Barycentric Graph Clustering

Jonathan Cohen
National Security Agency
Suite 6514
9800 Savage Road
Fort Meade, MD 20755-6513

Abstract

Clustering is an essential aid to human understanding. In its application to graphs, clustering allows a large graph to be partitioned into pieces that can be studied independently or collapsed to give a coarse picture. More importantly, it can be used to identify coherent subgraphs that bear particular attention.

As the size of graphs under examination has continued to increase, it has become increasingly important to devise graph algorithms that scale well. This need is felt particularly for graph clustering and for extracting subgraphs of interest, precisely because human comprehension of large graphs is impractical.

This paper presents a family of stochastic methods for clustering graphs into coherent subgraphs. These methods scale linearly with the graph size and do not require *a priori* specification of the number or size of clusters.

After a physical motivation, an exposition of the methods is presented, followed by experimental results on some sample graphs.

Table of Contents

1 Introduction *3*
- 1.1 Why we're here 3
- 1.2 A bit of prior art 4
- 1.3 A motivating observation 8

2 The Method *9*
- 2.1 The basic idea 9
- 2.2 Multiple models and notation 10
- 2.3 Model 1: a spring model 11
- 2.4 Model 2: barycentric model 12
- 2.5 Other models 12
- 2.6 The general model 13
- 2.7 Normalizing weights 14
- 2.8 Random starts 14
- 2.9 Cutting criterion 14
- 2.10 Slackening midway 15
- 2.11 Cleanup phase 16
- 2.12 Co-pendant no more? 16

3 Some experiments *17*
- 3.1 Examples and comparison with Newman clustering 17
- 3.2 Varying the number of iterations 23
- 3.3 Varying the number of random starts 28
- 3.4 Processing speed 29

4 Identifying tight clusters *30*
5 Things to do *33*
6 Concluding remarks *34*
Acknowledgements *34*
Appendix A: Picking the gain for the springs model *34*
Appendix B: Distribution of edge lengths *35*
Appendix C: Markov interpretation of edge length *37*
Appendix D: Relation to recursive spectral bisection. *37*
Appendix E: A deterministic algorithm? No. *38*
References *40*

1 Introduction

1.1 Why we're here

For decades, researchers and practitioners in social network analysis have sought methods of recognizing tight groups of actors, as revealed by their ties in social network graphs. More generally, those engaged in a variety of fields exploiting graphs have proposed many methods to locate sets of vertices that cohere. As the size of graphs under study has increased, workers have sought to develop automated ways of recognizing such groups, with attention turning from the initial question of what makes the most sensible definition of cohesive groups to the practical question of what are the fastest methods that still make sense.

In addition, considerable work has gone into exhaustive partitioning of graphs into subgraphs so that the constituents may be dealt with in turn, or so that a physical problem represented by the graph may be similarly partitioned.[1] Many of the approaches to graph partitioning either assume that the number of clusters is known in advance or that it is desirable to find clusters that are nearly equal in size, neither of which is generally appropriate.[2]

The author is interested in achieving the twin goals of discovering subgraphs that are tight, relative to the surrounding graph, and partitioning the entire graph sensibly. These are not incompatible, but seem not to have been considered jointly by most workers in the field. Methods aimed at finding tight subgraphs have usually not considered the bulk of the graph, and methods aimed at complete partitioning assign the connective detritus to adjacent clusters randomly.

Equally important, the author seeks an approach that scales well, and is free of assumptions about the size or number of clusters.

The paper presents a clustering method that achieves these goals — a family of them, in fact. In particular, the method's work scales in proportion to the size of the graph. The price for this remarkable scaling is that the approach is stochastic, meaning that it employs samples from a random number generator. As a consequence, the results may differ from run to run.

This communication begins with a quick review of other approaches, then suggests a physical motivation for the proposed approach. The new method is then outlined, with variations, and then described in detail. Finally, some results are offered, showing sensible clustering.

[1] The problem of dividing an electrical circuit across several circuit boards is a good example.

[2] See Danon, *et al.* 2006 for a discussion of cluster size inhomogeneity.

1.2 **A bit of prior art**

Clustering is a nearly inexhaustible subject, since it is found in every field, and since the results are judged by methods that are largely subjective. It would appear that there are at least as many clustering techniques as there are people who would cluster.

One can consider two broad categories of clustering for graphs: those exhaustive methods that seek to partition the entire graph, assigning each vertex to a cluster, and those methods that find only the cohesive groups, leaving the remaining vertices to wonder what all the fuss was about.

This paper presents a method whose aim falls somewhere between these two goals, so it is instructive to consider both classes of algorithms. Here is a quick review of some exhaustive methods:

Conventional graph agglomerative methods mirror the agglomerative methods for arbitrary objects: distances between objects are specified, as are distances between sets of objects. In the case of clustering graph vertices, the distances are stated in terms of graph distances. To begin, each object is placed into its own cluster; thereafter, the two closest clusters are combined, with this step repeated until a stopping point is reached. This approach is popular and well understood, but the work scales (at best) as $|V|^2 \log|V|$, where $|V|$ is the number of vertices. The recent proposal of Zhang *et al.* [2007b] is an example of a conventional agglomerative method.

Fast agglomerative methods are modifications of conventional agglomerative approaches. They recognize that the clusters are subgraphs, and that one may want to restrict putative cluster joins to those subgraphs that are adjacent. The most popular such method is due to Newman *et al.* [Newman 2003 and Clauset, Newman, and Moore 2004], which seeks to maximize "modularity" by a greedy approach. With careful data management, one can cluster to a depth[3] of δ with $|E|\delta\log|V|$ work, where $|E|$ is the number of edges.

One of the clustering methods peculiar to graphs is the Girvan-Newman algorithm [Girvan and Newman 2001] based on betweenness — a measure of link importance that counts the number of shortest paths (between every pair of vertices) that pass through the link. The method simply calculates the betweenness of each link, removes the link of highest betweenness, recalculates, and continues that process until a stopping point is reached. After stopping, each remaining component is a cluster. Girvan-Newman clustering is one of the most expensive algorithms, owing to the need to recalculate the betweenness after each edge deletion. The work scales as $|E|^2|V|$. Radicchi *et al.* [2004] offer an improvement using a local definition of betweenness, bringing the work down to $|V|^2$. Wilkinson and Huberman [2004] use repeated approximate measurements of betweenness that permit fuzzy clustering.

[3] "Depth" here is the depth of the dendrogram representing the clustering.

Much of the graph clustering literature deals with spectral clustering, which has its origins in the work of Miroslav Fiedler, though he was not pursuing clustering, *per se*. He observed that the smallest nonzero eigenvector of the graph's Laplacian [Fiedler 1973] said something about the graph connectivity, and the corresponding eigenvector, which he called the "characteristic valuation," [Fiedler 1975] had the surprising property that one could partition the vertices by whether or not their values in this eigenvector were above or below a cutoff point of ones choosing, with the result the two induced subgraphs were always each a single component. This eigenvector is popularly called the Fiedler vector.[4]

In general, spectral clustering [for example, Pothen *et al.* and 1990, De Wit 1991] looks at the eigenspace of matrices associated with the graph. Most popularly, values of the Fiedler vector are used to divide the vertex set, though other matrices and combinations of eigenvectors are used. Spectral clustering usually takes the form of recursive spectral bisection, in which some eigenvector is calculated and used to divide the graph into two halves, according to whether their corresponding entries in the eigenvector are larger or smaller than some split point. The process is repeated on each of the two halves, and so on. At best, the resulting work scales as $|E|\log|V|$. Recent work by Newman [2006] seeks to maximize "modularity" through spectral methods, but at considerably more expense than $|E|\log|V|$. Capocci *et al.* [2004] obtains division by clustering components in eigenvectors of the square of the adjacency matrix; the work scales at least as $|V|^2$.

Rosvall and Bergstrom [2007] present an information-theoretic approach, in which the graph is described in compressed form by cluster membership and intercluster edges. Ideal clusters maximize the mutual information between the actual graph and its compressed form. Maximization is conducted by simulated annealing, and is, therefore, not cheap.

Along the lines of matrices, Zhang *et al.* [2007a] use non-negative matrix factorization of a distance matrix to find fuzzy cluster membership. Specifying a number of clusters *k*, the algorithm takes $k|V|^2$ work. The result permits each vertex to belong to multiple clusters, and provides a degree of membership for each cluster.

Wu and Huberman [2004] consider the graph a resistive network. They pick two non-adjacent vertices as seeds. Establishing a voltage between these seeds, they then solve for voltages at all other nodes in an iterative, linear fashion. Dividing the voltages into those above and below a gap produces a putative bisection. Repeated trials with different seeds produce votes that are combined to produce a final bisection. The work scales as $|C||E|$, where $|C|$ is the user-specified number of clusters. Alves [2007] also

[4] More correctly, Fiedler called any scaling of the eigenvector "*a* characteristic valuation." Popular terminology uses "*the* Fiedler vector" to refer to any scaling.

treats the graph as a resistive network and partitions the graph on the similarity of resistance spectra of vertices, at the expense of $|V|^3$ work.

Flake *et al.* [2000] use a minimum cut to find the division between seeds in a group of interest and rest of the graph; partitioning the whole graph with such a scheme would be expensive. Donetti and Munoz [2004] combine spectral, agglomerative, and modularity clustering in an approach whose speed is limited by the need to obtain many eigenvectors of the Laplacian. One of the first attempts to cluster graphs was Kernighan and Lin [1970], who offered a $|V|^2 \log|V|$ algorithm that hill-climbed on an initially-random bisection by swapping pairs of vertices to minimize inter-cluster weights.

Multilevel clustering, in which the graph is interatively coarsened, then clustered, then iteratively refined, was proposed by Karypis and Kumar [1996], but for a fixed number of clusters. Their approach, which would require some $|E|\log|V|$ work on a serial machine, is actually aimed at parallel machines, with number of clusters commensurate with the number of processors. Wu *et al.* [2004] implement a multilevel version of k-medoids.

Another clustering method borrowed from outside of graphs is k-medoids. To begin, one picks an integer *k*, the number of clusters desired. Then *k* objects are picked at random to be cluster seeds, one per cluster. Each of the other objects is placed in the cluster whose seed is closest the object being examined. Then, for each cluster, a new cluster seed is picked from that cluster, better representing the cluster by being closest to center. The clusters are emptied except for these new seeds. Again, each remaining vertex is assigned to the cluster whose seed is the closest. The processes of seeding and assigning are repeated until a stopping point is reached. At the outset, $|V|^2 \log|V|$work is required to determine all of the inter-vertex distances. This is already too much work, and the method is largely ineffective. Better results are claimed by those doing a generalization called kernel k-means clustering, and faster processing may be obtained by being selective in computing inter-vertex distances. Moreover, Dhillon *et al.* [2005] show that a variety of clustering techniques, including some spectral methods, may be cast as kernel-based k-means approaches, and go on to offer an implementation via multilevel clustering with $|E|\log|V|$ work. Angelini *et al.* [2007] optimize a "ratio association" by deterministic annealing. They show that their ratio association, a normalized sum of intracluster links, is equivalent to a kernel k-means clustering, with the kernel derived from the adjacency graph. Finding an optimal clustering requires $|V|^3$ work.

A compendium and comparison of the state-of-the-art in 2005 is offered by Danon *et al.* [2005].

Most interestingly, several recent partitioning approaches claim linear processing time, though the author has yet to evaluate them. The label-passing algorithm of Raghavan *et al.* [2007] is a brilliantly simple method, in which vertices begin with unique labels, then, iteratively, each takes on the label held by a majority of its neighbors, with ties handled randomly. Rattigan, Maier, and Jensen [2007] propose dividing the graph into zones, recording the distances between vertices in terms of these zones. Such a division of coarse and fine distance offers the prospect of substantial speed increases to several standard methods. Finally, Auber *et al.* [2004] suggest clustering a graph by clustering scalar values assigned to each node called Strahler numbers.

Much of prior work, particularly in the realm of social network analysis, has been concerned with approaches that look only for cohesive subgraphs, rather than constructing exhaustive partitions. Initial work sought cliques, which are unnecessarily strict, are terrifically expensive to find, and are difficult to use, since they can overlap in a complicated manner. Later work looked at various relaxations of cliques (clans, clubs, k-cliques, k-plexes), which suffer the same drawbacks. A useful and computationally cheap group of mild cohesion is the k-core, a subgraph in which every vertex is connected with at least k other vertices. While it likely that a cohesive group of interest would be *contained* in a k-core, the k-core is far too promiscuous to be used to isolate cohesive groups by itself. See Wasserman and Faust (1994) for a review of cohesive subgraphs used in social network analysis.

The *k*-truss [Cohen 2005] identifies a subgraph in which each edge is supported by being in a minimum number, *k*–2, of triangles. Equivalently, each edge joins vertices that share at least *k*–2 common neighbors. The *k*-truss is easy to interpret, and is likely to be significant for *k* larger than, say, 4. One can locate all *k*-trusses (of specified *k*) with work of $\sum d^2(v)$, where $d(v)$ is the degree (valence) of vertex *v*. This work is usually larger than $\sum d(v) = |E|$.

1.3 A motivating observation

Consider a physical analog of a graph constructed as follows: let each graph vertex be represented by a ball, and let each graph edge by represented by a spring connecting the appropriate pair of balls. These springs are special, in that they have a relaxed length of zero (making them very hard to find if dropped). Given such an analog of the graph, we can grab the graph at several points and pull those points away from each other. As a result, springs will stretch in a manner that may be revealing about the structure of the graph, suggesting clusters of possible interest.

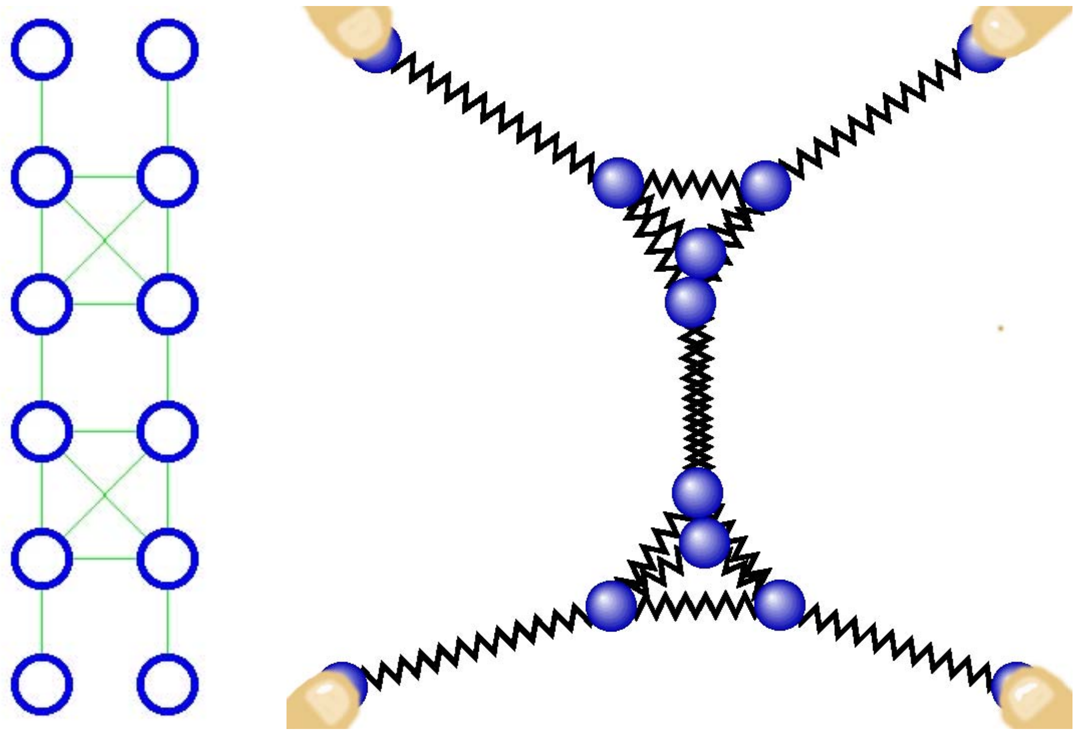


Figure 1. A Graph and Its Analog. The physical graph is being stretched by pulling on the four vertices of valence 1. Note that the two clusters (cliques of size 4) have internal links that are less stretched than the edges external to the clusters.

An example is pictured in Figure 1, which shows a graph on the left, and a physical version of the graph on the right. The physical graph is being stretched by pulling on the four obvious places to pull, namely on the vertices that have only one neighbor. The result clearly suggests two clusters of four vertices each, in that the springs within the clusters are not stretched as much as the springs outside of the clusters. In fact, these two putative clusters are indeed cliques of size 4, which constitute obvious cohesive subgraphs worthy of attention.

This observation suggests a general approach to finding cohesive sections of the graph: Construct a spring analog of the graph to study, pull on it, cut the edges (springs) that are relatively long, and treat the resulting components as clusters.

2 The Method

2.1 The basic idea

In the exposition that follows, it is assumed that the graph consists of a single component. In practice, each component of the graph is dealt with separately.

As the introduction suggested, we can make a physical model of the graph by representing each vertex by an infinitesimal ball, and representing each edge by joining a pair of balls by a spring of zero rest length. This model can then be represented in a computer, where it can be manipulated and observed.

The idea of pulling on the graph, while motivational, is probably not the way to proceed: one would inadvertently choose to pull apart vertices that should be in the same cluster. Instead, one can force the graph vertices to random locations, then release them and watch how the vertices move. Those vertices that are members of the same tight group will be brought together quickly, while those that are joined tenuously will be brought together slowly. So although the entire graph will shrink to a single point, the rate of shrinking will differentiate intracluster links from intercluster links.

The basic algorithm is as outlined in Figure 2. Repeated trials are conducted, each beginning with the vertices being placed in random locations. Once placed, the vertices are moved[5] under several iterations. In an iteration, each vertex is moved, acting under the influence of forces produced by its neighbors. (All of the vertex positions are updated simultaneously.)

At the end of each trial, the length of each edge is observed; these lengths are accumulated across trials. In the end, the only result of all of the trials is the average length obtained by each of the edges.

For each edge, its score is a comparison of its average length with the those of its neighborhood. Those edges with high scores, that is, those that are exceptionally long, are deemed to be intercluster edges, and are deleted. After deletions, the resulting components produce putative clusters.[6] Finally, some cleanup of the clusters is applied.

Each of these steps is amplified in the sections immediately below.

[5] The author experimented with both Gauss-Seidel iteration and Jacobi iteration. The question was whether or not all positions should be updated simultaneously, or whether they should be updated as they are computed, with the results immediately available for computing the other positional updates. It was quickly clear that updating all at once (Jacobi iteration) was the best choice. This may seem like a no-brainer, but the author has found differently for other algorithms.

[6] Unlike recursive spectral bisection, the approach here attempts to perform all division into clusters at a single time. In fact, the author experimented with repeating the process in an attempt to find subclusters within the clusters — doing recursion, in other words. It was not only unnecessary, but also not a good idea: most further divisions within clusters found at the top level turned out to be inappropriate.

```
normalize edge weights;
repeat $t$ times
{
      set all vertices to random locations;
      repeat $s$ times
      {
            move vertices according to the forces on them;
      };
      record lengths of each edge;
};
score edges by length averaged over the $t$ trials;
cut edges with high score;
form putative clusters from components;
cleanup clusters;
```

Figure 2. An Outline of the Algorithm. This is applied to each component of the graph separately.

2.2 Multiple models and notation

The author considered several models, each giving different "forces" to update the locations in an iteration. In each case, the iteration that takes one configuration to the next is accomplished by a linear transformation of the vertex positions. The models are described in sections below.

For each, let the graph have vertices 1, 2, …, *n*, with physical locations $\mathbf{p}_1,\ \mathbf{p}_2,\ ...,\ \mathbf{p}_n$. In the event that the locations are scalar quantities, we will denote the location of vertex *i* by x_i, and express the location of all vertices by the (column) vector $\mathbf{x} = (x_1, x_2,\ ...,\ x_n)^T$.

Let the edge (*i*, *j*), joining vertices *i* and *j*, have a weight of $w_{i,j}$. For notational purposes, a missing edge may be treated as having a weight of zero. Let $d_i = \sum_{1 \le j \le n} w_{ij}$ denote the (weighted) valence of the *i*th vertex. The set of vertices adjacent to vertex *i* will be denoted by N_i.

We will only consider undirected graphs, and will not distinguish between (*i*,*j*) and (*j*,*i*). We will also assume that the graph under consideration has only one component, since each component of a multi-component graph will be dealt with independently.

2.3 Model 1: a spring model

In this physical model, edge (i, j) may be represented by a spring of spring constant $k_{ij} = w_{ij}$.

By Hooke's law, the spring on edge (i,j) exerts a force of $-k_{ij}(\mathbf{p}_i - \mathbf{p}_j)$ on vertex i (and its negative on vertex j). Combining the effects of all neighbors, the force on vertex i is $-\sum_{j=1}^{n} k_{ij}(\mathbf{p}_i - \mathbf{p}_j)$. The purpose of each iteration is to move each vertex in the direction of, and in proportion to, the force on it; that is, an iteration adjusts vertex i by $\Delta\mathbf{p}_i = -\mu\sum_{j} k_{ij}(\mathbf{p}_i - \mathbf{p}_j)$, for some gain μ.

Note that each dimension in physical space acts independently, so that it is not necessary to work in more than one dimension.[7] Accordingly, let each vertex have a scalar location, and let those locations be contained in the single vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$.

In terms of this vector, an iteration produces a positional change $\Delta\mathbf{x} = -\mu L\mathbf{x}$, where

$$L = \begin{pmatrix} d_1 & -w_{12} & \cdots & -w_{1n} \\ -w_{21} & d_2 & \cdots & -w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -w_{n1} & -w_{n2} & \cdots & d_n \end{pmatrix}$$

is the Laplacian[8] of the graph. Incorporating the position increment, a convenient description of an iteration is $\mathbf{x}' = M\mathbf{x}$, where $M = I - \mu L$.

[7] Working in d-dimensional space, rather than 1-dimensional space, is likely to be as useful as taking d times as many random starts in 1-dimensional space, which is insufficient to justify the added complexity here. (It is possible that working in higher dimensions to get fewer random starts may offer benefit in some situations.) Despite being confined to one dimension, these ideal vertices can move through each other.

[8] Using the terminology of Bollobas [1998], this is the *combinatorial* Laplacian, otherwise known as the unnormalized Laplacian, the Kirchhoff matrix, and the admittance matrix. A related matrix, called the analytical (or normalized) Laplacian, is obtained by multiplying each row and column by the reciprocal of the square root of the diagonal element. The Laplacian matrix of a graph and the Laplacian operator in PDEs are related. In each case, $L\phi(x)$, that is, the Laplacian operating on some function ϕ, produces a function that is the difference between $\phi(x)$ and the average of $\phi(x)$ over the neighborhood of x. Consult the "discrete Laplace operator" entry in Wikipedia for exposition on this.

The gain μ is chosen to be as aggressive as possible, without causing oscillatory or other undesirable behavior. See Appendix A for a discussion of setting the gain.

2.4 Model 2: a barycentric model

Having established a model based on springs, one can now work directly from the iteration matrix $M = I - \mu L$ and abandon the literal spring analog. For purposes of stability, the gain is constrained by the highest valence in the entire graph (Appendix A), reducing the speed of "convergence." One might choose, instead, to update each vertex *i* using a tailored gain μ_i that depends only on the neighborhood of *i*, rather than the entire graph, so that $x_i' = x_i - \mu_i \sum L_{ij} x_j$. In particular, one may chose $\mu_i = 1/(d_i + 1)$.

The result of this choice is a new variant, in which the iteration matrix is

$$M = \begin{pmatrix} 1/(d_1+1) & w_{12}/(d_1+1) & \cdots & w_{1n}/(d_1+1) \\ w_{21}/(d_2+1) & 1/(d_2+1) & \cdots & w_{2n}/(d_2+1) \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}/(d_n+1) & w_{n2}/(d_n+1) & \cdots & 1/(d_n+1) \end{pmatrix}.$$

It is still the case that an iteration is described by $\mathbf{x}' = M\mathbf{x}$. Note that an iteration causes a vertex to take on the location that had been the (weighted) barycenter of its one-hop neighborhood, that is, the new location is the (weighted) average of its old location and the old locations of its neighbors.

All right eigenvalues of *M* are less than or equal to 1. (This is physically obvious: as each vertex is placed at the center of its neighborhood, the distances must contract.) The only right eigenvector that achieves an eigenvalue 1 is the all 1s vector, which simply maintains the origin of the entire system.[9]

2.5 Other models

The author tested a variety of other models, each differing only in the definition of the iteration matrix *M*. One considered by the author was a variation of the barycentric model, in which the individual gains were set to $\mu_i = 1/d_i$. The result was an iteration matrix of

[9] If more than one component were present, multiple eigenvectors (the indicators of the components) would have an eigenvalue of 1.

$$M = \begin{pmatrix} 0 & w_{12}/d_1 & \cdots & w_{1n}/d_1 \\ w_{21}/d_2 & 0 & \cdots & w_{2n}/d_2 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}/d_n & w_{n2}/d_n & \cdots & 0 \end{pmatrix}.$$

Here, an iteration gives each vertex the (weighted) mean of its neighbors' former locations, ignoring the old location of the vertex.[10] For many tests, this performed similarly to the barycentric model introduced above; for graphs with tree sections (such as Figure 6 later in the paper), it performed poorly.

2.6 The general model[11]

In summary, the bulk of the proposed algorithm is based on iteratively applying a linear transformation to an initially-random position vector **x**, in the form of multiplying by a matrix *M.* The iteration matrix is characterized as sparse (if the graph is), and as having a single (right) eigenvector with eigenvalue 1, and nonnegative entries, so that *M* is row-stochastic. In particular, multiplication by *M* moves each x_i to a convex combination of old positions of it and its neighbors; that is, $\mathbf{x}' = M\mathbf{x}$ produces $x_i' = c_{ii}x_i + \sum_{(i,j)\in E} c_{ij}x_j$, with $c_{ii} + \sum_{(i,j)\in E} c_{ij} = 1$ and $c_{ij} \geq 0$ for all *i* and *j*, for some set of coefficients $\{c_{ij}\}$. So, in some sense, one could say that these are all barycenter-seeking clustering approaches.

[10] This is reminiscent of Wu and Huberman's [2004] iterative solution to finding voltages for each node, and to the general iterative approach to finding the electric potential, satisfying $\nabla^2 V = 0$ in a volume, with specified values at the boundary.

[11] One model that the author considered and rejected was $M = I - \mu\mathcal{L}$, where

$$\mathcal{L} = \begin{pmatrix} 1 & \frac{-w_{12}}{\sqrt{d_1 d_2}} & \cdots & \frac{-w_{1n}}{\sqrt{d_1 d_n}} \\ \frac{-w_{21}}{\sqrt{d_2 d_1}} & 1 & \cdots & \frac{-w_{2n}}{\sqrt{d_2 d_n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{-w_{n1}}{\sqrt{d_n d_1}} & \frac{-w_{n2}}{\sqrt{d_n d_2}} & \cdots & 1 \end{pmatrix}$$

is the "analytic" or "normalized" Laplacian. This is something of a symmetrized version of the barycentric iteration matrix, falling between the barycentric and spring models. It's performance was similar to the springs model, but slightly inferior. It was also more difficult to interpret, and did not satisfy the general description that applied to the other models.

2.7 Normalizing weights

Before beginning, the edge weights are multiplicatively scaled so that the average weight is 1. This is important because the weights are combined with constants that are independent of the graph weights.

2.8 Random starts

For each start, the elements of **x** are chosen randomly. In the author's implementation, they are samples of independent normal(0,1) variates, generated using the Box-Muller algorithm[12]. One may choose to remove the mean of **x** (the contribution of the all-ones vector), but this is unnecessary, since it only changes the mean of the result. (It may improve the numerical stability.) The advantages of using normal samples are that any linear function of them will be normal as well and, since the starting coordinates are i.i.d. normal, any convex combination of them will have the same distribution.

2.9 Cutting criterion

Clusters are obtained by deleting relatively long edges, producing components that are then declared clusters.

Let the average length of edge (*i*, *j*), taken over all of the starts, be denoted by a_{ij}. Edge (*i*, *j*) is evaluated by computing its score $S_{ij} = a_{ij} - \overline{a}_{ij}$, where $\overline{a}_{ij}$ is some weighted average of a_{ij} over a neighborhood of the graph centered on (*i*, *j*). The (spatial) average is intended to measure what is "normal" for that region of the graph; those edges with a large score are deemed longer than average, and may be cut. In general, one could remove edges above a threshold of one's choosing, but the author found it sufficient to drop those edges whose scores were positive, that is, the author removed each edge with $a_{ij} > \overline{a}_{ij}$.

Practicality of this approach requires that the averages can all be computed in a time that scales no worse than $|V| + |E|$. Fortunately, this is easy if one is not too picky about the definition of the average. For each vertex, one may calculate an average $\mathcal{V}_i^{[1]} = \frac{1}{|N_i|}\sum_{j \in N_i} a_{ij}$; the superscript here indicates an average of the 1-hop neighborhood. (Recall that N_i denotes the neighbors of vertex *i*.) Given $\mathcal{V}_1^{[1]}$, $\mathcal{V}_2^{[1]}$, ...,$\mathcal{V}_n^{[1]}$, one can get a 1-neighborhood average of edges about edge (*i*,*j*):

[12] A good exposition of the Box-Muller transformation for producing normal samples is offered by Wikipedia.

$$\mathcal{E}_{ij}^{[1]} = \frac{|N_i|\mathcal{V}_i^{[1]} + |N_j|\mathcal{V}_j^{[1]} - a_{ij}}{|N_i| + |N_j| - 1}.$$

This process can be repeated to obtain averages over larger areas: in general, for $k > 1$,

$$\mathcal{V}_i^{[k]} = \frac{1}{|N_i|}\sum_{j \in N_i} \mathcal{E}_{ij}^{[k-1]} \quad \text{and} \quad \mathcal{E}_{ij}^{[k]} = \frac{|N_i|\mathcal{V}_i^{[k]} + |N_j|\mathcal{V}_j^{[k]} - \mathcal{E}_{ij}^{[k-1]}}{|N_i| + |N_j| - 1}.$$

Note that for $k > 1$, some edges get to make more contributions to $\mathcal{E}_{ij}^{[k]}$ than others.

A direct computation of every $\mathcal{E}_{ij}^{[1]} = a_{ij} + \sum_{k \in N_i - j} a_{ik} + \sum_{k \in N_j - i} a_{jk}$, would take on the order of $\sum N_i^2$ work, but the trick of performing the vertex averages followed by the edge averages takes only $|V| + |E|$ work. This local averaging is similar to the label-passing scheme of Raghavan *et al.* [2007].

The author chose to use $\bar{a}_{ij} = \mathcal{E}_{ij}^{[1]}$, the simplest case. In other words, edges with lengths larger than their 1-neighborhood average are cut.

2.10 Slackening midway

The author discovered that frequently two groups that should have been separate clusters remained joined by a few recalcitrant edges. His solution was to "slacken" long edges after half of the starts by setting their weights to zero. ("Long" edges here are those whose scores are positive.) At that point, all edge length averages are set to zero and the starts are resumed. The effect of setting some weights to zero is that those edges don't resist being stretched.

Effectively, then, there are two phases of starts: the first phase with the weights set in the obvious way, and the second phase with some edges having zero weight, those edges being the ones identified by the first phase as having edges larger than their neighborhood averages.

The two phases may appear to be merely two passes of clustering, one inside another. The difference is that actual cutting (at the end of the second phase) is based on averages $\{\bar{a}_{ij}\}$ that include the slackened edge lengths.

2.11 Cleanup phase

Edges that are longer than their local average are cut, producing components that form tentative clusters. These clusters can be cleaned up a bit, motivated by an idea suggested by Rattigan, Maier, and Jensen [2007]. They suggest cluster improvement by "modal reassignment," in which one examines the vertices at random, reassigning them to the clusters to which they have the most adjacency.[13] (This is also the essence of Raghavan *et al.* [2007].) This seems a bit too chummy for the author's application. In the author's implementation, he examines each vertex *v* deterministically, assigning it to a neighboring cluster *C* if *v* has at least twice as many neighbors in *C* as it does in any other cluster. This process is repeated a fixed number of times.

2.12 Co-pendant no more?

As the examples later will illustrate, barycentic clustering may be highly influenced by pendant vertices[14] when many adjoin the same vertex. This may be good, if one wants to find clusters of a hub-and-spoke nature. But if this is not the goal, then it is a good idea to ignore them. Consequently, the author chose to offer the option to keep or ignore pendant vertices in the clustering process. If the option to ignore pendant vertices is chosen, pendant members of clusters are also stripped from the clusters during cleanup.

[13] This differs somewhat from k-means or k-medoids, in which reassignment is based on distance from a single point (k-means) or member (k-medoids). Here, reassignment considers distance to the entire membership of a cluster; moreover, it is based on adjacency, a graph-specific view.

[14] A pendant vertex is one that has a single neighbor.

3 Some experiments

3.1 Examples and comparison with Newman clustering

Figures 3 through 7 show the results of clustering sample graphs using barycentric clustering and spring clustering. Newman clustering [Newman 2003 and Clauset, Newman, and Moore 2004] is shown for comparison purposes.

Parameter choices for the barycentric clustering were 5 iterations per random start and 30 random starts. Spring clustering was conducted with 10 iterations per random start and 30 random starts. These choices are justified by results throughout this section. Note that the spring model has an effective gain that is less than the barycentric model (see Appendix A), so that more iterations are needed to obtain similar results.

In each figure, clusters are distinguished by giving each a color. Necessarily, the colors find themselves being reused. Clusters consisting of a single member are dimmed out, as are edges between clusters.

Figure 3 shows an example of clustering a document similarity graph (subsequently called "the document graph") that contains cliques and near cliques. The figure indicates reasonable (and similar) behavior for the two clustering approaches introduced in this paper: they have identified tight clique-like structures, and have not created the run-on clusters produced by the Newman clustering.

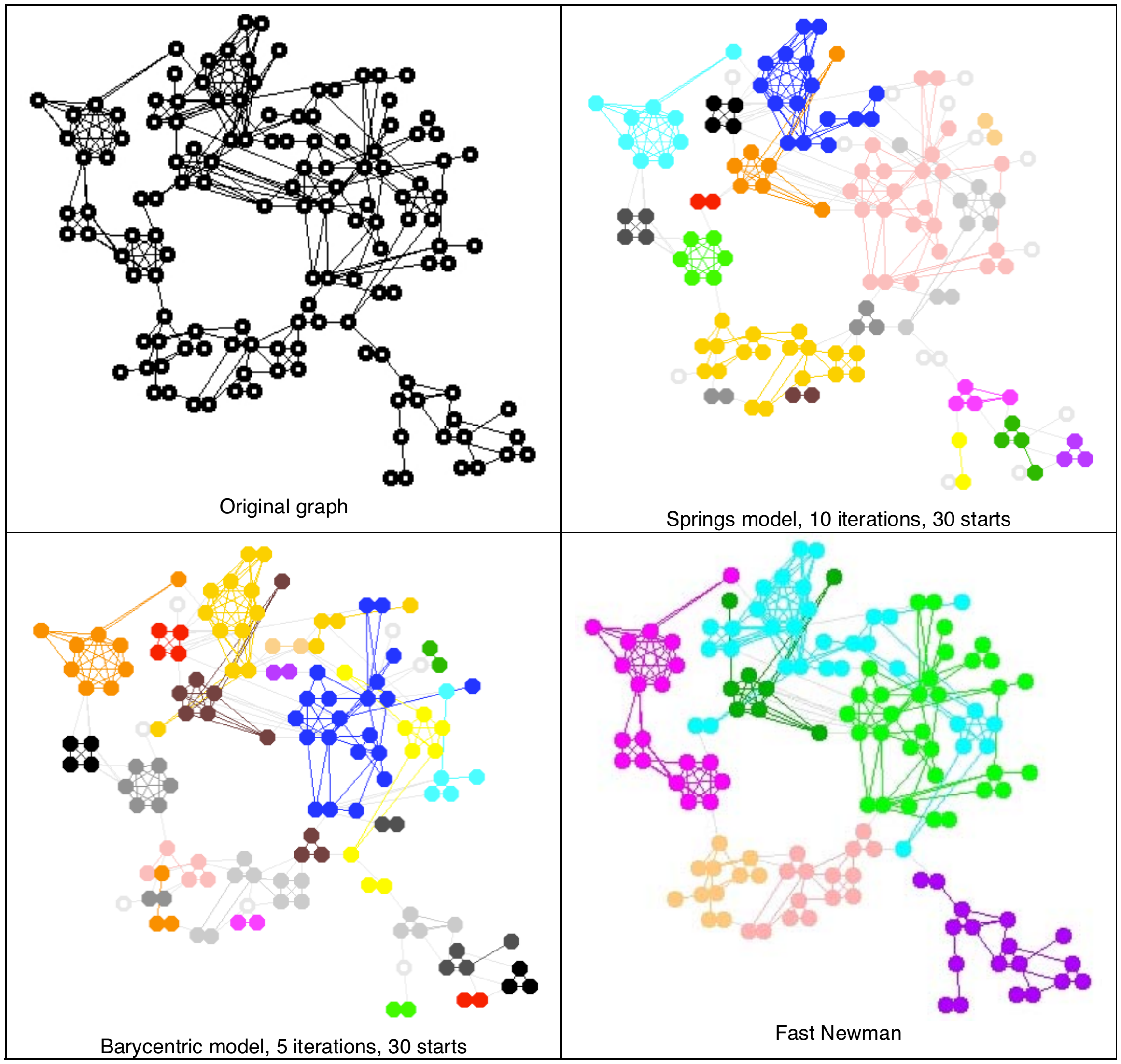
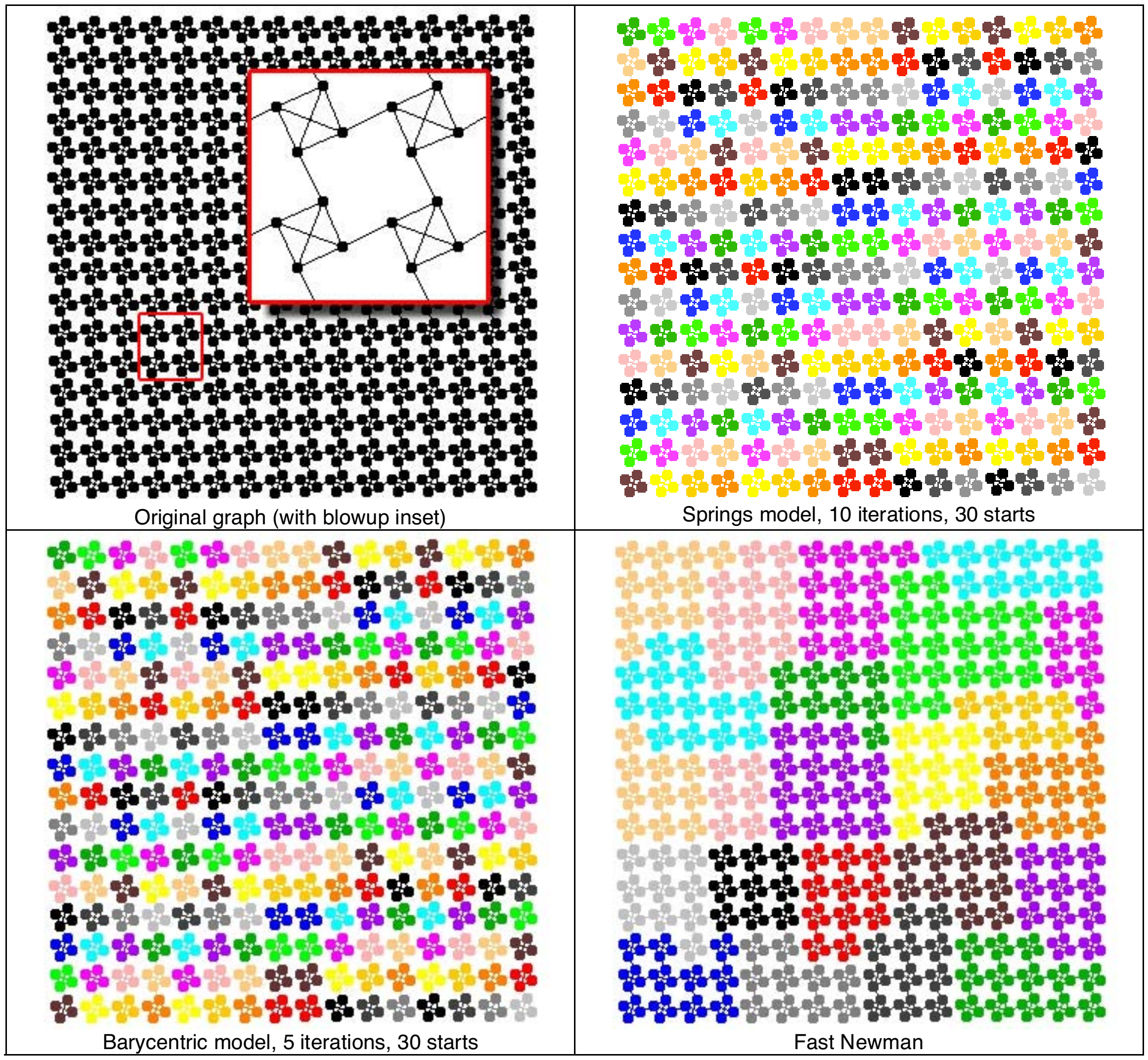

Original graph
Springs model, 10 iterations, 30 starts
Barycentric model, 5 iterations, 30 starts
Fast Newman

Figure 3. Results on the Document Graph. Each cluster is given a color, but note that colors may be reused. Clusters of a single vertex are dimmed, as are edges between clusters. Note the purple run-on cluster in the bottom right of the Fast Newman example.

Figure 4 shows the results of clustering a very contrived graph consisting of cliques of 4 members that are then interconnected. (This is subsequently called "the 4-cliques graph.) The springs and barycentric models recovered the cliques without error. The Newman method found the cliques, but joined them regionally. This is a well-known failure of Newman modularity (see Fortunato and Barthelemy 2007).



Original graph (with blowup inset)
Springs model, 10 iterations, 30 starts
Barycentric model, 5 iterations, 30 starts
Fast Newman

Figure 4. Results on the 4-Cliques Graph. Each cluster is given a color, but note that colors are reused. Clusters of a single vertex are dimmed, as are edges between clusters.

Figure 5 shows an example of a graph consisting of 30 groups of 30 vertices, with the edges suppressed in the figure. (This graph will be referred to as "the 30 groups of 30 graph".) Edges within and between the group vertices were constructed randomly as follows: For each possible intragroup edge, the edge was actually created with a 90% probability; for each possible intergroup edge, the edge was actually created with a 10% probability.

In this example, the groups were reconstructed flawlessly with the springs and barycenter methods. The Newman clustering results had many errors, resulting in clusters of size 29 through 117, with only 2 having 30 members.

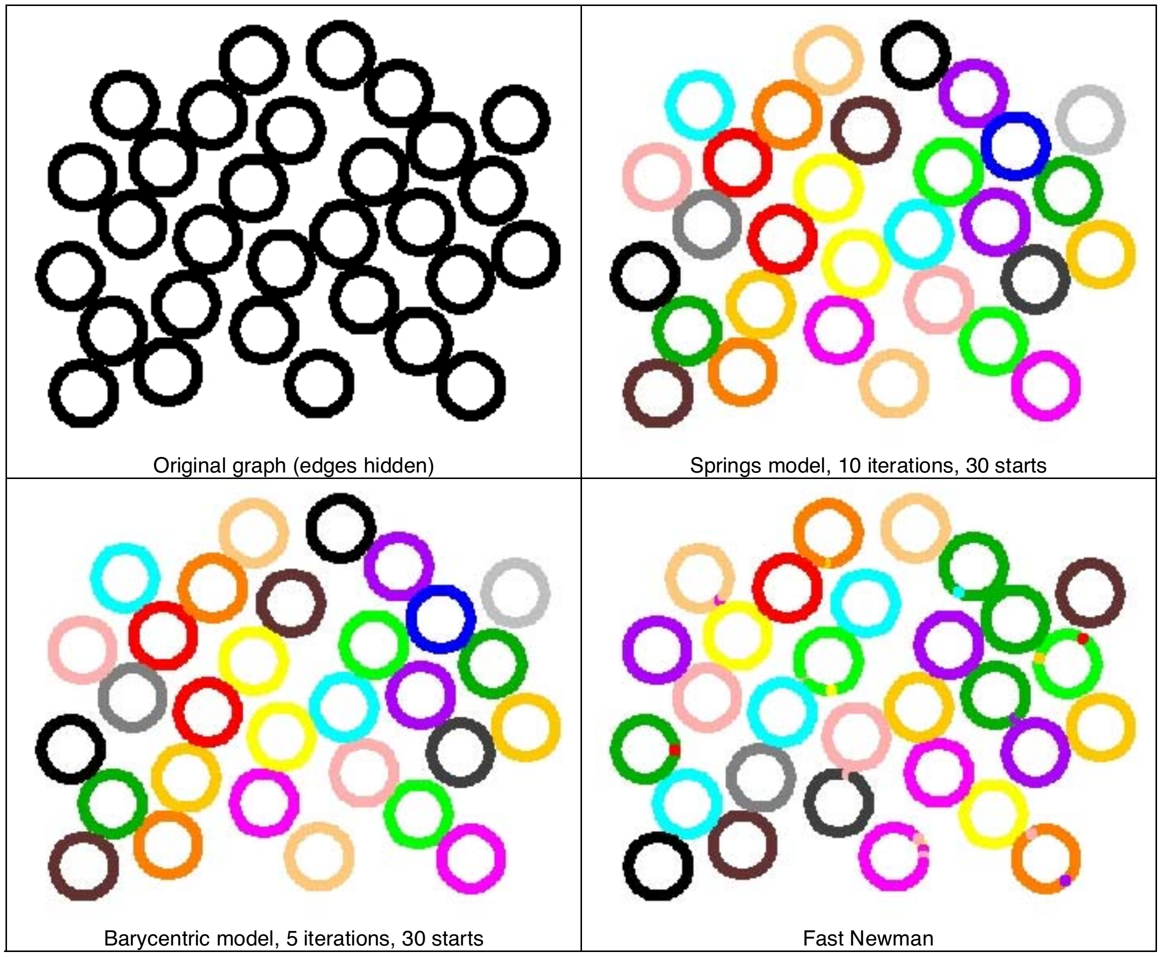


Figure 5. Results on a Graph of 30 Groups of 30 Vertices. The graph contains 900 vertices and 15,598 edges, which are hidden in the pictures. Each possible intragroup edge exists with a probability of 90%; each possible intergroup edge exists with a probability of 10%. Each cluster is given a color, but note that colors are reused.

Figure 6 shows an example with a graph that consists primarily of hub and spoke structures ("the hub and spokes graph"). The spring model fails to establish any meaningful clusters, finding only one rambling cluster, and single-member clusters on the periphery. Both the barycentric model and Newman clustering produce clusters that recognize hubs and their spokes. The barycentric model creates new clusters for each subsidiary hub, while Newman clustering may or may not include the subsidiary hub within the larger cluster. Newman clustering also includes connective vertices in adjacent hubs, which is arbitrary and questionable. (Note, too, the gray Newman cluster consisting of two subsidiary hubs with their spokes.)

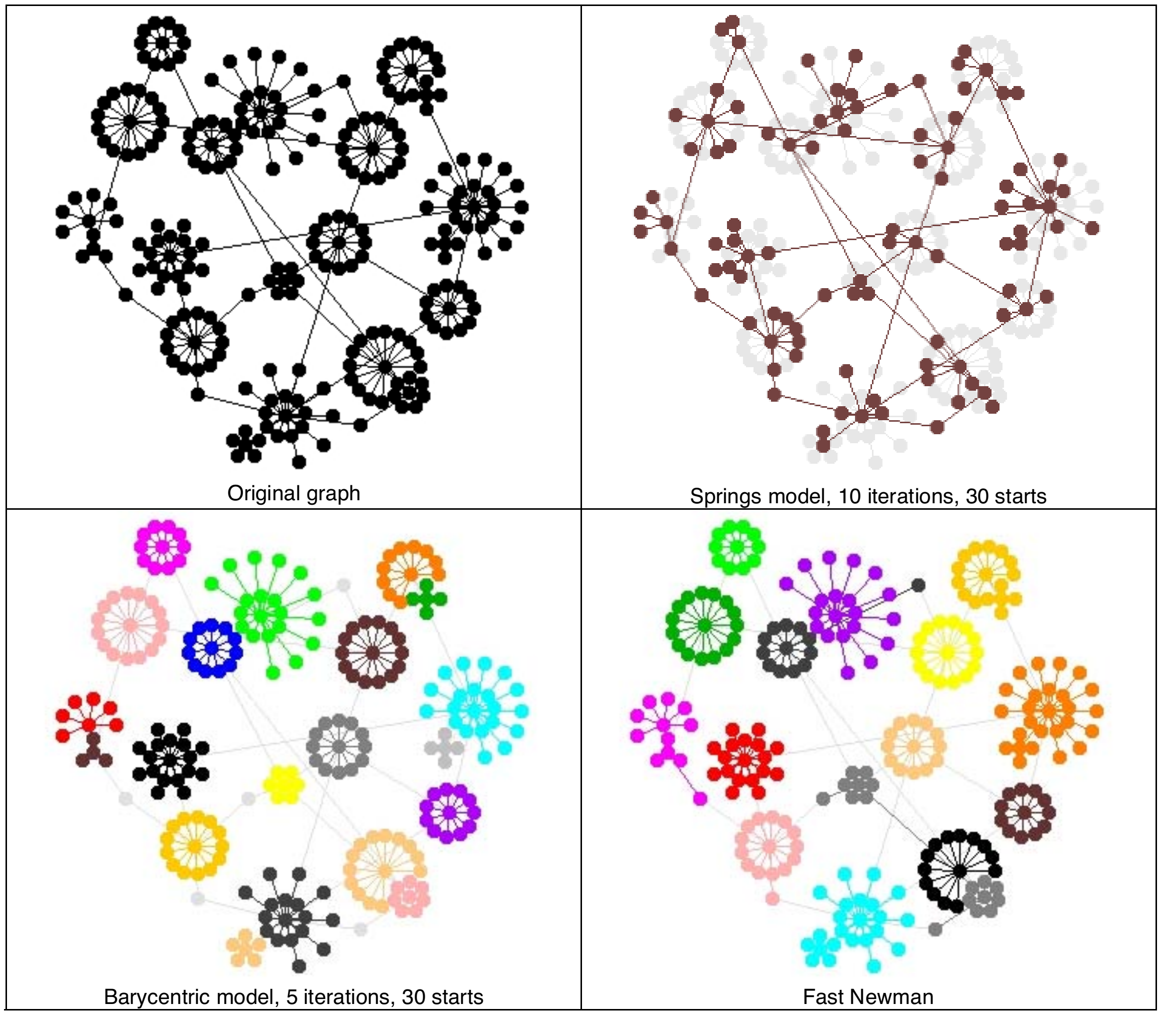

Original graph
Springs model, 10 iterations, 30 starts
Barycentric model, 5 iterations, 30 starts
Fast Newman

Figure 6. Results on Hub and Spoke Graph. Each cluster is given a color, but note that colors may be reused. Clusters of a single vertex are dimmed, as are edges between clusters.

Recall that the barycentric and spring algorithms are stochastic, so their results may vary from run to run. This is illustrated in Figure 7, in which four successive runs of the barycentric method are applied to same graph, with small variations in the resulting clusters. Variations can, of course, be much more profound than the ones shown here. The effect can be reduced by increasing the number of trials, thereby decreasing the score variance.

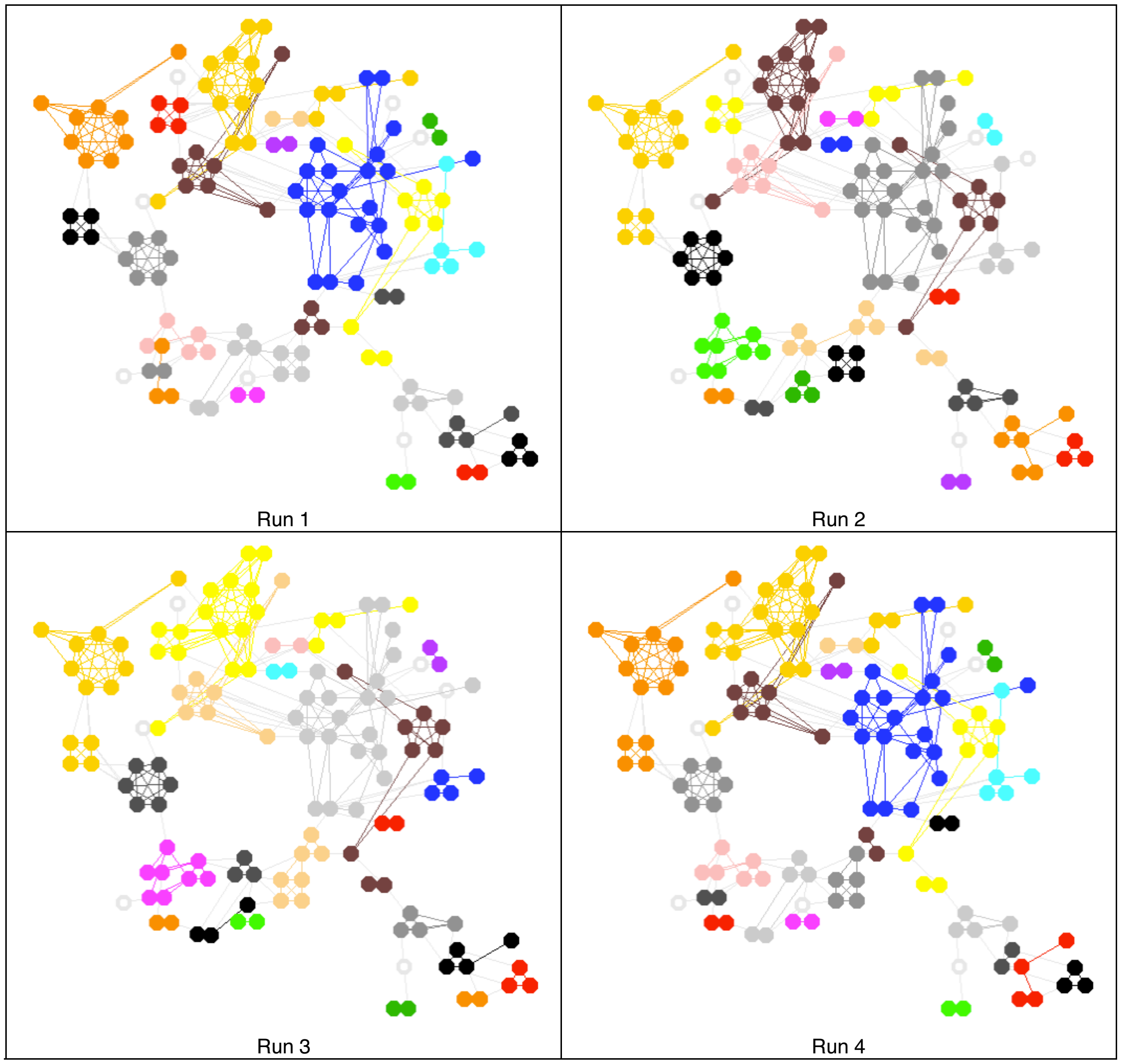


Figure 7. Four Runs of Barycentric Clustering on the Document Graph. This shows results of the same graph as Figure 3. Each cluster is given a color, but note that colors may be reused. Clusters of a single vertex are dimmed, as are edges between clusters.

3.2 Varying the number of iterations

Figures 8, 9, and 10 explore the effect of varying the number of iterations for the barycentric approach.

For the document graph case (Figure 8), only one run was conducted for each number of iterations. For the other examples, 100 runs were conducted and an average misclassification[15] was recorded.

The figures suggest that there is not terrific sensitivity to the parameter. For these examples, it would seem that anywhere from 2 to 20 iterations is fine. The author used 5 iterations for his other tests.

One remarkable thing is how effective just one iteration is. More than that, for the examples of the 4-cliques and the 30 groups of 30 graphs, trials of 100 runs never once produced a classification error when using 2 iterations.

[15] The misclassification values reported here resulted from a greedy algorithm that matched reported clusters with the ground truth groupings. Any vertex not matching properly was deemed an error, so that if two 30-member groups were combined, but otherwise correct, then 30 vertices would be marked as being in error.

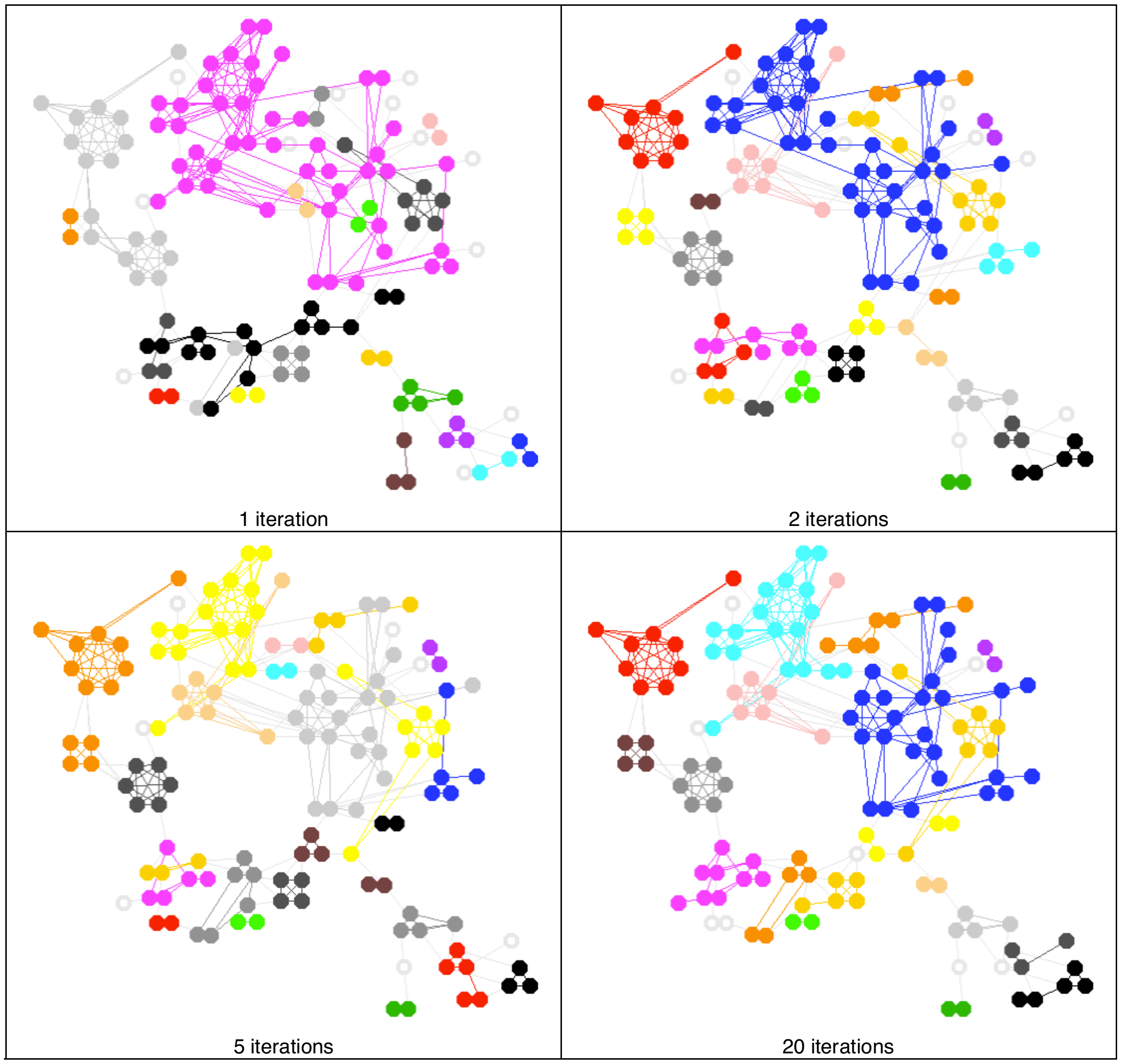


Figure 8. Changing Number of Iterations for Barycentric Clustering on the Document Graph. No slackening was done. This shows results of the same graph as Figure 3. Each cluster is given a color, but note that colors may be reused. Clusters of a single vertex are dimmed, as are edges between clusters.

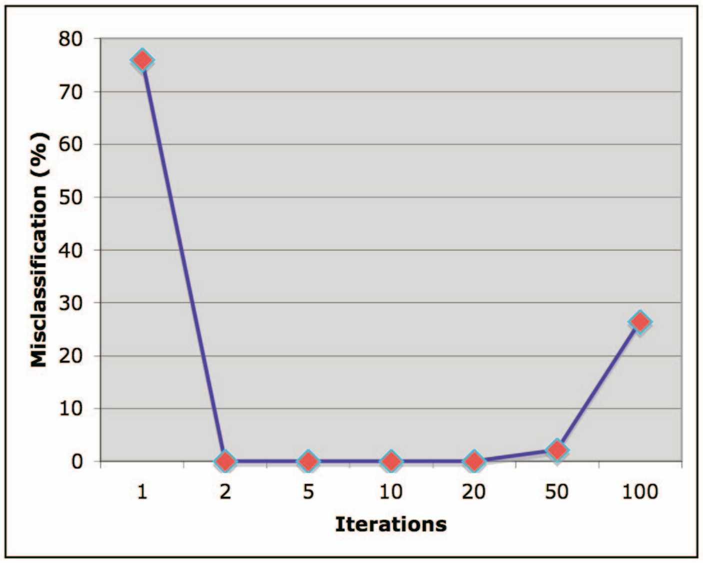


Figure 9. Misclassification Rate of Barycentric Clustering as a Function of Number of Iterations on the 30 Groups of 30 Graph. Clustering was conducted on the graph of Figure 5. One hundred trials of 30 random starts for each case. No errors occurred for 2 through 20 iterations.

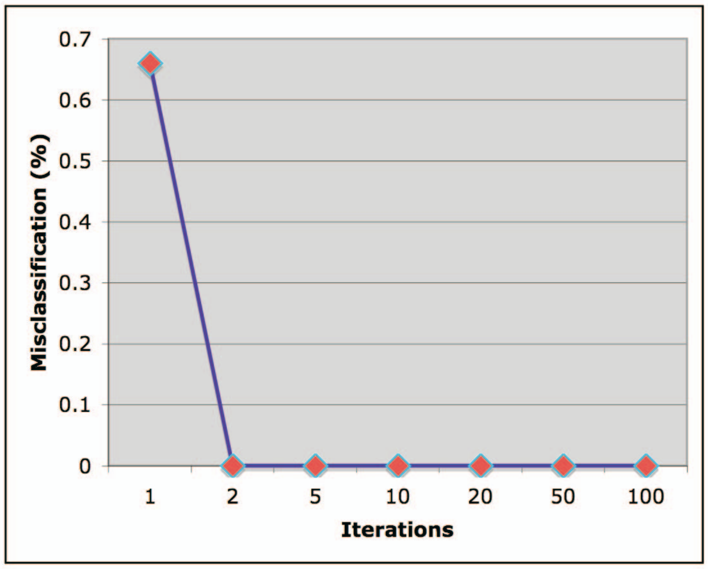


Figure 10. Misclassification Rate of Barycentric Clustering as a Function of Number of Iterations on 4-Cliques Graph. Clustering was conducted on the graph of Figure 4. One hundred trials of 30 random starts for each case. No errors occurred for 2 through 100 iterations.

Figures 11, 12, and 13 show the effects of varying the number of iterations for spring clustering. (The number of random starts was still fixed at 30.) Again, it seems that the number of iterations is not a critical parameter, suggesting that 10 through 20 iterations is fine. The author used 10 for his other results, twice as many as for the barycentric clustering. It is again remarkable that using only a few iterations results in something useful, though more iterations are needed for spring clustering than barycentric clustering.

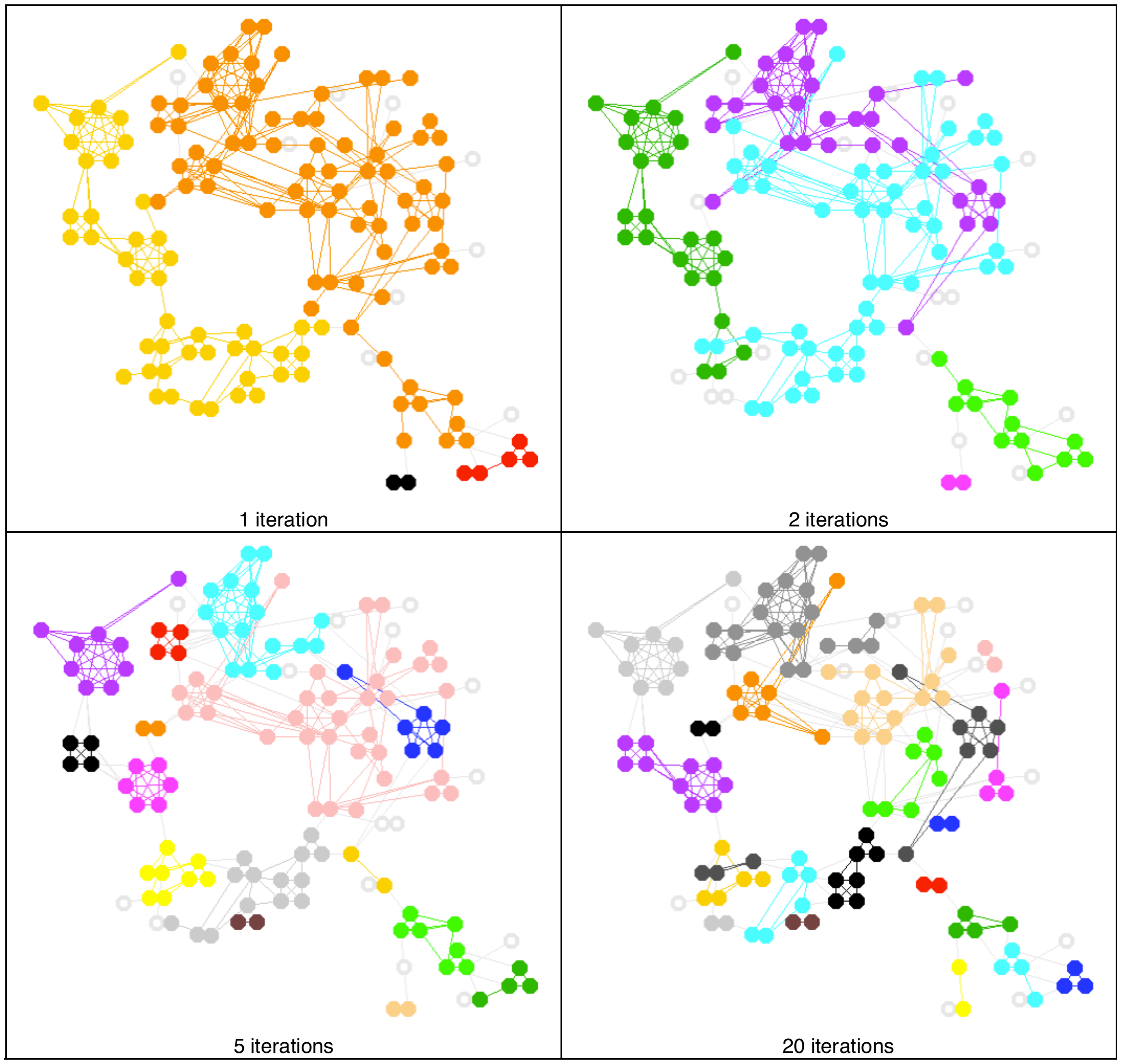


Figure 11. Changing Number of Iterations for Spring Clustering on the Document Graph. No slackening was done. This shows results of the same graph as Figure 3. Each cluster is given a color, but note that colors may be reused. Clusters of a single vertex are dimmed, as are edges between clusters.

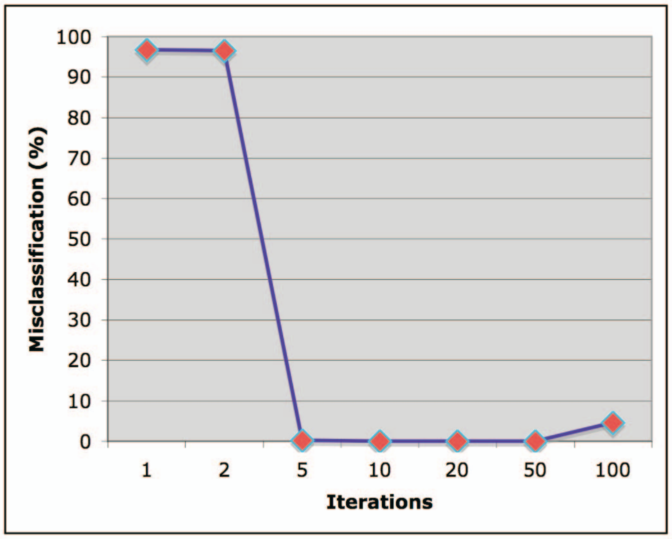


Figure 12. Misclassification Rate of Springs Clustering as a Function of Number of Iterations on the 30 Groups of 30 Graph. Clustering was conducted on the graph of Figure 5. One hundred trials of each case. No errors occurred for 10 through 50 iterations.

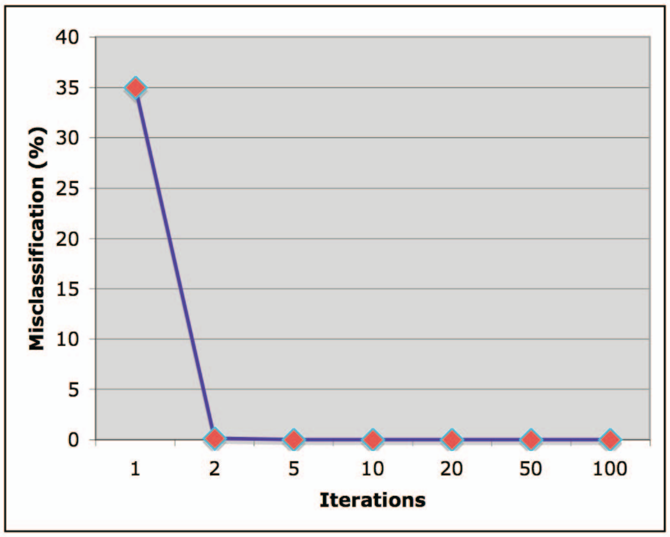


Figure 13. Misclassification Rate of Springs Clustering as a Function of Number of Iterations on the 4-Cliques Graph. Clustering was conducted on the graph of Figure 4. One hundred trials of each case. No errors occurred for 5 through 100 iterations.

3.3 Varying the number of random starts

Figures 15 and 16 briefly examine the role of the number of random starts for barycentric clustering, with the number of iterations per start fixed at 5. For these (very contrived) graphs, at least, 30 random starts was sufficient to produce no classification errors. It should be pointed out that the first half of the starts are used to do slackening, then the lengths are reset, and the second half form the final scores used for edge cutting.

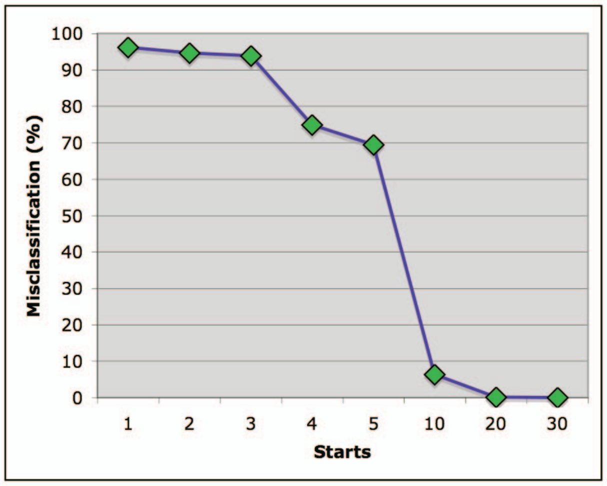


Figure 15. Barycentric Clustering Errors as a Function of Number of Starts for the 30 Groups of 30 Graph. This shows the number of vertices misclassified (as a total percentage of vertices) over 100 runs of the clustering algorithm, with 5 iterations per start. The 20 starts case had 1 run in which two clusters were incorrectly joined; the 30 starts case had no misclassifications.

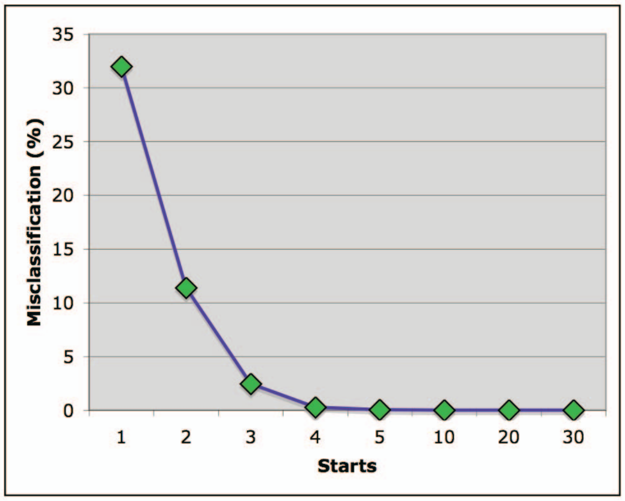


Figure 16. Barycentric Clustering Errors as a Function of Number of Starts for the 4-Cliques Graph. This shows the number of vertices misclassified (as a total percentage of vertices) over 100 runs of the clustering algorithm, with 5 iterations per start. The case of 10 starts had a single join error in one run; the cases with more starts had no errors.

3.4 Processing speed

Figure 17 shows the time taken for the algorithm to perform clustering on a variety of graphs, including creating structures, doing the actual clustering, and recording the result. The figure somewhat supports the claim that work scales with the size of the graph, as measured by the number of edges. In general, timing Java programs is difficult, because it is nearly impossible to control the effects of its multiple threads, including garbage collection. The author made little attempt to squeeze speed out of his implementation, and it is likely that a factor of two improvement could be obtained.

Graph Description	Number of Vertices	Number of Edges	Time (sec)	Work (μsecs / edge)
Document similarity graph of Figure 3	135	311	0.008	26
4-Cliques of Figure 4	1024	2106	0.043	20
30 groups of 30 of Figure 5	900	15598	0.175	11
Uniform random	900	15598	0.2	13
6 joined triangular grids of equal size	7633	22050	0.39	18
Uniform random	10000	40000	0.88	22
Uniform random	1600	140155	4.5	32
40 groups of 40, 50% prob of intracluster edges, 10% prob of intercluster edges	1600	140155	3	21

Figure 17. Clustering Time for a Variety of Graphs. Barycentric clustering with 30 random starts, 5 iterations per start. Tests performed in Java 1.5 on a 2.16 GHz Intel Core 2 Duo Macintosh.

4 Identifying tight clusters

An important problem is to locate highly coherent subgraphs quickly. The clustering methods outlined above are very fast, but partition the graph into clusters that may or may not be tight groups. Highly coherent subgraphs may be identified by applying some measure of tightness to each cluster obtained from the clustering algorithm.

This section proposes a few appropriate measures. The intent is to reward both density and size, since large, dense subgraphs are often of most interest. Throughout, let V_s and E_s denote the vertex and edge set, respectively, of the subgraph being examined.

$$\boxed{\text{Density coherence: } C_D = \frac{|E_s|}{|V_s|-1}.}$$

This pays no attention to the rest of the graph or to edge weights. It ranges from 1 to $|V_s|/2$. Note this is $|V_s|/2$ times the usual density; it gives more significance to larger groups.

$$\boxed{\text{Weighted density coherence: } C_W = \frac{1}{|V_s|-1}\sum_{(i,j)\in E_s} w_{ij}.}$$

This a generalization of density coherence, and rewards a cluster for having higher intracluster edge weights.

$$\boxed{\text{Score coherence: } C_S = -\frac{1}{|V_s|-1}\sum_{(i,j)\in E_s} S_{ij}.}$$

Recall that S_{ij} is the score for edge (*i*,*j*), which is larger for edges that are long relative to their neighbors.

Length coherence: $C_L = \ln\left[\dfrac{|E_s|}{(|V_s|-1)\displaystyle\sum_{(i,j)\in E_s} a_{ij}}\right]$.

Recall that a_{ij} is the average length for edge (*i*,*j*), which is smaller for edges that should be within clusters. The log is being taken here to keep the values from getting too big.

On the whole, the author found the density coherence (weighted and unweighted) and the length coherence measures to be useful. The coherence measure that used the edge scores was not very good, since these scores tend not have large magnitude for the intracluster edges.

The author implemented the unweighted density and length coherence measures, and presented the results rounded to the nearest integer. An example is pictured in Figure 18, in which the density coherence values are displayed. (Clusters with values of 1 are not labeled.) As this figures suggests, large numbers indicate large, tight subgraphs. Finding such tight subgraphs by other means is usually quite expensive.

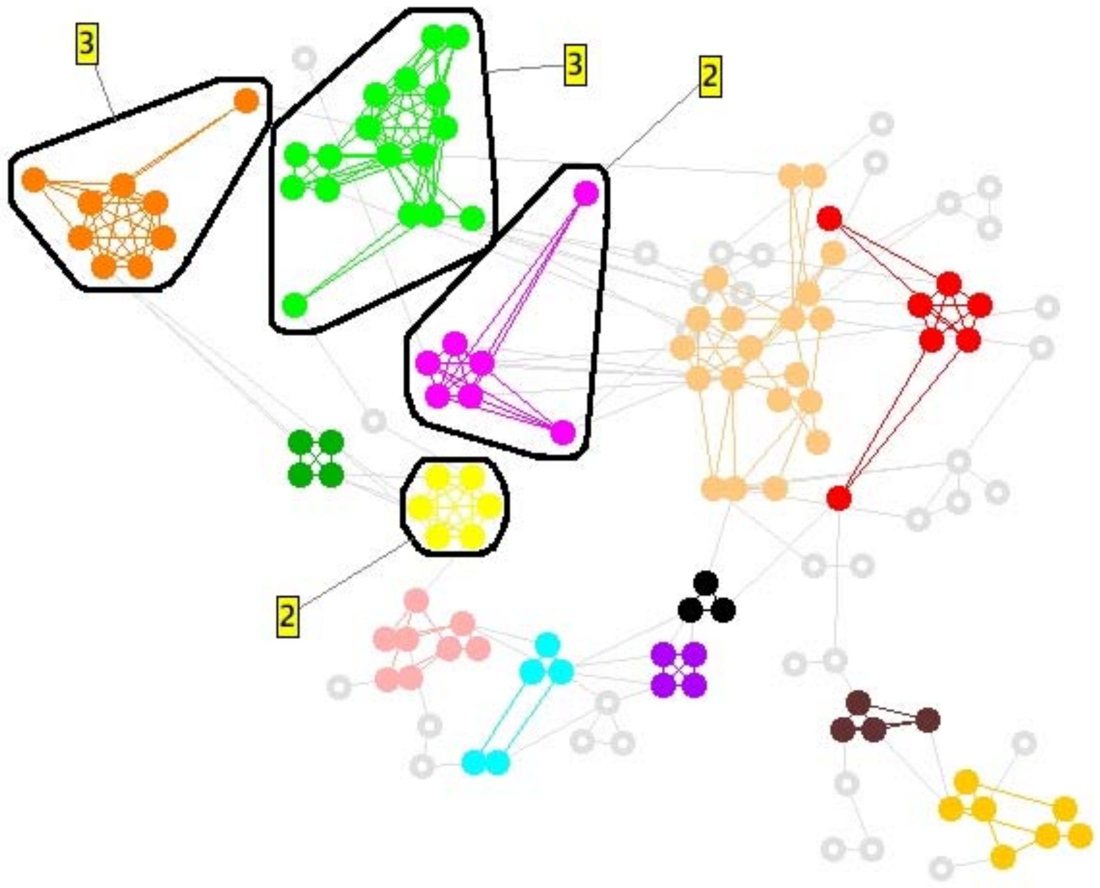


Figure 18. Clustering With Density Coherence Values. The values are rounded to the nearest integer. Those clusters lacking a label have a (rounded) value of 1.

5 Things to do

The edge score, merely a difference between the (average) edge length and the lengths of edges in the area, could be improved. Making the significance of an edge length be normalized by an average in its local area seems useful, since graphs can ramble on, joining subgraphs of very different nature together. But as it stands, for example, a graph consisting entirely of an unweighted clique would lead to edges that would be cut or not based entirely on numerical error. This example is easily fixed, but a more general improvement would be a good idea.

Each trial involves setting vertices to random positions and then releasing them. Is there some better choice of initialization or constraint? Is there, in fact, a better approach more akin to the original idea of pulling on vertices?

Steve Knox suggested having negative weights to let non-adjacent vertices repel each other. Does this offer any improvement?

The cleanup phase, in which vertices are coerced into changing clusters on the basis of adjacency, could be better. As implemented by the author, a vertex was moved to cluster *c* if it was found to have neighbors that occurred at least twice as often in *c* as in any other cluster. One should be able to do better by considering average lengths of edges (if not the edge scores).

The "slackening" process could use some improvement. As it stands, half of the starts are conducted, the long edges are given zero weight, and scores are computed based on the second half of the starts. Should the starts be evenly split, as they currently are, between before and after slackening? What is the best way to slacken? Should it be graded, rather than binary? Should it be incremental and conducted throughout the starts?

The testing conducted here is suspect, since it was confined to a small number of graphs, most of which were highly contrived. As such, there is a good chance that the tuning of parameters (and other decisions) were too closely fit to those examples and are not appropriate for more general application. The most obvious question (beyond general utility): are the number of iterations and number of random starts chosen here universal, or do they depend on the graph being examined? Experimentation on a more representative collection of graphs will tell.

Finally, comparison of this algorithm with other linear-time approaches is in order, particularly against the methods of Raghavan *et al.* [2007] and Rattigan, Maier, and Jensen [2007].

6 Concluding remarks

Modulo the reservations expressed in the preceding section, the algorithm presented here provides a way of partitioning a graph that performs well on test examples, and does so in a manner that scales linearly with the graph size.

The proposed algorithm avoids many common shortcomings of other approaches: it does not create run-on clusters, nor does it take single orphans connecting two or more legitimate clusters and randomly assign them to one of the neighboring clusters. It does not join regions of clusters merely because the graph gets larger, as does Newman clustering. Further it makes no assumptions about the number of clusters to be found or of their relative size, either or both of which are fundamental to many clustering approaches.

In addition to partitioning the entire graph, this algorithm may be used to find tight subgraphs by applying a coherence measure (given here or elsewhere) to each of the subgraphs obtained in the partition. The combined approach may well be faster than any other method of locating meaningful cohesive groups.

Acknowledgements

The author would like to thank Steve Knox for editing the manuscript and for offering helpful suggestions, including the use of Box-Muller random sample generation.

Appendix A: Picking the gain for the springs model

To avoid oscillation, one wants the diagonal elements of M to remain positive, suggesting that $\mu < 1/d_{\max}$ would be a good idea, where $d_{\max} = \max\{d_1, \ldots, d_n\}$ is the maximum weighted valence. But this might not be sufficient.

The Laplacian has nonnegative eigenvalues, exactly one of which is zero (since the graph has only one component). Suppose the eigenvalues are $0 = \lambda_1 \le \lambda_2 \le \ldots \le \lambda_n$. Then M has eigenvalues $1 - \mu\lambda_n \le 1 - \mu\lambda_{n-1} \le \ldots \le 1 - \mu\lambda_1 = 1$. For small gain, the action of the matrix is dominated by the eigenvectors associated with the eigenvalues $1 - \mu\lambda_1 = 1$ (which just preserves the mean, and is unimportant) and $1 - \mu\lambda_2$. This will remain the case as long as $|1 - \mu\lambda_2| > |1 - \mu\lambda_n|$, that is, as long as $\mu < 2/(\lambda_2 + \lambda_n)$. It is sufficient to choose

$$\mu < \frac{2}{\text{upperbound}\{\lambda_2\} + \text{upperbound}\{\lambda_n\}}.$$

Fiedler (1973) shows that $\lambda_2 \le \frac{n}{n-1} d_{\min}$, where $d_{\min} = \min\{d_1, \ ..., \ d_n\}$.

To bound the largest eigenvalue λ_n of *L*, suppose that the corresponding eigenvector is $\mathbf{v} = (v_1, \ ..., \ v_n)^T$ and *L* has been permuted such that $v_1 \ge |v_i|, \ i > 1$, (possibly by multiplying by –1). Let $m = v_1$. Since $\mathbf{v}$ is an eigenvector of eigenvalue λ_n, then $\lambda_n v_1 = d_1 v_1 - \sum_{j>1} k_{1j} v_j$, or $\lambda_n = d_1 - \frac{1}{m} \sum_{j>1} k_{1j} v_j$. Observing that $m \ge |v_i|, \ i > 1$, one gets the bound $\lambda_n \le 2\max\{d_i\}$. Anderson and Morley (1985) offer a slight sharpening of this bound, making it $\lambda_n \le \max\{d_i + d_j | (i,j) \in E\}$.

To keep $1 - \mu\lambda_2$ as the dominant eigenvalue of *M*, it is sufficient to choose $\mu = \frac{1}{d_{\min} + d_{\max}}$. Note that this also suffices to keep the diagonal elements of *M* positive.

Note that this gain is usually less than the gain used for the barycentric clustering,

Appendix B: Distribution of edge lengths

Consider the length of the edge (*i*,*j*), resulting from *s* iterations of any of the above models. Let $\mathbf{r}_k^T$ denote the *k*th row of matrix *M*, and let $\mathbf{X}^{(s)} = \left(X_1^{(s)}, \ X_2^{(s)}, \ ..., \ X_n^{(s)}\right)^T$ denote the position vector after *s* iterations. The set of initial positions $\mathbf{X}^{(0)}$ consists of independent *N*(0,1) variates.

After $s > 0$ iterations, we may write the difference between the vertex positions as

$$\Delta_{ij}^{(s)} = X_i^{(s)} - X_j^{(s)} = \left(\mathbf{r}_i - \mathbf{r}_j\right)^T M^{s-1} \mathbf{X}^{(0)}.$$

Note that $\Delta_{ij}^{(s)}$ is normal, and is characterized by

$$\mathrm{E}\{\Delta_{ij}^{(s)}\} = \left(\mathbf{r}_i - \mathbf{r}_j\right)^T M^{s-1} \mathrm{E}\{\mathbf{X}^{(0)}\} = 0$$

and

$$\begin{aligned}\mathrm{Var}\left\{\Delta_{ij}^{(s)}\right\} = \mathrm{E}\left\{\left(\Delta_{ij}^{(s)}\right)^2\right\} &= \mathrm{E}\left\{\left(\mathbf{r}_i - \mathbf{r}_j\right)^T M^{s-1}\mathbf{X}^{(0)}\mathbf{X}^{(0)T}\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\} \\ &= \left(\mathbf{r}_i - \mathbf{r}_j\right)^T M^{s-1}\,\mathrm{E}\left\{\mathbf{X}^{(0)}\mathbf{X}^{(0)T}\right\}\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right) \\ &= \left(\mathbf{r}_i - \mathbf{r}_j\right)^T M^{s-1} I \left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right) \\ &= \left\|\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\|^2.\end{aligned}$$

The length of edge (*i*,*j*) is $\delta_{ij}^{(s)} = \left|\Delta_{ij}^{(s)}\right|$. It has a central χ distribution, otherwise known as a semi-normal distribution, with an expectation of

$$\mathrm{E}\left\{\delta_{ij}^{(s)}\right\} = \sqrt{\frac{2}{\pi}}\;\left\|\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\| \tag{1}$$

and variance

$$\mathrm{Var}\left\{\delta_{ij}^{(s)}\right\} = \left(1 - \frac{2}{\pi}\right)\;\left\|\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\|^2.$$

After performing *t* trials, each with a new random start, we observe the average edge length $a_{ij}^{(s)} = \frac{1}{t}\sum_{\text{trials}}\delta_{ij}^{(s)}$. For *t* sufficiently large, we can approximate $a_{ij}^{(s)}$ as being normal with

$$\mathrm{E}\left\{a_{ij}^{(s)}\right\} = \mathrm{E}\left\{\delta_{ij}^{(s)}\right\} = \sqrt{\frac{2}{\pi}}\;\left\|\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\|$$

and

$$\mathrm{Var}\left\{a_{ij}^{(s)}\right\} = \frac{1}{t}\mathrm{Var}\left\{\delta_{ij}^{(s)}\right\} = \frac{1}{t}\left(1 - \frac{2}{\pi}\right)\;\left\|\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\|^2.$$

For a sufficient number of starts, the scores can be considered normal, but they are certainly not independent.

Appendix C: Markov interpretation of edge length

To interpret equation (1) for *the spring model*, note that each row vector $\mathbf{r}_i$ consists of positive values at position *i* and at all positions representing vertices adjacent to vertex *i*; the remaining values are zero.[16] In the unweighted graph case, $\mathbf{r}_i - \mathbf{r}_j$ has 1s in positions corresponding to neighbors of vertex *i* and not *j*, has –1s in positions corresponding to neighbors of vertex *j* and not *i*, has values that may be positive or negative at positions *i* and *j*, and has zero elsewhere. Applying $M^T = M$ to $\mathbf{r}_i - \mathbf{r}_j$ treats $\mathbf{r}_i - \mathbf{r}_j$ as a vertex position vector, and moves those positions according to the forces exerted by the springs. The more neighbors shared by vertices *i* and *j*, the smaller $\left\|\mathbf{r}_i - \mathbf{r}_j\right\|$ is, and the more applying *M* pulls them together.

More generally, since *M* is row-stochastic (each row of *M* is consists of nonnegative entries that sum to 1), one can interpret $\left(M^{s-1}\right)^T \mathbf{r}_i$ to be the occupancy probabilities of a Markov process on *n* states after $s-1$ steps, with initial probabilities $\mathbf{r}_i$ and transition probabilities M^T. Accordingly, $\mathrm{E}\left\{\delta_{ij}^{(s)}\right\} = \sqrt{\frac{2}{\pi}} \left\|\left(M^{s-1}\right)^T \left(\mathbf{r}_i - \mathbf{r}_j\right)\right\|$ measures the difference in occupancy probabilities that would result from the two initial distributions $\mathbf{r}_i$ and $\mathbf{r}_j$.[17]

Appendix D: Relation to recursive spectral bisection

Several workers (particularly Capocci *et al.* [2004]), have pointed out that spectral bisection can best be achieved by dividing at a large gap in some eigenvector of a matrix associated with the graph. We may be doing something similar. In our case, as *M* is applied repeatedly to an initial start **x**, the vector $M^s\mathbf{x}$ becomes dominated by the eigenvector of *M* with the largest eigenvalue (provided that it is not orthogonal to the initial vector), and looking for long edges in the graph is equivalent to looking for large differences between elements in $M^s\mathbf{x}$.

More specifically, in the case of the spring model, our choice of μ ensures that

[16] The positive value at position *i* assumes a properly chosen gain.

[17] One can actually use random walks for clustering. Harel and Koren [2001] model a random walk on the graph and use powers of the transition matrix to cluster directly by removing edges corresponding to small migration probabilities. Their method makes sense for their chosen application of Delaunay triangulation graphs, but does not apply well to graphs of larger degree. Brandes, *et al.* [2003] suggest something similar.

$$\frac{M^s\mathbf{x}}{(1-\mu\lambda_2)^s} \xrightarrow[s\to\infty]{} (\mathbf{F}^T\mathbf{x})\mathbf{F} + \frac{\mathbf{e}^T\mathbf{x}}{(1-\mu\lambda_2)^s}\mathbf{e},$$

where **F** is the Fiedler vector (eigenvector of the Laplacian with eigenvalue λ_2) and **e** is the all-ones vector. In a sense, the iterated vector of positions tends to the Fiedler vector, plus an inconsequential offset in the mean. (The mean drift can be removed entirely simply by removing the mean from the initial vector **x**.)

In the usual approach to recursive spectral bisection, a component of the graph is divided in two by determining its Fiedler vector, picking a splitting value, and partitioning the vertices in two, according to whether their corresponding values in the Fielder vector are smaller or larger than the splitting value. This process is repeated on the resulting halves, continuing recursively until a stopping point is reached. Much of the art resides in choosing the splitting value.

In our case, we will be making many fewer iterations than one would want to get a good estimate of the Fiedler vector. In addition, we will make many random starts, but will only split once, producing (perhaps) many clusters, rather than just two. It turns out, in fact, that we don't want to take so many steps that *M* is reduced merely to the effect produced by the Fiedler vector, since that is not good for making many simultaneous splits.

The existence of the mean drift is immaterial, since we only concern ourselves with the length of edges, which are *differences* between positions.

Appendix E: A deterministic algorithm? No.

Since $\mathrm{E}\{\delta_{ij}^{(s)}\} = \sqrt{\frac{2}{\pi}}\,\left\|\left(M^{s-1}\right)^T(\mathbf{r}_i - \mathbf{r}_j)\right\|$, one could make a deterministic algorithm, rather than a stochastic one, merely by computing $\left(M^{s-1}\right)^T\mathbf{r}_i$ for each vertex *i*, then computing the differences $\left(M^{s-1}\right)^T\mathbf{r}_i - \left(M^{s-1}\right)^T\mathbf{r}_j$ for each edge (*i*,*j*). Unfortunately, just the computation for the vertices would take $|V||E|$ work and require $|V|^2$ storage. One could avoid the storage issues by directly computing $\left\|\left(M^{s-1}\right)^T(\mathbf{r}_i - \mathbf{r}_j)\right\|$ for each edge (*i*,*j*), at the expense of $|E|^2$ work.

Another approach, for the spring model, would be to observe that for large s, $\left\|\left(M^{s-1}\right)^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right\|$ begins to measure $\left|\mathbf{F}^T\left(\mathbf{r}_i - \mathbf{r}_j\right)\right|$. It is tempting, then, to compute **F**, then dot it with the row differences to compute the edge lengths. This would bring the computation work back down to $|V| + |E|$. It is hard to imagine that a single vector (Fiedler or not) is insufficiently rich to represent *M* in establishing all of the desired splits. Indeed, experiments confirmed this.

References

Alves 2007: N. A. Alves, “Unveiling community structures in weighted networks,” *Physical Review E*, **76**, 2007, 036101.

Auber *et al.* 2004: D. Auber, M. Delest, and Y. Chiricota, “Strahler based graph clustering using convolution, *Proc. Of the Eighth International Conference on Information Visualization (IV ’04)*, pp. 44-51.

Anderson and Morley 1985: W.N. Anderson and T.D. Morley, “Eigenvalues of the Laplacian of a Graph,” *Lin. Multilin. Algebra*, **18**, 1985, 141-145.

Angelini *et al.* 2007: L. Angelini, S. Boccaletti, D. Marinazzo, M. Pellicoro, and S. Stramaglia, “Identification of network modules by optimization of ratio association,” *Chaos*, **17**, 2007, 023114.

Bollobas 1998: B. Bollobas, *Modern Graph Theory*, Springer, 1998.

Brandes *et al.* 2003: U. Brandes, M. Gaertler, and D. Wagner, “Experiments on Graph Clustering Algorithms,” *Algorithms – ESA 2003, Lecture Notes in Computer Science*, **2832**, 2003, pp. 568-579.

Capocci *et al.* 2004: A. Capocci, V. D. P. Servedio, G. Caldarelli, and F. Calaiori, “Communities Detection in Large Networks,” *Algorithms and Models for the Web-Graph, Proceedings of the Third International Workshop, WAW 2004*, LNCS 3243, pp. 181-187.

Clauset, Newman, and Moore 2004: Clauset, A, Newman, M. E. J., and Moore, C, “Finding community structure in very large networks,” arXiv:cond-mat/0408187v2 30 Aug 2004, and *Phys Rev E*, **70** (2004), 066111.

Cohen 2005: Cohen, J. D., “Trusses: cohesive subgraphs for social network analysis,” CiSE web site, XXX.

Danon *et al.* 2005: L. Danon, A. Diaz-Guilera, J. Duch, and A. Arenas, “Comparing community structure identification,” *Journal of Statistical Mechanics: Theory and Experiment*, 2005, P09008.

Danon *et al*. 2006: L. Danon, A. Diaz-Guilera, and A. Arenas, “The effect of size heterogeneity on community identification in complex networks,” *Journal of Statistical Mechanics: Theory and Experiment*, 2006, P11010.

De Wit 1991: D. De Wit, “Partitioning Sparse Graphs using the Second Eigenvector of their Graph Laplacian,” xrXiv:math.NA/0003036, 6 Mar 2000.

Dhillon *et al*. 2005: I. Dhillon, Y. Guan, and B. Kulis, “A Fast Kernel-based Multilevel Algorithm for Graph Clustering, *KDD ’05*, ACM 1-59593-135-X/05/0008, 2005.

Donetti and Munoz 2004: L. Donetti and M. A. Munoz, “Detecting network communities: a new systematic and efficient algorithm,” *Journal of Statistical Mechanics: Theory and Experiment*, 2004, P10012.

Fiedler 1973: M. Fiedler, “Algebraic Connectivity of Graphs,” *Czech. Math. J.*, 23 (98) 1973, 298-305.

Fiedler 1975: M. Fiedler, “A Property of Eigenvectors of Nonnegative Symmetric Matrices and its Application to Graph Theory,” *Czech. Math. J.*, 25 (100) 1975, 619-633.

Flake *et al*. 2000: G. W. Flake, S. Lawrence, and C. L. Giles, “Efficient Identification of Web Communities,” *Proc. KDD*, 2000, pp. 150-160.

Fortunato and Barthelemy 2007: S. Fortunato and M. Barthelemy, “Resolution limit in community detection,” *Proc. National Acad. Sci, USA*, **104**, 36, 2007, preprint arXiv:physics/0607100.

Girvan and Newman 2001: Girvan, M. and Newman, M. E. J., “Community structure in social and biological networks,” *Proc. National Acad. Sci, USA*, 99, 2003, pp. 7821-7826.

Harel and Koren 2001: D. Harel and Y. Koren, “Clustering Spatial Data Using Random Walks,” *KDD ’01*, pp. 281-286, 2001.

Karypis and Kumar 1996: G. Karypis and V. Kumar, “Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs,” *Proc. 1996*

ACM/IEEE conf. on Supercomputing, article number 35, 2996.

Kernighan and Lin 1970: B. W. Kernighan and S. Lin, ‘An Efficient Heuristic Procedure for Partitioning Graphs,” *Bell System Technical Journal*, **49** (2), Feb 1970, pp 291-307.

Newman 2003: Newman, M. E. J., “Fast algorithm for detecting community structure in networks,” arXiv:cont-mat/0309508v1 22 Sep 2003, and *Phys Rev E*, **69** (2004), 066133.

Newman 2006: Newman, M. E. J., “Finding community structure in networks using the eigenvectors of matrices,” *Physical Review E*, **74**, 2006, 036104.

Pothen *et al.* 1990: A. Pothen, H. D. Simon, and K.-P. Liou, “Partitioning sparse matrices with eigenvectors of graphs,” *SIAM Journal on Matrix Anal. Appl.*, **11** (3), July 1990, pp. 430-452.

Radicchi *et al.* 2004: F. Radicchi, C. Castellano, F. Cecconi, V. Loreta, and D. Parisi, “Defining and identifying communities in networks,” *Proc. National Acad. Sci.,* **101** (9), 2004, pp. 2658-2663 or arXiv/cond-mat/0309488.

Raghavan *et al.* 2007: U. N. Raghavan, R. Albert, and S. Kumara, “Near linear time algorithm to detect community structures in large-scale networks,” *Physical Review E*, **76**, 2007, 036106.

Rattigan, Maier, and Jensen 2007: M. J. Rattigan, M. Maier, and D. Jensen, “Graph Clustering with Network Structure Indices,” to appear in *Proc. 24th Intl. Conf. on Machine Learning, Corvallis*, OR, 2007.

Rosvall and Bergstrom 2007: M. Rosvall and C. T. Bergstrom, “An information-theoretic framework for resolving community structure in complex networks, *Proceedings of the National Academy of Sciences*, **104** (18), May 1, 2007, pp. 7327-7331.

Wasserman and Faust 1994:	Wasserman, S. and Faust., K, *Social Network Analysis: Methods and Applications*, Cambridge University Press, 1994.
Wilkinson and Huberman 2004:	D. M. Wilkinson and B. A. Huberman, “A method for finding communities of related genes,” *Proceedings of the National Academy of Sciences*, **101**, suppl. 1, April 6, 2004, pp. 5241-5248.
Wu *et al.* 2004:	A. Y. Wu, M. Garland, and J. Han, “Mining Scale-free Networks using Geodesic Clustering,” *KDD ’04*, pp. 719-724, 2004.
Wu and Huberman 2004:	F. Wu and B. A. Huberman, “Finding communities in linear time: a physics approach,” *European Physics Journal B*, **38**, 2004, pp 331-338.
Zhang *et al.* 2007a:	S. Zhang, R.-S. Wang, and X.-S. Zhang, “Uncovering fuzzy community structure in complex networks,” *Physical Review E*, **76**, 2007, 046103.
Zhang *et al.* 2007b:	S. Zhang, X.-M. Ning, and X.-S. Zhang, “Graph kernels, hierarchical clustering, and network community structure: experiments and comparative analysis,” *European Physical Journal B*, **57**, 67-74, 2007.