Networks
Mark Newman

Print publication date: 2010
Print ISBN-13: 9780199206650
Published to Oxford Scholarship Online: Sep-10
DOI: 10.1093/acprof:oso/9780199206650.001.0001

Matrix algorithms and graph partitioning

Abstract and Keywords

The preceding chapter discussed a variety of computer algorithms for calculating quantities of interest on networks, including degrees, centralities, shortest paths, and connectivity. This chapter continues the study of network algorithms with algorithms based on matrix calculations and methods of linear algebra applied to the adjacency matrix or other network matrices such as the graph Laplacian. It begins with a simple example — the calculation of eigenvector centrality — which involves finding the leading eigenvector of the adjacency matrix, and then moves on to some more advanced examples, including Fiedler's spectral partitioning method and algorithms for network community detection. Exercises are provided at the end of the chapter.

Keywords: network algorithms, network calculations, matrix calculations, linear algebra

M. E. J. Newman

DOI:10.1093/acprof:oso/9780199206650.003.0011

Abstract and Keywords

The preceding chapter discussed a variety of computer algorithms for calculating quantities of interest on networks, including degrees, centralities, shortest paths, and connectivity. This chapter continues the study of network algorithms with algorithms based on matrix calculations and methods of linear algebra applied to the adjacency matrix or other network matrices such as the graph Laplacian. It begins with a simple example — the calculation of eigenvector centrality — which involves finding the leading eigenvector of the adjacency matrix, and then moves on to some more advanced examples, including Fiedler's spectral partitioning method and algorithms for network community detection. Exercises are provided at the end of the chapter.

Keywords: network algorithms, network calculations, matrix calculations, linear algebra

In the preceding chapter we discussed a variety of computer algorithms for calculating quantities of interest on networks, including degrees, centralities, shortest paths, and connectivity. We continue our study of network algorithms in this chapter with algorithms based on matrix calculations and methods of linear algebra applied to the adjacency matrix or other network matrices such as the graph Laplacian. We begin with a simple example, the calculation of eigenvector centrality, which involves finding the leading eigenvector of the adjacency matrix, and then we move on to some more advanced examples, including Fiedler's spectral partitioning method and algorithms for network community detection.

11.1 Leading Eigenvectors and Eigenvector Centrality

As discussed in Section 7.2 , the eigenvector centrality of a vertex *i* in a network is defined to be the *i*th element of the leading eigenvector of the adjacency matrix, meaning the eigenvector corresponding to the largest (most positive) eigenvalue. Eigenvector centrality is an example of a quantity that can be calculated by a computer in a number of different ways, but not all of them are equally efficient. One way to calculate it would be to use a standard linear algebra method to calculate the complete set of eigenvectors of the adjacency matrix, and then discard all of them except the one corresponding to the largest eigenvalue. This, however, would be a wasteful approach, since it involves calculating (p. 346) a lot of things that we don't need. A simpler and faster method for calculating the eigenvector centrality is the *power method*.

If we start with essentially any initial vector x(0) and multiply it repeatedly by the adjacency matrix A, we get (11.1)

$$\mathbf{x}(t) = \mathbf{A}^t \mathbf{x}(0),$$

and, as shown in Section 7.2 , x(*t*) will converge [1] to the required leading eigenvector of A as *t* # infra. This is the power method, and, simple though it is, there is no faster method known for calculating the eigenvector centrality (or the leading eigenvector of any matrix). There are a few caveats, however:

1. The method will not work if the initial vector x(0) happens to be orthogonal to the leading eigenvector. One simple way to avoid this problem is to choose the initial vector to have all elements positive. This works because all elements of the leading eigenvector of a real matrix with nonnegative elements have the same sign,2 which means that any vector or (p. 347) thogonal to the leading eigenvector must contain both positive and negative

elements. Hence, if we choose all elements of our initial vector to be positive, we are guaranteed that the vector cannot be orthogonal to the leading eigenvector.

2. The elements of the vector have a tendency to grow on each iteration— they get multiplied by approximately a factor of the leading eigenvalue (p. 348) each time, which is usually greater than 1. Computers however cannot handle arbitrarily large numbers. Eventually the variables storing the elements of the vector will overflow their allowed range. To obviate this problem, we must periodically renormalize the vector by dividing all the elements by the same value, which we are allowed to do since an eigenvector divided throughout by a constant is still an eigenvector. Any suitable divisor will do, but we might, for instance, divide by the magnitude of the vector, thereby normalizing it so that its new magnitude is 1.

3. How long do we need to go on multiplying by the adjacency matrix before the result converges to the leading eigenvalue? This will depend on how accurate an answer we require, but one simple way to gauge convergence is to perform the calculation in parallel for two different initial vectors and watch to see when they reach the same value, within some prescribed tolerance. This scheme works best if, for the particular initial vectors chosen, at least some elements of the vector converge to the final answer from opposite directions for the two vectors, one from above and one from below. (We must make the comparisons immediately after the renormalization of the vector described in (2) above—if we compare unnormalized vectors, then most likely all elements will increase on every iteration and no convergence will be visible.) If we can find some elements that do this (and we usually can), then it is a fairly safe bet that the difference between the two values for such an element is greater than the difference of either from the true value of the same element in the leading eigenvector.

The power method can also be used to calculate the leading eigen*value* κ_1 of the adjacency matrix. Once the algorithm has converged to the leading eigenvector, one more multiplication by the adjacency matrix will multiply that vector by exactly a factor of κ_1. Thus, we can take the ratio of the values of any element of the vector at two successive iterations of the algorithm after convergence and that ratio should equal κ_1. Or we could take the average of the ratios for several different elements to reduce numerical errors. (We should however avoid elements whose values are very small, since a small error in such an element could lead to a large fractional error in the ratio; our accuracy will be better if we take the average of some of the larger elements.)

11.1.1 Computational Complexity

How long does the power method take to run? The answer comes in two parts. First, we need to know how long each multiplication by the adjacency matrix takes, and second we need to know how many multiplications are needed to (p. 349) get a required degree of accuracy in our answer.

If our network is stored in adjacency matrix form, then multiplying that matrix into a given vector is straightforward. Exactly n^2 multiplications are needed for one matrix multiplication—one for each element of the adjacency matrix. We can do better, however, if our network is in adjacency list form. Elements of the adjacency matrix that are zero contribute nothing to the matrix multiplication and so can be neglected. The adjacency list allows us to skip the zero terms automatically, since it stores only the non-zero ones anyway.

In an ordinary unweighted network each non-zero element of the adjacency matrix is equal to 1. Let $\{u_j\}$, $j = 1 \ldots k_i$ be the set of neighbors of vertex i (where k_i is the degree of i). Then the ith element of $\mathbf{Ax}$, which we denote $[\mathbf{Ax}]_i$, is given by

$$[Ax]_i = \Sigma_{j=1}^{ki} x_{u_j}$$

. The evaluation of this sum involves only k_i operations, so one element of the matrix multiplication can be completed in time proportional to ki and all elements can be completed in time proportional to $\sigma_i \; k_i = 2m$, where m is the total number of edges in the network, or in other words in $O(m)$ time.

And how many such multiplications must we perform? Equation (7.4) tells us that after t iterations our vector is equal to (11.2)

$$\mathbf{x}(t) = \kappa_1^t \sum_{i=1}^{n} c_i \left[\frac{\kappa_i}{\kappa_1}\right]^t \mathbf{v}_i,$$

where $\mathbf{v}_i$ is the normalized ith eigenvector, κ_i is the corresponding eigenvalue, and the c_i are constants whose values depend on the choice of initial vector. Rearranging slightly, we can write this as (11.3)

$$\frac{\mathbf{x}(t)}{c_1\kappa_1^t} = \mathbf{v}_1 + \frac{c_2}{c_1}\left(\frac{\kappa_2}{\kappa_1}\right)^t \mathbf{v}_2 + \ldots,$$

which gives us our estimate of the leading eigenvector $\mathbf{v}_1$ plus the dominant contribution to the error. Neglecting the smaller terms, the root-mean-square error on the eigenvector is then (11.4)

$$\sqrt{\left|\frac{\mathbf{x}(t)}{c_1\kappa_1^t}-\mathbf{v}_1\right|^2}=\frac{c_2}{c_1}\left(\frac{\kappa_2}{\kappa_1}\right)^t,$$

and if we want this error to be at most # then we require (11.5)

$$t \geq \frac{\ln(1/\epsilon)+\ln(c_1/c_2)}{\ln(\kappa_1/\kappa_2)}.$$

Neither # nor the constants c_1 and $c2$ depend on the network size. All the variation in the run time comes from the eigenvalues ?1 and $?_2$. The eigenvalues (p. 350) range in value from a maximum of κ_1 to a minimum of $\kappa_n \geq -|\kappa_1|$ and hence have a mean spacing of at most $2\kappa_1/(n - 1)$. Thus an order-of-magnitude estimate for the second eigenvalue is κ_2 # $\kappa_1 - a\kappa_1/n$, where a is a constant of order unity, and hence (11.6)

$$\ln\frac{\kappa_1}{\kappa_2} \simeq -\ln\left(1-\frac{a}{n}\right)=\frac{a}{n}+\mathrm{O}(n^{-2}).$$

Combining Eqs. (11.5) and (11.6) , we find that the number of steps required for convergence of the power method is $t = O(n)$ to leading order. 3

Overall therefore, the complete calculation of the eigenvector centralities of all n vertices of the network takes $O(n)$ multiplications which take $O(m)$ time each, or $O(mn)$ time overall, for a network stored in adjacency list format. If our network is sparse with m # n, a running time of $O(mn)$ is equivalent to $O(n^2)$. On the other hand, if the network is dense, with m # n_2, then $O(mn)$ is equivalent to $O(n3)$.

Conversely, if our network is stored in adjacency matrix format the multiplications take $O(n_2)$ time, as noted above, so the complete calculation takes $O(n3)$, regardless of whether the network is sparse or dense. Thus for the common case of a sparse matrix the adjacency list is the representation of choice for this calculation.

11.1.2 Calculating Other Eigenvalues and Eigenvectors

The power method of the previous section calculates the largest eigenvalue of a matrix and the corresponding eigenvector. This is probably the most common type of eigenvector calculation encountered in the study of networks, but there are cases where we wish to know other eigenvectors or eigenvalues as well. One example is the calculation of the so-called algebraic connectivity, which is the second smallest (or second most negative) eigenvalue of the graph Laplacian. As we saw in Section 6.13.3

, the algebraic connectivity is non-zero if and only if a network is connected (i.e., has just a single component). The algebraic connectivity also appears in Section 11.5 as a measure of how easily a network can be bisected into two sets of vertices such that only a small number of edges run between the sets. Moreover, as we will see the elements of the corresponding eigenvector of the Laplacian tell us exactly how that bisection (p. 351) should be performed. Thus it will be useful to us to have a method for calculating eigenvalues beyond the largest one and their accompanying eigenvectors.

There are a number of techniques that can be used to find non-leading eigenvalues and eigenvectors of matrices. For instance, we can calculate the eigenvector corresponding to the most negative eigenvalue by shifting all the eigenvalues by a constant amount so that the most negative one becomes the eigenvalue of largest magnitude. The eigenvalues of the graph Laplacian L, for instance, are all non-negative. If we number them in ascending order as in Section 6.13.2 , so that λ_1 ≤hellip; λ_2 … ≤ λ_n , with v_1, v_2, …, v_n being the corresponding eigenvectors, then (11.7)

$$(\lambda_n \mathbf{I} - \mathbf{L})\mathbf{v}_i = (\lambda_n - \lambda_i)\mathbf{v}_i ,$$

and hence v_i is an eigenvector of λ_n I?L with eigenvalue λ_n -λi. These eigenvalues are still all non-negative, but their order is reversed from those of the original Laplacian, so that the former smallest has become the new largest. Now we can calculate the eigenvector corresponding to the smallest eigenvalue of the Laplacian by finding the leading eigenvector of λ_n I - L using the technique described in Section 11.1 . We can also find the eigenvalue λ_1 by taking the measured value of λ_n - λ_1, subtracting λ_n , and reversing the sign. (Performing these calculations does require that we know the value of λ_n , so the complete calculation would be a two-stage process consisting of first finding the largest eigenvalue of L, then using that to find the smallest. 4)

In this particular case, it would not in fact be very useful to calculate the smallest eigenvalue or its associated eigenvector since, as we saw in Section 6.13.2 , the smallest eigenvalue of the Laplacian is always zero and the eigenvector is (1, 1, 1, …). However, if we can find the second-largest eigenvalue of a matrix we can use the same subtraction method also to find the second-smallest. And the second-smallest eigenvalue of the Laplacian is, as we have said, definitely of interest.

We can find the second-largest eigenvalue (and the corresponding eigenvector) using the following trick. Let v_1 be the normalized eigenvector corresponding to the largest eigenvalue of a matrix A, as found, for instance, by the power method of Section 11.1 . Then we choose any starting vector x as before (p. 352) and define (11.8)

$$\mathbf{y} = \mathbf{x} - (\mathbf{v}_1^T \mathbf{x})\mathbf{v}_1.$$

This vector has the property that (11.9)

$$\mathbf{v}_i^T\mathbf{y} = \mathbf{v}_i^T\mathbf{x} - (\mathbf{v}_1^T\mathbf{x})(\mathbf{v}_i^T\mathbf{v}_1) = \mathbf{v}_i^T\mathbf{x} - \mathbf{v}_1^T\mathbf{x}\,\delta_{i1}$$
$$= \begin{cases} 0 & \text{if } i = 1, \\ \mathbf{v}_i^T\mathbf{x} & \text{otherwise,} \end{cases}$$

where v $_i$ is again the *i*th eigenvector of A and δ $_{ij}$ is the Kronecker delta. In other words it is equal to x along the direction of every eigenvector of A except the leading eigenvector, in whose direction it has no component at all. This means that the expansion of y in terms of the eigenvectors of A, which is given by

$y = \Sigma_{i=1}^{n} c_i v_i$

with

$c_i = v_i^T y$

, has no term in v $_1$, since

$c_i = v_i^T y = 0$

Thus (11.10)

$$\mathbf{y} = \sum_{i=2}^{n} c_i \mathbf{v}_i,$$

with the sum starting at *i* = 2.

Now we use this vector y as the starting vector for repeated multiplication by A, as before. After multiplying y by A a total of *t* times, we have (11.11)

$$\mathbf{y}(t) = \mathbf{A}^t\mathbf{y}(0) = \kappa_2^t \sum_{i=2}^{n} c_i \left[\frac{\kappa_i}{\kappa_2}\right]^t \mathbf{v}_i.$$

The ratio κ $_i$ /κ$_2$ is less than 1 for all *i* > 2 (assuming only a single eigenvalue of value κ$_2$) and hence in the limit of large *t* all terms in the sum disappear except the first so that y(*t*) tends to a multiple of v $_2$ as *t* # #. Normalizing this vector, we then have our result for v $_2$.

This method has the same caveats as the original power method for the leading eigenvector, as well as one additional one: it is in practice possible for the vector y, Eq. (11.8) , to have a very small component in the direction of v $_1$. This can happen as a result of numerical error in the subtraction, or because our value for v $_1$ is not exactly correct. If y does have a component in the direction of v $_1$, then although it may start out small it will get magnified relative to the others when we multiply repeatedly by

A and eventually it may come to dominate $y(t)$, Eq. (11.11) , or at least to contribute a sufficiently large term as to make the calculation of v_2 inaccurate. To prevent this happening, we periodically perform a subtraction similar to that of Eq. (11.8) , removing any component in the direction of v_1 from $y(t)$, while leaving the components in all other directions untouched. (The subtraction process is sometimes referred to as *Gram–Schmidt orthogonalization*—a rather grand name for a simple (p. 353) procedure. The repeated application of the process to prevent the growth of unwanted terms is called *reorthogonalization*.)

We could in theory extend this method to find further eigenvectors and eigenvalues of our matrix, but in practice the approach does not work well beyond the first couple of eigenvectors because of cumulative numerical errors. Moreover it is also slow because for each additional eigenvector we calculate we must carry out the entire repeated multiplication process again. In practice, therefore, if we wish to calculate anything beyond the first eigenvector or two, other methods are used.

11.1.3 Efficient Algorithms for Computing all Eigenvalues and Eigenvectors of Matrices

If we wish to calculate all or many of the eigenvalues or eigenvectors of a matrix A then specialized techniques are needed. The most widely used such techniques involve finding an orthogonal matrix Q such that the similarity transform $T = Q^T A Q$ gives either a tridiagonal matrix (if A is symmetric) or a Hessenberg matrix (if A is asymmetric). If we can find such a transformation and if v_i is an eigenvector of A with eigenvalue κ_i, then, bearing in mind that for an orthogonal matrix $Q^{-1} = Q^T$, we have
(11.12)

$$\kappa_i \mathbf{Q}^T\mathbf{v}_i = \mathbf{Q}^T\mathbf{A}\mathbf{v}_i = \mathbf{T}\mathbf{Q}^T\mathbf{v}_i.$$

In other words, the vector $w_i = Q^T v_i$ is an eigenvector of T with eigenvalue κ_i. Thus if we can find the eigenvalues of T and the corresponding eigenvectors, we automatically have the eigenvalues of A as well, and the eigenvectors of A are simply $v_i = Q w_i$. Luckily there exist efficient numerical methods for finding the eigenvalues and eigenvectors of tridiagonal and Hessenberg matrices, such as the *QL algorithm* [273]. The QL algorithm takes time $O(n)$ to reach an answer for an $n \times n$ tridiagonal matrix and $O(n^2)$ for a Hessenberg one.

The matrix Q can be found in various ways. For a general symmetric matrix the *Householder algorithm* [273] can find Q in time $O(n^3)$. More often, however, we are concerned with sparse matrices, in which case there are faster methods. For a symmetric matrix, the *Lanczos algorithm* [217] can find Q in time $O(mn)$, where m is the number of network edges in an adjacency matrix, or more generally the number of non-zero elements in the matrix. For sparse matrices with $m \# n$ this gives a running time of $O(n^2)$, considerably better than the Householder method. A similar method, the *Arnoldi algorithm* [217], can find Q for an asymmetric matrix.

Thus, combining the Lanczos and QL algorithms, we expect to be able to find all eigenvalues and eigenvectors of a sparse symmetric matrix in time (p. 354) $O(mn)$, which is as good as the worst-case run time of our direct multiplication method for finding just the leading eigenvector. (To be fair, the direct multiplication is much simpler, so its overall run time will typically be better than that of the combined Lanczos/QL algorithm, although the scaling with system size is the same.)

While there is certainly much to be gained by learning about the details of these algorithms, one rarely implements them in practice. Their implementation is tricky (particularly in the asymmetric case), and has besides already been done in a careful and professional fashion by many software developers. In practice, therefore, if one wishes to solve eigensystem problems for large networks, one typically turns to commercial or freely available implementations in professionally written software packages. Examples of such packages include Matlab, LAPACK, and Mathematica. We will not go into more detail here about the operation of these algorithms.

11.2 Dividing Networks Into Clusters

We now turn to the topics that will occupy us for much of the rest of the chapter, *graph partitioning* and *community detection*. [5] Both of these terms refer to the division of the vertices of a network into groups, clusters, or communities according to the pattern of edges in the network. Most commonly one divides the vertices so that the groups formed are tightly knit with many edges inside groups and only a few edges between groups.

Consider Fig. 11.1 , for instance, which shows patterns of collaborations between scientists in a university department. Each vertex in this network represents a scientist and links between vertices indicate pairs of scientists who have coauthored one or more papers together. As we can see from the figure, this network contains a number of densely connected clusters of vertices, corresponding to groups of scientists who have worked closely together. Readers familiar with the organization of university departments will not be surprised to learn that in general these clusters correspond, at least approximately, to formal research groups within the department.

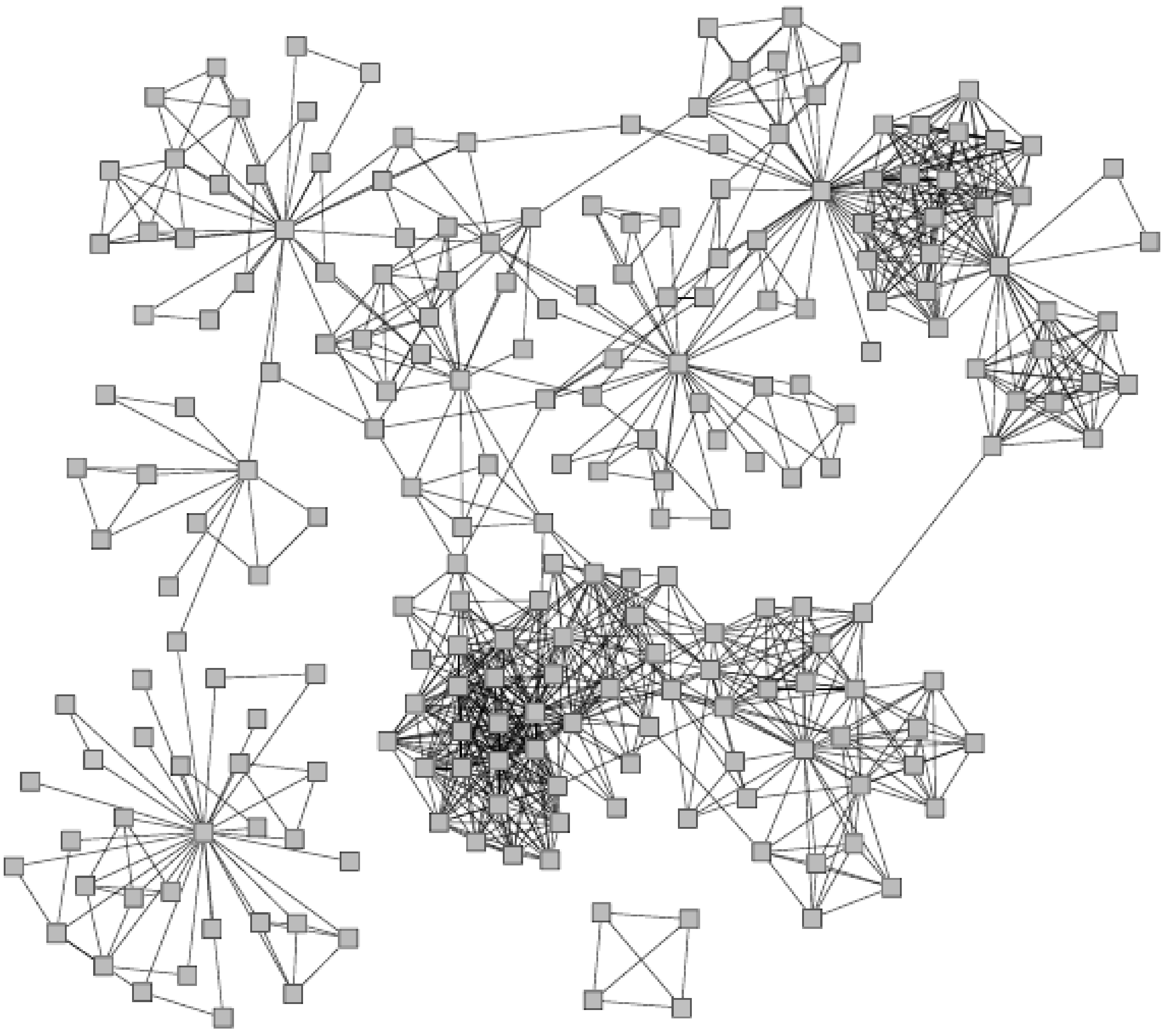
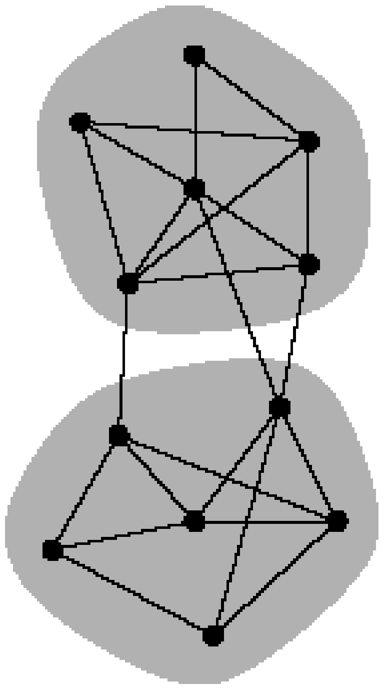


Figure 11.1: **Network of coauthorships in a university department.** The vertices in this network represent scientists in a university department, and edges links pairs of scientists who have coauthored scientific papers. The network has clear clusters or "community structure," presumably reflecting divisions of interests and research groups within the department.

But suppose one did not know how university departments operate and wished to study them. By constructing a network like that in Fig. 11.1 and then observing its clustered structure, one would be able to deduce the existence of groups within the larger department and by further investigation could probably (p. 355) quickly work out how the department was organized. Thus the ability to discover groups or clusters in a network can be a useful tool for revealing structure and organization within networks at a scale larger than that of a single vertex. In this particular case the network is small enough and sparse enough that the groups are easily visible by eye. Many of the networks that have engaged our interest in this book, however, are much larger or denser networks for which visual inspection is not a useful tool. Finding clusters in such networks is a task for computers and the algorithms that run on them. (p. 356)

11.2.1 Partitioning and Community Detection

There are a number of reasons why one might want to divide a network into groups or clusters, but they separate into two general classes that lead in turn to two corresponding types of computer algorithm. We will refer to these two types as *graph partitioning* and *community detection* algorithms. They are distinguished from one another by whether the number and size of the groups is fixed by the experimenter or whether it is unspecified.



Partition of a network into two groups of equal sizes.

Graph partitioning is a classic problem in computer science, studied since the 1960s. It is the problem of dividing the vertices of a network into a given number of non-overlapping groups of given sizes such that the number of edges between groups is minimized. The important point here is that the number and sizes of the groups are fixed. Sometimes the sizes are only fixed roughly—within a certain range, for instance

—but they are fixed nonetheless. For instance, a simple and prototypical example of a graph partitioning problem is the problem of dividing a network into two groups of equal size, such that the number of edges between them is minimized.

Graph partitioning problems arise in a variety of circumstances, particularly in computer science, but also in pure and applied mathematics, physics, and of course in the study of networks themselves. A typical example is the numerical solution of network processes on a parallel computer. In the last part of this book (Chapters 16 to 19) we will study processes that take place on networks, such as diffusion processes or the spread of diseases. These processes can be modeled mathematically by placing variables on the vertices of a network and evolving them according to equations that typically depend on the variables' current values and the values on neighboring vertices. The solution of such equations is often a laborious computational task, but it can be sped up by using a parallel computer, a computer with more than one processor or CPU. Many modern personal computers have two or more processors and large research organizations sometimes use parallel computers with very many processors. Solutions of network equations can be spread across several processors by assigning to each processor the task of solving the equations on a subset of the vertices. For instance, on a two-processor desktop computer we might give a half of the vertices to each processor.

The catch is that, unless the network consists of totally unconnected components, some vertices on one processor are always going to have neighbors that are on the other processor and hence the solution of their equations involves variables whose value is known only to the other processor. To complete the solution, therefore, those values have to be transmitted from the one processor to the other at regular intervals throughout the calculation and this is typically (p. 357) a slow process (or at least it's slow compared to the dazzling speed of most other computer operations). The time spent sending messages between processors can, in fact, be the primary factor limiting the speed of calculations on parallel computers, so it is important to minimize interprocessor communication as much as possible. One way that we do this is by minimizing the number of pairs of neighboring vertices assigned to different processors.

Thus we want to divide up the vertices of the network into different groups, one for each processor, such that the number of edges between groups is minimized. Most often we want to assign an equal or roughly equal number of vertices to each processor so as to balance the workload among them. This is precisely a graph partitioning problem of the type described above.

The other type of cluster finding problem in networks is the problem we call community detection. Community detection problems differ from graph partitioning in that the number and size of the groups into which the network is divided are not specified by the experimenter. Instead they are determined by the network itself: the goal of community detection is to find the natural fault lines along which a network separates. The sizes of the groups are not merely unspecified but might in principle vary widely from one group to another. A given network might divide into a few large groups, many small ones, or a mixture of all different sizes.

The most common use for community detection is as a tool for the analysis and understanding of network data. We saw in Fig. 11.1 an example of a network for which a knowledge of the group structure might help us understand the organization of the underlying system. Figure 7.10 on page 221 shows another example of clusters of vertices, in a network of friendships between US high-school students. In this case the network splits into two clear groups, which, as described in Section 7.13 , are primarily dictated by students' ethnicity, and this structure and others like it can give us clues about the nature of the social interactions within the community represented.

Community detection has uses in other types of networks as well. Clusters of nodes in a web graph for instance might indicate groups of related web pages. Clusters of nodes in a metabolic network might indicate functional units within the network.

Community detection is a less well-posed problem than graph partitioning. Loosely stated, it is the problem of finding the natural divisions of a network into groups of vertices such that there are many edges within groups and few edges between groups. What exactly we mean by "many" or "few," however, is debatable, and a wide variety of different definitions have been proposed, leading to a correspondingly wide variety of different algorithms for community detection. In this chapter we will focus mainly on the most widely used (p. 358) formulation of the problem, the formulation in terms of modularity optimization, but we will mention briefly a number of other approaches at the end of the chapter.

In summary, the fundamental difference between graph partitioning and community detection is that the number and size of the groups into which a network is divided is specified in graph partitioning but unspecified in community detection. However, there is also a difference between the goals of the two types of calculations. Graph partitioning is typically performed as a way of dividing up a network into smaller more manageable pieces, for example to perform numerical calculations. Community detection is more often used as a tool for understanding the structure of a network, for shedding light on largescale patterns of connection that may not be easily visible in the raw network topology.

Notice also that in graph partitioning calculations the goal is usually to find the best division of a network, subject to certain conditions, regardless of whether any good division exists. If the performance of a calculation on a parallel computer, for example, requires us to divide a network into pieces, then we had better divide it up. If there are no good divisions, then we must make do with the least bad one. With community detection, on the other hand, where the goal is normally to understand the structure of the network, there is no need to divide the network if no good division exists. Indeed if a network has no good divisions then that in itself may be a useful piece of information, and it would be perfectly reasonable for a community detection algorithm only to divide up networks when good divisions exist and to leave them undivided the rest of the time.

11.3 Graph Partitioning

In the next few sections we consider the graph partitioning problem and look at two well-known methods for graph partitioning. The first, the Kernighan-Lin algorithm, is not based on matrix methods (and therefore doesn't strictly belong in this chapter) but it provides a simple introduction to the partitioning problem and is worth spending a little time on. In Section 11.5 we look at a more sophisticated partitioning method based on the spectral properties of the graph Laplacian. This spectral partitioning method both is important in its own right and will also provide a basis for our discussion of community detection later in the chapter.

First, however, we address an important preliminary question: why does one need fancy partitioning algorithms at all? Partitioning is an easy problem to state, so is it not just as easy to solve? (p. 359)

11.3.1 Why Partitioning is Hard

The simplest graph partitioning problem is the division of a network into just two parts. Division into two parts is sometimes called *graph bisection*. Most of the algorithms we consider in this chapter are in fact algorithms for bisecting networks rather than for dividing them into arbitrary numbers of parts. This may at first appear to be a drawback, but in practice it is not, since if we can divide a network into two parts, then we can divide it into more than two by further dividing one or both of those parts. This repeated bisection is the commonest approach to the partitioning of networks into arbitrary numbers of parts.

Formally the graph bisection problem is the problem of dividing the vertices of a network into two non-overlapping groups of given sizes such that the number of edges running between vertices in different groups is minimized. The number of edges between groups is called the *cut size*. [6]

Simple though it is to describe, this problem is not easy to solve. One might imagine that one could bisect a network simply by looking through all possible divisions of the network into two parts of the required sizes and choosing the one with the smallest cut size. For all but the smallest of networks, however, this so-called *exhaustive search* turns out to be prohibitively costly in terms of computer time.

The number of ways of dividing a network of n vertices into two groups of n_1 and n_2 vertices respectively is $n!/(n_1!\, n_2!)$. Approximating the factorials using Stirling's formula

$$n! \simeq \sqrt{2\pi n}(n/e)^n$$

and making use of the fact that (11.13)

$$\frac{n!}{n_1!\,n_2!} \simeq \frac{\sqrt{2\pi n}(n/\mathrm{e})^n}{\sqrt{2\pi n_1}(n_1/\mathrm{e})^{n_1}\sqrt{2\pi n_2}(n_2/\mathrm{e})^{n_2}} = \frac{n^{n+1/2}}{n_1^{n_1+1/2} n_2^{n_2+1/2}}.$$

Thus, for instance, if we want to divide a network into two parts of equal size ½n the number of different ways to do it is roughly (11.14)

$$\frac{n^{n+1/2}}{(n/2)^{n+1}} = \frac{2^{n+1}}{\sqrt{n}}.$$

So the amount of time required to look through all of these divisions will go up roughly exponentially with the size of the network. Unfortunately, the exponential is a very rapidly growing function of its argument, which means the (p. 360) partitioning task quickly leaves the realm of the possible at quite moderate values of n. Values up to about $n = 30$ are feasible with current computers, but go much beyond that and the calculation becomes intractable.

One might wonder whether it is possible to find a way around this problem. After all, brute-force enumeration of all possible divisions of a network is not a very imaginative way to solve the partitioning problem. Perhaps one could find a way to limit one's search to only those divisions of the network that have a chance of being the best one? Unfortunately, there are some fundamental results in computer science that tell us that no such algorithm will ever be able to find the best division of the network in all cases. Either an algorithm can be clever and run quickly, but will fail to find the optimal answer in some (and perhaps most) cases, or it always finds the optimal answer but takes an impractical length of time to do it. These are the only options. 7

This is not to say, however, that clever algorithms for partitioning networks do not exist or that they don't give useful answers. Even algorithms that fail to find the very best division of a network may still find a pretty good one, and for many practical purposes pretty good is good enough. The goal of essentially all practical partitioning algorithms is just to find a "pretty good" division in this sense. Algorithms that find approximate, but acceptable, solutions to problems in this way are called *heuristic algorithms* or just *heuristics*. All the algorithms for graph partitioning discussed in this chapter are heuristic algorithms.

11.4 The Kernighan-Lin Algorithm

The *Kernighan-Lin algorithm*, proposed by Brian Kernighan 8 and Shen Lin in 1970 [171], is one of the simplest and best known heuristic algorithms for the graph bisection problem. The algorithm is illustrated in Fig. 11.2 .

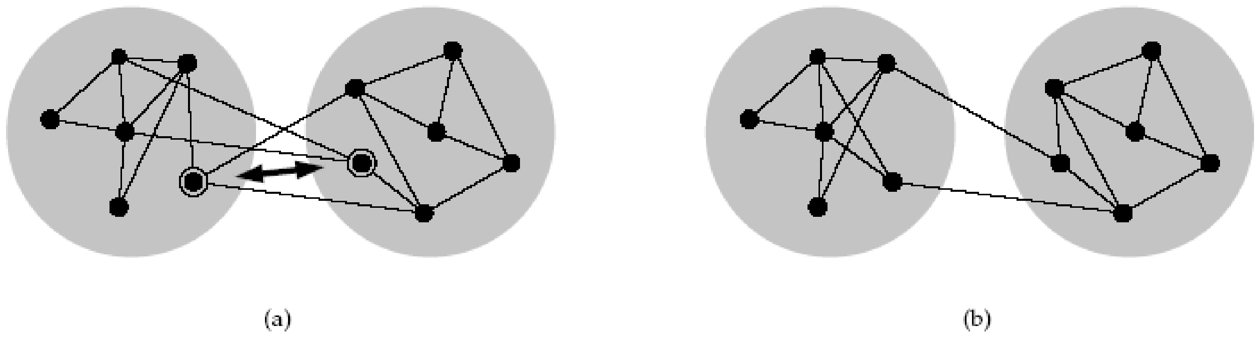


Figure 11.2: The Kernighan-Lin algorithm. (a) The Kernighan-Lin algorithm starts with any division of the vertices of a network into two groups (shaded) and then searches for pairs of vertices, such as the pair highlighted here, whose interchange would reduce the cut size between the groups. (b) The same network after interchange of the two vertices.

We start by dividing the vertices of our network into two groups of the required sizes in any way we like. For instance, we could divide the vertices randomly. Then, for each pair (i, j) of vertices such that i lies in one of the groups (p. 361) and j in the other, we calculate how much the cut size between the groups would change if we were to interchange i and j, so that each was placed in the other group. Among all pairs (i, j) we find the pair that reduces the cut size by the largest amount or, if no pair reduces it, we find the pair that increases it by the smallest amount. Then we swap that pair of vertices. Clearly this process preserves the sizes of the two groups of vertices, since one vertex leaves each group and another joins. Thus the algorithm respects the requirement that the groups take specified sizes.

The process is then repeated, but with the important restriction that each vertex in the network can only be moved once. Once a vertex has been swapped with another it is not swapped again (at least not in the current round of the algorithm–see below). Thus, on the second step of the algorithm we consider all pairs of vertices excluding the two vertices swapped on the first step.

And so the algorithm proceeds, swapping on each step that pair that most decreases, or least increases, the number of edges between our two groups, until eventually there are no pairs left to be swapped, at which point we stop. (If the sizes of the groups are unequal then there will be vertices in the larger group that never get swapped, equal in number to the difference between the sizes of the groups.)

When all swaps have been completed, we go back through every state that the network passed through during the swapping procedure and choose among them the state in which the cut size takes its smallest value. 9

(p. 362) Finally, this entire process is performed repeatedly, starting each time with the best division of the network found on the last time around and continuing until no improvement in the cut size occurs. The division with the best cut size on the last round is the final division returned by the algorithm.

Once we can divide a network into two pieces of given size then, as we have said, we can divide into more than two simply by repeating the process. For instance, if we want to divide a network into three pieces of equal size, we would first divide into two pieces, one twice the size of the other, and then further divide the larger one into two equally sized halves. (Note, however, that even if the algorithm were able to find the optimal division of the network in each of these two steps, there would be no guarantee that we would end up with the optimal division of the network into three equal parts. Nonetheless, we do typically find a reasonably good division, which, as we have said, is often good enough. This point is discussed further in Section 11.9 .)

Note that if we choose the initial assignment of vertices to groups randomly, then the Kernighan-Lin algorithm may not give the same answer if it is run twice on the same network. Two different random starting states could (though needn't necessarily) result in different divisions of the network. For this reason, people sometimes run the algorithm more than once to see if the results vary. If they do vary then among the divisions of the network returned on the different runs it makes sense to take the one with the smallest cut size.

As an example of the use of the Kernighan-Lin algorithm, consider Fig. 11.3 , which shows an application of the algorithm to a mesh, a two-dimensional network of the type often used in parallel finite-element computations. Suppose e want to divide this network into two parts of equal size. Looking at the complete network in Fig. 11.3a there is no obvious division—there is no easy cut or bottleneck where the network separates naturally—but we must do the best we can. Figure 11.3b shows the best division found by the Kernighan-Lin algorithm, which involves cutting 40 edges in the network. Though it might not be the best possible division of the network, this is certainly good enough for many practical purposes.

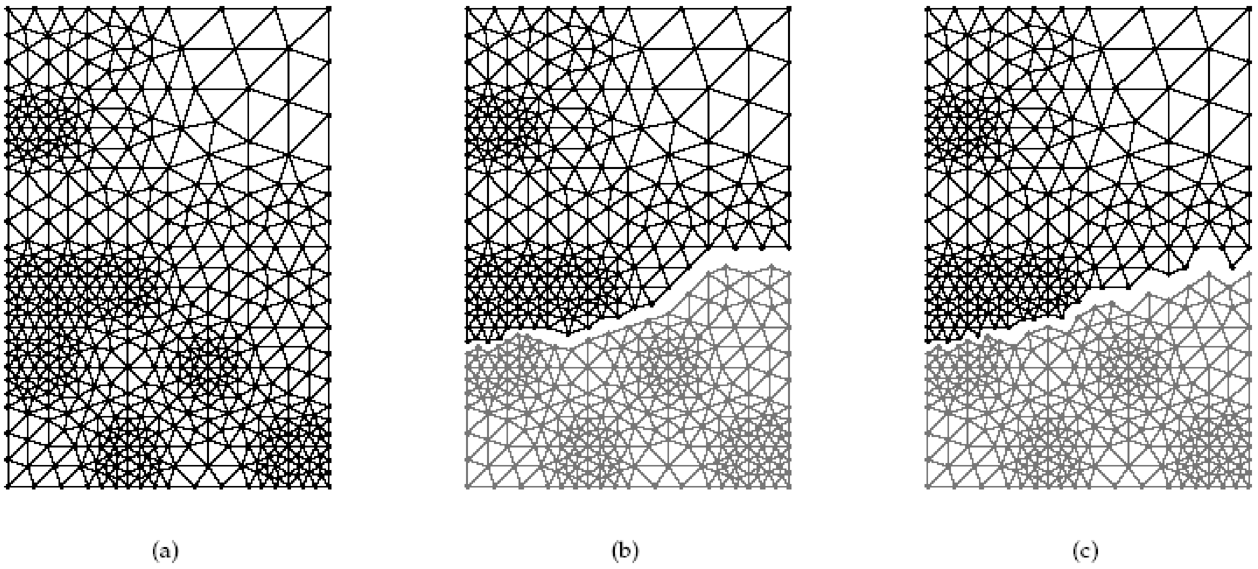


Figure 11.3: Graph partitioning applied to a small mesh network. (a) A mesh network of 547 vertices of the kind commonly used in finite element analysis. (b) The edges removed indicate the best division of the network into parts of 273 and 274 vertices found by the Kernighan-Lin algorithm. (c) The best division found by spectral partitioning. The network is from Bern *et al.* [35].

The primary disadvantage of the Kernighan-Lin algorithm is that it is quite slow. The number of swaps performed during one round of the algorithm is (p. 363) equal to the smaller of the sizes of the two groups, which lies between zero and ½*n* in a network of *n* vertices. Thus there are $O(n)$ swaps in the worst case. For each swap we have to examine all pairs of vertices in different groups, of which there are, in the worst case, $\frac{1}{2}n \times \frac{1}{2}n = \frac{1}{4}n^2 = O(n^2)$. And for each of these we need to determine the change in the cut size if the pair is swapped.

When a vertex *i* moves from one group to the other any edges connecting it to vertices in its current group become edges between groups after the swap. Let us suppose that there are k_i^{same} such edges. Similarly, any edges that *i* has to vertices in the other group, of which there are say k_i^{other}, become withingroup edges after the swap, but with one exception. If *i* is being swapped with vertex *j* and there is an edge between *i* and *j*, then that edge lies between groups before the swap and still lies between groups after the swap. Thus the change in the cut size due to the movement of *i* is

$$k_i^{\text{other}} - k_i^{\text{same}} - A_{ij}$$

. A similar expression applies for vertex *j* also and the total change in cut size as a result of the swap is (11.15)

$$\Delta = k_i^{\text{other}} - k_i^{\text{same}} + k_j^{\text{other}} - k_j^{\text{same}} - 2A_{ij}.$$

For a network stored in adjacency list form, the evaluation of this expression (p. 364) involves running through all the neighbors of i and j in turn, and hence takes time of order the average degree in the network, or $O(m/n)$, where m is, as usual, the total number of edges in the network.

Thus the total time for one round of the algorithm is $O(n \times n^2 \times m/n) = O(mn^2)$, which is $O(n^3)$ on a sparse network in which $m \# n$ or $O(n^4)$ on a dense network. This in itself would already be quite bad, but we are not yet done. This time must be multiplied by the number of rounds the algorithm performs before the cut size stops decreasing. It is not well understood how the number of rounds required varies with network size. In typical applications the number is small, maybe five or ten for networks of up to a few thousand vertices, and larger networks are currently not possible because of the demands of the algorithm, so in practice the number of rounds is always small. Still, it seems quite unlikely that the number of rounds would actually increase as network size grows, and even if it remains constant the time complexity of the algorithm will still be $O(mn^2)$, which is relatively slow.

We can improve the running time of the algorithm a little by a couple of tricks. If we initially calculate and store the number of neighbors,
k_i^{same}
and
k_i^{other}
, that each vertex has within and between groups and update it every time a vertex is moved, then we save ourselves the time taken to recalculate these quantities on each step of the algorithm. And if we store our network in adjacency matrix form then we can tell whether two vertices are connected (and hence evaluate A_{ij}) in time $O(1)$. Together these two changes allow us to calculate ? above in time $O(1)$ and improve the overall running time to $O(n^3)$. For a sparse graph this is the same as $O(mn^2)$, but for a dense one it gives us an extra factor of n.

Overall, however, the algorithm is quite slow. Even with $O(n^3)$ performance the algorithm is suitable only for networks up to a few hundreds or thousands of vertices, but not more.

11.5 Spectral Partitioning

So are there faster methods for partitioning networks? There are indeed, although they are typically more complex than the simple Kernighan-Lin algorithm, and may be correspondingly more laborious to implement. In this section we discuss one of the most widely used methods, the *spectral partitioning* method of Fiedler [118, 271], which makes use of the matrix properties of the graph Laplacian. We describe the spectral partitioning method as applied to the graph bisection problem, the problem of dividing a graph into two parts of specified sizes. As discussed in the previous section, division into more than two groups is typically achieved by repeated bisection, dividing and subdividing (p. 365) the network to give groups of the desired number and size.

Consider a network of n vertices and m edges and a division of that network into two groups, which we will call group 1 and group 2. We can write the cut size for the division, i.e., the number of edges running between the two groups, as (11.16)

$$R = \frac{1}{2} \sum_{\substack{i,j \text{ in} \\ \text{different} \\ \text{groups}}} A_{ij},$$

where the factor of ½ compensates for our counting each edge twice in the sum.

Let us define a set of quantities s_i , one for each vertex i, which represent the division of the network thus: (11.17)

$$s_i = \begin{cases} +1 & \text{if vertex } i \text{ belongs to group 1,} \\ -1 & \text{if vertex } i \text{ belongs to group 2.} \end{cases}$$

Then (11.18)

$$\tfrac{1}{2}(1 - s_i s_j) = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are in different groups,} \\ 0 & \text{if } i \text{ and } j \text{ are in the same group,} \end{cases}$$

which allows us to rewrite Eq. (11.16) as (11.19)

$$R = \frac{1}{4} \sum_{ij} A_{ij}(1 - s_i s_j),$$

with the sum now over all values of i and j. The first term in the sum is (11.20)

$$\sum_{ij} A_{ij} = \sum_i k_i = \sum_i k_i s_i^2 = \sum_{ij} k_i \delta_{ij} s_i s_j,$$

where k_i is the degree of vertex i as usual, δ_{ij} is the Kronecker delta, and we have made use of the fact that $\Sigma_j A_{ij} = k_i$ (see Eq. (6.19)) and

$s_i^2 = 1$

(since $s_i = \pm 1$). Substituting back into Eq. (11.19) we then find that (11.21)

$$R = \frac{1}{4}\sum_{ij}(k_i\delta_{ij} - A_{ij})s_i s_j = \frac{1}{4}\sum_{ij} L_{ij}s_i s_j,$$

where $L_{ij} = k_i \delta_{ij} - A_{ij}$ is the ijth element of the graph Laplacian matrix—see Eq. (6.44) .

Equation (11.21) can be written in matrix form as (11.22)

$$R = \tfrac{1}{4}\mathbf{s}^T\mathbf{L}\mathbf{s},$$

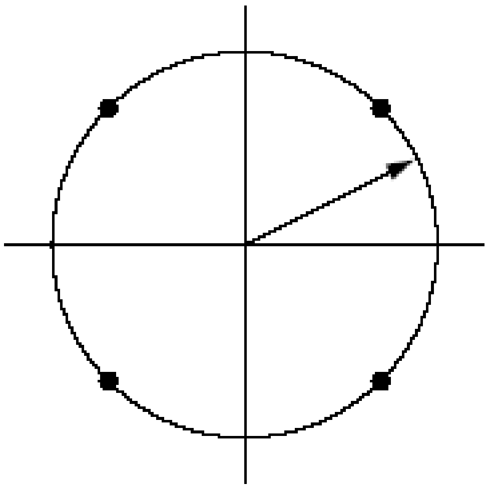
where s is the vectorwith elements s_i . This expression gives us a matrix formulation of the graph partitioning problem. The matrix L specifies the structure (p. 366) of our network, the vector s defines a division of that network into groups, and our goal is to find the vector s that minimizes the cut size (11.22) for given L.

You will probably not be surprised to learn that, in general, this minimization problem is not an easy one. If it were easy then we would have a corresponding easy way to solve the partitioning problem and, as discussed in Section 11.3.1 , there are good reasons to believe that partitioning has no easy solutions.

What makes our matrix version of the problem hard in practice is that the s_i cannot take just any values. They are restricted to the special values ±1. If they were allowed to take any real values the problem would be much easier; we could just differentiate to the find the optimum.

This suggests a possible approximate approach to the minimization problem. Suppose we indeed allow the s_i to take any values (subject to a couple of basic constraints discussed below) and then find the values that minimize R. These values will only be approximately the correct ones, since they probably won't be ±1, but they may nonetheless be good enough to give us a handle on the optimal partitioning. This idea leads us to the so-called *relaxation method*, which is one of the standard methods for the approximate solution of vector optimization problems such as this one. In the present context it works as follows.

The allowed values of the s_i are actually subject to two constraints. First, as we have said, each individual one is allowed to take only the values ±1. If we



The relaxation of the constraint allows s to point to any position on a hypersphere circumscribing the original hypercube, rather than just the corners of the hypercube.

regard s as a vector in a Euclidean space then this constraint means that the vector always points to one of the 2^n corners of an n-dimensional hypercube centered on the origin, and always has the same length, which is

$\sqrt{n}$

. Let us relax the constraint on the vector's direction, so that it can point in any direction in its n-dimensional space. We will however still keep its length the same. (It would not make sense to allow the length to vary. If we did that then the minimization of R would have the obvious trivial solution s = 0, which would tell us nothing.) So s will be allowed to take any value, but subject to the constraint that

$|s| = \sqrt{n}$

, or equivalently (11.23)

$$\sum_i s_i^2 = n.$$

Another way of putting this is that s can now point to any location on the surface of a hypersphere of radius

$\sqrt{n}$

in our n-dimensional Euclidean space. The hypersphere includes the original allowed values at the corners of the hypercube, but also includes other points in between.

The second constraint on the s_i is that the numbers of them that are equal to +1 and -1 respectively must equal the desired sizes of the two groups. If (p. 367) those two sizes are n_1 and n_2, this second constraint can be written as (11.24)

$$\sum_i s_i = n_1 - n_2.$$

or in vector notation (11.25)

$$\mathbf{1}^T \mathbf{s} = n_1 - n_2,$$

where **1** is the vector (1, 1, 1,...) whose elements are all 1. We keep this second constraint unchanged in our relaxed calculations, so that our partitioning problem, in its relaxed form, is a problem of minimizing the cut size, Eq. (11.22), subject to the two constraints (11.23) and (11.24).

This problem is now just a standard piece of algebra. We differentiate with respect to the elements s_i, enforcing the constraints using two Lagrange multipliers, which we denote λ and 2μ (the extra 2 being merely for notational convenience): (11.26)

$$\frac{\partial}{\partial s_i}\left[\sum_{jk} L_{jk} s_j s_k + \lambda\Bigl(n - \sum_j s_j^2\Bigr) + 2\mu\Bigl((n_1 - n_2) - \sum_j s_j\Bigr)\right] = 0.$$

Performing the derivatives, we then find that (11.27)

$$\sum_j L_{ij} s_j = \lambda s_i + \mu,$$

or, in matrix notation (11.28)

$$\mathbf{L}\mathbf{s} = \lambda \mathbf{s} + \mu \mathbf{1}.$$

We can calculate the value of μ by recalling that **1** is an eigenvector of the Laplacian with eigenvalue zero so that **L** · **1** = **0** (see Section 6.13.2). Multiplying (11.28) on the left by $\mathbf{1}^T$ and making use of Eq. (11.25), we then find that $\lambda(n_1 - n_2) + \mu n = 0$, or (11.29)

$$\mu = -\frac{n_1 - n_2}{n}\lambda.$$

If we define the new vector (11.30)

$$\mathbf{x} = \mathbf{s} + \frac{\mu}{\lambda}\mathbf{1} = \mathbf{s} - \frac{n_1 - n_2}{n}\mathbf{1},$$

then Eq. (11.28) tells us that (11.31)

$$\mathbf{L}\mathbf{x} = \mathbf{L}\left(\mathbf{s} + \frac{\mu}{\lambda}\mathbf{1}\right) = \mathbf{L}\mathbf{s} = \lambda\mathbf{s} + \mu\mathbf{1} = \lambda\mathbf{x},$$

where we have used L · 1 = 0 again.

In other words, x is an eigenvector of the Laplacian with eigenvalue λ. We are still free to choose which eigenvector it is-any eigenvector will satisfy (p. 368) Eq. (11.31) —and clearly we should choose the one that gives the smallest value of the cut size *R*. Notice, however, that (11.32)

$$\mathbf{1}^T\mathbf{x} = \mathbf{1}^T\mathbf{s} - \frac{\mu}{\lambda}\mathbf{1}^T\mathbf{1} = (n_1 - n_2) - \frac{n_1 - n_2}{n}n = 0,$$

where we have used Eq. (11.25) . Thus x is orthogonal to 1, which means that, while it should be an eigenvector of L, it cannot be the eigenvector (1, 1, 1, …) that has eigenvalue zero.

So which eigenvector should we choose? To answer this question we note that (11.33)

$$R = \tfrac{1}{4}\mathbf{s}^T\mathbf{L}\mathbf{s} = \tfrac{1}{4}\mathbf{x}^T\mathbf{L}\mathbf{x} = \tfrac{1}{4}\lambda\mathbf{x}^T\mathbf{x}.$$

But from Eq. (11.30) we have (11.34)

$$\begin{aligned}\mathbf{x}^T\mathbf{x} &= \mathbf{s}^T\mathbf{s} + \frac{\mu}{\lambda}(\mathbf{s}^T\mathbf{1} + \mathbf{1}^T\mathbf{s}) + \frac{\mu^2}{\lambda^2}\mathbf{1}^T\mathbf{1}\\ &= n - 2\frac{n_1 - n_2}{n}(n_1 - n_2) + \frac{(n_1 - n_2)^2}{n}n\\ &= 4\frac{n_1 n_2}{n},\end{aligned}$$

and hence (11.35)

$$R = \frac{n_1 n_2}{n}\lambda.$$

Thus the cut size is proportional to the eigenvalue λ. Given that our goal is to minimize *R*, this means we should choose x to be the eigenvector corresponding to the smallest allowed eigenvalue of the Laplacian. All the eigenvalues of the Laplacian are non-negative (see Section 6.13.2). The smallest one is the zero eigenvalue that corresponds to the eigenvector (1, 1, 1, ….) but we have already ruled this one out—x has to be orthogonal to this lowest eigenvector. Thus the best thing we can do is choose x proportional to the eigenvector v $_2$ corresponding to the second lowest eigenvalue λ_2, with its normalization fixed by Eq. (11.34) .

Finally, we recover the corresponding value of s from Eq. (11.30) thus: (11.36)

$$\mathbf{s} = \mathbf{x} + \frac{n_1 - n_2}{n}\mathbf{1},$$

or equivalently (11.37)

$$s_i = x_i + \frac{n_1 - n_2}{n}.$$

This gives us the optimal relaxed value of s.

As we have said, however, the real vector s is subject to the additional constraints that its elements take the values ±1 and moreover that exactly n_1 of (p. 369) them are +1 and the other n_2 are -1. Typically these constraints will prevent s from taking exactly the value given by Eq. (11.37) . Let us, however, do the best we can and choose s to be as close as possible to our ideal value subject to its constraints, which we do by making the product (11.38)

$$\mathbf{s}^T\left(\mathbf{x} + \frac{n_1 - n_2}{n}\mathbf{1}\right) = \sum_i s_i\left(x_i + \frac{n_1 - n_2}{n}\right)$$

as large as possible. The maximum of this expression is achieved by assigning s_i = +1 for the vertices with the largest (i.e., most positive) values of x_i +(n_1 - n_2)/n and s_i = -1 for the remainder.

Note however that the most positive values of x_i + (n_1?n_2)/n are also the most positive values of *xi*, which are in turn also the most positive elements of the eigenvector v $_2$ (to which, as we have said, x is proportional). So after this moderately lengthy derivation we actually arrive at a very simple final prescription for dividing our network. We calculate the eigenvector v $_2$, which has n elements, one for each vertex in the network, and place the n_1 vertices with the most positive elements in group 1 and the rest in group 2.

There is one further small subtlety. It is arbitrary which group we call group 1 and which we call group 2, and hence which one we assign to the more positive elements of the eigenvector and which to the more negative. Thus, if the sizes of the two groups are different there are two different ways of making the split—either the larger or the smaller group could correspond to the more positive values. (In the geometrical language of our vectors, this is equivalent to saying our eigenvector calculation might find the vector x that we actually want, or minus that vector—both are good eigenvectors of the Laplacian.) To get around this problem, we simply compute the cut size for both splits of the network and choose the one with the smaller value.

Thus our final algorithm is as follows:

1. Calculate the eigenvector v_2 corresponding to the second smallest eigenvalue λ_2 of the graph Laplacian.
2. Sort the elements of the eigenvector in order from largest to smallest.
3. Put the vertices corresponding to the n_1 largest elements in group 1, the rest in group 2, and calculate the cut size.
4. Then put the vertices corresponding to the n_1 smallest elements in group 1, the rest in group 2, and recalculate the cut size.
5. Between these two divisions of the network, choose the one that gives the smaller cut size.

In Fig. 11.3c we show the result of the application of this method to the same mesh network that we studied in conjunction with the Kernighan-Lin algorithm. In this case the spectral method finds a division of the network (p. 370) very similar to that given by the Kernighan-Lin algorithm, although the cut size is slightly worse—the spectral method cuts 46 edges in this case, where the Kernighan-Lin algorithm cut only 40. This is typical of the spectral method. It tends to find divisions of a network that have the right general shape, but are not perhaps quite as good as those returned by other methods.

An advantage of the spectral approach, however, is its speed. The timeconsuming part of the algorithm is the calculation of the eigenvector v_2, which takes time $O(mn)$ using either the orthogonalization method or the Lanczos method (see Section 11.1.2), or $O(n^2)$ on a sparse network having $m \# n$. This is one factor of n better than the $O(n^3)$ of the Kernighan-Lin algorithm, which makes the algorithm feasible for much larger networks. Spectral partitioning can be extended to networks of hundreds of thousands of vertices, where the Kernighan-Lin algorithm is restricted to networks of a few thousand vertices at most.

The second eigenvalue of the Laplacian has come up previously in this book in Section 6.13.3 , where we saw that it is non-zero if and only if a network is connected. The second eigenvalue is for this reason sometimes called the *algebraic connectivity* of a network. In this section we have seen it again in another context, that of partitioning. What happens if a network is not connected and the second eigenvalue is zero? In that case, the two lowest eigenvalues are the same, and the corresponding eigenvectors are indeterminate—any mixture of two eigenvectors with the same eigenvalue is also an eigenvector. This is not however a serious problem. If the network is not connected, having more than one component, then usually we are interested either in partitioning one particular component, such as the largest component, or in partitioning all components individually, and so we just treat the components separately as connected networks according to the algorithm above.

The algebraic connectivity itself appears in our expression for the cut size, Eq. (11.35), and indeed is a direct measure of the cut size, being directly proportional to it, at least within the "relaxed" approximation used to derive the equation. Thus the algebraic connectivity is a measure of *how easily* a network can be divided. It is small for networks that have good cuts and large for those that do not. This in a sense is a generalization of our earlier result that the algebraic connectivity is non-zero for connected networks and zero for unconnected ones—we now see that *how* non-zero it is is a measure of how connected the network is. (p. 371)

11.6 Community Detection

In the last few sections we looked at the problem of graph partitioning, the division of network vertices into groups of given number and size, so as to minimize the number of edges running between groups. A complementary problem,introduced in Section 11.2.1 , is that of community detection, the search for the naturally occurring groups in a network regardless of their number or size, which is used primarily as a tool for discovering and understanding the large-scale structure of networks.

The basic goal of community detection is similar to that of graph partitioning: we want to separate the network into groups of vertices that have few connections between them. The important difference is that the number or size of the groups is not fixed. Let us focus to begin with on a very simple example of a community detection problem, probably *the* simplest, which is analogous to the graph bisection problems we examined in previous sections. We will consider the problem of dividing a network into just two non-overlapping groups or communities, as previously, but now without any constraint on the sizes of the groups, other than that the sum of the sizes should equal the size n of the whole network. Thus, in this simple version of the problem, the number of groups is still specified but their sizes are not, and we wish to find the "natural" division of the network into two groups, the fault line (if any) along which the network inherently divides, although we haven't yet said precisely what we mean by that, so that the question we're asking is not yet well defined.

Our first guess at how to tackle this problem might be simply to find the division with minimum cut size, as in the corresponding graph partitioning problem, but without any constraint on the sizes of our groups. However, a moment's reflection reveals that this will not work. If we divide a network into two groups with any number of vertices allowed in the groups then the optimum division is simply to put all the vertices in one of the groups and none of them in the other. This trivial division insures that the cut size between the two groups will be zero—there will be no edges between groups because one of the groups contains no vertices! As an answer to our community detection problem, however, it is clearly not useful.

One way to do better would be to impose loose constraints of some kind on the sizes of the groups. That is, we could allow the sizes of the groups to vary, but not too much. An example of this type of approach is *ratio cut partitioning* in which, instead of minimizing the standard cut size R, we instead minimize the ratio $R/(n_1 n_2)$, where n_1 and n_2 are the sizes of the two groups. The denominator $n_1 n_2$ has its largest value, and hence reduces the ratio by the largest amount, when n_1 and n_2 are equal $n_1 = n_2 = \frac{1}{2} n$. For unequal group (p. 372) sizes the denominator becomes smaller the greater the inequality, and diverges when either group size becomes zero. This effectively eliminates solutions in which all vertices are placed in the same group, since such solutions never give the minimum value of the ratio, and biases the division towards those solutions in which the groups are of roughly equal size.

As a tool for discovering the natural divisions in a network, however, the ratio cut is not ideal. In particular, although it allows group sizes to vary it is still biased towards a particular choice, that of equally sized groups. More importantly, there is no principled rationale behind its definition. It works reasonably well in some circumstances, but there's no fundamental reason to believe it will give sensible answers or that some other approach will not give better ones.

An alternative strategy is to focus on a different measure of the quality of a division other than the simple cut size or its variants. It has been argued that the cut size is not itself a good measure because a good division of a network into communities is not merely one in which there are few edges between communities. On the contrary, the argument goes, a good division is one where there are *fewer than expected* such edges. If we find a division of a network that has few edges between its groups, but nonetheless the number of such edges is about what we would have expected were edges simply placed at random in the network, then most people would say we haven't found anything significant. It is not the total cut size that matters, but how that cut size compares with what we expect to see.

In fact, in the conventional development of this idea one considers not the number of edges between groups but the number within groups. The two approaches are equivalent, however, since every edge that lies within a group necessarily does not lie between groups, so one can calculate one number from the other given the total number of edges in the network as whole. We will follow convention here and base our calculations on the numbers of with in group edges.

Our goal therefore will be to find a measure that quantifies how many edges lie within groups in our network relative to the number of such edges expected on the basis of chance. This, however, is an idea we have encountered before. In Section 7.13.1 we considered the phenomenon of assortative mixing in networks, in which vertices with similar characteristics tend to be connected by edges. There we introduced the measure of assortative mixing known as modularity, which has a high value when many more edges in a network fall between vertices of the same type than one would expect by chance. This is precisely the type of measure we need to solve our current community detection problem. If we consider the vertices in our two groups to be vertices (p. 373) of two types then good divisions of the network into communities are precisely those that have high values of the corresponding modularity.

Thus one way to detect communities in networks is to look for the divisions that have the highest modularity scores and in fact this is the most commonly used method for community detection. Like graph partitioning, modularity maximization is a hard problem (see Section 11.3.1). It is believed that, as with partitioning, the only algorithms capable of always finding the division with maximum modularity take exponentially long to run and hence are useless for all but the smallest of networks [54]. Instead, therefore, we turn again to heuristic algorithms, algorithms that attempt to maximize the modularity in an intelligent way that gives reasonably good results most of the time.

11.7 Simple Modularity Maximization

One straightforward algorithm for maximizing modularity is the analog of the Kernighan-Lin algorithm [245]. This algorithm divides networks into two communities starting from some initial division, such as a random division into equally sized groups. The algorithm then considers each vertex in the network in turn and calculates how much the modularity would change if that vertex were moved to the other group. It then chooses among the vertices the one whose movement would most increase, or least decrease, the modularity and moves it. Then it repeats the process, but with the important constraint that a vertex once moved cannot be moved again, at least on this round of the algorithm.

And so the algorithm proceeds, repeatedly moving the vertices that most increase or least decrease the modularity. Notice that in this algorithm we are not swapping pairs as we did in the Kernighan-Lin algorithm. In that algorithm we were required to keep the sizes of the groups constant, so for every vertex removed from a group we also had to add one. Now we no longer have such a constraint and so we can move single vertices on each step.

When all vertices have been moved exactly once, we go back over the states through which the network has passed and select the one with the highest modularity. We then use that state as the starting condition for another round of the same algorithm, and we keep repeating the whole process until the modularity no longer improves.

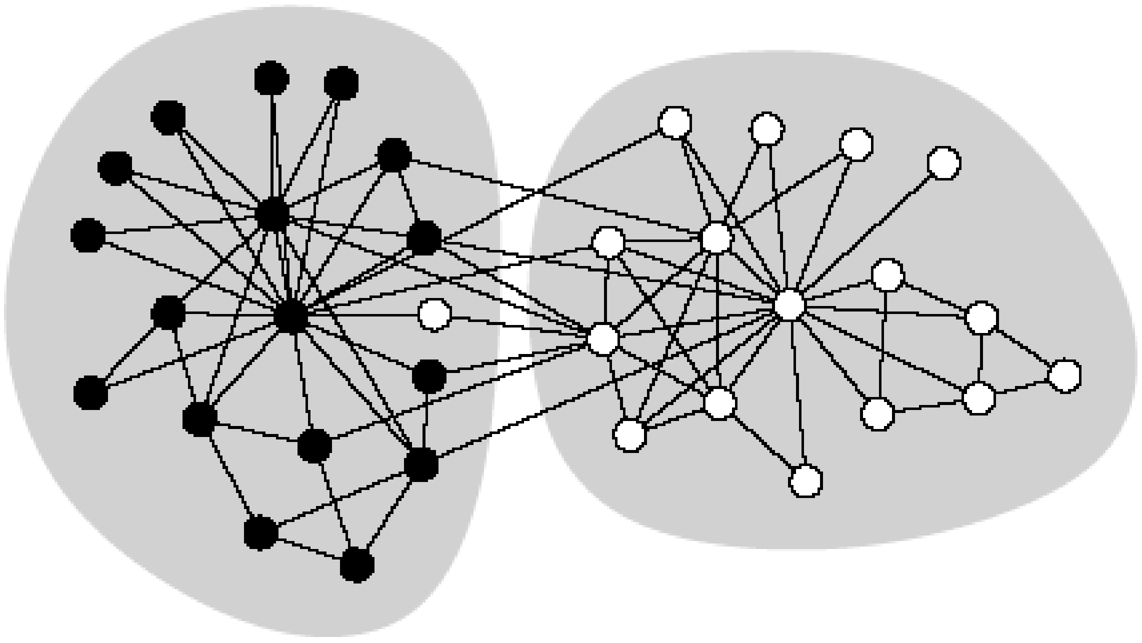


Figure 11.4: Modularity maximization applied to the karate club network. When we apply our vertex-moving modularity maximization algorithm to the karate club network, the best division found is the one indicated here by the two shaded regions, which split the network into two groups of 17 vertices each. This division is very nearly the same as the actual split of the network in real life (open and solid circles), following the dispute among the club's members. Just one vertex is classified incorrectly.

Figure 11.4 shows an example application of this algorithm to the "karate club" network of Zachary, which we encountered previously in Chapter 1 (see Fig. 1.2 on page 6). This network represents the pattern of friendships between members of a karate club at a North American university, as determined by direct observation of the club's members by the experimenter over a period (p. 374) of about two years. The network is interesting because during the period of observation a dispute arose among the members of the club over whether to raise the club's fees and as a result the club eventually split into two parts, of 18 and 16 members respectively, the latter departing to form their own club. The colors of the vertices in Fig. 11.4 denote the members of the two factions, while the shaded regions show the communities identified in the network by our vertex-moving algorithm. As we can see from the figure, the communities identified correspond almost perfectly to the known groups in the network. Just one vertex on the border between the groups is incorrectly assigned. Thus in this case our algorithm appears to have picked out structure of genuine sociological interest from an analysis of network data alone. It is precisely for results of this kind, that shed light on potentially important structural features of networks, that community detection methods are of interest.

The vertex moving algorithm is also quite efficient. At each step of the algorithm we have to evaluate the modularity change due to the movement of each of $O(n)$ vertices, and each such evaluation, like the corresponding ones for the Kernighan-Lin algorithm, can be achieved in time $O(m/n)$ if the network is stored as an adjacency list. Thus each step takes time $O(m)$ and there are n steps in one complete round of the algorithm for a total time of $O(mn)$. This is considerably better than the $O(mn^2)$ of the Kernighan-Lin algorithm, (p. 375) and the algorithm is in fact one of the better of the many proposed algorithms for modularity maximization.[10] The fundamental reason for the algorithm's speed is that when moving single vertices we only have to consider $O(n)$ possible moves at each step, by contrast with the $O(n_2)$ possible swaps of vertex pairs that must be consider in a step of the Kernighan-Lin algorithm.

11.8 Spectral Modularity Maximization

Having seen in the previous section an algorithm for modularity maximization analogous to the Kernighan-Lin algorithm, it is natural to ask whether there also exists an analog for community detection of the spectral graph partitioning algorithm of Section 11.5 . The answer is yes, there is indeed such an algorithm, as we now describe.

In Section 7.13.1 we wrote an expression for the modularity of a division of a network as follows (Eq. (7.69)): (11.39)

$$Q = \frac{1}{2m}\sum_{ij}\left(A_{ij} - \frac{k_i k_j}{2m}\right)\delta(c_i, c_j) = \frac{1}{2m}\sum_{ij} B_{ij}\,\delta(c_i, c_j),$$

where c_i is the group or community to which vertex i belongs, $\delta(m, n)$ is the Kronecker delta, and (11.40)

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m}.$$

Note that B_{ij} has the property (11.41)

$$\sum_j B_{ij} = \sum_j A_{ij} - \frac{k_i}{2m}\sum_j k_j = k_i - \frac{k_i}{2m}2m = 0,$$

and similarly for sums over i. (We have made use of Eq. (6.20) in the second equality.) This property will be important shortly.

Let us again consider the division of a network into just two parts (we will consider the more general case later) and again represent such a division by the quantities (11.42)

$$s_i = \begin{cases} +1 & \text{if vertex } i \text{ belongs to group 1,} \\ -1 & \text{if vertex } i \text{ belongs to group 2.} \end{cases}$$

(p. 376) We note that the quantity ½ $(s_i s_j + 1)$ is 1 if i and j are in the same group and zero otherwise, so that (11.43)

$$\delta(c_i, c_j) = \tfrac{1}{2}(s_i s_j + 1).$$

Substituting this expression into Eq. (11.39) , we find (11.44)

$$Q = \frac{1}{4m}\sum_{ij} B_{ij}\,(s_i s_j + 1) = \frac{1}{4m}\sum_{ij} B_{ij}\, s_i s_j,$$

where we have used Eq. (11.41) . In matrix terms we can write this as (11.45)

$$Q = \frac{1}{4m}\,\mathbf{s}^T\mathbf{B}\mathbf{s},$$

where s is, as before, the vector with elements s_i , and B is the $n \times n$ matrix with elements B_{ij} , also called the *modularity matrix*.

Equation (11.45) is similar in form to our expression, Eq. (11.22) , for the cut size of a network in terms of the graph Laplacian. By exploiting this similarity we can derive a spectral algorithm for community detection that is closely analogous to the spectral partitioning method of Section 11.5 .

We wish to find the division of a given network that maximizes the modularity Q. That is, we wish to find the value of s that maximizes Eq. (11.45) for a given modularity matrix B. The elements of s are constrained to take values ±1, so that the vector always points to one of the corners of an $n$dimensional hypercube, but otherwise there are no constraints on the problem. In particular, the number of elements with value +1 or -1 is not fixed as it was in the corresponding graph partitioning problem—the sizes of our communities are unconstrained.

As before, this optimization problem is a hard one, but it can be tackled approximately —and effectively—by a relaxation method. We relax the constraint that s must point to a corner of the hypercube and allow it to point in any direction, though keeping its length the same, meaning that it can take any real value subject only to the constraint that (11.46)

$$\mathbf{s}^T\mathbf{s} = \sum_i s_i^2 = n.$$

The maximization is now a straightforward problem. We maximize Eq. (11.44) by differentiating, imposing the constraint with a single Lagrange multiplier β: (11.47)

$$\frac{\partial}{\partial s_i}\left[\sum_{jk} B_{jk}s_js_k + \beta\Big(n - \sum_j s_j^2\Big)\right] = 0.$$

When we perform the derivatives, this gives us (11.48)

$$\sum_j B_{ij}s_j = \beta s_i,$$

(p. 377) or in matrix notation (11.49)

$$\mathbf{Bs} = \beta\mathbf{s}.$$

In other words, s is one of the eigenvectors of the modularity matrix. Substituting (11.49) back into Eq. (11.45) , we find that the modularity itself is given by (11.50)

$$Q = \frac{1}{4m}\beta\mathbf{s}^T\mathbf{s} = \frac{n}{4m}\beta,$$

where we have used Eq. (11.46) . For maximum modularity, therefore, we should choose s to be the eigenvector u1 corresponding to the largest eigenvalue of the modularity matrix.

As before, we typically cannot in fact choose s = u1, since the elements of s are subject to the constraint s_i = ±1. But we do the best we can and choose it as close to u1 as possible, which means maximizing the product (11.51)

$$\mathbf{s}^T\mathbf{u}_1 = \sum_i s_i[\mathbf{u}_1]_i,$$

where $[u1]_i$ is the *i*th element of u_1. The maximum is achieved when each term in the sum is non-negative, i.e., when (11.52)

$$s_i = \begin{cases} +1 & \text{if } [\mathbf{u}_1]_i > 0, \\ -1 & \text{if } [\mathbf{u}_1]_i < 0. \end{cases}$$

In the unlikely event that a vector element is exactly zero, either value of s_i is equally good and we can choose whichever we prefer.

And so we are led the following very simple algorithm. We calculate the eigenvector of the modularity matrix corresponding to the largest (most positive) eigenvalue and then assign vertices to communities according to the signs of the vector elements, positive signs in one group and negative signs in the other.

In practice this method works very well. For example, when applied to the karate club network of Fig. 11.4 it works perfectly, classifying every one of the 34 vertices into the correct group.

One potential problem with the algorithm is that the matrix B is, unlike the Laplacian, not sparse, and indeed usually has all elements non-zero. At first sight, this appears to make the algorithm's complexity significantlyworse than that of the normal spectral bisection algorithm; as discussed in Section 11.1.1 , finding the leading eigenvector of a matrix takes time O(mn), which is equivalent to O(n^3) in a dense matrix, as opposed to O(n^2) in a sparse one. In fact, however, by exploiting special properties of the modularity matrix it is still possible to find the eigenvector in time O(n_2) on a sparse network. The details can be found in [246].

(p. 378) Overall, this means that the spectral method is about as fast as, but not significantly faster than, the vertex-moving algorithm of Section 11.7 . Both have time complexity O(n^2) on sparse networks.[11] There is, however, merit to having both algorithms. Given that all practical modularity maximizing algorithms are merely heuristics—clever perhaps, but not by any means guaranteed to perform well in all cases—having more than one fast algorithm in our toolkit is always a good thing.

11.9 Division Into More Than Two Groups

The community detection algorithms of the previous two sections both perform a limited form of community detection, the division of a network into exactly two communities, albeit of unspecified sizes. But “communities” are defined to be the natural groupings of vertices in networks and there is noreason to suppose that networks will in general have just two of them. They might have two, but they might have more than two, and we would like to be able to find them whatever their number. Moreover we don't, in general, want to have to specify the number of communities; that number should be fixed by the structure of the network and not by the experimenter.

In principle, the modularity maximization method can handle this problem perfectly well. Instead of maximizing modularity over divisions of a network into two groups, we should just maximize it over divisions into any number of groups. Modularity is supposed to be largest for the best division of the network, no matter how many groups that division possesses.

There are a number of community detection algorithms that take this "free maximization" approach to determining community number, and we discuss some of them in the following section. First, however, we discuss a simpler approach which is a natural extension of the methods of previous sections and of our graph partitioning algorithms, namely repeated bisection of a network. We start by dividing the network first into two parts and then we further subdivide those parts in to smaller ones, and so on.

One must be careful about how one does this, however. We cannot proceed as one can in the graph partitioning case and simply treat the communities found in the initial bisection of a network as smaller networks in their (p. 379) own right, applying our bisection algorithm to those smaller networks. The modularity of the complete network does not break up (as cut size does) into independent contributions from the separate communities and the individual maximization of the modularities of those communities treated as separate networks will not, in general, produce the maximum modularity for the network as a whole.

Instead, we must consider explicitly the change δQ in the modularity of the entire network upon further bisecting a community c of size n_c . That change is given by (11.53)

$$\begin{aligned}\Delta Q &= \frac{1}{2m}\left[\frac{1}{2}\sum_{i,j\in c} B_{ij}(s_i s_j + 1) - \sum_{i,j\in c} B_{ij}\right]\\ &= \frac{1}{4m}\left[\sum_{i,j\in c} B_{ij}s_i s_j - \sum_{i,j\in c} B_{ij}\right] = \frac{1}{4m}\sum_{i,j\in c}\left[B_{ij} - \delta_{ij}\sum_{k\in c} B_{ik}\right]s_i s_j\\ &= \frac{1}{4m}\mathbf{s}^T\mathbf{B}^{(c)}\mathbf{s},\end{aligned}$$

where we have made use of

$s_i^2 = 1$

, and B(c) is the $n_c \times n_c$ matrix with elements (11.54)

$$\mathrm{B}_{ij}^{(c)} = B_{ij} - \delta_{ij}\sum_{k\in c} B_{ik}.$$

Since Eq. (11.53) has the same general form as Eq. (11.45) we can now apply our spectral approach to this generalized modularity matrix, just as before, to maximize δQ, finding the leading eigenvector and dividing the network according to the signs of its elements.

In repeatedly subdividing a network in this way, an important question we need to address is at what point to halt the subdivision process. The answer is quite simple. Given that our goal is to maximize the modularity for the entire network, we should only go on subdividing groups so long as doing so results in an increase in the overall modularity. If we are unable to find any division of a community that results in a positive change δQ in the modularity, then we should simply leave that community undivided. The practical indicator of this situation is that our bisection algorithm will put all vertices in one of its two groups and none in the other, effectively refusing to subdivide the community rather than choose a division that actually decreases the modularity. When we have subdivided our network to the point where all communities are in this indivisible state, the algorithm is finished and we stop.

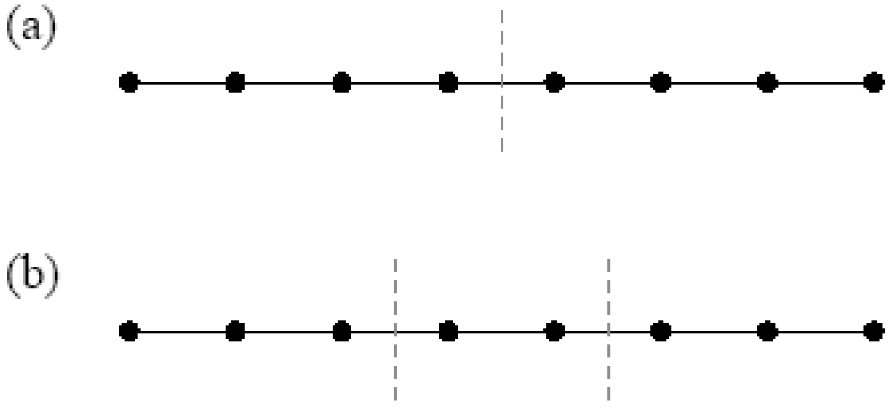


Figure 11.5: Division of a simple network by repeated maximization of the modularity. (a) The optimal bisection of this network of eight vertices and seven edges is straight down the middle. (b) The optimal division into an arbitrary number of groups is this division into three.

This repeated bisection method works well in many situations, but it is by no means perfect. A particular problem is that, as in the equivalent approach to graph partitioning, there is no guarantee that the best division of a network (p. 380) into, say, three parts, can be found by first finding the best division into two parts and then subdividing one of the two. Consider for instance the simple network shown in Fig. 11.5 , which consists of eight vertices joined together in a line. The bisection of this network with highest modularity is the one shown in Fig. 11.5a , down the middle of the network, splitting it into two equally sized groups of four vertices each. The best modularity if the number of groups is unconstrained, however, is that shown in Fig. 11.5b , with three groups of sizes 3, 2, and 3, respectively. A repeated optimal bisection algorithm would never find the division in 11.5b because, having first made the bisection in 11.5a, there is no further bisection that will get us to 11.5b.

As mentioned above, an alternative method for dividing networks into more than two communities is to attempt to find directly the maximum modularity over divisions into any number of groups. This approach can, in principle, find better divisions than repeated bisection, but in practice is more complicated to implement and often runs slower. A number of promising methods have been developed, however, some of which are discussed in the next section.

11.10 Other Modularity Maximization Methods

There are a great variety of general algorithms for maximizing (or minimizing) functions over sets of states, and in theory any one of them could be brought to bear on the modularity maximization problem, thereby creating a new community detection algorithm. We describe briefly here three approaches that have met with some success. Each of these approaches attempts to maximize modularity over divisions into any number of communities of any sizes and (p. 381) thus to determine both the number and size of communities in the process.

One of the most widely used general optimization strategies is *simulated annealing*, which proceeds by analogy with the physics of slow cooling or "annealing" of solids. It is known that a hot system, such as a molten metal, will, if cooled sufficiently slowly to a low enough temperature, eventually find its *ground state*, that state of the system that has the lowest possible energy. Simulated annealing works by treating the quantity of interest—modularity in this case—as an energy and then simulating the cooling process until the system finds the state with the lowest energy. Since we are interested in finding the highest modularity, not the lowest, we equate energy in our case with *minus* the modularity, rather than with the modularity itself.

The details of the simulated annealing method are beyond the scope of this book, but the application to modularity maximization is a straightforward one and it appears to work very well [85, 150, 151, 215, 281]. For example, Danon *et al.* [85] performed an extensive test in which they compared the performance of a large number of different community detection algorithms on standardized tasks and found that the simulated annealing method gave the best results of any method tested. The main disadvantage of the approach is that it is slow, typically taking several times as long to reach an answer as competing methods do.

Another general optimization method is the genetic algorithm, a method inspired by the workings of biological evolution. Just as fitter biological species reproduce more and so pass on the genes that confer that fitness to future generations, so one can consider a population of different divisions of the same network and assign to each a "fitness" proportional to its modularity. Over a series of generations one simulates the preferential "reproduction" of high modularity divisions, while those of low modularity die out. Small changes or mutations are introduced into the offspring divisions, allowing their modularity values either to improve or get worse and those that improve are more likely to survive in the next generation while those that get worse are more likely to be killed off. After many generations one has a population of divisions with good modularity and the best of these is the final division returned by the algorithm. Like simulated annealing the method appears to give results of high quality, but is slow, which restricts its use to networks of a few hundred vertices or fewer [295].

A third method makes use of a so-called *greedy algorithm*. In this very simple approach we start out with each vertex in our network in a one-vertex group of its own, and then successively amalgamate groups in pairs, choosing at each step the pair whose amalgamation gives the biggest increase in modularity, or the smallest decrease if no choice gives an increase. Eventually (p. 382) all vertices are amalgamated into a single large community and the algorithm ends. Then we go back over the states through which the network passed during the course of the algorithm and select the one with the highest value of the modularity. A naive implementation of this idea runs in time $O(n^2)$, but by making use of suitable data structures the run time can be improved to $O(n \log^2 n)$ on a sparse graph [71, 319]. Overall the algorithm works only moderately well: it gives reasonable divisions of networks, but the modularity values achieved are in general somewhat lower than those found by the other methods described here. On the other hand, the running time of the method may be the best of any current algorithm, and this is one of the few algorithms fast enough to work on the very largest networks now being explored. Wakita and Tsurumi [319] have given one example of an application to a network of more than five million vertices, something of a record for studies of this kind.

11.11 Other Algorithms For Community Detection

As we have seen, the problem of detecting communities in networks is a less well-posed one than the problem of graph partitioning. In graph partitioning the goal is clear: to find the division of a network with the smallest possible cut size. There is, by contrast, no universally agreed upon definition of what constitutes a good division of a network into communities. In the previous sections we have looked at algorithms based one particular definition in terms of the modularity function, but there are a number of other definitions in common use that lead to different algorithms. In the following sections we look briefly at a few of these other algorithms.

11.11.1 Betweenness-Based Methods

One alternative way of finding communities of vertices in a network is to look for the edges that lie between communities. If we can find and remove these edges, we will be left with just the isolated communities.

There is more than one way to quantify what we mean when we say an edge lies “between communities,” but one common approach is to use between ness centrality. As described in Section 7.7 , the betweenness centrality of a vertex in a network is the number of geodesic (i.e., shortest) paths in the network that pass through that vertex. Similarly, we can define an *edge betweenness* that counts the number of geodesic paths that run along edges and, as shown in Fig. 11.6 , edges that lie between communities can be expected to have high values of the edge betweenness.

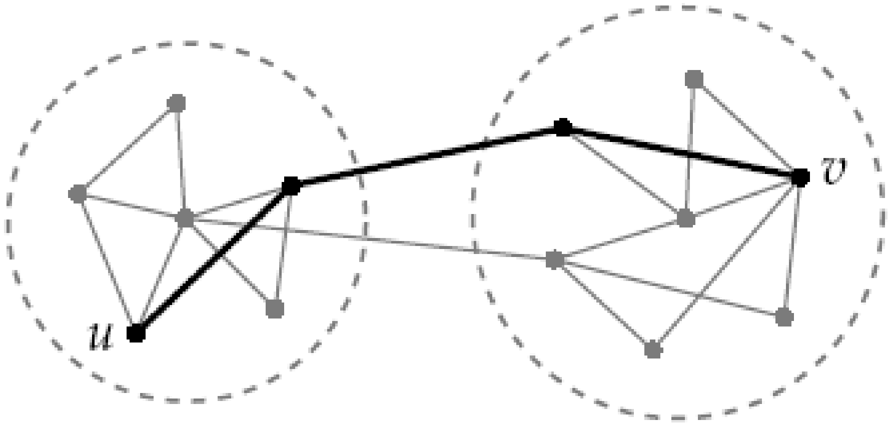


Figure 11.6: Identification of between-group edges. This simple example network is divided into two groups of vertices (denoted by the dotted lines), with only two edges connecting the groups. Any path joining vertices in different groups (such as vertices *u* and *v*) must necessarily pass along one of these two edges. Thus if we consider a set of paths between all pairs of vertices (such as geodesic paths, for instance), we expect the between-group edges to carry more paths than most. By counting the number of paths that pass along each edge we can in this way identify the between-group edges.

(p. 383)

The calculation of edge betweenness is precisely analogous to the vertex case: we consider the geodesic path or paths between every pair of vertices in the network (except vertices in different components, for which no such path exists), and count how many such paths go along each edge. Edge betweenness can be calculated for all edges in time $O(n(m+n))$ using a slightly modified version of the algorithm described in Section 10.3.6 [250].

Our algorithm for detecting communities is then as follows. We calculate the betweenness scores of all edges in our network and then search through them for the edge with the highest score and remove it. In removing the edge we will change the betweenness scores of some edges, because any shortest paths that previously traversed the removed edge will now have to be rerouted another way. So we must recalculate the betweenness scores following the removal. Then we search again for the edge with the highest score and remove it, and so forth. As we remove one edge after another an initially connected network will eventually split into two pieces, and then into three, and so on.

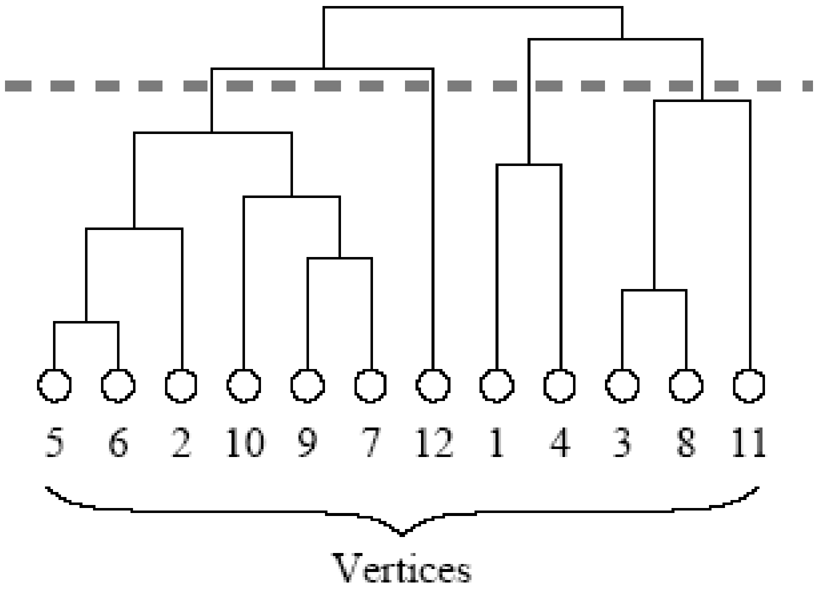


Figure 11.7: **A dendrogram.** The results of the edge betweenness algorithm can be represented as a tree or "dendrogram" in which the vertices are depicted (conventionally) at the bottom of the tree and the "root" at the top represent the whole network. The progressive fragmentation of the network as edges are removed one by one is represented by the successive branching of the tree as we move down the figure and the identities of the vertices in a connected subset at any point in the procedure can be found by following the lines of the tree down to the bottom of the picture. Each intermediate division of the network through which the algorithm passes corresponds to a horizontal cut through the dendrogram. For instance, the cut denoted by the dotted line in this dendrogram splits the network into four groups of 6, 1, 2, and 3 vertices respectively.

The progress of the algorithm can be represented using a tree or *dendrogram* like that depicted in Fig. 11.7 . At the bottom of the figure we have the "leaves" of the tree, which each represent one of the vertices of the network, and as we move up the tree, the leaves join together first in pairs and then in larger groups, until at the top of the tree all are joined together to form a single whole. Our algorithm in fact generates the dendrogram from the top, rather than the bottom, starting with a single connected network and splitting it repeatedly (p. 384) until we get to the level of single vertices. Individual intermediate configurations of the network during the run of the algorithm correspond to horizontal cuts through the dendrogram, as indicated by the dotted line in the figure. Each branch of the tree that intersects this dotted line represents one group of vertices, whose membership we can determine by following the branch down to its leaves at the bottom of the figure. Thus the dendrogram captures in a single diagram the configuration of groups in the network at every stage from start to finish of the algorithm.

This algorithm is somewhat different from previous ones, therefore, in that it doesn't give a single decomposition of a network into communities, but a selection of different possibilities, ranging from coarse divisions into just a few large communities (at the top of the dendrogram) to fine divisions into many small communities (at the bottom). It is up to the user to decide which of the many divisions represented is most useful for their purposes. One could in principle use a measure such as modularity to quantify the quality of the different divisions and select the one with the highest quality in this sense. This, however, somewhat misses the point. If high modularity is what you (p. 385) care about, then you are better off simply using a modularity maximization algorithm in the first place. It is more appropriate simply to think of this betweenness-based algorithm as producing a different kind of output, one that has its own advantages and disadvantages but that can undoubtedly tell us interesting things about network structure.

The betweenness-based algorithm is, unfortunately, quite slow. Aswe have said the calculation of betweenness for all edges takes time of order $O(n(m + n))$ and we have to perform this calculation before the removal of each of the m edges, so the entire algorithm takes time $O(mn(m + n))$, or $O(n^3)$ on a sparse graph with $m \# n$. This makes this algorithm one of the slower algorithms considered in this chapter. The algorithm gives quite good results in practice [138, 250], but has mostly been superseded by the faster modularity maximization methods of previous sections.

Nonetheless, the ability of the algorithm to return an entire dendrogram, rather than just a single division of a network, could be useful in some cases. The divisions represented in the dendrogram form a *hierarchical decomposition* in which the communities at one level are completely contained within the larger communities at all higher levels. There has been some interest in hierarchical structure in networks and hierarchical decompositions that might capture it. We look at another algorithm for hierarchical decomposition in Section 11.11.2 .

An interesting variation on the betweenness algorithm has been proposed by Radicchi *et al.* [276]. Their idea revolves around the same basic principle of identifying the edges between communities and removing them, but the measure used to perform the identification is different. Radicchi *et al.* observe that the edges that fall between otherwise poorly connected communities are unlikely to belong to short loops of edges, since doing so would require that there be two nearby edges joining the same groups —see Fig. 11.8 . Thus one way to identify the edges between communities would be to look for edges that belong to an unusually small number of short loops. Radicchi *et al.* found that loops of length three and four gave the best results. By repeatedly removing edges that belong to small numbers of such loops they were able to accurately uncover communities in a number of example networks.

An attractive feature of this method is its speed. The calculation of the number of short loops to which an edge belongs is a local calculation and can be performed for all edges in time that goes like the total size of the network. Thus, in the worst case, the running time of the algorithm will only go as $O(n^2)$ on a sparse graph, which is one order of system size faster than the betweenness-based algorithm and as fast as the earlier methods based on modularity maximization.

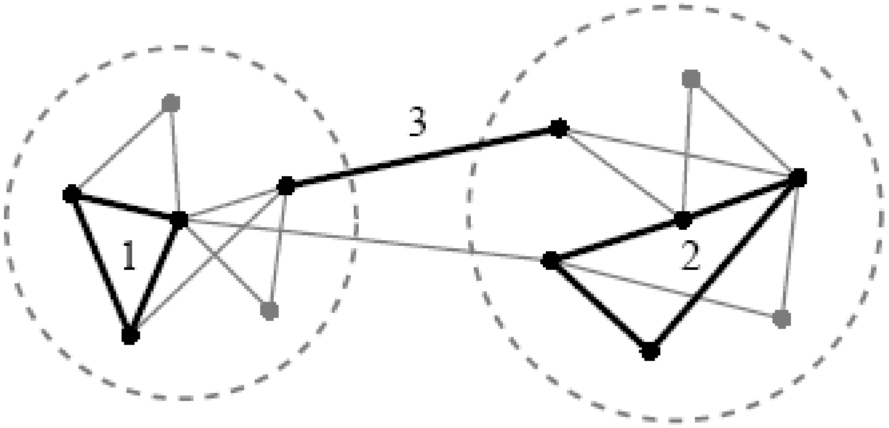


Figure 11.8: The algorithm of Radicchi *et al.* The algorithm of Radicchi *et al.* uses a different measure to identify between-group edges, looking for the edges that belong to the fewest short loops. In many networks, edges within groups typically belong to many short loops, such as the loops of length three and four labeled "1" and "2." But edges between groups, such as the edge labeled "3" here, often do not belong to such loops, because to do so would require there to be a return path along another between group edge, of which there are, by definition, few.

On the other hand, the algorithm of Radicchi *et al.* has the disadvantage that it only works on networks that have a significant number of short loops in the first place. This restricts the method primarily to social networks, which indeed have large numbers of short loops (see Section 7.9). Other types of network, such as technological and biological networks, tend to have smaller numbers of short loops, and hence there is little to distinguish between-group edges from within-group ones.

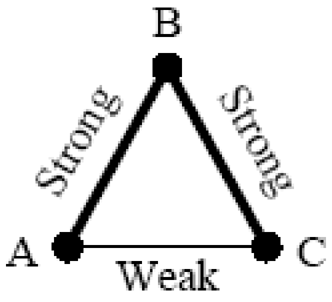
11.11.2 Hierarchical Clustering

The algorithms of the previous section differ somewhat from the other community detection algorithms in this chapter in that they produce a hierarchical decomposition of a network into a set of nested communities, visualized in the form of a dendrogram as in Fig. 11.7 , rather than just a single division into a unique set of communities. In this section we look at another algorithm that also produces a hierarchical decomposition, one of the oldest of community detection methods, the method of *hierarchical clustering*.[12]

Hierarchical clustering is not so much a single algorithm as an entire class (p. 387) of algorithms, with many variations and alternatives. Hierarchical clustering is an *agglomerative* technique in which we start with the individual vertices of a network and join them together to form groups. This contrasts with most of the other methods we have looked at for community detection and graph partitioning, which were *divisive* methods that took a complete network and split it apart. (One earlier algorithm, the greedy modularity maximization algorithm of Section 11.10 , was an agglomerative method.)

The basic idea behind hierarchical clustering is to define a measure of similarity or connection strength between vertices, based on the network structure, and then join together the closest or most similar vertices to form groups. We discussed measures of vertex similarity in networks at some length in Section 7.12 . Any of the measures of structural equivalence introduced there would be suitable as a starting point for hierarchical clustering, including cosine similarity (Section 7.12.1), correlation coefficients between rows of the adjacency matrix (Section 7.12.2), or the so-called Euclidean distance (Section 7.12.3). The regular equivalence measures of Section 7.12.4 might also be good choices, although the author is not aware of them having been used in this context.

That there are many choices for similarity measures is both a strength and a weakness of the hierarchical clustering method. It gives the method flexibility and allows it to be tailored to specific problems, but it also means that the method gives different answers depending on which measure we choose, and in many cases there is no way to know if one measure is more correct or will yield more useful information than another. Most often the choice of measure is determined more by experience or experiment than by argument from first principles.



If the connections (A,B) and (B,C) are strong but (A,C) is weak, should A and C be in the same group or not?

Once a similarity measure is chosen we calculate it for all pairs of vertices in the network. Then we want to group together those vertices having the highest similarities. This, however, leads to a further problem: the similarities can give conflicting messages about which vertices should be grouped. Suppose vertices A and B have high similarity, as do vertices B and C. One might A C therefore argue that A, B, and C should all be in a group together. But suppose that A and C have *low* similarity. Now we are left with a dilemma. Should A and C be in the same group or not?

The basic strategy adopted by the hierarchical clustering method is to start by joining together those pairs of vertices with the highest similarities, forming a group or groups of size two. For these there is no ambiguity, since each pair only has one similarity value. Then we further join together the groups that are most similar to form larger groups, and so on. When viewed in terms of agglomeration of groups like this, the problem above can be stated in a new and (p. 388) useful way. Our process requires for its operation a measure of the similarity between *groups*, so that we can join the most similar ones together. But what we actually have is a measure of similarity between individual vertices, so we need to combine these vertex similarities somehow to create similarities for the groups. If we can do this, then the rest of the algorithm is straightforward and the ambiguity is resolved.

There are three common ways of combining vertex similarities to give similarity scores for groups. They are called single-, complete-, and average-linkage clustering. Consider two groups of vertices, group 1 and group 2, containing n_1 and n_2 vertices respectively. There are then $n_1 n_2$ pairs of vertices such that one vertex is in group 1 and the other in group 2. In the *single-linkage clustering* method, the similarity between the two groups is defined to be the similarity of the *most similar* of these $n_1 n_2$ pairs of vertices. Thus if the values of the similarities of the vertex pairs range from 1 to 100, the similarity of the two groups is 100. This is a very lenient definition of similarity: only a single vertex pair need have high similarity for the groups themselves to be considered similar. (This is the origin of the name “single-linkage clustering”—similarity between groups is a function of the similarity between only the single most similar pair of vertices.)

At the other extreme, *complete-linkage clustering* defines the similarity between two groups to be the similarity of the *least similar* pair of vertices. If the similarities range from 1 to 100 then the similarity of the groups is 1. By contrast with single-linkage clustering this is a very stringent definition of group similarity: every single vertex pair must have high similarity for the groups to have high similarity (hence the name “complete-linkage clustering”).

In between these two extremes lies *average-linkage clustering*, in which the similarity of two groups is defined to be the mean similarity of all pairs of vertices. Average-linkage clustering is probably the most satisfactory choice of the three, being a moderate one—not extreme in either direction—and depending on the similarity of all vertex pairs and not just of the most or least similar pair. It is, however, relatively rarely used, for reasons that are not entirely clear.

The full hierarchical clustering method is as follows:

1. Choose a similarity measure and evaluate it for all vertex pairs.
2. Assign each vertex to a group of its own, consisting of just that one vertex. The initial similarities of the groups are simply the similarities of the vertices.
3. Find the pair of groups with the highest similarity and join them together into a single group.
4. Calculate the similarity between the new composite group and all others using one of the three methods above (single-, complete-, or average- (p. 389) linkage clustering).
5. Repeat from step 3 until all vertices have been joined into a single group.

In practice, the calculation of the new similarities is relatively straightforward. Let us consider the three cases separately. For single-linkage clustering the similarity of two groups is equal to the similarity of their most similar pair of vertices. In this case, when we join groups 1 and 2 together, the similarity of the composite group to another group 3, is the greater of the similarities of 1 with 3 and 2 with 3, which can be found in $O(1)$ time.

For complete-linkage clustering the similarity of the composite group is the smaller of the similarities of 1 with 3 and 2 with 3, which can also be found in $O(1)$ time.

The average-linkage case is only slightly more complicated. Suppose as before that the groups 1 and 2 that are to be joined have n_1 and n_2 vertices respectively. Then if the similarities of 1 with 3 and 2 with 3 were previously σ_{13} and σ_{23}, the similarity of the composite group with another group 3 is given by the weighted average (11.55)

$$\sigma_{12,3} = \frac{n_1\sigma_{13} + n_2\sigma_{23}}{n_1 + n_2}.$$

Again this can be calculated in O(1) time.

On each step of the algorithm we have to calculate similarities in this way for the composite group with every other group, of which there are O(n). Hence the recalculation of similarities will takeO(n) time on each step. Anaive search through the similarities to find the greatest one, on the other hand, takes time O(n^2), since there are O(n^2) pairs of groups to check, so this will be the most time-consuming step in the algorithm. We can speed things up, however, by storing the similarities in a binary heap (see Section 9.7 [13]), which allows us to add and remove entries in time O(log n) and find the greatest one in time O(1). This slows the recalculation of the similarities to O(n log n) but speeds the search for the largest to O(1).

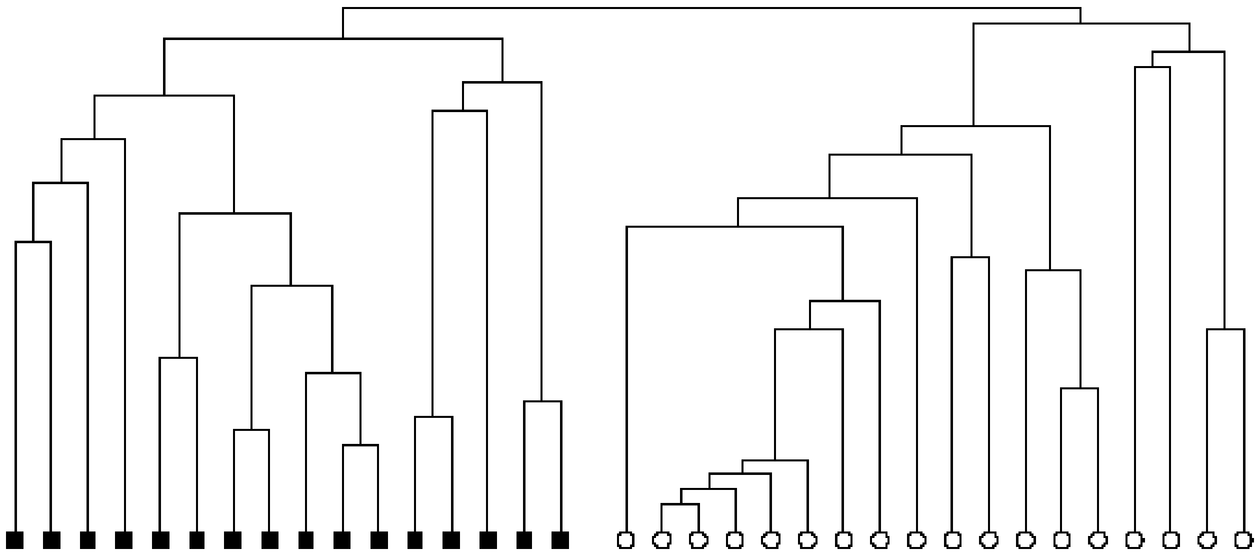


Figure 11.9: Partitioning of the karate club network by average linkage hierarchical clustering. This dendrogram is the result of applying the hierarchical clustering method described in the text to the karate club network of Fig. 11.4 , using cosine similarity as our measure of vertex similarity. The shapes of the nodes represent the two known factions in the network, as in the two previous figures.

Then the whole process of joining groups has to be repeated n - 1 times until all vertices have been joined into a single group. (To see this, simply consider that the number of groups goes down by one every time two groups are joined, so it takes n - 1 joins to go from n initial groups to just a single one (p. 390) at the end.) Thus the total running time of the algorithm is $O(n^3)$ in the naive implementation or $O(n^2 \log n)$ if we use a heap.[14]

And how well does it work in practice? The answer depends on which similarity measure one chooses and which linkage method, but a typical application, again to the karate club network, is shown in Fig. 11.9 . This figure shows what happens when we apply average-linkage clustering to the karate network using cosine similarity as our similarity measure. The figure shows the dendrogram that results from such a calculation and we see that there is a clear division of the dendrogram into two communities that correspond perfectly to the two known groups in the network.

Hierarchical clustering does not always work as well as this, however. In particular, though it is often good at picking out the cores of groups, where the vertices are strongly similar to one another, it tends to be less good at assigning peripheral vertices to appropriate groups. Such vertices may not be strongly similar to any others and so tend to get left out of the agglomerative (p. 391) clustering process until the very end. A common result of hierarchical clustering is therefore a set of tightly knit cores surrounded by a loose collection of single vertices or smaller groups. Such a result may nonetheless contain a lot of valuable information about the underlying network structure.

Many other methods have been proposed for community detection and there is not room in this book to describe them all. For the reader interested in pursuing the topic further the review articles by Fortunato [124] and Schaeffer [291] provide useful overviews.

Problems

11.1 Show that the inverse of a symmetric matrix $\mathbf{M}$ is given by $\mathbf{M}^{-1} = \mathbf{UDU}^T$ where $\mathbf{U}$ is the orthogonal matrix whose columns are the normalized eigenvectors of $\mathbf{M}$ and $\mathbf{D}$ is the diagonal matrix whose elements are the reciprocals of the eigenvalues of $\mathbf{M}$. Hence argue that the time complexity of the best algorithm for inverting a symmetric matrix can be no worse than the time complexity of finding all of its eigenvalues and eigenvectors. (In fact they are the same—both are $O(n)$ for an $n \times n$ matrix.)

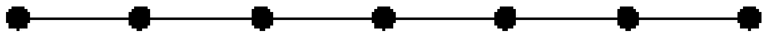
11.2 Consider a general $n \times n$ matrix $\mathbf{M}$ with eigenvalues μ_i where $i = 1 \dots n$.

1. a) Show that the matrix $\mathbf{M} - a\mathbf{I}$ has the same eigenvectors as $\mathbf{M}$ and eigenvalues μ_i μ a.
2. b) Suppose that the matrix's two eigenvalues of largest magnitude are both positive. Show that the time taken to find the leading eigenvector of the matrix using the

power method of Section 11.1 can be improved by performing the calculation instead for the matrix M − *a* I, where *a* is positive.

3. c) What stops us from increasing the constant *a* arbitrarily until the calculation takes no time at all?

11.3 Consider a “line graph” consisting of *n* vertices in a line like this:



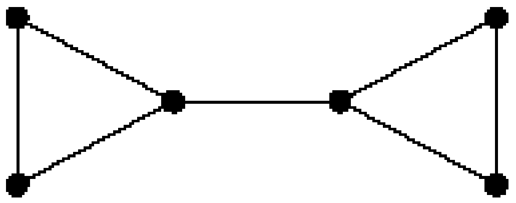
1. a) Show that if we divide the network into two parts by cutting any single edge, such that one part has *r* vertices and the other has *n*-*r*, the modularity, Eq. (7.76) , takes the value

$$Q = \frac{3 - 4n + 4rn - 4r^2}{2(n-1)^2}.$$

• (p. 392)

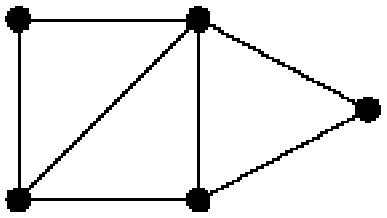
2. b) Hence show that when *n* is even the optimal such division, in terms of modularity, is the division that splits the network exactly down the middle.

11.4 Using your favorite numerical software for finding eigenvectors of matrices, construct the Laplacian and the modularity matrix for this small network:



1. a) Find the eigenvector of the Laplacian corresponding to the second smallest eigenvalue and hence perform a spectral bisection of the network into two equally sized parts.
2. b) Find the eigenvector of the modularity matrix corresponding to the largest eigenvalue and hence divide the network into two communities.You should find that the division of the network generated by the two methods is, in this case, the same.

11.5 Consider this small network with five vertices:



1. a) Calculate the cosine similarity for each of the

$\binom{5}{2} = 10$

pairs of vertices.

2. b) Using the values of the ten similarities construct the dendrogram for the single-linkage hierarchical clustering of the network according to cosine similarity.

(p. 393) (p. 394)

Notes:

(1) Technically the power method finds the eigenvector corresponding to the eigenvalue of largest absolute magnitude and hence the method would fail to find the eigenvector we want if the largest absolute magnitude belongs to a negative eigenvalue. For a matrix with all elements non-negative, however, such as the adjacency matrix, it turns out this can never happen. Here is a proof of this result for an undirected network where A is symmetric; the general case is covered, for example, in Ref. [217]. Let μ be the most negative eigenvalue of a real symmetric matrix A and let w be the corresponding eigenvector, with elements w_i. Then, given that

$w^T w = \Sigma_i w_i^2 > 0$

,

$$|\mu|\mathbf{w}^T\mathbf{w} = |\mu\mathbf{w}^T\mathbf{w}| = |\mathbf{w}^T\mathbf{A}\mathbf{w}| = \left|\sum_{ij} A_{ij}w_iw_j\right| \le \sum_{ij} |A_{ij}w_iw_j| = \sum_{ij} A_{ij}|w_i||w_j| = \mathbf{x}^T\mathbf{A}\mathbf{x},$$

, where x is the vector with components $|w_i|$. The inequality here follows from the so-called triangle inequality $|a+b| \le |a|+|b|$, which is true for all real numbers *a*, *b*. Rearranging, we now find that

$$|\mu| \le \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{w}^T\mathbf{w}} = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}},$$

, where we have made use of

$x^T x = \Sigma_i |w_i| = w^T w$

. Now we write x as a linear combination of the normalized eigenvectors v $_i$ of A thus: x = σ $_i$ c_i v i, where the c_i are real coefficients whose exact values are not important for this proof. Then, if κi is the eigenvalue corresponding to v i and κ_1 is the most positive eigenvalue, we have

$$\frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} = \frac{\sum_j c_j\mathbf{v}_j^T\mathbf{A}\sum_i c_i\mathbf{v}_i}{\sum_j c_j\mathbf{v}_j^T\sum_i c_i\mathbf{v}_i} = \frac{\sum_j c_j\mathbf{v}_j^T\sum_i c_i\kappa_i\mathbf{v}_i}{\sum_j c_j\mathbf{v}_j^T\sum_i c_i\mathbf{v}_i} = \frac{\sum_i c_i^2\kappa_i}{\sum_i c_i^2} \le \frac{\sum_i c_i^2\kappa_1}{\sum_i c_i^2} = \kappa_1,$$

, where we have made use of the orthogonality property

$v_j^T v_i = \delta ij$

. (The inequality is an exact equality if and only if x is an eigenvector with eigenvalue κ_1.) Putting these results together, we find that $|\mu| \leq \kappa_1$ and hence the most negative eigenvalue never has a magnitude greater than that of the most positive eigenvalue (although if we are unlucky the two magnitudes could be equal). The result proved here is one part of the *Perron-Frobenius theorem*. The other part, that the leading eigenvector has all elements non-negative, is proved in the following footnote.

(2) This result, like that in footnote 1, is a part of the Perron–Frobenius theorem. To prove it—at least for the case of symmetric A—let κ_1 be the most positive eigenvalue of A and let v be a corresponding eigenvector. (We will allow, for the moment, the possibility that there is more than one eigenvector with eigenvalue κ_1, though we show below that in fact this cannot happen in a connected network.) Note that $\kappa 1 \geq 0$ since the sum of the eigenvalues of A is given by Tr A ≥ 0, and hence at least one eigenvalue must be non-negative. Then, given that

$v^T v = \Sigma_i v_i^2 > 0$

and all elements of A are non-negative, we have

$$\kappa_1 \mathbf{v}^T\mathbf{v} = |\kappa_1 \mathbf{v}^T\mathbf{v}| = |\mathbf{v}^T\mathbf{A}\mathbf{v}| = \left|\sum_{ij} A_{ij} v_i v_j\right| \leq \sum_{ij} |A_{ij} v_i v_j| = \sum_{ij} A_{ij} |v_i||v_j| = \mathbf{x}^T\mathbf{A}\mathbf{x},$$

, where x is the vector with elements $|v_i|$. Rearranging this result, we find

$$\kappa_1 \leq \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{v}^T\mathbf{v}} = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}},$$

, where we have made use of $x^T x = \sigma i |v^i|^2 = v^T v$. As demonstrated in footnote 1 on page 346, for any vector x we have

$$\frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \leq \kappa_1,$$

, with the equality being achieved only when x is an eigenvector corresponding to eigenvalue κ_1. The only way to reconcile the two inequalities above is if they are in fact equalities in this case, implying that x must indeed be an eigenvector with eigenvalue κ^1. But x has all elements nonnegative, and hence there exists an eigenvector with eigenvalue κ^1 and all elements non-negative.

It is still possible that there might be more than one eigenvector with eigenvalue κ^1, and that one of the others might have negative elements. This, however, we can rule out as follows. Recall that eigenvectors with same eigenvalue can always be chosen orthogonal, and any eigenvector v that is orthogonal to the eigenvector with all elements non-negative would have to have both positive and negative elements in order that the product of the two vectors equal zero. Thus there is only one eigenvector with all elements non-negative.

Then, for eigenvector v, by the results above, the vector x with elements $|v_i|$ is necessarily equal to the unique eigenvector with all elements non-negative. Thus if v_i is one of the positive elements of v then $v_i = x_i$ and

$$\sum_j A_{ij}|v_j| = \sum_j A_{ij}x_j = \kappa_1 x_i = \kappa_1 v_i = \sum_j A_{ij}v_j,$$

, or, equivalently, $\sigma_j A_{ij}(|v_j| - v_j) = 0$. But $|v_j| - v_j \geq 0$ so this last result can only be true if for all j we have either $A_{ij} = 0$ or $v_j - |v_j| = 0$, meaning that $v_j = |v_j| \geq 0$. Thus if $v_i > 0$ then $v_j > 0$ whenever $A_{ij} \neq 0$. In network terms, if $v_i > 0$ then $v_j > 0$ for every neighbor of i. But then we can start at i and work outwards, moving from neighbor to neighbor and so demonstrate that $v_j > 0$ for every vertex and hence v = x and the leading eigenvector is unique.

The only exception to this last result is when the network has more than component, so that some vertices are not reachable from an initial vertex i. In that case, it is possible for the elements of the leading eigenvector corresponding to vertices in different components to have different signs. This, however, causes no problems for any of the results presented here.

(3) In fact, this estimate usually errs on the pessimistic side, since the spacing of the highest eigenvalues tends to be wider than the mean spacing, so that in practice the algorithm may be faster than the estimate would suggest.

(4) If we wish to be more sophisticated, we can note that it is sufficient to shift the eigenvalues by any amount greater than or equal to λ_n. Anderson and Morley [18] have shown that $\lambda_n \leq 2k_{\max}$ where $k_{\max}$ is the largest degree in the network, which we can find in time $O(n)$, considerably faster than we can find λ_n itself. Thus a quicker way to find the smallest eigenvalue would be to find the largest eigenvalue of $2k_{\max}\mathrm{I} - \mathrm{L}$.

(5) Community detection is sometimes also called "clustering," although we largely avoid this term to prevent confusion with the other, and quite different, use of the word clustering introduced in Section 7.9 .

(6) The problem is somewhat similar to the minimum cut problem of Section 6.12 , but we are now searching for the minimum cut over all possible bisections of a network, rather than just between a given pair of vertices.

(7) Technically, this statement has not actually been proved. Its truth hinges on the assumption that two fundamental classes of computational problem, called P and NP, are not the same. Although this assumption is universally believed to be true—the world would pretty much fall apart if it weren't—no one has yet proved it, nor even has any idea about where to start. Readers interested in the fascinating branch of theoretical computer science that deals with problems of this kind are encouraged to look, for example, at the book by Moore and Mertens [227].

(8) Some readers may be familiar with Kernighan's name. He was one of the authors of the original book describing the C programming language [172]. "Kernighan" is pronounced "Kernihan"—the "g" is silent.

(9) One might imagine that an equivalent procedure would be to go on swapping vertex pairs until no swap can be found that decreases the cut size. This, however, turns out to be wrong. It is perfectly possible for the cut size to decrease for a few steps of the algorithm, then increase, then decrease again. If we halt the algorithm the first time we see the cut size increasing, we run the risk of missing a later state with smaller cut size. Thus the correct algorithm is the one described here, with two separate processes, one of vertex swapping, and one of checking the states so generated to see which is optimal.

(10) If the network is stored in adjacency matrix form then the total run time can be improved further to $O(n^2)$, although for the common case of a sparse network this makes relatively little difference, and the adjacency matrix is costly in terms of memory space.

(11) Note, however, that the vertex moving algorithm takes time $O(n^2)$ for each round of the algorithm, but we have not calculated, and do not in fact know, how many rounds are needed in general. As with the Kernighan-Lin algorithm, it is reasonable to suppose that the number of rounds needed might increase, at least slowly, with network size, which would make the time complexity of the vertex moving algorithm poorer than that of the spectral algorithm.

(12) The word "clustering" as used here just refers to community detection. We have mostly stayed away from using this word in this chapter, to avoid confusion with the other use of the word clustering introduced in Section 7.9 (see footnote 5 on page 354), but the name "hierarchical clustering" is a well established and traditional one, and we use it here in deference to convention.

(13) The heap must be modified slightly from the one described in Section 9.7 . First, the partial ordering must be inverted so that the largest, not the smallest, element of the heap is at its root. Second, we need to be able to remove arbitrary items from the heap, not just the root item, which we do by deleting the relevant item and then moving the last item in the heap to fill the vacated space. Then we have to sift the moved item both up *and* down the heap, since it might be either too large or too small for the position in which it finds itself.

(14) For the special case of single-linkage clustering, there is a slightly faster way to implement the algorithm that makes use of a so-called union/find technique and runs in time $O(n^2)$. In practice the performance difference is not very large but the union/find method is considerably simpler to program. It is perhaps for this reason that single-linkage is more often used than complete- or average-linkage clustering.