgn(\mathbf{c}_u \cdot \mathbf{r})$ as one hash value for $\mathbf{c}_u$. Similar to the case of min-wise hashing, if we draw k different random projections and apply them to each content vector, the following result provides a way to estimate the cosine similarity of two vectors in constant time:

[7]Both content and attribute information are modeled as an auxiliary feature vector associated with each node in the graph, so that the formulation is applicable to text, image, and many other forms of information, all of which will be referred to as "content information" henceforth.

$$E\left[\frac{1}{k}\sum_{n=1}^{k} I[h_{\mathbf{r_n}}(u) = h_{\mathbf{r_n}}(v)]\right] = 1 - \frac{\arccos(\cos(\mathbf{c}_u, \mathbf{c}_v))}{\pi}$$

Combining structural and content similarities Before fusing two types of similarity, it is important to note that edges not currently present in the graph G should also be considered. The intuition is that if a pair of nodes shares common neighbors and contents but are not directly connected in G, they would otherwise be omitted.

To that end, one can independently find K neighbors with most similar content vectors for each node, and denote this set of edges as $E_{content}$.[8] For each node, we then consider its neighborhood under $E \cup E_{content}$ and combine the structural similarity and content similarity of each edge therein. For simplicity, the combination can be done as averaging with equal weight, and in practice no significant difference in the quality of resultant communities is observed when the weight is varied.

Local sampling can then be performed using the combined similarity values of edges, following the same procedure described previously. It is worth noting that the process can possibly introduce new edges that do not exist in G, via the creation of $E_{content}$, therefore the resultant graph is not always a subgraph of G. In the case of such edge "recovery", however, the corresponding edges always have high combined similarity and are very likely to be intra-community.

Sampling using combined similarity is particularly beneficial to graphs that are already sparse, where otherwise little improvement could be obtained by sampling using structural similarity alone. By considering content vectors during the sampling process, information encoded in edges is greatly enriched, leading to communities of higher quality with more meaningful real-world implications. On graphs of modest density, the combined sampling strategy also outperforms other approaches without significant detriment to efficiency.

2.3.3 Empirical Results and Summary

In this section, we have described various approaches in the realm of network sampling. Rather than working as standalone algorithms, those methods aim at preprocessing a network, keeping only a subset of nodes and/or edges in the network. The benefit to any subsequent community discovery algorithm is two-fold: preprocessing reduces the size of the network, and it preserves the community structure, instead of noise, in the network.

Network sampling algorithms can be divided into two general categories: those that keep a selected subset of nodes (i.e. node sampling in Sect. 2.3.1), and those that retain all nodes but fewer edges (i.e. edge sampling in Sect. 2.3.2). Existing work in

[8] An empirical guideline to select K is to let the size of $E_{content}$ be similar to that of E.

node sampling has approached it from various angles, and has contributed methods such as walk- and expansion-based sampling. Edge sampling has also shown promising results, as it biases towards edges that are more likely to reside in a community. Furthermore, structural similarity can be integrated with auxiliary information, such as content or attributes, when it is available, so that a more informed sampling decision can be reached.

The effectiveness and efficiency of sampling are evaluated in [46], where real-world networks are processed by various edge-sampling schemes, in order to compare sampled networks with the same set of nodes. The experiments are run on a combination of:

- Three social networks: Orkut ($|V| = 3072626$, $|E| = 117185083$), Twitter ($|V| = 146170$, $|E| = 83271147$),[9] Flickr ($|V| = 31019$, $|E| = 520040$).
- Four sampling methods: random edge sampling, forest fire (Sect. 2.3.1),[10] global and local similarity-based sampling (Sect. 2.3.2).
- Three multilevel community discovery algorithms (Sect. 2.2): METIS, MLR-MCL, Graclus.

Contrary to the benchmark networks used in Sect. 2.2, no ground truth information is associated with the three datasets evaluated here. Therefore, averaged conductance [8, 26] is used to measure the structural quality of communities that are discovered. The conductance of a community is the total number of edges leaving it, divided by the total number of edges with endpoint in the community, or the total number of edges with endpoint in the complement of the community, whichever is smaller. A low conductance value of a community represents a small inter-community edge volume, thus clear separation of it from the rest of the network. The conductance values of all communities are averaged to measure the overall structural quality.

The improvements of network sampling on conductance and running time are shown in Tables 2.6 and 2.7, respectively. The running time of a sampling method includes times for both the sampling process itself and community discovery on the subsequent sampled network. Across the board, local sampling based on structural similarity demonstrates the best balance between quality improvement and speedup. Although simpler sampling methods (random edge sampling, forest fire sampling) are faster, they do not preserve the community structure in the network, resulting in lower structural quality of the output communities. Global similarity-based sampling suffers from similar issues, as it tends to generate an extraordinarily high proportion of singletons, disconnecting them from the rest of the network.

[9]This is different from the Twitter network described in Sect. 2.2.5.

[10]Although forest fire is designed for node sampling, one can perform forest fire repeatedly, each time on a randomly-selected unburned node, until most nodes are burned. The collection of all burned edges are considered sampled edges.

Table 2.6 [Reproduced from [46]] Conductance of communities identified on the original networks ("Orig.") and networks preprocessed by random edge sampling ("RE"), forest fire sampling ("FF"), global similarity-based sampling ("GS"), and local similarity-based sampling ("LS")

	METIS					MLR-MCL					Graclus				
	Orig.	RE	FF	GS	LS	Orig.	RE	FF	GS	LS	Orig.	RE	FF	GS	LS
Orkut	0.85	0.82	0.82	0.76	**0.76**	0.78	0.85	0.86	0.91	**0.78**	N/A	N/A	N/A	N/A	N/A
Twitter	0.95	1.00	0.99	0.97	**0.96**	0.90	0.99	0.99	0.89	**0.86**	0.90	1.00	0.99	0.97	**0.91**
Flickr	0.87	0.91	0.91	**0.71**[11]	0.84	0.71	0.83	0.88	0.72	**0.70**	0.66	0.72	0.71	**0.66**	0.72

The lower the conductance, the higher quality the discovered communities have. The best-performing sampling method on each pair of network and community discovery algorithm is boldfaced. If a community discovery algorithm does not finish on a particular network, the corresponding cell is marked "N/A". Although global similarity-based sampling enables low conductance on Flickr, it generates 30 % singletons on this network, whose contributions to the conductance are not included

Table 2.7 [Reproduced from [46]] Running time (in seconds) on the original network and speedup achieved by network sampling, including the time of both sampling *and* community discovery on the resultant sampled network

	METIS					MLR-MCL					Graclus				
	Orig.	RE	FF	GS	LS	Orig.	RE	FF	GS	LS	Orig.	RE	FF	GS	LS
Orkut	14373	13x	12x	30x	36x	21079	6x	6x	39x	22x	N/A	N/A	N/A	N/A	N/A
Twitter	2307	35x	14x	85x	6x	14569	63x	16x	188x	22x	1518	138x	138x	66x	5x
Flickr	5	8x	9x	1x	3x	17	3x	3x	2x	4x	1	2x	2x	1x	2x

If a community discovery algorithm does not finish on a particular network, the corresponding cell is marked "N/A"

2.4 Exploiting Parallelism and Distributed Architectures in Community Discovery

All algorithms described so far in this chapter operate as single-thread processes on a single machine, and that leaves two problems to be addressed. First of all, the parallelism that is inherent to many community discovery algorithms has not been fully exploited. Two forms of parallelism are being considered here: data parallelism that allows accessing multiple segments of data for computation at the same time, and control parallelism that enables the execution of the algorithm on multiple nodes or communities simultaneously. The second problem is rooted in the fact that network data is growing at an unprecedented rate, and more and more frequently exceeds the RAM capacity of a single machine. To be able to handle very large networks, we will need to resort to various distributed computing architectures that have been developed over the years, such as message passing, MapReduce, and others.

In this section, we discuss some promising directions that have the potential of solving these two issues in community discovery. We will first emphasize the critical role that (sparse) matrix/vector operations play in many community discovery algorithms (Sect. 2.4.1). This underpins the possibility of scaling up community discovery by leveraging computation kernels on various frameworks, including OpenMP, Many Integrated Core (MIC), General-Purpose GPU (GPGPU), and Hadoop. While the discussion focuses on selected families of community discovery algorithms, the findings are generalizable to many others, as long as they are composed of matrix/vector operations.

We will also introduce the parallelization of several community discovery algorithms that are based on optimizing certain objective function values. For many algorithms in this category [20, 41, 53], the serial version was proposed first, and subsequent algorithm speedup is accomplished by parallelizing critical subroutines that are otherwise time-consuming, such as the calculation of objective function values. Depending on the problem scale, algorithms have been proposed for both shared-memory systems (Sect. 2.4.2) and distributed-memory systems (Sect. 2.4.2). We will introduce one instance for each type of system.

2.4.1 *Speeding up and Scaling up Matrix Operation-Based Algorithms*

Many community discovery problems such as Markov clustering, spectral clustering, and matrix factorization can be reduced to matrix operations. Much progress has been made in performing parallel matrix operations efficiently, which directly contributes to faster and more scalable community discovery algorithms. Here, we describe the optimizations for two types of them: Markov clustering (Sect. 2.4.1) and spectral clustering (Sect. 2.4.1).

Fast MLR-MCL with Efficient Sparse Matrix-Matrix Multiplication (SpMM) As seen in Sect. 2.2, MLR-MCL implements a multilevel version of regularized Markov clustering (R-MCL). Perhaps not surprisingly, the most time-consuming block in MLR-MCL is R-MCL, because it involves multiplying two large sparse matrices, M and M_G. If one can exploit the data sparsity and speed up the multiplication subroutine, the efficiency of MLR-MCL can be significantly improved. The same reasoning applies to other matrix operation-based community discovery algorithms as well.

To this end, researchers have recently investigated the effect of applying SpMM kernels to MLR-MCL on various parallel architectures. In [33], the authors successfully improve the performance of MLR-MCL on multi-processor CPUs by designing a novel SpMM computation kernel. In order to perform SpMM efficiently, memory footprint analysis is performed so that the workload is well-balanced among processors. Experiments on eleven networks report a speedup of up to 10x. More importantly, the utility of the SpMM kernel implemented in this work has a broader impact beyond one single algorithm. Given its superior performance against the Intel Math Kernel Library on both multi-processor CPU and the new multi-core MIC architecture (2x on average), similar speedups can be expected if the kernel is properly applied to other matrix-based community discovery algorithms.

Another promising direction in accelerating matrix operations is GPGPU programming. The architecture of GPUs is by design highly parallel, and computation can be performed by on-board chips accessing high-throughput on-board memory, without incurring latency between GPU and CPU. As a result, GPGPU has become a powerful and low-cost platform that is suitable for matrix-matrix multiplication and other similar operations. Such advantages have been leveraged in [6], where a CUDA-based implementation of MCL (CUDA-MCL) is proposed and has proven highly efficient. Future research efforts are warranted to extend this work to MLR-MCL, as well as other community discovery algorithms that can utilize SpMM or SpMV(ector) kernels.

Hadoop-Based Eigensolver for Scalable Spectral Clustering Spectral graph theory has been extensively studied [8], and researchers have drawn a close link between the spectral properties of a graph and its community structure. Briefly speaking, the goal of spectral clustering is to represent nodes by their representations induced from the top k eigenvectors of matrix $D^{-\frac{1}{2}}(D-A)D^{-\frac{1}{2}}$, equivalent to eigenvectors associated with the k smallest eigenvalues of matrix $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$. K-means clustering can be run on those length-k vectors to find community assignments for nodes [50].

While it may be tempting to apply spectral clustering on large networks using distributed computing, in practice non-trivial efforts are required. Storing network information is straightforward, as only $|V| * k \ll |E|$ values need to be stored. However, finding the k eigenvectors requires significant memory, and consequently existing eigensolvers (e.g. PLAPACK, ScaLAPACK) do not scale to billion-by-billion matrices.

To address this challenge, a Hadoop-based eigensolver called HEigen is proposed [18]. It employs the Lanczos algorithm, and selectively reorthogonalizes vectors to avoid generating spurious eigenvalues. Expensive operations such as matrix-vector and matrix-matrix multiplication are distributed using the MapReduce framework. HEigen also uses blocking to transfer data of multiple nodes at once, so that a significant amount of time is saved in network transfer. Furthermore, data size skewness is also exploited, and small vectors/matrices are sent to mappers as distributed caches in Hadoop.

According to the evaluation on HEigen's scalability and efficiency, it is able to scale to graphs with billions of edges. Several design choices facilitate reducing the running time, and the cached matrix-matrix multiplication kernel contributes most to the speedup (76X compared with a naive implementation of the multiplication kernel). However, the quality of spectral clustering is not measured, presumably due to the lack of ground truth for these very large networks or the difficulty in calculating clustering quality measures (e.g. conductance, normalized cut, normalized mutual information) itself. In the future, one remedy is to evaluate the performance of this Hadoop-based spectral clustering algorithm on synthetic networks generated by benchmark suites such as LFR [22], where the gold standard is readily available.

2.4.2 *Parallelizing Objective Function Optimization-Based Algorithms*

Here, we introduce two parallel community discovery algorithms that are based on optimizing the value of a certain objective function. SCD (Sect. 2.4.2) maximizes the weighted clustering coefficient, whereas ParMETIS (Sect. 2.4.2) minimizes the edge-cut among communities. They both can be parallelized thanks to repeating local computations. One distinction is that SCD is implemented on shared-memory systems, while ParMETIS is designed for distributed-memory systems, facing the additional challenge of reducing communication cost.

Scalable Community Detection (SCD) SCD [41] is a two-phase algorithm which parallelizes repeating computations. It sits on the Pareto frontier of time complexity and quality coordinates among all state-of-the-art algorithms. It optimizes a modularity-based function: Weighted Clustering Coefficient (WCC) [40], which essentially counts the number of triangles. The objective function and the movement function (discussed later) can be computed locally while requiring global information. As a result, the algorithm can be parallelized on a shared memory framework using techniques such as OpenMP and the like.

WCC is a community metric based on the number of triangles. Intuitively, for a node v and a community C, the more triangles v closes within C, the more likely v belongs to C, and the higher its WCC value with regard to C is. The WCC of a community is the average WCC over all nodes in the community, and the WCC of the community assignments in a network is the weighted average WCC over all communities.

SCD has two partitioning phases plus a preprocessing phase, which removes edges that are far from any triangle. The initial partitioning first computes the local clustering coefficient [47] of each node, and sorts all nodes in descending order of their local clustering coefficients. Then it uses each node in the ranking as the center of a star to partition the graph, which makes each community a star with a high local clustering coefficient node as its center. The local clustering coefficient computation can be parallelized where each thread computes the local clustering coefficients of a subset of nodes.

The refinement partitioning phase moves nodes among communities as follows. For each node in the graph, its movement function is computed. When all movement functions are obtained, the partition and WCC are updated. This process repeats until the WCC value converges. The movement function of a node decides whether and how to move a node based on the WCC gain of the action, and it can be either (1) moving the node to another selected community, or (2) keeping the node in its current community. Both outcomes can lead to the creation or removal of a single-node community. The computation of the movement function can also be parallelized where each thread computes the movement function of a subset of nodes.

ParMETIS ParMETIS [20] is the parallel version of METIS (Sect. 2.2.1). It is designed for tasks like solving linear systems and computing finite element meshes in a distributed manner. Hence the implicit assumption is that the number of partitions/communities is no greater than the number of processors. This framework achieves a significant time reduction compared with the serial version of METIS, while maintaining comparable quality in terms of edge-cut.

Framework and challenges Similar to the serial version, ParMETIS also contains three phases: contraction, partitioning and refinement. The contraction phase is similar to that in the serial version except that multiple processors participate in the computation of maximal matching. While it is straightforward in a shared-memory system, a great number of communications and synchronizations are required to obtain a good matching in a distributed-memory architecture. A *local matching*, where each processor only computes matching of vertices in its own memory, can perform decently only when the graph has already been well-partitioned and each partition component is stored in a processor. This pre-condition is unrealistic since it is exactly the goal of the algorithm: to obtain a good partitioning of the graph. On the other hand, a *global matching*, where each processor considers vertices in other processors while computing matching, requires a high degree of fine-grained inter-processor communication. Moreover, a distributed architecture may encounter circular matching among processors, which makes some local optimum inaccessible.

After the coarsening phase, the graph has already been significantly reduced and one processor can handle the partitioning computation efficiently. Yet, since the graph is stored distributedly, moving data to a single processor still incurs additional cost. ParMETIS computes a p-way partitioning (p is the number of processors) via a recursive bisection algorithm similar to [19], and each processor explores a recursive branch.

During the refinement phase, ParMETIS refines partitioning while uncoarsening the series of graphs. Again, circular movement poses the same challenge as that in the contraction phase. A scheme proposed by [10] suggests that multiple runs of vertex movement between disjoint pairwise partitions can avoid the circular movement problem. The bottleneck, however, is that if each partition i has k_i neighboring partitions, the number of runs is no less than $\max_i k_i$. Another drawback of this scheme is that the local greedy approach lacks a global view and can easily be trapped at a low quality local optimum.

A coloring-based scheme to avoid circular matching and movement To overcome the aforementioned challenges, a method approach based on graph coloring is proposed here. Recall that a coloring of a graph partitions a graph into κ independent sets, where κ is its chromatic number. The constraint that every time only the vertices of the same color are moved avoids circular matching and movement. To color a graph, Luby's algorithm [29], an incremental algorithm which works well on shared-memory architectures, is modified and adopted.

- **Concurrent coloring of graph vertices** On a distributed-memory architecture, a multinode and its neighbors might be stored in different processors. Therefore, the communication cost of inquiry a neighbor's location may be large. Moreover, the original Luby's algorithm has significant synchronization overheads. A variant of Luby's algorithm, which trades quality[11] for concurrency and time efficiency, is proposed here. The variant is simple: prior to the execution of Luby's algorithm, a *communication setup* phase occurs, where each vertex's external relation, i.e. whether a vertex has a neighbor located in another processor, is determined. Not all nodes are colored in each iteration, and the algorithm terminates when a large fraction of nodes are colored.
- **Matching after coloring** Matching after coloring can avoid circular matching and achieve high concurrency. The matching algorithm based on a colored graph is efficient when parallelized, and it works as follows: Initially each vertex has a mark variable called *Match* with a value of -1. At the end of computation, the *Match* variable of a matched vertex is its partner. Iterating on each color, each unmatched vertex selects one of its unmatched neighbors based on the heavy-edge heuristic, and updates both vertices' *Match* variables to be each other. Multiple vertices may attempt matching the same vertex. On a shared-memory architecture, this is solved by a first-come-first-match strategy. On a distributed-memory architecture, upon receiving all matching requests, the to-be-matched vertex determines its partner based on the heavy-edge heuristic and rejects other matching requests. Matching computation terminates after all colors have a chance to match.
- **Refining after coloring** Implemented in a way similar to the matching procedure, moving a vertex color by color from a partition to another also avoids circular movement. There are two optimization techniques to be considered here. The first one stems from the observation that on a distributed-memory architecture, the algorithm does not actually move a vertex from the processor containing one

[11]The computed independent set is no longer guaranteed to be maximal.

partition to the processor that contains another partition. Rather, it only need to assign each vertex (multinode) a partition ID. The other optimization deals with the balance constraint in each partition. Since one partition may be stored distributedly, at the end of each color's movement, the weights of the partitions are globally recomputed and each processor updates the related information.

2.4.3 Summary

In this section, we have listed several promising approaches of scaling up community discovery algorithms by exploiting the intrinsic parallelism and existing distributed computing frameworks such as message passing and MapReduce. We first point out that many community discovery algorithms essentially involve matrix-matrix multiplications and other similar operations, and the underlying matrices are often sparse. This enables us to leverage matrix operation kernels that have been proposed on various architectures including multi-processor CPU, many-core CPU, GPGPU, and Hadoop, and to obtain significant speedup of community discovery algorithms such as Markov clustering and spectral clustering. We then discuss how both shared-memory and distributed-memory systems can be adapted to perform community discovery based on optimizing objective function values. The two instances described, SCD and ParMETIS, are able to execute orders of magnitude faster than their serial counterparts, with only minor impact on the output quality.

There exists more work in the literature that aims at parallelizing community discovery algorithms, such as parallel label propagation [34], parallel Louvain method [48], and parallel modularity-based agglomerative clustering [43]. While we do not describe them in details in this section, they also deserve substantial research attention.

2.5 Conclusion

In this chapter, we have presented three categories of simple approaches that aim at performing effective community discovery in an efficient and scalable fashion. We first describe a generic contract-and-conquer multilevel framework, where a network is recursively contracted into a series of increasingly smaller networks, so that a core community discovery algorithm can run efficiently on the miniature networks. We then review the idea of sampling vertices and/or edges in a network via various strategies, and we show that one can indeed reduce the network size, thus accelerating the community discovery process, and retain its community structure at the same time. Finally, we outline existing efforts and future work directions for performing community discovery in a parallel or distributed computing environment, in order to ensure sufficient storage for large-scale networks as well as to speed up the computation. It is important to note that these three groups of methods are not mutually exclusive, and there are great potentials in combining them to design new community discovery algorithms that can process networks of tera-scale (or even larger) in a reasonable amount of time.

We believe that for a community discovery algorithm to be practical for large networks, it needs to follow simple but effective design principles. As a result, we focus on the three aforementioned categories which are well-tested in empirical evaluations and are highly flexible. For more thorough reviews of state-of-the-art methods in the field of community discovery in general, readers can refer to various recent surveys [14, 35, 37, 38].

Specifically, we limit the algorithms discussed in this chapter to those that discover disjoint communities, instead of allowing communities to overlap. Detecting overlapping communities is another focal point in community discovery that is being actively studied [23, 28, 53], and they can benefit from all the three approaches introduced in this chapter. Furthermore, most algorithms covered here also require parameters in some forms to determine the number of communities to discover. In practice, if such information is absent, one can often estimate the value based on some empirical rules of thumb, such as the one in [27].

Apart from overlapping and non-parametric community discovery, there are other outstanding challenges in the realm of user community discovery that are worthy of significant future efforts. In particular they include:

- **Community discovery in graph streams** Graph data in many domains are changing rapidly, such as the addition and removal of user connections in online social networks. In such cases, it becomes inefficient to apply static community discovery algorithms to a collection of network snapshots. Further complicating the problem is the fact that graph data may arrive in a streaming fashion, making it infeasible to store updates and revisit them in the future. The states of communities need to be maintained and updated in an efficient way, in order to avoid backlog in the processing pipeline. To date, only a small number of work [2] has approached this problem directly.
- **Community discovery in signed networks** A signed network is a network whose edges can represent positive or negative relationships. Such relationships can be explicit, e.g. stated friendship or rivalry between two social network users, or implicit, e.g. empirically-measured excitation or inhibition between two proteins. Existing literature on mining communities from signed networks [11, 52] only operates on small networks that have at most thousands of nodes, and has not attained the ability to efficiently handle large-scale user networks.
- **Community discovery combining structure and attribute/content information** Many network datasets are accompanied by a rich collection of auxiliary information, such as social network user attributes and content. Prior work has explored the use of these data in addition to structural information, and managed to identify communities of users that are dense in connection as well as coherent in attributes/content [44, 54, 55]. While improving the quality of communities that are discovered, these approaches are not as efficient as their structural information-

only counterparts. Therefore, it is important to design more efficient methods that can leverage attribute and content information in community discovery.

The three paradigms summarized in this chapter have all demonstrated superior flexibility and the ability to improve algorithmic efficiency and scalability. In the future, we envision wide adoption of these paradigms in well-investigated as well as emerging community discovery problems, in order to create more powerful tools for identifying user communities in large networks. Since these frameworks are not mutually exclusive but rather complementary, it is also viable to leverage multiple of them in conjunction, and the potential improvement of performance and output quality can be profound.

Acknowledgments We are thankful to the Editors and anonymous reviewers for their valuable comments, insightful suggestions and constructive feedback that greatly helped improving this article.

This work is supported by NSF Grants **IIS-1111118**, **CCF-1240651**, and **DMS-1418265**. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Adamic LA, Glance N (2005) The political blogosphere and the 2004 US election: divided they blog. In: Proceedings of the 3rd international workshop on link discovery. ACM, pp 36–43
2. Aggarwal CC, Zhao Y, Philip SY (2010) On clustering graph streams. In: SDM. SIAM, pp 478–489
3. Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. J Stat Mech: Theory Exp 2008(10):P10008
4. Broder AZ, Charikar M, Frieze AM, Mitzenmacher M (1998) Min-wise independent permutations. In: Proceedings of the thirtieth annual ACM symposium on theory of computing. ACM, pp 327–336
5. Bui TN, Jones C (1993) A heuristic for reducing fill-in in sparse matrix factorization. In: PPSC, pp 445–452
6. Bustamam A, Burrage K, Hamilton NA (2012) Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. IEEE/ACM Trans Comput Biol Bioinform (TCBB) 9(3):679–692
7. Charikar MS (2002) Similarity estimation techniques from rounding algorithms. In: Proceedings of the thirty-fourth annual ACM symposium on theory of computing. ACM, pp 380–388
8. Chung FR (1997) Spectral graph theory, vol 92. American Mathematical Society, Providence
9. Dhillon I, Guan Y, Kulis B (2007) Weighted graph cuts without eigenvectors a multilevel approach. IEEE Trans Pattern Anal Mach Intell 29(11):1944
10. Diniz PC, Plimpton S, Hendrickson B, Leland RW (1995) Parallel algorithms for dynamically partitioning unstructured grids. In: PPSC, pp 615–620
11. Doreian P, Mrvar A (2009) Partitioning signed social networks. Soc Netw 31(1):1–11
12. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions. In: 19th conference on design automation. IEEE, pp 175–181
13. Fiedler M (1973) Algebraic connectivity of graphs. Czechoslov Math J 23(2):298–305
14. Fortunato S (2010) Community detection in graphs. Phys Rep 486(3–5):75–174
15. George A, Liu J (1981) Computer solution of large sparse positive definite systems. Prentice Hall, Englewood Cliffs

16. Heath MT, Ng E, Peyton BW (1991) Parallel algorithms for sparse linear systems. SIAM Rev 33(3):420–460
17. Hubler C, Kriegel HP, Borgwardt K, Ghahramani Z (2008) Metropolis algorithms for representative subgraph sampling. In: Eighth IEEE international conference on data mining, ICDM'08. IEEE, pp 283–292
18. Kang U, Meeder B, Papalexakis EE, Faloutsos C (2014) HEigen: spectral analysis for billion-scale graphs. IEEE Trans Knowl Data Eng 26(2):350–362
19. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
20. Karypis G, Kumar V (1999) Parallel multilevel series k-way partitioning scheme for irregular graphs. Siam Rev 41(2):278–300
21. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(2):291–307
22. Lancichinetti A, Fortunato S, Radicchi F (2008) Benchmark graphs for testing community detection algorithms. Phys Rev E 78(4):046110
23. Lancichinetti A, Radicchi F, Ramasco JJ, Fortunato S (2011) Finding statistically significant communities in networks. PLOS ONE 6(4):e18961
24. Leskovec J, Faloutsos C (2006) Sampling from large graphs. In: Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 631–636
25. Leskovec J, Kleinberg J, Faloutsos C (2005) Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. ACM, pp 177–187
26. Leskovec J, Lang KJ, Dasgupta A, Mahoney MW (2008) Statistical properties of community structure in large social and information networks. In: Proceedings of the 17th international conference on world wide web. ACM, pp 695–704
27. Leskovec J, Lang KJ, Mahoney M (2010) Empirical comparison of algorithms for network community detection. In: Proceedings of the 19th international conference on world wide web. ACM, pp 631–640
28. Leung IX, Hui P, Lio P, Crowcroft J (2009) Towards real-time community detection in large networks. Phys Rev E 79(6):066107
29. Luby M (1986) A simple parallel algorithm for the maximal independent set problem. SIAM J Comput 15(4):1036–1053
30. Macskassy SA, Provost F (2007) Classification in networked data: a toolkit and a univariate case study. J Mach Learn Res 8:935–983
31. Maiya AS, Berger-Wolf TY (2010) Sampling community structure. In: Proceedings of the 19th international conference on world wide web. ACM, pp 701–710
32. Newman ME, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69(2):026113
33. Niu Q, Lai PW, Faisal SM, Parthasarathy S, Sadayappan P (2014) A fast implementation of mlr-mcl algorithm on multi-core processors. In: 21st annual international conference on high performance computing, HiPC 2014, Goa, India, 17–20 December 2014
34. Ovelgonne M (2013) Distributed community detection in web-scale networks. In: 2013 IEEE/ACM international conference on advances in social networks analysis and mining (ASONAM). IEEE, pp 66–73
35. Papadopoulos S, Kompatsiaris Y, Vakali A, Spyridonos P (2012) Community detection in social media. Data Min Knowl Discov 24(3):515–554
36. Parlett BN (1980) The symmetric eigenvalue problem, vol 7. SIAM, Philadelphia
37. Parthasarathy S, Faisal SM (2013) Network clustering. CRC Press, Boca Raton, pp 415–456
38. Parthasarathy S, Ruan Y, Satuluri V (2011) Community discovery in social networks: applications, methods and emerging trends. Social network data analytics. Springer, Berlin, pp 79–113
39. Pemmaraju S, Skiena S (2003) Computational discrete mathematics: combinatorics and graph theory with mathematica. Cambridge University Press, New York
40. Prat-Pérez A, Dominguez-Sal D, Brunat JM, Larriba-Pey JL (2012) Shaping communities out of triangles. In: Proceedings of the 21st ACM international conference on information and knowledge management. ACM, pp 1677–1681

41. Prat-Pérez A, Dominguez-Sal D, Larriba-Pey JL (2014) High quality, scalable and parallel community detection for large real graphs. In: Proceedings of the 23rd international conference on world wide web, international world wide web conferences steering committee, pp 225–236
42. Richter Y, Yom-Tov E, Slonim N (2010) Predicting customer churn in mobile networks through analysis of social groups. In: SDM. SIAM, vol 2010, pp 732–741
43. Riedy EJ, Meyerhenke H, Ediger D, Bader DA (2012) Parallel community detection for massive graphs. Parallel processing and applied mathematics. Springer, Berlin, pp 286–296
44. Ruan Y, Fuhry D, Parthasarathy S (2013) Efficient community detection in large networks using content and links. In: Proceedings of the 22nd international conference on world wide web, international world wide web conferences steering committee, pp 1089–1098
45. Satuluri V, Parthasarathy S (2009) Scalable graph clustering using stochastic flows: applications to community discovery. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 737–746
46. Satuluri V, Parthasarathy S, Ruan Y (2011) Local graph sparsification for scalable clustering. In: Proceedings of the 2011 international conference on management of data. ACM, pp 721–732
47. Soffer SN, Vázquez A (2005) Network clustering coefficient without degree-correlation biases. Phys Rev E 71(5):057101
48. Staudt CL, Meyerhenke H (2013) Engineering high-performance community detection heuristics for massive graphs. In: Proceedings of the 2013 42nd international conference on parallel processing. IEEE Computer Society, pp 180–189
49. Van Dongen SM (2000) Graph clustering by flow simulation. Ph.D. thesis, University of Utrecht
50. Von Luxburg U (2007) A tutorial on spectral clustering. Stat Comput 17(4):395–416
51. Xie J, Kelley S, Szymanski BK (2013) Overlapping community detection in networks: the state-of-the-art and comparative study. ACM Comput Surv (CSUR) 45(4):43
52. Yang B, Cheung WK, Liu J (2007) Community mining from signed social networks. IEEE Trans Knowl Data Eng 19(10):1333–1348
53. Yang J, Leskovec J (2013) Overlapping community detection at scale: a nonnegative matrix factorization approach. In: Proceedings of the sixth ACM international conference on web search and data mining. ACM, pp 587–596
54. Yang J, McAuley J, Leskovec J (2013) Community detection in networks with node attributes. In: 2013 IEEE 13th international conference on data mining (ICDM). IEEE, pp 1151–1156
55. Yang T, Jin R, Chi Y, Zhu S (2009) Combining link and content for community detection: a discriminative approach. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 927–936
56. Zachary WW (1977) An information flow model for conflict and fission in small groups. J Anthropol Res 33:452–473