Scaling Overlapping Clustering

Kyle Kloster
Purdue University
kyle.kloster@gmail.com

Merrielle Spain and Stephen Kelley
Lincoln Laboratory
Massachusetts Institute of Technology
{merrielle.spain,stephen.kelley}@ll.mit.edu

ABSTRACT

Identifying communities plays a central role in understanding the structure of large networks. As practitioners analyze progressively larger networks, it becomes increasingly important to understand the computational complexity of candidate algorithms. We examine the complexity of the link clustering algorithm (Ahn et al., 2010) for overlapping community detection. We provide new, tight bounds for the original implementation and propose modifications to reduce algorithmic complexity. These new bounds are a function of the number of wedges in the graph. Additionally, we demonstrate that for several community detection algorithms, wedges predict runtime better than commonly cited graph features. We conclude by proposing a method to reduce the wedges in a graph by removing high-degree vertices from the network, identifying communities with an optimized version of link clustering, and heuristically matching communities with the removed vertices in post-processing. We empirically demonstrate a large reduction in processing time on several common datasets.

1. INTRODUCTION

The existence, discovery, and analysis of community structure has become a vibrant research field in network science. The importance of these structures has been demonstrated in areas ranging from biology to sociology [3, 11]. Research literature proposing new community detection algorithms primarily focuses on community quality measurement. The computational properties of popular algorithms, however, are less well studied and focus on identifying upper bounds for algorithm complexity in terms of simple statistics such as the numbers of edges, numbers of vertices, max or average degree, and graph diameter. The effect of specific, local structures on the computational complexity of overlapping community detection algorithms remains largely unknown. As observed networks continue to grow in size, measuring the effect of these structures becomes important in guiding practitioners to identify efficient methods for solving a problem of interest. Moreover, a more precise understanding of how graph structure affects the runtime of different methods will better enable researchers to design new algorithms and modify current algorithms to be scalable.

This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

We examine the runtime of the popular link clustering [1] algorithm for overlapping community detection. We prove a tighter bound on computational complexity that is based on *wedges* (paths of length 2) in a network. We modify the original algorithm to achieve lower computational complexity, under certain conditions. We demonstrate this empirically on real networks generated from email logs of a medium sized enterprise.

Aside from an optimization of the link clustering algorithm and wedge-based bounds on complexity, we examine the runtime of several comparable community detection algorithms in practice. On real data, we demonstrate that often wedges provide a better predictor of runtime than other common network features. Finally, we propose a framework for reducing wedge count by removing high-degree vertices from the analysis and adding them back via a post-processing heuristic. We empirically demonstrate a significant reduction in runtime for our optimized link clustering algorithm on several common datasets.

We organize the paper as follows. Section 2 overviews the link clustering algorithm before analyzing the complexity of each component. We then prove the exact link clustering complexity and demonstrate that often cited bounds are missing an important factor. We harness these insights to identify bottlenecks and modify the algorithm to attain identical clustering results while greatly reducing computational complexity. Section 3 investigates how clustering algorithms scale as a function of graph properties. Section 4 presents pre- and post-processing steps to temporarily remove high-degree vertices. This produces comparable clustering quality to the original algorithm, while also showing a drastic reduction in computation in experiments. We conclude in Section 5.

2. OPTIMIZED LINK CLUSTERING

We begin by outlining the link clustering algorithm of Ahn, Bagrow, and Lehmann [1] for identifying overlapping clusters in networks. Then we analyze the computational complexity of each component of the algorithm, describe modifications for avoiding redundant computations, and further reduce the complexity by showing that a particular non-standard sorting subroutine is guaranteed to work more efficiently than traditional comparison-based sorting algorithms.

2.1 Preliminaries

Let G be an undirected, unweighted graph with adjacency matrix A. Denote by n the number of nodes in G (i.e., rows/columns in A), and by m the number of edges in G (half the number of nonzeros in A). Let $d(k)$ be the degree of node k. The *inclusive neighborhood* of a node j, which we denote by $N_+(j)$, consists of the neighborhood of node j together with the node j itself.

Following the authors' original notation, we label an edge with its endpoints so that an edge connecting nodes j and k is denoted e_{jk}. We call a pair of edges that share an endpoint a *wedge*; this is also called a pair of adjacent edges, or a path of length two. Given a pair of adjacent edges, e_{jk} and e_{kl}, we refer to the shared node k as the *keystone* and the non-shared endpoints, j and l, as the *imposts* with respect to this wedge.

Link clustering measures edge similarity as the Jaccard Index on the inclusive neighborhoods of wedge imposts. The similarity of two edges, e_{jk} and e_{kl}, is defined as follows:

$$S(e_{jk}, e_{kl}) = \frac{|N_+(j) \cap N_+(l)|}{|N_+(j) \cup N_+(l)|}.$$

In Section 2.4.1 we describe how we compute this efficiently.

The algorithm produces a dendrogram of edge merges via single-link agglomerative clustering. To identify the "best" level of the dendrogram, Ahn et al. define *partition density.* Intuitively, partition density measures the average link density of a partitioning. Given a subgraph S with $n > 2$ nodes and m edges, the *link density* of S is defined as

$$\mathrm{LD}\,(S) := \frac{m - (n-1)}{\binom{n}{2} - (n-1)}, \tag{1}$$

where 1 indicates full density (a clique), 0 indicates a minimally connected set of nodes (a tree), and $-\frac{2}{n-2}$ indicates isolated nodes. The partition density of an edge partition C with c blocks is defined as

$$PD := \frac{1}{m} \sum_{k=1}^{c} m_k \cdot \mathrm{LD}\,(C_k)\,, \tag{2}$$

where m_k is the number of edges in partition block C_k. This value represents the average link density, weighted by the number of edges within a block.

2.2 Link clustering complexity

Link clustering builds communities by iteratively joining similar edges. Here we describe the algorithm's four steps and determine their computational complexity.

1. **Similarity**: Compute similarity scores for all pairs of adjacent edges.
2. **Sort**: Sort the similarity scores in decreasing order.
3. **Merge**: Iteratively merge communities bridged by edges with the highest similarity score.
4. **Partition density**: Compute the partition density at each merge level. Output the clustering determined by the level of maximum partition density.

Lower bound We show a lower bound of $w \log w$, where w is the number of wedges in the graph. The algorithm computes exactly w similarity scores (one score for each pair of adjacent edges, i.e., one for each wedge), so w lower-bounds the work in Step 1. Sorting these w scores requires an additional $\Theta(w \log w)$ work in Step 2, assuing any comparison-based sorting algorithm is use. The merge step begins with m clusters (each of the m edges begins as its own cluster), merges two adjacent clusters at each step, and terminates after merging all clusters into connected components. Thus, merging involves exactly $m-c$ steps, where m is the number of edges in the graph and c is the number of connected components in the graph. Note that each of these $m-c$ steps might involve additional work, but here we are interested in only a lower bound. The trick is to check and update cluster membership without directly updating every edge's cluster label at each step. Correspondingly, computing the partition density at each merge level requires at least $m-c$ work. Outputting the clustering involves reporting the cluster labels of each of the m edges, and hence Step 4 requires at least m work. Out of these (admittedly simple) lower bounds, those for computing (w) and sorting $(w \log w)$ similarity scores are the largest; thus, link clustering requires, at the very least, $w \log w$ work. (We remark that we have not shown here that $w \log w$ is an *upper* bound – for now we are interested in a simple lower bound.)

Upper bound We show that $w \log w$ is also an upper bound for the work performed by link clustering, proving that the problem complexity is $\Theta(w \log w)$.

2.3 Original implementation

2.3.1 Similarity

The implementation of link clustering provided by its authors [1] iterates over the graph, treating each node as a keystone. For each keystone k, the algorithm considers each unordered pair of k's neighbors to be imposts, and retrieves the imposts' neighborhoods. Retrieving the neighborhood of node j requires $d(j)$ work. The original implementation retrieves a node's neighborhood every time it is an impost. For a fixed keystone k and neighbor j, the node j is an impost $d(k)-1$ times. The neighborhood of node j is retrieved that many times for node k alone. This pairing results in $d(j)d(k-1)$ work. Furthermore, node j serves as an impost for each of its (keystone) neighbors: for each neighbor, j_t, of node j, the work performed is

$$\sum_{t=1}^{d(j)} d(j)(d(j_t)-1) = d(j) \sum_{t=1}^{d(j)} (d(j_t)-1).$$

Finally, assuming every node's degree is $d = d_{\max}$, this gives a straightforward runtime bound for the original implementation of

$$O\left(\sum_{j=1}^{n} \left[d(j) \sum_{t=1}^{d(j)} d(j_t) \right] \right) = O(nd^3).$$

2.3.2 Sort

Sorting a set of n objects based on comparison, for example with merge sort, requires $\Theta(n \log n)$ work [8]. Here, the objects are similarity scores on wedges, so sorting requires $\Theta(w \log w)$ work. This component of the work appears to be omitted in other runtime bounds [12].

2.4 Improvements

2.4.1 Similarity

To compute the similarity scores between all pairs of adjacent edges, $\sum_{k=1}^{n} \binom{d(k)}{2} = O(\sum_{k=1}^{n} d(k)^2)$ pairs must be examined. The number of impost-impost edge pairs at a keystone k is the number of unordered pairs of edges attached to it, which is $\binom{d(k)}{2}$. While this is the *number* of computed similarity scores, the number of operations depends on implementation.

The dominant subroutine of this step retrieves the neighborhoods of two imposts and computes the intersection and union of those neighborhoods. The naïve approach performs redundant operations: if two nodes share many neighbors, then the algorithm repeatedly retrieves their neighborhoods and computes similarity.

We avoid redundant neighborhood retrievals during the similarity computations as follows. Computing the similarity score of two adjacent edges with imposts j and k requires computing the value $|N_+(j) \cap N_+(k)|$. This equals the number of neighbors shared by j and k, plus 2 if an edge connects j and k. Conveniently, $A^2(j,k)$ provides the number of neighbors shared by nodes j and k, and $A(j,k)$ indicates whether an edge connects j and k. Thus, $|N_+(j) \cap N_+(k)| = (A^2 + 2 \cdot A)_{j,k}$. Since these scores are symmetric, we compute the upper triangular piece of the matrix $(A^2 + 2 \cdot A)$. Observe that computing the upper triangular portion of A^2 can be accomplished in $O(w)$ work, as it consists of summing over all paths of length two (wedges) in the graph:

$$A^2(:,j) \quad = \quad A \cdot A(:,j) \quad = \quad \sum_{t \in N(j)} A(:,t)$$

which requires $\sum_{t \in N(j)} d(t)$ work. Repeating this for all n nodes, requires exactly $\sum_{j=1}^{n} \left(\sum_{t \in N(j)} d(t) \right)$ work. The degree $d(t)$ appears in this summation once for each neighbor j. As each term $d(t)$ appears exactly $d(t)$ times, resulting in $\sum_{t=1}^{n} d(t)^2 = \Theta(w)$ work. Adding $2 \cdot A$ requires at most work equivalent to the number of edges m (nonzeros in A). This value is dominated by w in a connected graph, so the total work for Step 1 is $\Theta(w)$.

Note that this work, $\Theta(w)$, is upper-bounded by $O(nd_{\max}^2)$ (simply upper-bound $d(k)^2$ with $d_{\max}^2$), confirming the runtime upper bound given in [12]. Alternatively, we could upper-bound $d(k)^2 \leq d(k) d_{\max}$ to obtain $d_{\max} \cdot m$ as a sharper upper bound.

2.4.2 Sort

Knowing additional structure about the scores can enable faster sorting. For example, in a complete graph, all wedges have the same score, and so sorting requires no work. For graphs with maximum degree less than some threshold, the similarity scores can be sorted using a modified bucket sort in work that scales *linearly* in w, the number of scores. This enables an algorithm with the optimal runtime of $\Theta(w)$ on power law graphs.

We use a so-called "max shelf" sorting routine [7], implemented to sort values taken from $(0,1]$. Intuitively, the max shelf procedure is a bucket sort. Max shelf maps a value from the interval $(0,1]$ to an integer, then uses that integer as an index to store the value in an array. More specifically, for any value $x \in (0,1]$, compute the ceiling of its reciprocal, i.e. $x \mapsto \lceil 1/x \rceil$. This constant-time operation will map any similarity score to an integer between 1 and $\lceil 1/\text{minimum score} \rceil$. Furthermore, this operation puts scores in the array in sorted order, because two similarity scores x and y are ordered as $x \leq y$ if and only if the mapping orders them $\lceil 1/y \rceil \leq \lceil 1/x \rceil$. Once the similarity scores have all been mapped into the array, they can then be read out of the array in sorted order.

This application of bucket sort requires a lower bound on the set of inputs (a parameter we refer to as `minval`) to guarantee that the array is large enough to handle the index $\lceil 1/x \rceil$ for each score x. To guarantee a completely accurate sort, max shelf also requires an upper bound on the reciprocal of the smallest possible distance between distinct inputs (a parameter we refer to as `gap max`). That is, for scores s_j for $j = 1:n$,

$$\texttt{gap max} = \max_{s_j \neq s_i} \left(\frac{1}{|s_i - s_j|} \right).$$

By scaling the set of scores we can guarantee the shelf mapping $x \mapsto \lceil 1/x \rceil$ will map distinct scores to distinct integers, and hence provide an accurate sort. The memory required for the shelf sort procedure is bounded above by $O(n + \texttt{gap max} \cdot (1.0 - \texttt{minval}))$. If the scores can be arbitrarily close (causing an arbitrarily large value for `gap max`), this can cause memory problems. However, when scores are guaranteed to be spaced out, this approach yields a linear time sorting routine, offering a $\log n$ speedup over traditional sorting methods. In the context of Jaccard similarity scores, `gap max` is bounded above by $4d_{\max}^2$, guaranteeing a linear-time sort whenever the max degree is not too large.

3. EMPIRICAL RESULTS

Here, we investigate how the runtimes of several popular clustering methods scale with a set of graph properties: number of edges, maximum degree, number of nodes, number of triangles, and number of wedges. We consider unweighted, undirected networks and define a wedge as a pair of adjacent edges. The number of wedges in a graph is $w = \sum_{k=1}^{n} \binom{d(k)}{2}$, where $d(k)$ is the degree of node k.

3.1 Algorithms

We consider the following community detection algorithms:

Infomap detects communities by compressing the description of information flows on networks [10]. It reduces the problem to solving a coding theory problem.

BigCLAM observes that overlaps between communities are densely connected [13]. It estimates non-negative latent membership strength of each node to each community.

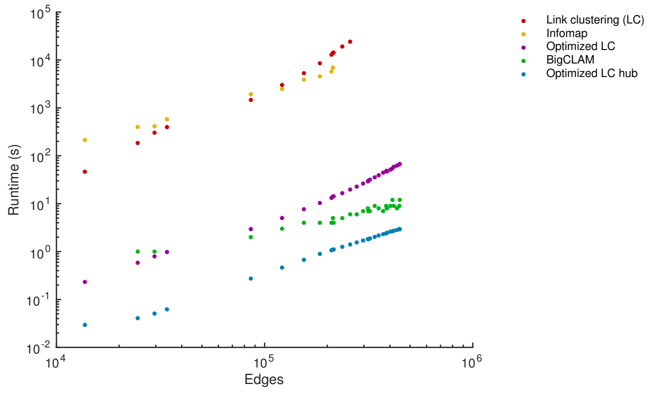

Figure 1: Runtime scales roughly as a function of edges. Our optimization (Optimized LC) greatly outperforms the original implementation (Link clustering). Infomap and BigCLAM are included for comparison. Section 4 adds hub removal to our algorithm (Optimized LC Hub).

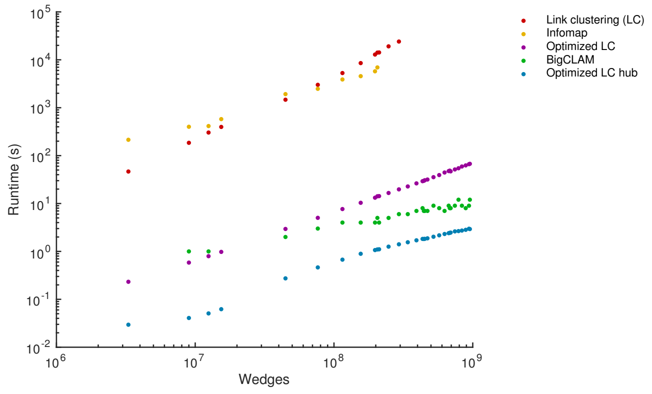

Figure 2: Runtime scales more precisely as a function of wedges.

Link clustering clusters edges instead of nodes, and allows a node to belong to all the communities to which its edges belong [1].

Optimized LC is an implementation of the algorithm proposed in this paper. Optimizations include a reduction in the number of similarity computations and improved sorting efficiency.

3.2 Data

We apply the community detection algorithms to increasing portions of a month of email data from a medium-sized enterprise (3,800 nodes and 45,000 edges). Figures and results are generated by examining the running time on a graph built from $\{1, 2, 3, \ldots\ 30\}$ days worth of data. By constructing networks in this manner we produce a sequence of increasingly dense graphs, allowing us to examine the effects of each feature on runtime.

3.3 Results

Figures 1 and 2 show that the runtimes scale loosely with the number of edges in the graph, and more tightly with the number of wedges. We observe slower runtimes for the original link clustering implementation, making it comparable with Infomap. Our improvements described in Section 2.4 make the runtimes comparable with BigCLAM. Note that BigCLAM parallelizes a matrix multiplication step to achieve some speedup, whereas the other implementations do not harness parallelization.

While qualitatively the data make cleaner lines in Figure 2 than Figure 1, we desire a quantitative comparison of how well graph features predict runtime. We choose a power law model with removed intercepts. For each algorithm, we scale the feature value and runtime data to [0, 1]. Then we fit a model of runtime as a power (α) of feature value. We measure the sum of squared error between the observed and predicted runtimes. This describes how well a model fits the data.

Table 1 shows the sum of squared error for the prediction of each algorithm's runtime given a graph feature. In three of four cases, wedges predict runtime best. For the runtimes of the original link clustering implementation, nodes predict runtime best, with wedges a close second. The implementation makes redundant computations when two nodes share multiple neighbors; we include our optimized link clustering as an example without this redundancy.

We test for statistical differences between the runtime prediction squared error scores with Friedman's test [5, 6]. We found statistically significant differences between features for each method (Link clustering $p = 0.0002$, Optimized LC $p < 0.01$, Infomap $p < 0.01$, BigCLAM $p = 0.003$). The best powers of number of wedges were as follows: Link clustering 1.54, Optimized LC 1.05, Infomap 0.66, and BigCLAM 1.07.

4. REMOVING AND REPLACING HUBS

Hubs, or nodes of high degree, can ruin the efficiency of many network analysis algorithms. A node of degree d introduces $\binom{d}{2} \approx d^2$ wedges. In graphs with huge degree nodes, algorithms that scale with the number of wedges become intractable.

To mitigate this explosion in work caused by hubs, we introduce a pre-processing routine for handling nodes of high degree. By extracting all nodes with degree higher than some threshold, before performing a clustering algorithm, we drastically reduce the amount of work and storage required. Specifically, we show that with this hub-handling subroutine, the link clustering algorithm performs at most $O(n\delta^2)$ work on power law networks, where δ is a small constant near the average degree.

In many networks large degree nodes are important or influential, and so ignoring their effect in clustering can potentially destroy network structure that would better inform clustering. However, in our limited experimentation we found no such degradation in clustering quality. Nevertheless, clustering labels for the removed nodes might be desirable. If cluster labels are desired for the removed nodes (or edges), then the deleted nodes must somehow be assigned to clusters in post-processing. We propose a method for efficiently and meaningfully assigning cluster labels to these "deleted" nodes. We describe the pre- and post-processing routines next.

Table 1: Error of predicted runtime of graph property$^\alpha$ for best α. Number of wedges predicts the runtime better than other graph features.

	Edges	Degree	Nodes	Triangles	Wedges
Link clustering	0.0023	0.0034	**0.0016**	0.0061	0.0017
Infomap	0.3474	0.3174	0.3157	0.3437	**0.3092**
Optimized LC	0.0073	0.0025	0.0036	0.0077	**0.0018**
BigCLAM	0.0439	0.0385	0.0380	0.0327	**0.0318**

Deletion threshold. Intuitively, a node connected to many other nodes is probably (1) already well-known or (2) worth analyzing individually. We suggest that it is reasonable to ignore such nodes during clustering. This threshold will likely vary across datasets and applications, so we leave the threshold as a user defined input parameter. That said, we recommend a threshold near $\sqrt{n}$ as a default setting. Since many networks have degree distributions that follow a power law, this strikes a balance between guaranteeing a fast runtime (nearly linear in n) and preserving structure.

Labeling deleted nodes. To generate cluster-label information for these deleted nodes, we suggest adding deleted nodes to any cluster where their addition will improve the link density; that is, any cluster well-connected with the deleted node. In Section 4.2 we show how to determine this efficiently. Conceptually, we add a deleted node v to a cluster C when the percent of nodes in C to which v links is at least as large as the current link density of C.

Non-partition edge density. Our edge labeling procedure possibly assigns some edges to multiple clusters and some to none. Our algorithm no longer produces a partition of the edges. The original algorithm [1] produces a partition, and Ahn et al. define the *partition density* on a partitioning of the edges. Our lack of an edge partition means that we cannot apply partition density directly. Instead, we generalize partition density to measure the same qualitative property, namely, the average edge density of the clusters produced. We call this quantity *average link density.*

4.1 Determining deletions

Deleting nodes prior to clustering aims to reduce runtime, without degrading cluster quality. To achieve this, we want to establish a threshold such that deleting higher degree nodes keeps enough nodes to maintain the clustering quality. Conceptually this should be possible: a "good" cluster should remain after losing several nodes (otherwise, it would be poorly connected). Thus, we hope that removing several hubs from a graph will minimally perturb the cluster quality (and our experiments support this).

Such hub deletion potentially can improve the cluster quality. High-degree nodes can connect nodes from disparate clusters, potentially introducing noisy edges. Our experimental evidence suggests that removing hubs can clarify structure: the average edge density increases when we remove high-degree nodes prior to clustering. We note that this procedure fits networks with a few high degree nodes that connect nearly all nodes (e.g., a communication network). For example, in a university email network, emails from the university president might span the entire university; messages from a college dean would go to every department inside that college. Ignoring these nodes clarifies the individual communities (e.g., departments, student clubs).

Respecting the natural community size in a particular dataset requires thoughtful selection of the degree threshold. We propose identifying the largest node degree within a cluster. That is, computing the maximum number of within-cluster neighbors for each cluster, and setting T to be a small constant times the largest such value. Deleting nodes with degree greater than T removes nodes that tie together many different clusters, and, hopefully, keeps nodes vital to community structure.

Heuristically, we find a threshold of $T = n/5$ works for small graphs, $n < 10^5$, and a threshold of $\sqrt{n}$ works well for larger graphs, $n \geq 10^5$. The threshold $\sqrt{n}$ appeals because it guarantees that shelf sort computes an exact sort (Section 2.4.2).

4.1.1 *Link clustering runtime without hubs*

The complexity of the link clustering algorithm is $\Theta(w)$ where $w = \sum_{k=1}^{n} \binom{d(k)}{2}$ is the number of *wedges* (walks of length two) in the graph. So we know that deleting a node of degree $d(k)$ reduces the runtime by exactly $\Theta\left(\binom{d(k)}{2}\right)$. In the case that the degree distribution follows a power law, we can show that deleting all nodes with degree above a threshold $\sqrt{n}$ produces a runtime of $\Theta(\delta^2 n)$, where δ is a small constant related to the minimum degree of the graph.

Deletion process. First, note that deleting the hubs is fast. The procedure computes node degrees to determine which nodes to delete (constant work per node), selects nodes with degrees above the threshold ($O(n)$ total work), and extracts the subgraph without those nodes (bounded by $O(m)$). The work involved in the full procedure is bounded by $O(n + m)$.

Speedup in power law graphs. Here, we consider networks with a power law degree sequence, a property common to communication networks. Let d be the maximum degree in the network. Then, following a similar analysis in [4], we assume the kth largest degree, $d(k)$, can be bounded by $d(k) \leq Qdk^{-p}$ for some constant p likely between 0.5 and 1 and some constant Q near the average degree. Then our bound on the work performed by link clustering (which equals the number of wedges) can be updated as follows. We know $w = \sum_{k=1}^{n} \binom{d(k)}{2}$, and bounding $\binom{d(k)}{2} \leq d(k)^2$ enables us to upper-bound $w \leq \sum_{k=1}^{n} d(k)^2$. This is then bounded by $\sum_{k=1}^{n} (Qdk^{-p})^2 = Q^2 d^2 \sum_{k=1}^{n} k^{-2p}$. Using a left-hand integral bound on this sum gives

$$\sum_{k=1}^{n} k^{-2p} \quad \leq \quad \int_{1}^{n} x^{-2p} \mathrm{d}x \quad = \quad \frac{1}{1-2p}(n^{1-2p} - 1),$$

yielding the following bound on work (the number of wedges):

$$w \leq Q^2 d^2 \tfrac{n^{1-2p}-1}{1-2p}. \tag{3}$$

For p near 1, this is upper-bounded by Q^2d^2, which explains theoretically why we witness a runtime scaling quadratically with the maximum degree. As p approaches 0.5, the fraction approaches $\log(n)$, leading to a runtime of $d^2\log(n)$ for denser power law graphs.

We remark that, in practice, real world networks rarely have a degree distribution that follows a power law exactly [2]. One common difference between an exact power law and a real world degree distribution is that the few largest degree nodes frequently skew upwards from the otherwise log-log linear relationship; for example, on Twitter the nodes for Justin Bieber or Barack Obama have a max degree of roughly $n/5$. With this in mind, a runtime of $O(d^2)$ can easily be $O(n^2)$ when the max degree is huge.

Speedup. Small graphs ($n < 10^5$) run quickly, so we focus here on the runtime of large graphs, for which we remove nodes of degree $\geq \sqrt{n}$. Naïvely plugging this new maximum degree into Equation (3) yields a simple upper bound on work of $O(Q^2 n \frac{n^{1-2p}-1}{1-2p})$. Depending on the power law exponent p, this can vary from $O(Q^2n)$ to $O(Q^2 n\log(n))$, where Q is near the average degree. Note that this bound is loose; removing hubs reduces the degrees of many remaining nodes.

4.2 Assigning hubs to clusters

Here, we discuss a procedure for assigning deleted nodes to clusters. The deleted high degree nodes can be assigned to every community to which they are adjacent. However, this would likely assign hubs to most clusters, which might obscure information.

Instead we do the following for each deleted node, v. For each cluster C containing a neighbor of v, we compare the link densities $\mathrm{LD}(C)$ and $\mathrm{LD}(C \cup \{v\})$. If introducing the node v improves the link density of C, then we assign v to that cluster. This process could assign an edge to multiple clusters, or to none; breaking the guarantee of an edge partition. It would be easy to flag such edges for further analysis; for example, an edge without a cluster could indicate anomalous communication.

We organize the remainder of the section as follows. In Section 4.2.1 we present lemmas that efficiently determine whether a deleted node v should be assigned to a given cluster C. Then, in Section 4.3 we describe the algorithm to assign labels to high degree nodes. Because this method loses the partition guarantee, we introduce a generalized definition of partition density.

4.2.1 Link density

Given a set S of nodes with m edges and n nodes, the *link density* of S is $\mathrm{LD}(S) := \frac{m-(n-1)}{\binom{n}{2}-(n-1)}$. The following reformulation of this will enable easier steps in later proofs:

$$\mathrm{LD}(S) = 2\frac{m-(n-1)}{(n-1)(n-2)}. \tag{4}$$

The proof is straightforward algebraic simplification.

LEMMA 1. *Consider a set of nodes, S, and a set of nodes T exterior to S. We want to know under what circumstances the link density of their union $T \cup S$ surpasses the link densities of the pieces. Let S have n nodes and m edges, and let T have k nodes. Let d be the number of edges that go from T to any node in S or in T. Assume the nodes of S and T are disjoint. Then we have $LD(S\cup T) \geq LD(S)$ iff the link density of "T and the edges from T to S" is greater than $LD(S)$; more rigorously, iff*

$$\frac{d-k}{\binom{k}{2}+nk-k} \geq \frac{m-(n-1)}{\binom{n}{2}-(n-1)}. \tag{5}$$

We arranged the expression in Inequality (5) to highlight the similarity of the sides of the inequality. In particular, note that link density is "(number of edges - minimum number of edges necessary to connect S)/(number of possible edges - minimum number of edges necessary to connect S)". Similarly, the left-hand expression in (5) is "(number of edges in T and from T to S - minimum number of edges necessary to connect all nodes of T to S)/(possible number of edges - minimum number of edges necessary to connect all nodes of T to S) ".

PROOF. We prove a related inequality simplified by substituting in variable names. let $A = m-(n-1)$, $B = d-k$, $a = (n-1)$, and $b = (n-2)$. Observe that the inequality $\frac{A+B}{(a+k)(b+k)} \geq \frac{A}{ab}$ can be rearranged:

$$(A+B)ab \geq A(a+k)(b+k) \tag{6}$$
$$(A+B)ab \geq Aab + A[(a+k)(b+k)-ab] \tag{7}$$
$$Bab \geq A[(a+k)(b+k)-ab], \tag{8}$$

and this is equivalent to $\frac{B}{[(a+k)(b+k)-ab]} \geq \frac{A}{ab}$. Note that this chain assumes there are no zero divisors; this is guaranteed when $n \geq 2$ and $k \geq 1$. Substituting in our original quantities, we have proved that

$$\frac{m-(n-1)+(d-k)}{((n-1)+k)((n-2)+k)} \geq \frac{m-(n-1)}{(n-1)(n-2)}$$

holds if and only if $\frac{(d-k)}{\binom{k}{2}+nk-k} \geq \frac{m-(n-1)}{\binom{n}{2}-(n-1)}$. □

This formula has the same intuition as the original link density formula: in order for the union $S \cup T$ to be more "edge dense" than S alone, the number of new edges, d, divided by the total number of possible new edges, $\binom{k}{2}+nk$, must be greater than $\mathrm{LD}(S)$. (Note that both quantities, "actual number of new edges" and "possible number of new edges," are normalized by subtracting k, the minimum number of edges necessary to connect all of T to S).

COROLLARY 1. *In the case that T consists of a single node, the above formula simplifies to the following: adding a node v to a set S will improve the link density of S iff*

$$\frac{d-1}{n-1} \geq LD(S) = 2\frac{m-(n-1)}{(n-1)(n-2)},$$

where d is the number of edges from v to S, n is the number of nodes in S, and m is the number of edges in S. This occurs iff

$$(d+1)(n-2) \geq 2(m-1). \tag{9}$$

This equivalence holds in the case $n = 2$, assuming S contains an edge.

Intuitively, adding a node v to a set S will improve the link density if and only if the degree of v per node in S is greater than the total number of edges of S per node of S. Interestingly, this formula holds even in the case $n = 2$, assuming the cluster of size 2 actually contains an edge; this is because of the way link density is defined on connected clusters of size 2, LD (one edge) = 0 instead of 1.

4.3 Algorithm summary

Computing these cluster labels efficiently requires:

1. Determining a list of clusters that neighbor v,
2. For each cluster C, counting the number of edges from v to C, and
3. Using the corollary to determine, for each C, whether adding v improves link density.

Determining and counting neighboring clusters. Given a deleted node v, we want to efficiently determine a list of adjacent clusters and how many neighbors v has in each adjacent cluster. Link clustering outputs a list of edge cluster labels. We iterate over this list marking the endpoints of each edge with the corresponding cluster. This produces a node-to-cluster map. Then, for each neighbor u of v, we add the cluster labels of u to the list of v's cluster-neighbors. The cluster-neighbor list tracks cluster labels with multiplicity; this way, this procedure determines which clusters the deleted node v is connected to, as well as how many connections v has to each of those clusters.

Work involved. This process observes each occurrence of v being incident to a cluster, by a neighbor of v being in that cluster. A node can be in as many clusters as its degree (since the community membership of a node's edges determines the node's community membership, and each remaining edge belongs to exactly one community). Thus, the summed degrees of its neighbors bounds how often a node can neighbor a community. Hence, the work is bounded by $\sum_{j=1}^{d(v)} d(v_j) \leq O(m-$ sum of degrees of deleted nodes$)$.

Identifying clusters that benefit from the deleted nodes. Note that Inequality (9) can determine if adding v to a cluster S will improve link density, without having to compute the link density of either S or $S \cup \{v\}$. This is because we already determined the number of edges from each deleted node v to each cluster C. With the cluster-neighbor information computed, this inequality enables us to efficiently determine the clusters to which we will add v.

4.4 Generalizing partition density

Given a set of clusters C that partition the edges of a graph, Ahn, Bagrow, and Lehmann compute the partition density, D of the clustering C as follows. For each cluster $T \in C$, let p_T be the link density of cluster T, i.e. $p_T = 2(m_T - (n_T - 1))/((n_T - 1)(n_T - 2))$. Then they average the link densities of clusters, weighted by the number of edges per cluster: $D = \frac{1}{m} \sum_{T \in C} m_T \cdot p_T$. Thus, the quantity D is at most 1, which occurs when every edge belongs to a fully dense cluster.

When edges belong to multiple clusters, this quantity breaks down. The sum $\sum_{T \in C} m_T \cdot p_T$ can exceed m because edges are counted multiple times when they belong to multiple clusters. The quantity D no longer stays bounded by 1, which we desire it to be.

To correctly normalize this sum, we divide by the total number of edge-cluster memberships, M. If c_e denotes the number of clusters to which edge e belongs, then $M := \sum_{e \in E} c_e$ is the total number of edge-cluster memberships (with multiplicity). If each edge belongs to exactly one cluster, then $M = m$; if edges belong to more than one cluster, then $M \geq m$. Dividing weighted link density by M yields $D_{\text{generalized}} = D_M = \frac{1}{M} \sum_{T \in C} m_T \cdot p_T$. This preserves the intuition of partition density: it measures the average edge density of clusters, weighting each cluster's edge density by the edges per cluster.

Issues. One problem with this definition of average cluster edge density is that we can skew the quantity to over-represent sparse (or dense) clusters by simply including many clusters centered around the same set of nodes. Consider what happens if we include a dense (or sparse) subset of nodes S in the overlapping clustering. For $S \cup \{v\}$ for $v \notin S$, or even $S \cup T$ for T disjoint from S, the set S becomes over-represented in the overlapping clustering. This biases the average cluster edge density toward the density (or sparsity) of S.

Although this is a potentially severe problem with generalizing partition density, our algorithm avoids this. Link clustering forms a partition of the edges, so no edge of the subgraph is double-counted before post-processing—thus, no node subset is overrepresented as in the pathological case describe above. Returning the high degree nodes to the graph cannot cause overrepresentation because this stage does not create any new clusters; it merely adds the deleted nodes to predetermined clusters. While imperfect, we believe this generalized notion of edge density provides a meaningful measure of overlapping cluster quality.

4.5 Empirical results

We evaluate the performance of the hub removal strategy on the unweighted graphs from an enterprise network email dataset as described in Section 3. We present the results of this analysis in Figure 2 with the previously outlined comparison algorithms. Hub removal greatly reduces the runtime of Optimized LC, making it run more quickly than even BigCLAM on these graphs.

Figure 3 shows that hub removal provides an additional runtime improvement on SNAP datasets [9]. We compare with Infomap [10] and BigCLAM [13] runtimes, and show that optimized link clustering, both with and without hub removal, was faster than BigCLAM. Figure 4 shows a quality evaluation comparing the ground truth communities with the discovered communities. We find that hub removal leaves

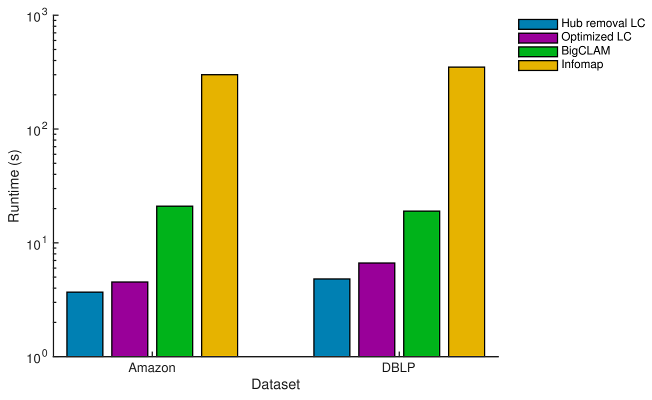

Figure 3: Runtime comparison on Amazon and DBLP. The original implementation runs out of memory on networks this large, but our optimized versions overcome this problem.

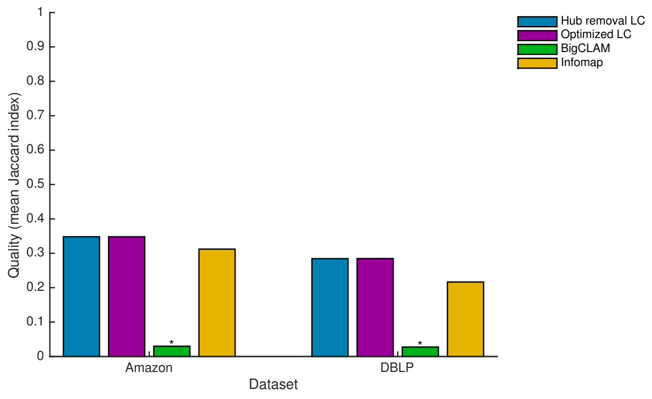

Figure 4: Link clustering quality on Amazon and DBLP. Quality is measured by how well the discovered clusters match ground truth under the Jaccard index. *We believe a formatting mismatch between BigCLAM and the SNAP datasets caused poor performance.

the link clustering quality intact.

5. CONCLUSION

We consider scalability of clustering. As a case study, we present tight complexity analysis of the link clustering method and show it scales with a sort on wedges. This identifies bottlenecks common to many clustering methods: sorting and high degree nodes. We show that Jaccard Index similarity scores on graphs can be structured so as to enable linear time sorting instead of $\Theta(n \log n)$ work (where n is the number of scores being sorted). Our experiments show that this provides a several order of magnitude reduction in runtime of the link clustering algorithm. Additionally, we investigate the removal of high degree nodes before clustering to improve runtime and memory usage, and their replacement after to avoid loss of clustering quality. Our experiments show that our insights yield a great reduction in the runtime of link clustering, without a reduction in community quality. We are hopeful that these insights will yield similar speed-ups for other clustering algorithms and graph analysis tools that do not scale well in the presence of large degree node.

6. ACKNOWLEDGMENTS

We would like to thank Chris Smith for measurement assistance and Kevin M. Carter for valuable feedback.

7. REFERENCES

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

[2] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 2009.

[3] J. H. Fowler and N. A. Christakis. Dynamic spread of happiness in a large social network: longitudinal analysis over 20 years in the Framingham heart study. *Bmj*, 2008.

[4] D. F. Gleich and K. Kloster. Sublinear column-wise actions of the matrix exponential on social networks. *Internet Mathematics*, 11(4-5):352–384, 2015.

[5] R. V. Hogg and J. Ledolter. *Engineering statistics.* Macmillan Pub Co, 1987.

[6] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods.* John Wiley & Sons, 2013.

[7] K. Kloster and D. F. Gleich. Personalized pagerank solution paths. *arXiv preprint arXiv:1503.00322*, 2015.

[8] D. E. Knuth. *The art of computer programming: sorting and searching.* Pearson Education, 1998.

[9] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. 2014.

[10] M. Rosvall and C. T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 2008.

[11] V. Spirin and L. A. Mirny. Protein complexes and functional modules in molecular networks. *Proceedings of the National Academy of Sciences*, 2003.

[12] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Computing Surveys (csur)*, 2013.

[13] J. Yang and J. Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *ACM international conference on web search and data mining*, 2013.