2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

Percolation Computation in Complex Networks

Fergal Reid
Clique Research Cluster
Complex Adaptive Systems Laboratory
University College Dublin
Email: fergal.reid@gmail.com

Aaron McDaid
Clique Research Cluster
Complex Adaptive Systems Laboratory
University College Dublin
Dublin 4, Ireland

Neil Hurley
Clique Research Cluster
Complex Adaptive Systems Laboratory
University College Dublin
Dublin 4, Ireland

***Abstract*—K-clique percolation is an overlapping community finding algorithm which extracts particular structures, comprised of overlapping cliques, from complex networks. While it is conceptually straightforward, and can be elegantly expressed using clique graphs, certain aspects of k-clique percolation are computationally challenging in practice. In this paper we investigate aspects of empirical social networks, such as the large numbers of overlapping maximal cliques contained within them, that make clique percolation, and clique graph representations, computationally expensive. We motivate a simple algorithm to conduct clique percolation, and investigate its performance compared to current best-in-class algorithms. We present improvements to this algorithm[1], which allow us to perform k-clique percolation on much larger empirical datasets. Our approaches perform much better than existing algorithms on networks exhibiting pervasively overlapping community structure, especially for higher values of k. However, clique percolation remains a hard computational problem; current algorithms still scale worse than some other overlapping community finding algorithms.**

I. Introduction

One particular type of percolation, important in the study of complex networks, studied by Palla et al. [1] is *k-clique percolation*, which Fortunato's review [2] of community detection methods describes as the most popular overlapping community detection method. A *clique* is a group of nodes in a network, such that every node is connected to each other node. Palla et al. argue that percolated k-cliques – groups of cliques of size k, that are connected together by cliques of size (k-1) – are important structures in complex networks, and a good way to find community structure. They discuss k-clique percolation in a variety of application contexts, including authorship networks, word association networks, and certain biological contexts, for example, claiming that percolated k-cliques in protein-protein interaction networks correspond to functional units of protein structure. Additionally, Palla, Barabasi, and Vicsek [3] apply k-clique percolation to the study of social dynamics, in mobile telecoms social network datasets.

More generally, the work of Evans [4] motivates 'clique graphs' as interesting constructions to use when studying structure in networks and finding communities. A 'clique graph' of a specific source network is formed by representing each clique in the source network by a node in the 'clique graph'. Pairs of nodes in the clique graph are then connected by edges where the corresponding pair of cliques overlap in the source network. Clique graphs have many interesting properties, not least that k-clique percolation can be expressed as a simple thresholding on the clique graph. Clique graph construction and k-clique percolation are thus linked. Evans argues that different types of partitioning and thresholding of the clique graph are interesting to examine for the purposes of overlapping community detection, and also to examine structure in complex networks generally.

However, in practice these approaches to find network structure present computational challenges. In a naive algorithmic approach to k-clique percolation, one algorithm to percolate k-cliques – for example as described in the review of Fortunato, citing the work of Everett and Borgatti [5] – is "*In order to find k-clique communities, one searches first for maximal cliques. Then a clique-clique overlap matrix O is built, which is an $n_c \times n_c$ matrix, n_c being the number of cliques; O_{ij} is the number of vertices shared by cliques i and j. To find k-cliques, one needs simply to keep the entries of O which are larger than or equal to $k-1$, set the others to zero and find the connected components of the resulting matrix.*" This simple algorithm, which essentially finds k-clique percolations by finding connected components in a certain construction of the clique graph, is quadratic in the number of maximal cliques in the graph, by virtue of the way that it builds the complete clique-clique overlap matrix. Furthermore, the number of cliques, and maximal cliques, in an empirical network may be *very* large, as we will discuss.

Some work has presented faster computational methods of obtaining percolated k-clique structures. The authors of the original Palla et al. paper have provided a faster CFinder [6] implementation to find percolated k-cliques. More recent work has been done by Kumpula et al. with their 'Sequential Clique Percolation' algorithm (SCP) [7] which further improves efficiency. However, these improved methods often perform poorly on networks with the kind of pervasively overlapping community structure we see in many real world social networks – an area of increasing interest in the applied study of community structure – and particularly poorly when performing percolation with high values of k. In this work, we consider several computational aspects of the problem of percolating structures in large complex networks. We focus our discussion on k-clique percolation, but the techniques we describe could be applied to the computation of many similar percolation problems.

[1]Source code available: http://sites.google.com/site/CliquePercComp

This work is supported by Science Foundation Ireland grant 08/SRC/I1407

978-0-7695-4799-2/12 $26.00 © 2012 IEEE
DOI 10.1109/ASONAM.2012.54

A. Structure of the paper

In Section II we define clique percolation. We also discuss clique graphs, which are a useful conceptual tool to understand the problem and to gain insight into various algorithms. We show how, even though cliques are indicative of community structure, there can be many more cliques than communities.

Algorithms typically fall into one of two classes: either they are based on finding the k-cliques or else they limit themselves to the *maximal* cliques with at least k nodes in them. We explain how these are equivalent and discuss an example of each kind from the literature.

In Section III we discuss some challenges that face all the examples in the literature and which are also faced by our algorithms. The number of cliques can be very large, as can the size of the clique graph. Also we explain why, while it is tempting to attempt to represent a community of cliques merely as the union of the nodes in its constituent cliques, such a method will lead to an incorrect algorithm.

In Section IV, we define and motivate our algorithms and evaluate the speed of them against SCP on a variety of synthetic networks and on empirical social networks.

II. K-clique Percolation

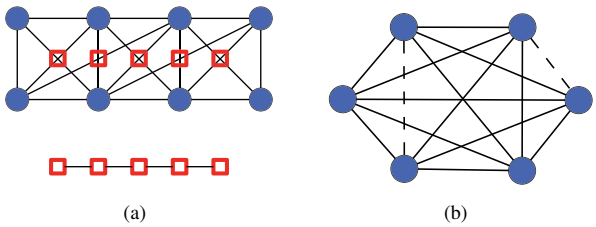
In k-clique percolation, if two cliques of size k share k-1 nodes, we say that these two cliques percolate into each other. The maximal sets of cliques, which satisfy the property that every clique in the set is reachable from every other clique in the set, through a path connecting percolating pairs, form the communities output by k-clique percolation. These communities are then typically output as the sets of nodes contained within each set of cliques – and may overlap.

A. Clique Graphs

A clique graph is formed from a source network by creating a node in the clique graph for each clique in the source network and joining two nodes in the clique graph if their corresponding cliques in the source network share nodes, or 'overlap' – see Figure 1(a) for a visualisation.

The edges in a clique graph may be weighted, corresponding to the overlap between the cliques. Thresholding such a graph – removing the nodes corresponding to cliques of size less than some value k, and removing edges with weight lower than $(k-1)$ – yields a graph in which the connected components are the cliques that would have percolated into each other, in k-clique percolation. Figures 3a, 3b, and 3c illustrate this process. As both thresholding the edges of a graph and calculating the connected components of a graph are computationally inexpensive, it is inexpensive to calculate the k-clique percolations, given the clique graph.

However, while clique graphs are an elegant construction, and a useful tool for reasoning about clique percolation, there are computational obstacles to their use on many empirical networks, as we shall see; in practice, the clique graph often takes a very long time to construct. We will also later see that it is in fact *not* necessary to compute the entire clique graph in order to compute the percolated communities.



(a) (b)

Fig. 1. (a) This illustration shows a very simple network, and its clique graph. We represent the source network with circular nodes. We represent the maximal cliques in this network as squares, placing a square at the center of each 4-clique. The corresponding maximal clique graph, with threshold of overlap 3, is shown below. The connected components in this graph – there is only one in our example – correspond to the cliques that percolate into each other – this entire graph would be one k-clique percolation, for $k = 4$.
(b) A maximal 6-clique. If the dashed edges are removed, this diagram shown now contains 4 maximal cliques. This counter-intuitive behaviour, whereby missing edges can increase the number of maximal cliques in a network, partly illustrates why a particular empirical network can have many more maximal cliques than it has edges, nodes, and communities.

B. Cliques vs. Maximal Cliques

Throughout this paper, we discuss k-clique percolation, with reference to two definitions of 'clique': A 'clique' is a fully connected sub-graph – a set of nodes all of which are connected to each other. A 'maximal clique' is a clique that is contained in no larger clique. Every maximal clique is a clique, by definition, but the opposite does not hold. Thus there are always more cliques than maximal cliques.

It is important to realise that cliques of size k will always percolate within a larger clique of size greater than k; thus a maximal clique of size greater than k contains a k-clique percolation (which may also extend outside it). Further, any k-clique sharing k-1 nodes with this maximal clique will be part of the same percolation. Hence, clique percolation can be equivalently defined in these two different ways; one definition is based on a clique graph of every clique with exactly k nodes, while another definition is based on a maximal-clique graph of every maximal clique of at least k nodes. Various algorithmic implementations make use of each of these definitions, with consequences for computational performance.

C. More cliques than communities

The large numbers of maximal cliques found in the 'Facebook 100' networks [8] shown in Table I – much greater than the number of nodes in each network – may be surprising, given the sociological interpretation of cliques as a community structure [9]; if a clique corresponds to the idea of a community, we may wonder why social networks contain so many more maximal cliques than nodes. However, in empirical social networks, we often observe a situation where a large maximal clique may overlap heavily with many other smaller maximal cliques that differ from it by only a few nodes. It is also the case, that if edges are randomly deleted from a single maximal clique, this maximal clique will often be replaced by many smaller maximal cliques. A visualisation of this is shown in Figure 1(b), where a single maximal clique, after having 2 edges removed, turns into 4 maximal cliques.

D. Previous approaches

1) CFinder: The original CFinder implementation, as described by Palla et al. in supplementary material [1], follows the method of Everett and Borgatti, finding all *maximal* cliques, by a custom method, and generating the overlap matrix between these cliques. Once this overlap matrix is found, percolations for all values of k can then easily be calculated. However, building the full overlap matrix – equivalent to the full clique graph – naively requires $O(n_c^2)$ clique-clique comparisons, in n_c the number of maximal cliques. Thus, CFinder performs poorly on the empirical networks we study.

2) SCP: SCP [7], on the other hand, works off a custom clique finding algorithm to find, not all the *maximal* cliques, but all cliques of size k. For each clique of size k, or 'k-clique', it is trivial to find the cliques of size k-1 inside it. SCP is based on a bipartite graph with the k-cliques on one side and the (k-1)-cliques on the other side. A (k-1)-clique is linked to every k-clique containing it. The connected components in this bipartite network correspond to the clique percolation communities. The idea is that two (k-1)-cliques percolate into each other if they are both members of the same k-clique, and this is equivalent to the conventional definition of two k-cliques being linked if they share k-1 nodes.

While SCP consistently outperforms CFinder [7], the important term for the performance of SCP on a network is thus not the number of maximal cliques of size at least k, but the number of cliques of size k-1. As we discuss in our experimental work, empirical social networks often contain maximal cliques of substantial size; given that such large maximal cliques contain very many smaller k-cliques, the fact that SCP has complexity in terms of the number of cliques of size k-1 often leads to poor performance in practice. We note that, as stated by Kumpula et al., SCP is most efficient for low sizes of k, and performs poorly as the value of k increases.

III. Computational Challenges

In this section, we look at some of the challenges that face any k-clique percolation algorithm. We discuss our specific solutions and algorithms in the next section.

A. Quantity of cliques

To understand why k-clique percolation is so computationally expensive on certain types of online social networks, we first quantify the numbers of maximal cliques in these empirical networks. The various algorithms deal with the edges in the clique graph differently, but each algorithm requires all the (maximal) cliques be found first.

Our primary interest, in this work, is in clique percolation in the challenging domain of modern online social networks, which exhibit pervasive overlap [10] [11]. The 'Facebook 100' dataset of Traud et al. [8] provides us with a range of similar empirical networks of differing size. We focus on the smallest 10 of these networks – shown in Table I – which capture the general topological features, in terms of densely overlapping community structure, but are also small enough to benchmark existing algorithms efficiently on. There are very large numbers of maximal cliques in these networks, many more than the number of nodes or edges.

TABLE I
THE VAST NUMBER OF MAXIMAL CLIQUES (SIZE>=3) PRESENT, EVEN IN RELATIVELY SMALL EMPIRICAL ON-LINE SOCIAL NETWORKS. 'CGBOUND' IS THE NUMBER OF EDGES, IN 1000S, WE CALCULATED IN EACH $k = 5$ MAXIMAL CLIQUE GRAPH, AFTER SEVERAL WEEKS COMPUTATION, AND IS AN UNDERESTIMATE OF THE TRUE VALUE. (*REED98 IS THE EXACT NUMBER, NOT AN UNDERESTIMATE.)

Network	Nodes	Edges	Maximal Cliques	Largest Clique	CGBound (1000s)
Caltech36	769	16,656	32,207	20	2,475
Reed98	962	18,812	33,991	16	1,774*
Simmons81	1,518	32,988	45,538	19	2,244
Haverford76	1,446	59,589	475,567	24	127
Swarthmore42	1,659	61,050	306,542	20	282
USFCA72	2,682	65,252	108,929	29	1,128
Mich67	3,748	81,903	154,971	27	843
Bowdoin47	2,252	84,387	331,738	23	235
Oberlin44	2,920	89,912	198,803	22	644
Amherst41	2,235	90,954	599,430	21	79

B. Clique Graph size

From the large numbers of cliques present in these types of pervasively overlapping network, it can be seen that naively performing $O(n_c^2)$ intersection tests, in n_c the number of cliques, in order to populate a clique-clique intersection matrix would be computationally prohibitive.

However, in addition, by attempting to measure the size of the maximal clique graph, we can see that even if we had a fast way to generate the full clique graph, it would be extremely unwieldy to work with, because of the vast number of edges in it. It is computationally expensive to obtain all the edges in the clique graph for these networks, but running a process to calculate maximal clique graph edges, for a limited amount of time, allows us to calculate a lower bound on the numbers of clique graph edges. Table I shows an underestimate of the number of edges in each clique graph, formed after over one week of computation. The number of actual edges present may be vastly greater, but merely dealing with clique graphs that are sometimes so many orders of magnitude larger than the original source network would be computationally prohibitive.

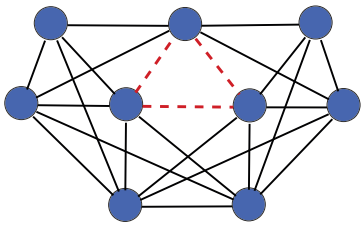
These figures illustrate that methods of clique percolation, or indeed general graph analysis, which require the full clique graph to be built, are unscalable on these networks. This motivates methods, such as those we present, which build only a subset of the clique graph.

C. The necessity of dealing with individual cliques

As cliques are percolating, we would ideally discard the individual cliques that comprise a percolating structure, and instead describe that structure as a set of nodes and edges. As there are so many more cliques than nodes or edges, this would increase efficiency. However, we cannot do this, because it is not possible to tell, given a set of nodes, and the edges between them, whether a given clique percolates into this structure, without knowing the (k-1)-cliques that comprise this structure. Counter-intuitively, a percolating k-clique structure

can contain (k-1)-cliques that are not part of the k-cliques that yield it. Figure 2 shows an example of this. As such, it is not possible to test a candidate clique for percolation, against a percolating structure, without maintaining a list of the k, or k-1, cliques that make it up. We believe this is a core computational issue in k-clique percolation.

Fig. 2. We cannot use an intermediate representation to store a percolating k-clique structure, without storing the individual cliques that comprise it: Consider the case where we store just the nodes and edges of the percolating 4-clique structure shown in the diagram. Even if the highlighted dashed triangle was part of another 4-clique, with an additional node, not shown, this other 4-clique would *not* percolate into the 4-clique-community comprising all the shown nodes; even though all three nodes and edges in the highlighted triangle are part of the percolated 4-clique community, the *triangle* is not part of any of the constituent 4-cliques.



IV. Our Algorithms

In this section we describe two algorithms. Our algorithms initially obtain the maximal cliques in the graph, and then attempt to minimise the number of clique overlap tests that must be carried out to obtain k-clique communities. Algorithm 1 attempts to build a *minimal spanning forest* over the maximal cliques, using a simple data structure to reduce unnecessary clique intersection tests. Algorithm 2 uses a hierarchical data structure to further reduce the number of full intersection tests needed, further improving performance.

1) Maximal Cliques: Like CFinder we have chosen to build our algorithms around maximal cliques with at least k nodes; this is in contrast to SCP, which uses k-cliques. This is because large maximal cliques are often present in empirical data; these contain large numbers of k-cliques and thus it is often more efficient to work with the maximal cliques. Figure 4 shows the clique- and maximal-clique distributions on a typical Facebook network – though there are many maximal cliques, there are many more k-cliques than there are maximal cliques of size greater than k, for all but the smallest values of k.

2) Clique finding: The Bron-Kerbosch clique finding algorithm [12] provides us with a fast way to find all maximal cliques in the network. While finding all cliques, or maximal cliques, is computationally hard on an arbitrary graph, on the networks that we examine in practice, this method finds all clique extremely fast, such that clique-finding is a small part of the computational time in our algorithms. Instead computation is dominated by comparing cliques against each other.

A. Algorithmic framework

Our goal is to find which cliques are in which connected components of the thresholded clique graph. We follow a procedure similar to the well known [13] connected components algorithm. We process one component at a time, by taking an arbitrary starting clique and identifying all the cliques in the same connected component as it. This process repeats until all cliques have been assigned to a component.

The key point is that we do not first generate the full clique graph; instead we integrate the process of generating clique graph edges, with that of finding connected components; and only generate clique graph edges as we need them for the connected components calculation. We can do this because, to calculate the k-clique percolations, we only need the minimal spanning forest of the clique graph (Fig 3); any other edges in the clique graph are essentially wasted intersection tests.

We say that all the cliques are initially *unvisited*, and that they each become *visited* as they are assigned to their component. As the component expands, there is a set of cliques described as the *frontier*. The expansion proceeds until the frontier is empty. At each iteration, a clique is selected from the frontier (the 'current frontier clique') and its *unvisited* neighbours in the clique graph are identified. A 'neighbouring' clique is a clique which has an intersection of at least k-1 nodes with the current frontier clique; this corresponds to an edge in the clique graph. These neighbours are moved into the frontier and marked as *visited* and our current frontier clique is moved out of the frontier, as it has now been fully processed. Eventually, there will be no unvisited neighbours of any of the frontier cliques and the frontier will gradually become empty – this completes the identification of this component. If we know that clique A is connected to clique B in the clique graph, and B to C, then there is no need to test if A is connected to C; thus we can ignore cliques that have already been visited, when searching for the neighbours of the current frontier clique.

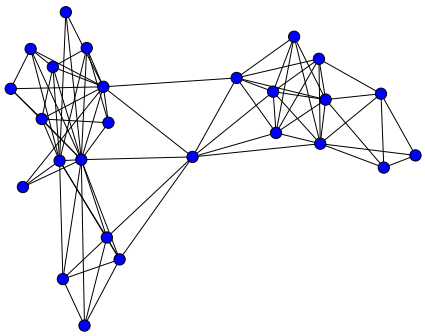
Our two algorithms differ in how they identify the unvisited neighbouring cliques of the current frontier node. Otherwise, they both use the framework which has just been described.

One naive way to identify the neighbouring unvisited cliques of the current frontier clique is to iterate through all the other cliques, testing the size of their intersection with the current frontier clique. If the intersection has at least k-1 nodes *and* if the clique is unvisited then it is a neighbouring unvisited clique. Our two algorithms embody different strategies to speed up this identification.

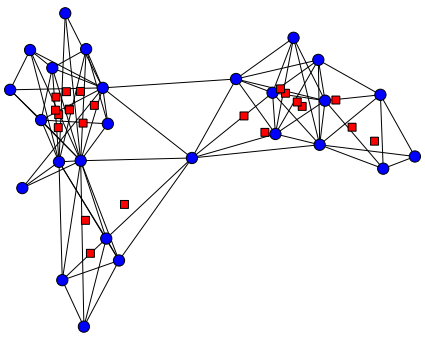
B. Algorithm 1

1) Node-to-cliques maps: For a given current frontier clique, testing against every other clique is expensive. We can vastly improve this, by realizing that two cliques can only neighbour each other in the clique graph, if they share at least one node (i.e. are adjacent by at least one node). We maintain a list, for each node, of all the cliques that the node is present in. Then, for each node in the current clique, we check the list of other cliques that node is in, to generate the set of cliques that overlap with the current clique by at least one node.

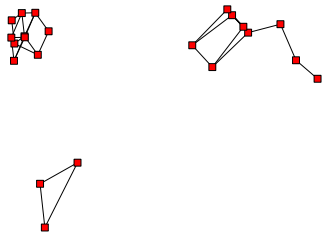
2) Visiting the cliques: The key step in Algorithm 1 is that as cliques are added to the current connected component, they are deleted from the map of nodes-to-cliques. This is how a clique is marked as 'visited', and not considered further. Pseudo-code is given in Listing 1.



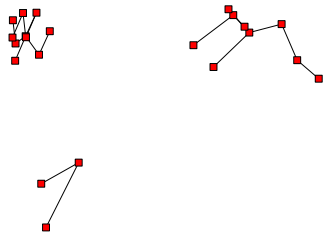
(a) The source network.



(b) Maximal cliques of size at least 4, represented by a red box at their centroids.



(c) The corresponding clique graph of maximal cliques overlapping by at least 3 nodes.



(d) A minimal spanning forest of the clique graph.

Fig. 3. A clique graph visualisation, showing the construction of the clique graph from a source network; the connected components in this graph, which correspond to clique percolations; and minimal spanning trees of these components – the minimum set of edges in the clique graph which must be found, and which put an absolute lower bound the number of clique-clique intersection tests to be done in these methods.

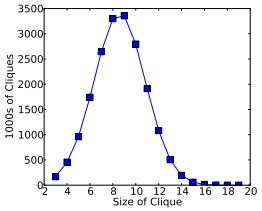
```
    for clique in cliques:
        if not clique in cliques_to_components_dict:
            current_component += 1
            cliques_to_components_dict[clique] = current_component
            frontier = set()
            frontier.add(clique)

            while len(frontier) > 0:
                current_clique = frontier.pop()
                for neighbour in get_unvisited_adjacent_cliques(current_clique, nodes_to_cliques_dict):
                    if len(current_clique.intersection(neighbour)) >= (k-1):
                        cliques_to_components_dict[neighbour] = current_component
                        frontier.add(neighbour)
                        for node in neighbour:
                            nodes_to_cliques_dict[node].remove(neighbour)
```

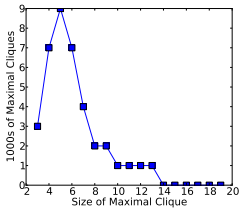
Listing 1. Python-like pseudo-code description of simple algorithm for improved clique percolation.

C. Experiments

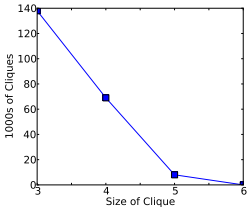
1) GN benchmarks: After Kumpula et al. [7], we performed benchmarks based on the Girvan-Newman synthetic benchmark, creating networks of increasing size, containing increasing numbers of communities, each of 32 nodes. We first note that these synthetic benchmarks are poor proxies for many real world social networks, in terms of the clique size distributions contained within them. Figure 4 shows a typical example of this difference in the size distributions of cliques, and maximal cliques, in these GN networks, versus the Facebook networks; the Facebook networks have more and larger cliques and maximal cliques. While the main advantage of our method over SCP is on networks with larger numbers of cliques, which these GN networks do not contain, we present, in Figure 5, present performance results as we increase the number of nodes in the GN network. At $k = 4$ our method, while slightly slower than SCP, is competitive, and scales similarly. However, when we start to look at higher values of k, we notice that the scaling behaviour of SCP degrades considerably faster than our method.



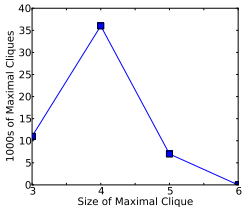
(a) Clique size distribution for Simmons81 Facebook network.



(b) Maximal clique size distribution for Simmons81 Facebook network.

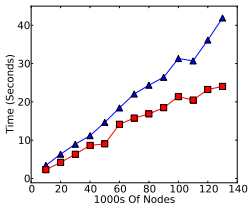


(c) Clique size distribution, GN benchmark.

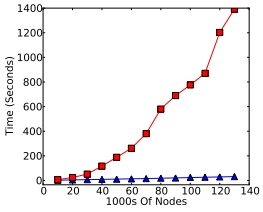


(d) Maximal clique size distribution, GN benchmark.

Fig. 4. Clique and maximal clique size distributions, for benchmark and empirical networks. The benchmark network is a GN network, of 10,000 nodes, constructed to a similar specification as in [7]. The Facebook network is that of the Simmons81 Facebook 100 dataset, and is typical of Facebook 100 data. We note that there are typically very many more cliques than maximal cliques; that the peak of the distributions for the empirical network is further to the left for the maximal cliques, than for all cliques; and that more cliques of much larger sizes are found in empirical data, than in these GN benchmarks, despite this particular GN benchmark network's larger size.



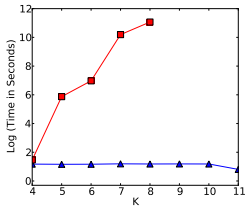
(a) Runtimes for K=4.



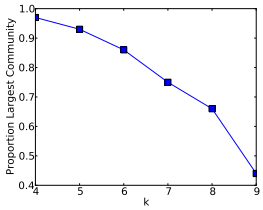
(b) Runtimes for K=5.

Fig. 5. GN benchmark results, on a similar benchmarking setup to [7]. SCP runtimes are shown by red squares; runtimes of our method are shown by blue triangles. Runtimes are observed to be similar when $k = 4$, with SCP marginally ahead; but for $k = 5$ our method appears to do much better. There are insufficient numbers of cliques of size 6 or greater present to allow higher values of k to be examined on the GN benchmarks.

2) CondMat benchmark: An example of using k-clique percolation to find communities in co-authorship data in the field of condensed matter physics, is described by Palla et al. [1]. The SCP paper [7] extends this benchmark, comparing the times of SCP and CFinder for finding k-clique percolations, in the Arxiv Condensed Matter e-print archive. We produce a similar benchmark, using the Arxiv ca-CondMat network from the SNAP dataset collection [14]. Figure 6(a) shows the runtime of SCP against that of our implementation, on this network, for varying values of k. Note that y-axis, showing runtime, is logarithmic. Both implementations quickly complete for low values of k, and, as shown in [7], far outperform CFinder; but as the value of k increases, the runtime of SCP deteriorates. As noted in the original SCP work "for networks containing large cliques the SCP method performs best for rather small values of k" [7]. Our method, by contrast, seems well suited to this particular type of network, and performs quickly, even as k is increased.



(a)



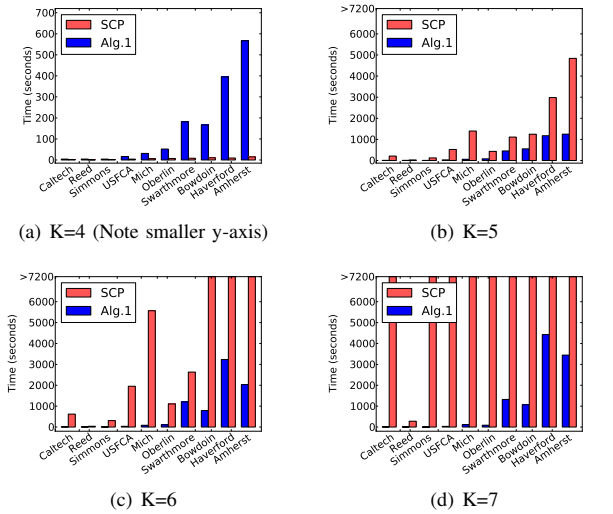
(b)

Fig. 6. (a) Results on the Arxiv CondMat network, from the SNAP dataset collection [14]. This network was also benchmarked in the work of Kumpula et al. [7]. This is a network which, perhaps due to the fact it is a one-mode projection of a bipartite author network, our algorithm seems to be particularly well on, as K grows, in contrast to SCP. The SCP implementation was terminated, on values of K larger than 8, for exceeding available memory. Any single process exceeding 75GB of RAM was terminated.
(b) The proportion of nodes assigned to any community, that are assigned to the largest community, for each value of k, on the Mich67 Facebook network. Clearly, on this particular network, low values of k simply find a core structure of the network with pervasive overlap, and do not serve to reveal modular units; larger values of k are required for k-clique percolation to be in any way meaningful on these networks.

3) Empirical social network data: We now produce a set of empirical benchmarks, not previously investigated in the k-clique percolation literature. We analyse the performance of SCP against our algorithm, for various values of k, on the ten smallest of the Facebook 100 networks. This allows us easily obtain results for higher values of k where SCP cannot complete in reasonable lengths of time. Figure 7 shows performance results for four values of k across these 10 Facebook networks. These empirical networks vary in their individual performance characteristics. In common with previous work on k-clique percolation, it is difficult to give meaningful closed form analysis of expected performance characteristics in terms of more general network parameters, such as the number of nodes, or edges; but the trend is clearly that as k increases, the performance of our method versus SCP clearly increases, across all networks. Even for the lowest value of k that we examine, our method finishes within a few minutes of SCP, while as the value of k increases, it becomes difficult to obtain runtimes for SCP. Furthermore, in additional to increasing runtime, the memory usage of SCP's data structures, which allow for fast intersection tests, grows prohibitive.

When analysing these networks, we often find that, for low values of k, a giant connected component emerges in the network, due to the pervasively overlapping structure of community in these networks. Figure 6(b) shows, for each value of k on the Mich67 Facebook network, the proportion of nodes which are in the largest k-clique percolation community with respect to the number of nodes which have been assigned to at least one k-clique percolation community. As can be clearly seen from this chart, smaller values of k do not usefully find modular structure in the network. An investigation of how meaningful these structures are as communities is, however,

beyond the scope of this computational work; but this question cannot even be investigated without an algorithm which runs reasonably quickly for higher values of k.



(a) K=4 (Note smaller y-axis) (b) K=5

(c) K=6 (d) K=7

Fig. 7. Benchmark results on the smallest 10 Facebook 100 graphs, for values of k between 4 and 7. Red columns denote SCP runtimes, blue denote Algorithm 1. Processes were terminated if they failed to complete within 2 hours (7200 seconds). Running SCP for additional time sometimes resulted in it using more than 50GB of RAM; the SCP binary implementation consumes a vast amount of RAM for some networks and values of k, presumably for its disjoint-set forest fast intersection data structure. Our method never uses more than a small multiple of the amount used to store the maximal cliques; never more than 512MB in these experiments. Experiments are run on a multiprocessor machine, in random sequence. SCP outperforms our algorithm for $k = 4$, but performs substantially worse, in both time and space requirements, as values of k get larger.

D. Counting the failed intersection tests

This type of algorithm involves testing pairs of cliques that share a node, to see if they intersect by at least k-1 nodes. For speed, our goal is to minimize the number of tests required, as this typically takes up the major portion of the runtime of such an algorithm. Each test either succeeds, if there are $>=$ (k-1) nodes in the intersection, or else fails. We cannot change the minimum number of successful intersection tests – there will be one such test for each edge in the minimal spanning tree, i.e. $numCliques - numCommunities$. But we can try to minimize the number of failed tests. In this subsection, we will look at a lower bound on the number of failed tests that must occur in Algorithm 1 described above. In many networks, the number of these failed tests is very large in comparison to the number of the successful tests, and dominates the computational cost. This motivates our second algorithm, which we describe in the next section.

Consider every pair of cliques that share at least one node. We do not need to perform an intersection test for every such pair – we can potentially avoid testing most pairs of cliques which percolate into the same community. But we cannot avoid testing the pairs of cliques which share a node, but never percolate into the same community. For each network, and for each value of k, we count the pairs of cliques which share at least one node and which do not percolate, and divide this by the necessary number of successful tests. Some of the highest ratios are shown in Table II. Even though Algorithm 1 is much faster than SCP for higher k, this is a computational bottleneck. Each failed intersection test is akin to a 'false positive'; we tested the cliques that shared a node, in case they shared k-1, but this turned out not to be the case.

University	k	Failed tests	Successful tests	Ratio
Berkeley13	4	10,466,831	1,888,501	5.5424
UC33	5	36,722,163	1,396,585	26.294
JMU79	6	87,790,729	1,007,912	87.101
JMU79	7	202,207,552	942,903	214.45
Baylor93	8	7,979,454,075	5,667,415	1407.9
JMU79	9	2,639,954,200	852,264	3097.5
JMU79	10	4,148,901,179	817,621	5074.3
UC61	11	5,257,215,707	948,764	5541.1
Bingham82	12	3,490,950,418	511,790	6821.0
UC61	13	6,271,564,288	849,242	7384.8
Rutgers89	14	2,988,388,319	386,558	7730.7
Rutgers89	15	2,388,357,546	339,698	7030.8
MIT8	16	3,941,888,908	322,212	12233.
MIT8	17	3,183,156,718	285,917	11133.
Howard90	18	3,675,540,181	148,661	24724.
Howard90	19	3,235,463,119	138,158	23418.
UC61	20	15,499,684,499	582,658	26601.

TABLE II
USING THE FACEBOOK100 NETWORKS. FOR EACH K BETWEEN 4 AND 20, THE UNIVERSITY WITH THE HIGHEST RATIO OF FAILED INTERSECTION TESTS TO THE NUMBER OF SUCCESSFUL INTERSECTION TESTS IS SHOWN.

E. Algorithm 2

Our second algorithm uses the same framework described in Section IV-A. The only change is that we have a faster way, given the current frontier clique, of finding its unvisited neighbouring cliques: a cheaper way of avoiding intersection tests, by detecting when two cliques will percolate in way that gives fewer 'false positives' than checking only whether they share a single node.

We use a complete binary tree where there are as many leaf nodes as there are maximal cliques. We will refer to the nodes in this binary tree as *tree-nodes*, and the tree-nodes which are at the leaves of our tree will be referred to as *leaf-nodes*. To avoid confusion, we use *graph-node* to refer to the nodes in our original network. Each tree-node will have a set of graph-nodes associated with it. For the leaf-nodes, this will be the set of graph-nodes that are in the maximal clique which we associated with that leaf-node. For the other tree-nodes in the binary tree, the set of graph-nodes will be the union of the graph-nodes associated with the leaf-nodes descended from it.

This tree is built once at the start of the algorithm and does not change. But each leaf-node has a boolean field associated with it which records whether the associated clique has already been percolated into a community (i.e. it has been *visited*). These *visited* fields will be initialized to False at the beginning, and will all gradually be set True as the algorithm proceeds and visits the cliques. For the other tree-nodes, this boolean field will be set to True when the fields of its children have been set to True. This means that a boolean field will be True if and only if all of its descendants have already been visited.

University	Nodes	Edges	Time(s)
Caltech36	769	16,656	50.94
Reed98	962	18,812	51.84
Simmons81	1,518	32,988	115.10
Haverford76	1,446	59,589	372.39
Swarthmore42	1,659	61,050	235.93
USFCA72	2,682	65,252	215.22
Mich67	3,748	81,903	421.19
Bowdoin47	2,252	84,387	246.39
Oberlin44	2,920	89,912	263.45
Amherst41	2,235	90,954	424.96
UMass92	16,516	519,385	18,544.87
UConn91	17,212	604,870	27,875.35
UPenn7	14,916	686,501	26,413.13
UNC28	18,163	766,800	87,315.59
USC35	17,444	801,853	84,481.39

TABLE III
THE RUNTIMES OF OUR MORE OPTIMIZED ALGORITHM, TO COMPUTE ALL VALUES OF k, ON THE SMALLEST 10 FACEBOOK NETWORKS, AND 5 LARGER ONES. THIS ALGORITHM WAS OFTEN ABLE TO COMPLETE ALL VALUES OF k IN LESS TIME THAN ALGORITHM 1 TOOK TO COMPLETE $k = 4$. FOR EVERY DATASET TESTED, THIS ALGORITHM CALCULATES ANY VALUE OF k FASTER THAN ALGORITHM 1.

Given the current frontier clique, we could visit all the leaf nodes and perform an intersection test against each. This would be the naive algorithm again. But we can ignore cliques that have been visited already. In particular, if a tree-node has been marked as visited, then we can ignore all the cliques that are descended from it. Also, given a particular tree-node we can test the intersection between the current frontier clique and the set of graph-nodes which are stored at this tree-node. If this intersection is less than k-1, then we know that there cannot be a neighbouring clique among the cliques descended from this tree-node. These two observations allow us to skip most of the intersection tests via the information present in the binary tree.

The search is a recursive search which starts at the root-node of the binary tree.

At each tree-node, we check if this tree-node has been marked as visited, and also check the size of the intersection. If the intersection is large enough and if the tree-node is still unvisited, then the search will proceed recursively into each of the two children of the tree-node. Otherwise, the recursive function will return and that tree-node (and all its descendants) will not be considered any further.

To save memory, we do not explicitly maintain a set of graph-nodes at each non-leaf tree-node. Instead we use a Bloom filter – a fast and memory-efficient structure which probabilistically records whether a given element is a member of a set. It sometimes produces false positives, which in our algorithm would sometimes lead to an overestimate of the size of the intersection between two cliques; this does not cause a correctness problem, but does mean that sometimes the recursive search proceeds slightly deeper than necessary.

When considering a single value of k, Algorithm 2 was always faster than Algorithm 1, except sometimes when $k = 3$. In Table III, we see the runtimes of Algorithm 2 on a number of Facebook networks. These runtimes are the time taken to compute the communities for *all* values of k. This algorithm frequently computes all values of k quicker than Algorithm 1 can compute the communities for $k = 4$.

V. CONCLUSION

We have examined k-clique percolation as a specific example of a computationally challenging percolation problem. We have shown that vast numbers of cliques exist in empirical social networks, and that k-clique percolation is a hard problem to implement well, due to the difficulty of producing intermediate representations of percolating structures. From the large number of overlapping cliques and maximal cliques that we observe in empirical networks, and the consequent large number of edges in the clique graphs we have constructed, we conclude that while clique graphs are an attractive conceptual tool, their utility is limited in applications of social network analysis. We have developed and thoroughly benchmarked a k-clique percolation algorithm that is conceptually simple, yet performs better than existing methods on many real world networks, especially when considering k-clique percolation with higher values of k. This method is challenged by the large numbers of cliques which share at least one node, but do not percolate. We introduce a second method which uses a more sophisticated data structure. With this method we can conduct clique percolation on larger pervasively overlapping networks than ever before. However, these methods remain fundamentally limited by the necessity of testing cliques against other cliques with which they share *some* nodes, but do not percolate. Given the number of cliques, clique percolation remains computationally challenging; other overlapping community detection methods appear more promising, from a computational standpoint. We provide software for researchers studying other percolation problems to leverage our results.

REFERENCES

[1] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[2] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, pp. 75–174, 2010.

[3] G. Palla, A. Barabasi, and T. Vicsek, "Quantifying social group evolution," *Nature*, vol. 446, no. 7136, pp. 664–667, 2007.

[4] T. Evans, "Clique graphs and overlapping communities," *Journal of Statistical Mechanics: Theory and Experiment*, p. P12037, 2010.

[5] M. Everett and S. Borgatti, "Analyzing clique overlap," *Connections*, vol. 21, no. 1, pp. 49–61, 1998.

[6] B. Adamcsek, G. Palla, I. Farkas, I. Derényi, and T. Vicsek, "CFinder: locating cliques and overlapping modules in biological networks," *Bioinformatics*, vol. 22, no. 8, p. 1021, 2006.

[7] J. Kumpula, M. Kivelä, K. Kaski, and J. Saramäki, "Sequential algorithm for fast clique percolation," *Physical Review E*, vol. 78, no. 2, p. 026109, 2008.

[8] A. Traud, E. Kelsic, P. Mucha, and M. Porter, "Community structure in online collegiate social networks," *arXiv*, vol. 809, 2008.

[9] R. Luce and A. Perry, "A method of matrix analysis of group structure," *Psychometrika*, vol. 14, no. 2, pp. 95–116, 1949.

[10] Y. Ahn, J. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, vol. 466, no. 7307, pp. 761–764, 2010.

[11] F. Reid, A. McDaid, and N. Hurley, "Partitioning Breaks Communities," *2011 Advances in Social Network Analysis and Mining*, 2011.

[12] C. Bron and J. Kerbosch, "Finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[13] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[14] J. Leskovec, "SNAP Stanford Network Analysis Project," http://snap.stanford.edu, 2009.