

Finding Overlapping Communities Using Disjoint Community Detection Algorithms

S. Gregory

Abstract Many algorithms have been designed to discover community structure in networks. Most of these detect disjoint communities, while a few can find communities that overlap. We propose a new, two-phase, method of detecting overlapping communities. In the first phase, a network is transformed to a new one by splitting vertices, using the idea of *split betweenness*; in the second phase, the transformed network is processed by a disjoint community detection algorithm. This approach has the potential to convert any disjoint community detection algorithm into an overlapping community detection algorithm. Our experiments, using several “disjoint” algorithms, demonstrate that the method works, producing solutions, and execution times, that are often better than those produced by specialized “overlapping” algorithms.

1 Introduction and Motivation

Networks are a natural representation for various kinds of complex system, in society, biology, and other fields. One of the most interesting properties of many types of network is their *community structure*: the existence of groups, or *communities*, of vertices that are more densely connected to each other than to vertices in other communities. Communities often represent related groups of individuals in the real world. The automatic discovery of network communities is very useful because, for example, it can help throw light on the structure of networks which are far too large for humans to make sense of manually, even with the help of visualization techniques.

There is currently no generally accepted definition of *community*, and no standard algorithm exists for discovering communities. Numerous algorithms, using a

Steve Gregory
Department of Computer Science, University of Bristol, Bristol BS8 1UB, England
e-mail: steve@cs.bris.ac.uk

variety of methods, have been developed; these vary in their effectiveness and speed for different types of network. Many algorithms were described in the survey papers of [8, 19], but recently many new algorithms have appeared, including some very fast ones with the potential to work on very large networks. Most community detection (CD) algorithms assume that networks are unipartite and have undirected, unweighted edges; we make the same assumptions in this paper.

An important difference between algorithms is their view of the relation between the communities in a network. The vast majority of algorithms, including [6, 11, 15, 20, 25, 27, 29], assume that vertices are members of a *flat* set of *disjoint* communities. This makes sense for many networks: for example, most employees work for a single employer, most papers are published in a single conference, etc. A few algorithms, including [2, 12, 13, 24, 30], allow communities to overlap, with each individual possibly appearing in more than one community. This is more realistic in some cases: for example, many researchers belong to more than one research community. Yet other algorithms [3, 5, 17] aim to detect a *hierarchy* of communities: for example, a number of research communities each divided into several research groups.

The dichotomy between “disjoint” and “overlapping” CD algorithms is unfortunate because it limits the application of each algorithm. If a network has overlapping communities, a “disjoint” algorithm cannot find them; conversely, if communities are known to be disjoint, a “disjoint” algorithm will generally perform better than an “overlapping” algorithm. For the best results for a given network, it is important to use the right kind of algorithm. (The question of how to choose the right kind of algorithm is outside the scope of the present paper.)

In this paper we present a method to allow *any* “disjoint” CD algorithm to be used instead for finding overlapping communities. This means that a user wishing to find overlapping communities need no longer be forced to use one of the small number of “overlapping” algorithms that exist, but can also choose from the many “disjoint” algorithms. Moreover, improved “disjoint” algorithms resulting from future research can potentially also be applied to the problem of detecting overlapping communities.

Our method is implemented by transforming a network into another network that can be fed into a “disjoint” CD algorithm, and then transforming the resulting disjoint communities into (potentially overlapping) communities of the original network. The transformation is based on the *split betweenness* principle introduced in the CONGA CD algorithm [12, 13].

The next section provides a brief overview of the CONGA algorithm, which inspired this work. In Section 3 we present our transformation algorithm, named Peacock, explain its design, and compare it with CONGA. Section 4 describes the results of experiments to detect overlapping communities in both synthetic and real-world networks. The experiments use a combination of Peacock with four existing “disjoint” algorithms, as well as two existing “overlapping” algorithms. Conclusions appear in Section 5.

2 The CONGA Algorithm

CONGA (Cluster-Overlap Newman Girvan Algorithm) [12] is a CD algorithm based on Girvan and Newman's [11, 22] "GN" algorithm but extended to detect overlapping communities. CONGA adds to the GN algorithm the ability to split vertices between communities, based on the new concept of split betweenness.

CONGA comprises a sequence of steps, each of which removes an edge from the network *or* splits a vertex into two vertices:

1. Calculate edge betweenness of edges and split betweenness of vertices.
2. Remove edge with maximum edge betweenness or split vertex with maximum split betweenness, if greater.
3. Recalculate edge betweenness and split betweenness.
4. Repeat from step 2 until no edges remain.

The *edge betweenness* [10, 11] of an edge e is the number of shortest paths, between all pairs of vertices, that pass along e . The *split betweenness* [12] of a vertex v is the number of shortest paths that would pass between the two parts of v if it were split. There are many ways to split a vertex into two; the *best split* is the one that maximizes the split betweenness. Ref. [12] gives an approximate, efficient algorithm for calculating split betweenness at the same time as edge betweenness.

In CONGA, a network is initially treated as a single community, assuming it is connected. After one or more iterations, step 2 causes the network to split into two components (communities). Communities are repeatedly split into two until only singleton communities remain. By representing the binary splits as a dendrogram, the network can be partitioned into any desired number of communities.

The algorithm has a worst-case time complexity of $O(n^3)$ for a sparse network. In practice, the speed depends on the number of vertices that are split (which increases the network size) and on how easily the network breaks into separate components. This is because, in step 3, betweenness need be calculated only for the component containing the removed edge or split vertex, or for both components if step 2 caused the component to split.

Ref. [13] presents an optimized version of CONGA, named CONGO (CONGA Optimized), which employs a *local* form of betweenness. In CONGO, edge betweenness and split betweenness are calculated by counting the number of *short* paths: those that are no longer than h (a parameter). This optimization reduces the time complexity to $O(n \log n)$ for a sparse network. For simplicity, we refer to both CONGA and CONGO by the name CONGA in the remainder of this paper.

3 The Peacock Algorithm

The Peacock algorithm is used in the context shown in Fig. 1. The system comprises the following phases:

1. The network is transformed to a new, larger, network. Each step of the transformation *splits* a vertex into two vertices and one edge. Assuming the original network was connected, the transformed one will also be connected. The names of the vertices involved in each splitting step are stored for later use; for example, if vertex v splits into $\{v, v'\}$, v' is recorded as a copy of v in the vertex names file. Additionally, all vertices in the transformed file are renamed to integers, for compatibility with some CD algorithms that impose this restriction.
2. The transformed network is input to a CD algorithm, which produces a *clustering*: a set of disjoint sets of vertices.
3. The disjoint clustering is converted to a (possibly overlapping) clustering by replacing the vertex names by those used in the original network. For example, if Peacock split v into $\{v, v'\}$ and these occur in two different sets in the disjoint clustering, the final clustering includes v in both sets, which therefore overlap.

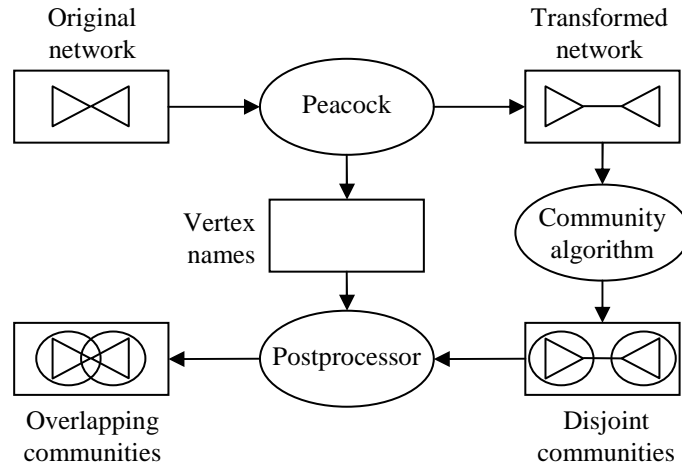


Fig. 1. Architecture of the Peacock system

The Peacock algorithm itself, which transforms the network, works as follows:

1. Calculate the split betweenness of all vertices.
2. Choose the vertex with the maximum split betweenness. Split it into two, according to its best split.
3. Recalculate the split betweenness of vertices, where this might have changed.
4. Repeat from step 2 until the maximum split betweenness is sufficiently small.
5. For each split vertex, place a new edge between the two resulting vertices.

In step 4, termination depends on the ratio between the maximum split betweenness (of all vertices) and the maximum edge betweenness (of all edges). The loop terminates when this ratio becomes less than s , a parameter of the algorithm.

In step 5, edges are placed in the order in which vertices split. For example, if v splits twice, creating v' and v'' , edges $\{v,v'\}$ and $\{v,v''\}$ are placed; if v' then splits twice, creating v''' and v'''' , $\{v',v'''\}$ and $\{v',v''''\}$ are placed, as shown in Fig. 3(a).

Fig. 2 shows an example of Peacock's transformation of a simple network. Fig. 2(a) is the original network. The maximum split betweenness is 40, for vertex a , while the maximum edge betweenness is 25, for edges $\{a,f\}$ and $\{a,g\}$. Provided the s parameter is less than 1.6 ($=40/25$), a will be split. Its best split is $(\{b,c,d\},\{f,g\})$, so the network will be transformed to that shown in Fig. 2(b). If s is small enough (less than 0.8) there are two more splitting steps that can be done, splitting h and then b , resulting finally in the network of Fig. 2(c).

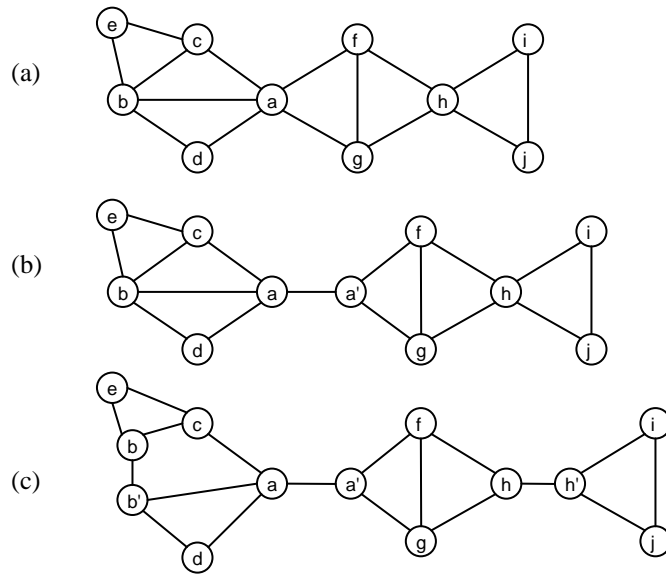


Fig. 2. Example of Peacock network transformation: (a) original network; (b) after first splitting step; (c) after all three splitting steps.

Provided the s parameter is small enough, its exact value is not critical to the result of the community detection. For example, if we ask a CD algorithm to divide the network of Fig. 2(b) into two disjoint communities, the result will usually be $\{\{a,b,c,d,e\}, \{a',f,g,h,i,j\}\}$. If we feed the network of Fig. 2(c) into the same algorithm, it will find larger communities, $\{\{a,b,b',c,d,e\}, \{a',f,g,h,h',i,j\}\}$, but both of these solutions are postprocessed to the same pair of overlapping communities: $\{\{a,b,c,d,e\}, \{a,f,g,h,i,j\}\}$. We return to the choice of s below in this section.

The Peacock algorithm is quite similar to CONGA. One difference is that CONGA does not bridge the gaps formed when a vertex splits. Another difference is that CONGA interleaves the vertex splitting steps (as described above) with edge removal steps (as in the GN algorithm [11]). In CONGA, both vertex split-

ting and edge removal steps act to break down a network into separate components which represent communities. Peacock is not intended to detect communities, and so it keeps the network connected.

Design Alternatives

We choose to recalculate betweenness in each iteration, instead of simply splitting the vertices that initially have a split betweenness greater than a certain value. This is for the same reason as the GN and CONGA algorithms recalculate betweenness in each iteration: the network structure changes, breaking into separate components, and the values of betweenness rapidly become out of date. Besides, sometimes a vertex needs to be split more than once; its betweenness after its first split cannot be calculated at the beginning.

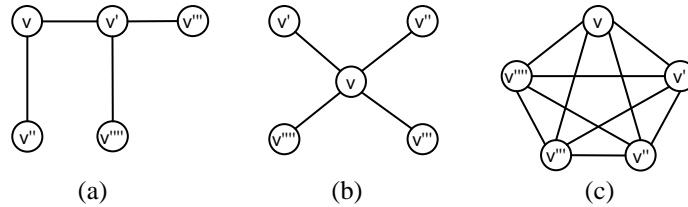


Fig. 3. Alternative ways to connect split vertices: (a) method used in Peacock; (b) connecting vertex to each of its copies; (c) connecting vertices in a clique.

Another key design decision is whether, and how, to bridge the gaps formed when vertices split. The method used is to place an edge across each gap as it is created. For example, if v splits twice, creating v' and v'' , and v' splits twice, creating v''' and v'''' , the edges placed are shown in Fig. 3(a). This method was chosen because the edge betweenness of the new edge approximately equals the split betweenness of the split vertex. (It is not identical because the network contains extra vertices and longer paths following the split.) This helps make the community structure apparent in the transformed network fed to the CD algorithm. The following alternatives were also considered:

1. Do not add any edges. The problem with this method is that the network is likely to break into disconnected components during the transformation process, which affects the communities that can be found by the CD algorithm.
2. Place an edge only when necessary to prevent the network splitting into two components. This avoids the above problem, but the network still breaks into almost-separate components connected by few edges, so the results are poor.
3. Place an edge between the original vertex (v) and each of the copies of it (v' , v'' , v''' , v''''), as in Fig. 3(b). This method gives worse results than the chosen one.

4. Join the original vertex and its copies in a clique, as in Fig. 3(c). This method sometimes works well but is very sensitive to the value of parameter s . This is because a clique, especially a large one, is treated as one community by most CD algorithms, so there is no advantage in transforming a vertex to a clique.

Another issue is the value of parameter s . In most networks, the maximum split betweenness is slightly greater than the maximum edge betweenness, so setting s to a value greater than about 1 or 2 leaves the network unchanged. A smaller value of s causes more vertices to be split and the network to increase in size. We have experimented with s ranging from 0.005 to 0.5, and found remarkably little difference in the solution quality. Some CD algorithms favour a larger value while some prefer a smaller value, but the difference is small. As regards execution time, a large value of s is preferable, so that the CD algorithm will have a smaller network to process. We settled on a value of $s=0.1$ for all experiments in the next section.

The final design decision is the value of the parameter h . For CONGA, reducing h usually reduces both solution quality and execution time. Using Peacock, combined with the GN algorithm, we varied h for each phase: reducing h for Peacock had a much smaller effect on solution quality than reducing h for the GN algorithm. The same is true of the other CD algorithms. This suggests that local betweenness is a more acceptable optimization for “splitting” than for community detection. We therefore used $h=2$ for all experiments in the next section.

4 Experiments

To evaluate Peacock, we combined it with several disjoint CD algorithms. These were chosen because they are modern algorithms with the potential to handle large networks, *and* implementations of them, by their authors, were readily available:

1. CNM. Clauset, Newman, and Moore’s “fast modularity” algorithm of [6, 31].
2. WT. The algorithm of Wakita and Tsurumi [27, 32] (rev. 159): an optimization of the CNM algorithm.
3. BGLL. The “fast unfolding” algorithm of Blondel, Guillaume, Lambiotte, and Lefebvre [3, 33] (February 2008 version): another modularity-maximizing algorithm, claimed to be faster than CNM or WT.
4. PL. The “Walktrap” algorithm of Pons and Latapy [25, 34] (v0.2), which works by generating random walks which tend to get trapped in communities.

We compare the results with results from two existing overlapping CD algorithms, whose code is also available:

1. CFinder. The “clique percolation” algorithm of Palla, Derényi, Farkas, and Vicsek [1, 24, 35] (v1.21).
2. CONGA. Gregory’s CONGA algorithm [12, 13, 36] (v1.59) with $h=2$.

For CNM, WT, PL, and CONGA, the user can choose the desired number of communities, although for WT there is a minimum number of communities that can be found. In contrast, BGLL and CFinder find a small number of solutions, each with a fixed number of communities.

Experiments with Synthetic Networks

A good way to evaluate a CD algorithm is by generating artificial networks based on a known community structure and comparing the known communities with those found by the algorithm. The comparison can be done in various ways, including the F-measure and Mutual Information measure [9]. The Adjusted Rand index [16], a variant of the Rand index [26] that excludes the effects of chance, is often considered the most accurate. However, it is not ideal for solutions containing overlapping clusters because it does not consider the number of clusters containing each pair of vertices. We therefore use the Omega index [7]: an extension of the Adjusted Rand index for solutions with overlapping clusters.

We randomly generated a set of networks containing n vertices divided into c equally-sized communities, each containing nr/c vertices. Vertices are randomly and evenly distributed between communities so that each vertex is a member of r communities on average. r is a measure of overlap: $r=1$ means that communities are disjoint and $r=c$ means that each community contains all vertices. The network is constructed by placing edges between pairs of vertices randomly, with probability ip_{in} if there are i (≥ 1) communities to which both vertices belong, and p_{out} otherwise. All networks used in the experiments are connected. Results shown are the average of 100 runs.

In these experiments we evaluate Peacock ($h=2$) combined with CNM, WT, and PL, and compared these with CONGA ($h=2$) and CFinder. For most of the algorithms we ask for c communities, where c is the known number of communities in the network. This is impossible with CFinder, whose only parameter is k (cluster density), so we show the results from CFinder for *all* values of k .

Fig. 4 shows results for 256 vertices in 32 communities. The overlap is 2, so each community contains 16 vertices. As p_{out} increases, the community structure becomes less evident and the solution quality decreases, more sharply for CONGA than for CFinder. Peacock+PL behaves similarly to CONGA, but Peacock+CNM is much better – comparable with CFinder – while Peacock+WT is slightly worse.

Fig. 5 shows the effect of increasing the density of intracommunity edges, which should increase the solution quality. All combined algorithms perform better than CONGA for low p_{in} , with Peacock+WT slightly worse than the others.

In Fig. 6 we fix p_{in} and p_{out} and vary the overlap, r . CONGA's performance declines as r increases above 2. Peacock+PL behaves slightly better while Peacock+CNM is better than CONGA or CFinder. Again, Peacock+WT performs slightly less well than Peacock+CNM.

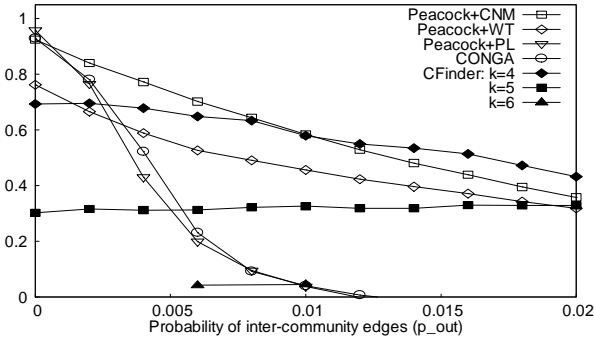


Fig. 4. Omega index for random networks with $n=256$, $c=32$, $r=2$, $p_{in}=0.5$, various p_{out} .

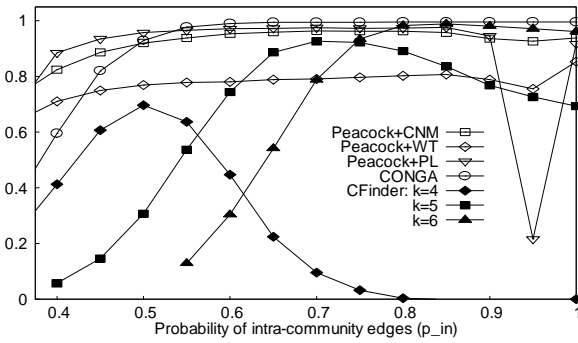


Fig. 5. Omega index for random networks with $n=256$, $c=32$, $r=2$, $p_{out}=0$, various p_{in} .

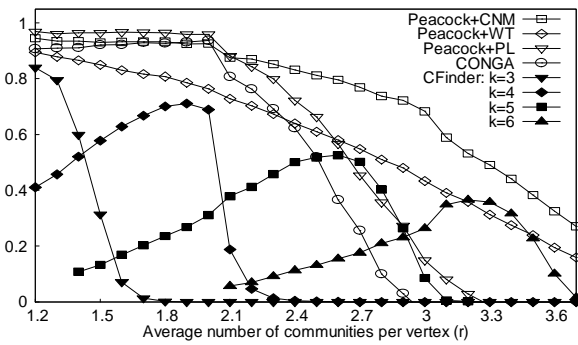


Fig. 6. Omega index for random networks with $n=256$, $c=32$, $p_{in}=0.5$, $p_{out}=0$, various r .

Fig. 7 shows the effect of varying the network size while keeping the community size constant. Peacock+CNM and Peacock+PL both perform better than CONGA, with Peacock+WT slightly worse than the others.

In Fig. 8 the network size is fixed but the number (and therefore size) of the communities varies. Peacock+CNM performs better than CONGA; the other combined algorithms perform slightly worse, but still better than CFinder.

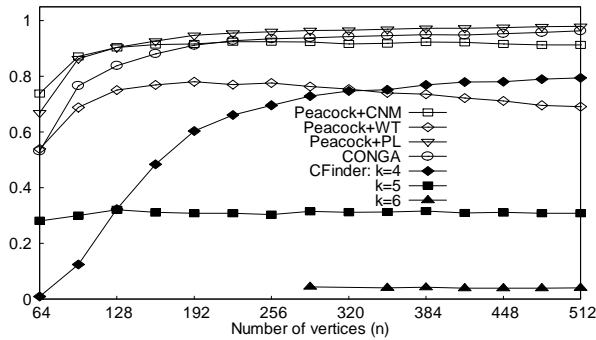


Fig. 7. Omega index for random networks with $c=n/8$, $r=2$, $p_{in}=0.5$, $p_{out}=0$, various n .

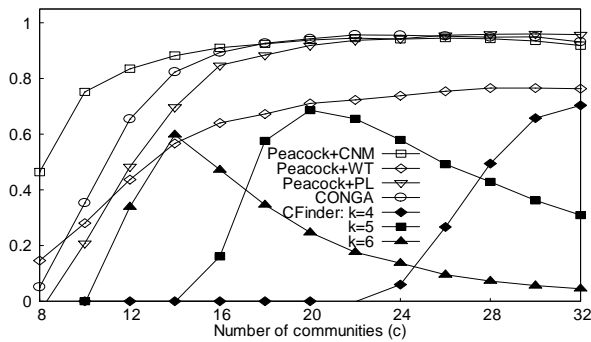


Fig. 8. Omega index for random networks with $n=256$, $r=2$, $p_{in}=0.5$, $p_{out}=0$, various c .

Fig. 9 shows how the total execution time, to detect the specified number of communities, varies with network size. All these networks contain overlapping communities of a small fixed size with overlap 1.2. All programs were run under Linux on an AMD Opteron 250 CPU at 2.4GHz. For each of the combined “Peacock+X” algorithms, the execution time plotted comprises the time for the network-transformation phase (using Peacock) *plus* the time for the CD phase (using algorithm X). The time for the Peacock phase is similar to CONGA’s execution time: it increases almost linearly with size, at least for the experiment shown here.

The time for the CD phase depends on which algorithm is used *and* how sensitive that algorithm is to network size, since it has to process the transformed network, which is larger than the original one whose size is shown on the horizontal axis.

This shows that, for large networks, execution time is greatest for CFinder and least for CONGA, while Peacock+WT is the fastest of the combined algorithms.

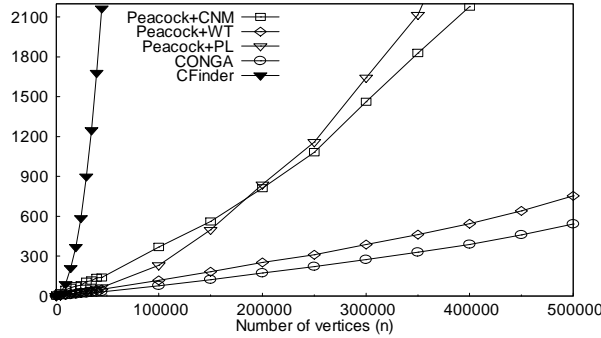


Fig. 9. Execution time (seconds) for random networks $c=n/8$, $r=1.2$, $p_{in}=0.5$, $p_{out}=0$, various n .

Experiments with Real-World Networks

We have run the CD algorithms on several real-world networks, listed in Table 1. The table shows the source of each network, its size, and the times for the various algorithms to generate solutions, on an AMD Opteron 250 at 2.4GHz. The total execution time is the sum of the execution time of the Peacock network-transformation phase (column 7) and the time of the CD phase (last four columns). Again, the time for the Peacock phase is similar to CONGA's execution time.

When evaluating a CD algorithm on real-world networks, there is usually no known "correct" solution. Solution quality must be assessed in a different way: for example, by modularity [21, 22], which measures the relative number of intra-community and intercommunity edges. A high modularity indicates that there are more intracommunity edges than would be expected by chance.

The original modularity measure, Q , is defined only for disjoint communities, but Nicosia *et al.* [23] proposed a new modularity measure, Q_{ov} , which is defined also for overlapping communities. Q_{ov} is defined so that $Q_{ov}=0$ when all vertices belong to one community or all belong to singleton communities, while higher values of Q_{ov} indicate stronger community structure. Each vertex may belong to each community with any *belonging coefficient*. For each vertex, the belonging coefficients for all communities sum to 1.

Table 1. Results on real-world networks

Name	Ref	Vertices	Edges	Execution time (s)						
				CONGA	CFinder	Peacock: combined algorithm				
						Peacock	Phase 2			
							CNM	WT	PL	BGLL
netscience	20	379	914	1.3	0.25	0.35	0.6	0.9	0.03	0.06
cond-mat-2003	18	27519	116181	1127	1134	1088	82.8	10.8	24.1	3.71
blogs	28	3982	6803	6.5	3.05	5.23	6.74	2.15	0.43	0.32
blogs2	28	30557	82301	294	415	289	86.3	9.8	33.9	∞
PGP	4	10680	24316	83	34745	89.8	18.6	4.62	2.66	0.90
email	14	1133	5451	30.2	4.00	32.6	4.00	2.61	0.45	0.24
word_association	24	7205	31784	175	96.5	176	33.2	4.47	6.49	1.78
protein-protein	24	2445	6265	8.6	2.75	7.56	4.84	2.00	0.41	∞

We use modularity (Q_{ov}) here to evaluate solutions on real-world networks. The belonging coefficient of each vertex is set to $1/c$, where c is the number of communities it belongs to; i.e., vertices belong equally to all communities they are in.

Fig. 10 shows the modularity of the eight networks listed in Table 1. “netscience” and “cond-mat-2003” are collaboration networks of coauthorships, of different sizes. Peacock+PL finds the solutions with the highest modularity, over a certain range; otherwise, the best results are obtained by Peacock+CNM. Both give a higher modularity than CONGA. CFinder finds several solutions, one of which has a slightly higher modularity than the other algorithms.

“blogs” and “blogs2” are networks of communication relationships between owners of blogs on the MSN (Windows Live™) Spaces website. “blogs2” is much larger than “blogs” and has a higher average degree. “PGP” and “email” are other social networks representing PGP key signing and email, respectively. For all four networks, the story is the same as for “netscience” and “cond-mat-2003”: Peacock+PL gives the best results over a certain range, Peacock+CNM gives consistently good results, and both perform better than CONGA. For the “blogs” network, Peacock+WT also does well.

The last two, “word_association” and “protein_protein”, are non-social networks, from psychology and biology, respectively, both from [24]. For the first of these, Peacock+PL finds a higher-modularity solution than CONGA, but not quite as good as CFinder’s best solution. For the second, Peacock+PL finds even better solutions than CFinder’s best, while Peacock+CNM also performs well.

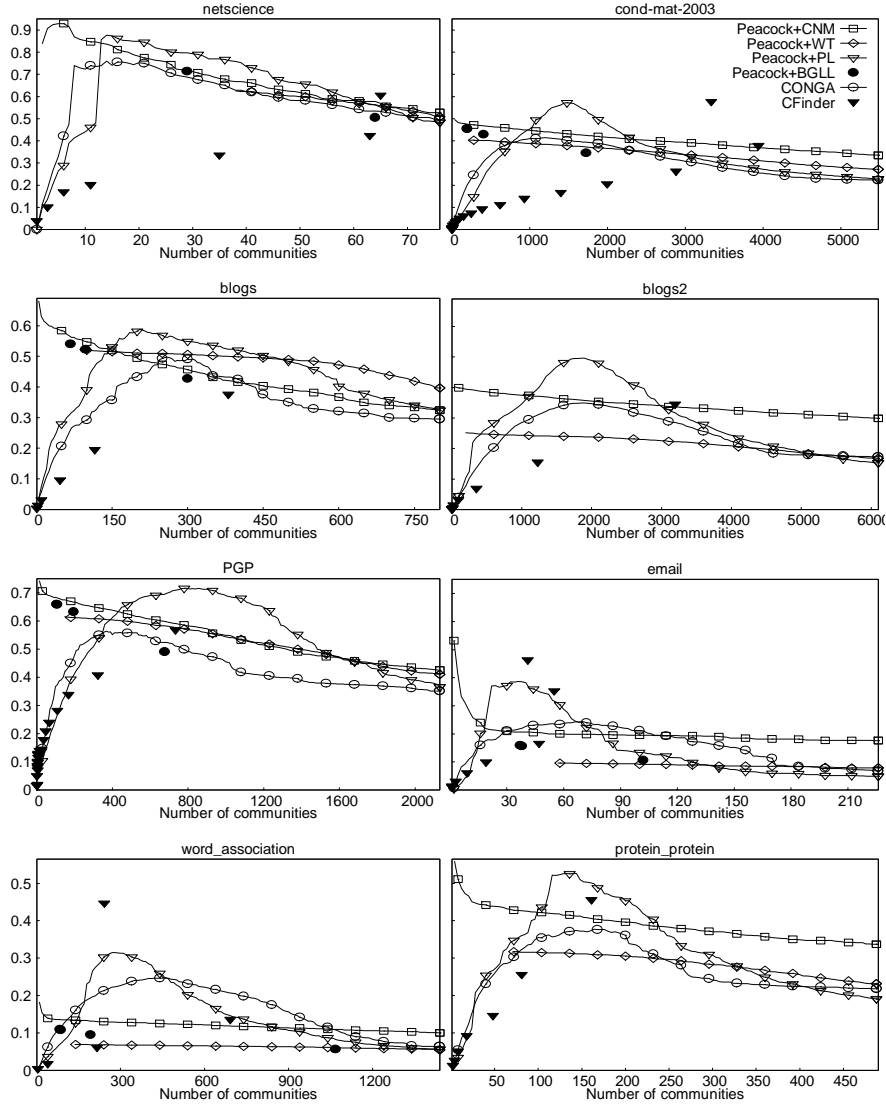


Fig. 10. Modularity of real-world networks. The y-axis shows the Q_{ov} modularity.

5 Conclusions

We have proposed a novel, two-phase, approach to detecting overlapping communities in networks. In principle, this is attractive because it separates the “overlap-

ping” and “community detection” issues, allowing the best algorithm to be selected for each phase. For the first phase, we have presented the Peacock algorithm, based on the “split betweenness” principle. While the CONGA algorithm uses the betweenness principle for both overlapping and community detection, Peacock uses it only for the former. Interestingly, the local form of betweenness [13] works better in Peacock than it does in CONGA, suggesting that our approach should yield good execution speed.

The results reported in Section 4 seem to confirm that our approach is viable. In terms of solution quality, the two-phase algorithm works well, especially with the CNM or PL algorithm as the second phase. In most cases, these combined algorithms outperform the two specialized “overlapping” CD algorithms.

Concerning execution time, for the small networks shown in Table 1, the time for the combined algorithms is dominated by the Peacock phase, whose execution time is similar to CONGA’s, while CFinder’s execution time is usually better but sometimes worse. For larger networks, as Fig. 7 shows, CFinder’s execution time increases rapidly with network size, while the time for the 2-phase algorithm becomes dominated by the second (community detection) phase. The time for the Peacock phase, as for CONGA, increases almost linearly with size. For large networks, most of the total execution time is occupied by the CD phase, unless this is done by the (fast) WT algorithm. The time for the CD phase varies according to the algorithm used, but will always be longer than for detecting disjoint communities, because the algorithm needs to process the larger, transformed, network.

Future work includes evaluating the Peacock algorithm in conjunction with even more disjoint CD algorithms, including those that have yet to be designed. The implementation of the Peacock algorithm, including its postprocessor, is available at <http://www.cs.bris.ac.uk/~steve/networks/>.

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., Magdon-Ismael, M.: Efficient Identification of Overlapping Communities. In: *ISI 2005. LNCS*, vol. 3495, pp. 27--36. Springer, Heidelberg (2005)
3. Blondel, V.D., Guillaume, J-L., Lambiotte, R., Lefebvre, E.: Fast Unfolding of Communities in Large Networks. *J. Stat. Mech.*, P10008 (2008)
4. Boguñá, M., Pastor-Satorras, R., Diaz-Guilera, A., Arenas, A.: Models of Social Networks Based on Social Distance Attachment. *Phys. Rev. E* 70, 056122 (2004)
5. Clauset, A., Moore, C., Newman, M.E.J.: Hierarchical Structure and the Prediction of Missing Links in Networks. *Nature* 453, 98--101 (2008)
6. Clauset, A., Newman, M.E.J., Moore, C.: Finding Community Structure in Very Large Networks. *Phys. Rev. E* 70, 066111 (2004)
7. Collins, L.M., Dent, C.W.: Omega: A General Formulation of the Rand Index of Cluster Recovery Suitable for Non-disjoint Solutions. *Multivar. Behav. Res.* 23, 231--242 (1988)
8. Danon, L., Diaz-Guilera, A., Duch, J., Arenas, A.: Comparing Community Structure Identification. *J. Stat. Mech.* P09008 (2005)

9. Fred, A.L.N., Jain, A.K.: Robust Data Clustering. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 128--133. IEEE Press, New York (2003)
10. Freeman, L.C.: A Set of Measures of Centrality Based on Betweenness. *Sociometry* 40, 35--41 (1977)
11. Girvan, M., Newman, M.E.J.: Community Structure in Social and Biological Networks. *P. Natl. Acad. Sci. USA* 99, 7821--7826 (2002)
12. Gregory, S.: An Algorithm to Find Overlapping Community Structure in Networks. In: PKDD 2007. LNAI, vol. 4702, pp. 91--102. Springer, Heidelberg (2007)
13. Gregory, S.: A Fast Algorithm to Find Overlapping Communities in Networks. In: PKDD 2008. LNAI, vol. 5211, pp. 408--423. Springer, Heidelberg (2008)
14. Guimera, R., Danon, L., Diaz-Guilera, A., Giral, F., Arenas, A.: Self-similar Community Structure in a Network of Human Interactions. *Phys. Rev. E* 68, 065103(R) (2003)
15. Hofman, J.M., Wiggins, C.H.: Bayesian Approach to Network Modularity. *Phys. Rev. Lett.* 100, 258701 (2008)
16. Hubert, L., Arabie, P.: Comparing partitions. *J. Classif.* 2, 193--218 (1985)
17. Lancichinetti, A., Fortunato, S., Kertesz, J.: Detecting the Overlapping and Hierarchical Community Structure of Complex Networks. Eprint arXiv:0802.1218v1 at arxiv.org (2008)
18. Newman, M.E.J.: The Structure of Scientific Collaboration Networks. *P. Natl. Acad. Sci. USA* 98, 404--409 (2001)
19. Newman, M.E.J.: Detecting Community Structure in Networks. *Eur. Phys. J. B* 38, 321--330 (2004)
20. Newman, M.E.J.: Finding Community Structure in Networks Using the Eigenvectors of Matrices. *Phys. Rev. E* 74, 036104 (2006)
21. Newman, M.E.J.: Modularity and Community Structure in Networks. *P. Natl. Acad. Sci. USA* 103, 8577--8582 (2006)
22. Newman, M.E.J., Girvan, M.: Finding and Evaluating Community Structure in Networks. *Phys. Rev. E* 69, 026113 (2004)
23. Nicosia, V., Mangioni, G., Carchiolo, V., Malgeri, M.: Extending Modularity Definition for Directed Graphs with Overlapping Communities. Eprint arXiv:0801.1647v3 at arxiv.org (2008)
24. 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)
25. Pons, P., Latapy, M.: Computing Communities in Large Networks Using Random Walks. *J. Graph Algorithms and Applications* 10, 2, 191--218 (2006)
26. Rand, W.M.: Objective Criteria for the Evaluation of Clustering Methods. *J. Am. Stat. Assoc.* 66, 846--850 (1971)
27. Wakita, K., Tsurumi, T.: Finding Community Structure in a Mega-scale Social Networking Service. In: IADIS International Conference on WWW/Internet 2007, pp. 153--162 (2007)
28. Xie, N.: Social Network Analysis of Blogs. MSc Dissertation. University of Bristol (2006)
29. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.: SCAN: a Structural Clustering Algorithm for Networks. In: 13th International Conference on Knowledge Discovery and Data Mining, KDD 2007, pp. 824--833. ACM, New York (2007)
30. Zhang, S., Wang, R., Zhang, X.: Identification of Overlapping Community Structure in Complex Networks Using Fuzzy C-means Clustering. *Physica A* 374, 1, 483--490 (2007)
31. Clauset, A.: <http://cs.unm.edu/~aaron/research/fastmodularity.htm>
32. Wakita, K.: <http://www.is.titech.ac.jp/~wakita/en/software/community-analysis-software>
33. Guillaume, J-L.: <http://findcommunities.googlepages.com>
34. Pons, P.: <http://psl.pons.free.fr/index.php?item=prog&lang=en>
35. CFinder: <http://www.cfinder.org>
36. Gregory, S.: <http://www.cs.bris.ac.uk/~steve/networks>