# An Algorithm to Find Overlapping Community Structure in Networks

Steve Gregory

Department of Computer Science
University of Bristol, BS8 1UB, England
steve@cs.bris.ac.uk

**Abstract.** Recent years have seen the development of many graph clustering algorithms, which can identify community structure in networks. The vast majority of these only find disjoint communities, but in many real-world networks communities overlap to some extent. We present a new algorithm for discovering overlapping communities in networks, by extending Girvan and Newman's well-known algorithm based on the *betweenness* centrality measure. Like the original algorithm, ours performs hierarchical clustering — partitioning a network into any desired number of clusters — but allows them to overlap. Experiments confirm good performance on randomly generated networks based on a known overlapping community structure, and interesting results have also been obtained on a range of real-world networks.

## 1 Introduction and Motivation

Many complex systems in the real world can be represented abstractly as networks (or graphs). Recently, with increasing availability of data about large networks and the need to understand them, the study of networks has become an important topic. A property that has been extensively studied is the existence of community structure in networks. A *cluster* (or *community* or *module*) is a subgraph such that the density of edges within it (*intracluster edges*) is greater than the density of edges between its vertices and those outside it (*intercluster edges*). A wide range of algorithms have been developed to discover communities in a network, including [4, 6, 11, 12, 13, 14].

Probably the best-known algorithm for finding community structure is Girvan and Newman's algorithm [6, 14], based on the betweenness centrality measure [5]. The *betweenness* (strictly, the *shortest-path betweenness*) of edge $e$, $c_B(e)$, is defined as the number of shortest paths, between all pairs of vertices, that pass along $e$. A high betweenness means that the edge acts as a bottleneck between a large number of vertex pairs and suggests that it is an intercluster edge. Although the algorithm is quite slow and is no longer the most effective clustering algorithm, it does give relatively good results. The algorithm works as follows:

1. Calculate edge betweenness of all edges in network.
2. Find edge with highest edge betweenness and remove it.
3. Recalculate edge betweenness for all remaining edges.
4. Repeat from step 2 until no edges remain.

J.N. Kok et al. (Eds.): PKDD 2007, LNAI 4702, pp. 91–102, 2007.

This is a hierarchical, divisive, clustering algorithm. Initially, the $n$-vertex network (if connected) forms a single cluster. After one or more iterations, removing an edge (step 2) causes the network to split into two components (clusters). As further edges are removed, each component again splits, until $n$ singleton clusters remain. The result is a *dendrogram*: a binary tree in which the distance of nodes from the root shows the order in which clusters were split. A cross-section of the dendrogram at any level represents a division of the network into any desired number of clusters.

In step 3, edge betweenness need not be recalculated for the whole network, but only for the component containing the edge removed in step 2, or for both components if removing the edge caused the component to split. (The edge betweenness of an edge depends only on the vertices and edges in the same component as it.)

Most algorithms assume that communities are disjoint, placing each vertex in only one cluster. However, in the real world, communities often overlap. For example, in collaboration networks an author might work with researchers in many groups, in biological networks a protein might interact with many groups of proteins, and so on.

In this paper we present a new algorithm to find overlapping community structure in networks. It is a hierarchical, divisive algorithm, based on Girvan and Newman's but extended with a novel method of splitting vertices. We describe the design of the algorithm in Section 2. In Section 3 we present some results on both artificial (computer-generated) and real-world networks. Section 4 compares our algorithm with a few others that can detect overlapping communities. Conclusions appear in Section 5.

## 2 Finding Overlapping Clusters

In any divisive hierarchical clustering algorithm, clusters are repeatedly divided into smaller (normally disjoint) clusters that together contain the same items. To allow overlapping clusters, there needs to be some way of splitting (copying) an item so that it can be included in more than one cluster when the cluster divides.

In the context of network clustering, assuming it is based entirely on the network structure, it seems reasonable to assume that each vertex should be in the same cluster as at least one of its neighbours, unless it is in a singleton cluster or no cluster at all. Therefore, a vertex $v$ should be split into at most $d(v)$ copies, where $d(v)$ is the degree of $v$. We need to decide how many times a vertex should be split, and when a vertex should be split (e.g., at the beginning or when dividing a cluster).

Our algorithm extends Girvan and Newman's algorithm (the "GN algorithm") with a specific method of deciding *when* and *how* to split vertices. As in the original work, we only consider unipartite networks with undirected, unweighted edges. We name our new algorithm "CONGA" (Cluster-Overlap Newman Girvan Algorithm).

**Splitting Vertices.** In the GN algorithm, the basic operation is removing an edge. We introduce a second operation: splitting a vertex. If split, a vertex $v$ always splits into *two* vertices $v_1$ and $v_2$: edges with $v$ as an endvertex are redirected to $v_1$ or $v_2$ such that $v_1$ and $v_2$ each has at least one edge. By splitting repeatedly, a vertex $v$ can eventually split into at most $d(v)$ vertices. Vertices are split incrementally during the clustering process. This binary splitting fits well into the GN algorithm because, like removing an edge, splitting a vertex may cause its cluster to split into two.

**Split Betweenness.** The key point of the CONGA algorithm is the notion of "split betweenness". This provides a way to decide (1) *when* to split a vertex, instead of removing an edge, (2) *which* vertex to split, and (3) *how* to split it. Clearly, $v$ should only be split into $v_1$ and $v_2$ if these two vertices belong to different clusters. We could verify this by counting the number of shortest paths that would pass between $v_1$ and $v_2$ *if* they were joined by an edge. Then, if there were more shortest paths on $\{v_1,v_2\}$ than on any real edge, the vertex should be split; otherwise, an edge should be removed as usual. This is the basis of our method of splitting a vertex, which is as follows.

For any split of vertex $v$ into $v_1$ and $v_2$, we add a new "imaginary" edge between $v_1$ and $v_2$. If $u$ is a neighbour of $v_1$ and $w$ is a neighbour of $v_2$, all shortest paths that passed through $v$ along edges $\{u,v\}$, $\{v,w\}$ now pass along $\{u,v_1\}$, $\{v_1,v_2\}$, $\{v_2,w\}$. The imaginary edge has zero cost: the lengths of paths traversing it are unchanged, and no new shortest paths are created: paths beginning from $v$ do not traverse this edge. We then calculate the betweenness $c_B(\{v_1,v_2\})$ of the imaginary edge. In general, there are $2^{d(v)-1}-1$ ways to split $v$ into two. We call the split that maximizes $c_B(\{v_1,v_2\})$ the *best split* of $v$, and the maximum value of $c_B(\{v_1,v_2\})$ the *split betweenness* of $v$.

We modify the GN algorithm so that, at each step, it considers the split betweenness of every vertex as well as the edge betweenness of every edge. If the maximum split betweenness is greater than the maximum edge betweenness, the corresponding vertex is split, using its best split. (Note that imaginary edges are never actually added to the network, but are used only during the calculation of the split betweenness.)

Fig. 1(a) shows a network comprising two overlapping clusters: $\{a,b,c\}$ and $\{a,d,e\}$. Labels on the edges show edge betweennesses (with shortest paths counted in both directions). Fig. 1(b) shows $a$'s best split into $a_{bc}$ and $a_{de}$, with the imaginary edge (betweenness 8) shown as a dashed line. Fig. 1(c-d) shows the other two possible splits of $a$. In these, the imaginary edge has a lower betweenness, 4, proving that the split of Fig. 1(b) is the best split and the split betweenness of $a$ is 8. Because this is greater than any edge betweenness, $a$ should indeed be split.

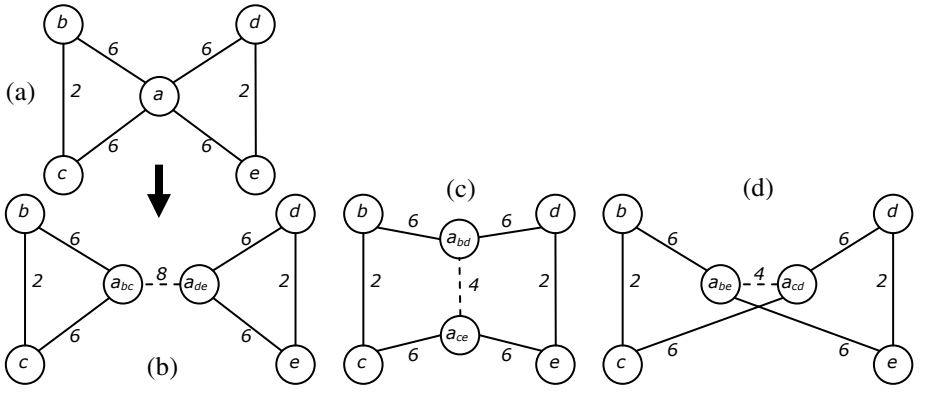


**Fig. 1.** (a) Network. (b) Best split of vertex $a$. (c), (d) Other splits of vertex $a$.

Fig. 2 shows a network which does not exhibit clustering. Here, any (2+2) split of $a$ is a best split. The split betweenness of $a$ is 8, which is the same as the betweenness of each edge. Therefore, by default, we remove any edge instead of splitting $a$.

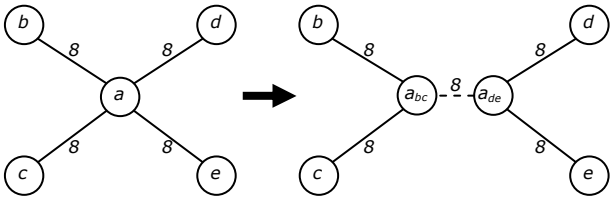


**Fig. 2.** Best split of vertex $a$: split betweenness of $a$ is 8

Our method will never split a vertex into $v_1$ and $v_2$ such that $v_1$ has only one neighbour, $u$. This is because the betweenness of $\{v_1,v_2\}$ would be the same as that of $\{u,v_1\}$, as shown in Fig. 3, so removing edge $\{u,v\}$ would be preferred over splitting $v$. As a consequence of this, vertices with degree less than 4 are never split. In general, there are now only $2^{d(v)-1}$-$d(v)$-1 ways to split a vertex into two.

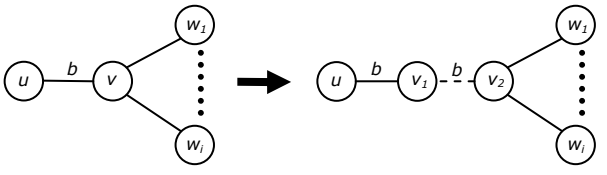


**Fig. 3.** Vertex will not split into vertices with degree 1

**Vertex Betweenness and Split Betweenness.** The split betweenness of a vertex $v$ is the number of shortest paths passing between any member of $n_1$ and any member of $n_2$ via $v$, where $n_1$ and $n_2$ are disjoint sets containing all neighbours of $v$. By definition, this is no greater than the total number of shortest paths passing through $v$: the *vertex betweenness* of $v$ [5]. It is simple to calculate vertex betweenness $c_B(v)$ from edge betweenness $c_B(e)$ [7]:

$$c_B(v) = \frac{1}{2}\sum_{e\in\Gamma(v)} c_B(e) - (n-1) \tag{1}$$

where $\Gamma(v)$ is the set of edges with $v$ as an endvertex and $n$ is the number of vertices in the component containing $v$. Therefore, as an optimization, we can use vertex betweenness as an upper bound on split betweenness: if the vertex betweenness of $v$ is no greater than the maximum edge betweenness, there is no need to calculate $v$'s split betweenness.

**Calculating Split Betweenness.** To calculate the split betweenness, and best split, of a vertex $v$, we first compute the pair betweennesses of $v$. The *pair betweenness* of $v$ for $\{u,w\}$, where $u$ and $w$ are neighbours of $v$ and $u \neq w$, is the number of shortest paths that traverse both edges $\{u,v\}$ and $\{v,w\}$. The vertex betweenness of $v$ is the sum of all of its pair betweennesses.

We can represent the pair betweennesses of $v$, degree $k$, by a $k$-clique in which each vertex is labelled by one of $v$'s neighbours and each edge $\{u,w\}$ is labelled by the pair betweenness "score" of $v$ for $\{u,w\}$. Then, to find the best split of $v$:

1. Choose edge $\{u,w\}$ with minimum score.
2. Coalesce $u$ and $w$ to a single vertex, $uw$.
3. For each vertex $x$ in the clique, replace edges $\{u,x\}$, score $b_1$, and $\{w,x\}$, score $b_2$, by a new edge $\{uw,x\}$ with score $b_1+b_2$.
4. Repeat from step 1 $k$-2 times (in total).

The labels on the remaining two vertices show the split, and the score on the remaining edge is the split betweenness.

This algorithm is not guaranteed to find the best split. To do that, we would need to try *all* edges in step 1 of each iteration, which would require exponential time. Our "greedy" method is much more efficient and, in practice, usually finds the best split or a close approximation to it. Fig. 4 shows how it finds the best split of vertex $a$ of Fig. 1. There are $k$-2 = 2 phases; the edge chosen in step 1 of each phase is highlighted.

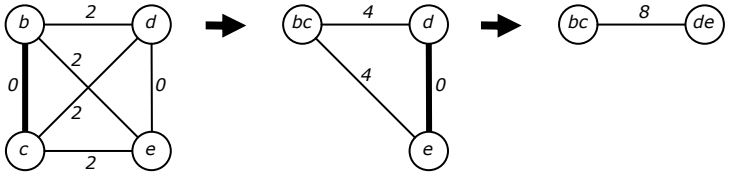


**Fig. 4.** Finding the best split of vertex $a$ of Fig. 1

**Calculating Pair Betweennesses.** Pair betweennesses are computed while calculating edge betweenness, by a straightforward modification of the GN algorithm. The GN algorithm increments the betweenness of edge $\{i,j\}$ for all shortest paths beginning at each vertex $s$. CONGA does this *and* increments the pair betweennesses of $i$ for all pairs $\{j,k\}$ such that $k$ is a neighbour of $i$ on a path between $s$ and $i$.

There is some overhead, in both time and space, in computing pair betweennesses during the betweenness calculation. In most cases this information is not used because we can often determine, from the vertex betweenness, that a vertex should not be split. Therefore, our betweenness calculation is split into two phases, as shown below.

**The CONGA Algorithm.** Our complete algorithm is as follows:

1. Calculate edge betweenness of all edges in network.
2. Calculate vertex betweenness of vertices, from edge betweennesses, using Eq. (1).
3. Find candidate set of vertices: those whose vertex betweenness is greater than the maximum edge betweenness.
4. If candidate set is non-empty, calculate pair betweennesses of candidate vertices, and then calculate split betweenness of candidate vertices, using Eq. (1).
5. Remove edge with maximum edge betweenness or split vertex with maximum split betweenness (if greater).
6. Recalculate edge betweenness for all remaining edges in same component(s) as removed edge or split vertex.
7. Repeat from step 2 until no edges remain.

**Complexity and Efficiency.** The GN algorithm has a worst-case time complexity of $O(m^2n)$, where $m$ is the number of edges and $n$ is the number of vertices. In CONGA, each vertex splits into an average of up to $2m/n$ vertices, so the number of vertices after splitting is $O(m)$; the number of iterations is still $O(m)$ and the number of edges is unchanged. This makes the time complexity $O(m^3)$ in the worst case.

In practice, the speed depends on the number of vertices that are split. If more are split, more iterations are needed, the network becomes larger, and step 4 needs to be performed more frequently. Conversely, vertex splitting can cause the network to decompose into separate components more readily, which reduces the execution time.

## 3 Results

In this section we compare CONGA with the GN algorithm, to assess the effect of our extensions. We have tested both algorithms on computer-generated networks based on a known, possibly overlapping, community structure. Each network contains $n$ vertices divided into $c$ equally-sized communities, each containing $nr/c$ vertices. Vertices are randomly and evenly distributed between communities such that each vertex is a member of $r$ ($\geq 1$) communities on average. Edges are randomly placed between pairs of vertices with probability $p_{in}$ if the vertices belong to the same community and $p_{out}$ otherwise. In the special case where both *r and* $p_{out}$ are 0, the network will be disconnected. Apart from this, all of our experiments use connected networks, constructed with a sufficiently high value of $r$ or $p_{out}$, or both.

We measure how well each algorithm can recover the community structure from a network by using it to compute $c$ clusters and comparing the result with the $c$ known communities. Admittedly, $c$ is not generally known for real-world networks, but this is still a useful and common way to assess clustering algorithms; e.g., [6, 14].

We calculate two values (all averaged over 10 graphs):

- *recall*: the fraction of vertex pairs belonging to the same community that are also in the same cluster.
- *precision*: the fraction of vertex pairs in the same cluster that also belong to the same community.

First (Fig. 5), we generated networks of 256 vertices divided into 32 communities, set $p_{out} = 0$ (i.e., no intercommunity edges) and $p_{in} = 0.5$, and increased the amount of overlap from $r = 1$ (i.e., no overlap) to $r = 3$. The number of edges (and hence the average degree) increases roughly quadratically with $r$, because the average community size is proportional to *r and* each vertex is a member of $r$ communities. So the average degree is 4 for $r = 1$ but increases to approximately 15 for $r = 2$ and 32 for $r = 3$.

For the GN algorithm, as $r$ increases, recall declines steadily because the (non-overlapping) clusters are smaller than the communities; precision is quite high, though certainly not perfect, in this range. Suddenly, at around $r = 2$, recall increases and precision decreases, as most vertices are placed in a single cluster. In contrast, CONGA behaves very well up to about $r = 2$ and then deteriorates gradually.

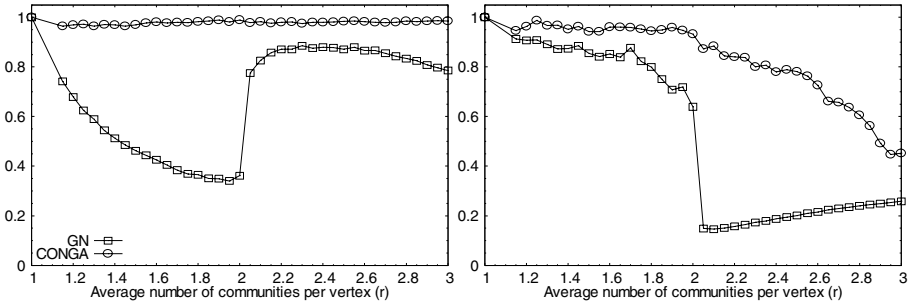


**Fig. 5.** recall (left), precision (right): $n$=256, $c$=32, $p_{in}$=0.5, $p_{out}$=0, various $r$

We have repeated this experiment for various values of $c$ and $p_{in}$. The curves always show a similar shape, though the value of $r$ at which precision drops varies.

To evaluate the algorithm on real-world networks, there is no correct solution with which to compare, so the quality of a clustering must be assessed in a different way. This is usually done by measuring the relative number of intracluster and intercluster edges, for example, by the *modularity* measure [13, 14]. However, there is no widely accepted alternative measure for use with overlapping clusters, but a promising candidate is the *average degree* measure [3]. We define the *vertex average degree* (*vad*) of a set of clusters $S$, where each cluster is a set of vertices, as:

$$vad(S) = \frac{2\sum_{C \in S} |E(C)|}{\sum_{C \in S} |C|} \tag{2}$$

Another useful measure is the *overlap* of a set of clusters $S$: the sum of the cluster sizes divided by the size of the union of all clusters. (We do not claim that *vad* and overlap are mutually independent measures; that is outside the scope of this paper.)

We have run the CONGA and GN algorithms on several real-world examples, listed in Table 1. Execution times are shown for a 2.4GHz Pentium 4 processor.

**Table 1.** Algorithm's results on real-world networks

| Name | Ref. | Vertices | Edges | Runtime (s) |
|---|---|---|---|---|
| Karate club | [19] | 34 | 78 | 0.2 |
| Dolphins | [9] | 62 | 159 | 0.5 |
| College football | [6] | 115 | 613 | 7.8 |
| Network science | [12] | 379 | 914 | 12.5 |
| Blogs | [18] | 3982 | 6803 | 30411 |
| Words | [10] | 1000 | 3471 | 6767 |

"Karate club" [19], discussed in [6], represents a social network based on two disjoint communities. The communities are not reflected clearly in the network structure: there are eight intercommunity edges. GN finds an almost perfect (relative to the real-world situation) two-cluster solution, misclassifying one vertex. CONGA finds a

different solution with a small overlap, 1.03. The *vad* is 4.45 for CONGA and 4.0 for GN, suggesting that the overlapping clustering is a good one (albeit incorrect).

"Dolphins" [9], discussed in [14], is a social network of dolphins, also based on two disjoint communities. Here there are only six intercommunity edges. GN finds the two communities correctly and CONGA finds the same division but with two vertices from the larger community included in both clusters: the overlap is 1.03. The *vad* is 4.91 for CONGA and 4.94 for GN.

"College football" [6] is a network based on games between teams that belong to 15 disjoint real-world communities. This network has many intercommunity edges. Neither algorithm finds a perfect 15-cluster solution; the one found by CONGA has a lower *vad* (5.87 *vs*. 7.18) and a large overlap: 1.75.

"Network science" [12] is a collaboration network of coauthorships. For such networks it is impossible to determine the number of real-world communities, and it seems reasonable to assume they might overlap. CONGA's solution has a higher *vad* than GN's for 14 or more clusters, and overlap increases with the number of clusters. CONGA's solution for 33 clusters is illustrated in Fig. 6: each cluster is identified by a letter or digit and each vertex is labelled with the cluster(s) to which it belongs.

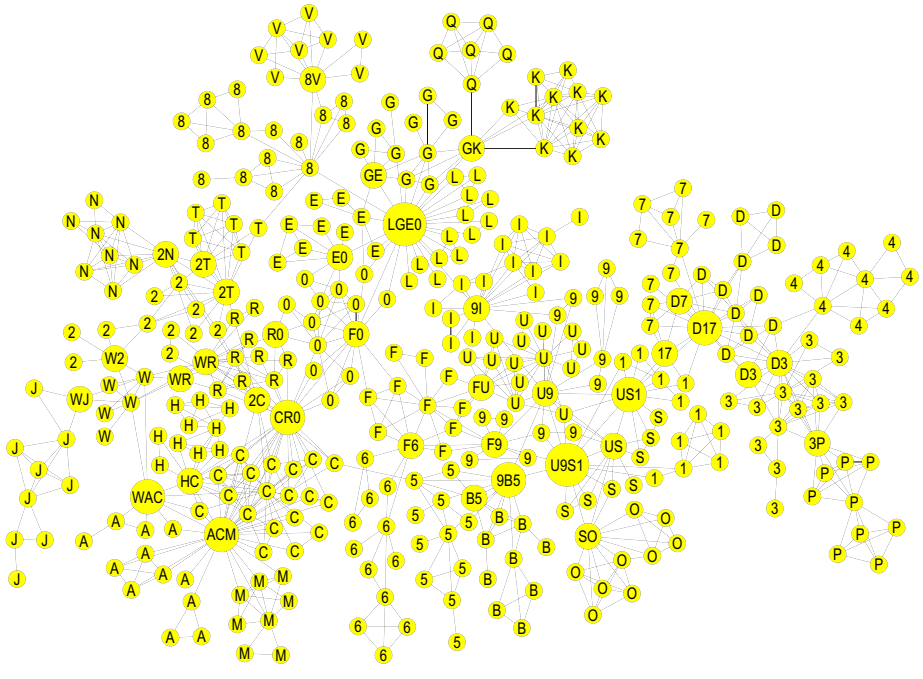


**Fig. 6.** Network science collaboration network divided into 33 overlapping clusters

"Blogs" [18] is a network of blogs on the MSN (now known as Windows Live™ Spaces) platform. An edge links people who have left two or more comments on each other's blog, and so are deemed to be acquainted. CONGA's solution has a consistently higher *vad* than GN's, especially for more than 90 clusters. The overlap increases with the number of clusters but levels off, reaching a maximum of 1.39.

"Words" is a non-social network: a contiguous 1000-vertex subgraph of a word association network from [15], converted from an original directed, weighted version [10]. CONGA successfully groups related words. For example, dividing it into 400 clusters, the word "form" appears in four: {contract, document, form, order, paper, signature, write}, {blank, entry, fill, form, up}, {compose, create, form, make}, {form, mold, shape}. (Related words in this network are not necessarily synonyms, as they are in this example.) Again, the *vad* for CONGA's solution is consistently higher than GN's; the overlap increases and tails off, reaching a maximum of 2.23.

## 4 Related Work

Pinney and Westhead [16, 17] have also proposed extending the GN algorithm with the ability to split vertices between clusters. The decision of whether to split a vertex or remove an edge is based entirely on edge betweenness and vertex betweenness. The highest-betweenness edge is removed only if its two endvertices have similar betweenness; i.e., if their ratio is between $\alpha$ and $1/\alpha$, where $\alpha$ is a parameter with suggested value 0.8 [16]. Otherwise the vertex with highest betweenness is temporarily removed. When a component splits into two or more subcomponents, each removed vertex is split and copied into each subcomponent, and all edges between the vertex copy and the subcomponent are restored, including any removed in previous steps. We have implemented this algorithm and compared it with CONGA; see below.

The clique percolation algorithm of Palla *et al.* [15], implemented in CFinder [1], finds overlapping clusters in a different way. Instead of dividing a network into its most loosely connected parts, it identifies the most densely connected parts. The parameter is not the number of clusters to be found but their density, $k$. A cluster is defined as the set of $k$-cliques that can all be reached from each other via a sequence of adjacent $k$-cliques; two $k$-cliques are *adjacent* if they share $k$-1 vertices. Each vertex may be in many clusters, or even none: e.g., degree-1 vertices are always ignored. We have run CFinder (v1.21) to compare its results with CONGA's; see below.

Baumes *et al.* [2, 3] present a collection of algorithms to find overlapping clusters. One algorithm iteratively improves a candidate cluster by adding vertices to and removing vertices from it while its density improves. Another removes vertices from a network until it breaks into disjoint components, forming initial clusters, and then replaces each removed vertex into one or more of the clusters, which might overlap.

Li *et al.* [8] form overlapping clusters using both the structure of the network *and* the content of vertices and edges. The first phase of their algorithm finds densely connected "community cores", similarly to the method of [15]. In the second phase, clusters are formed from cores by adding further triangles and edges whose content (assessed using keywords) is similar to that of the core.

**Experiments.** We have run the Pinney and Westhead ("P&W") and CFinder algorithms on computer-generated networks, to compare with CONGA. The number of communities $c$ was input to both CONGA and P&W, but CFinder cannot make use of this information, so CFinder is clearly disadvantaged. To compensate for this, we show the CFinder results for all values of $k$ (CFinder's only parameter). For each experiment we plot the F-measure: the harmonic mean of recall and precision.

Fig. 7(a) shows results on the networks of Section 3: $p_{in}$ and $p_{out}$ are fixed while $r$ is varied. CONGA gives the best results of all algorithms tested, but performance declines for all algorithms for high $r$. CFinder gives its best performance for $r$=2, so in fairness to CFinder we use this value in subsequent experiments. In Fig. 7(b) we fix $r$ and $p_{out}$ and vary $p_{in}$. CONGA gives the best results, and they improve as $p_{in}$ increases. In contrast, CFinder, for each $k$, reaches a peak at a different value of $p_{in}$; for smaller values its recall is reduced while for larger values its precision drops.

In Fig. 7(c) we fix $r$ and $p_{in}$ and vary $p_{out}$. This time, CONGA's performance suffers as $p_{out}$ increases, because of reduced precision, while CFinder's performance is more stable. Finally, in Fig. 7(d), we test the hypothesis that CFinder should be more effective in cases where the number of communities is not known. We do this by generating networks in which a (varying) number, $u$, of the 256 individuals are placed in singleton communities and the remainder are divided between the 32 main communities; because $p_{out}$>0 these networks are still connected. In this experiment, CFinder with $k$=4 performs slightly better than CONGA. For both algorithms, recall decreases as $u$ increases but CFinder's precision improves while CONGA's declines.

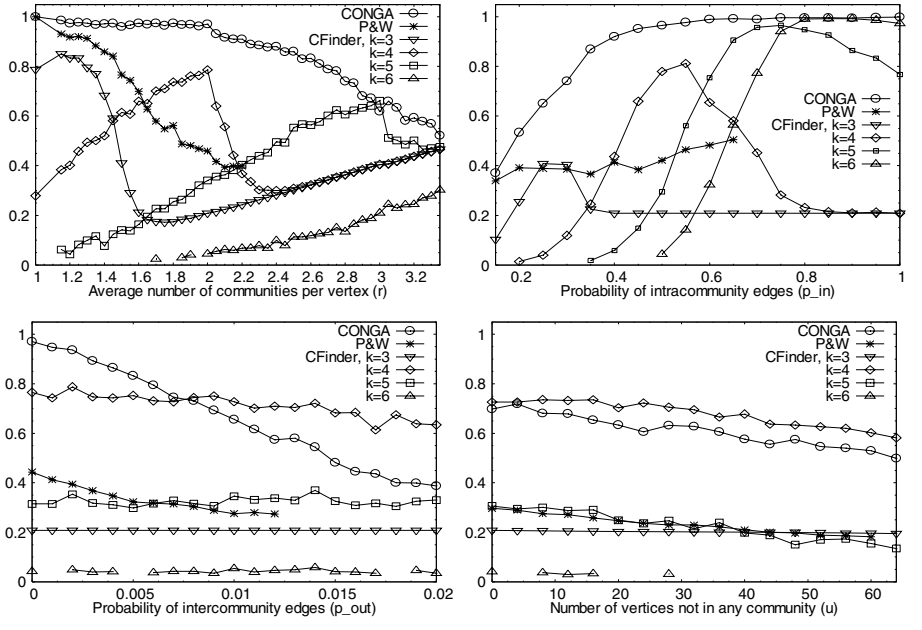


**Fig. 7.** F-measure for random networks with $n$=256, $c$=32. (a: upper left) $p_{in}$=0.5, $p_{out}$=0, various $r$; (b: upper right) $r$=2, $p_{out}$=0, various $p_{in}$; (c: lower left) $r$=2, $p_{in}$=0.5, various $p_{out}$; (d: lower right) $r$=2, $p_{in}$=0.5, $p_{out}$=0.008, various $u$.

Fig. 8 shows the execution times of all algorithms for the experiments of Fig. 7(a). For CONGA and P&W these times include the generation of the complete dendrogram, from which the solution for any number of clusters can be quickly extracted. The process is not stopped after the network is divided into 32 clusters. For CFinder,

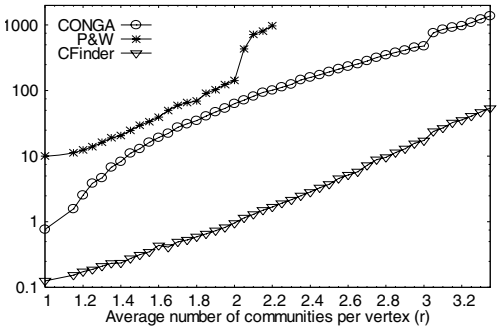


**Fig. 8.** Execution time (seconds) for $n$=256, $c$=32, $p_{in}$=0.5, $p_{out}$=0, various $r$

the times include the generation of solutions for all values of $k$. CONGA and P&W were implemented by the author in Java. Each experiment was run on a machine with dual AMD Opteron 250 CPUs (2.4GHz).

In summary, CONGA and CFinder seem to have complementary strengths and weaknesses: each may be better for a different application. CFinder is substantially faster than CONGA. P&W behaves in a similar way to CONGA but with worse results (for these networks); however, we have only tested it with one value of its parameter (α). The execution time of P&W is also the worst, but this may be because of the poor implementation rather than the algorithm itself.

## 5 Conclusions

We have presented an algorithm that seems to be effective in discovering overlapping communities in networks. Good results have been obtained for a range of random networks with overlap of more than 2, which is large relative to the number of communities: if a network has only 32 communities, an overlap of 3 means that each vertex is in the same community as ¼ of the whole network. As the number of communities is increased, the algorithm can cope with a larger overlap. The algorithm is not fast, but its speed is comparable with that of the GN algorithm from which it is derived.

Future work includes trying to improve the algorithm further and applying similar ideas to faster clustering algorithms than the GN algorithm. It is also worth investigating alternative ways of measuring the quality of an overlapping clustering; e.g., the *vad* measure. Finally, it would be interesting to study the overlapping nature of real-world networks, a subject that has received little attention (but see [15]). For example, it may be that the collaboration network of Fig. 6 naturally divides into a small number of disjoint clusters, possibly corresponding to research groups, but to decompose it further requires clusters to overlap.

Further information related to this paper, including the networks analysed and more results, can be found at http://www.cs.bris.ac.uk/~steve/networks/ .

**Acknowledgements.** I am very grateful to Peter Flach for his expert advice on several drafts of this paper. Thanks are also due to John Pinney for explaining his algorithm, and the four anonymous referees for their detailed comments.

## References


1. Adamcsek, B., Palla, G., Farkas, I., Derényi, I., Vicsek, T.: CFinder: locating cliques and overlapping modules in biological networks. Bioinformatics 22, 1021–1023 (2006)
2. Baumes, J., Goldberg, M., Krishnamoorty, M., Magdon-Ismail, M., Preston, N.: Finding communities by clustering a graph into overlapping subgraphs. In: Proc. IADIS Applied Computing 2005, pp. 97–104 (2005)
3. Baumes, J., Goldberg, M., Magdon-Ismail, M.: Efficient identification of overlapping communities. In: Kantor, P., Muresan, G., Roberts, F., Zeng, D.D., Wang, F.-Y., Chen, H., Merkle, R.C. (eds.) ISI 2005. LNCS, vol. 3495, pp. 27–36. Springer, Heidelberg (2005)
4. Brandes, U., Gaertler, M., Wagner, D.: Experiments on graph clustering algorithms. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 568–579. Springer, Heidelberg (2003)
5. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry 40, 35–41 (1977)
6. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. Natl. Acad. Sci. USA 99, 7821–7826 (2002)
7. Koschützki, D., Lehmann, K.A., Peeters, L., Richter, S., Tenfelde-Podehl, D., Zlotowski, O.: Centrality indices. In: Brandes, U., Erlebach, T. (eds.) Network Analysis. LNCS, vol. 3418, Springer, Heidelberg (2005)
8. Li, X., Liu, B., Yu, P.S.: Discovering overlapping communities of named entities. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 593–600. Springer, Heidelberg (2006)
9. Lusseau, D., Schneider, K., Boisseau, O.J., Haase, P., Slooten, E., Dawson, S.M.: The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. Behavioral Ecology and Sociobiology 54, 396–405 (2003)
10. Nelson, D.L., McEvoy, C.L., Schreiber, T.A.: The University of South Florida word association, rhyme and word fragment norms (1998), http://w3.usf.edu/FreeAssociation/
11. Newman, M.E.J.: Fast algorithm for detecting community structure in networks. Phys. Rev. E 69, 066133 (2004)
12. Newman, M.E.J.: Finding community structure in networks using the eigenvectors of matrices. Phys. Rev. E 74, 036104 (2006)
13. Newman, M.E.J.: Modularity and community structure in networks. Proc. Natl. Acad. Sci. USA 103, 8577–8582 (2006)
14. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113 (2004)
15. Palla, G., Derényi, I., Farkas, I., Vicsek, T.: Uncovering the overlapping community structure of complex networks in nature and society. Nature 435, 814–818 (2005)
16. Pinney, J.W.: Personal communication
17. Pinney, J.W., Westhead, D.R.: Betweenness-based decomposition methods for social and biological networks. In: Barber, S., Baxter, P.D., Mardia, K.V., Walls, R.E. (eds.) Interdisciplinary Statistics and Bioinformatics, pp. 87–90. Leeds University Press (2006)
18. Xie, N.: Social network analysis of blogs. MSc dissertation. University of Bristol (2006)
19. Zachary, W.W.: An information flow model for conflict and fission in small groups. Journal of Anthropological Research 33, 452–473 (1977)