moment often tends to a modest constant value as the network becomes large. But for networks with highly skewed degree distributions $\langle k^2 \rangle$ can become very large and in the case of a power-law degree distribution with exponent $\alpha < 3$ it formally diverges (see Section 8.4.2) and with it so does the expected running time of our algorithm.

More realistically, if the network is a simple graph with no multiedges, then the maximum allowed degree is $k = n$ and the degree distribution is cut off, which means that the second moment scales at worst as $n^{3-\alpha}$ (Eq. (8.22)) while the first moment remains constant. This in turn implies that the running time of our clustering coefficient algorithm on a scale-free network would go as $n \times n^{3-\alpha} \times n^{3-\alpha} = n^{7-2\alpha}$. For values of α in the typical range of $2 \le \alpha \le 3$ (Table 8.1), this gives running times that vary from a minimum of $O(n)$ for $\alpha = 3$ to a maximum of $O(n^3)$ for $\alpha = 2$. For the lower values of α this makes the calculation of the clustering coefficient quite arduous, taking a time that increases sharply as the network gets larger.

So can we improve on this algorithm? There are various possibilities. Most of the work of the algorithm is in the "find" operation to determine whether there is an edge between a given pair of vertices, and the algorithm will be considerably faster if we can perform this operation more efficiently. One simple (though memory-inefficient) method is to make use of the hybrid matrix/list data structure of Section 9.6, which can perform the find operation in constant time.[5] Even in this case, however, the number of find operations that must be performed is still equal to the number of connected triples in the network, which means the running time is given by Eq. (10.4), and hence still formally diverges on a network with a power-law degree distribution. On a simple graph for which the power law is cut off at $k = n$, it will go as $n^{4-\alpha}$, which ranges from $O(n)$ to $O(n^2)$ for values of α in the interesting range $2 \le \alpha \le 3$. This is better than our earlier algorithm, but still relatively poor for the lower values of α.

These difficulties are specific to the case of scale-free networks. In other cases there is usually no problem calculating the clustering coefficient quickly. Some alternative algorithms have been proposed for calculating approximate values of the clustering coefficient rapidly, such as the algorithm of Schank and Wagner [292], and these may be worth considering if you need to perform calculations on very large networks.

[5]Other possible ways to improve the algorithm are to use the adjacency tree structure of Section 9.5, or to use the adjacency list but always test for the presence of an edge between two vertices by searching the neighbors of the lower-degree vertex to see if the higher is among them (rather than the algorithm described above, which chooses which one to search at random).

10.3 SHORTEST PATHS AND BREADTH-FIRST SEARCH

We now move on to some more complex algorithms, algorithms for calculating mostly non-local quantities on the networks, such as shortest paths between vertices. The study of each of these algorithms has three parts. Two are, as before, the description of the algorithm and the analysis of its running time. But now we also include a proof that the algorithm described performs the calculation it claims to. For the previous algorithms in this chapter such proofs were unnecessary; the algorithms were direct implementations of the equations defining the quantities calculated. As we move onto more complex algorithms, however, it will become much less obvious why those algorithms give the results they do, and to gain a full understanding we will need to examine their working in some detail.

The first algorithm we look at is the standard algorithm for finding shortest distances in a network, which is called *breadth-first search*.[6] A single run of the breadth-first search algorithm finds the shortest (geodesic) distance from a single source vertex s to *every other* vertex in the same component of the network as s. In some cases we want to know only the shortest distance between a single pair of vertices s, t, but there is no procedure known for calculating such a distance that is faster in the worst case than calculating the distances from s to every other vertex using breadth-first search and then throwing away all of the results except for the one we want.[7]

With only minor modifications, as we will describe, breadth-first search can also find the geodesic path one must take to realize each shortest distance and if there is more than one geodesic path, it can find all such paths. It works also on both directed and undirected networks, although our description will focus on the undirected case.

10.3.1 DESCRIPTION OF THE ALGORITHM

Breadth-first search finds the shortest distance from a given starting vertex s to every other vertex in the same component as s. The basic principle behind the algorithm is illustrated in Fig. 10.1. Initially we know only that s has distance 0 from itself and the distances to all other vertices are unknown. Now we find all the neighbors of s, which by definition have distance 1 from s. Then we find all the neighbors of *those* vertices. Excluding those we have already visited, these

[6]In physics, breadth-first search is sometimes called the "burning algorithm."

[7]In many cases we may find the result we want before calculating all distances, in which case we can save ourselves the effort of calculating the rest, but in the worst case we will have to calculate them all. See Section 10.3.4 for more discussion.

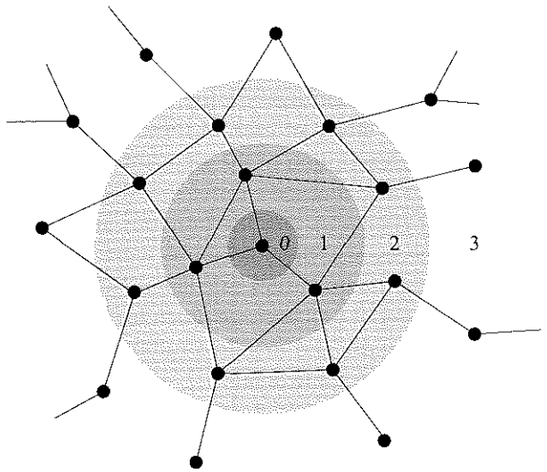


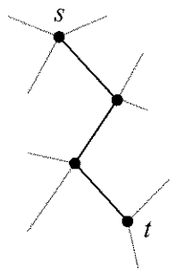
Figure 10.1: Breadth-first search. A breadth-first search starts at a given vertex, which by definition has distance 0 from itself, and grows outward in layers or waves. The vertices in the first wave, which are the immediate neighbors of the starting vertex, have distance 1. The neighbors of those neighbors have distance 2, and so forth.

vertices must have distance 2. And *their* neighbors, excluding those we have already visited have distance 3, and so on. On every iteration, we grow the set of vertices visited by one step.

This is the basic idea of breadth-first search. Now let us go over it more carefully to see how it works in practice and show that it really does find correct geodesic distances. We begin by noting the following fact:

> Every vertex whose shortest distance from s is d has a network neighbor whose shortest distance from s is $d-1$.

This follows since if the shortest path from s to a vertex t is of length d then the penultimate vertex along that path, which is a neighbor of t, can, by definition, be reached in $d-1$ steps and hence cannot have shortest distance greater than $d-1$. It also cannot have shortest distance less than $d-1$ because it if did there would be a path to t of length less than d.



A network path from s to t of length d (where $d = 3$ in this case) necessarily includes a path of length $d-1$ (i.e., 2) from s to an immediate neighbor of t.

Now suppose that we already know the distance to every vertex on the network that is d steps or less from our source vertex s. For example, we might know all the distances to vertices at distance 2 or less from the central vertex in Fig. 10.1. For every neighbor of one of the vertices at distance d there exists a path of length $d+1$ to that neighbor: we can get to the vertex at distance d along a path of length d and then we take one more step to its neighbor. Thus every such neighbor is *at most* $d+1$ steps from s, but it could be less than $d+1$ from s if there is another shorter path through the network. However, we already know whether there is a shorter path to any particular vertex, since by

hypothesis we know the distance to every vertex d steps or less from s.

Consider the set of all vertices that are neighbors of vertices at distance d but that are not already known to have distance d or less from s. We can say immediately that (1) all neighbors in this set have distance $d+1$ from s, and (2) that there are no other vertices at distance $d+1$. The latter follows from the property cited above: all vertices at distance $d+1$ must be neighbors of vertices at distance d. Thus we have found the set of vertices at distance $d+1$, and hence we now know the distances to all vertices that are $d+1$ or less from s.

Now we just repeat the process. On each round of the algorithm we find all the vertices one step further out from s than on the last round. The algorithm continues until we reach a point at all the neighbors of vertices at distance d are found already to have known distances of d or less. This implies that there are no vertices of distance $d+1$ and hence, by the property above, no vertices of any greater distance either, and so we must have found every vertex in the component containing s.

As a corollary of the process of finding distance, breadth-first search thus also finds the component to which vertex s belongs, and indeed breadth-first search is the algorithm of choice for finding components in networks.

10.3.2 A NAIVE IMPLEMENTATION

Let us now consider how we would implement breadth-first search on our computer. The simplest approach (but not, as we will see, the best) would go something like this. We create an array of n elements to store the distance of each vertex from the source vertex s, and initially set the distance of vertex s from itself to be zero while all other vertices have unknown distance from s. Unknown distances could be indicated, for instance, by setting the corresponding element of the array to -1, or some similar value that could never occur in reality.

We also create a distance variable d to keep track of where we are in the breadth-first search process and set its value initially to zero. Then we do the following:

1. Find all vertices that are distance d from s, by going through the distance array, element by element.
2. Find all the neighbors of those vertices and check each one to see if its distance from s is unknown (denoted, for example, by an entry -1 in the distance array).
3. If the number of neighbors with unknown distances is zero, the algorithm is over. Otherwise, if the number of neighbors with unknown dis-

tances is non-zero, set the distance of each of those neighbors to $d + 1$.

4. Increase the value of d by 1.
5. Repeat from step 1.

When the algorithm is finished we are left with an array that contains the distances to every vertex in the component of the network that contains s (and every vertex in every other component has unknown distance).

How long does this algorithm take? First of all we have to set up the distance array, which has one element for each vertex. We spend a constant amount of time setting up each element, so overall we spend $O(n)$ time setting up the distance array.

For the algorithm proper, on each iteration we go through all n vertices looking for those with distance d. Most will not have distance d in which case we pass over them, spending only $O(1)$ time on each. Thus there is a basic cost of $O(n)$ time for each iteration. The total number of iterations we will for the moment call r, and overall we thus spend $O(rn)$ time on this part of the algorithm, in the worst case.

However, when we *do* come across a vertex with distance d, we must pause at that vertex and spend an additional amount of time checking each of its neighbors to see if their distances are unknown and assigning them distance $d + 1$ if they are. If we assume that the network is stored in adjacency list format (see Section 9.4) then we can go through the neighbors of a vertex in $O(m/n)$ on average, and during the whole course of the algorithm we pause like this at each vertex exactly once so that the total extra time we spend on checking neighbors of vertices is $n \times O(m/n) = O(m)$.

Thus the total running time of the algorithm, including set-up, is $O(n + rn + m)$.

And what is the value of the parameter r? The value of r is the maximum distance from our source vertex s to any other vertex. In the worst case, this distance is equal to the diameter of the network (Section 6.10.1) and the worst-case diameter is simply n, which is realized when the network is just a chain of n vertices strung one after another in a line. Thus in the worst case our algorithm will have running time $O(m + n^2)$ (where we have dropped the first n because we are keeping only the leading-order terms).

This is very pessimistic, however. As discussed in Sections 8.2 and 12.7 the diameter of most networks increases only as $\log n$, in which case our algorithm would run in time $O(m + n \log n)$ to leading order. This may be a moot point, however, since we can do significantly better than this if we use a little cunning in the implementation of our algorithm.

10.3.3 A BETTER IMPLEMENTATION

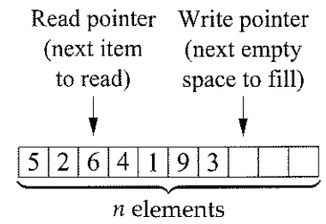
The time-consuming part of the implementation described in the previous section is step 1, in which we go through the list of distances to find vertices that are distance d from the starting vertex s. Since this operation involves checking the distances of all n vertices, only a small fraction of which will be at distance d, it wastes a lot of time. Observe, however, that in each wave of the breadth-first search process we find and label all vertices with a given distance $d + 1$. If we could store a list of these vertices, then on the next wave we wouldn't have to search through the whole network for vertices at distance $d + 1$; we could just use our list.

The most common implementation of this idea makes use of a *first-in/first-out buffer* or *queue*, which is nothing more than an array of (in this case) n elements that store a list of labels of vertices. On each sweep of the algorithm, we read the vertices with distance d from the list, we use these to find the vertices with distance $d + 1$, add those vertices with distance $d + 1$ to the list, and repeat.

To do this in practice, we fill up the queue array starting from the beginning. We keep a pointer, called the write pointer, which is a simple integer variable whose value indicates the next empty location at the end of the queue that has not been used yet. When we want to add an item to the queue, we store it in the element of the array pointed to by the write pointer and then increase the pointer by one to point to the next empty location.

At the same time we also keep another pointer, the read pointer, which points to the next item in the list that is to be read by our algorithm. Each item is read only once and once it is read the read pointer is increased by one to point to the next unread item.

Here is a sketch of the organization of the queue:



Our breadth-first search algorithm now uses two arrays of n elements, one for the queue and one for the distances from s to each other vertex. The algorithm is as follows.

1. Place the label of the source vertex s in the first element of the queue, set the read pointer to point to it, and set the write pointer to point to the second element, which is the first empty one. In the distance array, record the distance of vertex s from itself as being zero and the distances to all other vertices as "unknown" (for instance, by setting the corresponding elements of the distance array to -1, or some similar impossible value).
2. If the read and write pointers are pointing to the same element of the queue array then the algorithm is finished. Otherwise, read a vertex label from the element pointed to by the read pointer and increase that pointer by one.
3. Find the distance d for that vertex by looking in the distance array.
4. Go through each neighboring vertex in turn and look up its distance in the distance array as well. If it has a known distance, leave it alone. If it has an unknown distance, assign it distance $d+1$, store its label in the queue array in the element pointed to by the write pointer, and increase the write pointer by one.
5. Repeat from step 2.

Note the test applied in step 2: if the read pointer points to the same element as the write pointer, then there is no vertex to be read from the queue (since the write pointer always points to an empty element). Thus this test tells us when there are no further vertices waiting to have their neighbors investigated.

Note also that this algorithm reads all the vertices with distance d from the queue array one after another and uses them to find all the vertices with distance $d+1$. Thus all vertices with the same distance appear one after another in the queue array, with the vertices of distance $d+1$ immediately after those of distance d. Furthermore, each vertex appears in the queue array at most once. A vertex may of course be a neighbor of more than one other, but a vertex is assigned a distance and put in the queue only on the first occasion on which it is encountered. If it is encountered again, its distance is known rather than unknown, and hence it is not again added to the queue. Of course, a vertex may not appear in the queue array at all if it is never reached by the breadth-first search process, i.e., if it belongs to a different component from s.

Thus the queue does exactly what we wanted it to: it stores all vertices with a specified distance for us so that we have the list handy on the next sweep of the algorithm. This spares us from having to search through the network for them and so saves us a lot of time. In all other respects the algorithm works exactly as in the simple implementation of Section 10.3.2 and gives the same answers.

How long does this implementation of the algorithm take to run? Again there is an initial time of $O(n)$ to set up the distance array (see Section 10.3.2).

Then, for each element in the queue, which means for each of the vertices in the same component as s, we do the following operations: we run through its neighbors, of which there are $O(m/n)$ on average, and either calculate their distance and add them to the queue, or do nothing if their distance is already known. Either way the operations take $O(1)$ time. Thus for each vertex in the component, of which there are in the worst case n, we spend time $O(m/n)$ and hence we require overall at most a time $n \times O(m/n) = O(m)$ to complete the algorithm for all n vertices.

Thus, including the time to set up the distance array, the whole algorithm takes time $O(m+n)$, which is better than the $O(m+n\log n)$ of the naive implementation (Section 10.3.2). For the common case of a sparse network with $m \propto n$, $O(m+n)$ is equivalent to $O(n)$ and our algorithm runs in time proportional to the number of vertices.[8] This seems just about optimal, since the algorithm is calculating the distance of all n vertices from the source vertex s. Thus it is assigning n numbers to the n elements of the distance array, which in the best possible case must take $O(n)$ time.

On a sparse network, therefore, the breadth-first search algorithm does as well as we can hope for in finding the distances from a single vertex to all others, and indeed it is the fastest known algorithm for performing this operation.

10.3.4 VARIANTS OF BREADTH-FIRST SEARCH

There are a number of minor variants of breadth-first search that merit a mention. First, one might wish to calculate the shortest distance between only a single pair of vertices s and t, rather than between s and all others. As mentioned in Section 10.3 there is no known way to do this faster than using breadth-first search. We can, however, improve the running time slightly by the obvious tactic of stopping the algorithm as soon as the distance to the target vertex t has been found. There is no point in continuing to calculate distances to the remaining vertices once we have the answer we want. In the worst case, the calculation still takes $O(m+n)$ time since, after all, our particular target vertex t might turn out to be the last one the algorithm finds. If we are lucky, however, and encounter the target early then the running time might be considerably shorter.

Conversely, we sometimes want to calculate the shortest distance between every pair of vertices in an entire network, which we can do by performing a breadth-first search starting at each vertex in the network in turn. The total

[8] On the other hand, for a dense network where $m \propto n^2$, we have a running time of $O(n^2)$.