SIAM Review
Vol. 65, No. 1, pp. 59–143

Flow-Based Algorithms for Improving Clusters: A Unifying Framework, Software, and Performance*

Kimon Fountoulakis†
Meng Liu‡
David F. Gleich‡
Michael W. Mahoney§

Abstract. Clustering points in a vector space or nodes in a graph is a ubiquitous primitive in statistical data analysis, and it is commonly used for exploratory data analysis. In practice, it is often of interest to "refine" or "improve" a given cluster that has been obtained by some other method. In this survey, we focus on principled algorithms for this *cluster improvement problem*. Many such cluster improvement algorithms are flow-based methods, by which we mean that operationally they require the solution of a sequence of maximum flow problems on a (typically implicitly) modified data graph. These cluster improvement algorithms are powerful, both in theory and in practice, but they have not been widely adopted for problems such as community detection, local graph clustering, semisupervised learning, etc. Possible reasons for this are the steep learning curve for these algorithms, the lack of efficient and easy-to-use software, and the lack of detailed numerical experiments on real-world data that demonstrate their usefulness. Our objective here is to address these issues. To do so, we guide the reader through the whole process of understanding how to implement and apply these powerful algorithms. We present a unifying fractional programming optimization framework that permits us to distill, in a simple way, the crucial components of all these algorithms. This also makes apparent similarities and differences among related methods. Viewing these cluster improvement algorithms via a fractional programming framework suggests directions for future algorithm development. Finally, we develop efficient implementations of these algorithms in our LocalGraphClustering Python package, and we perform extensive numerical experiments to demonstrate the performance of these methods on social networks and image-based data graphs.

Key words. improving clusters, flow-based methods, local graph clustering

MSC codes. 60J20, 91D30, 91C20, 62H30, 90C35

DOI. 10.1137/20M1333055

*Received by the editors April 22, 2020; accepted for publication (in revised form) January 4, 2022; published electronically February 9, 2023.

https://doi.org/10.1137/20M1333055

Funding: The work of the first author was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) through grants RGPIN-2019-04067 and DGECR-2019-00147. The work of the second and third authors was partially supported by NSF grants IIS-1546488 and CCF-1909528, the NSF Center for Science of Information STC through grant CCF-0939370, DOE grant DE-SC0014543, NASA, and the Sloan Foundation. The work of the fourth author was partially supported by ARO, DARPA, NSF via the CSoI STC, ONR, Cray, and Intel.

†School of Computer Science, University of Waterloo, Waterloo, ON, N2L3G1, Canada (kfountou@uwaterloo.ca).

‡Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA (liu1740@purdue.edu, dgleich@purdue.edu).

§ICSI and Department of Statistics, University of California at Berkeley, Berkeley, CA, USA (mmahoney@stat.berkeley.edu).

Contents

Part I. Introduction and Overview of Main Results.

1. Introduction. Clustering is the process of taking a set of data as input and returning meaningful groups of that data as output. The literature on clustering is tremendously and notoriously extensive (von Luxburg, Williamson, and Guyon, 2012; Ben-David, 2018); see also comments by Hand in the discussion of Friedman and Meulman (2004). It can seem that nearly every conceivable perspective on the clustering problem—from statistical to algorithmic, from optimization-based to information theoretic, from applications to formulations to implementations—that could be explored, has been explored. Applications of clustering are far too numerous to

discuss meaningfully, and they are often of greatest practical interest for "soft" downstream objectives such as those common in *exploratory data analysis*. Yet, despite comprehensive research into the problem, useful and surprising new results on clustering are still discovered on a regular basis (Kleinberg, 2002; Ackerman and Ben-David, 2008; Awasthi et al., 2015; Abbe, 2018).

Graph clustering is a special instance of the general clustering problem where the input is a graph, in this case, a set of nodes and edges, and the output is a meaningful grouping of the graph's nodes. The ubiquity of sparse relational data from internet-based applications to biology, from complex engineered systems to neuroscience, as well as new problems inspired by these domains (Newman, 2010; Easley and Jo, 2010; Brandes and Erlebach, 2005) (and within *SIAM Review* during the past decade (Estrada and Higham, 2010; Red et al., 2011; Grindrod and Higham, 2013; Liberti et al., 2014; Bienstock, Chertkov, and Harnett, 2014; Jia et al., 2015; Bertozzi and Flenner, 2016; Estrada and Hatano, 2016; Rombach et al., 2017; Fosdick et al., 2018; Fennell and Gleeson, 2019; Shi, Altafini, and Baras, 2019; Ehrhardt and Wolfe, 2019)), has precipitated a recent surge of graph clustering research (Newman, 2006; Leskovec et al., 2009; Eckles, Karrer, and Ugander, 2017). For instance, in graph and network models of complex systems, the *community detection* or *module detection* problem is a specific instance of the graph clustering problem in which one seeks to identify clusters that exhibit relationships distinctly different from other parts of the network. Consequently, there are now a large number of tools and techniques that generate clusters from graph data.

The tools and techniques we study in this survey arise from a different and complementary perspective. As such, they are designed to solve a different and complementary problem. The clustering problem itself is somewhat ill-defined, but one often applies it in practice while performing exploratory data analysis. That is, one uses a clustering algorithm to "play with" and "explore" the data, tweaking the clustering to see what insights about the data are revealed. Motivated by this, and the well-known fact that the output of even the best clustering algorithm is typically imperfectly suited to the downstream task of interest (for example, Carrasco et al. (2003) mentions "neither [...] seems to yield really good [...] clusterings of our dataset, so we have resorted to hand-built combinations"), we are interested in tools and techniques that seek to *improve* or *refine* a given cluster—or more generally a representative set of vertices—in a fashion that is computationally efficient, that yields a result with strong optimality guarantees, and that is useful in practice.

Somewhat more formally, here is the *cluster improvement problem*: Given a graph $G = (V, E)$ and a subset of vertices R that serve as a *reference cluster* (or *seed set*), find a nearby set S that results in an *improved cluster*. That is,

> when given as input a graph $G = (V, E)$ and a set $R \subset V$,
> a *cluster improvement algorithm* returns a set $S \subset V$,
> where S is in some sense "better" than R.

A very important point here is that both G and R are regarded as input to the cluster improvement problem. This is different from more traditional graph clustering, which typically takes only G as input, and it is a source of potential confusion. See Figure 1, which we explain in depth in section 1.1, for an illustration.

How to choose the set R, which is part of the input to a cluster improvement algorithm, is an important practical problem (akin to how to construct the input graph in more traditional graph clustering). It depends on the application of interest, and we will see several examples of it.

In the settings we will investigate in this survey, we will be (mainly) interested in graph conductance (which we will define formally in section 2.6) as the cluster quality metric. Thus, the optimization goal will be to produce a set S with smaller (i.e., better) conductance than R. Generally speaking, a set of small conductance in a graph is a hint at a bottleneck revealing an underlying cluster. While we focus on conductance, the techniques we review are more general and powerful. For example, these ideas, algorithms, and approaches can be adapted to other graph clustering objectives such as ratio cut (Lang and Rao, 2004), normalized cut (Hochbaum, 2013), and other closely related "edge counting" objective functions and scenarios (Veldt, Klymko, and Gleich, 2019; Veldt, Wirth, and Gleich, 2019). We return to the utility of conductance as an objective function to improve clusters, even those output from related objectives and algorithms, via an example in section 1.1.

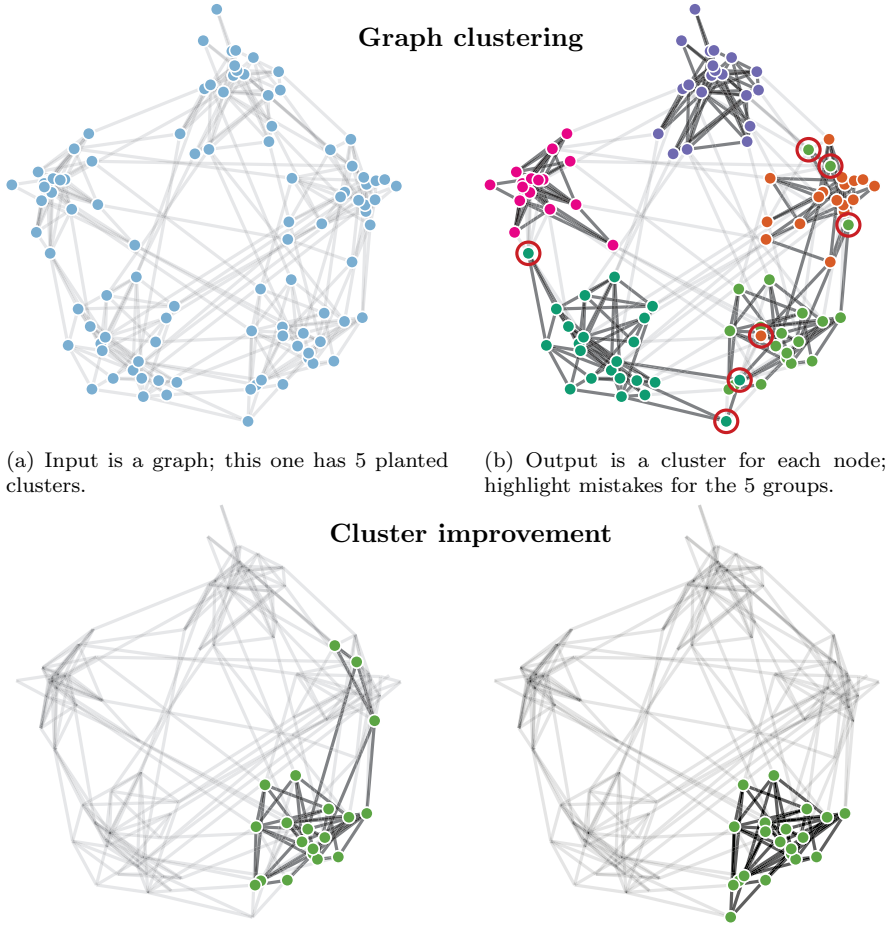
We define the precise improvement problems via optimization in subsequent sections. For now, we treat them as black-box algorithms and just explain how they might be used. These introductory examples use one of two algorithms, MQI (Lang and Rao, 2004) and LocalFlowImprove (Orecchia and Zhu, 2014), which we will study in depth. Both of these cluster improvement algorithms execute an intricate sequence of maximum flow (max-flow) or minimum cut (min-cut) computations on graphs derived from G and R. A technical difference between the two algorithms with important practical consequences is as follows:

> MQI always returns a set S of *exactly optimal* conductance *contained within the reference cluster* R; whereas
> LocalFlowImprove finds an improved cluster S with conductance *at least as good* as that found by MQI, *by both omitting vertices of* R *and adding vertices outside* R.

In addition to these two algorithms, we will also discuss the FlowImprove (Andersen and Lang, 2008) method in depth.

1.1. Cluster Improvement: Compared with Graph Clustering. To start, consider Figure 1, in which we consider a synthetic graph model called a stochastic block model. In our instance of the stochastic block model, we plant 5 clusters of 20 vertices. Edges between vertices in the same cluster occur at random with probability 0.3. Edges between vertices in different clusters occur at random with probability 0.0157. A popular algorithm for graph clustering is the Louvain method (Blondel et al., 2008). On this problem input instance, running the Louvain method often produces a clustering with a small number of errors (Aside 1). By using the LocalFlowImprove algorithm on each cluster returned by Louvain, we can directly refine the clusters output by the Louvain method (i.e., we can choose our input set R to be the output of some other method). This example involves running the improvement algorithm once for each cluster returned by the Louvain method. Doing so results in a perfectly accurate clustering for this instance. That said, the Louvain method is designed to *partition* the dataset and insists on a cluster for each node, whereas improving each cluster may result in some vertices being unassigned to a cluster or assigned to multiple clusters. Although this does not occur in this instance on the block model, it ought to be expected in general. There are a variety of ways to address this difference in

Aside 1. *For this particular example, there are ways of getting a completely accurate answer that involve rerunning the Louvain method or tweaking parameters. Our point is simply that we can easily improve existing clustering pipelines with flow-based improvement methods.*



(a) Input is a graph; this one has 5 planted clusters.

(b) Output is a cluster for each node; we highlight mistakes for the 5 groups.

(c) Input is a graph and a seed set of nodes.

(d) Output is an improved set of nodes.

Fig. 1 *Graph clustering (known as community detection in some areas) is a problem where the input is a graph and the output is a labeling or partition indicator for each node, indicating the group/cluster to which each node belongs. This is illustrated in* (a) *and* (b)*. Cluster improvement is different. In cluster improvement problems, the input is both a graph and a set of nodes, and the output is a set of nodes that is improved in some sense. As an example, in* (c)*, we show the input as the same graph from* (a) *along with one of the groups from* (b) *that has a few mistakes. The result of cluster improvement in* (d) *has no mistakes. See the replication details in the appendix.*

output given the domain specific usage. For instance, to reobtain a partition, one can create clusters of unassigned vertices and pick a single assignment out of the multiple assignments based on problem or application specific criteria.

For this example, we'd like to highlight the difference in objective functions between the modularity measure optimized by the Louvain algorithm and the conductance measure optimized by LocalFlowImprove. Despite differences in these objectives (modularity compared with conductance), many clustering objective functions are related in their design to balance boundary and size tradeoffs (i.e., isoperimetry). Consequently, exactly or optimally improving a related objective is likely to result in benefits for nearby measures. Moreover, conductance and modularity are indeed close

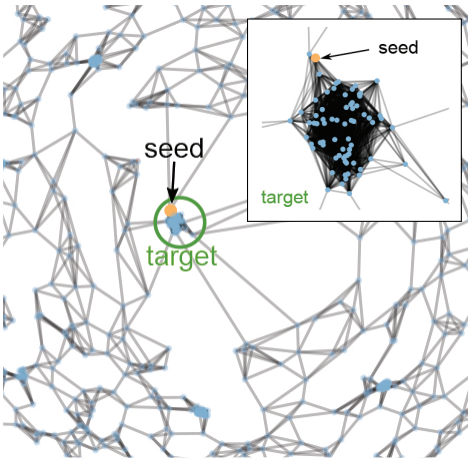
cousins, as is established either by how they make cut and volume tradeoffs (Gleich and Mahoney, 2016) or by relationships with Markov stability (Delvenne, Yaliraki, and Barahona, 2010). Thus, it is not surprising that LocalFlowImprove is able to assist Louvain, despite the difference in objectives. (Let us also note that flow-based algorithms can be designed around a variety of more general objective functions as well; see section 3.7.) Thus, this example mixes aspects that commonly arise in real-world uses: (i) the end goal (find the hidden structures), (ii) an objective function formulation of a related goal (optimize modularity), and (iii) an algorithmic procedure for that task (Louvain method). Given the output from (iii), the improvement algorithms produce an *exactly* optimal solution to a nearby problem that (in this case) captures exactly the true end goal (i).

1.2. Cluster Improvement: Compared with Seeded Graph Diffusion. Another common scenario in applied work with graphs is what we will call a target identification problem. In this setting, there is a large graph and we are given only one vertex, or a very small number of vertices, from a hidden target set. See Figure 2(a) for an illustration. Seeded graph diffusions are a common technique for this class of problems. In a seeded graph diffusion, the input is a seed node s and the output is a set of nearby graph vertices related to s (Zhu, Ghahramani, and Lafferty, 2003; Faloutsos, McCurley, and Tomkins, 2004; Zhou et al., 2004; Tong, Faloutsos, and Pan, 2006; Kloumann and Kleinberg, 2014). Arguably, the most well-known and widely applied of these seeded graph methods is seeded PageRank (Andersen, Chung, and Lang, 2006; Gleich, 2015). In essence, seeded PageRank problems identify related vertices as places where a random walk in the graph is likely to visit when it is frequently restarted at s.

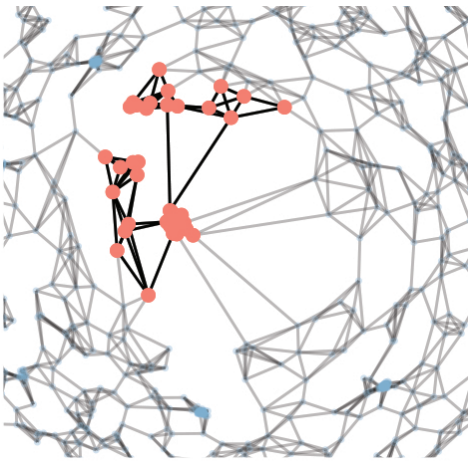
Cluster improvement algorithms are different than, but closely related to, *seeded graph diffusion* problems. This relationship is both formal and applied. It is related in a formal (and obvious) sense because seeded PageRank and its relatives correspond to an optimization problem that will also provably identify sets of small conductance (Andersen, Chung, and Lang, 2006). It is related in an applied sense for the following (important, but initially less obvious) reason: the improvement methods we describe are excellent choices to refine clusters produced by seeded PageRank and related Laplacian-based spectral graph methods (Lang, 2005; Fountoulakis et al., 2019c; Veldt, Gleich, and Mahoney, 2016). The basic reason for this is that spectral methods often exhibit a "leak" nearby a boundary. For instance, if a node at the boundary of an idealized target cluster is visited with a nontrivial probability from a random walk, then neighbors will also be visited with nontrivial probability. In particular, this means that such spectral methods tend to output clusters with larger conductance, more false positives (in terms of the target set), and sometimes fewer true positives as well.

An illustration of this *leaking out* of a spectral method is given in Figure 2. Here, we are using the algorithms to study a graph with a planted target cluster of 72 vertices in the center of a much larger 3000 node graph. If we run a seeded PageRank algorithm from a node near the boundary of the target, then the result set expands too far beyond the target cluster (Figure 2(b)). If we then run the MQI cluster improvement method on the output of seeded PageRank, then we accurately identify the target cluster alone (Figure 2(c)). Likewise, if we simply expand the seed node into a slightly larger set by adding all of the seed's neighbors, and we then perform a single run of the LocalFlowImprove method, then we will accurately identify this set.

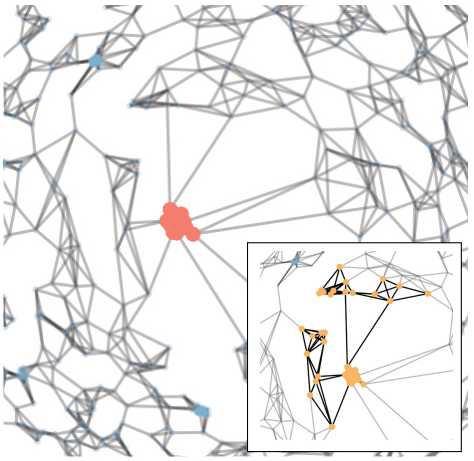
1.3. Cluster Improvement: Compared with Image Segmentation. Our final introductory example is given in Figure 3, and it illustrates these improvement al-



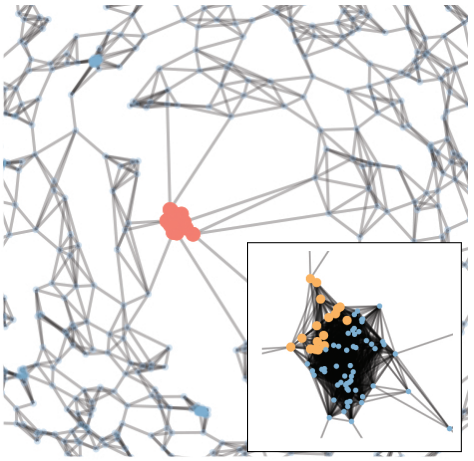
(a) Our target and a seed node (orange)



(b) The seeded PageRank result (red)



(c) MQI-based improvement (red) of the seeded PageRank result set (inset orange nodes)



(d) LocalFlowImprove result (red) on a one-step neighborhood of the seed (inset orange nodes)

Fig. 2 *Cluster improvement with MQI (Lang and Rao,* 2004*) and LocalFlowImprove (Orecchia and Zhu,* 2014*) on a large graph. We show a piece of a larger graph with a target cluster in the middle of* (a) *and an expanded view of the target and seed in the inset of* (a)*. If we run a seeded PageRank-based method to search for a cluster near the seed, then the result* leaks out *into the rest of the graph and fails to capture the boundary of the cluster, as shown in* (b)*. If, using the seeded PageRank result as the reference set R (shown in orange in the inset of* (c)*), we run MQI, then we accurately identify the target in* (c) *in red. Likewise, if, using the one-step neighborhood of the seed as R (shown in orange in the inset of* (d)*), we run LocalFlowImprove, then we also accurately identify the target* (d) *in red. See Appendix* A *for details.*

gorithms in the context of image segmentation. Here, an input image is translated into a weighted graph through a standard technique. The goal of that technique is to ensure that similar regions of the image appear as clusters in the resulting graph; this standard process is described formally in Appendix B. On this graph representing an

image, the target set identification problem from section 1.2 yields an effective image segmentation procedure, albeit with a much larger set of seed nodes.

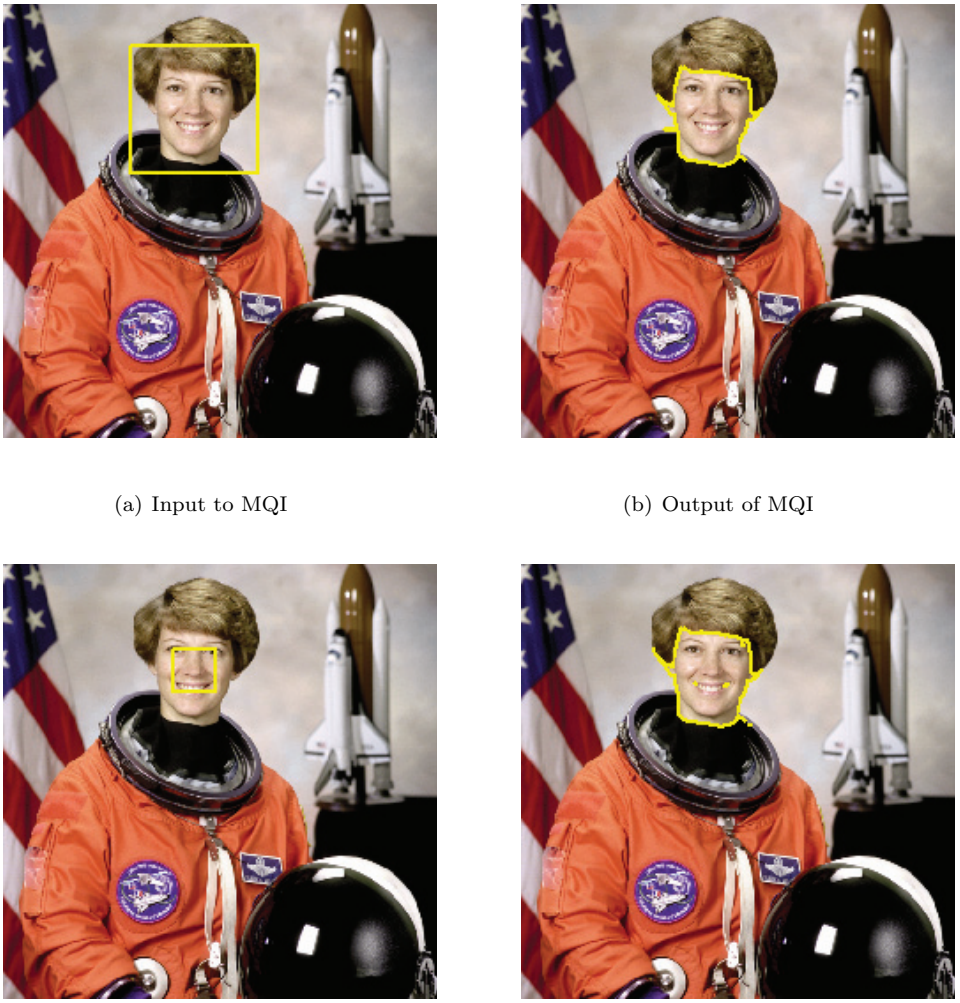
We focus on the face of the astronaut Eileen Marie Collins (a retired NASA astronaut and United States Air Force colonel) (Wikipedia, 2021) as our target set. Figure 3(a) shows a superset of the face. When given as the input set to the MQI cluster improvement method (which, recall, always returns a subset of the input), the result closely tracks the face, as is shown in Figure 3(b). Note that there are still a small number of false positives around the face—see the region left of the neck below the ear—but the number of false positives decreases dramatically with respect to the input. Similarly, when given a subset of the face, we can use LocalFlowImprove (which, recall, can expand or contract the input seed set) to find most of it. We present in Figure 3(c) the input cluster to LocalFlowImprove, which is clearly a subset of the face; the output cluster for LocalFlowImprove is shown in Figure 3(d), which again closely tracks the face with a few false negatives around the mouth.

ASIDE 2. *These image segmentation examples are used to illustrate properties of the algorithms that are difficult to visualize on natural graphs. They are not intended to represent state-of-the-art segmentation procedures.*

1.4. Overview and Summary. One challenge with the flow-based cluster improvement literature is that (so far) it has lacked the simplicity of related spectral methods and seeded graph diffusion methods like PageRank (Gleich, 2015; Zhu, Ghahramani, and Lafferty, 2003; Faloutsos, McCurley, and Tomkins, 2004; Zhou et al., 2004; Tong, Faloutsos, and Pan, 2006; Kloumann and Kleinberg, 2014). These spectral methods are often easy to explain in terms of random walks, Markov chains, linear systems, and intuitive notions of *diffusion*. Instead, the flow-based literature involves complex and seemingly arbitrary graph constructions that are then used, almost like magic (at least to researchers and downstream scientists not deeply familiar with flow-based algorithms), to show impressive theoretical results. Our goal here is to pull back the curtain on these constructions and provide a unified framework based on a class of optimization methods known as fractional programming.

The connection between flow-based local graph clustering and fractional programming is not new; e.g., Lang and Rao (2004) cite one relevant paper (Gallo, Grigoriadis, and Tarjan, 1989). Both Lang and Rao (2004) and Andersen and Lang (2008) mention binary search for finding optimal ratios akin to root-finding. Hochbaum (2010) was the first to develop a general framework of root-finding algorithms for global flow-based fractional programming problems. However, specialization of these results to the FlowImprove problem requires special treatment which is not discussed in Hochbaum (2010). That said, our purpose in using these connections is that they make the methods simpler to understand. Thus, we will make the connection extremely clear, and we will demonstrate that our fractional programming optimization perspective unifies *all* existing flow-based cluster improvement methods. *Indeed, it is our hope that this perspective will be used to develop new theoretically principled and practically useful methodologies.*

1.5. Reproducible Software: The LocalGraphClustering Package. In addition to this detailed and unified explanation of the flow-based improvement methods, we have implemented these algorithms in a software package with a user-friendly Python interface. The software is called LocalGraphClustering (Fountoulakis et al., 2019b) (with which, in addition to implementing flow improvement methods that we



(a) Input to MQI

(b) Output of MQI

(c) Input to LocalFlowImprove

(d) Output of LocalFlowImprove

Fig. 3 *Illustration of cluster improvement with MQI (Lang and Rao,* 2004*) and LocalFlowImprove (Orecchia and Zhu,* 2014*) on an image. In Figure* 3(a)*, we show the input set of nodes to MQI. The set of nodes consists of the pixels inside the yellow square. Note that MQI looks for good clusters within the input square, and the target cluster is the face of Eileen Marie Collins (a retired NASA astronaut and United States Air Force colonel) (Wikipedia,* 2021*). In Figure* 3(b)*, we show the output, which demonstrates that MQI-based cluster improvement decreases the number of false positives. In Figure* 3(c)*, we show the input set of nodes to LocalFlowImprove. The set of nodes consists of the pixels inside the yellow square. Note that LocalFlowImprove looks for good clusters around the region of the input square and the target cluster is the face of the Eileen Marie Collins. In Figure* 3(d)*, we show the output, which demonstrates that LocalFlowImprove-based cluster improvement increases the number of true positives. See Appendix* A *for details.*

review here, we implement spectral diffusion methods for clustering, methods for multilabel classification, network community profiles, and network drawing methods). As an example of the use of this package, running seeded PageRank followed by MQI for the results shown in Figure 2 is as simple as the following code:

```
import localgraphclustering as lgc             # load the package
G = lgc.GraphLocal("geograph-example.edges")   # load the graph
seed = 305                                     # set the seed and compute
R,cond = lgc.spectral_clustering(G,[seed],method='l1reg') # seeded PageRank
S,cond = lgc.flow_clustering(G,R,method='mqi') # improve with MQI
```

This software also enables us to explore a number of interesting applications of flow-based cluster improvement algorithms that demonstrate uses beyond simply improving the conductance of sets. The implementation of the methods scales to graphs with billions of edges when used appropriately. In this survey, we explore graphs with up to 117 million edges (section 9.4).

This package is useful in general. For reproducibility we also provide code that reproduces all the experiments that are presented in this survey.

1.6. Outline. There are three major parts to our survey, and these are designed to be relatively modular to enable one to read parts separately (e.g., to focus on the theoretical results or the empirical results).

In the first part, we introduce the fundamental concepts and techniques, both informally as in this introduction and formally through our notation and fractional programming sections (sections 2 and 3). In particular, we introduce graph cluster metrics such as conductance in section 2.6. We also introduce fundamental ideas related to *local graph computations* in section 2.7, which discusses the distinction between strongly and weakly local graph algorithms. These ideas are then used to explain the precise objective functions and settings for flow-based cluster improvement algorithms in section 3. This part continues with an overview of how these methods fit into the broader literature of graph-based algorithms (section 4), and it includes a brief discussion of other scenarios where *max-flow* and *min-cut* algorithms are used as a fundamental computational primitive (section 4.5), as well as infinite-dimensional analogues to these ideas (section 4.7). We also include a number of ideas that show how the methods generalize beyond using conductance.

In the second part, we provide the technical core of the survey. We begin our description of the details of the methods with a review of concepts from maximum flow and minimum cuts (section 5). In particular, this section has a careful derivation of these problems as duals in terms of linear programs. The next three sections, sections 6 to 8, cover the three algorithms that we use in the experiments: MQI, FlowImprove, and LocalFlowImprove. For each algorithm, we provide a thorough discussion on how to define each step of the algorithm. On a high level, these algorithms require at each iteration the solution of a max-flow problem. However, in order to actually implement these methods one requires the construction of a locally modified version of the given graphs.

In the final part, we provide an extensive empirical evaluation and demonstration of these algorithms (section 9). This is done in the context of a number of datasets where it is possible to illustrate clearly and easily the benefits of these techniques. Examples in this evaluation include images, as we saw in the introduction, as well as road networks, social networks, and nearest neighbor graphs that represent relationships among galaxies. This section also includes experiments on graphs with up to 117 million edges. We also describe strategies to generate local network visualizations

from these local graph clustering methods that highlight characteristic differences in how the flow-based methods treat networks.

In addition, we provide an appendix with full reproducibility details for all of the figures and tables (Appendices A and B). These include references to specific Python notebooks for replication of the experiments.

2. Notation, Definitions, and Terminology. We begin by reviewing specific mathematical assumptions, notation, and terminology that we will use in what follows. To start, we use the following standard notation:

$\mathbb{Z}$	denotes the set of integer numbers,
$\mathbb{R}$	denotes the set of real-valued numbers,
$\mathbb{R}_+$	denotes the set of real-valued nonnegative numbers,
$\mathbb{R}^n$	denotes the set of real-valued vectors of length n,
$\mathbb{R}^{n\times n}$	denotes the set of real-valued $n \times n$ matrices,
$\mathbb{R}^n_+$	denotes the set of real-valued nonnegative vectors of length n, and
$\mathbb{R}^{n\times n}_+$	denotes the set of real nonnegative $n \times n$ matrices.

2.1. Graph Notation. Given a graph $G = (V, E)$, we let V denote the set of nodes and E denote the set of edges. We assume an undirected, weighted graph throughout, although some of the constructions and concepts involved in a flow computation are often best characterized through directed graphs. (See also Aside 3.) For an unweighted graph, everything we do will be equivalent to assigning an edge weight of 1 to all edges. We also assume that the given graphs have no self-loops.

The cardinality of the set V is denoted by n, i.e., there are n nodes, and we assume that the nodes are arbitrarily ordered from 1 to n. Therefore, we can write $V := \{1, 2, \ldots, n\}$. We use v_i to denote node i, and when it is clear, we will use i to denote that node. We assume that the edges E in the graph are arbitrarily ordered. The cardinality of the set E is denoted by m, i.e., there are m edges. We will use e_{ij} to denote an edge. Also, if a node j is a neighbor of node i, we denote this relationship by $j \sim i$.

Aside 3. *Our techniques would extend to any clustering function on directed graphs that defines a hypergraph using techniques from Benson, Gleich, and Leskovec* (2016) *based on motif enumeration. For adaptations of these techniques to hypergraphs, see Veldt, Benson, and Kleinberg* (2020; 2022) *for some examples.*

A path is a sequence of edges which connect a sequence of distinct vertices. A connected component is a subset of nodes such that there exists a path between any pair of nodes in that subset.

We frequently work with subsets of vertices. Let $S \subseteq V$, for example. Then $\bar{S}$ denotes the complement of subset $S \subseteq V$, formally, $\bar{S} = \{v \in V \mid v \notin S\}$. The notation ∂S represents the node-boundary of the set S; formally, it denotes the set of nodes that are in $\bar{S}$ and are connected with an edge to at least one node in S. In set notation, we have $\partial S = \{v,$ where $v \in \bar{S}$ and there exists $(u, v) \in E$ with $u \in S\}$.

2.2. Matrices and Vectors for Graphs. Here, we define matrices that can be used to define models and objective functions on graph data. They can also provide a compact way to understand and describe algorithms that operate on graphs.

The *adjacency matrix* $A \in \{0, 1\}^{n\times n}$ (or $\in \mathbb{R}^{n\times n}_+$ if the graph is weighted) provides perhaps the most simple representation of a graph using a matrix. In A, row i corresponds to node i in the graph, and element A_{ij} is nonzero if and only if nodes i and j are connected with an edge in the given graph. The value of A_{ij} is the edge weight for a weighted graph, or simply 1 for an unweighted graph. Since we are

working with undirected graphs, the adjacency matrix is symmetric, i.e., $A_{ij} = A_{ji}$, where A_{ij} is the element at the ith row and jth column of matrix A.

The *diagonal weighted degree matrix* $D \in \mathbb{Z}_+^{n\times n}$ (or $\in \mathbb{R}_+^{n\times n}$ if the graph is weighted) is a matrix that stores the degree information for every node. The element D_{ii} is the sum of weights of the edges of node i, i.e., $D_{ii} := \sum_{j\in V:j\sim i} A_{ij}$; and off-diagonal elements, i.e., D_{ij} for $i \neq j$, equal zero.

The *degree vector* is defined as $d = \mathrm{diag}(D)$, where $\mathrm{diag}(\cdot)$ takes as input a vector or a matrix and returns, respectively, a diagonal matrix with the vector in the diagonal or a vector with diagonal elements of a matrix.

The *edge-by-node incidence matrix* $B \in \{0, -1, 1\}^{m\times n}$ (where, recall, n is the number of nodes, and m is the number of edges) is often used to measure differences among nodes. Each row of this matrix represents an edge, and each column represents a node. For example, row k in B represents the kth edge in the graph (arbitrarily ordered) that corresponds (say) to nodes i and j in the graph. Row k in B then has exactly two nonzero elements, -1 for the source of the edge and 1 for the target of the edge, at the i and j positions, respectively. If the graph is undirected, then we can arbitrarily choose which node is the source and which node is the target on an edge, without loss of generality. Note that because we assume no self-loops, the incidence matrix contains the full information about the edges of the graph.

The *diagonal edge-weight or edge-capacity matrix* $C \in \mathbb{R}_+^{m\times m}$ is a diagonal matrix where each diagonal element corresponds to the weight of an edge in the graph. This matrix is the identity for an unweighted graph. For example, the kth diagonal element corresponds to the weight of the kth edge in the graph.

The *Laplacian matrix* $L \in \mathbb{Z}^{n\times n}$ (or $\in \mathbb{R}^{n\times n}$ if the graph is weighted) is defined as $L = D - A$ or, equivalently, $L = B^T CB$.

Vectors of all-ones and all-zeros, denoted 1_n and 0_n, respectively, are column vectors of length n. If the dimensions of each vector are clear from the context, then we omit the subscript. The *indicator vector* $\mathbf{1}_i$ is a column vector that is equal to 1 at the ith index and zero elsewhere. If the indicator is used with a node, then the length of the vector $\mathbf{1}_i$ is n. For an edge, its length is m.

If S is a subset of nodes or a subset of indices and A is any matrix, e.g., the adjacency matrix, then A_S is a submatrix of A that corresponds to the rows and columns with indices in S. Likewise, $\mathbf{1}_S$ is a column vector with ones in entries for S. These indicator vectors have length n.

2.3. Vector Norms. We denote the vector 1-norm by $\|x\|_1 = \sum_i |x_i|$ and the 2-norm by $\|x\|_2 = \sqrt{\sum_i (x_i)^2}$. We will use these norms to measure differences among nodes that are represented in a vector x, i.e., every node corresponds to an element in vector x. For example, $\|Bx\|_1 = \sum_{e_{ij}\in E} |x_i - x_j|$ is the sum of differences among node representations in x. In the case of weighted graphs, this can be generalized to $\|Bx\|_{C,1} = \sum_{e_{ij}\in E} C_{e_{ij}}|x_i - x_j| = \sum_{e_{ij}\in E} A_{ij}|x_i - x_j|$. For the 2-norm, we have $\|Bx\|_{C,2}^2 = \sum_{e_{ij}\in E} C_{e_{ij}}(x_i - x_j)^2 = \sum_{e_{ij}\in E} A_{ij}(x_i - x_j)^2$.

2.4. Graph Cuts and Volumes Using Set and Matrix Notation. Much of our discussion will move fluidly between set-based descriptions and matrix-based descriptions. Here, we give a simple example of how this works in terms of a graph cut and volume of a set.

Graph Cut. We say that a pair of complement sets $(S, \bar{S})$, where $S \subseteq V$, is a *global graph partition* of a given graph with node set V. Given a partition $(S, \bar{S})$, the *cut of the partition* is the sum of weights of edges between S and $\bar{S}$, which can be denoted

by either

$$\text{cut}(S, \bar{S}) = \sum_{i \in S, j \in \bar{S}} A_{ij} \quad \text{or} \quad \text{cut}(S) = \sum_{i \in S, j \in \bar{S}} A_{ij}. \tag{2.1}$$

Instead of using set notation to denote a partition of the graph, i.e., $(S, \bar{S})$, we can use indicator vector notation $x = \mathbf{1}_S \in \{0,1\}^n$ to denote a partition. In this case, the cut of the partition is

$$\text{cut}(S, \bar{S}) = \sum_{i,j} A_{ij} |x_i - x_j| = \|B\mathbf{1}_S\|_{C,1}. \tag{2.2}$$

Note that both expressions are symmetric in terms of S and $\bar{S}$.

Graph Volume. The *volume of a set of nodes* S is equal to the sum of the degrees of all nodes in S, i.e.,

$$\text{vol}(S) = \sum_{i \in S} d_i. \tag{2.3}$$

We will use the notation $\text{vol}(G)$ to denote the *volume of the graph*, which is equal to $\text{vol}(V)$. Using this definition and our matrix definitions above, we have that the *volume of a subset of nodes* is $\text{vol}(S) = \mathbf{1}_S^T d$.

2.5. Relative Volume. FlowImprove and LocalFlowImprove formulations are simpler to explain by introducing the idea of *relative volume*. The *relative volume* of S with respect to R and κ is

$$\text{rvol}(S; R, \kappa) = \text{vol}(S \cap R) - \kappa \, \text{vol}(S \cap \bar{R}). \tag{2.4}$$

The relative volume is a very useful concept that we will use to define the objective functions of the local flow-based problems MQI, FlowImprove, and LocalFlowImprove. The purpose of the relative volume is to measure the volume of the intersection of S with the input seed set nodes R, while penalizing the volume of the intersection of S with the complement $\bar{R}$. This is important when we define the objective functions of MQI, FlowImprove, and LocalFlowImprove, since we want to penalize sets S that have little intersection with R and high intersection with $\bar{R}$. This makes sense, since in local flow-based improvement methods the goal is often to improve the input set R; thus we want the output S of a method to be "related" to R more than to $\bar{R}$.

2.6. Cluster Quality Metrics. Here, we discuss scores that we use to evaluate the quality of a cluster. For all of these measures, *smaller values correspond to better clusters*, i.e., they correspond to a cluster of higher quality.

Conductance. The *conductance function* is defined as the ratio between the number of edges that connect the two sides of the partition $(S, \bar{S})$ and the minimum "volume" of S and $\bar{S}$:

$$\phi(S) = \frac{\text{cut}(S)}{\min(\text{vol}(S), \text{vol}(\bar{S}))}.$$

A set of minimal conductance is a fundamental bottleneck in a graph. For example, small conductance in a set is often interpreted as an information bottleneck revealing community or module structure, or (relatedly) as a bottleneck to the mixing of random walks on the graph. Note that conductance values are always between 0 and 1, and they can be interpreted as probabilities. (Formally, this is the probability that a random walk moves between S and $\bar{S}$ in a single prescribed step after the walk has fully mixed.)

Normalized Cuts. The *normalized cut function* is a related notion that provides a score that is often used in image segmentation problems (Shi and Malik, 2000), where a graph is constructed from a given image and the objective is to partition the graph into two or more segments. In the case of a bipartition problem, the normalized cut score reduces to

$$\mathrm{ncut}(S) = \frac{\mathrm{cut}(S)}{\mathrm{vol}(S)} + \frac{\mathrm{cut}(\bar{S})}{\mathrm{vol}(\bar{S})}.$$

The normalized cut and conductance scores are related, in that $\phi(S) \leq \mathrm{ncut}(S) \leq 2\phi(S)$. There is a related concept, called ncut' (Sharon et al., 2006; Hochbaum, 2010) that just measures the cut to volume ratio for a single set $\mathrm{ncut}'(S) = \mathrm{cut}(S)/\mathrm{vol}(S)$. Observe that this is equal to $\phi(S)$ for any set with less than half of the volume.

Expansion. The *expansion function* or *expansion score* is defined as the ratio between the number of edges that connect the two sides of the partition $(S, \bar{S})$ and the minimum "size" of S and $\bar{S}$:

$$\tilde{\phi}(S) = \frac{\mathrm{cut}(S)}{\min(|S|, |\bar{S}|)}.$$

Compared to the conductance score, which uses the volume (related to number of edges) of the sets S and $\bar{S}$ in the denominator, the expansion score counts the number of nodes in S or $\bar{S}$. This has the property that the expansion score is less affected by high degree nodes. Similarly to conductance, smaller expansion scores correspond to better clusters. However, these values are not necessarily between 0 and 1.

ASIDE 4. *Our definition of expansion used here is sometimes used as the definition for sparsity. The literature is not entirely consistent on these terms.*

Sparsity. The *sparsity measure* of a set is a topic that arises often in theoretical computer science. It is closely related to expansion, but measures the fraction of edges that exist in the cut compared to the total possible number:

$$\psi(S) = \frac{\mathrm{cut}(S)}{|S||\bar{S}|}.$$

This value is always between 0 and 1. Also, $\tilde{\phi}(S) \leq n\psi(S) \leq 2\tilde{\phi}(S)$ because $n\psi(S) = \frac{\mathrm{cut}(S)}{|\bar{S}|} + \frac{\mathrm{cut}(\bar{S})}{|\bar{S}|}$. Hence, sparsity is a scaled measure akin to normalized cut.

Ratio Cut. The *ratio cut function* provides a score that is often used in data clustering problems, where a graph is constructed by measuring similarities among the data and the objective is to partition the data into multiple clusters (Hagen and Kahng, 1992). In the case of the bipartition problem, the ratio cut score reduces to

$$\mathrm{rcut}(S) = \frac{\mathrm{cut}(S)}{|S|}.$$

Observe that the ratio cut and expansion scores are related, in the sense that the latter is equal to the former if the input set of nodes S has cardinality less than or equal to $n/2$. The ratio cut was popularized due to its importance in image segmentation problems (Felzenszwalb and Huttenlocher, 2004). Usually, this ratio is minimized by performing a spectral relaxation (von Luxburg, 2007).

2.7. Strongly and Weakly Local Graph Algorithms. Local graph algorithms and locally biased graph algorithms are the "right" setting in which to discuss cluster improvement algorithms on large-scale data graphs. For the purposes of this survey, there are two key types of (related but quite distinct) local graph algorithms:

- **Strongly local graph algorithms.** These algorithms take as input a graph G and a reference cluster of vertices R, and they have a runtime and resource usage that only depend on the size of the reference cluster R (or the output S, but not the size of the entire graph G).
- **Weakly local graph algorithms.** These algorithms take as input a graph G and a reference cluster of vertices R, and they return an answer whose size will depend on R, but whose runtime and resource usage may depend on the size of the entire graph G (as well as the size of R).

That is, in both cases, one wants to find a good or better cluster near R, and in both cases one outputs a small cluster S that is near R, but in one case the runtime of the algorithm is independent of the size of the graph G, while in the other case the runtime depends on the size of G. For more about local and locally biased graph algorithms, we recommend Gleich and Mahoney (2016), Fountoulakis, Gleich, and Mahoney (2017), and also Mahoney, Orecchia, and Vishnoi (2012) and Lawlor, Budavári, and Mahoney (2016b,a) for overviews.

It is easy to quantify the size of the output S as being small, but, in general, the *locality of an algorithm*, i.e., how many nodes/edges are touched at intermediate steps, may depend on how the graph is represented. We typically assume something akin to an adjacency list representation that enables

- constant time access to a list of neighbors; and
- constant or nearly constant (e.g., $O(\log |V|)$ time access to an arbitrary edge.

Moreover, the cost of building this structure is *not counted* in the runtime of the algorithm, e.g., since it may be a one-time cost when the graph is stored. Note that, in addition to a reference cluster R, these algorithms could also take information about vertices in a reference set, such as a vector of values.

The importance of these characterizations and this discussion is as follows:

for strongly local graph algorithms

the runtime is independent of the size of the graph.

In particular, this means that the algorithm does not even touch all of the nodes of the graph G. This makes a strongly local graph algorithm an extremely useful tool for studying large data graphs. For instance, in Figure 2, none of the algorithms used information from more than about 500 vertices of the total 3000 vertices of the graph, and this result wouldn't have changed at all if the entire graph was 3 million vertices (or more, as in Shun et al. (2016)).

To contrast with strongly local graph algorithms, most graph and mesh partitioning tools—and even the improved and refined variations—are global in nature. In other words, the methods take as input a graph, and the output of the methods is a global partitioning of the entire graph. In particular, this means that the methods have runtime which depends on the size of the whole graph. This makes it *very* challenging to apply these methods to even moderately large graphs.

3. Main Theoretical Results: Flow-Based Cluster Improvement and Fractional Programming Framework. In this section, we will introduce and discuss the fractional programming problem and its relevance to flow-based cluster improvement. The motivation is that work on cluster improvement algorithms has thus far proceeded largely on a case-by-case basis, but as we will describe, fractional programming is a

class of optimization problems that provides a way to generalize and unify existing cluster improvement algorithms.

3.1. Cluster Improvement Objectives and Their Properties. For the problem of conductance-based cluster improvement, the three methods we consider exactly optimize the following objective functions:

$$\begin{aligned}
\text{MQI:} \quad & \underset{S \subset V}{\text{minimize}} \ \frac{\operatorname{cut}(S)}{\operatorname{vol}(S)} \\
& \text{subject to} \ \ S \subseteq R, \\
\text{FlowImprove:} \quad & \underset{S \subset V}{\text{minimize}} \ \frac{\operatorname{cut}(S)}{\operatorname{rvol}(S; R, \operatorname{vol}(R)/\operatorname{vol}(\bar{R}))} \\
& \text{subject to} \ \ \operatorname{rvol}(S; \ldots) > 0, \\
\text{LocalFlowImprove}(\delta): \ & \underset{S \subset V}{\text{minimize}} \ \frac{\operatorname{cut}(S)}{\operatorname{rvol}(S; R, \operatorname{vol}(R)/\operatorname{vol}(\bar{R}) + \delta)} \\
\delta \geq 0 \quad & \text{subject to} \ \ \operatorname{rvol}(S; \ldots) > 0.
\end{aligned}$$

The constraint $\operatorname{rvol}(S; \ldots) > 0$ simply means that we only consider sets where the denominator is positive (we omit repeating all the parameters from the denominator for simplicity). Because we are minimizing over discrete sets, there is not a closure problem with the resulting strict inequality ($\operatorname{rvol}(S, \ldots) > 0$), so these are all well-posed.

Recall that $\operatorname{rvol}(S; R, \kappa) = \operatorname{vol}(S \cap R) - \kappa \operatorname{vol}(S \cap \bar{R})$. This definition implies that sets S such that $\operatorname{rvol}(S; R, \operatorname{vol}(R)/\operatorname{vol}(\bar{R})) \leq 0$ cannot be optimal solutions for FlowImprove, and that even fewer sets can be optimal for LocalFlowImprove. On the other hand, note that LocalFlowImprove(δ) interpolates between FlowImprove ($\delta = 0$) and MQI ($\delta = \infty$) because when δ is sufficiently large, then the term $\operatorname{vol}(S \cap \bar{R})$ that arises in rvol must be 0 in order for the set S to be feasible for the nonnegative rvol constraint. In fact, if $\delta > \operatorname{vol}(R)(1 - 1/\operatorname{vol}(\bar{R}))$, then positive denominators alone will require $S \subset R$.

To better understand the connections among these three objectives, we begin by stating a simple property of them. The following theorem states that conductance gets *smaller*, i.e., *better*, as we move from MQI to LocalFlowImprove to FlowImprove.

THEOREM 3.1. *Let G be an undirected, connected graph with nonnegative weights. Let $R \subset V$ satisfy $\operatorname{vol}(R) \leq \operatorname{vol}(\bar{R})$, where $\bar{R}$ is the complement of R. Let S_{MQI}, S_{FI}, and S_{LFI} be the optimal solutions of the MQI, FlowImprove, and LocalFlowImprove(δ) objectives, respectively. If the solutions of FlowImprove and LocalFlowImprove satisfy $\operatorname{vol}(S_{FI}) \leq \operatorname{vol}(\bar{S}_{FI})$ and $\operatorname{vol}(S_{LFI}) \leq \operatorname{vol}(\bar{S}_{LFI})$ (that is, the solution set is on the small side of the cut), then, for any $\delta \geq 0$ in LocalFlowImprove, we have that*

$$\phi(S_{FI}) \leq \phi(S_{LFI}) \leq \phi(S_{MQI}).$$

Proof. The first result, that $\phi(S_{\text{LFI}}) \leq \phi(S_{\text{MQI}})$, is a simple and useful exercise we repeat from Veldt, Gleich, and Mahoney (2016, Theorem 4). Note that if $S \subseteq R$, then $\phi(S) = \frac{\operatorname{cut}(S)}{\operatorname{rvol}(S;R,\kappa)}$ for any κ. Now, note that for all rvol terms in the LocalFlowImprove(δ) objective with $\delta > 0$, we have $\kappa \geq \operatorname{vol}(R)/\operatorname{vol}(\bar{R})$. Moreover, solutions are constrained to only consider sets where rvol is positive. Thus, for the value of κ used in LocalFlowImprove, and also any positive κ, we have

$$\phi(S_{\text{LFI}}) = \frac{\operatorname{cut}(S_{\text{LFI}})}{\operatorname{vol}(S_{\text{LFI}})} \leq \frac{\operatorname{cut}(S_{\text{LFI}})}{\operatorname{vol}(S_{\text{LFI}}) - \kappa \operatorname{vol}(S_{\text{LFI}} \cap \bar{R})} \leq \frac{\operatorname{cut}(S_{\text{LFI}})}{\operatorname{rvol}(S_{\text{LFI}}; R, \kappa)}.$$

Table 1 *Characteristics of the MQI, FlowImprove, and LocalFlowImprove methods.*

Method	Strongly local	Explores beyond R	Easy to implement	Section
MQI	✓		✓	Section 6
FlowImprove		✓	✓	Section 7
LocalFlowImprove	✓	✓		Section 8

Next, note that for the chosen setting of κ, we have that $\mathrm{rvol}(S; R, \kappa) > 0$ for all $S \subseteq R$. Thus, we have

$$\phi(S_{\mathrm{LFI}}) \leq \underset{S \subseteq R}{\text{minimum}} \frac{\mathrm{cut}(S)}{\mathrm{rvol}(S; R, \kappa)} = \underset{S \subseteq R}{\text{minimum}}\, \phi(S) = \phi(S_{\mathrm{MQI}}).$$

This shows that both LocalFlowImprove and FlowImprove give better conductance sets than MQI.

For the second inequality, we use an alternative characterization of LocalFlowImprove as discussed in Orecchia and Zhu (2014). LocalFlowImprove(δ) is equivalent to solving the following optimization problem for some constant C:

$$\begin{array}{ll} \underset{S \subset V}{\text{minimize}} & \frac{\mathrm{cut}(S)}{\mathrm{rvol}(S; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))} \\ \text{subject to} & \frac{\mathrm{vol}(S \cap R)}{\mathrm{vol}(S)} \geq C, \mathrm{rvol}(S; \ldots) > 0. \end{array}$$

FlowImprove solves the same problem without the constraint involving C. Then we have

$$\frac{\mathrm{cut}(S_{FI})}{\mathrm{rvol}(S_{FI}; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))} \leq \frac{\mathrm{cut}(S_{LFI})}{\mathrm{rvol}(S_{LFI}; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))},$$

$$\frac{\mathrm{cut}(S_{FI})}{\mathrm{cut}(S_{LFI})} \leq \frac{\mathrm{rvol}(S_{FI}; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))}{\mathrm{rvol}(S_{LFI}; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))}.$$

If $\phi(S_{FI}) > \phi(S_{LFI})$, we have

$$\frac{\mathrm{cut}(S_{FI})}{\mathrm{cut}(S_{LFI})} > \frac{\mathrm{vol}(S_{FI})}{\mathrm{vol}(S_{LFI})}.$$

Thus,

$$\frac{\mathrm{rvol}(S_{FI}; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))}{\mathrm{rvol}(S_{LFI}; R, \mathrm{vol}(R)/\mathrm{vol}(\bar{R}))} \geq \frac{\mathrm{cut}(S_{FI})}{\mathrm{cut}(S_{LFI})} > \frac{\mathrm{vol}(S_{FI})}{\mathrm{vol}(S_{LFI})}.$$

If we now substitute the definition of rvol and $\mathrm{vol}(S \cap \bar{R}) = \mathrm{vol}(S) - \mathrm{vol}(S \cap R)$,

$$\frac{(1 + \mathrm{vol}(R)/\mathrm{vol}(\bar{R})) \cdot \mathrm{vol}(S_{FI} \cap R) - \mathrm{vol}(R)/\mathrm{vol}(\bar{R}) \cdot \mathrm{vol}(S_{FI})}{(1 + \mathrm{vol}(R)/\mathrm{vol}(\bar{R})) \cdot \mathrm{vol}(S_{LFI} \cap R) - \mathrm{vol}(R)/\mathrm{vol}(\bar{R}) \cdot \mathrm{vol}(S_{LFI})} > \frac{\mathrm{vol}(S_{FI})}{\mathrm{vol}(S_{LFI})},$$

$$\frac{\mathrm{vol}(S_{FI} \cap R)}{\mathrm{vol}(S_{FI})} > \frac{\mathrm{vol}(S_{LFI} \cap R)}{\mathrm{vol}(S_{LFI})} \geq C.$$

This means that S_{FI} also satisfies the additional constraint in the optimization problem of LocalFlowImprove. But S_{FI} has smaller objective value, which is a contradiction to the fact that S_{LFI} is the optimal solution of the LocalFlowImprove optimization problem. ∎

Theorem 3.1 suggests that one should always use FlowImprove to minimize the conductance around a reference set R, but there are other aspects to implementation that should be taken into account. The three most important, summarized in Table 1, are described here.

- **Locality of algorithm.** For strongly local algorithms, the output is a small cluster around the reference set R *and* the runtime depends only on the size of the output and is independent of the size of the graph. Only the former is true for weakly local algorithms. As we will show in the coming sections, both MQI and LocalFlowImprove are strongly local. This enables both of them to be run quickly on very large graphs, assuming R is not too large and δ is not too small.
- **Exploration properties of algorithm.** Some methods "shrink" the input, in the sense that the output is a subset of the input, while other methods do not have this restriction, i.e., they can (depending on the input graph and seed set) possibly shrink or expand the input. This classification is particularly useful when we view the methods as a way to explore the graph around a given set of seed nodes. For example, MQI only explores the region induced by R, and so it is not suitable for various tasks that involve finding *new nodes*.
- **Ease of implementation.** A final important property of methods concerns how easy they are to implement. MQI and FlowImprove are easy to implement because they rely on standard primitives like simple MaxFlow computations. This means that one can black-box max-flow computations by calling existing efficient software packages. For LocalFlowImprove, however, finding a strongly local algorithm requires a more delicate algorithm. Therefore, we consider it to be a more difficult algorithm to implement.

As a simple and quick justification of the locality property of the solution (which is distinct from an algorithmic approach to achieving it), note the following simple-to-establish relationship between δ and the size of the output set for LocalFlowImprove. This was originally used in Veldt, Klymko, and Gleich (2019) as a small subset of a proof.

LEMMA 3.2. *Let G be an undirected, connected graph with nonnegative weights. Let S^* be an optimal solution of the LocalFlowImprove objective with* $\mathrm{vol}(R) \leq \mathrm{vol}(\bar{R})$. *Then* $\mathrm{vol}(S^*) < \left(1 + \frac{\mathrm{vol}(\bar{R})}{\mathrm{vol}(R)+\delta\,\mathrm{vol}(\bar{R})}\right)\mathrm{vol}(R)$.

Proof. For simplicity, let $\sigma = \mathrm{vol}(R)/\,\mathrm{vol}(\bar{R}) + \delta$. Then, because the denominator in any solution must be positive, we have $0 < \mathrm{vol}(S^* \cap R) - \sigma\,\mathrm{vol}(S^* \cap \bar{R})$. Note that $\mathrm{vol}(S^* \cap \bar{R}) = \mathrm{vol}(S^*) - \mathrm{vol}(S^* \cap R)$, so $0 < (1+\sigma)\,\mathrm{vol}(R \cap S^*) - \sigma\,\mathrm{vol}(S^*)$. Thus, $\mathrm{vol}(S^*) < (1 + 1/\sigma)\,\mathrm{vol}(R)$. The result follows by substituting the definition of σ. ∎

As we will show, all of the algorithms for these objectives fit into a standard fractional programming framework, which provides a useful setting in which to reason about the opportunities and tradeoffs. An even more general setting for such problems is that of *quotient cut problems*, which we discuss in section 3.7. While they are often described in this literature on a case-by-case basis, quotient cut problems are all instances of the more general fractional programming class of problems.

3.2. The Basic Fractional Programming Problem. A fractional program is a ratio of two objective functions: $N(x)$ for the numerator and $D(x)$ for the denomina-

tor. It is often defined with respect to a subset S of $\mathbb{R}^n$,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & N(x)/D(x) \\ \text{subject to} & x \in S, \end{array} \tag{3.1}$$

where $D(x) > 0$ for all $x \in S$. Fractional programming is an important branch of nonlinear optimization (Frenk and Schaible, 2009). The key idea in fractional programming is to relate (3.1) to the function

$$f(\delta) = \text{minimize } N(x) - \delta D(x) \text{ subject to } x \in S,$$

which captures the minimum value of the objective function for this minimization problem as a function of δ. Below, we use "argmin" as the expression for an input or argument that minimizes the problem. Note that $f(\delta) \leq 0$ if there exists x such that $N(x)/D(x) \leq \delta$. Moreover, if $N(x)$ and $D(x)$ are linear functions and S is a set described by linear constraints, then $f(\delta)$ can be easily computed by solving a linear program, for instance.

We now specialize this general framework for cluster improvement. Note that we will continue to use δ as the ratio between the numerator and denominator rather than as the LocalFlowImprove parameter until section 9.

3.3. Fractional Programming for Cluster Improvement. When we consider the objective functions from section 3.1, note that we can translate them into problems closely related to the fractional programming problem (3.1). Let $Q \subseteq V$ represent a subset of vertices. For MQI, this is R itself and for the others, it is just V. Now let $g(S \subseteq Q) \to \mathbb{R}$ represent the denominator terms for the MQI, FlowImprove, or LocalFlowImprove objectives from section 3.1. Then, from a fractional programming perspective on the problems, we are interested in solving

ASIDE 5. *Most commonly, fractional programming is defined for subsets of $\mathbb{R}^n$ as the domain. In our case, we use set-based domains.*

$$\begin{array}{ll} \text{minimize} & \phi_g(S) := \begin{cases} \frac{\text{cut}(S)}{g(S)}, & g(S) > 0, \\ \infty & \text{otherwise}, \end{cases} \\ \text{subject to} & S \subseteq Q. \end{array} \tag{3.2}$$

Let us assume that there is at least one feasible set $S \subseteq Q$ where $g(S) > 0$. This is satisfied for all the examples above when $S = R$. Also note that if $Q = V$ and if $g(V) > 0$, the entire node set V is immediately a solution. For FlowImprove and LocalFlowImprove, though, $g(V) \leq 0$, and so V is never a solution and, in fact, the value of κ in FlowImprove is chosen exactly so that $g(V) = 0$.

As discussed above, we will use a sequence of related parametric problems to find the optimal solution. Thus, we introduce the parametric function

$$z(S, \delta) := \text{cut}(S) - \delta g(S),$$

where the parameter $\delta \in \mathbb{R}$. We also define the function

$$\hat{z}(\delta) := \underset{S}{\text{minimize}}\ z(S, \delta), \text{ where } S \subseteq Q, g(S) > 0. \tag{3.3}$$

Computing the value of $\hat{z}(\delta)$ is a key component that we will discuss in section 3.6 and also sections 6 to 8. Given this, we can consider solving the equation

$$\hat{z}(\delta) = 0, \tag{3.4}$$

which is a simple root-finding problem because $\hat{z}(\delta)$ is monotonically increasing as $\delta \to 0$ and also $\hat{z}(0) \geq 0$ and $\hat{z}(\phi_g(R)) \leq 0$ (for our objectives). Note that $\hat{z}(0) = 0$ if $\text{cut}(S) = 0$ for some set $S \subseteq Q$ with $g(S) > 0$, which can happen for a disconnected graph.

We now provide a theorem that establishes the relationship between the root-finding problem (3.4) and the basic fractional programming problem (3.2). This theorem establishes that in solving problem (3.4), we solve problem (3.2) as well. A similar theorem can be found in Dinkelbach (1967).

THEOREM 3.3. *Let G be an undirected, connected graph with nonnegative weights. A set of nodes S^* is a solution of problem* (3.2) *iff*

$$\hat{z}\left(\frac{\text{cut}(S^*)}{g(S^*)}\right) = 0.$$

Proof. For the first part of the proof, let us assume that S^* is a solution of problem (3.2). This implies that $g(S^*) > 0$. We have that

$$\delta^* := \frac{\text{cut}(S^*)}{g(S^*)} \leq \phi_g(S) \quad \forall S \subseteq Q, g(S) > 0.$$

Hence,

$$\text{cut}(S^*) - \delta^* g(S^*) = 0$$

and

$$\text{cut}(S) - \delta^* g(S) \geq 0 \quad \forall S \subseteq Q, g(S) > 0.$$

Using the above we have that $\{\min z(S, \delta^*) \mid S \subseteq Q, g(S) > 0\}$ is bounded below by zero, and this bound is achieved by S^*. Therefore, $\hat{z}(\delta^*) = 0$, $z(S^*, \delta^*) = 0$.

For the second part of the proof, assume that $\hat{z}(\delta^*) = 0$ such that

$$\delta^* = \frac{\text{cut}(S^*)}{g(S^*)} \tag{3.5}$$

for some optimal S^* of the minimization problem in $\hat{z}$. Then

$$\text{cut}(S^*) - \delta^* g(S^*) = 0 \leq \text{cut}(S, \bar{S}) - \delta^* g(S) \quad \forall S \subseteq Q, g(S) > 0. \tag{3.6}$$

From the second inequality, we have that $\phi_g(S) \geq \delta^*$ for all $S \subseteq Q, g(S) > 0$. This means that the optimal solution of problem (3.2) is bounded below by δ^*. From the first equation above, we find that this bound is achieved by S^*. Therefore, S^* solves problem (3.2). ∎

3.4. Dinkelbach's Algorithm for Fractional Programming. Based on Theorem 3.3, the root of problem (3.4) will be the optimal value of the general cluster improvement problem (3.2). To find the root of problem (3.4), we will use a modified version of Dinkelbach's algorithm (Dinkelbach, 1967).

Dinkelbach's Algorithm. Dinkelbach's algorithm is given in Algorithm 3.1. Note that we had to modify the original algorithm slightly since we do not assume that $g(S) > 0$ for all $S \subseteq Q$.

Algorithm 3.1 Dinkelbach's Algorithm.

1: Initialize $k := 1$, $S_1 := R$, and $\delta_1 := \phi_g(S_1)$.
2: **while** we have not exited via the else clause **do**
3: Compute $\hat{z}(\delta_k)$ by solving $S_{k+1} := \operatorname{argmin}_S z(S, \delta_k)$ subject to $S \subseteq Q$
4: **if** $\phi_g(S_{k+1}) < \delta_k$ {recall $\phi_g(S) = \infty$ if $g(S) \leq 0$} **then**
5: $\delta_{k+1} := \phi_g(S_{k+1})$
6: **else**
7: δ_k is optimal, return previous solution S_k.
8: $k := k + 1$

Convergence of Dinkelbach's Algorithm. We now provide a theorem that establishes that the subproblem at step 3 of Algorithm 3.1 does not output infeasible solutions, such as an S that satisfies $g(S) \leq 0$. Based on this, we can establish that the objective function of problem (3.2) is decreased at each iteration of Algorithm 3.1.

THEOREM 3.4 (convergence). *Let G be an undirected, connected graph with nonnegative weights. Let δ^* be the optimal value of problem* (3.2)*. The subproblem in step* 3 *of Algorithm* 3.1 *cannot have solutions that satisfy $g(S) \leq 0$ for $\delta > \delta^*$. Such solutions are in the solution set of the subproblem if and only if $\delta \leq \delta^*$. Moreover, the sequence δ_k, which is set to be equal to $\phi_g(S_k)$, decreases monotonically at each iteration. The algorithm returns a solution where $g(S_k) > 0$.*

Proof. For the first part of the theorem, let $\delta \geq 0$, $\hat{S} \in \{\operatorname{argmin} z(S, \delta)\}$, and let us assume for the sake of contradiction that $g(\hat{S}) \leq 0$. Then

$$z(S, \delta) \geq z(\hat{S}, \delta) \geq 0 \ \forall S \subseteq Q.$$

Hence,

$$\phi_g(S) \geq \delta \ \forall S \in \{S \subset Q \mid g(S) > 0\};$$

however, this can only be true if $\delta \leq \delta^*$. Otherwise, for $\delta > \delta^*$ we have a contradiction, and this implies that $g(\hat{S}) > 0$. Therefore, a solution $\hat{S} \in \{\operatorname{argmin} z(S, \delta)\}$ satisfies $g(\hat{S}) > 0$, unless $\delta \leq \delta^*$.

For the second part of the theorem, let k be such that $\delta_k > \delta^*$. Then, we have that $z(S_{k+1}, \delta_k) < 0$, since $z(S_{k+1}, \delta_k) < z(S_k, \delta_k) = 0$ (where we obtain 0 by the definition of δ_k and S_k). Because $z(S_{k+1}, \delta_k) < 0$, we have that $\phi_g(S_{k+1}) = \delta_{k+1} < \delta_k = \phi_g(S_k)$. Note that because $g(S_{k+1}) > 0$ for any $\delta_k > \delta^*$, then we must have $\delta_{k+1} \geq \delta^*$.

Note that because of the algorithm, δ_k can never be less than δ^*. Thus, the remaining case is to detect that $\delta_k = \delta^*$. Suppose this is the case and also $g(S_{k+1}) > 0$; then $\delta_{k+1} = \delta_k = \delta^*$, and based on Theorem 3.3 the algorithm terminates with an optimal solution because either S_{k+1} or S_k is a solution. If $\delta_k = \delta^*$ and $g(S_{k+1}) \leq 0$, then the algorithm terminates (because $\phi_g(S_{k+1}) = \infty$). Thus, S_k must have been optimal (if not, then $g(S_{k+1})$ must be larger than 0) and so the algorithm outputs an optimal solution. ∎

Iteration Complexity of Dinkelbach's Algorithm. The iteration complexity of a method allows us to deduce a bound on the number of iterations necessary. We now provide an iteration complexity result for Algorithm 3.1. This involves two other results. We begin with Lemma 3.5, which describes several interesting properties of Algorithm 3.1 that have an important practical implication. Specifically, it shows that $g(S_{k+1}) <$

$g(S_k)$, which has important *practical* implications since it shows that Algorithm 3.1 is searching for subsets S that have a *smaller* value of the function g. Lemma 3.5 will then allow us to prove an iteration complexity result of Algorithm 3.1 in Theorem 3.6. A similar result can be found in Gallo, Grigoriadis, and Tarjan (1989, Lemma 4.3), but we repeat it in Lemma 3.5 for completeness. In Lemma 3.5, we also show that the numerator of the objective function in problem (3.2) decreases monotonically.

LEMMA 3.5. *If Algorithm* 3.1 *proceeds to iteration* $k+1$*, then it satisfies both* $g(S_{k+1}) < g(S_k)$ *and* $\mathrm{cut}(S_{k+1}) < \mathrm{cut}(S_k)$.

Proof. Consider iterations k and $k-1$ and assume that $\delta_k > \delta^*$. Then, from Theorem 3.4, in iteration $k-1$, we have that $z(S_k, \delta_{k-1}) < z(S_{k-1}, \delta_{k-1}) = 0$. In iteration k, we have that

$$z(S_{k+1}, \delta_k) = \mathrm{cut}(S_{k+1}) - \delta_k g(S_{k+1}) < 0.$$

By adding and subtracting $\delta_{k-1} g(S_{k+1})$ to and from the latter, we get

$$z(S_{k+1}, \delta_k) = \mathrm{cut}(S_{k+1}) - \delta_{k-1} g(S_{k+1}) + \delta_{k-1} g(S_{k+1}) - \delta_k g(S_{k+1}) < 0.$$

Note that the first two terms on the right side of the equality are the minimization problem, for that gave the solution S_k. Hence, we can lower-bound $\mathrm{cut}(S_{k+1}) - \delta_{k-1} g(S_{k+1})$ via S_k to get

$$\mathrm{cut}(S_k) - \delta_{k-1} g(S_k) + \delta_{k-1} g(S_{k+1}) - \delta_k g(S_{k+1}) \le z(S_{k+1}, \delta_k) < 0.$$

Because $z(S_k, \delta_k) = 0$, we find that $\mathrm{cut}(S_k) = \delta_k g(S_k)$. Thus, using this in the inequality above, we get

$$\delta_k g(S_k) - \delta_{k-1} g(S_k) + \delta_{k-1} g(S_{k+1}) - \delta_k g(S_{k+1}) \le z(S_{k+1}, \delta_k) < 0,$$

which is equivalent to

$$(g(S_k) - g(S_{k+1}))(\delta_k - \delta_{k-1}) < 0.$$

However, because the algorithm monotonically decreases δ_k in each iteration, we have that $\delta_{k-1} - \delta_k < 0$, and therefore we must have that

$$g(S_k) > g(S_{k+1}).$$

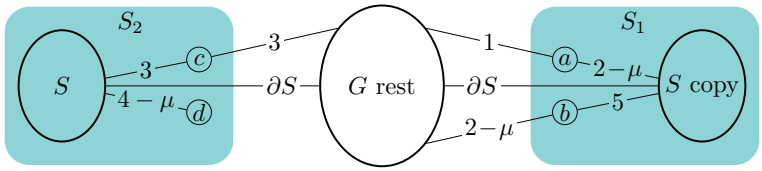
This means that the denominator of the objective function in problem (3.2) decreases monotonically. Additionally, from Theorem 3.4 we have that the objective function decreases monotonically. These two imply that the numerator of the objective function, i.e., $\mathrm{cut}(S, \bar{S})$, decreases monotonically. ∎

Given this result, we can establish the following theorem, which provides an iteration complexity for Algorithm 3.1. This basic result can be improved, as we describe next in section 3.5.

THEOREM 3.6 (iteration complexity for Dinkelbach's algorithm). *Consider using Dinkelbach's algorithm (Algorithm* 3.1*) for solving MQI, FlowImprove, or LocalFlowImprove on an undirected, connected graph with nonnegative integer weights when starting with the set* R*. Then the algorithm needs at most* $\mathrm{cut}(R) \le \mathrm{vol}(R)$ *iterations to converge to a solution.*

Proof. For all of the above programs, R is a feasible set and thus we can initialize our algorithms with R. From Lemma 3.5, we have that $\mathrm{cut}(S)$ decreases monotonically at each iteration. Since we assume that the graph is integer-weighted, then $\mathrm{cut}(S)$ is integer-valued and so $\mathrm{cut}(R)$ gives an upper bound on the number of iterations. Note that $\mathrm{cut}(R) \leq \mathrm{vol}(R)$ for any set and so the algorithms need at most $\mathrm{cut}(R) \leq \mathrm{vol}(R)$ iterations to converge to a solution S^*. ∎

Remark 3.7. A weakness of the previous result is that it does not give a complexity result for graphs with noninteger weights. For weighted graphs with noninteger weights, if the weights come from an ordered field where the minimum relative spacing between elements is μ, such as would exist for rational-valued weights or floating-point weights, then the above argument gives $\mathrm{cut}(R)/\mu$ iterations. This is essentially tight as the following construction gives two sets whose cut and volume differ only by μ.



Here, S and S copy are duplicates of the same subgraph, so their cut ∂S is identical. Assume S is small enough that we do not need to take into consideration the min term in conductance. Then note that $\phi(S_1) = \frac{\mathrm{cut}(S_1)}{\mathrm{vol}(S_1)} = \frac{\mathrm{cut}(S_2)-\mu}{\mathrm{vol}(S_2)-\mu}$. Furthermore, there is no obvious way to detect this scenario as we have a set of well-spaced distinct edge weights $(1,\ 2-\mu,\ 3,\ 4-\mu,\ 5-\mu)$. (Assume all other edges in the graph have weight 1.)

For this reason, we do not consider the iteration complexity of algorithms for graphs with noninteger weights and we would recommend the algorithm in the next section for finding an approximate answer.

3.5. A Faster Version of Dinkelbach's Algorithm via Root-Finding. Algorithm 3.1 requires at most $\mathrm{vol}(Q)$ iterations to converge to an *exact* solution for nonnegative integer-weighted graphs. If we are not interested in exact solutions, then we can improve the iteration complexity of Algorithm 3.1 by performing a binary search on δ. This is possible because it is easy to find bounds on the optimal range of δ. We have zero as a simple lower bound and, for the MQI, FlowImprove, and LocalFlowImprove objectives, $\phi(R) \leq 1$ is an easy-to-compute upper bound on the optimal δ. Algorithm 3.2 presents a modified version of Dinkelbach's algorithm that accomplishes this. In particular, the subproblem in step 4 in Algorithm 3.2 is the same as the subproblem in step 3 of the original Algorithm 3.1.

At steps 5 to 8 of Algorithm 3.2, we make the decision to update $\delta_{\max}$ and $\delta_{\min}$ based on the optimal value of the subproblem. We further store the best solution so far in $S_{\max}$. In step 10, we test whether another solve with $\delta_{\max}$ produces a solution with a better objective than $S_{\max}$. This test allows us to certify that $S_{\max}$ is optimal if the subsequent objective is not lower.

In order to have a convergent algorithm, we have to guarantee that this decision results in a well-defined binary search. In the following theorem, we address this issue.

THEOREM 3.8 (convergence of Algorithm 3.2 and iteration complexity). *Let G be an undirected, connected graph with nonnegative weights. The binary search procedure in Algorithm* 3.2 *is well-defined, in the sense that the binary search interval includes*

Algorithm 3.2 Fast Dinkelbach's Algorithm for Problem (3.2).

1: Initialize $k := 1$, $\delta_{\min} := 0$, $\delta_{\max} \geq p := \{\max \phi_g(S) \mid S \subseteq Q\}$, and $\varepsilon \in (0, 1]$
2: **while** $\delta_{\max} - \delta_{\min} > \varepsilon \delta_{\min}$ **do**
3: $\quad \delta_k := (\delta_{\max} + \delta_{\min})/2$
4: $\quad$ Compute $\hat{z}(\delta_k)$ by solving $S_{k+1} := \operatorname{argmin}_S z(S, \delta_k)$ subject to $S \subseteq Q$
5: $\quad$ **if** $g(S_{k+1}) > 0$ {then $\delta_k \geq \delta^*$} **then**
6: $\quad\quad \delta_{\max} := \phi_g(S_{k+1})$ and set $S_{\max} := S_{k+1}$ {note $\phi_g(S_{k+1}) \leq \delta_k$}
7: $\quad$ **else**
8: $\quad\quad \delta_{\min} := \delta_k$
9: $\quad k := k + 1$
10: Return $\operatorname{argmin}_{S \subseteq Q} z(S, \delta_{\max})$ or $S_{\max}$ based on minimum ϕ_g

the optimal solution, and the condition in step 5 *tells us if the optimal solution is above or below* δ_k*. Moreover, the sequence* δ_k *of Algorithm* 3.2 *converges to an approximate solution* $|\delta^* - \delta_k|/\delta^* \leq \varepsilon$ *in* $\mathcal{O}(\log(\delta_{\max}/\varepsilon))$ *iterations, where* $\delta^* = \phi_g(S^*)$ *and* S^* *is an optimal solution to problem* (3.2)*.*

Proof. Let $p := \{\max \phi_g(S) \mid S \subseteq Q\}$ and $\delta_{\max} \geq p$. Let S^* be an optimal solution of problem (3.2). From Theorem 3.3, we see that for (S^*, δ^*) we have that $z(S^*, \delta^*) = 0$, which gives $\phi_g(S^*) = \delta^*$. Therefore, $\delta^* \in [0, \delta_{\max}]$. We will use this interval as our search space for the binary search. Moreover, if $g(S_{k+1}) > 0$, then we get from Theorem 3.4 that $\delta_k > \delta^*$. Therefore, we can use δ_k to update $\delta_{\max}$ in step 6. In fact, because we have a specific set, we know that $\phi_g(S_{k+1}) \leq \delta_k$ and so we can use a slightly tighter update. However, if $g(S_{k+1}) \leq 0$, then we see from Theorem 3.4 that $\delta_k \leq \delta^*$, and we can use δ_k to define $\delta_{\min}$ in step 8. If the initial $\delta_{\max}$ is greater than p, then it is easy to see that Algorithm 3.2 converges to an optimal solution of problem (3.2) in at most $\log(\delta_{\max}/\varepsilon)$ iterations, where $\varepsilon > 0$ is an accuracy parameter. ∎

Note that Theorem 3.8 is an improvement over Theorem 3.6. The former requires $\mathcal{O}(\log(\delta_{\max}/\varepsilon))$ iterations in the worst case, while the latter states that Dinkelbach's algorithm requires $\text{vol}(Q)$ (number of edges) iterations. Similar results about binary search have been discussed in Lang and Rao (2004); Andersen and Lang (2008); Hochbaum (2010). Among other details, what is missing from these references is an exact quantification of the value of ε necessary for an exact solution, which we provide in subsequent sections.

3.6. The Algorithmic Components of Cluster Improvement. We have now shown how to solve cluster improvement problems in the form of problem (3.2) via either Dinkelbach's algorithm or the bisection-based root-finding variation. The last component of the algorithmic framework is a solver for the subproblem (3.3) in the appropriate step (3 or 4) of each algorithm. Solving these subproblems is where the MinCut- and MaxFlow-based algorithms arise as they allow us to test $\hat{z}(\delta) < 0$. In sections 6 to 8 we work through how appropriate MinCut and MaxFlow problems can be derived constructively.

At this point, we summarize the major results and give an overview of the runtimes of the methods we will establish in these sections. In particular, in Table 2, we provide pointers to algorithms and convergence theorems for each method, and we also provide a short summary of runtimes for each method where we make it clear that the subproblem solve time is a dominant term.

Table 2 *Specifics of MQI, FlowImprove, and LocalFlowImprove as special cases of Dinkelbach's Algorithm* 3.1 *and its binary search version Algorithm* 3.2. *In the table, R is the input seed set of nodes. The column "Subproblem" refers to the specialized subsolver that is used to solve the subproblem at step* 3 *of Algorithm* 3.1 *or step* 4 *of Algorithm* 3.2. *The "Augmented Graph" entry refers to an augmented graph construction that is used to understand the subproblem that is solved at each iteration of Dinkelbach's algorithm. Note that we omit all log-factors and constants from the runtimes of the algorithms; more detailed runtimes can be found in the referenced theorems. We use $\tilde{\mathcal{O}}$ as $\mathcal{O}$-notation without logarithmic factors.*

Method	*Dinkelbach* and Runtime	*Binary Search* and Runtime	*Subproblem* Construction, Runtime, and Solvers
MQI	Algorithm 6.1 $\mathcal{O}(\mathrm{cut}(R) \cdot \texttt{subproblem})$ Theorem 6.3 (Lang and Rao, 2004)	Algorithm 6.2 $\tilde{\mathcal{O}}(\texttt{subproblem})$ Theorem 6.5 (Lang and Rao, 2004)	Problem (6.3) Augmented Graph 1 MaxFlow with $\mathrm{vol}(R)$ edges (section 6.1)
FlowImprove	Algorithm 7.1 $\mathcal{O}(\mathrm{cut}(R) \cdot \texttt{subproblem})$ Theorem 7.3 (Andersen and Lang, 2008)	Algorithm 7.2 $\tilde{\mathcal{O}}(\texttt{subproblem})$ Theorem 7.5 (Andersen and Lang, 2008)	Problem (7.3) Augmented Graph 2 MaxFlow with $\mathrm{vol}(G)$ edges (section 7.1)
LocalFlow-Improve(δ) $\sigma = \delta + \frac{\mathrm{vol}(R)}{\mathrm{vol}(\bar{R})}$	SimpleLocal $\mathcal{O}(\mathrm{cut}(R) \cdot \texttt{subproblem})$ Theorem 8.3 (Veldt, Gleich, and Mahoney, 2016)	Algorithm 8.1 $\tilde{\mathcal{O}}(\texttt{subproblem})$ Theorem 8.3 (Orecchia and Zhu, 2014)	Problem (8.3) Augmented Graph 3 $\tilde{\mathcal{O}}((1+1/\sigma)^2\,\mathrm{vol}(R)^2)$ with Alg. 8.3 (sections 8.1–8.3)

3.7. Beyond Conductance and Degree-Weighted Nodes. Our discussion and analysis of fractional programming for cluster improvement objectives has, so far, focused on the MQI, FlowImprove, and LocalFlowImprove problems as unified through problem (3.2). However, there is a broader class of objectives that generalizes beyond these specific types of cuts and volume ratios. We will highlight a few definitions that are reasonably straightforward to understand, although we will return to the MQI, FlowImprove, and LocalFlowImprove definitions above in what follows.

As an instance of a more generalized setting, we can define a generalized volume of a set S, which we call ν, with respect to an arbitrary vector of positive weights w,

$$\nu(S; w) = \sum_{i \in S} w_i = \mathbf{1}_S^T w.$$

Note that setting w to be the degree vector d gives the standard definition of volume, i.e., $\nu(S; d) = \mathrm{vol}(S)$. Then we can seek solutions of

$$\begin{array}{ll} \underset{S \subset V}{\text{minimize}} & \dfrac{\mathrm{cut}(S)}{\nu(S \cap R; w) - \kappa \nu(S \cap \bar{R}; w)} \\ \text{subject to} & \text{denominator} > 0 \end{array}$$

as a generalized notion of MQI, FlowImprove, and LocalFlowImprove (where $\kappa \geq \nu(R; w)/\nu(\bar{R}; w)$).

A particularly useful instance is where w is simply the vector of all ones, 1_n, in which case $\nu(S, 1_n)$ is simply the cardinality of the set S. In this case, $\mathrm{cut}(S)/\nu(S, 1_n)$ is the expansion or ratio cut value of a set (section 2.6). This approach was used in the original MQI paper (Lang and Rao, 2004), which discussed ratio cuts instead of conductance values. This more general notion of volume also appeared in the

FlowImprove paper (Andersen and Lang, 2008) in order to unify the analysis of ratio cuts and conductance objectives. While these two choices have been explored, of course, the theory allows us to choose virtually any vector and this gives a large amount of flexibility. The MaxFlow and MinCut constructions for the subproblems in subsequent sections would need to be adjusted to account for this type of arbitrary choice. This is reasonably straightforward given our derivations. For example, we could set $w = \sqrt{d}$ to generate a hybrid objective between expansion and conductance.

As another example of how the framework can be even more general, we mention the ideas from Veldt, Klymko, and Gleich (2019) that *penalize* excluding nodes from R in the solution set S. These penalties can be set sufficiently large that we can solve variations of FlowImprove and LocalFlowImprove where all the nodes in R *must* be in the result; for instance,

$$\begin{array}{ll} \underset{S \subset V}{\text{minimize}} & \dfrac{\text{cut}(S)}{\nu(S \cap R; w) - \kappa\nu(S \cap \bar{R}; w)} \\ \text{subject to} & R \subset S, \text{denominator} > 0. \end{array}$$

They can also, however, be set smaller such that we aim to have *most* of R within the solution S. This scenario is helpful when the elements of R may have a *confidence* associated with them.

All of the analysis in subsequent sections—including the locality of computations —applies to these more general settings; however, the generalized details often obscure the simplicity of and connections among the methods. Thus, we do not conduct the most general description possible but simply emphasize that it is possible and useful to do so.

4. Cluster Improvement, Flow-Based, and Other Related Methods. As we have already briefly discussed, graph clustering is a well-established problem with an extensive literature. Cluster improvement algorithms have received comparatively little attention. In this section, we will discuss how the cluster improvement problem and algorithms for solving it are both similar to and different from other related techniques in the literature. Our goal is to draw a helpful distinction and explain the relationships among cluster improvement problems and algorithms and a number of other (sometimes substantially but sometimes superficially) related topics.

For instance, we will discuss how the cluster improvement perspective yields the best results on graph and mesh partitioning benchmark problems (section 4.1). We will then highlight key differences between the types of graphs arising in scientific and distributed computing and the types of graphs based on sparse relational data and complex systems (section 4.2), which strongly motivates the use of *local algorithms* for these data. These local graph clustering algorithms, in turn, have strong relationships with the community detection problem in networks as well as with inferring metadata, which we will explore more concretely in the empirical sections.

Taking a step back, we explain our cluster improvement algorithms in terms of finding sets of small conductance, and so we also briefly survey the state of conductance optimization techniques more generally (section 4.4). Likewise, our algorithms are all based on the use of a network flow optimizer as a subroutine to accomplish something else. Since this scenario is surprisingly common, e.g., because there are fast algorithms for network flow computations, we highlight a few notable applications of network flow–based computing (section 4.5) as well as the current state of the art for computing network flows (section 4.6).

Finally, we conclude this section by relating our cluster improvement perspective

to network flows in continuous domains (section 4.7), total variation metrics, and a wide range of work using graph cuts and flows in image segmentation (section 4.8).

4.1. Graph and Mesh Partitioning in Scientific Computing. Graph and mesh partitioning are important tools in parallel and distributed computing, where the goal is to partition a computation into *many large* pieces that can be treated with minimal dependencies among the pieces. This can then be used to maximize parallelism and minimize communication in large scientific computing algorithms (Pothen, Simon, and Liou, 1990; Simon, 1991; Karypis and Kumar, 1998; Hendrickson and Leland, 1994b, 1995; Karypis and Kumar, 1999; Hendrickson and Leland, 1994; Walshaw and Cross, 2007, 2000; Pellegrini and Roman, 1996; Knight, Carson, and Demmel, 2014). The traditional inputs to graph partitioning for scientific computing are graphs representing computational dependencies involved in solving a spatially discretized partial differential equation. In these problems, there is often a strong underlying geometry, in which nodes are localized in space and edges are between nearby nodes. Furthermore, one of the key goals (indeed, almost a constraint in this application) is that the partitions should be very well balanced so that no piece is much larger than the others.

In the context of this literature, our goal is not to produce an overall partitioning of the graph. Rather, given a piece of a partition, our tools and algorithms should enable a user to *improve* that partition in light of an objective function such as graph conductance or another related objective. Indeed, work on improving and refining the quality of an initial graph bisection can be found in the Fiduccia–Mattheyses implementation of the Kernighan–Lin method (Fiduccia and Mattheyses, 1982). Given a quality score for a two-way partition of a graph and a desired balance size, this algorithm searches among a class of local moves that could improve the quality of the partition. This improvement technique is incorporated, for instance, into the SCOTCH (Pellegrini and Roman, 1996), Chaco (Hendrickson and Leland, 1994), and METIS (Karypis and Kumar, 1998) partitioners.

This strategy for partition-and-improvement is also a highly successful paradigm for generating the best quality bisections and partitions on benchmark data. For example, on the Walshaw collection of partitioning test cases (Soper, Walshaw, and Cross, 2004), around half of the current best-known results are the result of improving an existing partitioning using an improvement algorithm (Henzinger, Noe, and Schulz, 2020). This has occurred a few times in the past as well (Sanders and Schulz, 2011; Hein and Setzer, 2011; Lang and Rao, 2004). There are important differences between the applications we consider (which are more motivated by machine learning and data science) and those in mesh partitioning for scientific computing. Most notably, having good balance among all the partitions is extremely important for efficient parallel and distributed computing, but it is much less so for social and information networks, as we discuss in the next section.

4.2. The Nature of Clusters in Sparse Relational Data and Complex Systems. Beyond the runtime difference between local and global graph analysis tools, there is another important reason to consider local graph analysis for sparse relational data such as social and information networks, machine learning, and complex systems. There is strong evidence that large-scale graphs arising in these fields (Leskovec et al., 2009, 2008; Leskovec, Lang, and Mahoney, 2010; Gargi et al., 2011; Jeub et al., 2015) have interesting small-scale structure, as opposed to interesting and nontrivial large-scale global structure. Even aside from runtime considerations, this means that global graph methods tend to have trouble identifying these small and good clusters and thus

may not be very applicable to many large graphs that arise in large-scale data applications. As a simple example of the impact the differences in data may have on a method, note that for graphs such as discretizations of a partial differential equation, simply enlarging a spatially coherent set of vertices results in a set of better conductance (until it is more than half the graph). On the other hand, the sets of small conductance in machine learning and social network based graphs tend to be small, in which case enlarging them simply makes them worse in terms of conductance. This has been quantified by the Network Community Profile (NCP) plot (Leskovec et al., 2009; Jeub et al., 2015).

4.3. Local Graph Clustering, Community Detection, and Metadata Inference. Local graph clustering is, by far, the most highly developed setting for local graph algorithms. A local graph clustering method seeks a cluster nearby the reference set R, which can be as small as a single node. Cluster improvement algorithms are, from this perspective, instances of local graph clustering where the input is a good cluster R and the output is an even better cluster S. Local graph clustering itself emerged simultaneously out of the study of partitioning graphs for improvement in theoretical runtime of Laplacian solvers (Spielman and Teng, 2013) and the limitations of global algorithms applied to graphs based on machine learning and data analysis (Lang, 2005; Andersen and Lang, 2006; Andersen, Chung, and Lang, 2006). Subsequently, there have been a large number of developments in both theory, practice, and applications. These include

- improved theoretical bounds (Zhu, Lattanzi, and Mirrokni, 2013; Andersen et al., 2016);
- novel recovery scenarios (Kloumann and Kleinberg, 2014);
- optimization-based approaches and formulations (Gleich and Mahoney, 2014, 2015; Fountoulakis, Gleich, and Mahoney, 2017; Fountoulakis et al., 2019c);
- heat kernel–based approaches (Chung, 2007a, 2009; Chung and Simpson, 2014; Kloster and Gleich, 2014; Avron and Horesh, 2015);
- Krylov- and Lanczos-based approaches (Li et al., 2015; Shi et al., 2017);
- local higher-order clustering based on triangles (Yin et al., 2017; Tsourakakis, Pachocki, and Mitzenmacher, 2017);
- large-scale parallel approaches (Shun et al., 2016).

One reason for the diversity of methods in this area is that local graph clustering is a common technique used to study the community structure of a complex system or social network (Leskovec et al., 2009, 2008; Leskovec, Lang, and Mahoney, 2010). The communities, or modules, of a network represent a coarse-grained view of the underlying system (Newman, 2006; Palla et al., 2005). In particular, local clustering, local improvement, and local refinement algorithms are often used to generate overlapping groups of communities from any community partition (Lancichinetti, Fortunato, and Kertész, 2009; Xie, Kelley, and Szymanski, 2013; Whang, Gleich, and Dhillon, 2016). This is often called a *local optimization and expansion* methodology.

Another application of local graph clustering is metadata inference. The metadata inference problem is closely related to semisupervised learning, where the input is a graph and a set of labels with many missing entries. The goal is to *interpolate* the labels around the remainder of the graph. Hence, any local clustering method can also be used for semisupervised learning problems (Joachims, 2003; Zhou et al., 2004; Liu and Chang, 2009; Belkin, Niyogi, and Sindhwani, 2006; Zhu, Ghahramani, and Lafferty, 2003) (and thus metadata inference). That said, the metadata application raises a variety of statistical consistency questions (Ha, Fountoulakis, and Mahoney,

2020), methodological questions due to a no-free-lunch theorem (Peel, Larremore, and Clauset, 2017), and data suitability questions (Peel, 2017). We omit discussion of these questions in the interest of brevity and note that some caution with this approach is advisable.

Among the local graph clustering methods, the Andersen–Chung–Lang algorithm for seeded PageRank computation (Andersen, Chung, and Lang, 2006) is often the de facto choice. This method has both useful theoretical and useful empirical properties, namely, recovery guarantees in terms of small conductance clusters (Andersen, Chung, and Lang, 2006; Zhu, Lattanzi, and Mirrokni, 2013) and extremely fast computation (Andersen, Chung, and Lang, 2006). It also has close relationships to many other perspectives on graph problems (see, e.g., Gleich and Mahoney (2015), Fountoulakis et al. (2019c), and Fountoulakis, Gleich, and Mahoney (2017)), including robust and 1-norm regularized versions of these problems.

Cluster improvement algorithms are a natural fit for both community detection and metadata inference settings. Given any partition of the network, set of communities, set of overlapping communities, or any other set of vertex sets, we can study the results of improving each set individually. This is exactly the setting of Figure 1, where we were able to find a better partition of the network given an initial partition (although these techniques may not result in a partition). Second, for metadata inference, we simply seek to use a given label as a reference set that we *improve*. We explore these applications from an empirical perspective in section 9, where we compare them to a relative of the Andersen–Chung–Lang method for these tasks.

4.4. Conductance Optimization. Taking a step back, the cluster improvement algorithms we discuss improve the *conductance* or *ratio cut* scores. Finding the overall minimum conductance set in a graph is a well-known NP-hard problem (Shahrokhi, 1990; Leighton and Rao, 1999). That said, there exist approximation algorithms based on linear programming (Leighton and Rao, 1988, 1999), semidefinite programming (Arora, Rao, and Vazirani, 2009), and so-called cut-matching games (Khandekar, Rao, and Vazirani, 2009; Orecchia et al., 2012). A full comparison and discussion of these ideas is beyond the scope of this survey. We note that these techniques are not often implemented due to complexities in the theory needed to get the sharpest possible bounds. However, they do inspire new scalable approaches, for instance, (Lang, Mahoney, and Orecchia, 2009).

4.5. Network Flow–Based Computing. More broadly, beyond conductance optimization, our work relates to the idea of using *network flow* itself as a fundamental computing primitive. By this, we mean that many other algorithms can be cast as instances of network flow or sequences of network flow problems. When this is possible, it enables us to use highly optimized solvers for this specific purpose that often outperform more general methods. Bipartite matching is a well-known textbook example of this scenario (Kleinberg and Tardos, 2005, section 7.5). Other examples include finding the densest subgraph of a network, which is the subset of vertices with highest average degree. Formally, if we define

$$\text{density}(S) = \frac{\text{vol}(S) - \text{cut}(S)}{|S|},$$

then the set S that maximizes this quantity is polynomial time computable via a sequence of network flow problems (Goldberg, 1984). Another instance is one of the many definitions of *communities* on the web that can be solved exactly as a max-flow

problem (Flake, Lawrence, and Giles, 2000). More relevant to our setting is the work of Hochbaum (2013), who showed that the sets that satisfy

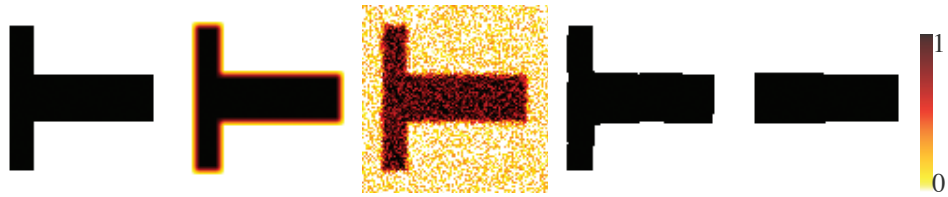
$$\underset{S}{\text{minimize}} \ \frac{\text{cut}(S)}{\text{vol}(S)} \quad \text{and} \quad \underset{S}{\text{minimize}} \ \frac{\text{cut}(S)}{|S|}$$

can be found in polynomial time through a sequence of max-flow and min-cut computations. Although feasible to compute, in general these sets are unlikely to be interesting on many machine learning and data analysis based graphs, as they will tend to be very large sets that cut off a small piece of the rest of the graph. (Formally, suppose there exists a node of degree 1 in an unweighted graph; then the complement set of that node will be the solution.) Among other reasons, this is why we use the objective functions that are symmetric in S and $\bar{S}$.

Four other interesting cases show the diversity of this technique. First, the semisupervised learning algorithm of Blum and Chawla (2001) uses the min-cut algorithm to identify other vertices likely to share the same label as those that are given. The second case is the use of flows to estimate a gradient in an algorithm for ranking a set of data due to Osting, Darbon, and Osher (2013). Third, there are useful connections between *matching* algorithms (which can be solved as flow problems) and semisupervised learning problems (Jacobs, Merkurjev, and Esedoḡlu, 2018). Finally, there is a recent set of research on *total variation* or TV norms in graphs and their connections to network flow (Jung et al., 2019). These were originally conceptualized for semisupervised learning but can also be used to build local clustering mechanisms that optimize a combination of 2-norm and 1-norm objectives with max-flow techniques (Jung and SarcheshmehPour, 2021).

4.6. Recent Progress on Network Flow Algorithms. Having flow as a subroutine is useful because there is a large body of work in both theory and practice that concerns making flow computations fast. For an excellent survey of the overall problem, the challenges, and recent progress, we recommend Goldberg and Tarjan (2014). This overview touches on the exciting line of work in theory that showed a connection between Laplacian linear system solving and approximate max-flow computations (Christiano et al., 2011; Lee, Rao, and Srivastava, 2013) as well as recent progress on the exact problem (Orlin, 2013). We refer readers also to Lee and Sidford (2013) and Liu and Sidford (2020), as well as to software packages that compute maximum flows fast (Dezso, Alpár, and Kovács, 2011).

4.7. Continuous and Infinite-Dimensional Network Flow and Cuts. Our approach in this survey begins with a finite graph based on data and is entirely finite-dimensional. Alternative approaches seek to understand problems in the continuous or infinite-dimensional setting. For instance, Strang (1983) posed a continuous max-flow problem in a domain, where the goal is to identify a function that satisfies continuous generalizations of the flow conditions. As a quick example of these generalizations, recall that the cut of a set S can be computed as $\|Bx\|_{C,1}$. The TV of an indicator function for a set generalizes the cut quantity to a continuous domain. This connection, and its relationship to sharp boundaries, motivates TV image denoising (Rudin, Osher, and Fatemi, 1992) as well as ideas of continuous minimum cuts (Chan, Esedoḡlu, and Nikolova, 2006). Continued development of the theory (Strang, 2010) has led to interesting new connections between the infinite-dimensional and finite-dimensional cases (Yuan, Bae, and Tai, 2010). There are strong connections in motivation between our cluster improvement framework and finding optimal continuous functions



(a) Original (b) Boundary blur (c) Blur and noise (d) MQI-like ($\delta = 0.11$) solution (e) MQI-like ($\delta = 0.04$) solution

Fig. 4 *An example of using MQI-like procedures to reconstruct a binary image* (a) *from a blurry* (b)*, noisy* (c) *sample. Here, the result set is a binary image, which is a set in the grid graph. The value δ is from the fractional programming subproblem* (3.3) *with a custom denominator term as described in the reproduction details. Using $\delta = 0.11$ produces* 209 *error pixels around the boundary* (d)*. Reducing δ to* 0.04 (e) *produces a convex shape due to the conductance-like bias in this setting where convex shapes are optimal for isoperimetric-like objectives on grids.*

in these settings—for example, we can think of sharpening a blurry, noisy image as improving a cluster (see Figure 4)—but the details of the algorithms and data are markedly different. In particular, we think of the cluster improvement routine largely as a strongly local operation. Understanding how these ideas generalize to continuous or infinite-dimensional scenarios is an important problem raised by our approach.

4.8. Graph Cuts and Max-Flow-Based Image Segmentation. One final application of maximum flows is graph cut–based image processing (Boykov and Veksler, 2006; Marlet, 2017). The general setting in which these arise is an energy minimization framework (Greig, Porteous, and Seheult, 1989; Kolmogorov and Zabih, 2004) with binary variables. The goal is to identify a binary latent feature in an image as an exact or approximate solution of an optimization problem. An extremely large and useful class of these energy functions can be solved via a single or a sequence of max-flow computations. The special properties of the max-flow problems on image-like data motivated the development of specialized max-flow solvers that, empirically, have runtimes that scale linearly in the size of the data (Boykov and Kolmogorov, 2004).

This methodology has a number of applications in image segmentation in two- and three-dimensional images (Boykov and Funka-Lea, 2006) such as MRIs. For instance, one task in medical imaging is separating water from fat in an MRI, for which a graph cut–based approach is highly successful (Hernando et al., 2010). More recently, deep learning–based methods have often provided a substantial boost in performance for image processing tasks. Even these, however, benefit from a cluster improvement perspective. Multiple papers have found that postprocessing or refining the output of a convolutional neural net using a graph cut approach yields improved results in segmenting tumors (Ullah et al., 2018; Ma et al., 2018). These recent applications are an extremely close fit for our cluster improvement framework, where the goal is to find a small object in a big network starting from a good reference region. We often illustrate the benefits of and differences between our methodologies with the closely related problem of refining a local image segmentation output, e.g., in Figure 3.

Part II. Technical Details behind the Main Theoretical Results.

5. Minimum Cut and Maximum Flow Problems. As a simple introduction to our presentation of the technical details of MQI, FlowImprove, and LocalFlowImprove, we will start with the min-cut and max-flow problems. We will review the basics of

these problems from an optimization and duality perspective. This is because our technical discussions in subsequent sections will involve related, but more intricate, transformations and will use max-flow problems as subroutines. To simplify the text, we use the names MinCut and MaxFlow to refer to the s-t min-cut and s-t max-flow problems, which are the fully descriptive terms for these problems.

5.1. MinCut. Given a graph $G = (V, E)$, let s and t be two special nodes where s is commonly called the *source node* and t is the *sink node*. The undirected MinCut problem is

$$\begin{array}{ll} \underset{S}{\text{minimize}} & \text{cut}(S, \bar{S}) \\ \text{subject to} & s \in S, t \in \bar{S},\ S \subseteq V. \end{array} \tag{5.1}$$

The objective function of the MinCut problem measures the sum of the weights of edges between the sets S and $\bar{S}$. The constraints encode the idea that we want to separate the source from the sink and so we want the source node s to be in S and the sink node t to be in $\bar{S}$. Putting the objective function and the constraints together, we see that the purpose of the MinCut problem is to find a partition $(S, \bar{S})$ that minimizes the number of edges needed to separate node s from node t. As an example, see Figure 5, where we demonstrate the optimal partition for the MinCut problem (5.1) on a toy graph.

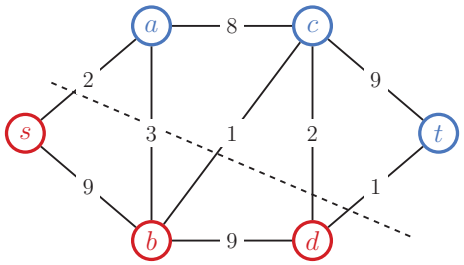


Fig. 5 *Demonstration of the optimal MinCut solution of problem* (5.1). *The numbers show the weight of each edge. The red nodes (s, b, d) and the blue nodes (a, c, t) denote the optimal partitions* $(S, \bar{S})$, *respectively, for problem* (5.1). *The black dashed line denotes the edges that are being cut, i.e., the edges that cross the partition between* S *and* $\bar{S}$. *The optimal objective value of problem* (5.1) *for this example is equal to* 9.

We can express the MinCut problem in other equivalent ways, some of which are more convenient for analysis and implementations. For example, we use indicator vector notation and the incidence matrix from section 2 to represent problem (5.1) as

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} \\ \text{subject to} & x_s = 1, x_t = 0, x \in \{0, 1\}^n. \end{array} \tag{5.2}$$

Expressing the MinCut problem with this notation will be especially useful later when we develop a unified framework for many cluster improvement algorithms. In practice, when implementing a solver for this problem, we need not take the binary constraints into account. This is because we can relax them *without changing the objective value* to obtain the following equivalent form of the MinCut problem:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} \\ \text{subject to} & x_s = 1, x_t = 0, x \in \mathbb{R}^n. \end{array} \tag{5.3}$$

It can be shown that there exists a solution to (5.3) that has the same objective function value as the optimal solution of (5.2). Given any solution to the relaxed problem, the integral solution can be obtained by an exact rounding procedure. In that sense, the relaxed problem (5.3) and the integral problem (5.2) are equivalent (Papadimitriou and Steiglitz, 1982). In the next subsection, we will obtain a solution to (5.2) through the MaxFlow problem.

5.2. Network Flow and MaxFlow. We provide a basic definition of a network flow, which is crucial for defining MaxFlow. For more details about network flows we recommend reading the notes of Trevisan (2011).

Network flows are commonly defined on directed graphs. Given an undirected graph, we will simply allow flow to go in both directions of an edge. This means that instead of doubling the number of edges, which is a common technique in the literature, we fix an arbitrary direction of the edges, encoded in the B matrix, and let flow go in either direction by simply allowing the flow variables to be negative. Also, in the context of flows, edge weights are usually called edge capacities. We will use these terms interchangeably, but we tend to use capacities when discussing flow and weights when discussing cuts.

A network flow is a mapping that assigns values to edges, i.e., a mapping $f : E \to \mathbb{R}$ from the set of edges E to $\mathbb{R}$, which also satisfies capacity and flow conservation constraints. We view f as a vector that encodes this mapping for a fixed ordering of the edges consistent with the incidence matrix. The capacity constraints are easy to state. Let $c = \mathrm{diag}(C)$ be the capacity for each edge; then we need

$$-c \leq f \leq c$$

so that the flow along an edge is bounded by its respective capacity. The flow preservation constraints ensure that flow is only created at the source and removed at the sink and that all other nodes neither create nor destroy flow. This can be evaluated using the incidence matrix that, given a flow f mapping, computes the changes via $B^T f$. Consequently, flow conservation is written

$$B^T f = q - p,$$

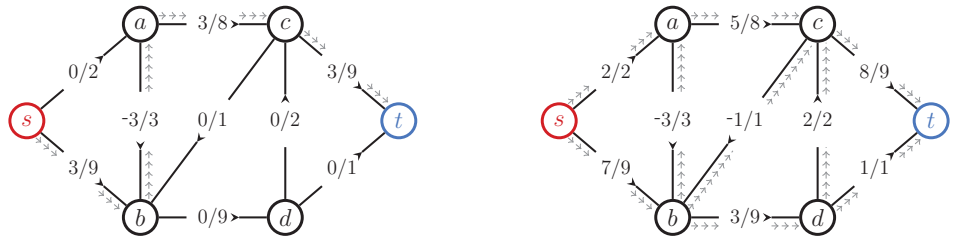
where $p_s \in \mathbb{R}, p_i = 0$ for all $i \in V \backslash \{s\}$ and $q_t \in \mathbb{R}, q_i = 0$ for all $i \in V \backslash \{t\}$.

The max-flow problem is to compute a feasible network flow with the maximum amount of flow that emerges from the source and reaches the sink. The corresponding MaxFlow optimization problem can be expressed as

$$\begin{array}{ll} \underset{f,p,q}{\text{maximize}} & p^T \mathbf{1}_s \\ \text{subject to} & B^T f = q - p, \\ & p_s \in \mathbb{R}, p_i = 0 \ \forall i \in V \backslash \{s\}, \\ & q_t \in \mathbb{R}, q_i = 0 \ \forall i \in V \backslash \{t\}, \\ & -c \leq f \leq c. \end{array} \tag{5.4}$$

See Figure 6 for a visual depiction of the flow variables and the optimal solution of problem (5.4) for the same graph used in Figure 5.

We will obtain the MaxFlow problem (5.4) by computing the Lagrange dual of the relaxed MinCut problem (5.3). For basics about Lagrangian duality, we refer the reader to Chapter 5 in Boyd and Vandenberghe (2004). The process of obtaining the dual of a problem is important, because it will allow us to understand how to



(a) Demonstration of a (nonoptimal) flow

(b) Demonstration of the MaxFlow solution

Fig. 6 *In this figure, all edges are undirected edges but each edge has an arrow in the middle indicating the positive direction of flow. A negative flow value on an edge means that the flow is flowing against the positive direction. The numerators in each expression show the flow that passes through an edge, and the denominators show the capacity of each edge. In* (a)*, we demonstrate a flow that starts from the source node* s *and ends at the sink node* t *and has value equal to* 3*. The path of that flow is highlighted by gray dashed arrows and includes nodes* s*,* b*,* a*,* c*, and* t*. Note that the flow in* (a) *is not optimal since we can send more flow from the source to the sink while satisfying the constraints of problem* (5.4)*. The optimal solution of the MaxFlow problem* (5.4) *for this toy graph is shown in* (b)*. The optimal flow that can be sent from the source to the sink is equal to* 9*.*

implement flow-based clustering methods in subsequent sections. First, we will convert problem (5.3) into an *equivalent* linear program

$$\begin{aligned} \underset{x,u,v}{\text{minimize}} \quad & c^T u + c^T v \\ \text{subject to} \quad & Bx = u - v, \\ & x_s = 1, x_t = 0, x \in \mathbb{R}^n, \\ & u, v \geq 0. \end{aligned} \tag{5.5}$$

This can be done by starting with problem (5.3) and following standard steps in the conversion of a linear program into standard form. Here, this involves introducing nonnegative variables u and v such that $Bx = u - v$ and then writing the objective as above. (Note that due to the minimization, at optimality, we will never have both u and v nonzero in the same index.) Consequently, the Lagrangian function of problem (5.5) is given by

$$\begin{aligned} L(u, v, x, f, s, g, p, q) &= c^T u + c^T v - f^T(Bx - u + v) - s^T u - g^T v \\ &\quad - p^T(x - \mathbf{1}_s) + q^T x \\ &= (f - s + c)^T u + (-f - g + c)^T v + (-B^T f - p \\ &\quad + q)^T x + p^T \mathbf{1}_s, \end{aligned} \tag{5.6}$$

where $s, g \geq 0$, $f \in \mathbb{R}^m$, $p_s \in \mathbb{R}$ and $p_i = 0$ for all $i \in V \backslash \{s\}$, and $q_t \in \mathbb{R}$ and $q_i = 0$ for all $i \in V \backslash \{t\}$. The latter constraints are important for Lagrangian duality because they guarantee that the dual function (that we derive below) will provide a lower bound for the optimal solution of the primal problem (5.3). See Chapter 5 in Boyd and Vandenberghe (2004). The dual function is

$$h(f, s, g, p, q) := \min_{u,v,x} \; L(u, v, x, f, s, g, p, q). \tag{5.7}$$

Note that the Lagrangian function L is a linear function with respect to u, v, x. Therefore, we can obtain an analytic form for the dual function by requiring the partial derivatives of L with respect to u, v, and x to be zero. The following three equations arise from the latter process:

$$B^T f + p - q = 0_n, \qquad f - s + c = 0_n, \qquad -f - g + c = 0_n.$$

By substituting these conditions into (5.6), we have

$$h(f, s, g, p, q) = p^T \mathbf{1}_s, \tag{5.8}$$

with a domain that is defined by the constraints

$$\begin{array}{ccc} B^T f + p - q = 0_n, & f - s + c = 0_n, & -f - g + c = 0_n, \\ p_s \in \mathbb{R}, p_i = 0 \ \forall i \in V \backslash \{s\}, & q_t \in \mathbb{R}, q_i = 0 \ \forall i \in V \backslash \{t\}, & s, g \geq 0. \end{array}$$

Thus, we obtain that the dual problem of problem (5.5) is

$$\begin{array}{ll} \underset{f,s,g,p,q}{\text{maximize}} & p^T \mathbf{1}_s = h(f, s, g, p, q) \\ \text{subject to} & B^T f = q - p, \\ & f - s + c = 0, \\ & -f - g + c = 0, \\ & p_s \in \mathbb{R}, p_i = 0 \ \forall i \in V \backslash \{s\}, \\ & q_t \in \mathbb{R}, q_i = 0 \ \forall i \in V \backslash \{t\}, \\ & s, g \geq 0. \end{array} \tag{5.9}$$

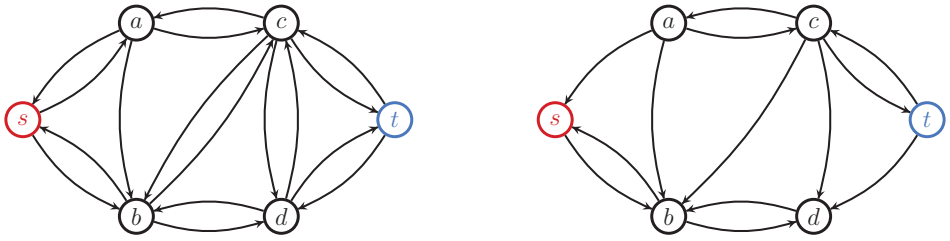
By eliminating the variables s and g we obtain the MaxFlow problem (5.4). (These correspond to *slack* variables associated with $-c \leq f \leq c$.)

Both the primal (5.5) and the dual (5.9) are feasible (with a trivial cut and a zero flow, respectively) and also have finite solutions (0 is a lower bound on the cut and $\mathrm{vol}(G) = 1^T c$ is an upper bound on the flow). So, strong duality will hold between the two solutions at optimality, and the optimal value of the MaxFlow problem (5.4) is equal to the optimal value of the relaxed MinCut problem (5.3) (which is equal to the optimal value of (5.2)). This fact is often one component of the so-called MaxFlow-MinCut Theorem. Another important component is discussed next.

5.3. From MaxFlow to MinCut. Assume that we have solved the MaxFlow problem to optimality and that we have obtained the optimal flow f. Then the MaxFlow-MinCut Theorem is a statement about the equivalence between the objective function value of the optimal solution to the MinCut problem (5.1) and the objective function value of the optimal solution of the MaxFlow problem (5.4). In many cases, obtaining this quantity suffices, but in some cases, we want to work with the actual solutions themselves.

To obtain the optimal MinCut solution from an optimal MaxFlow solution, we define the notion of a *residual graph*. A residual graph G_f of a given G has the same set of nodes as G, but for each edge $e_{ij} \in E$, it has a forward edge $\tilde{e}_{ij}$, i.e., from node i to node j, with capacity $\max(c_{ij} - f_{ij}, 0)$ and a backward edge $\hat{e}_{ji}$, i.e., from node j to node i, with capacity $\max(f_{ij}, 0)$, where f is the optimal solution of the MaxFlow problem (5.4). A depiction of a residual graph for a given flow is shown in Figure 7.

Note that there cannot exist a path from s to t in the residual graph at a max-flow solution (otherwise, we would be able to increase the flow!). Consequently, we can look at the set S of vertices that are reachable starting from the source node s (this



(a) The residual graph of the nonoptimal flow Figure 6(a)

(b) The residual graph of the MaxFlow solution Figure 6(b)

Fig. 7 *The two subfigures show the directed residual graph for the flows from Figure* 6*. The edge capacities are removed for simplicity. Edges are only shown if they have positive capacity. Note that the flow in Figure* 6(a) *is not optimal since we can send more flow from the source to the sink while satisfying the constraints of problem* (5.4)*; this is equivalent to having an* s *to* t *path in the residual graph in* (a)*. In* (b)*, we show the corresponding residual graph for the optimal flow and note that in the residual graph of the MaxFlow solution there is no path from the source node to the sink node.*

can be algorithmically identified using a breadth-first or depth-first search starting from s). It is now a standard textbook argument that the cut of the set S, which does not contain t, is equal to the maximum flow.

5.4. MaxFlow Solvers for Weighted and Unweighted Graphs. MaxFlow problems can be solved substantially faster than general linear programs. See our discussion in section 4.6 for more information on state-of-the-art solvers.

It is often assumed that the graphs are unweighted or have integer positive weights. All of the MaxFlow problems we need to solve will be weighted with rational weights that depend on the current estimate of the ratio in the fractional programming problem. Many of the same algorithms can be applied for weighted problems as well. We explicitly mention both Dinic's algorithm (Dinitz, 1970) and the Push–Relabel algorithm (Goldberg and Rao, 1998), both of which can be implemented for the types of weighted graphs we need. In our implementations, we use Dinic's algorithm. In these cases, however, the runtime becomes slightly tricky to state and is fairly pessimistic. Consequently, when we have a runtime that depends on MaxFlow, we simply state the number of edges involved in the computation as a proxy for the runtime.

6. The MQI Problem and Algorithm. In this section, we will describe the MaxFlow Quotient-Cut Improvement (MQI) algorithm, due to Lang and Rao (2004). This cluster improvement method takes as input a graph $G = (V, E)$ and a reference set $R \subset V$, with $\mathrm{vol}(R) \leq \mathrm{vol}(G)/2$, and it returns as output an "improved" cluster, in the sense that the output is a subset of R of minimum conductance.

The basic MQI problem is

$$\begin{array}{ll} \underset{S}{\text{minimize}} & \dfrac{\mathrm{cut}(S)}{\mathrm{vol}(S)} \\ \text{subject to} & S \subseteq R. \end{array} \tag{6.1}$$

Due to the assumption that $\mathrm{vol}(R) \leq \mathrm{vol}(G)/2$, this problem is equivalent to

$$\begin{array}{ll} \underset{S}{\text{minimize}} & \phi(S) \\ \text{subject to} & S \subseteq R. \end{array} \tag{6.2}$$

In the equivalence with conductance, this constraint that $\mathrm{vol}(R) \leq \mathrm{vol}(G)/2$ is crucial because it makes the problem polynomially solvable. Without this constraint, the problem with conductance is intractable, but we can still minimize the cut to volume ratio even when $\mathrm{vol}(R) > \mathrm{vol}(G)/2$.

ASIDE 6. *A curious implication of the MQI objective is that it is NP-hard to find a set S with* $\mathrm{vol}(S) \leq \mathrm{vol}(G)/2$ *that even* contains *the set of minimum conductance.*

Recall that this MQI problem is related to the fractional programming problem (3.2) by setting $g(S) := \mathrm{vol}(S)$ and $Q = R$. Lang and Rao (2004) describe an algorithm to solve the MQI problem which is equivalent to what is presented as Algorithm 6.1 (they describe solving (6.3) via the flow procedure we will highlight shortly). It is easy to see that this algorithm is simply Algorithm 3.1 for fractional programming specialized to this scenario. Consequently, we can apply our standard theory.

Algorithm 6.1 MQI (Lang and Rao, 2004).

1: Initialize $k := 1$, $S_1 := R$, and $\delta_1 := \phi(S_1)$.
2: **while** we have not exited via else clause **do**
3: Solve $S_{k+1} := \mathrm{argmin}_{S \subseteq R} \mathrm{cut}(S) - \delta_k \mathrm{vol}(S)$
4: **if** $\phi(S_{k+1}) < \delta_k$ **then**
5: $\delta_{k+1} := \phi(S_{k+1})$
6: **else**
7: δ_k is optimal, return previous solution S_k.
8: $k := k + 1$

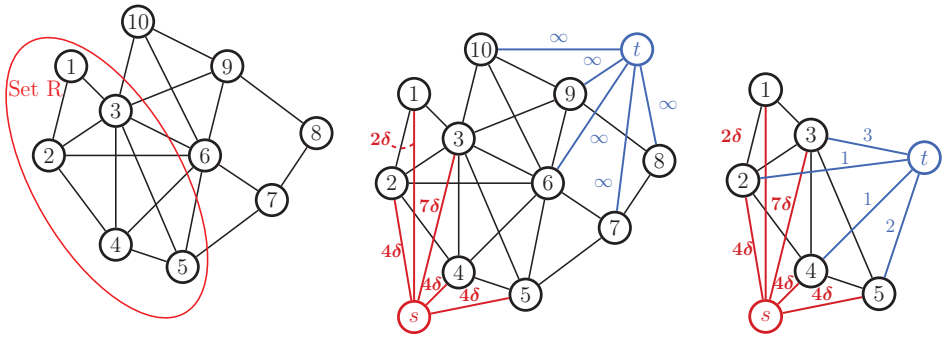
The following theorem implies that MQI monotonically decreases the objective function in problem (6.1) at each iteration. It was first shown by Lang and Rao (2004), but it is a corollary of Theorem 3.4. Note that δ_k is equal to the objective function of problem (6.1) evaluated at S_k.

THEOREM 6.1 (convergence of MQI). *Let G be an undirected, connected graph with nonnegative weights and let R be a subset of vertices with* $\mathrm{vol}(R) \leq \mathrm{vol}(\bar{R})$. *The sequence δ_k monotonically decreases at each iteration of MQI.*

6.1. Solving the MQI Subproblem Using MaxFlow Algorithms. In this subsection, we will discuss how to solve efficiently the subproblem at step 3 of MQI Algorithm 6.1, namely,

$$\begin{array}{ll} \mathrm{argmin}_S & \mathrm{cut}(S) - \delta\, \mathrm{vol}(S) \\ \text{subject to} & S \subseteq R. \end{array} \tag{6.3}$$

To summarize this subsection, the subproblem corresponds to a MinCut-like problem and by introducing a number of modifications, we can turn it into an instance of a MinCut problem. This enables us to use MaxFlow solvers to compute a binary solution efficiently. The final solver will run a MaxFlow problem on the subgraph of G induced by R along with a few additional edges.



(a) Graph and seed set R (b) Nonlocal MinCut problem (c) Local MinCut problem

Fig. 8 *Illustration of the augmented graph for solving the MQI subproblem. Panel* (a) *illustrates a small graph and a seed set R denoted by the red ellipse. This set includes nodes with IDs* 1 *to* 5. *Panel* (b) *demonstrates the addition of a source node s and sink node t that involves the entire graph but solves the subproblem. Panel* (c) *illustrates the collapse of all nodes in $\bar{R}$ into a single sink node t. Edges from R to $\bar{R}$ are maintained with the same weights but they are rewired to the sink node t. The final MinCut problem in* (c) *can be solved via a MaxFlow problem from the source to the sink.*

By translating problem (6.3) into indicator notation, we have

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} - \delta x^T d \\ \text{subject to} & x_i = 0 \ \forall i \in \bar{R}, x \in \{0,1\}^n. \end{array} \tag{6.4}$$

This is not a MinCut problem as stated, but there exists an equivalent problem that is a MinCut problem. To generate this problem, we'll go through two steps. First, we'll shift the objective to be nonnegative. This is necessary because a MinCut problem always has a nonnegative objective. Second, we'll introduce a source and sink to handle the terms that are not of the form $\|Bx\|_{C,1}$ and the equality constraints. Again, this step is necessary because these problems must have a source and sink.

For step 1, note that the maximum negative term is $\delta \mathbf{1}^T d$. (It's actually smaller due to the equality constraints, but this overestimate will suffice.) Thus, we shift the objective by this value and regroup terms:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} + \delta(\mathbf{1} - x)^T d \\ \text{subject to} & x_i = 0 \ \forall i \in \bar{R}, x \in \{0,1\}^n. \end{array} \tag{6.5}$$

Note that $\mathbf{1} - x$ is simply an indicator for $\bar{S}$, the complement solution set. Consequently, we want to introduce a penalty for each node placed in $\bar{S}$. To do so, we introduce a source node s that will connect to each node of the graph with weight proportional to the degree of each node. (A penalty for $\bar{S}$ corresponds to an edge from the source s.) Since nothing in $\bar{R}$ can be in the solution, we can introduce a sink node t and connect it with infinite weight edges to each node in $\bar{R}$. Thus, these edges will never be cut at optimality, as a finite-valued objective is possible. Also note that the infinite weight can be replaced by a sufficiently large graph-dependent weight to achieve the same effect.

This MinCut construction is given in Figure 8(b), although this omits the edges from s to nodes in $\bar{R}$. This construction, however, is not amenable to a strongly local

Augmented Graph 1 for the subproblem at step 3 of MQI Algorithm 6.1.

1: Extract the subgraph with nodes in R and the edges of these nodes, which we denote by $E(R)$.
2: Add to the set of nodes R a source node s and a sink node t.
3: Add to the set of edges $E(R)$ an edge from the source node s to every node in the seed set of nodes R with weight the degree of that node times δ.
4: For any edge in G from R to $\bar{R}$, rewire it to node t and combine multiple edges by summing their weights.

solution method, as it naively involves the entire graph. Note that, in practice, we can form the graph construction in Figure 8(c) with the collapsed vertices without ever examining the whole graph.

To generate a strongly local method, note that we can collapse all the vertices in $\bar{R}$ and t into a single supersink t. This simply involves rewiring all edges (u, v) where $u \notin \bar{R}$ and $v \in \bar{R}$ into a new edge (u, t), where we handle multiedges by summing their weights. This results in a number of s to t edges, one for each node in $\bar{R}$, which we can further delete as they exert a constant penalty of $\delta \operatorname{vol}(\bar{R})$ on the final objective. An illustration is given in Figure 8(c), where importantly there are only a small number of nodes in $\bar{R}$ that are collapsed into the sink node t, but $\bar{R}$ could have had thousands or millions or billions of nodes. In that case, the final graph would still have only a very small number of nodes, in which case strongly local algorithms would be *much* faster.

To recap, see the Augmented Graph 1 procedure shown above. We now give an explicit instance of the MinCut problem to illustrate how it maps to our desired binary objective. Let $B(R)$ and $C(R)$ be the incidence and weight matrices for the subgraph induced by the set R. Then consider the incidence and diagonal edge-weight matrices of the modified graph, which are

$$\tilde{B} := \begin{array}{c} \begin{matrix} s & R & t \end{matrix} \\ \begin{bmatrix} \mathbf{1} & -I & 0 \\ 0 & B(R) & 0 \\ 0 & I & -\mathbf{1} \end{bmatrix} \end{array}, \qquad \tilde{C} := \begin{bmatrix} \delta D_R & 0 & 0 \\ 0 & C(R) & 0 \\ 0 & 0 & Z \end{bmatrix},$$

where D_R is the submatrix of D that corresponds to nodes in R (ordered conformally), and Z is a diagonal matrix that stores the weights of the rewired edges from R to the sink t, i.e.,

$$Z_{ii} = \sum_e c_e, \text{ where } e \text{ is an edge from } i \in R \text{ to any node in } \bar{R}\ .$$

(These weights can be zero if there are no edges leaving from a node $i \in R$.) The first column of matrix $\tilde{B}$ corresponds to the source node, the last column corresponds to the sink node, and all other columns in between correspond to nodes in R. The first block δD_R in $\tilde{C}$ corresponds to edges from the source to nodes in R, the second block C_R in $\tilde{C}$ corresponds to edges from R to R, and the third block Z in $\tilde{C}$ corresponds to edges from nodes in R to the sink node t. Let

$$\tilde{x} := \begin{bmatrix} x_s \\ x_R \\ x_t \end{bmatrix}, \text{ so that } \tilde{x}_1 = x_s \text{ and } \tilde{x}_{|R|+2} = x_t;$$

then the MinCut problem with respect to the modified graph is

$$\begin{array}{ll} \underset{\tilde{x}}{\text{minimize}} & \|\tilde{B}\tilde{x}\|_{\tilde{C},1} = \|B(R)x_R\|_{C(R),1} + \delta \mathbf{1}^T D_R(\mathbf{1}_R - x_R) + \mathbf{1}^T Z x_R \\ \text{subject to} & \tilde{x}_1 = 1, \tilde{x}_{|R|+2} = 0, \tilde{x}_i \in \{0,1\}. \end{array} \tag{6.6}$$

It is straightforward to verify that problem (6.6) is equivalent to a shifted version of problem (6.5), where the objectives differ by $\delta \operatorname{vol}(\bar{R})$. Finally, to obtain a solution of the original problem, we have to further decrease the objective by the constant $\delta \operatorname{vol}(R)$.

To solve this MinCut problem, we then simply use an undirected MaxFlow solver. The input has $\mathcal{O}(\operatorname{vol}(R))$ edges and $|R| + 2$ nodes.

6.2. Iteration Complexity. We now specialize our general analysis in section 3 and present an iteration complexity result for Algorithm 6.1. First, we present Lemma 6.2, which will be used in the iteration complexity result in Theorem 6.3.

An interesting property of MQI that is shown in Lemma 6.2 is that the volume of S_k monotonically decreases at each iteration, i.e., $\operatorname{vol}(S_{k+1}) < \operatorname{vol}(S_k)$. This result has important *practical* implications since it shows that MQI is searching for subsets S that have *smaller* volume than the set S_1. Moreover, Lemma 6.2 shows that the numerator of problem (6.1) decreases monotonically.

LEMMA 6.2. *Let G be an undirected, connected graph with nonnegative weights. If the MQI algorithm proceeds to iteration $k+1$, it satisfies both* $\operatorname{vol}(S_{k+1}) < \operatorname{vol}(S_k)$ *and* $\operatorname{cut}(S_{k+1}) < \operatorname{cut}(S_k)$.

Proof. The proof for this claim is given in the proof of Lemma 3.5. ∎

THEOREM 6.3 (iteration complexity of MQI). *Let G be a connected, undirected graph with nonnegative integer weights. Algorithm* 6.1 *has at most* $\operatorname{cut}(R)$ *iterations before converging to a solution.*

Proof. This is just an explicit specialization of Theorem 3.6. ∎

Remark 6.4 (time per iteration). At each iteration a weighted MaxFlow problem is being solved. Therefore, the worst-case time of MQI will be its iteration complexity times the cost of computing a MaxFlow on a graph of size $\operatorname{vol}(R)$. Here $\operatorname{vol}(R)$ is an upper bound on the number of edges incident to vertices in R because the weights are integers.

6.3. A Faster Version of the MQI Algorithm. The original MQI algorithm requires at most $\operatorname{vol}(R)$ iterations to converge to the optimal solution for graphs with integer weights. After at most that many iterations the algorithm returns the *exact* output. However, if we are not interested in exact solutions, we can improve the iteration complexity of MQI to at most $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$, where $\varepsilon > 0$ is an accuracy parameter. To achieve this we will use binary search for the variable δ. It is true that for (S^*, δ^*) we have $\operatorname{cut}(S^*)/\operatorname{vol}(S^*) = \delta^*$. Therefore, $\delta^* \in [0,1]$. We will use this interval as our search space for the binary search. The modified algorithm is shown in Algorithm 6.2. This algorithm is an instance of Algorithm 3.2. Note that the subproblem in step 4 of Algorithm 6.2 is the same as the subproblem in step 3 of the original Algorithm 6.1. The only part that changes is that we introduce binary search for δ.

Putting the iteration complexity of Fast MQI together with its per iteration computational complexity, we get the following theorem.

THEOREM 6.5 (iteration complexity of the Fast MQI Algorithm 6.2). *Let G be an undirected, connected graph with nonnegative weights. Let R be a subset of vertices*

Algorithm 6.2 Fast MQI.

Initialize $k := 1$, $\delta_{\min} := 0$, $\delta_{\max} := \phi(R)$, and $\varepsilon \in (0, 1]$
while $\delta_{\max} - \delta_{\min} > \varepsilon\delta_{\min}$ **do**
  $\delta_k := (\delta_{\max} + \delta_{\min})/2$
  Solve $S_{k+1} := \operatorname{argmin}_{S \subseteq R} \operatorname{cut}(S) - \delta_k \operatorname{vol}(S)$ via MaxFlow on Augmented Graph 1.
  if $\operatorname{vol}(S_{k+1}) > 0$ {Then δ_k is above δ^*} **then**
    $\delta_{\max} := \phi(S_{k+1})$, and set $S_{\max} = S_{k+1}$ {Note $\phi(S_{k+1}) \le \delta_k$}
  else
    $\delta_{\min} := \delta_k$
  $k := k + 1$
Return $\operatorname{argmin}_{S \subseteq R} \operatorname{cut}(S) - \delta_{\max} \operatorname{vol}(S)$ or $S_{\max}$ based on minimum conductance.

with $\operatorname{vol}(R) \le \operatorname{vol}(\bar{R})$. *The sequence* δ_k *of Algorithm* 6.2 *converges to an approximate solution* $|\delta^* - \delta_k|/\delta^* \le \varepsilon$ *in* $\mathcal{O}(\log 1/\varepsilon)$ *iterations, where* $\delta^* = \phi(S^*)$ *and* S^* *is an optimal solution to problem* (6.1). *Moreover, if* G *has nonnegative integer weights, then the algorithm will return the exact minimizer when* $\varepsilon < \frac{1}{\operatorname{vol}(R)^2}$.

Proof. The iteration complexity of MQI is an immediate consequence of Theorem 3.8. The exact solution result is a consequence of the smallest difference between values of conductance among subsets of R for integer-weighted graphs. Let S_1 and S_2 be arbitrary subsets of vertices in R with $\phi(S_1) > \phi(S_2)$. Then

$$\phi(S_1) - \phi(S_2) = \frac{\operatorname{cut}(S_1)\operatorname{vol}(S_2) - \operatorname{cut}(S_2)\operatorname{vol}(S_1)}{\operatorname{vol}(S_1)\operatorname{vol}(S_2)} \ge (\operatorname{vol}(R))^{-2}.$$

The last inequality occurs because if $\operatorname{cut}(S_1)\operatorname{vol}(S_2) - \operatorname{cut}(S_2)\operatorname{vol}(S_1)$ is an integer, the smallest possible difference is 1. At termination, Fast MQI satisfies $\delta_{\max} - \delta_{\min} \le \varepsilon\delta_{\min}$. By the above difference bound, the next objective function value that is larger than δ^* is at least $\delta^* + \frac{1}{\operatorname{vol}(R)^2}$. Therefore, setting $\varepsilon < \frac{1}{\operatorname{vol}(R)^2}$, we find that $\delta_{\max} < \delta^* + \frac{1}{\operatorname{vol}(R)^2}$. ∎

Remark 6.6 (time per iteration). Each iteration involves a weighted MaxFlow problem on a graph with volume equal to $O(\operatorname{vol}(R))$.

7. The FlowImprove Problem and Algorithm. In this section, we will describe the FlowImprove method, due to Andersen and Lang (2008). This cluster improvement method was designed to address the issue that the MQI algorithm will always return an output set that is strictly a subset of the reference set R. The FlowImprove method also takes as input a graph $G = (V, E)$ and a reference set $R \subset V$, with $\operatorname{vol}(R) \le \operatorname{vol}(G)/2$, and it also returns as output an "improved" cluster. Here, the output is "improved" in the sense that it is a set with conductance at least as good as R that is also highly correlated with R.

To state the FlowImprove method, consider the following variant of conductance:

$$\phi_R(S) = \begin{cases} \dfrac{\operatorname{cut}(S, \bar{S})}{\operatorname{rvol}(S; R, \theta)} & \text{when the denominator is positive,} \\ \infty & \text{otherwise,} \end{cases} \tag{7.1}$$

where $\theta = \operatorname{vol}(R)/\operatorname{vol}(\bar{R})$ and where the value is ∞ if the denominator is negative. This particular value of θ arises as the smallest value such that the $\operatorname{rvol}(S; R, \theta) =$

$\mathrm{vol}(S \cap R) - \theta\,\mathrm{vol}(S \cap \bar{R})$ denominator is exactly zero when $S = V$ and hence will rule out trivial solutions. (This idea is equivalent to picking θ so that the total weight of edges connected to the source is equal to the total weight of edges connected to the sink in the upcoming flow problem (Andersen and Lang, 2008).) Note that this setup is also equivalent to the statement in section 3.1 where the denominator constraint is adjusted to be a positive infinity value.

For any set S with $\mathrm{vol}(S) \leq \mathrm{vol}(\bar{S})$, since $\mathrm{rvol}(S; R, \theta) = \mathrm{vol}(S \cap R) - \theta\,\mathrm{vol}(S \cap \bar{R})$, it holds that $\phi_R(S) \geq \phi(S)$. Thus, this modified conductance score $\phi_R(\cdot)$ provides an upper bound on the true conductance score $\phi(\cdot)$ for sets that are not too big; but this objective provides a bias toward R, in the sense that the denominator penalizes sets S that are outside of the reference set R.

Consequently, the FlowImprove problem is

$$\boxed{\begin{array}{ll} \underset{S}{\text{minimize}} & \phi_R(S) \\ \text{subject to} & S \subset V. \end{array}} \tag{7.2}$$

This FlowImprove problem is related to the fractional programming problem (3.2) by setting $g(S) := \mathrm{vol}(S \cap R) - \theta\,\mathrm{vol}(S \cap \bar{R})$ and $Q = V$. Andersen and Lang (2008) describe an algorithm to solve the FlowImprove problem which is equivalent to what we present as Algorithm 7.1. It is easy to see that this algorithm is a special case of Algorithm 3.1 for general fractional programming.

Algorithm 7.1 FlowImprove (Andersen and Lang, 2008).

Initialize $k = 1$, $S_1 := R$, and $\delta_1 = \phi_R(S_1)$.
while we have not exited via else clause **do**
  Solve $S_{k+1} := \operatorname{argmin}_S \ \mathrm{cut}(S) - \delta_k \left(\mathrm{vol}(S \cap R) - \theta\,\mathrm{vol}(S \cap \bar{R})\right)$
  if $\phi_R(S_{k+1}) < \delta_k$ **then**
    $\delta_{k+1} := \phi_R(S_{k+1})$
  else
    δ_k is optimal, return previous solution S_k.
  $k := k + 1$

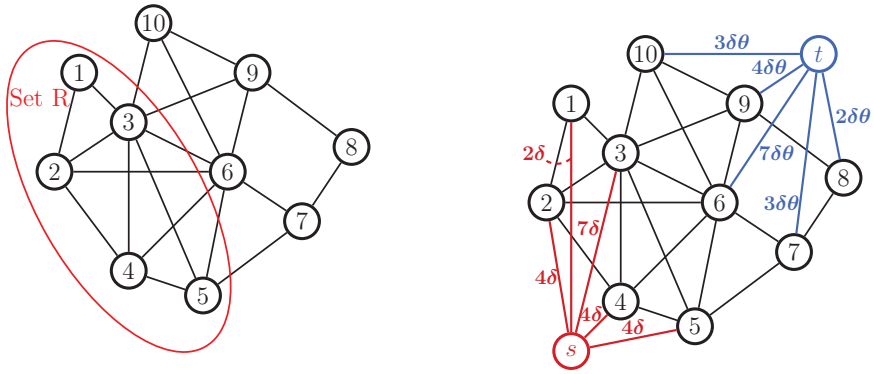
The following theorem implies that FlowImprove monotonically decreases the objective function in problem (7.2) at each iteration. It was first shown by Andersen and Lang (2008), but it is a corollary of Theorem 3.4. Note that δ_k is equal to the objective function of problem (7.2) evaluated at S_k.

THEOREM 7.1 (convergence of FlowImprove). *Let G be an undirected, connected graph with nonnegative weights. Let R be a subset of vertices with* $\mathrm{vol}(R) \leq \mathrm{vol}(\bar{R})$. *The sequence δ_k monotonically decreases at each iteration of FlowImprove (Algorithm* 7.1*).*

7.1. The FlowImprove Subproblem. In this subsection, we will discuss how to solve efficiently the subproblem at step 3 of FlowImprove. We will follow similar steps to those for MQI in section 6.1; that is, we convert the MinCut-like problem into a true MinCut problem on an augmented graph, and then we use MaxFlow to find the set minimizing the objective. As a summary and overview, see the Augmented Graph 2 procedure and an example of this new modified graph in Figure 9. (Observe that here we do *not* have a fourth step where we combine multiple edges, as we did in Augmented Graph 1 and Figure 8(c)—thus, the FlowImprove Algorithm 7.1 will *not* be strongly local.)

Augmented Graph 2 for the subproblem at step 3 of FlowImprove Algorithm 7.1.

1: Add to the set of nodes V a source node s and a sink node t.
2: Add to the set of edges E an edge from the source node s to every node in the seed set of nodes R with weight the degree of that node times δ.
3: Add to the set of edges E an edge from the sink node t to every node in the set of nodes $\bar{R}$ with weight the degree of that node times $\delta\theta$.



(a) Graph and seed set R (b) MinCut graph for FlowImprove subproblem

Fig. 9 *Illustration of the augmented graph for solving the FlowImprove subproblem. Panel* (a) *illustrates the same graph and seed set from Figure* 8. *Panel* (b) *demonstrates the addition of a source node* s *and a sink node* t*, along with corresponding edges from node* s *to nodes in* R *and node* t *to every node in* $\bar{R}$*. The MinCut problem in panel* (b) *can be solved to identify a set via a MaxFlow problem from the source to the sink.*

Turning back to the derivation of this formulation, the MinCut subproblem at step 3 of the FlowImprove problem is equivalent to

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} - \delta x^T \hat{d}_R + \delta\theta x^T \hat{d}_{\bar{R}} \\ \text{subject to} & x \in \{0,1\}^n, \end{array} \tag{7.3}$$

where $\hat{d}_R$ is an n-dimensional vector that is equal to d for components with index in R and zero elsewhere. Similarly for $\hat{d}_{\bar{R}}$. Consequently, $d = \hat{d}_R + \hat{d}_{\bar{R}}$, where the two pieces have disjoint support.

As with the previous case, we shift this and then add sources and sinks. First, the largest possible negative value is at least $\delta \mathbf{1}^T \hat{d}_R$. Adding this yields

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} + \delta(\mathbf{1} - x)^T \hat{d}_R + \delta\theta x^T \hat{d}_{\bar{R}} \\ \text{subject to} & x \in \{0,1\}^n. \end{array} \tag{7.4}$$

Again, we have penalty terms associated with S (given by nonzero entries of x) and $\bar{S}$ (given by nonzero entries of $\mathbf{1} - x$). For these, we introduce a source and sink. The source connects to penalties associated with $\bar{S}$ and the sink connects to penalties associated with S. Note that these penalties partition into two groups, associated with R and $\bar{R}$. Consequently, we add a source node s and connect it to all nodes in R with weight $\delta\hat{d}_R$, and we also add a sink node t and connect it to all nodes in $\bar{R}$ with weight $\delta\theta\hat{d}_{\bar{R}}$.

The resulting MinCut problem is associated with the incidence matrix and the diagonal edge-weight matrix of a modified problem as follows:

$$\tilde{B} := \begin{matrix} \\ \mathbf{1} \\ 0 \\ 0 \end{matrix} \begin{matrix} s \\ \begin{bmatrix} \mathbf{1} \\ 0 \\ 0 \end{bmatrix} \end{matrix} \begin{matrix} V \\ \begin{bmatrix} -I_R \\ B \\ I_{\bar{R}} \end{bmatrix} \end{matrix} \begin{matrix} t \\ \begin{bmatrix} 0 \\ 0 \\ -\mathbf{1} \end{bmatrix} \end{matrix}, \qquad \tilde{C} := \begin{bmatrix} \delta D_R & 0 & 0 \\ 0 & C & 0 \\ 0 & 0 & \delta\theta D_{\bar{R}} \end{bmatrix}.$$

Here, D_R and $D_{\bar{R}}$ are diagonal submatrices of D corresponding to nodes in R and $\bar{R}$, respectively, and I_R and $I_{\bar{R}}$ are matrices where each row contains an indicator vector for a node in R and $\bar{R}$, respectively. These matrices give $I_R x = x_R$ and $I_{\bar{R}} x = x_{\bar{R}}$ and are ordered in the same way as D_R and $D_{\bar{R}}$. Let

$$\tilde{x} := \begin{bmatrix} x_s \\ x \\ x_t \end{bmatrix}, \text{ so that } \tilde{x}_1 = x_s \text{ and } \tilde{x}_{|R|+2} = x_t;$$

then the MinCut problem with respect to the modified graph is

$$\begin{array}{ll} \underset{\tilde{x}}{\text{minimize}} & \|\tilde{B}\tilde{x}\|_{\tilde{C},1} = \|Bx\|_{C,1} + \delta(\mathbf{1} - x_R)^T d_R + \delta\theta x_{\bar{R}} d_{\bar{R}} \\ \text{subject to} & \tilde{x}_1 = 1, \tilde{x}_{n+2} = 0, \tilde{x}_i \in \{0,1\}\ \forall i = 2, \ldots, n+1. \end{array} \tag{7.5}$$

Again, note that this objective corresponds to a constant shift with respect to problem (7.3). This problem can be solved via MaxFlow to give a set solution.

7.2. Iteration Complexity. In Lemma 7.2 we show that when using FlowImprove, the denominator of problem (7.2), i.e., $\mathrm{vol}(S \cap R) - \theta\,\mathrm{vol}(S \cap \bar{R})$, decreases monotonically at each iteration. Moreover, the numerator of problem (7.2) decreases monotonically as well.

LEMMA 7.2. *If the FlowImprove algorithm proceeds to iteration $k+1$, it satisfies* $\mathrm{vol}(S_{k+1} \cap R) - \mathrm{vol}(S_k \cap R) < \theta\left(\mathrm{vol}(S_{k+1} \cap \bar{R}) - \mathrm{vol}(S_k \cap \bar{R})\right)$ *and* $\mathrm{cut}(S_{k+1}) < \mathrm{cut}(S_k)$.

Proof. This result is a specialization of Lemma 3.5 and the proof is the same. ∎

THEOREM 7.3 (iteration complexity of the FlowImprove Algorithm 7.1). *Let G be a connected, undirected graph with nonnegative integer weights. Then Algorithm* 7.1 *needs at most* $\mathrm{cut}(R)$ *iterations to converge to a solution.*

Proof. This is just an explicit specialization of Theorem 3.6. ∎

Remark 7.4 (time per iteration). At each iteration a weighted MaxFlow problem is being solved; see section 7.1. The MaxFlow problem size is proportional to the whole graph.

7.3. A Faster Version of the FlowImprove Algorithm. The original FlowImprove algorithm requires at most $\mathrm{cut}(R) \leq \mathrm{vol}(R)$ iterations to converge to the optimal solution. After at most that many iterations the algorithm returns the *exact* output. However, if we are not interested in exact solutions we can improve the iteration complexity of FlowImprove to at most $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$, where $\varepsilon > 0$ is an accuracy parameter. To achieve this we will use binary search for the variable δ. It is true that $\phi_R(R) \in [0,1]$, and therefore, $\delta^* \in [0,1]$. We will use this interval as our search space for the binary search. The modified algorithm is shown in Algorithm 7.2. Note that

the subproblem in step 4 of Algorithm 7.2 is the same as the subproblem in step 3 of the original Algorithm 7.1. The only part that changes is that we introduce binary search for δ.

Algorithm 7.2 Fast FlowImprove.

1: Initialize $k := 1$, $\delta_{\min} := 0$, $\delta_{\max} := 1$, and $\varepsilon \in (0, 1]$
2: **while** $\delta_{\max} - \delta_{\min} > \varepsilon\delta_{\min}$ **do**
3: $\quad \delta_k := (\delta_{\max} + \delta_{\min})/2$
4: $\quad$ Solve $S_{k+1} := \operatorname{argmin}_S \operatorname{cut}(S) - \delta_k \left(\operatorname{vol}(S \cap R) - \theta \operatorname{vol}(S \cap \bar{R})\right)$ via MaxFlow
5: $\quad$ **if** $\operatorname{vol}(S_{k+1} \cap R) > \theta \operatorname{vol}(S_{k+1} \cap \bar{R})$ {Then δ_k is above δ^*} **then**
6: $\quad\quad \delta_{\max} := \phi_R(S_{k+1})$ and set $S_{\max} := S_{k+1}$ {Note $\phi_R(S_{k+1}) \le \delta_k$}
7: $\quad$ **else**
8: $\quad\quad \delta_{\min} := \delta_k$
9: $\quad k := k + 1$
10: Return $\operatorname{argmin}_S \operatorname{cut}(S) - \delta_{\max}(\operatorname{vol}(S \cap R) - \theta \operatorname{vol}(S \cap \bar{R}))$ or $S_{\max}$ based on min ϕ_R.

Putting the iteration complexity of Fast FlowImprove together with its per iteration computational complexity, we get the following theorem.

THEOREM 7.5 (iteration complexity of the Fast FlowImprove Algorithm 7.2). *Let G be an undirected, connected graph with nonnegative weights. Let R be a subset of V with $\operatorname{vol}(R) \le \operatorname{vol}(\bar{R})$. The sequence δ_k of Algorithm* 7.2 *converges to an approximate solution $|\delta^* - \delta_k|\delta^* \le \varepsilon$ in $\mathcal{O}(\log 1/\varepsilon)$ iterations, where $\delta^* = \phi_R(S^*)$ and S^* is an optimal solution to problem* (7.2). *Moreover, if G has nonnegative integer weights, then the algorithm will return the exact minimizer when $\varepsilon < \frac{1}{\operatorname{vol}(R)^2 \operatorname{vol}(\bar{R})}$.*

Proof. Iteration complexity of FlowImprove is an immediate consequence of Theorem 3.8. The exact solution result is a consequence of the smallest difference between values of relative conductance for integer-weighted graphs. Let S_1 and S_2 be arbitrary sets of vertices in the graph with $\phi_R(S_1) > \phi_R(S_2)$. Let $k_1 = \operatorname{cut}(S_1) \operatorname{vol}(S_2 \cap R) - \operatorname{cut}(S_2) \operatorname{vol}(S_1 \cap R)$ and $k_2 = \operatorname{cut}(S_1) \operatorname{vol}(S_2 \cap \bar{R}) - \operatorname{cut}(S_2) \operatorname{vol}(S_1 \cap \bar{R})$. Both are integers. Then

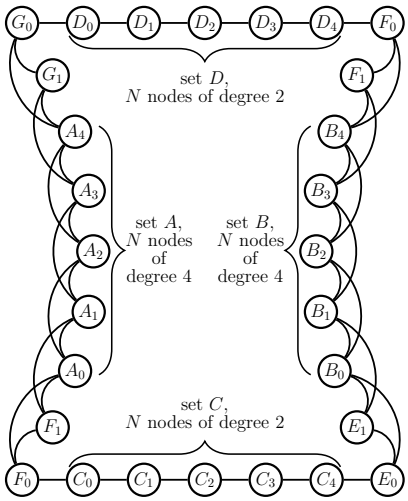
$$\frac{\operatorname{cut}(S_1)}{\operatorname{rvol}(S_1; R, \theta)} - \frac{\operatorname{cut}(S_2)}{\operatorname{rvol}(S_2; R, \theta)} = \frac{k_1 \operatorname{vol}(\bar{R}) - k_2 \operatorname{vol}(R)}{\operatorname{vol}(\bar{R}) \operatorname{rvol}(S_1; R, \theta) \operatorname{rvol}(S_2; R, \theta)} \ge \frac{1}{\operatorname{vol}(R)^2 \operatorname{vol}(\bar{R})}.$$

The last inequality occurs because k_1 and k_2 are integers, and thus the smallest positive value of $k_1 \operatorname{vol}(\bar{R}) - k_2 \operatorname{vol}(R)$ is 1. The rest of the argument for the exact solution is the same as the proof of Theorem 6.5. ∎

The subproblem is the same and so the cost per iteration is the same as discussed in Remark 7.4.

7.4. Nonlocality in FlowImprove. The runtime bounds for FlowImprove assume that we may need to solve a MaxFlow problem with size proportional to the entire graph. We now show that this is essentially tight and that in general the solution of a FlowImprove problem is not strongly local. Indeed, the following example shows that FlowImprove will return one fourth of the graph even when started with a set R that is a singleton.

LEMMA 7.6. *Consider a cycle graph with some extra edges connecting neighbors or neighbors in some parts (illustrated on the right) with* $4N + 8$ *nodes in* 4 *major regions. Each set* A *and* B *has* N *nodes of degree* 4 *corresponding to a contiguous piece of the cycle graph with neighbors and neighbors of neighbors connected. Each set* C *and* D *has* N *degree* 2 *nodes. This introduces two extra nodes, of degree* 3*, between each pair of adjacent degree* 2 *and degree* 4 *regions. Consider using any node of degree* 4 *as the seed node to the FlowImprove algorithm. Then, at optimality, FlowImprove will return a set with* $N+4$ *nodes that is a continuous degree* 4 *region plus the four adjacent degree* 3 *nodes.*



Proof. Without loss of generality, suppose we seed on a node from set A. According to Lemma 7.2, when Dinkelbach's algorithm for FlowImprove proceeds from iteration to iteration, it must return a set with a strictly smaller cut value; otherwise the seed set R was optimal. This means FlowImprove will only return one of the following sets (due to symmetry, there may be equivalent sets that we don't list):

1. The seed node with cut 4.
2. A continuous subset of the A region, G_0, G_1, and a continuous subset of the set D, with cut 3.
3. All of the A region, two adjacent degree 3 nodes (without loss of generality, G_0 and G_1) on one end and one adjacency degree 3 node on the other edge (F_1), with cut 3.
4. All of the A region and all adjacency degree 3 nodes (G_0, G_1, F_0, F_1), with cut 2.
5. All of the A region and all adjacency degree 3 nodes (G_0, G_1, F_0, F_1, and additional nodes from sets C and D), with cut 2.

The goal is to show that case 4 is optimal, i.e., has the smallest objective value. Obviously, case 5 cannot be optimal since it has the same cut value as case 4 but smaller relative volume. Similarly, case 3 has the same cut value as case 2 but smaller relative volume. So case 3 won't be optimal either. So we only need to compare $\phi_R(S_1)$, $\phi_R(S_2)$, and $\phi_R(S_4)$. Observe that in this setting, $\theta = \frac{\mathrm{vol}(R)}{\mathrm{vol}(\bar{R})} = \frac{4}{(2N-1)4+2N\cdot 2+8\cdot 3} = \frac{1}{3N+5}$, so we can compute that

$$\phi_R(S_4) = \frac{2}{4 - \theta(4(N-1) + 3 \cdot 4)} = \frac{3N+5}{4N+6} < 1 = \phi_R(S_1).$$

On the other hand, suppose that in case 2 there are $1 \le k < N$ nodes from A and $m \ge 0$ nodes from D; then we can write

$$\phi_R(S_2) = \frac{3}{4 - \theta(4(k-1) + 3 \cdot 2 + 2m)} \ge \frac{3}{4 - 6\theta} = \frac{9N+15}{12N+14} > \phi_R(S_4).$$

So case 4 is optimal. ∎

7.5. Relationship with PageRank. The FlowImprove subproblem (7.5) is closely related to the PageRank problem if the 1-norm objective is translated into a 2-norm

objective and we relax to real-valued vectors (and make a small perturbation to the resulting systems). This was originally observed, in slightly different ways, in our previous work (Gleich and Mahoney, 2014, 2015). For the same matrix $\tilde{B}$, consider the problem

$$\begin{array}{ll} \underset{\tilde{x}}{\text{minimize}} & \|\tilde{B}\tilde{x}\|^2_{\tilde{C},2} \\ \text{subject to} & x_s = \tilde{x}_1 = 0, x_t = \tilde{x}_{n+2} = 1, \tilde{x}_i \in \{0,1\}\ \forall i = 2,\dots,n+1. \end{array} \tag{7.6}$$

Note that this problem, with the binary constraints, is exactly equivalent to the original problem. However, if we relax the binary constraints to real-valued vectors and substitute in $x_1 = 1$ and $x_{n+2} = 0$, then this is a strongly convex quadratic objective, which can be solved as the following linear system:

$$(B^T C B + \delta \operatorname{diag}(\tilde{d}_R) + \theta\delta \operatorname{diag}(\tilde{d}_{\bar{R}}))x = \delta/2\tilde{d}_R. \tag{7.7}$$

Here, $B^T C B = L = D - A$ is the Laplacian of the original graph. Also, if we have $\theta = 1$ (or simply assume this is true), then $\delta \operatorname{diag}(\tilde{d}_R) + \theta\delta \operatorname{diag}(\tilde{d}_{\bar{R}}) = \delta D$. This yields the linear system

$$(L + \delta D)x = \delta/2 \quad \Leftrightarrow \quad (I - \tfrac{1}{1+\delta} A D^{-1})Dx = \delta/(2+2\delta)\tilde{d}_R.$$

The second system is equivalent to a rescaled PageRank problem for an undirected graph $(I - \alpha A D^{-1})y = \gamma v$, where $y = Dx$. This form, or a scaled version, is widely used in practice (Gleich, 2015).

8. The LocalFlowImprove (and SimpleLocal) Problem and Algorithm. In this section, we will describe the LocalFlowImprove method, due to Orecchia and Zhu (2014), and the related SimpleLocal method, due to Veldt, Gleich, and Mahoney (2016). This cluster improvement method was designed to address the issue that FlowImprove is weakly (and not strongly) local, i.e., that the FlowImprove method has a runtime that depends on the size of the entire input graph and not just on the size of the reference set R. The setup is the same: the LocalFlowImprove method takes as input a graph $G = (V, E)$ and a reference set $R \subset V$, with $\operatorname{vol}(R) \le \operatorname{vol}(G)/2$, and it returns as output an "improved" cluster.

To understand the LocalFlowImprove method, consider the following variant of conductance:

$$\phi_{R,\sigma}(S) = \begin{cases} \dfrac{\operatorname{cut}(S, \bar{S})}{\operatorname{vol}(S \cap R) - \sigma \operatorname{vol}(S \cap \bar{R})} & \text{when the denominator is positive,} \\ \infty & \text{otherwise,} \end{cases} \tag{8.1}$$

where $\sigma \in [\operatorname{vol}(R)/\operatorname{vol}(\bar{R}), \infty)$. This is identical to FlowImprove (7.1), but we change θ into σ and allow it to vary. Given this, the basic LocalFlowImprove problem is

$$\boxed{\begin{array}{ll} \underset{S}{\text{minimize}} & \phi_{R,\sigma}(S) \\ \text{subject to} & S \subset V. \end{array}} \tag{8.2}$$

On the surface, it is straightforward to adapt FlowImprove to LocalFlowImprove. Simply "repeating" the entire previous section with σ instead of θ will result in correct algorithms. For example, the original algorithm proposed for LocalFlowImprove by Orecchia and Zhu (2014) is presented in a fashion equivalent to Algorithm 8.1, which is simply an instance of the bisection-based fractional programming Algorithm 3.2.

The key difference between FlowImprove and LocalFlowImprove is that by setting σ larger than $\mathrm{vol}(R)/\mathrm{vol}(\bar{R})$, we will be able to show that the runtime is independent of the size of the input graph. Recall that we have already shown that the *output set* has a graph size independent bound in Lemma 3.2.

ASIDE 7. *For the theory in this section, we parameterize the LocalFlowImprove objective with σ instead of $\theta+\delta$ as in sections* 3.1 *and* 9. *This choice reduces the number of constants in the statement of theorems. The previous choice of δ is designed to highlight the FlowImprove to MQI spectrum.*

This strongly local aspect of LocalFlowImprove manifests in the subproblem solve step. Put another way, we need to crack open the black-box flow techniques in order to make them run in a way that scales with the size of the output rather than the size of the input. As a simple example of how we'll need to look inside the black box, note that when $\sigma = \infty$, then LocalFlowImprove corresponds to MQI as discussed in section 3.1, which has an extremely simple strongly local algorithm. We want algorithms that will be able to take advantage of this property without needing to be told this will happen. Consequently, in this section, we are going to discuss the subproblem solver extensively.

In particular, we will cover how to adapt a sequence of standard MaxFlow solves to be strongly local, as in the SimpleLocal method of Veldt, Gleich, and Mahoney (2016) (section 8.1), as well as improvements that arise from using blocking flows and adapting Dinic's algorithm (sections 8.2 and 8.3). We will also cover differences with solvers with different types of theoretical tradeoffs that were discussed in the original Orecchia and Zhu (2014) paper (section 8.4).

Note that the SimpleLocal algorithm of Veldt, Gleich, and Mahoney (2016) did not use binary search on δ as in Algorithm 8.1 (and nor do our implementations), but instead it used the original Dinkelbach's algorithm. As we have pointed out a few times, binary search is not as useful as it may seem for these problems, as a few iterations of Dinkelbach's method are often sufficient on real-world data. The point here is that the tradeoff between bisection and the greedy Dinkelbach's method is independent of the *subproblem* solves that are the heart of what differentiates LocalFlowImprove from FlowImprove. Finally, note that Algorithm 8.1 is also a special instance of Algorithm 3.2.

Algorithm 8.1 LocalFlowImprove (Orecchia and Zhu, 2014).

Initialize $k := 1$, $\delta_{\min} := 0$, $\delta_{\max} := 1$, $\sigma \in \left[\frac{\mathrm{vol}(R)}{\mathrm{vol}(\bar{R})}, \infty\right)$, and $\varepsilon \in (0, 1]$
while $\delta_{\max} - \delta_{\min} > \varepsilon\delta_{\min}$ **do**
  $\delta_k := (\delta_{\max} + \delta_{\min})/2$
  Solve $S_{k+1} := \mathrm{argmin}_S \mathrm{cut}(S) - \delta_k \left(\mathrm{vol}(S \cap R) - \sigma\,\mathrm{vol}(S \cap \bar{R})\right)$ via MaxFlow
  if $\mathrm{vol}(S_{k+1} \cap R) > \sigma\,\mathrm{vol}(S_{k+1} \cap \bar{R})$ {Then $\delta_k \geq \delta^*$} **then**
    $\delta_{\max} := \phi_{R,\sigma}(S_{k+1})$ and set $S_{\max} := S_{k+1}$ {Note $\phi_{R,\sigma}(S_{k+1}) \leq \delta_k$}
  else
    $\delta_{\min} := \delta_k$
  $k := k + 1$
Return $\mathrm{argmin}_S \mathrm{cut}(S) - \delta_{\max}(\mathrm{vol}(S \cap R) - \sigma\mathrm{vol}(S \cap \bar{R}))$ or $S_{\max}$ based on min $\phi_{R,\sigma}$.

The iteration complexity of Algorithm 8.1 is now just a standard application of the fractional programming theory.

THEOREM 8.1 (iteration complexity of LocalFlowImprove). *Let G be an undirected, connected graph with nonnegative weights. Let R be a subset of nodes with*

$\mathrm{vol}(R) \leq \mathrm{vol}(\bar{R})$. *In Algorithm* 8.1*, the sequence* δ_k *converges to an approximate optimal value* $|\delta^* - \delta_k|/\delta^* \leq \varepsilon$ *in* $\mathcal{O}(\log 1/\varepsilon)$ *iterations, where* $\delta^* = \phi_{R,\sigma}(S^*)$ *and* S^* *is an optimal solution to problem* (8.2)*. Moreover, if* G *has nonnegative integer weights and* $\sigma = (\eta + \mathrm{vol}(R))/\mathrm{vol}(\bar{R})$ *for an integer value of* η*, then the algorithm will return the exact minimizer when* $\varepsilon < \frac{1}{\mathrm{vol}(R)^2\,\mathrm{vol}(\bar{R})}$.

Proof. The first part is an immediate consequence of Theorem 3.8 with $\delta_{\max} = 1$. The exact solution result is a consequence of the smallest difference between values of relative conductance for integer weights. Let S_1 and S_2 be arbitrary sets of vertices in the graph with $\phi_{R,\sigma}(S_1) > \phi_{R,\sigma}(S_2)$. Let $k_1 = \mathrm{cut}(S_1)\,\mathrm{vol}(S_2 \cap R) - \mathrm{cut}(S_2)\,\mathrm{vol}(S_1 \cap R)$ and $k_2 = \mathrm{cut}(S_1)\,\mathrm{vol}(S_2 \cap \bar{R}) - \mathrm{cut}(S_2)\,\mathrm{vol}(S_1 \cap \bar{R})$, which are both integers. Then

$$\frac{\mathrm{cut}(S_1)}{\mathrm{rvol}(S_1; R, \sigma)} - \frac{\mathrm{cut}(S_2)}{\mathrm{rvol}(S_2; R, \sigma)} = \frac{k_1\,\mathrm{vol}(\bar{R}) - k_2(\eta + \mathrm{vol}(R))}{\mathrm{vol}(\bar{R})\mathrm{rvol}(S_1; R, \sigma)\mathrm{rvol}(S_2; R, \sigma)} \geq \frac{1}{\mathrm{vol}(R)^2\,\mathrm{vol}(\bar{R})}.$$

The inequality follows because the integrality of k_1 and k_2 ensures that the smallest positive value of $k_1\,\mathrm{vol}(\bar{R}) - k_2(\eta + \mathrm{vol}(R))$ is 1. The rest of the argument on the exact solution is the same as in the proof of Theorem 6.5. ∎

Augmented Graph 3 for the subproblem at step 4 of LocalFlowImprove Algorithm 8.1. This is identical to the FlowImprove procedure with σ instead of θ; for LocalFlowImprove we develop algorithms to work with this problem implicitly.

1: Add to the set of nodes V a source node s and a sink node t.
2: Add to the set of edges E an edge from the source node s to every node in the seed set of nodes R with weight the degree of that node times δ.
3: Add to the set of edges E an edge from the sink node t to every node in the set of nodes $\bar{R}$ with weight the degree of that node times $\delta\sigma$, where $\sigma \in [\mathrm{vol}(R)/\mathrm{vol}(\bar{R}), \infty)$.

Moreover, the subproblem construction and augmented graph are identical to FlowImprove, except with σ instead of θ. For the construction of the modified graph to use at the subproblem step, see Augmented Graph 3. The MinCut problem for a specific value of $\delta = \delta_k$ from the algorithm is also equivalent with σ instead of θ,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} + (\mathbf{1} - \delta)x^T\hat{d}_R + \delta\sigma x^T\hat{d}_{\bar{R}} \\ \text{subject to} & x \in \{0,1\}^n, \end{array} \tag{8.3}$$

using the same notation as in (7.3). (Here, we have not implemented the subsequent step of associating terms with sources and sinks, because that follows an identical reasoning to the FlowImprove problem.)

However, in practice, we *never explicitly* build this augmented graph, as that would *immediately* preclude a strongly local algorithm, where the runtime depends on $\mathrm{vol}(R)$ instead of n or m (the number of vertices or edges). Instead, the algorithms seek to iteratively identify a *local graph*, whose size is bounded by a function of $\mathrm{vol}(R)$ and σ that has all of R and just enough of the rest of G to be able to guarantee a solution to (8.3).

As some quick intuition as to *why* the LocalFlowImprove subproblem might have this property, we recall Lemma 3.2, which showed that there is a bound on the output size that is independent of the graph size. We further note the following *sparsity-promoting* intuition in the LocalFlowImprove subproblem.

LEMMA 8.2 (originally from Veldt, Gleich, and Mahoney (2016, Theorem 1)). *The subproblem solve in LocalFlowImprove* (8.3) *corresponds to a degree-weighted* 1*-norm regularized variation on the subproblem solve in FlowImprove* (7.4)*. More specifically, the* 1*-norm regularized problem is*

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Bx\|_{C,1} + \hat{\delta}(\mathbf{1} - x)^T \hat{d}_R + \hat{\delta}\theta x^T \hat{d}_{\bar{R}} + \kappa \|Dx\|_1 \\ \textit{subject to} & x \in \{0,1\}^n \end{array} \tag{8.4}$$

with $\hat{\delta} = \delta + \kappa$ *and* $\kappa = \frac{\delta\sigma - \delta}{1+\theta}$*, where* $\theta = \operatorname{vol}(R)/\operatorname{vol}(\bar{R})$*.*

Proof. The proof follows from expanding (8.4) using $\hat{\delta}$ and $\kappa\|Dx\|_1 = \kappa x^T \hat{d}_R + \kappa x^T \hat{d}_{\bar{R}}$ for indicator vectors, and then ignoring constant terms. ∎

ASIDE 8. *We also note that this idea of adding a* 1*-norm penalty is a common design pattern to create strongly local algorithms.*

Given the rich literature on solving 1-norm regularized problems in time much smaller than the ambient problem space or with provably fewer samples (Tibshirani, 1996; Efron et al., 2004; Candès, Romberg, and Tao, 2006; Donoho and Tsaig, 2008), these results are perhaps somewhat less surprising.

In the remainder of this section, we will explain two solution techniques for the subproblem solve that will guarantee the runtime in the following theorem for finding the set that minimizes the LocalFlowImprove objective.

THEOREM 8.3 (runtime of LocalFlowImprove, based on Veldt, Gleich, and Mahoney (2016)). *Let* G *be a connected, undirected graph with nonnegative integer weights. A LocalFlowImprove problem can be solved via Dinkelbach's Algorithm* 3.1 *or Algorithm* 8.1*. The algorithms terminate in worst-case time*

$$\mathcal{O}(\operatorname{cut}(R) \cdot \texttt{subproblem}) \text{ for Dinkelbach and } \mathcal{O}(\log \tfrac{1}{\varepsilon} \cdot \texttt{subproblem}) \text{ for bisection.}$$

Let $\gamma = 1 + \frac{1}{\sigma}$*. For solving the subproblem, we have the following possible runtimes:*

Algorithm 8.2	$\gamma \operatorname{vol}(R)$ *calls to MaxFlow on* $\gamma \operatorname{vol}(R)$ *edges*	*(MaxFlow-based),*
Algorithm 8.3	$\mathcal{O}\left(\gamma^2 \operatorname{vol}(R)^2 \log[\gamma \operatorname{vol}(R)]\right)$	*(BlockingFlow-based).*

Proof. This result can be obtained by combining the iteration complexity of Dinkelbach's Theorem 3.6 or LocalFlowImprove from Theorem 8.1 with either the runtime of the MaxFlow-based SimpleLocal subsolver Algorithm 8.2 or the runtime of the blocking flow algorithm in Theorem 8.5. ∎

See section 8.4 for details on faster algorithms from Orecchia and Zhu (2014).

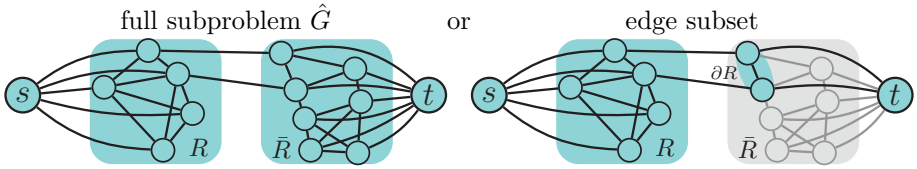
8.1. Strongly Local Constructions of the Augmented Graph. Before we present algorithms for the LocalFlowImprove subproblem, we discuss a crucial result from Orecchia and Zhu (2014) that reduces Augmented Graph 3 for the MaxFlow problem to a reduced modified graph that includes only nodes relevant to the optimal solution. The crux of this section is an appreciation of the following statement:

> *An unsaturated edge in a flow is an edge where the flow value is strictly less than the capacity. If, in a solution of MaxFlow on the augmented graph, there is an unsaturated edge from a node in* $\bar{R}$ *to* t*, then that node is not in the solution MinCut set.*

This result is a fairly simple structural statement about how we might *verify* a solution to such a MaxFlow problem. We will illustrate it first using a simple example where

the optimal solution set is contained within R, akin to MQI but without that explicit constraint, and then we move to the more general case which will involve introducing the idea of a *bottleneck set* B. Throughout these discussions, we will use $\hat{G}$ to denote the full s,t augmented graph construction for a LocalFlowImprove subproblem with R, δ, σ fixed.

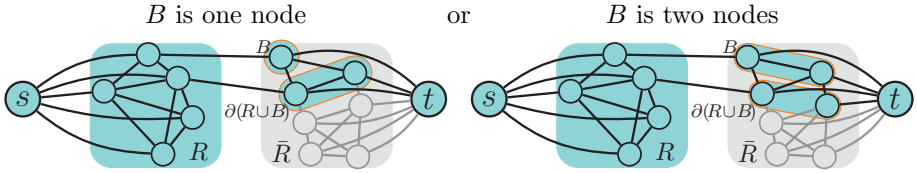
Consider what happens in solving a MaxFlow on $\hat{G}$, where $\sigma > \text{vol}(R)$. In this scenario, LocalFlowImprove will always return $S \subseteq R$ and this will be true on the subproblem solve as well. (See the discussion in section 3.1.) We will show how we can locally certify a solution on $\hat{G}$—without even creating the entire augmented graph. We first note the structure of the $\hat{G}$ partitioned into the following sets: R, ∂R, and everything else, i.e., $\bar{R} - \partial R$. This results in the following view of the subproblem:



Suppose that we *delete* all the gray edges and solve the resulting MaxFlow problem (or just solve the problem for the teal-colored subset). This will result in a MaxFlow problem on a subset of the edges of the augmented graph—and one that has size bounded by $\text{vol}(R)$. In any solution of the resulting MaxFlow problem, we have that all of the edges from ∂R to t will be unsaturated, meaning that the flow along those edges will be strictly smaller than the capacity. This is straightforward to see because the total flow out of the source is $\delta \,\text{vol}(R)$ and *each edge* from ∂R to t has weight $d_i \delta \sigma > d_i \delta \,\text{vol}(R)$. Consequently, the nodes in ∂R will always be on the sink side of the MinCut solution.

This ability of unsaturated edges to provide a local guarantee that we have found a solution arises from two aspects. First, we have a strict edge subset of the true augmented graph, so any flow value we compute will be a lower bound on the max-flow objective function on the entire graph. Second, we have not removed any edges from the source. Consequently, we can locally certify this solution because none of the edges leading to t are saturated, so the bottleneck must have been outside of the boundary of R. Put another way, since the edges from ∂R to t are unsaturated, there is no way the omitted gray nodes and edges could have helped get more flow from the source to the sink.

Now, suppose that σ is smaller such that at least one node in the boundary of R has a saturated edge to t. Then we lose the proof of optimality because it's possible those missing gray nodes and edges could have been used to increase the flow. Suppose, however, we add those *bottleneck* nodes in ∂R to a set B and solve for the MaxFlow on the subgraph of $\hat{G}$ with all edges among s, t, R, B, $\partial(R \cup B)$ as follows:



As long as the bottleneck is not in the boundary $\partial(R \cup B)$, then we have an optimal solution. The best way to think about this is to look at the missing edges in the

Algorithm 8.2 MaxFlowSimpleLocal (Veldt, Gleich, and Mahoney, 2016).

Set $B := \emptyset$
while the following procedure has not yet returned **do**
  Solve the MaxFlow problem on $G_{R,\sigma,\delta}(B)$ consistent with previous iterations.
  Let J denote the vertices in $\partial(R \cup B)$ whose saturated edges to the sink t.
  if $|J| = 0$ **then** return the MinCut set S as the solution
  else $B \leftarrow B \cup J$ and repeat

picture. If all the edges to t from the boundary $\partial(R \cup B)$ are unsaturated, then we must have a solution, as the other edges could not have increased the flow.

Of course, there may still be saturated edges in the boundary, but this suggests a simple algorithm. To state it, let $G_{R,\sigma,\delta}(B)$ be the s,t MaxFlow problem with

- all vertices $\{s, t\} \cup R \cup B \cup \partial(R \cup B)$,
- all edges from the source s to nodes in set R,
- all edges with nodes in the set $B \cup \partial(R \cup B)$ to the sink node t,
- all edges from nodes in $R \cup B$ to nodes in V.

We iteratively grow B by nodes whose edges to t are saturated in a MaxFlow solve on $G_{R,\sigma,\delta}(B)$, starting with B empty. This procedure is described in Algorithm 8.2, which uses the idea of solving MaxFlow problems consistently with previous iterations, to which we will return shortly. What this means is that among multiple optimal solutions, we choose the one that would saturate edges to B in the same way as previous solutions. There is a simple way to enforce this by using the residual graph, and this really just means that once a node goes into B, it stays in B.

Locally finding the set B is, in a nutshell, the idea behind strongly local algorithms for LocalFlowImprove. These strongly local algorithms construct the set B for each subproblem solve by doing exactly what we describe here, along with using a few small ideas to make them go fast. The algorithms to accomplish this will always produce a set B whose size is bounded in terms of σ and $\mathrm{vol}(R)$, as guaranteed by the following result.

LEMMA 8.4 (Lemma 4.3 (Orecchia and Zhu, 2014)). *We have* $\mathrm{vol}(B) \leq \frac{1}{\sigma}\mathrm{vol}(R)$ *for every iteration for the iteratively growing procedure in Algorithm* 8.2.

Proof. The proof follows because each time a node v is added to B, we know there is a flow that saturates the edge with weight $\sigma\delta d_v$. Since the total flow from s is $\delta\,\mathrm{vol}(R)$, this implies that if $\mathrm{vol}(B) \geq \mathrm{vol}(R)/\sigma$, then we have expanded enough edges to t to guarantee that the flow can be fully realized with no bottlenecks. ∎

8.2. Blocking Flow. In each iteration of Algorithm 8.2, we need to identify the set J. We motivated this set with a *maximum flow* on the graph $G_{R,\sigma,\delta}(B)$. It turns out that we do not actually need to solve a MaxFlow problem. Instead, the concept of a *blocking flow* suffices. The difference is subtle but important. A blocking flow is a flow such that every path from the source to the sink contains at least one saturated edge. For a depiction of a blocking flow, see Figure 10. See also the helpful descriptions in Williamson (2019, Chapter 4). By this definition, a maximum flow is always a blocking flow because the source and sink are disconnected in the residual graph.

The relevance of blocking flows is that after finding a blocking flow and looking at the residual graph, then the distance between the source and sink increases by one. This is essentially what we do with the set B and J: If B is not yet optimal and

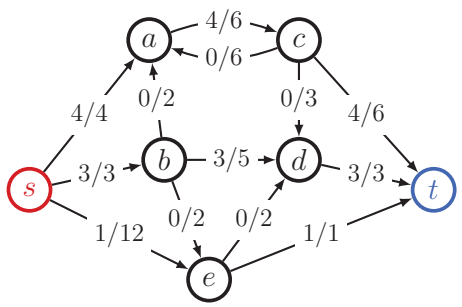


Fig. 10 *We demonstrate a flow that starts from the source node s and ends at the sink node t. This flow includes the paths (s,b,d,t), (s,a,c,t), and (s,e,t). Note that this flow is a blocking flow since every path from s to t includes at least one saturated edge. It is not a maximum flow because there is a path from t to s with reversed edges.*

we find nodes J in Algorithm 8.2, then the *length* of the next path from the source to the sink that is found to become optimal must increase by one. Blocking flow algorithms use this property, along with other small properties of the residual graph, to accelerate flow computations. We defer additional details to Williamson (2019) in the interest of space.

The best algorithm for computing blocking flows was suggested in Sleator and Tarjan (1983). There, the authors proposed a link-cut tree data structure that is used to develop a strongly polynomial time algorithm for weighted graphs that computes blocking flows in $\mathcal{O}(m \log n)$ time, where m is the number of edges in the given graph. Blocking flows are a major tool and subroutine inside other solvers for MaxFlow problems. For example, Dinic's algorithm (Dinitz, 1970) simply runs successive blocking flow computations on the residual graph to compute a maximum flow (see Williamson (2019, Algorithm 4.1) as well). This iteratively finds the maximum flow *up to a distance d*. Here, blocking flows serve the purpose of giving us a lower bound on the maximum flow that could saturate some edges of the graph.

8.3. The SimpleLocal Subsolver. For the SimpleLocal subsolver, we will use the concept of local bottleneck graph $G_R(B)$ that was introduced in section 8.1. (We omit σ, δ for simplicity.) The only other idea involved is that we can *iteratively* update the entire flow itself using the residual graph. So, rather than solving MaxFlow at each step, we compute a blocking flow to find new elements J and update the residual graph. This ensures that the flow between iterations is consistent in the fashion we mentioned in Algorithm 8.2. The algorithm is presented in Algorithm 8.3. SimpleLocal is exactly Dinic's algorithm but specialized for our LocalFlowImprove problem.

THEOREM 8.5 (iteration complexity and runtime for SimpleLocal). *Let G be an undirected, connected graph with nonnegative weights. SimpleLocal requires $(1 + \frac{1}{\sigma})\operatorname{vol}(R)$ iterations to converge to the optimal solution of the MaxFlow subproblem and $\mathcal{O}(\operatorname{vol}(R)^2(1+\frac{1}{\sigma})^2 \log[(1+\frac{1}{\sigma})\operatorname{vol}(R)])$ runtime.*

Proof. Dinic's algorithm converges in at most $(1 + \frac{1}{\sigma})\operatorname{vol}(R)$ iterations (Proposition A.1 and Lemma 4.3 of Orecchia and Zhu (2014)). Each iteration requires a blocking flow operation that costs $\mathcal{O}((1 + \frac{1}{\sigma})\operatorname{vol}(R)\log[(1 + \frac{1}{\sigma})\operatorname{vol}(R)])$ time (Lemma 4.2 of Orecchia and Zhu (2014)). Hence, SimpleLocal requires $\mathcal{O}(\operatorname{vol}(R)^2(1+\frac{1}{\sigma})^2 \log[(1+\frac{1}{\sigma})\operatorname{vol}(R)])$ time. ∎

Algorithm 8.3 SimpleLocal (Veldt, Gleich, and Mahoney, 2016).

1: Initialize the flow variables f to zero and $B := \emptyset$
2: **while** True **do**
3: Compute a blocking flow $\hat{f}$ for the residual graph of $G(B)$ with the flow f; if flow is zero, then stop.
4: $f \leftarrow f + \hat{f}$
5: Let J denote the vertices in $\partial(R \cup B)$ whose edges to the sink node t get saturated using the new flow variables f.
6: $B \leftarrow B \cup J$
7: The current flow variables f are optimal for the MaxFlow in $G(B)$; return a MinCut set S on the source side s.

In Veldt, Gleich, and Mahoney (2016), SimpleLocal is described using MaxFlow to compute the blocking flows in step 3 of Algorithm 8.3. We also used Dinkelbach's algorithm instead of binary search. Otherwise, however, the two algorithms are identical. In practice, both of those modifications result in faster computations, although they are slower in theory.

8.4. More Sophisticated Subproblem Solvers. More advanced solvers for the LocalFlowImprove algorithm are possible in theory. For instance, Orecchia and Zhu (2014) also present a solver based on the Goldberg–Rao Push-Relabel method (Goldberg and Rao, 1998) that will yield a strongly local algorithm. Finally, note that the goal in using these algorithms is often to minimize the *conductance* of a set S instead of the relative conductance $\phi_{R,\sigma}(S)$, in which case relative conductance is just a computationally useful proxy. The analysis of Orecchia and Zhu (2014) shows that running Algorithm 8.3 for a bounded number of iterations either will return a set S that minimizes the relative conductance exactly or will find an easy-to-identify bottleneck set S' that has conductance $\phi(S') \leq 2\delta$. Using this second property, the authors are able to relate the runtime of the algorithm to the conductance of the set returned for a slightly different type of guarantee than exactly solving the LocalFlowImprove subproblem (Orecchia and Zhu, 2014, Theorem 1a).

Part III. Empirical Performance and Conclusion.

9. Empirical Evaluation. In this section, we provide a detailed empirical evaluation of the cluster improvement algorithms we have been discussing. The focus of this evaluation is on illustrating how the methods behave and how they might be incorporated into a wide range of use cases. The specific results we show include:

1. **Reducing conductance.** (Section 9.1.) Flow-based cluster improvement algorithms are effective at finding sets of smaller conductance near the reference set, as the theory promises. This is illustrated with examples from a road network (see Figure 12 and Table 3, where the algorithm finds geographic features to make the conductance small) as well as on a data-defined graph from astronomy; see Figures 13 and 14. We also illustrate empirically Theorem 3.1, which states that FlowImprove and LocalFlowImprove always return smaller conductance sets than MQI. In our experiments, these improvement algorithms commonly return sets of nodes in which the conductance is cut in half, occasionally reducing it by up to one order of magnitude or more.
2. **Growing and shrinking.** (Section 9.2.) Flow-based improvement algorithms are useful for the target set recovery task (basically, the task of finding a desired set

of vertices in a graph, when given a nearby reference set of nodes), even when the conductance of the input is not especially small. In particular, we show how these methods can grow and shrink input sets to identify these hidden target sets when seeded nearby, by improving precision (the fraction of correct results) or recall (the fraction of all possible results). In this case, we use a weighted graph constructed from images, where the goal is to identify an object inside the image; see Figure 15. We also use a social network, where the goal is to identify students with a specific class year or major within the Johns Hopkins school community; see Figure 16.

3. **Semisupervised learning.** (Section 9.3.) Going beyond simple unsupervised clustering methods, semisupervised learning is the task of predicting the labels of nodes in a graph, when the nearby nodes share the same label and when given a set of true labels. Flow-based improvement algorithms accurately refine large collections of labeled data in semisupervised learning experiments. Our experiments show that flow algorithms are effective for this task, to a greater extent when one is given large collections of true labels, and somewhat less so when one is given only a small number of true labels; see Figure 17.
4. **Scalable implementations.** (Section 9.4.) Our software implementations of these algorithms can be used to find thousands of clusters in a given graph in parallel. These computations scale to large graphs; see Table 4. Our implementations use Dinkelbach's method and Dinic's algorithm for exact solutions of the MaxFlow problems.
5. **Locally biased flow-based coordinates.** (Section 9.5.) We can use our flow improvement algorithms to define locally biased coordinates or embeddings in a manner analogous to how global spectral methods are often used to define global coordinates or embeddings for data; see Figures 19 and 20. This involves a novel flow-based coordinate system that will highlight subtle hidden structure in data that is distinctly different from what is found by spectral methods, as illustrated on road networks and in the spectra of galaxies.

To simplify and shorten the captions, throughout the remainder of this section, we will use the abbreviations MQI, FI (FlowImprove), and LFI (LocalFlowImprove). Because LFI depends on a parameter δ, we will simply write LFI-δ, e.g., LFI-1.0. The formal interpretation of this parameter is LocalFlowImprove$(R, \sigma = \mathrm{vol}(R)/\mathrm{vol}(\bar{R}) + \delta)$, where δ is a nonnegative real number. Recall that LFI-0.0 is equivalent to FI and LFI-∞ is equivalent to MQI.

ASIDE 9. ***Large set results.*** *The sets found by* FI *and* LFI *may not have* $\mathrm{vol}(S) \leq \mathrm{vol}(\bar{S})$. *For instance, the* FI *result in Figure* 12(d) *has* $\mathrm{vol}(S) > \mathrm{vol}(\bar{S})$. *In our computer codes, we always give* $\mathrm{vol}(S) \leq \mathrm{vol}(\bar{S})$ *and flip* S *and* $\bar{S}$ *to force this property. Figure* 12(d) *reverses this flip to show the relationship with* R.

9.1. Flow-Based Cluster Improvement Algorithms Reduce Conductance. The first result we wish to illustrate is that the algorithms MQI, FI, and LFI reduce the conductance of the input reference set, as dictated by our theory. For this purpose, we are going to study the US highway network as a graph (see Figure 11). Edges in this network represent nationally funded highways, and nodes represent intersections. Ferry routes are included, and there exist other major roads that are not in this data. This network has substantial local and small-scale structure that makes it a useful example. It has a natural large-scale geometry that makes it easy to understand visually, and it has large (in terms of number of nodes) good (in terms of conductance) partitions.

We create a variety of reference sets for our flow improvement methods to refine.

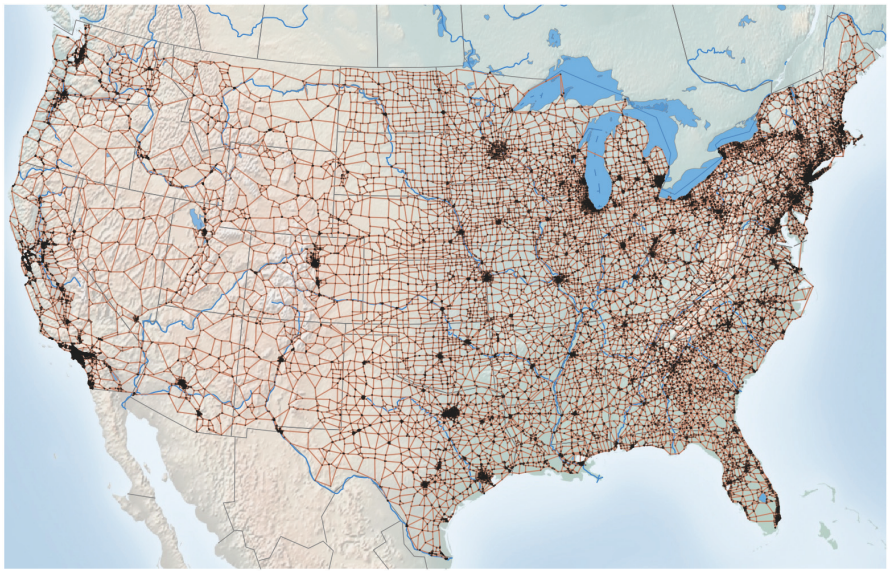
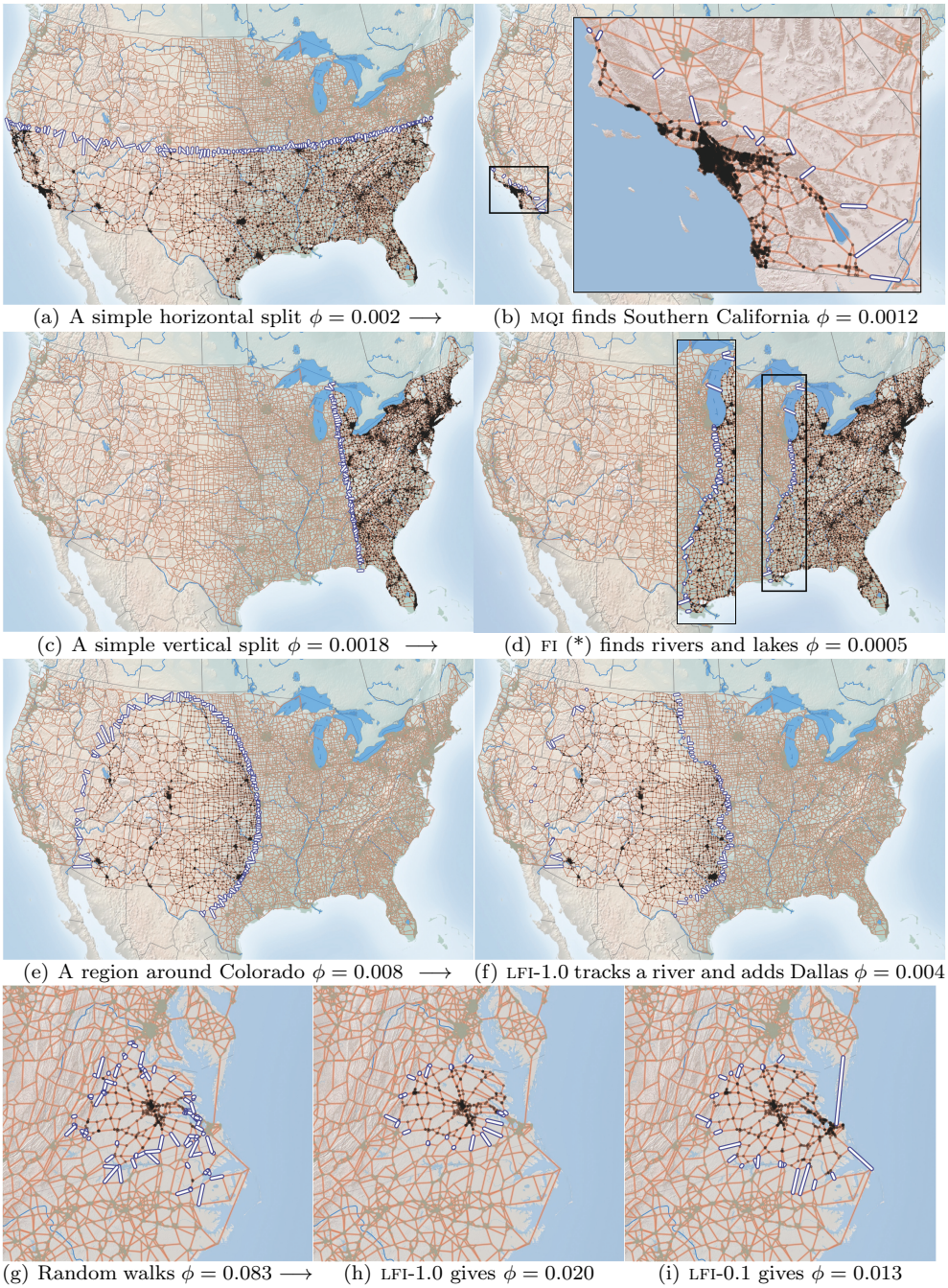


Fig. 11 *The US National Highway Network as a simplified graph has* 51,144 *nodes and* 86,397 *undirected edges. Edges represent roads and are shown as orange lines, and nodes are places where roads meet and are shown as black dots. This display highlights major topographical features as well as major rivers. Mountain ranges, rivers, and lakes create interesting fine-scale features for our flow algorithms to find. There are also dense local regions around cities akin to small well-connected pieces of social networks.*

In Figure 12, nodes in black show a set and purple edges with white interior show the cut.

- We start with two partitions of the network, one horizontal and one vertical (Figures 12(a) and 12(c)). These are simple-to-create sets based on using latitude and longitude, and they roughly bisect the country into two pieces. They are also inherently good conductance sets due to the structure of roads on the surface of the Earth: they are tied to the two-dimensional geometric structure, and thus they have good isoperimetric or surface-to-volume properties.
- Next, we consider a large region in the western US centered on Colorado (Figure 12(e)). Again, this set is shown in black, and the purple edges (with white interior) highlight the cut. The rest of the graph is shown in orange.
- We further consider using the vertices visited in 200 random walks of length 60 around the capital of Virginia (Figure 12(g)). This example will show our ability to refine a set which, due to the noise in the random walks, is of lower quality.
- Finally, we consider the result of the METIS program for bisection, which represents our ability to refine a set that is already high quality. This is not shown because it looks visually indistinguishable from Figure 12(d), although the cut and volume are slightly different, as discussed below and in Table 3.

The conductance improvement results from a number of our algorithms are shown in Table 3 and Figure 12. The table shows additional results that are not present in the figure. We make several observations. First, as given by MQI, the optimal subset



(a) A simple horizontal split $\phi = 0.002 \longrightarrow$ (b) MQI finds Southern California $\phi = 0.0012$

(c) A simple vertical split $\phi = 0.0018 \longrightarrow$ (d) FI (*) finds rivers and lakes $\phi = 0.0005$

(e) A region around Colorado $\phi = 0.008 \longrightarrow$ (f) LFI-1.0 tracks a river and adds Dallas $\phi = 0.004$

(g) Random walks $\phi = 0.083 \longrightarrow$ (h) LFI-1.0 gives $\phi = 0.020$ (i) LFI-0.1 gives $\phi = 0.013$

Fig. 12 *Our flow-based cluster improvement algorithms reduce the conductance of simple input sets by finding natural features including mountains, rivers, and cities. The purple edges highlight the boundary of the set shown in black nodes, and ϕ is the conductance of the depicted set. Panel* (a) *shows an input that cuts the map horizontally and* (b) *is the corresponding output of* MQI. *Panel* (c) *shows an input that cuts the map vertically and* (d) *shows the output of* FI. *Panel* (e) *shows an input which corresponds to a large region in the western US centered on Colorado and* (f) *shows the output of* LFI. *Finally, panel* (g) *shows an input around the capital of Virginia, which has been created using random walks, and* (h) *and* (i) *are the corresponding output of* LFI-1.0 *and* LFI-0.1, *respectively.*

*See the large set results aside (Aside 9).

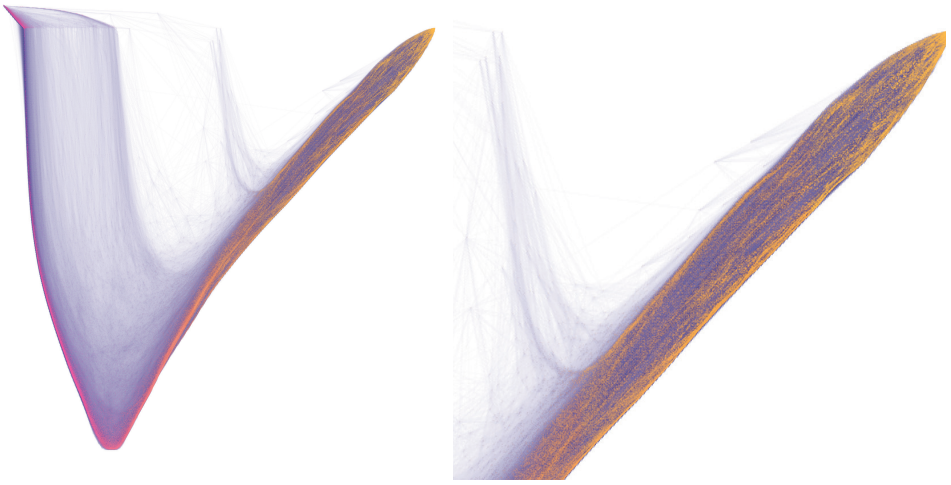
Table 3 *The results of applying our algorithms to input sets of various quality for the graph of Figure 12. A few of the sets and cuts are illustrated in Figure 12. All of the methods reduce the conductance score considerably, with improvement ratios from 131% to 621%. The smallest improvements happen when the input is high quality, such as the output from* METIS.

Input					Result					
	cut	vol.	size	cond.	Alg.	cut	vol.	size	cond.	ratio
Horiz.	233	85335	25054	0.0027	MQI	12	9852	2763	0.0012	225%
					FI	29	35189	10471	0.0008	330%
Vert.	131	72780	21552	0.0018	MQI	29	35195	10473	0.0008	220%
					FI	42	84582	25030	0.0005	365%
Colorado region	195	23377	6982	0.0083	MQI	9	1799	506	0.0050	167%
					LFI-1.0	97	23617	7037	0.0041	203%
					LFI-0.1	101	26613	7941	0.0038	220%
					FI	42	84204	24916	0.0005	1672%
Virginia random walks	112	1344	393	0.0833	LFI-1.5	23	1067	312	0.022	386%
					LFI-1.0	24	1212	357	0.0198	420%
					LFI-0.1	26	1938	572	0.0134	621%
METIS	56	85926	25422	0.0007	MQI	42	84594	25034	0.0005	131%
					LFI-0.1	42	84594	25034	0.0005	131%
					FI	42	84594	25034	0.0005	131%

of the horizontal split of Figure 12(a) identifies a region in the lower US, specifically, the Southern California region around Los Angeles, San Diego, and Santa Barbara (Figure 12(b)). The Southern California area is separated by mountains and deserts that are spanned by just 12 national highways that connect to the rest of the country. Second, the result of FI on the vertical split of Figure 12(c) of the US traces the Mississippi, Ohio, and Wabash rivers up to Lake Michigan (Figure 12), splitting just 42 highways and ferry routes. Note that although we start with the reference on the *East Coast*, the set returned by the algorithm is entirely disjoint. This is because optimizing the FI objective expanded the set to be larger than half the volume, which caused the returned set to flip to the other coast. Third, the region around Colorado in Figure 12(e) is refined by LFI-1.0 to include Dallas (which was split in the initial set) and follows the Missouri river up into Montana. Finally, a set of random walks around the Virginia capital visit much of the interior region of the state, albeit in a noisy fashion. Using LFI-1.0 (Figure 12(h)) refines the edges of this region to reduce conductance. Reducing δ to 0.1 and using LFI-0.1 (Figure 12(i)) results in a bigger set that includes the nearby city (and dense region) of Norfolk. Note that, for the high quality METIS partition, all of our algorithms return exactly the same result. (Again, these are not shown because the results are indistinguishable.) We also note that this set is the overall smallest conductance result in the entire table because the volume is slightly larger than vertical split experiments.

Overall, these results show the ability of our flow improvement algorithms to improve conductance by up to a factor of 16 in the best-case scenario and by a ratio of 1.31 on the high quality METIS partition. The most useful summary from these figures is as follows:

- Reducing the value of δ in LFI corresponds to finding smaller conductance sets compared to MQI. We also observe that reducing δ in LFI results in larger clusters in terms of number of nodes and volume.



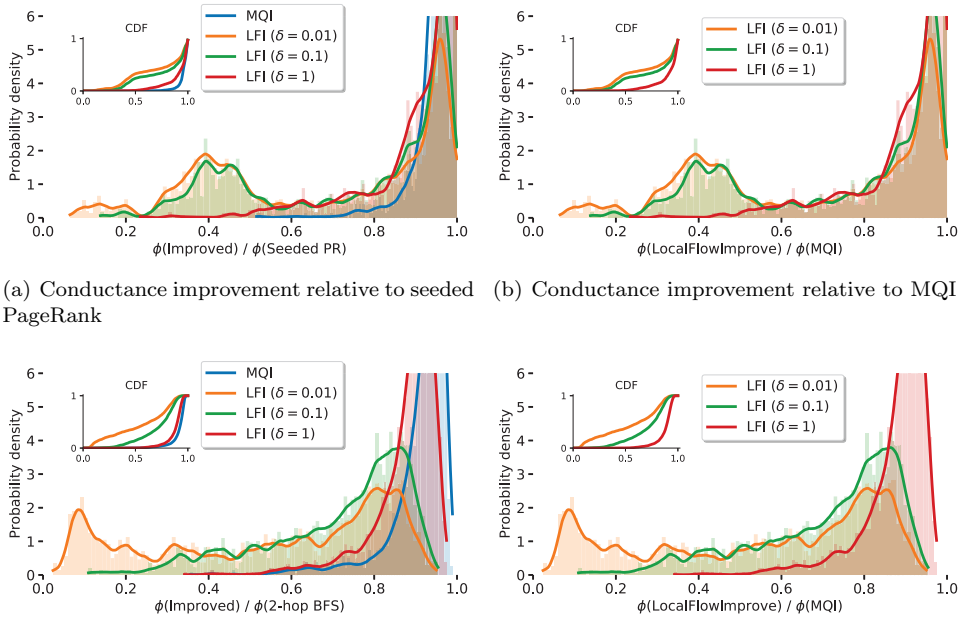
(a) The full graph (b) Zoom into top-right

Fig. 13 *The Main Galaxy Sample (MGS) dataset has* 517,182 *nodes and* 32,229,812 *edges. This display shows an eigenvector embedding of the graph along with edges shown in light blue. The edges dominate the visualization in parts and nodes are only shown where there is sufficient density. The node color is determined by the horizontal coordinate (pink to orange). The right part of the visualization (dark orange to light orange coordinates in* (b) *hints at structure hidden within the upper band, which we will study in section* 9.5.

- As predicted by Theorem 3.1, the results for LFI and FI are always better in terms of conductance than MQI in terms of conductance.

While visually useful in understanding our algorithms, obtaining such results on a road network is less useful and less interesting than obtaining similar results on graphs representing data with fewer or different structural constraints. Thus, we now illustrate these points in another, larger dataset with a study of around 2500 improvement calls. This second dataset is a $k = 32$ nearest neighbor graph constructed on the Main Galaxy Sample (MGS) in SDSS Data Release 7. We briefly review the details of this standard type of graph construction and provide further details in Appendix A. This data begins with the emission spectra of 517,182 galaxies in 3841 bands. We create a node for each galaxy and connect vertices if either is within the 16 closest vertices to the other based on a Euclidean distance-like measure (see Appendix A). The graph is then weighted proportional to this distance. The result is a weighted undirected graph with 517,182 nodes and 15,856,315 edges (and 517,182 self-loops) representing nearest neighbor relationships among galaxy spectra. Figure 13 provides a visualization of a global Laplacian eigenvector embedding of this graph. For more details on this dataset, we refer readers to Lawlor, Budavári, and Mahoney (2016a,b).

In this case, we compute reference sets using seeded PageRank using a random node, followed by a sweepcut procedure by Andersen, Chung, and Lang (2006) to locally optimize the conductance of the result. Consequently, the reference sets we start with are already fairly high quality. Then we run MQI, LFI-1, LFI-0.1, and LFI-0.01 on the results. We repeat this experiment 2526 times. The output to input conductance ratio is shown in Figure 14 with reference to the original reference conductance from



(a) Conductance improvement relative to seeded PageRank

(b) Conductance improvement relative to MQI

(c) Conductance improvement relative to 2-hop BFS

(d) Conductance improvement relative to MQI

Fig. 14 *A summary of* 3102 *(top row) and* 2585 *(bottom row) experiments in the MGS dataset that show* (i) *that reducing* δ *in* LFI *produces sets of smaller conductance, when the input set is from another conductance minimizing procedure (seeded PageRank, top row) or a* 2*-hop breadth first search (BFS) set (bottom row), and also* (ii) *that* LFI *and* FI *always find smaller conductance sets than* MQI*. The inset figures shows the cumulative density function (CDF) of the probability density.*

seeded PageRank (Figure 14(a)) and also with reference to the MQI conductance (Figure 14(b)). Like the previous experiments with the road network, reducing δ in this less easily visualizable dataset results in improved conductance. Also, like in the previous experiments, LFI always reduces the conductance more than MQI does.

The point of these initial experiments is to demonstrate that these algorithms achieve their primary goal of finding small conductance sets in a variety of scenarios. They can do so both in a graph with an obviously geometric structure as well as in a graph without an obvious geometric structure that was constructed from noisy observational data. In addition, they can do so starting from higher or lower quality inputs. In the next section, we evaluate our algorithms on specific tasks where finding small conductance sets is not the end goal.

9.2. Finding Nearby Targets by Growing and Shrinking. Another use for cluster improvement methods is to recover a hidden target set of vertices from a nearby reference set, e.g., a conjectured subregion of a graph or a coherent section of an image. The goal here is accuracy in returning the vertices of this set, and we can measure this in terms of precision and recall. Let T be a target set we seek to find, and let S be the set returned by the algorithm. Then the precision score is $|T \cap S|/|S|$, which is the fraction of results that were correct, and the recall score is $|T \cap S|/|T|$,

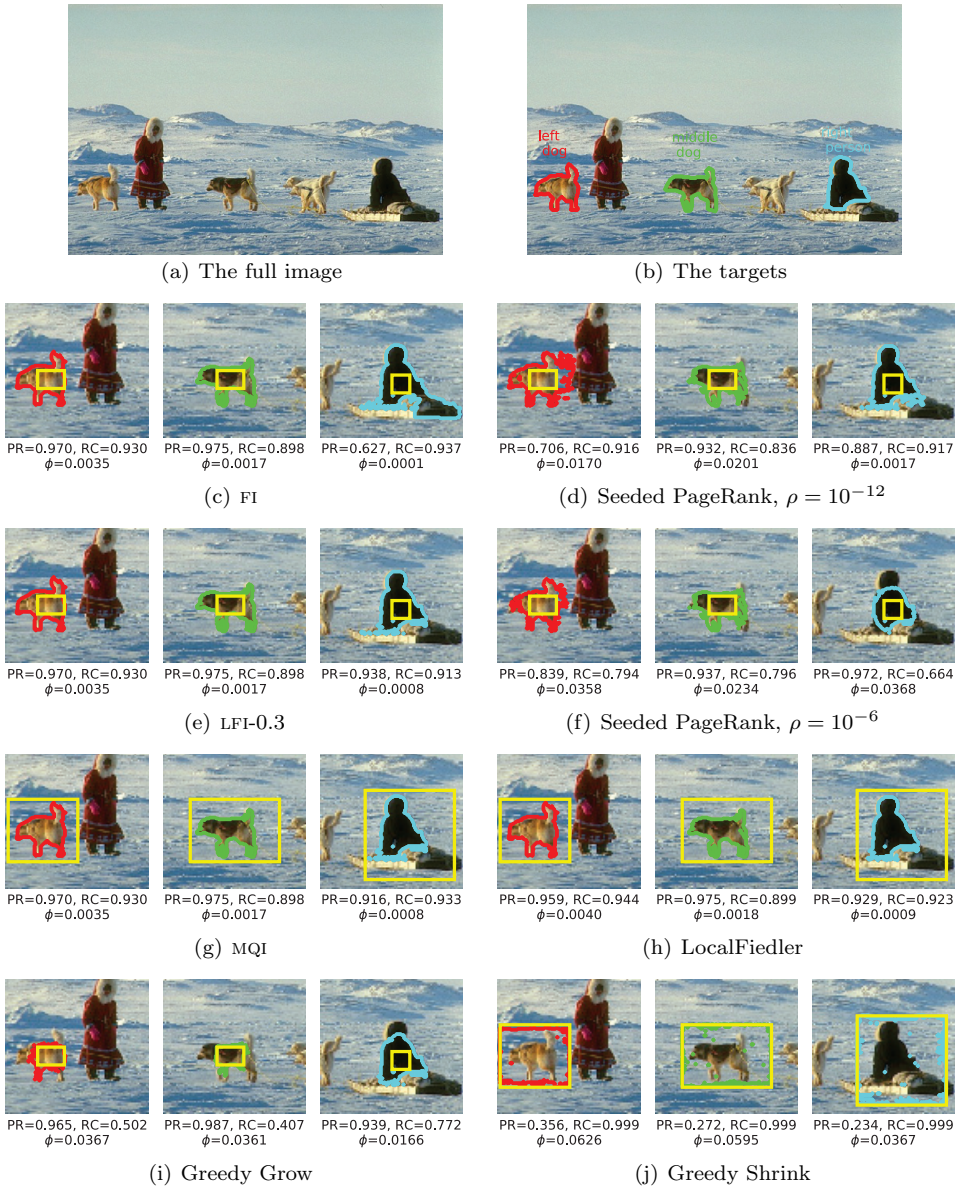
which is the fraction of all results that were obtained. The ideal scenario is that both precision and recall are near 1.

We begin by looking at the simple scenario when the initial reference R is entirely contained within T, and also a scenario when R is a strict superset of T. This setting allows us to see how the flow-based algorithms grow or shrink sets to find these targets T, and it gives us a useful comparison against simple greedy improvement algorithms as well as against spectral graph-based approaches. For simplicity of illustration, we examine these algorithms on weighted graphs constructed from images. The construction of a graph based on an image is explained in Appendix B.

The results of the experiment are shown in Figure 15. We consider three distinct targets within a large image, as shown in Figures 15(a) and 15(b): the left dog, middle dog, and right person. In our first case, the reference is entirely contained within the target. In this case, we can use either FI or LFI to attempt to enlarge to the target. (Note that we cannot use MQI, as the target set is larger than the seed set.) For comparison, we use a seeded PageRank algorithm as well. The choice of PageRank is because PageRank largely corresponds to replacing $\|Bx\|_1$ in the flow-based objective with the minorant function $\|Bx\|_2^2$ as discussed in section 7.5. We use two seeded PageRank scenarios that correspond to both FI and LFI; see Figures 15(c) to 15(f). These show that spectral methods that grow tend to either find a region that is too big or fail to grow large enough to capture the entire region. This is quantified by a substantial drop in precision compared with the flow method. Second, we consider the case when the target is contained within the reference set. This corresponds to the MQI setting as well as a variation of spectral clustering called Local Fiedler (Chung, 2007b) (because it uses the eigenvector with minimal eigenvalue in a submatrix of the Laplacian). The results are given in Figures 15(g) and 15(h), and they show a small precision advantage for the flow-based methods (see the text below each image). Finally, for reference, in Figures 15(i) and 15(j) we also include the results of a purely greedy strategy that grows or shrinks the reference set R to improve the conductance. This is able to find reasonably good results for only one of the test cases and shows that these sets are not overly *simple* to identify, e.g., since they cannot be detected by algorithms that trivially grow or expand the seed set.

Next, we repeat these target set experiments using the Johns Hopkins network, a less visualizable network, for which we see similar results. The data are a subset of the Facebook100 dataset from Red et al. (2011) and Traud, Mucha, and Porter (2012). The graph is unweighted. It represents "friendship" ties and it has 5157 nodes and 186,572 edges. This dataset comes with 6 features: major, second major, high school, gender, dorm, and year. We construct two targets by using the following features: students with a class year of 2009 and student with major id 217. The visualization shows that major id 217 looks like it will be a fairly good cluster as the graph visualization has moved the bulk away. However, the conductance of this set is 0.26. Indeed, neither of these sets has particularly small conductance, which makes the target identification problem much harder than in the images. Both sets are illustrated in Figure 16(a).

Here, we use a simple breadth first search (BFS) method to generate the input to MQI and LFI to mirror the previous experiment with images. Given a single and arbitrary node in the target cluster, we generate a seed set R by including its neighborhood within 2 hops. Like the previous examples, we then use MQI to refine precision and LFI to boost recall. We repeat this generating and refining procedure 25 times for distributional statistics. The inputs as well as results from MQI and LFI are shown



(a) The full image (b) The targets

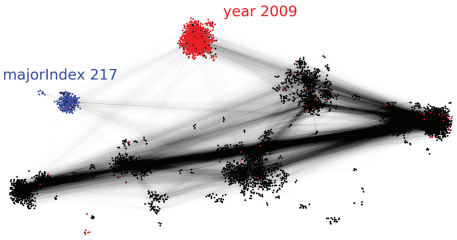
(c) FI (d) Seeded PageRank, $\rho = 10^{-12}$

(e) LFI-0.3 (f) Seeded PageRank, $\rho = 10^{-6}$

(g) MQI (h) LocalFiedler

(i) Greedy Grow (j) Greedy Shrink

Fig. 15 *Illustration of finding targets within an image* (a) *corresponding to the three low-conductance regions shown in* (b)*. The reference sets given to* MQI*,* FI*, and* LFI *are denoted by the yellow regions, which need to be either grown or shrunk to find the target. For growing, we compare against seeded PageRank, which is a spectral analogue of* FI *and* LFI*; for shrinking, we compare against a local Fiedler vector, a spectral analogue of* MQI*, as well as simple greedy approaches for both. The flow-based methods capture the borders nicely and give high recall for growing and high precision for shrinking. Among other things, in this case,* FI *grows too large on the right person* (c)*, whereas* LFI (e) *captures this target better. "RC" stands for recall and "PR" stands for precision.*

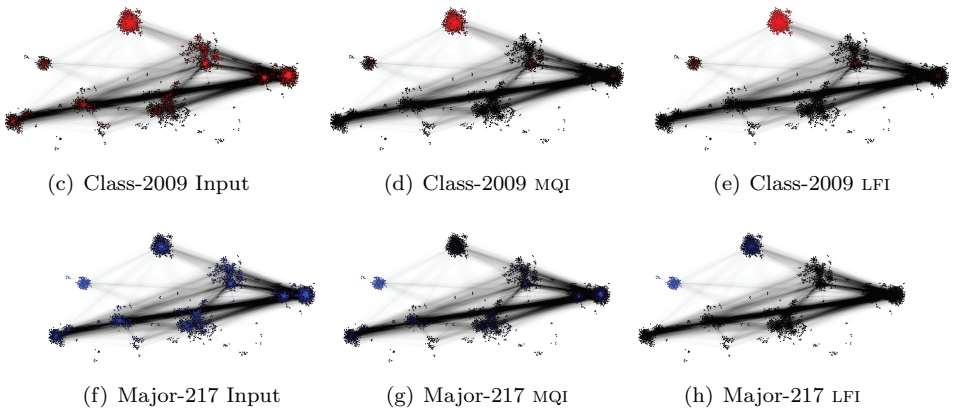


Feature	Vol.	Size	Cond.
Major-217	10696	200	0.26
Class-2009	32454	886	0.19

(a) Target sets for Johns Hopkins

Target	Set	Size	Cond.	Prec.	Rec.
Major-217	Input	1282	0.58	0.15	0.94
	MQI	Result 203	0.19	0.90	0.90
	LFI	Result 218	0.18	0.88	0.95
Class-2009	Input	1129	0.52	0.35	0.51
	MQI	Result 472	0.29	0.96	0.50
	LFI	Result 802	0.18	0.94	0.83

(b) Median statistics on input sets as well as MQI and LFI results



(c) Class-2009 Input (d) Class-2009 MQI (e) Class-2009 LFI

(f) Major-217 Input (g) Major-217 MQI (h) Major-217 LFI

Fig. 16 *The* MQI *and* LFI*-0.1 algorithms can also find target sets in the Johns Hopkins Facebook social network, even though they have fairly large conductance, which makes them more challenging. The algorithms use as the reference set a simple* 2*-hop BFS set with low precisions starting from a random target node. The layout for this graph has been obtained using the force-directed algorithm, which is available from the graph-tool project (Peixoto,* 2014*). The colors show the regions that are excluded (black) or included (red or blue) by each input set or algorithm over* 25 *trials.*

in Figures 16(b) to 16(h). The colors show the regions that are excluded (black) or included (red or blue) by each input set or algorithm over 25 trials.

We first summarize in Figure 16(b) the increase in precision for MQI and the increase in both precision and recall for LFI. The remaining figures illustrate the sets on top of the graph layout showing where the error occurs or the regions missed over these 25 trials. In particular, in Figures 16(c) and 16(f) we illustrate the BFS input for the target clusters. These inputs include a lot of false positives. In Figures 16(d) and 16(g) we illustrate the corresponding outputs of MQI. Note that MQI removes the most false positives by contracting the input set, but the outputs of MQI can have low recall as MQI can only shrink the input set. In Figures 16(e) and 16(h) we show that LFI is able to both contract and expand the input set and it obtains a good approximation to the target cluster.

This particular set of examples is designed to illustrate algorithm behavior in a simplified and intuitive setting. In both cases we see results that reflect the interaction

between the algorithm and the data transformation. For the images, we create a graph designed to correlate with segments, then run an algorithm designed to improve and find additional structure. For the social network, we simply take the data as given, without any attempt to change it to make algorithms perform better. However, with more specific tasks in mind, we would suggest altering the graph data in light of its targeted use as in Gleich and Mahoney (2015), Benson, Gleich, and Leskovec (2016), and Peel (2017).

9.3. Using Flow-Based Algorithms for Semisupervised Learning. Semisupervised learning on graphs is the problem of inferring the value of a hidden label on all vertices, when given a few vertices with known labels. Algorithms for this task need to assume that the graph edges represent a high likelihood of sharing a label. For instance, one of the datasets we will study is the MNIST dataset. Each node in the MNIST graph represents an image of a handwritten digit, and edges connect two images based on nearest neighbor relationships. The idea is that images that show similar digits should share many edges. Hence, knowing a few node labels would allow one to infer the hidden labels. Note that this is a related, but distinct, problem to the target set identification problem (section 9.2). The major difference is that we need to handle multiple values of a label and produce a value of the label for all vertices.

ASIDE 10. *If edges from the graph do not represent a high likelihood of a shared attribute, then there are scenarios where the graph data itself can be transformed such that this becomes the case* (*Peel,* 2017).

An early paper on this topic suggested that MinCut and flow-based approaches should be useful (Blum and Chawla, 2001). In our experiments, we compare flow-based algorithms LFI and FI with seeded PageRank, and we find that the flow-based algorithms are more sensitive to an increase in the size of the set of known labels. In the following experiments, this manifests as an increase in the recall while keeping the precision fixed. For these experiments, MQI is not a useful strategy as the purpose is to *grow and generalize* from a fixed and known set of labels to the rest of the graph.

There are three datasets we use to evaluate the algorithm for semisupervised learning: a synthetic stochastic block model, the MNIST digits data, and a citation network.

- **SBM**. SBM is a synthetic stochastic block model network. It consists of 6000 nodes in three classes, where each class has 2000 nodes. The probability of having a link between nodes in the same class is 0.005, while the probability of having a link between nodes in different classes is 0.0005. The network we use in the experiment has 36102 links. By our construction, the edges preferentially indicate class similarity.
- **MNIST**. MNIST is a k-NN (nearest neighbor) network (Lecun et al., 1998). The raw data consists of 60000 images. Each image represents a handwritten sample of one arabic digit. Thus, there are 10 total classes. In the graph, each image is represented by a single node and then connected to its 10 nearest neighbors based on Euclidean distance between the images when represented as vectors of grayscale pixels. We assume that edges indicate class similarity.
- **PubMed**. PubMed is a citation network (Namata et al., 2012). It consists of 19717 scientific publications about diabetes with 44338 citation links. Each article is labeled with one of three types. By our assumption, articles about one type of diabetes cite others about the same type more often.

The experiment goes as follows: for each class, we randomly select a small subset of nodes and we fix the labels of these nodes as known. We then run a spectral method

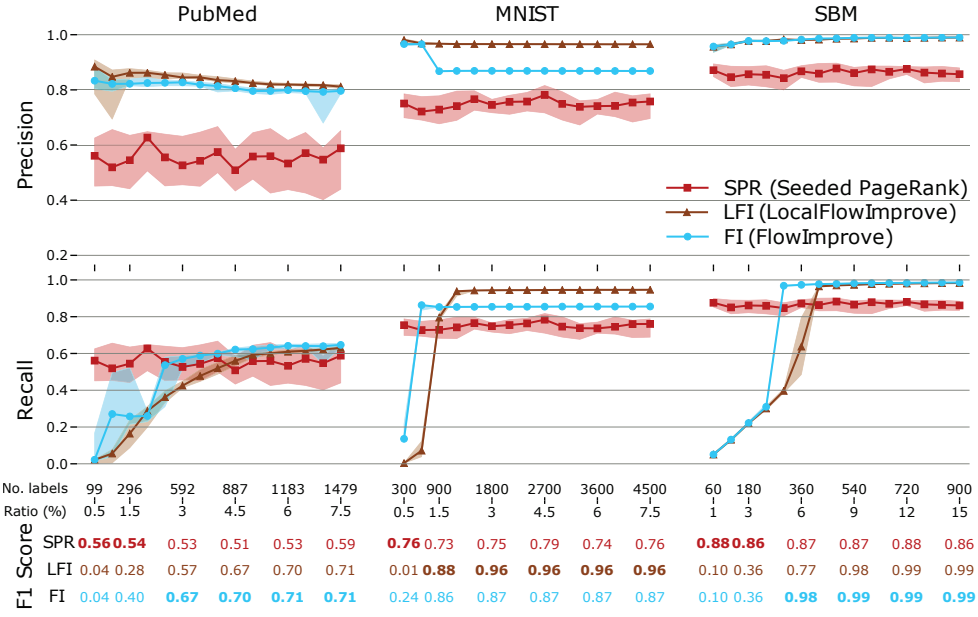


Fig. 17 *The horizontal axis shows the number of true labels included in the seeds and the plots are aligned with the tables so you can read off the $F1$ score as well as the associated precision and recall for each choice. These results on semisupervised learning show that the flow-based methods* LFI-0.1 *and* FI *are more sensitive to the number of known true labels included in the reference seed sets than seeded PageRank.*

or flow method with this set of nodes as reference. We vary the number of labeled nodes included from 0.5% to 15% of the class size. For each fixed number of labeled nodes, we repeat this 30 times to generate a distribution of precision, recall, and $F1$ scores (where $F1$ is the harmonic mean of precision and recall), and we represent an aggregate view of this procedure. For the flow methods, the output is a binary vector with 1 suggesting the node belongs to the class of reference nodes. Thus, it's possible that some nodes are classified into multiple classes, while other nodes remain unclassified. We consider the first case as false positives and the second case as false negatives when computing precision and recall. For the spectral method, we use the real-valued solution vector to uniquely assign a node to a class.

The results are shown in Figure 17 and show that the flow-based methods have uniformly high precision. As the set of known labels increases, the recall increases, yielding a higher overall $F1$ score. Furthermore, the regularization in LFI-0.1 causes the set sizes to be smaller than FI, which manifests as a decrease in recall compared with FI. In terms of why the flow-based algorithms have low recall with small groups of known labels sizes, note Lemma 3.5, which requires that the cut and denominator are reduced at each step. This makes it challenging for the algorithms and objectives to produce high recall when started with small sets unless the sets are exceptionally well separated.

9.4. Improving Thousands of Clusters on Large-Scale Data. In practice, it is often the case that one might want to explore thousands of clusters in a single large graph. For example, this is a common task in many computational topology pipelines (Lum et al., 2013). Another example that requires thousands of clusters

Table 4 *Runtimes in seconds for generating and improving clusters on small-scale biological networks and large-scale social networks. The input cluster to the flow-based improvement methods is the output of seeded PageRank. It takes around* 20 *minutes to generate the input clusters for large-scale social networks. Running the flow-based improvement algorithms takes around the same amount of time, except for* LFI-0.3 *on LiveJournal, which takes roughly* 30 *minutes. The time measurements reflect the pleasingly parallel computation of results for all clusters on a* 16*-core machine.*

				Time (s)				
graph	nodes	edges	clusters found	seeded PR	MQI	LFI-0.3	LFI-0.6	LFI-0.9
sfld	232	16k	342	18	0.5	1.7	1.6	1.5
ppi	1096	13k	1199	46	1	2.6	2.5	2.3
orkut	3M	117M	13799	1130	171	838	701	628
livej	4.8M	69M	31622	1057	105	1940	1326	1094

is computing the network community profile (Leskovec et al., 2009, 2008; Leskovec, Lang, and Mahoney, 2010), which shows a conductance-vs.-size result for a large number of sets, as a characteristic feature of a network. In this section, we will explore the runtime of the flow-based solvers on two small biological networks and two large social networks, where nodes are individuals and edges represent declared friendship relationships:

- **sfld** has 232 nodes and 15570 edges in this graph (Brown et al., 2006).
- **ppi** has 1096 nodes and 13221 edges (Pagel et al., 2004).
- **orkut** has 3,072,441 nodes and 117,185,083 edges (Mislove et al., 2007). This dataset can be accessed via Leskovec and Krevl (2014).
- **livej** has 4,847,571 nodes and 68,993,773 edges (Mislove et al., 2007). This dataset can be accessed via Leskovec and Krevl (2014).

The goal is to enable studies such as those discussed above using instead the flow-based algorithms as a subroutine on the graphs. To generate seed sets to refine, we use seeded PageRank. Each input set is the result of a seeded PageRank algorithm on a random node with a variety of settings to generate sets of size from a few nodes to up to around 10000 nodes. For each resulting set, we then run the MQI, LFI-0.9, LFI-0.6, and LFI-0.3 improvement methods. Our code (Fountoulakis et al., 2019b), which has a Python interface and methods that are implemented using C++, is used for all of these experiments and runtimes. Our machine environment has a dual CPU Intel E5-2670 (8 cores) CPU with 128 GB RAM. We parallelize over individual runs of the seeded PageRank and flow methods using the Python Multiprocessing module using a common shared graph structure. Note that each individual run is independent.

In this way, we are able to explore tens of thousands of clusters in around 30–40 minutes, as we demonstrate in Table 4. There, we present runtimes for producing the seeded PageRank sets and then refining them with the flow-based methods. Note that the fastest method is MQI, which is even faster than the seeded PageRank method that generates the input sets. This is because MQI only explores the input subcluster, while LFI reaches outside of the input seed set of nodes. Also, note the dependence of the runtime for LFI on the parameter δ. The larger the parameter δ for LFI, the smaller the part of the graph that it explores outside of the input set of nodes. This property is captured in Table 4 by the runtime of LFI.

Fig. 18 *A spectral embedding of the US Highway Network corresponding to the first and second nontrivial eigenvectors of the Laplacian matrix. The embedded locations of major cities are labeled as well. Node color is determined by the true longitude of a node, which shows that the first eigenvector of the Laplacian correlates with an east-west split of the network. This global embedding, however, compresses major regions of the northeastern US (Washington, New York, Boston) as well as the western US (Los Angeles, San Diego, Phoenix).*

9.5. Using Flow-Based Methods for Local Coordinates. A common use case for global (but also local (Lawlor, Budavári, and Mahoney, 2016a,b)) spectral methods on graphs and networks is using eigenvector information in order to define coordinates for the vertices of a graph. This is often called a spectral embedding or eigenvector embedding (Hall, 1970); it may use two or more eigenvectors directly or with simple transformations of them in order to define coordinates for each node (Ng, Jordan, and Weiss, 2001). The final choice is typically made for aesthetic reasons. An example of a spectral embedding for the US highway network is shown in Figure 18. One of the problems with such global embeddings is that they often squash interesting and relevant regions of the network into filamentary structures. For instance, notice that both the extreme pieces of this embedding compress massive and interesting population centers of the US on the East and West Coasts. Alternative eigenvectors show different but similar structure. A related problem is that they smooth out interesting features, making them difficult to use.

Semisupervised eigenvectors are one way to address this aspect of global spectral embeddings (Hansen and Mahoney, 2014; Lawlor, Budavári, and Mahoney, 2016a,b). These seek orthogonal vectors that minimize a constrained Rayleigh quotient. One challenge in using related ideas to study *flow-based* computation is that the solution of flow problems is fundamentally discrete and binary; that is, a spectral solution produces a real-valued vector whose entries, e.g., for seeded PageRank, can be interpreted as probabilities. We can thus meaningfully discuss and interpret suboptimal, orthogonal solutions. Flow computations only give 0 or 1 values, where orthogonality implies disjoint sets.

ASIDE 11. *A bigger issue with spectral embeddings is that they often produce useless results for many large networks; see Lang* (2005). *Here, we use networks where these techniques yield interesting results.*

In this section, we investigate how flow-based methods can be used to compute real-valued coordinates that can show different types of structure within data compared with spectral methods. In the interests of space, we are going to be entirely procedural with our description; justification is provided in Appendix A.

Given a reference set R, we randomly choose N subsets (we use 500–2500 subsets) of R with exactly k entries; for each subset we add all nodes within distance d and call the resulting sets $R_1, \ldots, R_N$. These serve as inputs to the flow algorithms. For each

Algorithm 9.1 The local flow-based algorithm to generate flow-based coordinates.

Require: A graph G, a set R, and parameters
- N: the number of sets to sample
- k: the size of each subset
- d: the expansion distance
- c: the dimension of the final embedding
- `improve`: a cluster improvement algorithm

Ensure: An embedding of the graph into c coordinates for each node

Let n be the number of vertices.
Allocate X, an n-by-N matrix of zeros.
for i in 1 to n **do**
  Let T be a sample of k entries from R at random without replacement
  Let R_i be the set of T and also all vertices within distance d from T
  Let S_i be the set that results from `improve`(G, R_i)
  Set $X[v, i] = 1$ for all $v \in S_i$
Compute the rank-c truncated SVD of X and let U be the left singular vectors.
Return U; each row gives the c coordinates for a node

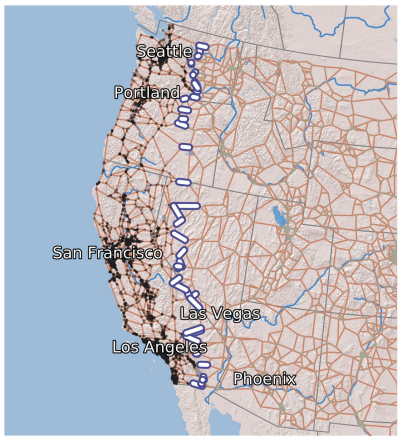
subset, we compute the result of a flow-based improvement algorithm, which gives us sets S_i. For each S_i, we form an indicator vector over the vertices, x_i, where the entry is 1 if the vector is in the set, and 0 otherwise. We assemble these vectors as columns of a matrix X, and we use the coordinates of the dominant two left singular vectors as flow-based coordinates. This procedure is given in Algorithm 9.1. Note also that this procedure can be performed with spectral algorithms as well (see the appendix for additional details).

The results of using Algorithm 9.1 (see parameters in Appendix A) to generate local coordinates for a set of vertices on the West Coast of the US highway map are shown in Figure 19. The set of vertices shown in Figure 19(a) is in a region where the spectral embedding compresses substantial information. This region is shown on a map in Figure 19(b), and it includes major population centers on the West Coast. In Figure 19(c), we show the result of a local spectral embedding that uses seeded PageRank in Algorithm 9.1, along with a few small changes that are discussed in our reproducibility section. (Here, we note that these changes do not change the character of our findings, they simply make the spectral embedding look better.) In the spectral embedding, the region shows two key areas: (1) Seattle, Portland, and San Francisco and (2) Los Angeles, San Diego, and Phoenix. In Figure 19(d), we show the result of the local flow-based embedding that uses LFI-0.1 as the algorithm. This embedding clearly and distinctly highlights major population centers, and it does so in a way that is clearly qualitatively different from spectral methods.

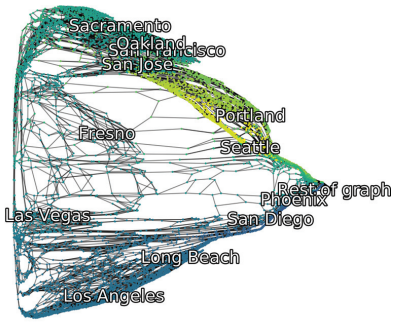
We repeat this analysis on the MGS dataset to highlight the local structure in a particularly dense region of the spectral embedding that was used for Figure 13. The seed region we use is shown in Figure 20(a) and has 201,252 vertices, which represents almost half the total graph. We again use Algorithm 9.1 (see parameters in Appendix A) to obtain local spectral (Figure 20(b)) and local flow embeddings (Figure 20(c)). Again, we find that the local flow embedding shows considerable substructure that is useful for future analysis.

As a simple validation that this substructure is real, we use the two-dimensional embedding coordinates as input to a k-means clustering procedure on both the local

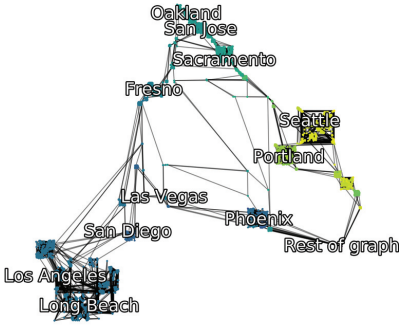
(a) The subset of nodes (left) from the spectral embedding of the US Highway Network used to compute local embeddings



(b) The same subset shown on a map with major cities labeled



(c) The local spectral embedding of the US West Coast cities

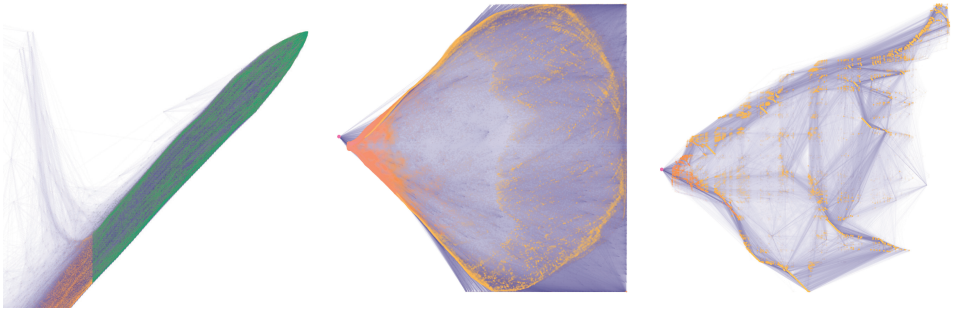


(d) The local flow embedding of the US West Coast cities

Fig. 19 *We select a subset of* 7143 *nodes that are compressed in the spectral embedding of the US Highway network (shown in red and blue in* (a) *and* (b)*) that represent the majority of major cities on the West Coast. Note that interior cities such as Phoenix and Las Vegas are not included in the set. In* (c) *and* (d) *we show the results of running the pipeline from Algorithm* 9.1 *to generate local spectral and local flow-based embeddings in two dimensions. The color of a node is determined by its north-south latitude. Note that both include Phoenix and Las Vegas. The local flow embedding clearly and distinctly delineates clusters corresponding to major population centers, whereas the local spectral embedding shows a smooth view with only two major regions:* (1) *Northern California to Seattle and* (2) *Southern California to Phoenix and Las Vegas.*

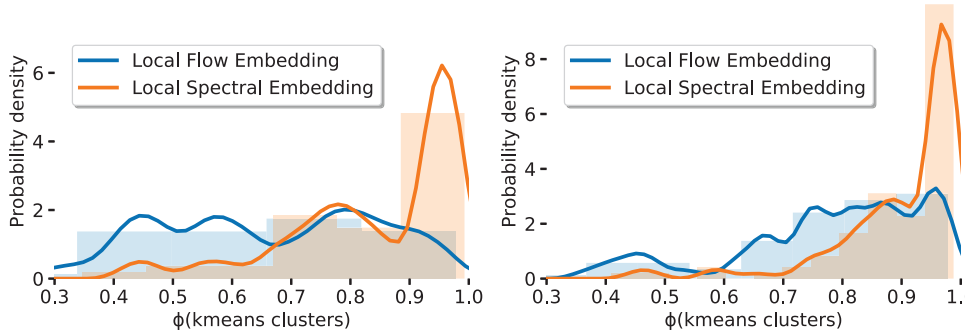
spectral and the local flow coordinates. For each cluster that results from this procedure, we compute its conductance. Histograms of conductance values are shown in Figure 21 for $k = 50$ and $k = 100$. Both of these histograms show consistently smaller conductance values for the flow-based embedding.

10. Discussion and Conclusion. Our goal with this survey is to highlight the advantages and wide utility of flow-based algorithms for improving clusters. The literature on these methods is much smaller than for other graph computation method-



(a) The seed region (b) Local spectral embedding (c) Local flow embedding

Fig. 20 *Local spectral and local flow embeddings of the large,* 201,252 *node, seed region, shown in green in* (a)*, that is compressed in the global spectral embedding from Figure* 13*. In* (b)*, the local spectral embedding shows the nodes colored with the same colors as in Figure* 13*. Nodes that were not touched by the local embedding are shown with the big node on the right-hand side. In* (c)*, the local flow embedding is shown with the same color scheme and same big node on the right-hand side. Note that the spectral embedding does not show any clear substructure besides a top-bottom split. In contrast, the flow embedding shows a number of pockets of structure indicative of small conductance subsets.*



(a) $k = 50$ (b) $k = 100$

Fig. 21 *A histogram of cluster conductance scores that come from using k-means on the two-dimensional local spectral and local flow embeddings from Figure* 20*. These show that the flow embedding produces clusters with smaller conductance, and they support the intuition from Figure* 20 *that the additional structure suggested by the flow embedding reflects meaningful substructure within the data.*

ologies, despite attractive theoretical benefits. For example, global spectral methods based on random walks or eigenvectors are ubiquitous in computer science, machine learning, and statistics. Here, we have illustrated similar possibilities for flow-based methods. We have also shown that these local flow-based improvement algorithms can scale to very large graphs, often returning outputs without even touching most of the graph, and that many popular machine learning and data analysis uses of flow-based methods can be applied to them. This is the motivation behind our software package where these algorithms have been implemented (Fountoulakis et al., 2019b). An alternative implementation is available in Julia (Veldt, 2019). These results and methods open the door for novel analyses of very large graphs.

As an example of these types of novel analyses, note that the fractional ratio δ_k inside MQI, FlowImprove, and LocalFlowImprove (Algorithms 6.1, 7.1, and 8.1) can be interpreted as a ratio between the numerator and denominator. This enables one to search for a value of δ that would correspond to a given solution. See the ideas in Veldt, Wirth, and Gleich (2019) for how to use search methods to choose δ for a specific application of clustering.

In our explanation of the theory behind the methods, we often encountered situations where we could have made more general, albeit more complex, statements. Our guiding principle was to make it easy to appreciate the opportunities for these methods. As an example of where there are more general results, note that much of the theory of this survey holds where $\mathrm{vol}(R) \geq \mathrm{vol}(\bar{R})$ for the seed. For instance, the MQI, FlowImprove, and LocalFlowImprove procedures are all well-defined algorithms in these scenarios, although our theory statements list the explicit requirement that $\mathrm{vol}(R) \leq \mathrm{vol}(\bar{R})$. What happens in these scenarios is that some of the details of the runtime and other aspects change. In terms of another generalization, the methods could have been stated in terms of a general volume function as noted in section 3.7. Again, however, this setting becomes more complex to state for noninteger volume functions and when considering a number of other subtle issues. In these cases, we looked for the explanations that would make the underlying issues clear and focused on conductance in order to do so.

There are a number of interesting directions that are worth exploring further. First, in the theory from this survey, the binary search or bisection-based search methods have superior worst-case times. However, in practice, these methods are rarely used. For instance, our own implementations always use the Dinkelbach greedy framework. This is because this strategy commonly terminates in just a few iterations, as was noted in both the MQI and FlowImprove papers (Lang and Rao, 2004; Andersen and Lang, 2008), yet there is still no theoretically satisfying explanation. To provide some data, for the LocalFlowImprove experiments in Figure 14, we never needed to evaluate more than 10 values of the fractional ratio to find the optimal solution in Dinkelbach's algorithm. As evidence that this effect is real, note that Lemma 3.5 actually shows that $\mathrm{cut}(R)$ is a bound on the number of iterations for Dinkelbach's algorithm. But for weighted graphs, this becomes $\mathrm{cut}(R)/\mu$, where μ is the minimum spacing between elements (think of the floating-point machine value ε). A specific case where this type of insight would be helpful is in terms of weighted graphs with nonnegative weights. Dinkelbach's algorithm does not appear to be much slower on such problems, yet the worst-case theory bound is extremely bad and depends on the minimum spacing between elements.

Another direction of exploration is a set of algorithms that span the divide between the fractional program and the MinCut problem. For instance, it not necessary to completely solve the flow problem to optimality (until the last step) since all that is needed is a result that there is a better solution available. This offers a wide space in which to deploy recent advances in Laplacian-based solvers to handle the problem, especially because the electrical flow-based solutions largely correspond to PageRank problems. It seems optimistic, but reasonable, to expect good solutions in time that are more like a random walk or spectral algorithm.

Finally, another direction for future research is to study these algorithms in hypergraphs (Veldt, Benson, and Kleinberg, 2020) and other types of higher-order structures. This was a part of early work on graph cuts in images that showed that problems where hyperedges had at most three nodes could be solved exactly (Kolmogorov and

Zabih, 2004). More recently, hypergraphs have been used to identify refined structure in networks based on motifs (Li and Milenkovic, 2017).

In closing, our hope is that this survey and the associated LocalGraphClustering package (Fountoulakis et al., 2019b) helps to make these powerful and useful methods—basic flow-based analysis of smaller graphs, but especially local flow-based analysis of very large data graphs—more common in the future.

Part IV. Replicability Appendices and References.

Appendix A. Replicability Details for Figures and Tables. In the interest of reproducibility and replicability, we provide additional details on the methods underlying the figures. To replicate these experiments, see our publicly available code (Fountoulakis et al., 2019a). All of the seeded PageRank examples in this survey use an ℓ_1-regularized PageRank method (Fountoulakis et al., 2019c). We use ρ to denote the regularization parameter in ℓ_1-regularized PageRank, α to denote the teleportation parameter in ℓ_1-regularized PageRank, and δ to denote the parameter of LocalFlowImprove.

Our implementations use Dinkelbach's method, Algorithm 3.1, for the fractional programming problem and Dinic's algorithm for exact solutions of the weighted MaxFlow problems at each step. Put another way, for MQI, we use Algorithm 6.1 and Dinic's algorithm to solve the MaxFlow problems. For FlowImprove and LocalFlowImprove, we use the Dinkelbach variation on Algorithm 8.1 with Dinic's algorithm used to compute blocking flows in Algorithm 8.3. We use the same implementation for LocalFlowImprove and FlowImprove and simply set $\delta = 0$ for FlowImprove. Using Dinkelbach's method and Dinic's MaxFlow has a worse runtime in theory, but better performance in practice. The implementations always return the smallest connected set that achieves the minimum of the objective functions. They also always return a set with less than half the total volume of the graph.

Figure 1. We use the implementation of the Louvain algorithm in Aynaud (2018). We use our own code to generate the SBM. The code for this experiment is in the notebook `sbm_demo.ipynb` in the subdirectory `ssl` available in Fountoulakis et al. (2019a).

Figure 2. This is a geometric-like stochastic block model "hybrid." A short description of the data-generation procedure follows but the code serves as the precise description. Create g groups of n points. Each group is assigned the same random two-dimensional spatial coordinate from a standard mean 0, variance 1 normal distribution. But each node within a group is also perturbed by a mean 0 random normally distributed amount with variance σ. Add p additional points with normally distributed ρ (a "center" group). These determine the coordinates of all the nodes. Now, add edges to k nearest neighbors and also within radius ϵ. For this experiment, we set $g = 25$, $n = 100$, $\sigma = 0.05$, $p = 2000$, $\rho = 5$, $k = 5$, and $\epsilon = 0.06$. The code is in the Jupyter notebook `Geograph-Intro.ipynb` (Fountoulakis et al., 2019a).

Figure 3. The image can be downloaded from van der Walt et al. (2014). It is turned into a graph using the procedure described in Appendix B. In particular, we set $r = 80$, $\sigma_p^2 = \mathcal{O}(10^2)$, and $\sigma_c^2 = l/10$, where l is the maximum of the row and column lengths of the image. The code for this experiment is in the Jupyter notebook `astronaut.ipynb` in the subdirectory `usroads` available in Fountoulakis et al. (2019a).

Figure 4. The original image is 100 by 100 pixels with a 13 pixel wide by 77 pixel tall vertical strip and a 77 pixel wide by 25 pixel tall horizontal strip that

intersect in a 13-by-25 pixel region. This setup and intersection produce a rotated T-like shape centered in the 100-by-100 pixel grid. To generate the noisy, blurred figure, we used a Moffat kernel (Moffat, 1969) that arises from stellar photography (parameters $\alpha = 1.5$, $\beta = 1.2$, and length scale 5) and added uniform $[-0.1, 0.1]$ noise (roughly 38% of max blurred value 0.261) for each pixel before scaling by $1/0.3$ and clamping to the $[0, 1]$ range. This stellar photography scenario was chosen to simulate a binary image reconstruction scenario. Let f be a noisy, blurry image with values in the range $[0, 1]$. Let G be the grid graph associated with the grid underlying the image f. Make sure to read section 6 before reading the details of the reconstruction algorithm. Then we construct the following augmented graph: connect s to $i \in V$ with weight $\delta f_i d_i$ (where d is the degree); connect t to $i \in V$, where $f_i = 0$ with weight ∞. We show the MinCut solution S for the two values of δ explained in the problem. This corresponds to a minimization problem similar to (6.3), namely, minimize $\operatorname{cut}(S) - \delta \sum_{i \in S} d_i f_i$ subject to $S \subseteq \{i \mid f_i > 0\}$, which uses a *biased* notion of volume $\nu(S) = \sum_{i \in S} d_i f_i$ (as in section 3.7). In this case, if δ is 0.04, then we can no longer continue improving the result and we end up with the convex set. For $\delta = 0.11$, we have the rough reconstruction of the original shape. For our own purposes, we used a Julia implementation (Veldt, 2019) of the flow code that is available in the `mqi-images` subdirectory in Fountoulakis et al. (2019a).

Figure 12. All details are given in the main text of the survey. The code is in the Jupyter notebook `usroads-figures.ipynb` in the subdirectory `usroads` available in Fountoulakis et al. (2019a).

Table 3. This table provides additional details for the results of Figure 12. The code for this experiment is in the Jupyter notebook `usroads-figures.ipynb` in the subdirectory `usroads` available in Fountoulakis et al. (2019a).

Figure 13. This dataset has been obtained from Lawlor, Budavári, and Mahoney (2016a). It is a $k = 32$ nearest neighbor graph constructed on the MGS in SDSS Data Release 7. Each galaxy is captured in a 3841-band spectral profile. Each spectrum is normalized based on the median signal over 520 bands selected in Lawlor, Budavári, and Mahoney (2016a). Since the results are sensitive to this set and it is not available elsewhere, the indices of the bands were as follows:

856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875,
876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895,
896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915,
916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935,
936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955,
956, 957, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265,
1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1281, 1282,
1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299,
1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316,
1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329, 1330, 1331, 1332, 1333,
1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350,
1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367,
1368, 1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1383, 1384,
1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401,
1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418,
1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 1435,
1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964,
1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981,
1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998,
1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015,
2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2106, 2107, 2108, 2109, 2110,
2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127,
2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144,
2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161,
2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178,
2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195,
2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212,
2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229,
2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246,
2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258

We create a node for each galaxy and connect vertices if either is within the 16 closest vertices to the other based on a Euclidean distance after this median normalization.

The graph is then weighted proportional to this distance and the distance to the 8th nearest neighbor based on a k-nearest neighbor tuning procedure in manifold learning. (The results for spectral embeddings are somewhat sensitive to this procedure.) Formally, let ρ_i be the distance to the 8th nearest neighbor (or ∞ if all of these distances are 0). We add a weighted undirected edge based on node i to node j with distance $d_{i,j}$ as $W_{i,j} = \exp(-(d_{i,j}/\rho_i))$. If i and j are both nearest neighbors, then we increment the weights, so the construction is symmetric. Each node also has a self-loop with weight 1. The adjacency matrix of the graph has 32,229,812 nonzeros, which is 15,856,315 edges and 517,182 self-loops. The code for this experiment is in the Jupyter notebook `hexbingraphplots_global.jl` in the subdirectory `flow_embedding/hexbin_plots` available in Fountoulakis et al. (2019a). The full code to process the graph is available upon request.

Figure 14. For this experiment we used seeded PageRank to find the seed set for the flow algorithms MQI and LocalFlowImprove. We set the teleportation parameter of the seeded PageRank algorithm to 0.01. The code for this experiment is in the Jupyter notebook `plot_cluster_improvement.ipynb` in the subdirectory `cluster_improvement` available in Fountoulakis et al. (2019a).

Figure 15. In our experiments constructing the graph from the image, we follow Appendix B and we set $r = 80$, $\sigma_p^2 = \mathcal{O}(10^2)$, and $\sigma_c^2 = l/10$, where l is the maximum of the row and column lengths of the image. The code for this experiment is in the Jupyter notebook `image_segmentation.ipynb` in the subdirectory `image_segmentation` available in Fountoulakis et al. (2019a).

Figure 16. The input is a 2-hop BFS set starting from a random target node. We independently generate 25 such BFS sets. The transparency level of red or blue nodes is determined by the ratio of including each node in the resulting sets. The code for this experiment is in the Jupyter notebook `social.ipynb` in the subdirectory `social` available in Fountoulakis et al. (2019a). Specific details about tuning can also be found in the code.

Figure 17. For every class we randomly select a small percentage of labeled nodes; the exact percentages are given in the main text. The nodes that are selected from each class are considered a single seed set. For each seed set and for each class we use seeded PageRank with teleportation parameter equal to 0.01. This procedure provides one PageRank vector per class. For each unlabeled node in the graph we look at the corresponding coordinates in the PageRank vectors and we give to each unlabeled node the label that corresponds to the largest value in the PageRank vectors. For flow methods, for every labeled node that is used, we run one step of BFS to expand the single seed node to a seed set. The expanded seed set is used as input to the flow methods. We find a cluster and each node in the cluster is considered to have the same label as the seed node. Based on this technique, it is possible that one node can be allocated to more than one class; we consider such nodes as false positives. The code for this experiment is in the Jupyter notebook `semisupervised_learning.ipynb` in the subdirectory `ssl` available in Fountoulakis et al. (2019a). The MNIST graph was weighted for this experiment. The distance between two images is computed by a radial basis function with width 2. To make robust the process of rounding a diffusion vector to class labels, we use a strategy from Gleich and Mahoney (2015) that involves rounding to classes based on the node with the smallest rank in the ranked list of each diffusion vector.

Table 4. The code for this experiment is in the Jupyter notebooks in the subdirectory `large_scale` available in Fountoulakis et al. (2019a).

Figure 18. We use the eigenvector of the Laplacian matrix $D - A$ associated with the smallest nonzero eigenvalues to compute the vectors v_1 and v_2. The coordinates of the plot are generated by assigning x and y based on the rank of a node in v_1 and v_2 in a sorted order. This has the effect of stretching out the eigenvector layout, which often compresses many nodes at similar point. The color of the nodes is proportional to the east-west latitude. The code for this experiment is in the notebook `usroads-embed.ipynb` in the subdirectory `usroads` in Fountoulakis et al. (2019a).

Figure 19. We use Algorithm 9.1 with $N = 500$ sets, $k = 1$, $d = 20$, $c = 2$, along with LFI-0.1 as the improve algorithm. For the local spectral embedding, we use the same seeding parameters as seeded PageRank with $\rho =$ 1e-6. When we create the matrix X for seeded PageRank, we take the base-10 logarithm of the result value (which is always between 0 and 1). For vertices with 0 values, we assign them -10, which is lower than any other value. This gave a more useful embedding and helped the spectral embedding show more structure. The node labeled "Rest of graph" was manually placed in both cases because the embedding does not suggest a natural place for it. Here, we also used the ranks of the nodes in a sorted order, which helps to spread out nodes that are all placed in exactly the same location. The code for this experiment is in the Jupyter notebook `usroads-embed.ipynb` in the subdirectory `usroads` available in Fountoulakis et al. (2019a).

Figure 20. We use Algorithm 9.1 with $N = 500$ sets, $k = 1$, $d = 3$, $c = 2$, along with LFI-0.1 as the improve algorithm. We used the same local spectral methodology as in Figure 19. The large red node, which represents the remainder of the graph and all "unembedded nodes," is manually placed to highlight edges to the rest of the graph. Here, we also used the ranks of the nodes in a sorted order, which helps to spread out nodes that are all placed in exactly the same location. The code for this experiment is in the Python script `flow_embedding.py` in the subdirectory `flow_embedding` available in Fountoulakis et al. (2019a) and Jupyter notebooks in the subdirectory `flow_embedding/hexbin_plots` available in Fountoulakis et al. (2019a). The Python script needs to be run first to generate data and then the notebook can be used to generate the figures.

Figure 21. The code is in the Jupyter notebooks `social.ipynb` in the subdirectory `flow_embedding/cond_hists` available in Fountoulakis et al. (2019a). They both need the embedding results from Figure 20 to generate the figures.

The Rationale for the Local Flow Embedding Procedure. We now briefly justify the motivation for the structure of the local flow embedding algorithm. The key idea is that spectral algorithms are based on linear operations: if we have any way of sampling the reference set R with a normalized set indicator T such that $E[T] = \frac{1}{|R|}\mathbf{1}_R$, then if f is a linear function, such as an exact seeded PageRank computation, we have $E[f(T)] = f(\frac{1}{|R|}\mathbf{1}_R)$. This expectation corresponds to the seeded PageRank result on the entire set. To include another dimension, we could seek to find an orthogonal direction to $E[f(T)]$, such as is done with constrained eigenvector computations. It is this linear function perspective that inspired our flow embedding algorithm: collect samples of $f(T_i)$ into a matrix and then use the SVD on the samples of T to approximate $E[f(T)]$ and the orthogonal component (given by the second singular vector). While some of these arguments can be formalized and made rigorous for a linear function, that is an orthogonal discussion (pun intended). Here, we simply make the observation that this perspective enables us to use a nonlinear procedure f without any issue. This gave rise to Algorithm 9.1, which differs only in that we grow the sets $T \to R_i$ by including all vertices within graph distance d.

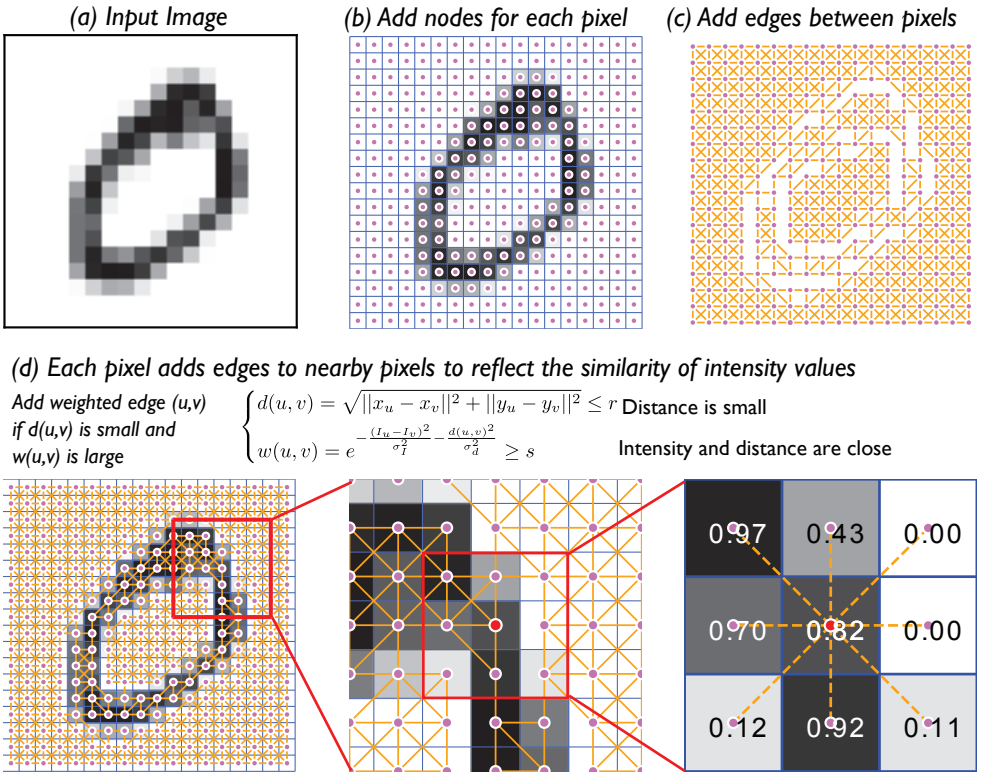


Fig. 22 *We turn an image into a graph by adding a node for every pixel* (b). *Then we connect the nodes if the associated pixels are close by (distance less than r) as well as have similar pixel values). We weight the edge by the degree of similarity. The resulting graph has small conductance sets when there are regions with similarly colored pixels.*

Appendix B. Converting Images to Graphs. For illustration purposes, we use images to generate graphs in various examples throughout this survey. The purpose of this construction is that visually distinct segments of the picture should have small conductance. Given an image, we create a weighted nearest-neighbor graph using a Gaussian kernel as described in Shi and Malik (2000). We create a node for each pixel. Then we connect pixels with weighted edges. In particular, let w_{ij} denote the the weight of the edge between pixels i and j, let $p_i \in \mathbb{R}^2$ be the position of pixel i, $c_i \in \mathbb{R}^3$ the color representation of pixel i, σ_d^2 the variance for the position, and σ_I^2 the variance for the color. Then, we define the edge weights as

$$w_{ij} := \begin{cases} e^{-\frac{\|p_i - p_j\|_2^2}{\sigma_d^2} - \frac{\|c_i - c_j\|_2^2}{\sigma_I^2}} & \text{if } \|p_i - p_j\|^2 \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

Note that there is a region r that restricts the feasible edges, illustrated in Figure 22.

Acknowledgments. We would like to thank many individuals for discussions about these ideas over the years. We would also like to especially thank Nate Veldt for a careful reading of an initial draft, Charles Colley for reviewing a later draft, both

Di Wang and Satish Rao for discussions on geometric aspects of flow algorithms, and finally Kent Quanrud for many helpful pointers.

REFERENCES

E. Abbe (2018), *Community detection and stochastic block models: Recent developments*, J. Mach. Learn. Res., 18, art. 177, http://jmlr.org/papers/v18/16-480.html. (Cited on p. 62)

M. Ackerman and S. Ben-David (2018), *Measures of clustering quality: A working set of axioms for clustering*, in Advances in Neural Information Processing Systems 21, Curran Associates, pp. 121–128, http://papers.nips.cc/paper/3491-measures-of-clustering-quality-a-working-set-of-axioms-for-clustering.pdf. (Cited on p. 62)

R. Andersen, F. Chung, and K. Lang (2006), *Local graph partitioning using PageRank vectors*, in FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 475–486. (Cited on pp. 65, 87, 88, 118)

R. Andersen, S. O. Gharan, Y. Peres, and L. Trevisan (2016), *Almost optimal local graph clustering using evolving sets*, J. ACM, 63, art. 15. (Cited on p. 87)

R. Andersen and K. J. Lang (2006), *Communities from seed sets*, in Proceedings of the 15th International Conference on the World Wide Web, ACM, pp. 223–232. (Cited on p. 87)

R. Andersen and K. J. Lang (2008), *An algorithm for improving graph partitions*, in Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 651–660. (Cited on pp. 63, 67, 83, 84, 85, 100, 101, 130)

S. Arora, S. Rao, and U. Vazirani (2009), *Expander flows, geometric embeddings and graph partitioning*, J. ACM, 56 (2), art. 5. (Cited on p. 88)

H. Avron and L. Horesh (2015), *Community detection using time-dependent personalized PageRank*, in Proceedings of the 32nd International Conference on Machine Learning, PMLR, pp. 1795–1803, http://proceedings.mlr.press/v37/avron15.pdf. (Cited on p. 87)

P. Awasthi, A. S. Bandeira, M. Charikar, R. Krishnaswamy, S. Villar, and R. Ward (2015), *Relax, no need to round: Integrality of clustering formulations*, in Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ACM, pp. 191–200. (Cited on p. 62)

T. Aynaud (2018), *Python-Louvain*, https://github.com/taynaud/python-louvain. (Cited on p. 131)

M. Belkin, P. Niyogi, and V. Sindhwani (2006), *Manifold regularization: A geometric framework for learning from labeled and unlabeled examples*, J. Mach. Learn. Res., 7, pp. 2399–2434. (Cited on p. 87)

S. Ben-David (2018), *Clustering—what both theoreticians and practitioners are doing wrong*, in Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18), pp. 7962–7964, https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17420. (Cited on p. 61)

A. Benson, D. F. Gleich, and J. Leskovec (2016), *Higher-order organization of complex networks*, Science, 353 (6295), pp. 163–166, https://doi.org/10.1126/science.aad9029. (Cited on pp. 70, 123)

A. L. Bertozzi and A. Flenner (2016), *Diffuse interface models on graphs for classification of high dimensional data*, SIAM Rev., 58, pp. 293–328, https://doi.org/10.1137/16M1070426. (Cited on p. 62)

D. Bienstock, M. Chertkov, and S. Harnett (2014), *Chance-constrained optimal power flow: Risk-aware network control under uncertainty*, SIAM Rev., 56, pp. 461–495, https://doi.org/10.1137/130910312. (Cited on p. 62)

V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre (2008), *Fast unfolding of communities in large networks*, J. Statist. Mech. Theory Exper., 10, art. P10008. (Cited on p. 63)

A. Blum and S. Chawla (2001), *Learning from labeled and unlabeled data using graph mincuts*, in Proceedings of the Eighteenth International Conference on Machine Learning, Morgan Kaufmann, pp. 19–26, https://web.archive.org/web/20190718171105/http://www.aladdin.cs.cmu.edu/papers/pdfs/y2001/mincut.pdf. (Cited on pp. 89, 123)

S. Boyd and L. Vandenberghe (2004), *Convex Optimization*, Cambridge University Press. (Cited on pp. 92, 93)

Y. Boykov and G. Funka-Lea (2006), *Graph cuts and efficient N-D image segmentation*, Internat. J. Comput. Vision, 70, pp. 109–131. (Cited on p. 90)

Y. Boykov and V. Kolmogorov (2004), *An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision*, IEEE Trans. Pattern Anal. Mach. Intell., 26, pp.

1124–1137. (Cited on p. 90)
Y. Boykov and O. Veksler (2006), *Graph cuts in vision and graphics: Theories and applications*, in Handbook of Mathematical Models in Computer Vision, Springer-Verlag, Boston, MA, pp. 79–96. (Cited on p. 90)
U. Brandes and T. Erlebach, eds. (2005), *Network Analysis: Methodological Foundations*, Springer. (Cited on p. 62)
S. D. Brown, J. A. Gerlt, J. L. Seffernick, and P. C. Babbitt (2006), *A gold standard set of mechanistically diverse enzyme superfamilies*, Genome Biology, 7, art. R8. (Cited on p. 125)
E. J. Candès, J. Romberg, and T. Tao (2006), *Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information*, IEEE Trans. Inform. Theory, 52, pp. 489–509. (Cited on p. 109)
J. J. Carrasco, D. C. Fain, K. J. Lang, and L. Zhukov (2003), *Clustering of bipartite advertiser-keyword graph*, in Proceedings of the Workshop on Clustering Large Data Sets at the 2003 International Conference on Data Mining, pp. 72–79. (Cited on p. 62)
T. F. Chan, S. Esedoḡlu, and M. Nikolova (2006), *Algorithms for finding global minimizers of image segmentation and denoising models*, SIAM J. Appl. Math., 66, pp. 1632–1648, https://doi.org/10.1137/040615286. (Cited on p. 89)
P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng (2011), *Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs*, in Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing, pp. 273–282. (Cited on p. 89)
F. Chung (2007a), *The heat kernel as the PageRank of a graph*, Proc. Natl. Acad. Sci. USA, 104, pp. 19735–19740. (Cited on p. 87)
F. Chung (2007b), *Random walks and local cuts in graphs*, Linear Algebra Appl., 423, pp. 22–32. (Cited on p. 120)
F. Chung (2009), *A local graph partitioning algorithm using heat kernel PageRank*, Internet Math., 6, pp. 315–330. (Cited on p. 87)
F. Chung and O. Simpson (2014), *Computing heat kernel PageRank and a local clustering algorithm*, in Combinatorial Algorithms (IWOCA 2014), Springer, pp. 110–121. (Cited on p. 87)
J.-C. Delvenne, S. N. Yaliraki, and M. Barahona (2010), *Stability of graph communities across time scales*, Proc. Natl. Acad. Sci. USA, 107, pp. 12755–12760, https://doi.org/10.1073/pnas.0903215107. (Cited on p. 65)
B. Dezso, J. Alpár, and P. Kovács (2011), *LEMON—an open source C++ graph template library*, Electron. Notes Theoret. Comput. Sci., 264, pp. 23–45. (Cited on p. 89)
E. Dinitz (1970), *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Dokl. Akad. Nauk SSSR, 11, pp. 1277–1280, https://web.archive.org/web/20190215224206/https://www.cs.bgu.ac.il/~dinitz/D70.pdf. (Cited on pp. 95, 112)
W. Dinkelbach (1967), *On nonlinear fractional programming*, Management Sci., 13, pp. 492–498. (Cited on p. 79)
D. L. Donoho and Y. Tsaig (2008), *Fast solution of ℓ_1-norm minimization problems when the solution may be sparse*, IEEE Trans. Inform. Theory, 54, pp. 4789–4812, https://doi.org/10.1109/tit.2008.929958. (Cited on p. 109)
D. Easley and K. Jo (2010), *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*, Cambridge University Press, New York. (Cited on p. 62)
D. Eckles, B. Karrer, and J. Ugander (2017), *Design and analysis of experiments in networks: Reducing bias from interference*, J. Causal Infer., 5, art. 20150021, https://doi.org/10.1515/jci-2015-0021. (Cited on p. 62)
B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani (2004), *Least angle regression*, Ann. Statist., 32, pp. 407–499. (Cited on p. 109)
B. Ehrhardt and P. J. Wolfe (2019), *Network modularity in the presence of covariates*, SIAM Rev., 61, pp. 261–276, https://doi.org/10.1137/17M1111528. (Cited on p. 62)
E. Estrada and N. Hatano (2016), *Communicability angle and the spatial efficiency of networks*, SIAM Rev., 58, pp. 692–715, https://doi.org/10.1137/141000555. (Cited on p. 62)
E. Estrada and D. J. Higham (2010), *Network properties revealed through matrix functions*, SIAM Rev., 52, pp. 696–714, https://doi.org/10.1137/090761070. (Cited on p. 62)
C. Faloutsos, K. S. McCurley, and A. Tomkins (2004), *Fast discovery of connection subgraphs*, in Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 118–127. (Cited on pp. 65, 67)
P. F. Felzenszwalb and D. P. Huttenlocher (2004), *Efficient graph-based image segmentation*, Internat. J. Comput. Vision, 59, pp. 167–181. (Cited on p. 73)
P. G. Fennell and J. P. Gleeson (2019), *Multistate dynamical processes on networks: Analysis through degree-based approximation frameworks*, SIAM Rev., 61, pp. 92–118, https://doi.org/

10.1137/16M1109345. (Cited on p. 62)
C. M. Fiduccia and R. M. Mattheyses (1982), *A linear-time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference, ACM, pp. 175–181, http://dl.acm.org/citation.cfm?id=800263.809204. (Cited on p. 86)
G. W. Flake, S. Lawrence, and C. L. Giles (2000), *Efficient identification of web communities*, in Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 150–160. (Cited on p. 89)
B. K. Fosdick, D. B. Larremore, J. Nishimura, and J. Ugander (2018), *Configuring random graph models with fixed degree sequences*, SIAM Rev., 60, pp. 315–355, https://doi.org/10.1137/16M1087175. (Cited on p. 62)
K. Fountoulakis, D. F. Gleich, and M. W. Mahoney (2017), *An optimization approach to locally-biased graph algorithms*, Proc. IEEE, 105, pp. 256–272. (Cited on pp. 74, 87, 88)
K. Fountoulakis, M. Liu, D. Gleich, and M. W. Mahoney (2019a), *Code for Experiments of the Present Paper*, https://github.com/dgleich/flowpaper-code/tree/master/figures. (Cited on pp. 131, 132, 133, 134)
K. Fountoulakis, M. Liu, D. Gleich, and M. W. Mahoney (2019b), *LocalGraphClustering API*, https://github.com/kfoynt/LocalGraphClustering. (Cited on pp. 67, 125, 129, 131)
K. Fountoulakis, F. Roosta-Khorasani, J. Shun, X. Cheng, and M. W. Mahoney (2019c), *Variational perspective on local graph clustering*, Math. Program. Ser. B, 174, pp. 553–573. (Cited on pp. 65, 87, 88, 131)
H. Frenk and S. Schaible (2009), *Fractional programming*, in Encyclopedia of Optimization, Springer, Boston, MA, pp. 1080–1091. (Cited on p. 78)
J. H. Friedman and J. J. Meulman (2004), *Clustering objects on subsets of attributes (with discussion)*, J. R. Stat. Soc. Ser. B Stat. Methodol., 66, pp. 815–849. (Cited on p. 61)
G. Gallo, M. D. Grigoriadis, and R. E. Tarjan (1989), *A fast parametric maximum flow algorithm and applications*, SIAM J. Comput., 18, pp. 30–55, https://doi.org/10.1137/0218003. (Cited on pp. 67, 81)
U. Gargi, W. Lu, V. Mirrokni, and S. Yoon (2011), *Large-scale community detection on YouTube for topic discovery and exploration*, in Proceedings of the Fifth International AAAI Conference on Weblogs and Social Media, pp. 486–489. (Cited on p. 86)
D. F. Gleich (2015), *PageRank beyond the web*, SIAM Rev., 57, pp. 321–363, https://doi.org/10.1137/140976649. (Cited on pp. 65, 67, 106)
D. F. Gleich and M. W. Mahoney (2014), *Anti-differentiating approximation algorithms: A case study with min-cuts, spectral, and flow*, in Proceedings of the 31st International Conference on Machine Learning, PMLR, pp. 1018–1025. (Cited on pp. 87, 106)
D. F. Gleich and M. W. Mahoney (2015), *Using local spectral methods to robustify graph-based learning algorithms*, in Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 359–368. (Cited on pp. 87, 88, 106, 123, 133)
D. F. Gleich and M. W. Mahoney (2016), *Mining large graphs*, in Handbook of Big Data, CRC Press, pp. 191–220, https://doi.org/10.1201/b19567-17. (Cited on pp. 65, 74)
A. V. Goldberg (1984), *Finding a Maximum Density Subgraph*, M.S. Thesis CSD-84-171, University of California at Berkeley, https://web.archive.org/web/20151129022137/http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-171.pdf. (Cited on p. 88)
A. V. Goldberg and S. Rao (1998), *Beyond the flow decomposition barrier*, J. ACM, 45, pp. 783–797, https://doi.org/10.1145/290179.290181. (Cited on pp. 95, 113)
A. V. Goldberg and R. E. Tarjan (2014), *Efficient maximum flow algorithms*, Commun. ACM, 57, pp. 82–89. (Cited on p. 89)
D. M. Greig, B. T. Porteous, and A. H. Seheult (1989), *Exact maximum a posteriori estimation for binary images*, J. R. Statist. Soc. Ser. B Methodol., 51, pp. 271–279. (Cited on p. 90)
P. Grindrod and D. J. Higham (2013), *A matrix iteration for dynamic network summaries*, SIAM Rev., 55, pp. 118–128, https://doi.org/10.1137/110855715. (Cited on p. 62)
W. Ha, K. Fountoulakis, and M. W. Mahoney (2020), *Statistical guarantees for local graph clustering*, in Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics, PMLR, 2020, pp. 2687–2697. (Cited on p. 87)
L. Hagen and A. B. Kahng (1992), *New spectral methods for ratio cut partitioning and clustering*, IEEE Trans. Comput.-Aided Des., 11, pp. 1074–1085. (Cited on p. 73)
K. M. Hall (1970), *An r-dimensional quadratic placement algorithm*, Management Sci., 17, pp. 219–229, http://www.jstor.org/stable/2629091. (Cited on p. 126)
T. J. Hansen and M. W. Mahoney (2014), *Semi-supervised eigenvectors for large-scale locally-biased learning*, J. Mach. Learn. Res., 15, pp. 3691–3734, http://dl.acm.org/citation.cfm?id=2627435.2750363. (Cited on p. 126)
M. Hein and S. Setzer (2011), *Beyond spectral clustering—tight relaxations of balanced*

graph cuts, in Advances in Neural Information Processing Systems 24, Curran Associates, pp. 2366–2374, http://papers.nips.cc/paper/4261-beyond-spectral-clustering-tight-relaxations-of-balanced-graph-cuts.pdf. (Cited on p. 86)

B. Hendrickson and R. Leland (1994a), *The Chaco User's Guide, Version* 2.0, Technical Report SAND94–2692, Sandia National Labs, Albuquerque, NM, https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/guide.pdf. (Cited on p. 86)

B. Hendrickson and R. Leland (1994b), *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM J. Sci. Comput., 16, pp. 452–469, https://doi.org/10.1137/0916028. (Cited on p. 86)

B. Hendrickson and R. W. Leland (1995), *A multi-level algorithm for partitioning graphs*, in Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, pp. 1–14. (Cited on p. 86)

A. Henzinger, A. Noe, and C. Schulz (2020), *ILP-based local search for graph partitioning*, ACM J. Exp. Algorithmics, 25, art. 1.9, https://doi.org/10.1145/3398634. (Cited on p. 86)

D. Hernando, P. Kellman, J. P. Haldar, and Z.-P. Liang (2010), *Robust water/fat separation in the presence of large field inhomogeneities using a graph cut algorithm*, Magnetic Reson. Med., 63, pp. 79–90. (Cited on p. 90)

D. S. Hochbaum (2010), *Polynomial time algorithms for ratio regions and a variant of normalized cut*, IEEE Trans. Pattern Anal. Mach. Intell., 32, pp. 889–898. (Cited on pp. 67, 73, 83)

D. S. Hochbaum (2013), *A polynomial time algorithm for Rayleigh ratio on discrete variables: Replacing spectral techniques for expander ratio, normalized cut and Cheeger constant*, Oper. Res., 61, pp. 184–198. (Cited on pp. 63, 89)

M. Jacobs, E. Merkurjev, and S. Esedoḡlu (2018), *Auction dynamics: A volume constrained MBO scheme*, J. Comput. Phys., 354, pp. 288–310, https://doi.org/10.1016/j.jcp.2017.10.036. (Cited on p. 89)

L. G. S. Jeub, P. Balachandran, M. A. Porter, P. J. Mucha, and M. W. Mahoney (2015), *Think locally, act locally: Detection of small, medium-sized, and large communities in large networks*, Phys. Rev. E, 91, art. 012821. (Cited on pp. 86, 87)

P. Jia, A. MirTabatabaei, N. E. Friedkin, and F. Bullo (2015), *Opinion dynamics and the evolution of social power in influence networks*, SIAM Rev., 57, pp. 367–397, https://doi.org/10.1137/130913250. (Cited on p. 62)

T. Joachims (2003), *Transductive learning via spectral graph partitioning*, in Proceedings of the 20th International Conference on Machine Learning (ICML-03), AAAI Press, pp. 290–297. (Cited on p. 87)

A. Jung, A. O. Hero, A. C. Mara, S. Jahromi, A. Heimowitz, and Y. C. Eldar (2019), *Semi-supervised learning in network-structured data via total variation minimization*, IEEE Trans. Signal Process., 67, pp. 6256–6269, https://doi.org/10.1109/tsp.2019.2953593. (Cited on p. 89)

A. Jung and Y. SarcheshmehPour (2021), *Local graph clustering with network lasso*, IEEE Signal Process. Lett., 28, pp. 106–110, https://doi.org/10.1109/lsp.2020.3045832. (Cited on p. 89)

G. Karypis and V. Kumar (1998), *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20, pp. 359–392, https://doi.org/10.1137/S1064827595287997. (Cited on p. 86)

G. Karypis and V. Kumar (1999), *Parallel multilevel series k-way partitioning scheme for irregular graphs*, SIAM Rev., 41, pp. 278–300, https://doi.org/10.1137/S0036144598334138. (Cited on p. 86)

R. Khandekar, S. Rao, and U. Vazirani (2009), *Graph partitioning using single commodity flows*, J. ACM, 56, art. 19, https://doi.org/10.1145/1538902.1538903. (Cited on p. 88)

J. Kleinberg (2002), *An impossibility theorem for clustering*, in Proceedings of the 15th International Conference on Neural Information Processing Systems, ACM, pp. 463–470, http://dl.acm.org/citation.cfm?id=2968618.2968676. (Cited on p. 62)

J. Kleinberg and E. Tardos (2005), *Algorithm Design*, Addison-Wesley Longman. (Cited on p. 88)

K. Kloster and D. F. Gleich (2014), *Heat kernel based community detection*, in Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1386–1395. (Cited on p. 87)

I. M. Kloumann and J. M. Kleinberg (2014), *Community membership identification from small seed sets*, in Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1366–1375. (Cited on pp. 65, 67, 87)

N. Knight, E. Carson, and J. Demmel (2014), *Exploiting data sparsity in parallel matrix powers computations*, in Parallel Processing and Applied Mathematics, Springer, Berlin, pp. 15–25. (Cited on p. 86)

V. Kolmogorov and R. Zabih (2004), *What energy functions can be minimized via graph cuts?*,

IEEE Trans. Pattern Anal. Mach. Intell., 26, pp. 147–159. (Cited on pp. 90, 130)

A. Lancichinetti, S. Fortunato, and J. Kertész (2009), *Detecting the overlapping and hierarchical community structure in complex networks*, New J. Phys., 11, art. 033015, https://doi.org/10.1088/1367-2630/11/3/033015. (Cited on p. 87)

K. Lang (2005), *Fixing two weaknesses of the spectral method*, in Advances in Neural Information Processing Systems 18 (NIPS2005), pp. 715–722, http://books.nips.cc/papers/files/nips18/NIPS2005_0529.pdf. (Cited on pp. 65, 87, 126)

K. Lang and S. Rao (2004), *A flow-based method for improving the expansion or conductance of graph cuts*, in IPCO 2004: Integer Programming and Combinatorial Optimization, Springer, pp. 325–337. (Cited on pp. 63, 66, 67, 68, 83, 84, 86, 95, 96, 130)

K. J. Lang, M. W. Mahoney, and L. Orecchia (2009), *Empirical evaluation of graph partitioning using spectral embeddings and flow*, in Proceedings of the 8th International Symposium on Experimental Algorithms, Springer, pp. 197–208. (Cited on p. 88)

D. Lawlor, T. Budavári, and M. W. Mahoney (2016a), *Mapping the similarities of spectra: Global and locally-biased approaches to SDSS galaxies*, Astrophys. J., 833, art. 26. (Cited on pp. 74, 118, 126, 132)

D. Lawlor, T. Budavári, and M. W. Mahoney (2016b), *Mapping the Similarities of Spectra: Global and Locally-Biased Approaches to SDSS Galaxy Data*, preprint, https://arxiv.org/abs/1609.03932. (Cited on pp. 74, 118, 126)

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner (1998), *Gradient-based learning applied to document recognition*, Proc. IEEE, 86, pp. 2278–2324, https://doi.org/10.1109/5.726791. (Cited on p. 123)

Y. T. Lee, S. Rao, and N. Srivastava (2013), *A new approach to computing maximum flows using electrical flows*, in Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, pp. 755–764. (Cited on p. 89)

Y. T. Lee and A. Sidford (2013), *Path Finding* II*: An* $\tilde{\mathcal{O}}(m\sqrt{n})$ *Algorithm for the Minimum Cost Flow Problem*, preprint, https://arxiv.org/abs/1312.6713. (Cited on p. 89)

T. Leighton and S. Rao (1988), *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms*, in 29th Annual Symposium on Foundations of Computer Science, IEEE, pp. 422–431. (Cited on p. 88)

T. Leighton and S. Rao (1999), *Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms*, J. ACM, 46, pp. 787–832. (Cited on p. 88)

J. Leskovec and A. Krevl (2014), *SNAP Datasets: Stanford Large Network Dataset Collection*, http://snap.stanford.edu/data. (Cited on p. 125)

J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney (2008), *Statistical properties of community structure in large social and information networks*, in WWW '08: Proceedings of the 17th International Conference on World Wide Web, ACM, pp. 695–704. (Cited on pp. 86, 87, 125)

J. Leskovec, K. Lang, and M. Mahoney (2010), *Empirical comparison of algorithms for network community detection*, in WWW '10: Proceedings of the 19th International Conference on World Wide Web, ACM, pp. 631–640. (Cited on pp. 86, 87, 125)

J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney (2009), *Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters*, Internet Math., 6, pp. 29–123. (Cited on pp. 62, 86, 87, 125)

P. Li and O. Milenkovic (2017), *Inhomogeneous hypergraph clustering with applications*, in Advances in Neural Information Processing Systems 30, Curran Associates, pp. 2308–2318, http://papers.nips.cc/paper/6825-inhomogeneous-hypergraph-clustering-with-applications.pdf. (Cited on p. 131)

Y. Li, K. He, D. Bindel, and J. E. Hopcroft (2015), *Uncovering the small community structure in large networks: A local spectral approach*, in Proceedings of the 24th International Conference on World Wide Web, pp. 658–668. (Cited on p. 87)

L. Liberti, C. Lavor, N. Maculan, and A. Mucherino (2014), *Euclidean distance geometry and applications*, SIAM Rev., 56, pp. 3–69, https://doi.org/10.1137/120875909. (Cited on p. 62)

W. Liu and S.-F. Chang (2009), *Robust multi-class transductive learning with graphs*, in IEEE Conference on Computer Vision and Pattern Recognition, pp. 381–388. (Cited on p. 87)

Y. P. Liu and A. Sidford (2020), *Faster energy maximization for faster maximum flow*, in Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020), ACM, New York, pp. 803–814, https://doi.org/10.1145/3357713.3384247. (Cited on p. 89)

P. Y. Lum, G. Singh, A. Lehman, T. Ishkanov, M. Vejdemo-Johansson, M. Alagappan, J. Carlsson, and G. Carlsson (2013), *Extracting insights from the shape of complex data using topology*, Sci. Rep., 3, art. 1236. (Cited on p. 124)

Z. Ma, X. Wu, Q. Song, Y. Luo, Y. Wang, and J. Zhou (2018), *Automated nasopharyngeal carcinoma segmentation in magnetic resonance images by combination of convolutional neural*

networks and graph cut, Experimental Therapeutic Med., 16, pp. 2511–2521, https://doi.org/10.3892/etm.2018.6478. (Cited on p. 90)

M. W. Mahoney, L. Orecchia, and N. K. Vishnoi (2012), *A local spectral method for graphs: With applications to improving graph partitions and exploring data graphs locally*, J. Mach. Learn. Res., 13, pp. 2339–2365. (Cited on p. 74)

R. Marlet (2017), *Graph Cuts and Application to Disparity Map Estmation*, https://web.archive.org/web/20221214152458/https://imagine.enpc.fr/~marletr/enseignement/mva/mva-2017/mva-2017-graphcuts.pdf. (Cited on p. 90)

A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee (2007), *Measurement and analysis of online social networks*, in Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, pp. 29–42. (Cited on p. 125)

A. Moffat (1969), *A theoretical investigation of focal stellar images in the photographic emulsion and application to photographic photometry*, Astron. Astrophys., 3, pp. 455–461. (Cited on p. 132)

G. Namata, B. London, L. Getoor, and B. Huang (2012), *Query-driven active surveying for collective classification*, in 10th International Workshop on Mining and Learning with Graphs, p. 8. (Cited on p. 123)

M. Newman (2010), *Networks: An Introduction*, Oxford University Press, New York. (Cited on p. 62)

M. E. J. Newman (2006), *Modularity and community structure in networks*, Proc. Natl. Acad. Sci. USA, 103, pp. 8577–8582. (Cited on pp. 62, 87)

A. Y. Ng, M. I. Jordan, and Y. Weiss (2001), *On spectral clustering: Analysis and an algorithm*, in Advances in Neural Information Processing Systems 14, MIT Press, pp. 849–856, http://papers.nips.cc/paper/2092-on-spectral-clustering-analysis-and-an-algorithm.pdf. (Cited on p. 126)

L. Orecchia, L. J. Schulman, U. V. Vazirani, and N. K. Vishnoi (2012), *On partitioning graphs via single commodity flows*, in Proceedings of the 44th Annual ACM Symposium on Theory of Computing, pp. 1141–1160. (Cited on p. 88)

L. Orecchia and Z. A. Zhu (2014), *Flow-based algorithms for local graph clustering*, in Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1267–1286. (Cited on pp. 63, 66, 68, 76, 84, 106, 107, 109, 111, 112, 113)

J. B. Orlin (2013), *Max flows in $O(nm)$ time, or better*, in Proceedings of the 45th Annual ACM Symposium on Theory of Computing, pp. 765–774. (Cited on p. 89)

B. Osting, J. Darbon, and S. Osher (2013), *Statistical ranking using the ℓ^1-norm on graphs*, Inverse Probl. Imaging, 7, pp. 907–926, https://doi.org/10.3934/ipi.2013.7.907. (Cited on p. 89)

P. Pagel, S. Kovac, M. Oesterheld, B. Brauner, I. Dunger-Kaltenbach, G. Frishman, C. Montrone, P. Mark, V. Stümpflen, H.-W. Mewes, A. Ruepp, and D. Frishman (2004), *The MIPS mammalian protein–protein interaction database*, Bioinform., 21, pp. 832–834. (Cited on p. 125)

G. Palla, I. Derényi, I. Farkas, and T. Vicsek (2005), *Uncovering the overlapping community structure of complex networks in nature and society*, Nature, 435, pp. 814–818. (Cited on p. 87)

C. Papadimitriou and K. Steiglitz (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall. (Cited on p. 92)

L. Peel (2017), *Graph-based semi-supervised learning for relational networks*, in Proceedings of the 2017 SIAM International Conference on Data Mining, pp. 435–443, https://doi.org/10.1137/1.9781611974973.49. (Cited on pp. 88, 123)

L. Peel, D. B. Larremore, and A. Clauset (2017), *The ground truth about metadata and community detection in networks*, Sci. Adv., 3, art. e1602548, https://doi.org/10.1126/sciadv.1602548. (Cited on p. 88)

T. P. Peixoto (2014), *The Graph-Tool Python Library*, figshare, http://figshare.com/articles/graph_tool/1164194. (Cited on p. 122)

F. Pellegrini and J. Roman (1996), *SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, in International Conference on High-Performance Computing and Networking, Springer, pp. 493–498. (Cited on p. 86)

A. Pothen, H. D. Simon, and K.-P. Liou (1990), *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11, pp. 430–452, https://doi.org/10.1137/0611030. (Cited on p. 86)

V. Red, E. D. Kelsic, P. J. Mucha, and M. A. Porter (2011), *Comparing community structure to characteristics in online collegiate social networks*, SIAM Rev., 53, pp. 526–543, https://doi.org/10.1137/080734315. (Cited on pp. 62, 120)

P. Rombach, M. A. Porter, J. H. Fowler, and P. J. Mucha (2017), *Core-periphery structure in networks (revisited)*, SIAM Rev., 59, pp. 619–646, https://doi.org/10.1137/17M1130046.

(Cited on p. 62)
L. I. Rudin, S. Osher, and E. Fatemi (1992), *Nonlinear total variation based noise removal algorithms*, Phys. D, 60, pp. 259–268. (Cited on p. 89)
P. Sanders and C. Schulz (2011), *Engineering multilevel graph partitioning algorithms*, in Proceedings of the 19th European Conference on Algorithms (ESA'11), Springer-Verlag, Berlin, Heidelberg, pp. 469–480. (Cited on p. 86)
F. Shahrokhi (1990), *The maximum concurrent flow problem*, J. ACM, 37, pp. 318–334. (Cited on p. 88)
E. Sharon, M. Galun, D. Sharon, R. Basri, and A. Brandt (2006), *Hierarchy and adaptivity in segmenting visual scenes*, Nature, 442 (7104), pp. 810–813. (Cited on p. 73)
G. Shi, C. Altafini, and J. S. Baras (2019), *Dynamics over signed networks*, SIAM Rev., 61, pp. 229–257, https://doi.org/10.1137/17M1134172. (Cited on p. 62)
J. Shi and J. Malik (2000), *Normalized cuts and image segmentation*, IEEE Trans. Pattern Anal. Mach. Intell., 22, pp. 888–905. (Cited on pp. 73, 135)
P. Shi, K. He, D. Bindel, and J. Hopcroft (2017), *Local Lanczos spectral approximation for community detection*, in Proceedings of ECML-PKDD, Springer, pp. 651–667. (Cited on p. 87)
J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney (2016), *Parallel local graph clustering*, Proc. VLDB Endowment, 9, pp. 1041–1052. (Cited on pp. 74, 87)
H. D. Simon (1991), *Partitioning of unstructured problems for parallel processing*, Comput. Syst. Engrg., 2, pp. 135–148. (Cited on p. 86)
D. D. Sleator and R. E. Tarjan (1983), *A data structure for dynamic trees*, J. Comput. Syst. Sci., 3, pp. 362–391. (Cited on p. 112)
A. J. Soper, C. Walshaw, and M. Cross (2004), *A combined evolutionary search and multilevel optimisation approach to graph-partitioning*, J. Global Optim., 29, pp. 225–241. (Cited on p. 86)
D. A. Spielman and S. H. Teng (2013), *A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning*, SIAM J. Comput., 42, pp. 1–26, https://doi.org/10.1137/080744888. (Cited on p. 87)
G. Strang (1983), *Maximal flow through a domain*, Math. Program., 26, pp. 123–143. (Cited on p. 89)
G. Strang (2010), *Maximum flows and minimum cuts in the plane*, J. Global Optim., 47, pp. 527–535. (Cited on p. 89)
R. Tibshirani (1996), *Regression shrinkage and selection via the lasso*, J. R. Statist. Soc. Ser. B Methodol., 58, pp. 267–288, http://www.jstor.org/stable/2346178. (Cited on p. 109)
H. Tong, C. Faloutsos, and J.-Y. Pan (2006), *Fast random walk with restart and its applications*, in ICDM '06: Proceedings of the Sixth International Conference on Data Mining, IEEE, pp. 613–622. (Cited on pp. 65, 67)
A. L. Traud, P. J. Mucha, and M. A. Porter (2012), *Social structure of Facebook networks*, Phys. A, 391, pp. 4165–4180. (Cited on p. 120)
L. Trevisan (2011), *Combinatorial Optimization: Exact and Approximate Algorithms*, lecture notes for CS261 at Stanford University, https://web.archive.org/web/20200501020454/http://theory.stanford.edu/~trevisan/books/cs261.pdf. (Cited on p. 92)
C. E. Tsourakakis, J. Pachocki, and M. Mitzenmacher (2017), *Scalable motif-aware graph clustering*, in Proceedings of the 26th International Conference on World Wide Web, pp. 1451–1460. (Cited on p. 87)
M. Ullah, A. Iltaf, Q. Hou, F. Ali, and C. Liu (2018), *A foreground extraction approach using convolutional neural network with graph cut*, in 2018 IEEE 3rd International Conference on Image, Vision and Computing (ICIVC), pp. 40–44. (Cited on p. 90)
S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors (2014), *scikit-image: Image processing in Python*, PeerJ, 2, art. e453. (Cited on p. 131)
L. N. Veldt, D. F. Gleich, and M. W. Mahoney (2016), *A simple and strongly-local flow-based method for cut improvement*, in International Conference on Machine Learning, JMLR, pp. 1938–1947, http://jmlr.org/proceedings/papers/v48/veldt16.html. (Cited on pp. 65, 75, 84, 106, 107, 109, 111, 113)
N. Veldt (2019), *PushRelabel Local Flow Algorithms*, Github software, https://github.com/nveldt/PushRelabelMaxFlow. (Cited on pp. 129, 132)
N. Veldt, A. R. Benson, and J. Kleinberg (2020), *Minimizing localized ratio cut objectives in hypergraphs*, in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20), ACM, New York, pp. 1708–1718, https://doi.org/10.1145/3394486.3403222. (Cited on pp. 70, 130)
N. Veldt, A. R. Benson, and J. Kleinberg (2022), *Hypergraph cuts with general splitting func-*

tions, SIAM Rev., 64, pp. 650–685, https://doi.org/10.1137/20M1321048. (Cited on p. 70)

N. Veldt, C. Klymko, and D. F. Gleich (2019), *Flow-based local graph clustering with better seed set inclusion*, in Proceedings of the 2019 SIAM International Conference on Data Mining, pp. 378–386. (Cited on pp. 63, 77, 85)

N. Veldt, A. Wirth, and D. F. Gleich (2019), *Learning resolution parameters for graph clustering*, in The World Wide Web Conference, ACM, pp. 1909–1919. (Cited on pp. 63, 130)

U. von Luxburg (2007), *A tutorial on spectral clustering*, Statist. Comput., 17, pp. 395–416. (Cited on p. 73)

U. von Luxburg, R. C. Williamson, and I. Guyon (2012), *Clustering: Science or art?*, in Proceedings of ICML Workshop on Unsupervised and Transfer Learning, pp. 65–79, http://proceedings.mlr.press/v27/luxburg12a.html. (Cited on p. 61)

C. Walshaw and M. Cross (2000), *Mesh partitioning: A multilevel balancing and refinement algorithm*, SIAM J. Sci. Comput., 22, pp. 63–80, https://doi.org/10.1137/S1064827598337373. (Cited on p. 86)

C. Walshaw and M. Cross (2007), *Jostle: Parallel multilevel graph-partitioning software—an overview*, in Mesh Partitioning Techniques and Domain Decomposition Techniques, Civil-Comp Ltd., pp. 27–58. (Cited on p. 86)

J. J. Whang, D. F. Gleich, and I. S. Dhillon (2016), *Overlapping community detection using neighborhood-inflated seed expansion*, Trans. Knowledge Data Engrg., 28, pp. 1272–1284, http://arxiv.org/abs/1503.07439. (Cited on p. 87)

Wikipedia (2021), *Eileen Collins*, https://en.wikipedia.org/wiki/Eileen_Collins [accessed 16 September 2021]. (Cited on pp. 67, 68)

D. P. Williamson (2019), *Network Flow Algorithms*, Cambridge University Press, https://doi.org/10.1017/9781316888568. (Cited on pp. 111, 112)

J. Xie, S. Kelley, and B. K. Szymanski (2013), *Overlapping community detection in networks: The state-of-the-art and comparative study*, ACM Comput. Surv., 45, art. 43. (Cited on p. 87)

H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich (2017), *Local higher-order graph clustering*, in Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 555–564. (Cited on p. 87)

J. Yuan, E. Bae, and X.-C. Tai (2010), *A study on continuous max-flow and min-cut approaches*, in 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 2217–2224. (Cited on p. 89)

D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf (2004), *Learning with local and global consistency*, in Annual Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference, pp. 321–328. (Cited on pp. 65, 67, 87)

X. Zhu, Z. Ghahramani, and J. D. Lafferty (2003), *Semi-supervised learning using Gaussian fields and harmonic functions*, in Proceedings of the 20th International Conference on Machine Learning (ICML-03), AAAI Press, pp. 912–919. (Cited on pp. 65, 67, 87)

Z. A. Zhu, S. Lattanzi, and V. Mirrokni (2013), *A local algorithm for finding well-connected clusters*, in Proceedings of the 30th International Conference on Machine Learning, JMLR, pp. 396–404. (Cited on pp. 87, 88)