Chapter 10

Mining Social-Network Graphs

There is much information to be gained by analyzing the large-scale data that is derived from social networks. The best-known example of a social network is the "friends" relation found on sites like Facebook. However, as we shall see there are many other sources of data that connect people or other entities.

In this chapter, we shall study techniques for analyzing such networks. An important question about a social network is to identify "communities," that is, subsets of the nodes (people or other entities that form the network) with unusually strong connections. Some of the techniques used to identify communities are similar to the clustering algorithms we discussed in Chapter 7. However, communities almost never partition the set of nodes in a network. Rather, communities usually overlap. For example, you may belong to several communities of friends or classmates. The people from one community tend to know each other, but people from two different communities rarely know each other. You would not want to be assigned to only one of the communities, nor would it make sense to cluster all the people from all your communities into one cluster.

Also in this chapter we explore efficient algorithms for discovering other properties of graphs. We look at "simrank," a way to discover similarities among nodes of a graph. We explore triangle counting as a way to measure the connectedness of a community. We give efficient algorithms for exact and approximate measurement of the neighborhood sizes of nodes in a graph. Finally, we look at efficient algorithms for computing the transitive closure.

10.1 Social Networks as Graphs

We begin our discussion of social networks by introducing a graph model. Not every graph is a suitable representation of what we intuitively regard as a social

network. We therefore discuss the idea of "locality," the property of social networks that says nodes and edges of the graph tend to cluster in communities. This section also looks at some of the kinds of social networks that occur in practice.

10.1.1 What is a Social Network?

When we think of a social network, we think of Facebook, Google+, or another website that is called a "social network," and indeed this kind of network is representative of the broader class of networks called "social." The essential characteristics of a social network are:

1. There is a collection of entities that participate in the network. Typically, these entities are people, but they could be something else entirely. We shall discuss some other examples in Section 10.1.3.

2. There is at least one relationship between entities of the network. On Facebook or its ilk, this relationship is called *friends*. Sometimes the relationship is all-or-nothing; two people are either friends or they are not. However, in other examples of social networks, the relationship has a degree. This degree could be discrete; e.g., friends, family, acquaintances, or none as in Google+. It could be a real number; an example would be the fraction of the average day that two people spend talking to each other.

3. There is an assumption of nonrandomness or locality. This condition is the hardest to formalize, but the intuition is that relationships tend to cluster. That is, if entity A is related to both B and C, then there is a higher probability than average that B and C are related.

10.1.2 Social Networks as Graphs

Social networks are naturally modeled as undirected graphs. The entities are the nodes, and an edge connects two nodes if the nodes are related by the relationship that characterizes the network. If there is a degree associated with the relationship, this degree is represented by labeling the edges.

Example 10.1: Figure 10.1 is an example of a tiny social network. The entities are the nodes A through G. The relationship, which we might think of as "friends," is represented by the edges. For instance, B is friends with A, C, and D.

Is this graph really typical of a social network, in the sense that it exhibits locality of relationships? First, note that the graph has nine edges out of the $\binom{7}{2} = 21$ pairs of nodes that could have had an edge between them. Suppose X, Y, and Z are nodes of Fig. 10.1, with edges between X and Y and also between X and Z. What would we expect the probability of an edge between

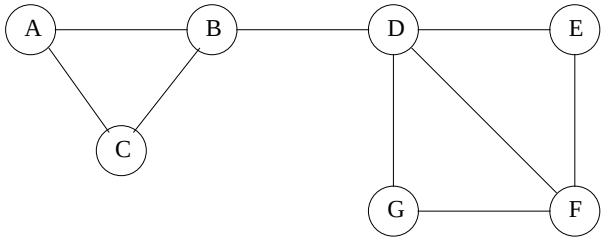

Figure 10.1: Example of a small social network

Y and Z to be? If the graph were large, that probability would be very close to the fraction of the pairs of nodes that have edges between them, i.e., 9/21 = .429 in this case. However, because the graph is small, there is a noticeable difference between the true probability and the ratio of the number of edges to the number of pairs of nodes. Since we already know there are edges (X, Y) and (X, Z), there are only seven edges remaining. Those seven edges could run between any of the 19 remaining pairs of nodes. Thus, the probability of an edge (Y, Z) is 7/19 = .368.

Now, we must compute the probability that the edge (Y, Z) exists in Fig. 10.1, given that edges (X, Y) and (X, Z) exist. What we shall actually count is pairs of nodes that could be Y and Z, without worrying about which node is Y and which is Z. If X is A, then Y and Z must be B and C, in some order. Since the edge (B, C) exists, A contributes one positive example (where the edge does exist) and no negative examples (where the edge is absent). The cases where X is C, E, or G are essentially the same. In each case, X has only two neighbors, and the edge between the neighbors exists. Thus, we have seen four positive examples and zero negative examples so far.

Now, consider $X = F$. F has three neighbors, D, E, and G. There are edges between two of the three pairs of neighbors, but no edge between G and E. Thus, we see two more positive examples and we see our first negative example. If $X = B$, there are again three neighbors, but only one pair of neighbors, A and C, has an edge. Thus, we have two more negative examples, and one positive example, for a total of seven positive and three negative. Finally, when $X = D$, there are four neighbors. Of the six pairs of neighbors, only two have edges between them.

Thus, the total number of positive examples is nine and the total number of negative examples is seven. We see that in Fig. 10.1, the fraction of times the third edge exists is thus 9/16 = .563. This fraction is considerably greater than the .368 expected value for that fraction. We conclude that Fig. 10.1 does indeed exhibit the locality expected in a social network. □

10.1.3 Varieties of Social Networks

There are many examples of social networks other than "friends" networks. Here, let us enumerate some of the other examples of networks that also exhibit locality of relationships.

Telephone Networks

Here the nodes represent phone numbers, which are really individuals. There is an edge between two nodes if a call has been placed between those phones in some fixed period of time, such as last month, or "ever." The edges could be weighted by the number of calls made between these phones during the period. Communities in a telephone network will form from groups of people that communicate frequently: groups of friends, members of a club, or people working at the same company, for example.

Email Networks

The nodes represent email addresses, which are again individuals. An edge represents the fact that there was at least one email in at least one direction between the two addresses. Alternatively, we may only place an edge if there were emails in both directions. In that way, we avoid viewing spammers as "friends" with all their victims. Another approach is to label edges as *weak* or *strong*. Strong edges represent communication in both directions, while weak edges indicate that the communication was in one direction only. The communities seen in email networks come from the same sorts of groupings we mentioned in connection with telephone networks. A similar sort of network involves people who text other people through their cell phones.

Collaboration Networks

Nodes represent individuals who have published research papers. There is an edge between two individuals who published one or more papers jointly. Optionally, we can label edges by the number of joint publications. The communities in this network are authors working on a particular topic.

An alternative view of the same data is as a graph in which the nodes are papers. Two papers are connected by an edge if they have at least one author in common. Now, we form communities that are collections of papers on the same topic.

There are several other kinds of data that form two networks in a similar way. For example, we can look at the people who edit Wikipedia articles and the articles that they edit. Two editors are connected if they have edited an article in common. The communities are groups of editors that are interested in the same subject. Dually, we can build a network of articles, and connect articles if they have been edited by the same person. Here, we get communities of articles on similar or related subjects.

In fact, the data involved in Collaborative filtering, as was discussed in Chapter 9, often can be viewed as forming a pair of networks, one for the customers and one for the products. Customers who buy the same sorts of products, e.g., science-fiction books, will form communities, and dually, products that are bought by the same customers will form communities, e.g., all science-fiction books.

10.1.4 Graphs With Several Node Types

There are other social phenomena that involve entities of different types. We just discussed under the heading of "collaboration networks," several kinds of Graphs that are really formed from two types of nodes. Authorship networks can be seen to have author nodes and paper nodes. In the discussion above, we built two social networks by eliminating the nodes of one of the two types, but we do not have to do that. We can rather think of the structure as a whole.

For a more complex example, users at a site like deli.cio.us place tags on Web pages. There are thus three different kinds of entities: users, tags, and pages. We might think that users were somehow connected if they tended to use the same tags frequently, or if they tended to tag the same pages. Similarly, tags could be considered related if they appeared on the same pages or were used by the same users, and pages could be considered similar if they had many of the same tags or were tagged by many of the same users.

The natural way to represent such information is as a k-partite graph for some $k > 1$. We met bipartite graphs, the case $k = 2$, in Section 8.3. In general, a *k-partite graph* consists of k disjoint sets of nodes, with no edges between nodes of the same set.

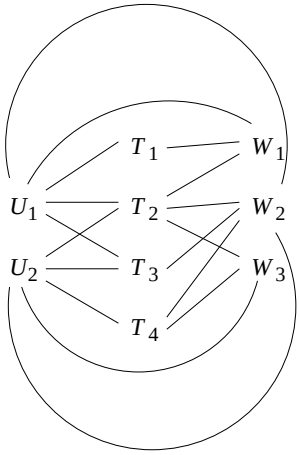

Figure 10.2: A tripartite graph representing users, tags, and Web pages

Example 10.2: Figure 10.2 is an example of a tripartite graph (the case $k = 3$

of a k-partite graph). There are three sets of nodes, which we may think of as users $\{U_1, U_2\}$, tags $\{T_1, T_2, T_3, T_4\}$, and Web pages $\{W_1, W_2, W_3\}$. Notice that all edges connect nodes from two different sets. We may assume this graph represents information about the three kinds of entities. For example, the edge (U_1, T_2) means that user U_1 has placed the tag T_2 on at least one page. Note that the graph does not tell us a detail that could be important: who placed which tag on which page? To represent such ternary information would require a more complex representation, such as a database relation with three columns corresponding to users, tags, and pages. □

10.1.5 Exercises for Section 10.1

Exercise 10.1.1: It is possible to think of the edges of one graph G as the nodes of another graph G'. We construct G' from G by the *dual construction*:

1. If (X, Y) is an edge of G, then XY, representing the unordered set of X and Y is a node of G'. Note that XY and YX represent the same node of G', not two different nodes.
2. If (X, Y) and (X, Z) are edges of G, then in G' there is an edge between XY and XZ. That is, nodes of G' have an edge between them if the edges of G that these nodes represent have a node (of G) in common.

(a) If we apply the dual construction to a network of friends, what is the interpretation of the edges of the resulting graph?

(b) Apply the dual construction to the graph of Fig. 10.1.

! (c) How is the degree of a node XY in G' related to the degrees of X and Y in G?

!! (d) The number of edges of G' is related to the degrees of the nodes of G by a certain formula. Discover that formula.

! (e) What we called the dual is not a true dual, because applying the construction to G' does not necessarily yield a graph isomorphic to G. Give an example graph G where the dual of G' *is* isomorphic to G and another example where the dual of G' is *not* isomorphic to G.

10.2 Clustering of Social-Network Graphs

An important aspect of social networks is that they contain communities of entities that are connected by many edges. These typically correspond to groups of friends at school or groups of researchers interested in the same topic, for example. In this section, we shall consider clustering of the graph as a way to identify communities. It turns out that the techniques we learned in Chapter 7 are generally unsuitable for the problem of clustering social-network graphs.

10.2.1 Distance Measures for Social-Network Graphs

If we were to apply standard clustering techniques to a social-network graph, our first step would be to define a distance measure. When the edges of the graph have labels, these labels might be usable as a distance measure, depending on what they represented. But when the edges are unlabeled, as in a "friends" graph, there is not much we can do to define a suitable distance.

Our first instinct is to assume that nodes are close if they have an edge between them and distant if not. Thus, we could say that the distance $d(x, y)$ is 0 of there is an edge (x, y) and 1 if there is no such edge. We could use any other two values, such as 1 and ∞, as long as the distance is closer when there is an edge.

Neither of these two-valued "distance measures" – 0 and 1 or 1 and ∞ – is a true distance measure. The reason is that they violate the triangle inequality when there are three nodes, with two edges between them. That is, if there are edges (A, B) and (B, C), but no edge (A, C), then the distance from A to C exceeds the sum of the distances from A to B to C. We could fix this problem by using, say, distance 1 for an edge and distance 1.5 for a missing edge. But the problem with two-valued distance functions is not limited to the triangle inequality, as we shall see in the next section.

10.2.2 Applying Standard Clustering Methods

Recall from Section 7.1.2 that there are two general approaches to clustering: hierarchical (agglomerative) and point-assignment. Let us consider how each of these would work on a social-network graph. First, consider the hierarchical methods covered in Section 7.2. In particular, suppose we use as the intercluster distance the minimum distance between nodes of the two clusters.

Hierarchical clustering of a social-network graph starts by combining some two nodes that are connected by an edge. Successively, edges that are not between two nodes of the same cluster would be chosen randomly to combine the clusters to which their two nodes belong. The choices would be random, because all distances represented by an edge are the same.

Example 10.3: Consider again the graph of Fig. 10.1, repeated here as Fig. 10.3. First, let us agree on what the communities are. At the highest level, it appears that there are two communities $\{A, B, C\}$ and $\{D, E, F, G\}$. However, we could also view $\{D, E, F\}$ and $\{D, F, G\}$ as two subcommunities of $\{D, E, F, G\}$; these two subcommunities overlap in two of their members, and thus could never be identified by a pure clustering algorithm. Finally, we could consider each pair of individuals that are connected by an edge as a community of size 2, although such communities are uninteresting.

The problem with hierarchical clustering of a graph like that of Fig. 10.3 is that at some point we are likely to chose to combine B and D, even though they surely belong in different clusters. The reason we are likely to combine B and D is that D, and any cluster containing it, is as close to B and any cluster

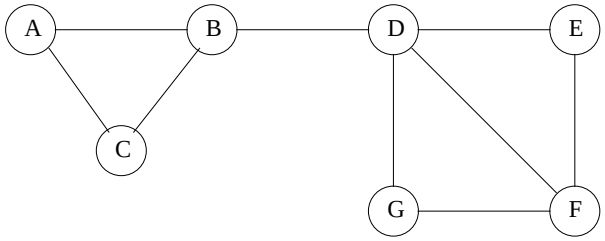

Figure 10.3: Repeat of Fig. 10.1

containing it, as A and C are to B. There is even a 1/9 probability that the first thing we do is to combine B and D into one cluster.

There are things we can do to reduce the probability of error. We can run hierarchical clustering several times and pick the run that gives the most coherent clusters. We can use a more sophisticated method for measuring the distance between clusters of more than one node, as discussed in Section 7.2.3. But no matter what we do, in a large graph with many communities there is a significant chance that in the initial phases we shall use some edges that connect two nodes that do not belong together in any large community. □

Now, consider a point-assignment approach to clustering social networks. Again, the fact that all edges are at the same distance will introduce a number of random factors that will lead to some nodes being assigned to the wrong cluster. An example should illustrate the point.

Example 10.4: Suppose we try a k-means approach to clustering Fig. 10.3. As we want two clusters, we pick $k = 2$. If we pick two starting nodes at random, they might both be in the same cluster. If, as suggested in Section 7.3.2, we start with one randomly chosen node and then pick another as far away as possible, we don't do much better; we could thereby pick any pair of nodes not connected by an edge, e.g., E and G in Fig. 10.3.

However, suppose we do get two suitable starting nodes, such as B and F. We shall then assign A and C to the cluster of B and assign E and G to the cluster of F. But D is as close to B as it is to F, so it could go either way, even though it is "obvious" that D belongs with F.

If the decision about where to place D is deferred until we have assigned some other nodes to the clusters, then we shall probably make the right decision. For instance, if we assign a node to the cluster with the shortest average distance to all the nodes of the cluster, then D should be assigned to the cluster of F, as long as we do not try to place D before any other nodes are assigned. However, in large graphs, we shall surely make mistakes on some of the first nodes we place. □

10.2.3 Betweenness

Since there are problems with standard clustering methods, several specialized clustering techniques have been developed to find communities in social networks. In this section we shall consider one of the simplest, based on finding the edges that are least likely to be inside a community.

Define the *betweenness* of an edge (a, b) to be the number of pairs of nodes x and y such that the edge (a, b) lies on the shortest path between x and y. To be more precise, since there can be several shortest paths between x and y, edge (a, b) is credited with the fraction of those shortest paths that include the edge (a, b). As in golf, a high score is bad. It suggests that the edge (a, b) runs between two different communities; that is, a and b do not belong to the same community.

Example 10.5: In Fig. 10.3 the edge (B, D) has the highest betweenness, as should surprise no one. In fact, this edge is on every shortest path between any of A, B, and C to any of D, E, F, and G. Its betweenness is therefore $3 \times 4 = 12$. In contrast, the edge (D, F) is on only four shortest paths: those from A, B, C, and D to F. □

10.2.4 The Girvan-Newman Algorithm

In order to exploit the betweenness of edges, we need to calculate the number of shortest paths going through each edge. We shall describe a method called the *Girvan-Newman* (GN) Algorithm, which visits each node X once and computes the number of shortest paths from X to each of the other nodes that go through each of the edges. The algorithm begins by performing a breadth-first search (BFS) of the graph, starting at the node X. Note that the level of each node in the BFS presentation is the length of the shortest path from X to that node. Thus, the edges that go between nodes at the same level can never be part of a shortest path from X.

Edges between levels are called *DAG* edges ("DAG" stands for directed, acyclic graph). Each DAG edge will be part of at least one shortest path from root X. If there is a DAG edge (Y, Z), where Y is at the level above Z (i.e., closer to the root), then we shall call Y a *parent* of Z and Z a *child* of Y, although parents are not necessarily unique in a DAG as they would be in a tree.

Example 10.6: Figure 10.4 is a breadth-first presentation of the graph of Fig. 10.3, starting at node E. Solid edges are DAG edges and dashed edges connect nodes at the same level. □

The second step of the GN algorithm is to label each node by the number of shortest paths that reach it from the root. Start by labeling the root 1. Then, from the top down, label each node Y by the sum of the labels of its parents.

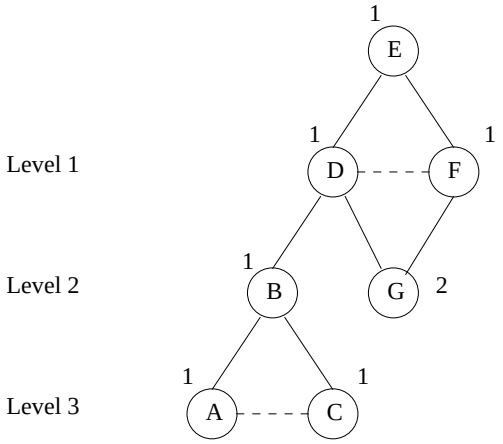

Figure 10.4: Step 1 of the Girvan-Newman Algorithm

Example 10.7: In Fig. 10.4 are the labels for each of the nodes. First, label the root E with 1. At level 1 are the nodes D and F. Each has only E as a parent, so they too are labeled 1. Nodes B and G are at level 2. B has only D as a parent, so B's label is the same as the label of D, which is 1. However, G has parents D and F, so its label is the sum of their labels, or 2. Finally, at level 3, A and C each have only parent B, so their labels are the label of B, which is 1. □

The third and final step is to calculate for each edge e the sum over all nodes Y of the fraction of shortest paths from the root X to Y that go through e. This calculation involves computing this sum for both nodes and edges, from the bottom. Each node other than the root is given a *credit* of 1, representing the shortest path to that node. This credit may be divided among nodes and edges above, since there could be several different shortest paths to the node. The rules for the calculation are as follows:

1. Each leaf in the DAG (a *leaf* is a node with no DAG edges to nodes at levels below) gets a credit of 1.

2. Each node that is not a leaf gets a credit equal to 1 plus the sum of the credits of the DAG edges from that node to the level below.

3. A DAG edge e entering node Z from the level above is given a share of the credit of Z proportional to the fraction of shortest paths from the root to Z that go through e. Formally, let the parents of Z be $Y_1, Y_2, \ldots, Y_k$. Let p_i be the number of shortest paths from the root to Y_i; this number was computed in Step 2 and is illustrated by the labels in Fig. 10.4. Then the credit for the edge (Y_i, Z) is the credit of Z times p_i divided by $\sum_{j=1}^{k} p_j$.

After performing the credit calculation with each node as the root, we sum the credits for each edge. Then, since each shortest path will have been discovered twice – once when each of its endpoints is the root – we must divide the credit for each edge by 2.

Example 10.8: Let us perform the credit calculation for the BFS presentation of Fig. 10.4. We shall start from level 3 and proceed upwards. First, A and C, being leaves, get credit 1. Each of these nodes have only one parent, so their credit is given to the edges (B, A) and (B, C), respectively.

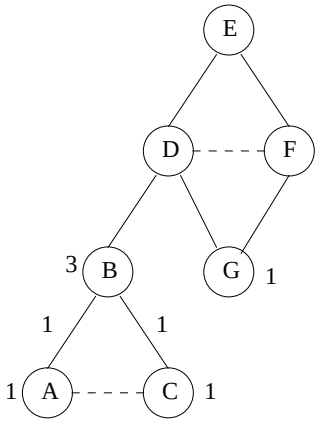

Figure 10.5: Final step of the Girvan-Newman Algorithm – levels 3 and 2

At level 2, G is a leaf, so it gets credit 1. B is not a leaf, so it gets credit equal to 1 plus the credits on the DAG edges entering it from below. Since both these edges have credit 1, the credit of B is 3. Intuitively 3 represents the fact that all shortest paths from E to A, B, and C go through B. Figure 10.5 shows the credits assigned so far.

Now, let us proceed to level 1. B has only one parent, D, so the edge (D, B) gets the entire credit of B, which is 3. However, G has two parents, D and F. We therefore need to divide the credit of 1 that G has between the edges (D, G) and (F, G). In what proportion do we divide? If you examine the labels of Fig. 10.4, you see that both D and F have label 1, representing the fact that there is one shortest path from E to each of these nodes. Thus, we give half the credit of G to each of these edges; i.e., their credit is each $1/(1 + 1) = 0.5$. Had the labels of D and F in Fig. 10.4 been 5 and 3, meaning there were five shortest paths to D and only three to F, then the credit of edge (D, G) would have been 5/8 and the credit of edge (F, G) would have been 3/8.

Now, we can assign credits to the nodes at level 1. D gets 1 plus the credits of the edges entering it from below, which are 3 and 0.5. That is, the credit of D is 4.5. The credit of F is 1 plus the credit of the edge (F, G), or 1.5. Finally, the edges (E, D) and (E, F) receive the credit of D and F, respectively, since each of these nodes has only one parent. These credits are all shown in Fig. 10.6.

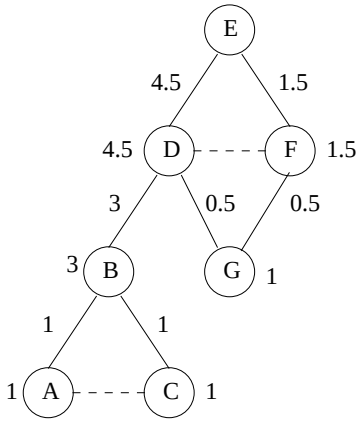

Figure 10.6: Final step of the Girvan-Newman Algorithm – completing the credit calculation

The credit on each of the edges in Fig. 10.6 is the contribution to the betweenness of that edge due to shortest paths from E. For example, this contribution for the edge (E, D) is 4.5. □

To complete the betweenness calculation, we have to repeat this calculation for every node as the root and sum the contributions. Finally, we must divide by 2 to get the true betweenness, since every shortest path will be discovered twice, once for each of its endpoints.

10.2.5 Using Betweenness to Find Communities

The betweenness scores for the edges of a graph behave something like a distance measure on the nodes of the graph. It is not exactly a distance measure, because it is not defined for pairs of nodes that are unconnected by an edge, and might not satisfy the triangle inequality even when defined. However, we can cluster by taking the edges in order of increasing betweenness and add them to the graph one at a time. At each step, the connected components of the graph form some clusters. The higher the betweenness we allow, the more edges we get, and the larger the clusters become.

More commonly, this idea is expressed as a process of edge removal. Start with the graph and all its edges; then remove edges with the highest betweenness, until the graph has broken into a suitable number of connected components.

Example 10.9 : Let us start with our running example, the graph of Fig. 10.1. We see it with the betweenness for each edge in Fig. 10.7. The calculation of the betweenness will be left to the reader. The only tricky part of the count is to observe that between E and G there are two shortest paths, one going

through D and the other through F. Thus, each of the edges (D, E), (E, F), (D, G), and (G, F) are credited with half a shortest path.

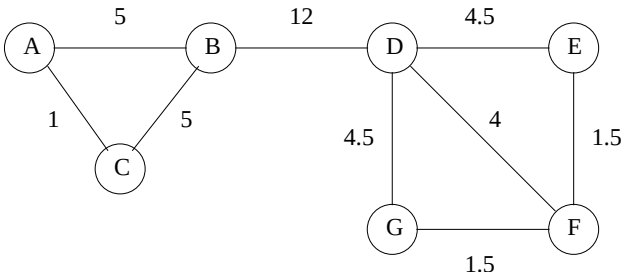

Figure 10.7: Betweenness scores for the graph of Fig. 10.1

Clearly, edge (B, D) has the highest betweenness, so it is removed first. That leaves us with exactly the communities we observed make the most sense: $\{A, B, C\}$ and $\{D, E, F, G\}$. However, we can continue to remove edges. Next to leave are (A, B) and (B, C) with a score of 5, followed by (D, E) and (D, G) with a score of 4.5. Then, (D, F), whose score is 4, would leave the graph. We see in Fig. 10.8 the graph that remains.

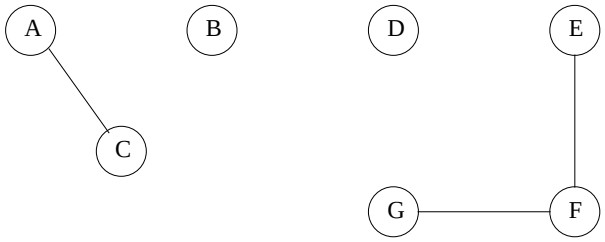

Figure 10.8: All the edges with betweenness 4 or more have been removed

The "communities" of Fig. 10.8 look strange. One implication is that A and C are more closely knit to each other than to B. That is, in some sense B is a "traitor" to the community $\{A, B, C\}$ because he has a friend D outside that community. Likewise, D can be seen as a "traitor" to the group $\{D, E, F, G\}$, which is why in Fig. 10.8, only E, F, and G remain connected. □

10.2.6 Exercises for Section 10.2

Exercise 10.2.1: Figure 10.9 is an example of a social-network graph. Use the Girvan-Newman approach to find the number of shortest paths from each of the following nodes that pass through each of the edges. (a) A (b) B.

Speeding Up the Betweenness Calculation

If we apply the method of Section 10.2.4 to a graph of n nodes and e edges, it takes $O(ne)$ running time to compute the betweenness of each edge. That is, BFS from a single node takes $O(e)$ time, as do the two labeling steps. We must start from each node, so there are n of the computations described in Section 10.2.4.

If the graph is large – and even a million nodes is large when the algorithm takes $O(ne)$ time – we cannot afford to execute it as suggested. However, if we pick a subset of the nodes at random and use these as the roots of breadth-first searches, we can get an approximation to the betweenness of each edge that will serve in most applications.

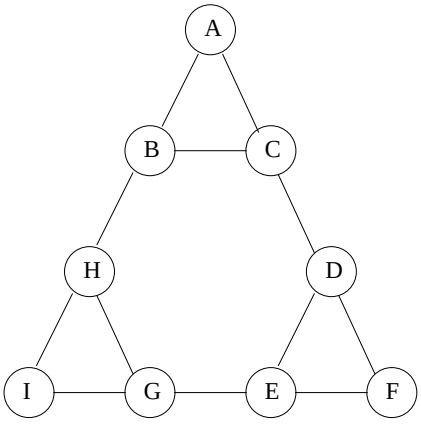

Figure 10.9: Graph for exercises

Exercise 10.2.2: Using symmetry, the calculations of Exercise 10.2.1 are all you need to compute the betweenness of each edge. Do the calculation.

Exercise 10.2.3: Using the betweenness values from Exercise 10.2.2, determine reasonable candidates for the communities in Fig. 10.9 by removing all edges with a betweenness above some threshold.

10.3 Direct Discovery of Communities

In the previous section we searched for communities by partitioning all the individuals in a social network. While this approach is relatively efficient, it does have several limitations. It is not possible to place an individual in two different communities, and everyone is assigned to a community. In this section, we shall

see a technique for discovering communities directly by looking for subsets of the nodes that have a relatively large number of edges among them. Interestingly, the technique for doing this search on a large graph involves finding large frequent itemsets, as was discussed in Chapter 6.

10.3.1 Finding Cliques

Our first thought about how we could find sets of nodes with many edges between them is to start by finding a large *clique* (a set of nodes with edges between any two of them). However, that task is not easy. Not only is finding maximal cliques NP-complete, but it is among the hardest of the NP-complete problems in the sense that even approximating the maximal clique is hard. Further, it is possible to have a set of nodes with almost all edges between them, and yet have only relatively small cliques.

Example 10.10: Suppose our graph has nodes numbered $1, 2, \ldots, n$ and there is an edge between two nodes i and j unless i and j have the same remainder when divided by k. Then the fraction of possible edges that are actually present is approximately $(k-1)/k$. There are many cliques of size k, of which $\{1, 2, \ldots, k\}$ is but one example.

Yet there are no cliques larger than k. To see why, observe that any set of $k+1$ nodes has two that leave the same remainder when divided by k. This point is an application of the "pigeonhole principle." Since there are only k different remainders possible, we cannot have distinct remainders for each of $k+1$ nodes. Thus, no set of $k+1$ nodes can be a clique in this graph. □

10.3.2 Complete Bipartite Graphs

Recall our discussion of bipartite graphs from Section 8.3. A *complete bipartite graph* consists of s nodes on one side and t nodes on the other side, with all st possible edges between the nodes of one side and the other present. We denote this graph by $K_{s,t}$. You should draw an analogy between complete bipartite graphs as subgraphs of general bipartite graphs and cliques as subgraphs of general graphs. In fact, a clique of s nodes is often referred to as a *complete graph* and denoted K_s, while a complete bipartite subgraph is sometimes called a *bi-clique.*.

While as we saw in Example 10.10, it is not possible to guarantee that a graph with many edges necessarily has a large clique, it *is* possible to guarantee that a bipartite graph with many edges has a large complete bipartite subgraph.[1] We can regard a complete bipartite subgraph (or a clique if we discovered a large one) as the nucleus of a community and add to it nodes with many edges to existing members of the community. If the graph itself is

[1]It is important to understand that we do not mean a *generated* subgraph – one formed by selecting some nodes and including all edges. In this context, we only require that there be edges between any pair of nodes on different sides. It is also possible that some nodes on the same side are connected by edges as well.

k-partite as discussed in Section 10.1.4, then we can take nodes of two types and the edges between them to form a bipartite graph. In this bipartite graph, we can search for complete bipartite subgraphs as the nuclei of communities. For instance, in Example 10.2, we could focus on the tag and page nodes of a graph like Fig. 10.2 and try to find communities of tags and Web pages. Such a community would consist of related tags and related pages that deserved many or all of those tags.

However, we can also use complete bipartite subgraphs for community finding in ordinary graphs where nodes all have the same type. Divide the nodes into two equal groups at random. If a community exists, then we would expect about half its nodes to fall into each group, and we would expect that about half its edges would go between groups. Thus, we still have a reasonable chance of identifying a large complete bipartite subgraph in the community. To this nucleus we can add nodes from either of the two groups, if they have edges to many of the nodes already identified as belonging to the community.

10.3.3 Finding Complete Bipartite Subgraphs

Suppose we are given a large bipartite graph G , and we want to find instances of $K_{s,t}$ within it. It is possible to view the problem of finding instances of $K_{s,t}$ within G as one of finding frequent itemsets. For this purpose, let the "items" be the nodes on one side of G, which we shall call the *left* side. We assume that the instance of $K_{s,t}$ we are looking for has t nodes on the left side, and we shall also assume for efficiency that $t \leq s$. The "baskets" correspond to the nodes on the other side of G (the *right* side). The members of the basket for node v are the nodes of the left side to which v is connected. Finally, let the support threshold be s, the number of nodes that the instance of $K_{s,t}$ has on the right side.

We can now state the problem of finding instances of $K_{s,t}$ as that of finding frequent itemsets F of size t. That is, if a set of t nodes on the left side is frequent, then they all occur together in at least s baskets. But the baskets are the nodes on the right side. Each basket corresponds to a node that is connected to all t of the nodes in F. Thus, the frequent itemset of size t and s of the baskets in which all those items appear form an instance of $K_{s,t}$.

Example 10.11 : Recall the bipartite graph of Fig. 8.1, which we repeat here as Fig. 10.10. The left side is the nodes $\{1, 2, 3, 4\}$ and the right side is $\{a, b, c, d\}$. The latter are the baskets, so basket a consists of "items" 1 and 4; that is, $a = \{1, 4\}$. Similarly, $b = \{2, 3\}$, $c = \{1\}$ and $d = \{3\}$.

If $s = 2$ and $t = 1$, we must find itemsets of size 1 that appear in at least two baskets. $\{1\}$ is one such itemset, and $\{3\}$ is another. However, in this tiny example there are no itemsets for larger, more interesting values of s and t, such as $s = t = 2$. □

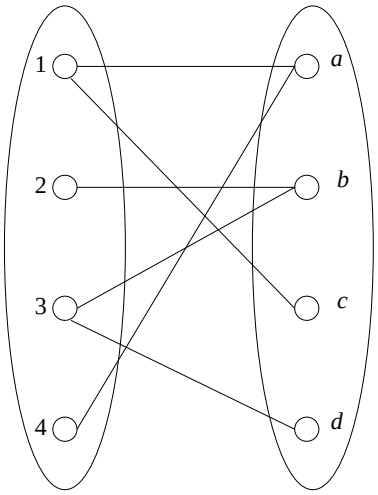

Figure 10.10: The bipartite graph from Fig. 8.1

10.3.4 Why Complete Bipartite Graphs Must Exist

We must now turn to the matter of demonstrating that any bipartite graph with a sufficiently high fraction of the edges present will have an instance of $K_{s,t}$. In what follows, assume that the graph G has n nodes on the left and another n nodes on the right. Assume the two sides have the same number of nodes simplifies the calculation, but the argument generalizes to sides of any size. Finally, let d be the average degree of all nodes.

The argument involves counting the number of frequent itemsets of size t that a basket with d items contributes to. When we sum this number over all nodes on the right side, we get the total frequency of all the subsets of size t on the left. When we divide by $\binom{n}{t}$, we get the average frequency of all itemsets of size t. At least one must have a frequency that is at least average, so if this average is at least s, we know an instance of $K_{s,t}$ exists.

Now, we provide the detailed calculation. Suppose the degree of the ith node on the right is d_i; that is, d_i is the size of the ith basket. Then this basket contributes to $\binom{d_i}{t}$ itemsets of size t. The total contribution of the n nodes on the right is $\sum_i \binom{d_i}{t}$. The value of this sum depends on the d_i's, of course. However, we know that the average value of d_i is d. It is known that this sum is minimized when each d_i is d. We shall not prove this point, but a simple example will suggest the reasoning: since $\binom{d_i}{t}$ grows roughly as the tth power of d_i, moving 1 from a large d_i to some smaller d_j will reduce the sum of $\binom{d_i}{t} + \binom{d_j}{t}$.

Example 10.12: Suppose there are only two nodes, $t = 2$, and the average degree of the nodes is 4. Then $d_1 + d_2 = 8$, and the sum of interest is $\binom{d_1}{2} + \binom{d_2}{2}$. If $d_1 = d_2 = 4$, then the sum is $\binom{4}{2} + \binom{4}{2} = 6 + 6 = 12$. However, if $d_1 = 5$ and

$d_2 = 3$, the sum is $\binom{5}{2} + \binom{3}{2} = 10 + 3 = 13$. If $d_1 = 6$ and $d_1 = 2$, then the sum is $\binom{6}{2} + \binom{2}{2} = 15 + 1 = 16$. □

Thus, in what follows, we shall assume that all nodes have the average degree d. So doing minimizes the total contribution to the counts for the itemsets, and thus makes it least likely that there will be a frequent itemset (itemset with with support s or more) of size t. Observe the following:

- The total contribution of the n nodes on the right to the counts of the itemsets of size t is $n\binom{d}{t}$.
- The number of itemsets of size t is $\binom{n}{t}$.
- Thus, the average count of an itemset of size t is $n\binom{d}{t}/\binom{n}{t}$; this expression must be at least s if we are to argue that an instance of $K_{s,t}$ exists.

If we expand the binomial coefficients in terms of factorials, we find

$$n\binom{d}{t}/\binom{n}{t} = nd!(n-t)!t!/\big((d-t)!t!n!\big) \;=$$

$$n(d)(d-1)\cdots(d-t+1)/\big(n(n-1)\cdots(n-t+1)\big)$$

To simplify the formula above, let us assume that n is much larger than d, and d is much larger than t. Then $d(d-1)\cdots(d-t+1)$ is approximately d^t, and $n(n-1)\cdots(n-t+1)$ is approximately n^t. We thus require that

$$n(d/n)^t \geq s$$

That is, if there is a community with n nodes on each side, the average degree of the nodes is d, and $n(d/n)^t \geq s$, then this community is guaranteed to have a complete bipartite subgraph $K_{s,t}$. Moreover, we can find the instance of $K_{s,t}$ efficiently, using the methods of Chapter 6, even if this small community is embedded in a much larger graph. That is, we can treat all nodes in the entire graph as baskets and as items, and run A-priori or one of its improvements on the entire graph, looking for sets of t items with support s.

Example 10.13: Suppose there is a community with 100 nodes on each side, and the average degree of nodes is 50; i.e., half the possible edges exist. This community will have an instance of $K_{s,t}$, provided $100(1/2)^t \geq s$. For example, if $t = 2$, then s can be as large as 25. If $t = 3$, s can be 11, and if $t = 4$, s can be 6.

Unfortunately, the approximation we made gives us a bound on s that is a little too high. If we revert to the original formula $n\binom{d}{t}/\binom{n}{t} \geq s$, we see that for the case $t = 4$ we need $100\binom{50}{4}/\binom{100}{4} \geq s$. That is,

$$\frac{100 \times 50 \times 49 \times 48 \times 47}{100 \times 99 \times 98 \times 97} \geq s$$

The expression on the left is not 6, but only 5.87. However, if the average support for an itemset of size 4 is 5.87, then it is impossible that all those itemsets have support 5 or less. Thus, we can be sure that at least one itemset of size 4 has support 6 or more, and an instance of $K_{6,4}$ exists in this community. □

10.3.5 Exercises for Section 10.3

Exercise 10.3.1: For the running example of a social network from Fig. 10.1, how many instances of $K_{s,t}$ are there for:

(a) $s = 1$ and $t = 3$.

(b) $s = 2$ and $t = 2$.

(c) $s = 2$ and $t = 3$.

Exercise 10.3.2: Suppose there is a community of $2n$ nodes. Divide the community into two groups of n members, at random, and form the bipartite graph between the two groups. Suppose that the average degree of the nodes of the bipartite graph is d. Find the set of maximal pairs (t, s), with $t \leq s$, such that an instance of $K_{s,t}$ is guaranteed to exist, for the following combinations of n and d:

(a) $n = 20$ and $d = 5$.

(b) $n = 200$ and $d = 150$.

(c) $n = 1000$ and $d = 400$.

By "maximal," we mean there is no different pair (s', t') such that both $s' \geq s$ and $t' \geq t$ hold.

10.4 Partitioning of Graphs

In this section, we examine another approach to organizing social-network graphs. We use some important tools from matrix theory ("spectral methods") to formulate the problem of partitioning a graph to minimize the number of edges that connect different components. The goal of minimizing the "cut" size needs to be understood carefully before proceeding. For instance, if you just joined Facebook, you are not yet connected to any friends. We do not want to partition the friends graph with you in one group and the rest of the world in the other group, even though that would partition the graph without there being any edges that connect members of the two groups. This cut is not desirable because the two components are too unequal in size.

10.4.1 What Makes a Good Partition?

Given a graph, we would like to divide the nodes into two sets so that the *cut*, or set of edges that connect nodes in different sets is minimized. However, we also want to constrain the selection of the cut so that the two sets are approximately equal in size. The next example illustrates the point.

Example 10.14: Recall our running example of the graph in Fig. 10.1. There, it is evident that the best partition puts $\{A, B, C\}$ in one set and $\{D, E, F, G\}$ in the other. The cut consists only of the edge (B, D) and is of size 1. No nontrivial cut can be smaller.

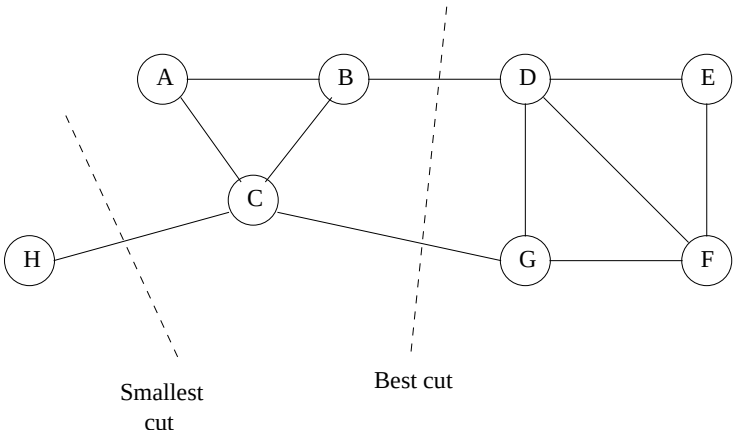

Figure 10.11: The smallest cut might not be the best cut

In Fig. 10.11 is a variant of our example, where we have added the node H and two extra edges, (H, C) and (C, G). If all we wanted was to minimize the size of the cut, then the best choice is to put H in one set and all the other nodes in the other set. But it should be apparent that if we reject partitions where one set is too small, then the best we can do is to use the cut consisting of edges (B, D) and (C, G), which partitions the graph into two equal-sized sets $\{A, B, C, H\}$ and $\{D, E, F, G\}$. □

10.4.2 Normalized Cuts

A proper definition of a "good" cut must balance the size of the cut itself against the difference in the sizes of the sets that the cut creates. One choice that serves well is the "normalized cut." First, define the *volume* of a set S of nodes, denoted $Vol(S)$, to be the number of edges with at least one end in S.

Suppose we partition the nodes of a graph into two disjoint sets S and T. Let $Cut(S, T)$ be the number of edges that connect a node in S to a node in T.

Then the *normalized cut* value for S and T is

$$\frac{Cut(S,T)}{Vol(S)} + \frac{Cut(S,T)}{Vol(T)}$$

Example 10.15: Again consider the graph of Fig. 10.11. If we choose $S = \{H\}$ and $T = \{A, B, C, D, E, F, G\}$, then $Cut(S,T) = 1$. $Vol(S) = 1$, because there is only one edge connected to H. On the other hand, $Vol(T) = 11$, because all the edges have at least one end at a node of T. Thus, the normalized cut for this partition is $1/1 + 1/11 = 1.09$.

Now, consider the preferred cut for this graph consisting of the edges (B, D) and (C, G). Then $S = \{A, B, C, H\}$ and $T = \{D, E, F, G\}$. $Cut(S,T) = 2$, $Vol(S) = 6$, and $Vol(T) = 7$. The normalized cut for this partition is thus only $2/6 + 2/7 = 0.62$. □

10.4.3 Some Matrices That Describe Graphs

To develop the theory of how matrix algebra can help us find good graph partitions, we first need to learn about three different matrices that describe aspects of a graph. The first should be familiar: the *adjacency matrix* that has a 1 in row i and column j if there is an edge between nodes i and j, and 0 otherwise.

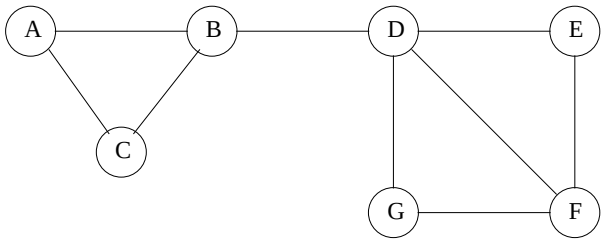

Figure 10.12: Repeat of the graph of Fig. 10.1

Example 10.16: We repeat our running example graph in Fig. 10.12. Its adjacency matrix appears in Fig. 10.13. Note that the rows and columns correspond to the nodes $A, B, \ldots, G$ in that order. For example, the edge (B, D) is reflected by the fact that the entry in row 2 and column 4 is 1 and so is the entry in row 4 and column 2. □

The second matrix we need is the *degree matrix* for a graph. This graph has entries only on the diagonal. The entry for row and column i is the degree of the ith node.

Example 10.17: The degree matrix for the graph of Fig. 10.12 is shown in Fig. 10.14. We use the same order of the nodes as in Example 10.16. For

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 10.13: The adjacency matrix for Fig. 10.12

instance, the entry in row 4 and column 4 is 4 because node D has edges to four other nodes. The entry in row 4 and column 5 is 0, because that entry is not on the diagonal. □

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Figure 10.14: The degree matrix for Fig. 10.12

Suppose our graph has adjacency matrix A and degree matrix D. Our third matrix, called the *Laplacian matrix*, is $L = D - A$, the difference between the degree matrix and the adjacency matrix. That is, the Laplacian matrix L has the same entries as D on the diagonal. Off the diagonal, at row i and column j, L has -1 if there is an edge between nodes i and j and 0 if not.

Example 10.18: The Laplacian matrix for the graph of Fig. 10.12 is shown in Fig. 10.15. Notice that each row and each column sums to zero, as must be the case for any Laplacian matrix. □

10.4.4 Eigenvalues of the Laplacian Matrix

We can get a good idea of the best way to partition a graph from the eigenvalues and eigenvectors of its Laplacian matrix. In Section 5.1.2 we observed how the principal eigenvector (eigenvector associated with the largest eigenvalue) of the transition matrix of the Web told us something useful about the importance of Web pages. In fact, in simple cases (no taxation) the principal eigenvector is the PageRank vector. When dealing with the Laplacian matrix, however, it turns

$$\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 4 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix}$$

Figure 10.15: The Laplacian matrix for Fig. 10.12

out that the smallest eigenvalues and their eigenvectors reveal the information we desire.

The smallest eigenvalue for every Laplacian matrix is 0. and its corresponding eigenvector is $[1, 1, \ldots, 1]$. To see why, let L be the Laplacian matrix for a graph of n nodes, and let $\mathbf{1}$ be the column vector of all 1's with length n. We claim $L\mathbf{1}$ is a column vector of all 0's. To see why, consider row i of L. Its diagonal element has the degree d of node i. Row i also will have d occurrences of -1, and all other elements of row i are 0. Multiplying row i by column vector $\mathbf{1}$ has the effect of summing the row, and this sum is clearly $d + (-1)d = 0$. Thus, we can conclude $L\mathbf{1} = 0\mathbf{1}$, which demonstrates that 0 is an eigenvalue and $\mathbf{1}$ its corresponding eigenvector.

There is a simple way to find the second-smallest eigenvalue for any matrix, such as the Laplacian matrix, that is *symmetric* (the entry in row i and column j equals the entry in row j and column i). While we shall not prove this fact, the second-smallest eigenvalue of L is the minimum of $\mathbf{x}^T L\mathbf{x}$, where $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is a column vector with n components, and the minimum is taken under the constraints:

1. The length of $\mathbf{x}$ is 1; that is $\sum_{i=1}^{n} x_i^2 = 1$.
2. $\mathbf{x}$ is orthogonal to the eigenvector associated with the smallest eigenvalue.

Moreover, the value of $\mathbf{x}$ that achieves this minimum is the second eigenvector.

When L is a Laplacian matrix for an n-node graph, we know something more. The eigenvector associated with the smallest eigenvalue is $\mathbf{1}$. Thus, if $\mathbf{x}$ is orthogonal to $\mathbf{1}$, we must have

$$\mathbf{x}^T\mathbf{1} = \sum_{i=1}^{n} x_i = 0$$

In addition for the Laplacian matrix, the expression $\mathbf{x}^T L\mathbf{x}$ has a useful equivalent expression. Recall that $L = D - A$, where D and A are the degree and adjacency matrices of the same graph. Thus, $\mathbf{x}^T L\mathbf{x} = \mathbf{x}^T D\mathbf{x} - \mathbf{x}^T A\mathbf{x}$. Let us evaluate the term with D and then the term for A. $D\mathbf{x}$ is the column vector $[d_1x_1, d_2x_2, \ldots, d_nx_n]$, where d_i is the degree of the ith node of the graph. Thus, $\mathbf{x}^T D\mathbf{x}$ is $\sum_{i=1}^{n} d_i x_i^2$.

Now, turn to $\mathbf{x}^T A\mathbf{x}$. The ith component of the column vector $A\mathbf{x}$ is the sum of x_j over all j such that there is an edge (i, j) in the graph. Thus, $-\mathbf{x}^T A\mathbf{x}$ is the sum of $-2x_i x_j$ over all pairs of nodes $\{i, j\}$ such that there is an edge between them. Note that the factor 2 appears because each set $\{i, j\}$ corresponds to two terms, $-x_i x_j$ and $-x_j x_i$.

We can group the terms of $\mathbf{x}^T L\mathbf{x}$ in a way that distributes the terms to each pair $\{i, j\}$. From $-\mathbf{x}^T A\mathbf{x}$, we already have the term $-2x_i x_j$. From $\mathbf{x}^T D\mathbf{x}$, we distribute the term $d_i x_i^2$ to the d_i pairs that include node i. As a result, we can associate with each pair $\{i, j\}$ that has an edge between nodes i and j the terms $x_i^2 - 2x_i x_j + x_j^2$. This expression is equivalent to $(x_i - x_j)^2$. Therefore, we have proved that $\mathbf{x}^T L\mathbf{x}$ equals the sum over all graph edges (i, j) of $(x_i - x_j)^2$.

Recall that the second-smallest eigenvalue is the minimum of this expression under the constraint that $\sum_{i=1}^{n} x_i^2 = 1$. Intuitively, we minimize it by making x_i and x_j close whenever there is an edge between nodes i and j in the graph. We might imagine that we could choose $x_i = 1/\sqrt{n}$ for all i and thus make this sum 0. However, recall that we are constrained to choose $\mathbf{x}$ to be orthogonal to $\mathbf{1}$, which means the sum of the x_i's is 0. We are also forced to make $\sum_{i=1}^{n} x_i^2$ be 1, so all components cannot be 0. As a consequence, $\mathbf{x}$ must have some positive and some negative components.

We can obtain a partition of the graph by taking one set to be the nodes i whose corresponding vector component x_i is positive and the other set to be those whose components are negative. This choice does not guarantee a partition into sets of equal size, but the sizes are likely to be close. We believe that the cut between the two sets will have a small number of edges because $(x_i - x_j)^2$ is likely to be smaller if both x_i and x_j have the same sign than if they have different signs. Thus, minimizing $\mathbf{x}^T L\mathbf{x}$ under the required constraints will tend to give x_i and x_j the same sign if there is an edge (i, j).

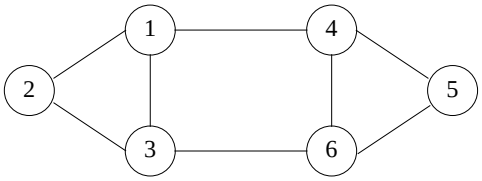

Figure 10.16: Graph for illustrating partitioning by spectral analysis

Example 10.19: Let us apply the above technique to the graph of Fig. 10.16. The Laplacian matrix for this graph is shown in Fig. 10.17. By standard methods or math packages we can find all the eigenvalues and eigenvectors of this matrix. We shall simply tabulate them in Fig. 10.18, from lowest eigenvalue to highest. Note that we have not scaled the eigenvectors to have length 1, but could do so easily if we wished.

The second eigenvector has three positive and three negative components. It makes the unsurprising suggestion that one group should be $\{1, 2, 3\}$, the

$$\begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 3 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & -1 & -1 & 3 \end{bmatrix}$$

Figure 10.17: The Laplacian matrix for Fig. 10.16

nodes with positive components, and the other group should be $\{4, 5, 6\}$. □

Eigenvalue	0	1	3	3	4	5
Eigenvector	1	1	−5	−1	−1	−1
	1	2	4	−2	1	0
	1	1	1	3	−1	1
	1	−1	−5	−1	1	1
	1	−2	4	−2	−1	0
	1	−1	1	3	1	−1

Figure 10.18: Eigenvalues and eigenvectors for the matrix of Fig. 10.17

10.4.5 Alternative Partitioning Methods

The method of Section 10.4.4 gives us a good partition of the graph into two pieces that have a small cut between them. There are several ways we can use the same eigenvectors to suggest other good choices of partition. First, we are not constrained to put all the nodes with positive components in the eigenvector into one group and those with negative components in the other. We could set the threshold at some point other than zero.

For instance, suppose we modified Example 10.19 so that the threshold was not zero, but -1.5. Then the two nodes 4 and 6, with components -1 in the second eigenvector of Fig. 10.18, would join 1, 2, and 3, leaving five nodes in one component and only node 6 in the other. That partition would have a cut of size two, as did the choice based on the threshold of zero, but the two components have radically different sizes, so we would tend to prefer our original choice. However, there are other cases where the threshold zero gives unequally sized components, as would be the case if we used the third eigenvector in Fig. 10.18.

We may also want a partition into more than two components. One approach is to use the method described above to split the graph into two, and then use it repeatedly on the components to split them as far as desired. A second approach is to use several of the eigenvectors, not just the second, to partition the graph. If we use m eigenvectors, and set a threshold for each, we can get a

partition into 2^m groups, each group consisting of the nodes that are above or below threshold for each of the eigenvectors, in a particular pattern.

It is worth noting that each eigenvector except the first is the vector $\mathbf{x}$ that minimizes $\mathbf{x}^T L\mathbf{x}$, subject to the constraint that it is orthogonal to all previous eigenvectors. This constraint generalizes the constraints we described for the second eigenvector in a natural way. As a result, while each eigenvector tries to produce a minimum-sized cut, the fact that successive eigenvectors have to satisfy more and more constraints generally causes the cuts they describe to be progressively worse.

Example 10.20: Let us reconsider the graph of Fig. 10.16, for which the eigenvectors of its Laplacian matrix were tabulated in Fig. 10.18. The third eigenvector, with a threshold of 0, puts nodes 1 and 4 in one group and the other four nodes in the other. That is not a bad partition, but its cut size is four, compared with the cut of size two that we get from the second eigenvector.

If we use both the second and third eigenvectors, we put nodes 2 and 3 in one group, because their components are positive in both eigenvectors. Nodes 5 and 6 are in another group, because their components are negative in the second eigenvector and positive in the third. Node 1 is in a group by itself because it is positive in the second eigenvector and negative in the third, while node 4 is also in a group by itself because its component is negative in both eigenvectors. This partition of a six-node graph into four groups is too fine a partition to be meaningful. But at least the groups of size two each have an edge between the nodes, so it is as good as we could ever get for a partition into groups of these sizes. □

10.4.6 Exercises for Section 10.4

Exercise 10.4.1: For the graph of Fig. 10.9, construct:

(a) The adjacency matrix.

(b) The degree matrix.

(c) The Laplacian matrix.

! **Exercise 10.4.2:** For the Laplacian matrix constructed in Exercise 10.4.1(c), find the second-smallest eigenvalue and its eigenvector. What partition of the nodes does it suggest?

!! **Exercise 10.4.3:** For the Laplacian matrix constructed in Exercise 10.4.1(c), construct the third and subsequent smallest eigenvalues and their eigenvectors.

10.5 Simrank

In this section, we shall take up another approach to analyzing social-network graphs. This technique, called "simrank," applies best to graphs with several

types of nodes, although it can in principle be applied to any graph. The purpose of simrank is to measure the similarity between nodes of the same type, and it does so by seeing where random walkers on the graph wind up when started at a particular node. Because calculation must be carried out once for each starting node, it is limited in the sizes of graphs that can be analyzed completely in this manner.

10.5.1 Random Walkers on a Social Graph

Recall our view of PageRank in Section 5.1 as reflecting what a "random surfer" would do if they walked on the Web graph. We can similarly think of a person "walking" on a social network. The graph of a social network is generally undirected, while the Web graph is directed. However, the difference is unimportant. A walker at a node N of an undirected graph will move with equal probability to any of the *neighbors* of N (those nodes with which N shares an edge).

Suppose, for example, that such a *walker* starts out at node T_1 of Fig. 10.2, which we reproduce here as Fig. 10.19. At the first step, it would go either to U_1 or W_1. If to W_1, then it would next either come back to T_1 or go to T_2. If the walker first moved to U_1, it would wind up at either T_1, T_2, or T_3 next.

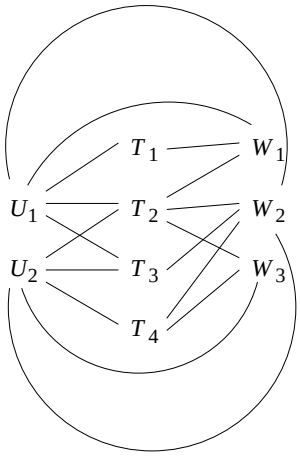

Figure 10.19: Repeat of Fig. 10.2

We conclude that, starting at T_1, there is a good chance the walker would visit T_2, at least initially, and that chance is better than the chance it would visit T_3 or T_4. It would be interesting if we could make the inference that tags T_1 and T_2 are therefore related or similar in some way. The evidence is that they have both been placed on a common Web page, W_1, and they have also been used by a common tagger, U_1.

However, if we allow the walker to continue traversing the graph at random,

then the probability that the walker will be at any particular node does not depend on where it starts out. This conclusion comes from the theory of Markov processes that we mentioned in Section 5.1.2, although the independence from the starting point requires additional conditions besides connectedness that the graph of Fig. 10.19 does satisfy.

10.5.2 Random Walks with Restart

We see from the observations above that it is not possible to measure similarity to a particular node by looking at the limiting distribution of the walker. However, we have already seen, in Section 5.1.5, the introduction of a small probability that the walker will stop walking at random. Then, we saw in Section 5.3.2 that there were reasons to select only a subset of Web pages as the teleport set, the pages that the walker would go to when they stopped surfing the Web at random.

Here, we take this idea to the extreme. As we are focused on one particular node N of a social network, and want to see where the random walker winds up on short walks from that node, we modify the matrix of transition probabilities to have a small additional probability of transitioning to N from any node. Formally, let M be the *transition matrix* of the graph G. That is, the entry in row i and column j of M is $1/k$ if node j of G has degree k, and one of the adjacent nodes is i. Otherwise, this entry is 0. We shall discuss teleporting in a moment, but first, let us look at a simple example of a transition matrix.

Example 10.21 : Figure 10.20 is an example of a very simple network involving three pictures, and two tags, "Sky" and "Tree" that have been placed on some of them. Pictures 1 and 3 have both tags, while Picture 2 has only the tag "Sky." Intuitively, we expect that Picture 3 is more similar to Picture 1 than Picture 2 is, and an analysis using a random walker with restart at Picture 1 will support that intuition.

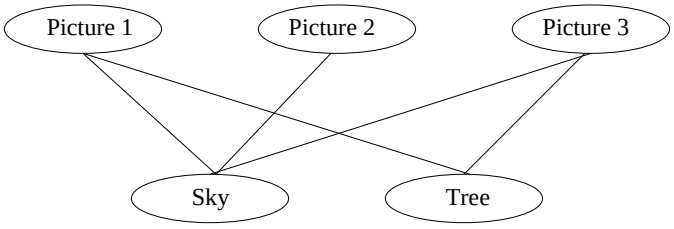

Figure 10.20: A simple bipartite social graph

We shall use, as an order of the nodes Picture 1, Picture 2, Picture 3, Sky,

Tree. Then the transition matrix for the graph of Fig. 10.20 is

$$\begin{bmatrix} 0 & 0 & 0 & 1/3 & 1/2 \\ 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 1/3 & 1/2 \\ 1/2 & 1 & 1/2 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \end{bmatrix}$$

For example, the fourth column corresponds to the node "Sky," and this node connects to each of the tree picture nodes. It therefore has degree three, so the nonzero entries in its column must be 1/3. The picture nodes correspond to the first three rows and columns, so the entry 1/3 appears in the first three rows of column 4. Since the "Sky" node does not have an edge to either itself or the "Tree" node, the entries in the last two rows of column 4 are 0. □

As before, let us use β as the probability that the walker continues at random, so $1 - \beta$ is the probability the walker will teleport to the initial node N. Let $\mathbf{e}_N$ be the column vector that has 1 in the row for node N and 0's elsewhere. Then if $\mathbf{v}$ is the column vector that reflects the probability the walker is at each of the nodes at a particular round, and $\mathbf{v}'$ is the probability the walker is at each of the nodes at the next round, then $\mathbf{v}'$ is related to $\mathbf{v}$ by:

$$\mathbf{v}' = \beta M \mathbf{v} + (1 - \beta)\mathbf{e}_N$$

Example 10.22: Assume M is the matrix of Example 10.21 and $\beta = 0.8$. Also, assume that node N is for Picture 1; that is, we want to compute the similarity of other pictures to Picture 1. Then the equation for the new value $\mathbf{v}'$ of the distribution that we must iterate is

$$\mathbf{v}' = \begin{bmatrix} 0 & 0 & 0 & 4/15 & 2/5 \\ 0 & 0 & 0 & 4/15 & 0 \\ 0 & 0 & 0 & 4/15 & 2/5 \\ 2/5 & 4/5 & 2/5 & 0 & 0 \\ 2/5 & 0 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v} + \begin{bmatrix} 1/5 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Since the graph of Fig. 10.20 is connected, the original matrix M is stochastic, and we can deduce that if the initial vector $\mathbf{v}$ has components that sum to 1, then $\mathbf{v}'$ will also have components that sum to 1. As a result, we can simplify the above equation by adding 1/5 to each of the entries in the first row of the matrix. That is, we can iterate the matrix-vector multiplication

$$\mathbf{v}' = \begin{bmatrix} 1/5 & 1/5 & 1/5 & 7/15 & 3/5 \\ 0 & 0 & 0 & 4/15 & 0 \\ 0 & 0 & 0 & 4/15 & 2/5 \\ 2/5 & 4/5 & 2/5 & 0 & 0 \\ 2/5 & 0 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v}$$

If we start with $\mathbf{v} = \mathbf{e}_N$, then the sequence of estimates of the distribution of the walker that we get is

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1/5 \\ 0 \\ 0 \\ 2/5 \\ 2/5 \end{bmatrix} \begin{bmatrix} 35/75 \\ 8/75 \\ 20/75 \\ 6/75 \\ 6/75 \end{bmatrix} \begin{bmatrix} 95/375 \\ 8/375 \\ 20/375 \\ 142/375 \\ 110/375 \end{bmatrix} \begin{bmatrix} 2353/5625 \\ 568/5625 \\ 1228/5625 \\ 786/5625 \\ 690/5625 \end{bmatrix} \cdots \begin{bmatrix} .345 \\ .066 \\ .145 \\ .249 \\ .196 \end{bmatrix}$$

We observe from the above that in the limit, the walker is more than twice as likely to be at Picture 3 than at Picture 2. This analysis confirms the intuition that Picture 3 is more like Picture 1 than Picture 2 is. □

There are several additional observations that we may take away from Example 10.22. First, remember that this analysis applies only to Picture 1. If we wanted to know what pictures were most similar to another picture, we would have to start the analysis over for that picture. Likewise, if we wanted to know about which tags were most closely associated with the tag "Sky" (an uninteresting question in this small example, since there is only one other tag), then we would have to arrange to have the walker teleport only to the "Sky" node.

Second, notice that convergence takes time, since there is an initial oscillation. That is, initially, all the weight is at the pictures, and at the second step most of the weight is at the tags. At the third step, most weight is back at the pictures, but at the fourth step much of the weight moves to the tags again. However, in the limit there is convergence, with 5/9 of the weight at the pictures and 4/9 of the weight at the tags. In general, the process converges for any connected k-partite graph.

10.5.3 Exercises for Section 10.5

Exercise 10.5.1: If, in Fig. 10.20 you start the walk from Picture 2, what will be the similarity to Picture 2 of the other two pictures? Which do you expect to be more similar to Picture 2?

Exercise 10.5.2: If, in Fig. 10.20 you start the walk from Picture 3, what do you expect the similarity to the other two pictures to be?

! **Exercise 10.5.3:** Repeat the analysis of Example 10.22, and compute the similarities of Picture 1 to the other pictures, if the following modifications are made to Fig. 10.20:

(a) The tag "Tree" is added to Picture 2.

(b) A third tag "Water" is added to Picture 3.

(c) A third tag "Water" is added to both Picture 1 and Picture 2.

Note: the changes are independently done for each part; they are not cumulative.

10.6 Counting Triangles

One of the most useful properties of social-network graphs is the count of triangles and other simple subgraphs. In this section we shall give methods for estimating or getting an exact count of triangles in a very large graph. We begin with a motivation for such counting and then give some methods for doing so efficiently.

10.6.1 Why Count Triangles?

If we start with n nodes and add m edges to a graph at random, there will be an expected number of triangles in the graph. We can calculate this number without too much difficulty. There are $\binom{n}{3}$ sets of three nodes, or approximately $n^3/6$ sets of three nodes that might be a triangle. The probability of an edge between any two given nodes being added is $m/\binom{n}{2}$, or approximately $2m/n^2$. The probability that any set of three nodes has edges between each pair, if those edges are independently chosen to be present or absent is approximately $(2m/n^2)^3 = 8m^3/n^6$. Thus, the expected number of triangles in a graph of n nodes and m randomly selected edges is approximately $(8m^3/n^6)(n^3/6) = \frac{4}{3}(m/n)^3$.

If a graph is a social network with n participants and m pairs of "friends," we would expect the number of triangles to be much greater than the value for a random graph. The reason is that if A and B are friends, and A is also a friend of C, there should be a much greater chance than average that B and C are also friends. Thus, counting the number of triangles helps us to measure the extent to which a graph looks like a social network.

We can also look at communities within a social network. It has been demonstrated that the age of a community is related to the density of triangles. That is, when a group has just formed, people pull in their like-minded friends, but the number of triangles is relatively small. If A brings in friends B and C, it may well be that B and C do not know each other. As the community matures, B and C may interact because of their membership in the community. Thus, there is a good chance that at sometime the triangle $\{A, B, C\}$ will be completed.

10.6.2 An Algorithm for Finding Triangles

We shall begin our study with an algorithm that has the fastest possible running time on a single processor. Suppose we have a graph of n nodes and $m \geq n$ edges. For convenience, assume the nodes are integers $1, 2, \ldots, n$.

Call a node a *heavy hitter* if its degree is at least $\sqrt{m}$. A *heavy-hitter triangle* is a triangle all three of whose nodes are heavy hitters. We use separate algorithms to count the heavy-hitter triangles and all other triangles. Note that the number of heavy-hitter nodes is no more than $2\sqrt{m}$, since otherwise the sum of the degrees of the heavy hitter nodes would be more than $2m$. Since each

edge contributes to the degree of only two nodes, there would then have to be more than m edges.

Assuming the graph is represented by its edges, we preprocess the graph as follows:

1. Compute the degree of each node. This part requires only that we examine each edge and add 1 to the count of each of its two nodes. The total time required is $O(m)$.

2. Create an index on edges, with the pair of nodes at its ends as the key. That is, the index allows us to determine, given two nodes, whether the edge between them exists. A hash table suffices. It can be constructed in $O(m)$ time, and the expected time to answer a query about the existence of an edge is a constant, at least in the expected sense.[2]

3. Create another index of edges, this one with key equal to a single node. Given a node v, we can retrieve the nodes adjacent to v in time proportional to the number of those nodes. Again, a hash table, this time with single nodes as the key, suffices in the expected sense.

We shall order the nodes as follows. First, order nodes by degree. Then, if v and u have the same degree, recall that both v and u are integers, so order them numerically. That is, we say $v \prec u$ if and only if either

(i) The degree of v is less than the degree of u, or

(ii) The degrees of u and v are the same, and $v < u$.

Heavy-Hitter Triangles: There are only $O(\sqrt{m})$ heavy-hitter nodes, so we can consider all sets of three of these nodes. There are $O(m^{3/2})$ possible heavy-hitter triangles, and using the index on edges we can check if all three edges exist in $O(1)$ time. Therefore, $O(m^{3/2})$ time is needed to find all the heavy-hitter triangles.

Other Triangles: We find the other triangles a different way. Consider each edge (v_1, v_2). If both v_1 and v_2 are heavy hitters, ignore this edge. Suppose, however, that v_1 is not a heavy hitter and moreover $v_1 \prec v_2$. Let $u_1, u_2, \ldots, u_k$ be the nodes adjacent to v_1. Note that $k < \sqrt{m}$. We can find these nodes, using the index on nodes, in $O(k)$ time, which is surely $O(\sqrt{m})$ time. For each u_i we can use the first index to check whether edge (u_i, v_2) exists in $O(1)$ time. We can also determine the degree of u_i in $O(1)$ time, because we have counted all the nodes' degrees. We count the triangle $\{v_1, v_2, u_i\}$ if and only if the edge (u_i, v_2) exists, and $v_1 \prec u_i$. In that way, a triangle is counted only once – when v_1 is the node of the triangle that precedes both other nodes of the triangle

[2]Thus, technically, our algorithm is only optimal in the sense of expected running time, not worst-case running time. However, hashing of large numbers of items has an extremely high probability of behaving according to expectation, and if we happened to choose a hash function that made some buckets too big, we could rehash until we found a good hash function.

according to the $\prec$ ordering. Thus, the time to process all the nodes adjacent to v_1 is $O(\sqrt{m})$. Since there are m edges, the total time spent counting other triangles is $O(m^{3/2})$.

We now see that preprocessing takes $O(m)$ time. The time to find heavy-hitter triangles is $O(m^{3/2})$, and so is the time to find the other triangles. Thus, the total time of the algorithm is $O(m^{3/2})$.

10.6.3 Optimality of the Triangle-Finding Algorithm

It turns out that the algorithm described in Section 10.6.2 is, to within an order of magnitude the best possible. To see why, consider a complete graph on n nodes. This graph has $m = \binom{n}{2}$ edges and the number of triangles is $\binom{n}{3}$. Since we cannot enumerate triangles in less time than the number of those triangles, we know any algorithm will take $\Omega(n^3)$ time on this graph. However, $m = O(n^2)$, so any algorithm takes $\Omega(m^{3/2})$ time on this graph.

One might wonder if there were a better algorithm that worked on sparser graphs than the complete graph. However, we can add to the complete graph a chain of nodes with any length up to n^2. This chain adds no more triangles. It no more than doubles the number of edges, but makes the number of nodes as large as we like, in effect lowering the ratio of edges to nodes to be as close to 1 as we like. Since there are still $\Omega(m^{3/2})$ triangles, we see that this lower bound holds for the full range of possible ratios of m/n.

10.6.4 Finding Triangles Using MapReduce

For a very large graph, we want to use parallelism to speed the computation. We can express triangle-finding as a multiway join and use the technique of Section 2.5.3 to optimize the use of a single MapReduce job to count triangles. It turns out that this use is one where the multiway join technique of that section is generally much more efficient than taking two two-way joins. Moreover, the total execution time of the parallel algorithm is essentially the same as the execution time on a single processor using the algorithm of Section 10.6.2.

To begin, assume that the nodes of a graph are numbered $1, 2, \ldots, n$. We use a relation E to represent edges. To avoid representing each edge twice, we assume that if $E(A, B)$ is a tuple of this relation, then not only is there an edge between nodes A and B, but also, as integers, we have $A < B$.[3] This requirement also eliminates loops (edges from a node to itself), which we generally assume do not exist in social-network graphs anyway, but which could lead to "triangles" that really do not involve three different nodes.

Using this relation, we can express the set of triangles of the graph whose edges are E by the natural join

[3]Do not confuse this simple numerical ordering of the nodes with the order $\prec$ that we discussed in Section 10.6.2 and which involved the degrees of the nodes. Here, node degrees are not computed and are not relevant.

$$E(X,Y) \bowtie E(X,Z) \bowtie E(Y,Z) \tag{10.1}$$

To understand this join, we have to recognize that the attributes of the relation E are given different names in each of the three uses of E. That is, we imagine there are three copies of E, each with the same tuples, but with a different schemas. In SQL, this join would be written using a single relation $E(A, B)$ as follows:

```
SELECT e1.A, e1.B, e2.B
FROM E e1, E e2, E e3
WHERE e1.A = e2.A AND e1.B = e3.A AND e2.B = e3.B
```

In this query, the equated attributes $e1.A$ and $e2.A$ are represented in our join by the attribute X. Also, $e1.B$ and $e3.A$ are each represented by Y; $e2.B$ and $e3.B$ are represented by Z.

Notice that each triangle appears once in this join. The triangle consisting of nodes v_1, v_2, and v_3 is generated when X, Y, and Z are these three nodes in numerical order, i.e., $X < Y < Z$. For instance, if the numerical order of the nodes is $v_1 < v_2 < v_3$, then X can only be v_1, Y is v_2, and Z is v_3.

The technique of Section 2.5.3 can be used to optimize the join of Equation 10.1. Recall the ideas in Example 2.9, where we considered the number of ways in which the values of each attribute should be hashed. In the present example, the matter is quite simple. The three occurrences of relation E surely have the same size, so by symmetry, attributes X, Y, and Z will each be hashed to the same number of buckets. In particular, if we hash nodes to b buckets, then there will be b^3 reducers. Each Reduce task is associated with a sequence of three bucket numbers (x, y, z), where each of x, y, and z is in the range 1 to b.

The Map tasks divide the relation E into as many parts as there are Map tasks. Suppose one Map task is given the tuple $E(u, v)$ to send to certain Reduce tasks. First, think of (u, v) as a tuple of the join term $E(X, Y)$. We can hash u and v to get the bucket numbers for X and Y, but we don't know the bucket to which Z hashes. Thus, we must send $E(u, v)$ to all Reducer tasks that correspond to a sequence of three bucket numbers $\big(h(u), h(v), z\big)$ for any of the b possible buckets z.

But the same tuple $E(u, v)$ must also be treated as a tuple for the term $E(X, Z)$. We therefore also send the tuple $E(u, v)$ to all Reduce tasks that correspond to a triple $\big(h(u), y, h(v)\big)$ for any y. Finally, we treat $E(u, v)$ as a tuple of the term $E(Y, Z)$ and send that tuple to all Reduce tasks corresponding to a triple $\big(x, h(u), h(v)\big)$ for any x. The total communication required is thus $3b$ key-value pairs for each of the m tuples of the edge relation E. That is, the minimum communication cost is $O(mb)$ if we use b^3 Reduce tasks.

Next, let us compute the total execution cost at all the Reduce tasks. Assume that the hash function distributes edges sufficiently randomly that the Reduce tasks each get approximately the same number of edges. Since the total

number of edges distributed to the b^3 Reduce tasks is $O(mb)$, it follows that each task receives $O(m/b^2)$ edges. If we use the algorithm of Section 10.6.2 at each Reduce task, the total computation at a task is $O\big((m/b^2)^{3/2}\big)$, or $O(m^{3/2}/b^3)$. Since there are b^3 Reduce tasks, the total computation cost is $O(m^{3/2})$, exactly as for the one-processor algorithm of Section 10.6.2.

10.6.5 Using Fewer Reduce Tasks

By a judicious ordering of the nodes, we can lower the number of reduce tasks by approximately a factor of 6. Think of the "name" of the node i as the pair $\big(h(i), i\big)$, where h is the hash function that we used in Section 10.6.4 to hash nodes to b buckets. Order nodes by their name, considering only the first component (i.e., the bucket to which the node hashes), and only using the second component to break ties among nodes that are in the same bucket.

If we use this ordering of nodes, then the Reduce task corresponding to list of buckets (i, j, k) will be needed only if $i \leq j \leq k$. If b is large, then approximately $1/6$ of all b^3 sequences of integers, each in the range 1 to b, will satisfy these inequalities. For any b, the number of such sequences is $\binom{b+2}{3}$ (see Exercise 10.6.4). Thus, the exact ratio is $(b+2)(b+1)/(6b^2)$.

As there are fewer reducers, we get a substantial decrease in the number of key-value pairs that must be communicated. Instead of having to send each of the m edges to $3b$ Reduce tasks, we need to send each edge to only b tasks. Specifically, consider an edge e whose two nodes hash to i and j; these buckets could be the same or different. For each of the b values of k between 1 and b, consider the list formed from i, j, and k in sorted order. Then the Reduce task that corresponds to this list requires the edge e. But no other Reduce tasks require e.

To compare the communication cost of the method of this section with that of Section 10.6.4, let us fix the number of Reduce tasks, say k. Then the method of Section 10.6.4 hashes nodes to $\sqrt[3]{k}$ buckets, and therefore communicates $3m\sqrt[3]{k}$ key-value pairs. On the other hand, the method of this section hashes nodes to approximately $\sqrt[3]{6k}$ buckets, thus requiring $m\sqrt[3]{6}\sqrt[3]{k}$ communication. Thus, the ratio of the communication needed by the method of Section 10.6.4 to what is needed here is $3/\sqrt[3]{6} = 1.65$.

Example 10.23: Consider the straightforward algorithm of Section 10.6.4 with $b = 6$. That is, there are $b^3 = 216$ Reduce tasks and the communication cost is $3mb = 18m$. We cannot use exactly 216 Reduce tasks with the method of this section, but we can come very close if we choose $b = 10$. Then, the number of Reduce tasks is $\binom{12}{3} = 220$, and the communication cost is $mb = 10m$. That is, the communication cost is 5/9th of the cost of the straightforward method. □

10.6.6 Exercises for Section 10.6

Exercise 10.6.1: How many triangles are there in the graphs:

(a) Figure 10.1.

(b) Figure 10.9.

! (c) Figure 10.2.

Exercise 10.6.2: For each of the graphs of Exercise 10.6.1 determine:

(*i*) What is the minimum degree for a node to be considered a "heavy hitter"?

(*ii*) Which nodes are heavy hitters?

(*iii*) Which triangles are heavy-hitter triangles?

! **Exercise 10.6.3:** In this exercise we consider the problem of finding squares in a graph. That is, we want to find quadruples of nodes a, b, c, d such that the four edges (a, b), (b, c), (c, d), and (a, d) exist in the graph. Assume the graph is represented by a relation E as in Section 10.6.4. It is not possible to write a single join of four copies of E that expresses all possible squares in the graph, but we can write three such joins. Moreover, in some cases, we need to follow the join by a selection that eliminates "squares" where one pair of opposite corners are really the same node. We can assume that node a is numerically lower than its neighbors b and d, but there are three cases,depending on whether c is

(*i*) Also lower than b and d,

(*ii*) Between b and d, or

(*iii*) Higher than both b and d.

(a) Write the natural joins that produce squares satisfying each of the three conditions above. You can use four different attributes W, X, Y, and Z, and assume that there are four copies of relation E with different schemas, so the joins can each be expressed as natural joins.

(b) For which of these joins do we need a selection to assure that opposite corners are really different nodes?

!! (c) Assume we plan to use k Reduce tasks. For each of your joins from (a), into how many buckets should you hash each of W, X, Y, and Z in order to minimize the communication cost?

(d) Unlike the case of triangles, it is not guaranteed that each square is produced only once, although we can be sure that each square is produced by only one of the three joins. For example, a square in which the two nodes at opposite corners are each lower numerically than the each of the other two nodes will only be produced by the join (*i*). For each of the three joins, how many times does it produce each square that it produces at all?

! **Exercise 10.6.4:** Show that the number of sequences of integers $1 \leq i \leq j \leq k \leq b$ is $\binom{b+2}{3}$. *Hint*: show that these sequences can be placed in a 1-to-1 correspondence with the binary strings of length $b+2$ having exactly three 1's.

10.7 Neighborhood Properties of Graphs

There are several important properties of graphs that relate to the number of nodes one can reach from a given node along a short path. In this section we look at algorithms for solving problems about paths and neighborhoods for very large graphs. In some cases, exact solutions are not feasible for graphs with millions of nodes. We therefore look at approximation algorithms as well as exact algorithms.

10.7.1 Directed Graphs and Neighborhoods

In this section we shall use a directed graph as a model of a network. A *directed graph* has a set of nodes and a set of *arcs*; the latter is a pair of nodes written $u \to v$. We call u the *source* and v the *target* of the arc. The arc is said to be *from u to v*.

Many kinds of graphs can be modeled by directed graphs. The Web is a major example, where the arc $u \to v$ is a link from page u to page v. Or, the arc $u \to v$ could mean that telephone subscriber u has called subscriber v in the past month. For another example, the arc could mean that individual u is following individual v on Twitter. In yet another graph, the arc could mean that research paper u references paper v.

Moreover, all undirected graphs can be represented by directed graphs. Instead of the undirected edge (u, v), use two arcs $u \to v$ and $v \to u$. Thus, the material of this section also applies to graphs that are inherently undirected, such as a friends graph in a social network.

A *path* in a directed graph is a sequence of nodes $v_0, v_1, \ldots, v_k$ such that there are arcs $v_1 \to v_{i+1}$ for all $i = 0, 1, \ldots, k-1$. The *length* of this path is k, the number of arcs along the path. Note that there are $k+1$ nodes in a path of length k, and a node by itself is considered a path of length 0.

The *neighborhood of radius d* for a node v is the set of nodes u for which there is a path of length at most d from v to u. We denote this neighborhood by $N(v, d)$. For example, $N(v, 0)$ is always $\{v\}$, and $N(v, 1)$ is v plus the set of nodes to which there is an arc from v. More generally, if V is a set of nodes, then $N(V, d)$ is the set of nodes u for which there is a path of length d or less from at least one node in the set V.

The *neighborhood profile* of a node v is the sequence of sizes of its neighborhoods $|N(v, 1)|, |N(v, 2)|, \ldots$. We do not include the neighborhood of distance 0, since its size is always 1.

Example 10.24: Consider the undirected graph of Fig. 10.1, which we reproduce here as Fig. 10.21. To turn it into a directed graph, think of each edge

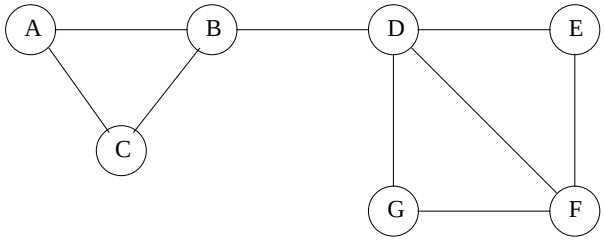

Figure 10.21: Our small social network; think of it as a directed graph

as a pair of arcs, one in each direction. For instance, the edge (A, B) becomes the arcs $A \to B$ and $B \to A$. First, consider the neighborhoods of node A. We know $N(A, 0) = \{A\}$. $N(A, 1) = \{A, B, C\}$, since there are arcs from A only to B and C. $N(A, 2) = \{A, B, C, D\}$ and $N(A, 3) = \{A, B, C, D, E, F, G\}$. Neighborhoods for larger radius are all the same as $N(A, 3)$.

On the other hand, consider node B. We find $N(B, 0) = \{B\}$, $N(B, 1) = \{A, B, C, D\}$, and $N(B, 2) = \{A, B, C, D, E, F, G\}$. We know that B is more central to the network than A, and this fact is reflected by the neighborhood profiles of the two nodes. Node A has profile $3, 4, 7, 7, \ldots$, while B has profile $4, 7, 7, \ldots$. Evidently, B is more central than A, because at every distance, its neighborhood is at least as large as that of A. In fact, D is even more central than B, because its neighborhood profile $5, 7, 7, \ldots$ dominates the profile of each of the nodes. □

10.7.2 The Diameter of a Graph

The *diameter* of a directed graph is the smallest integer d such that for every two nodes u and v there is a path of length d or less from u to v. In a directed graph, this definition only makes sense if the graph is *strongly connected*; that is, there is a path from any node to any other node. Recall our discussion of the Web in Section 5.1.3, where we observed that there is a large strongly connected subset of the Web in the "center," but that the Web as a whole is not strongly connected. Rather, there are some pages that reach nowhere by links, and some pages that cannot be reached by links.

If the graph is undirected, the definition of diameter is the same as for directed graphs, but the path may traverse the undirected edges in either direction. That is, we treat an undirected edge as a pair of arcs, one in each direction. The notion of diameter makes sense in an undirected graph as long as that graph is connected.

Example 10.25: For the graph of Fig. 10.21, the diameter is 3. There are some pairs of nodes, such as A and E, that have no path of length less than 3. But every pair of nodes have a path from one to the other with length at most 3. □

Six Degrees of Separation

There is a famous game called "six degrees of Kevin Bacon," the object of which is to find paths of length at most six in the graph whose nodes are movie stars and whose edges connect stars that played in the same movie. The conjecture is that in this graph, no movie star is of distance more than six from Kevin Bacon. More generally, any two movie stars can be connected by a path of length at most six; i.e., the diameter of the graph is six. A small diameter makes computation of neighborhoods more efficient, so it would be nice if all social-network graphs exhibited a similar small diameter. In fact, the phrase "six degrees of separation," refers to the conjecture that in the network of all people in the world, where an edge means that the two people know each other, the diameter is six. Unfortunately, as we shall discuss in Section 10.7.3, not all important graphs exhibit such tight connections.

We can compute the diameter of a graph by computing the sizes of its neighborhoods of increasing radius, until at some radius we fail to add any more nodes. That is, for each node v, find the smallest d such that $|N(v,d)| = |N(v,d+1)|$. This d is the tight upper bound on the length of the shortest path from v to any node it can reach. Call it $d(v)$. For instance, we saw from Example 10.24 that $d(A) = 3$ and $d(B) = 2$. If there is any node v such that $|N\big(v,d(v)\big)|$ is not the number of nodes in the entire graph, then the graph is not strongly connected, and we cannot offer any finite integer as its diameter. However, if the graph is strongly connected, then the diameter of the graph is $\max_v\big(d(v)\big)$.

The reason this computation works is that one way to express $N(v,d+1)$ is the union of $N(v,d)$ and the set of all nodes w such that for some u in $N(v,d)$ there is an arc $u \to w$. That is, we start with $N(v,d)$ and add to it the targets of all arcs that have a source in $N(v,d)$. If all the arcs with source in $N(v,d)$ are already in $N(v,d)$, then not only is $N(v,d+1)$ equal to $N(v,d)$, but all of $N(v,d+2), N(v,d+3), \ldots$ will equal $N(v,d)$. Finally, we observe that since $N(v,d) \subseteq N(v,d+1)$ the only way $|N(v,d)|$ can equal $|N(v,d+1)|$ is for $N(v,d)$ and $N(v,d+1)$ to be the same set. Thus, if d is the smallest integer such that $|N(v,d)| = |N(v,d+1)|$, it follows that every node v can reach is reached by a path of length at most d.

10.7.3 Transitive Closure and Reachability

The *transitive closure* of a graph is the set of pairs of nodes (u,v) such that there is a path from u to v of length zero or more. We shall sometimes write

this assertion as path(u, v).[4] A related concept is that of *reachability*. We say node u *reaches* node v if path(u, v). The problem of computing the transitive closure is to find all pairs of nodes u and v in a graph for which path(u, v) is true. The reachability problem is, given a node u in the graph, find all v such that path(u, v) is true.

These two concepts relate to the notions of neighborhoods that we have seen earlier. In fact, path(u, v) is true if and only if v is in $N(u, \infty)$, which we define to be $\cup_{i \geq 0}\ N(u, i)$. Thus, the reachability problem is to compute the union of all the neighborhoods of any radius for a given node u. The discussion in Section 10.7.2 reminds us that we can compute the reachable set for u by computing its neighborhoods up to that smallest radius d for which $N(u, d) = N(u, d + 1)$.

The two problems – transitive closure and reachability – are related, but there are many examples of graphs where reachability is feasible and transitive closure is not. For instance, suppose we have a Web graph of a billion nodes. If we want to find the pages (nodes) reachable from a given page, we can do so, even on a single machine with a large main memory. However, just to produce the transitive closure of the graph could involve 10^{18} pairs of nodes, which is not practical, even using a large cluster of computers.[5]

10.7.4 Transitive Closure Via MapReduce

When it comes to parallel implementation, transitive closure is actually more readily parallelizable than is reachability. If we want to compute $N(v, \infty)$, the set of nodes reachable from node v, without computing the entire transitive closure, we have no option but to compute the sequence of neighborhoods, which is essentially a breadth-first search of the graph from v. In relational terms, suppose we have a relation $Arc(X, Y)$ containing those pairs (x, y) such that there is an arc $x \to y$. We want to compute iteratively a relation $Reach(X)$ that is the set of nodes reachable from node v. After i rounds, $Reach(X)$ will contain all those nodes in $N(v, i)$.

Initially, $Reach(X)$ contains only the node v. Suppose it contains all the nodes in $N(v, i)$ after some round of MapReduce. To construct $N(v, i + 1)$ we need to join *Reach* with the *Arc* relation, then project onto the second component and perform a union of the result with the old value of *Reach*. In SQL terms, we perform

```
SELECT DISTINCT Arc.Y
FROM Reach, Arc
```

[4] Technically, this definition gives us the *reflexive and transitive closure* of the graph, since path(v, v) is always considered true, even if there is no cycle that contains v.

[5] While we cannot compute the transitive closure completely, we can still learn a great deal about the structure of a graph, provided there are large strongly connected components. For example, the Web graph experiments discussed in Section 5.1.3 were done on a graph of about 200 million nodes. Although they never listed all the pairs of nodes in the transitive closure, they were able to describe the structure of the Web.

```
WHERE Arc.X = Reach.X;
```

This query asks us to compute the natural join of $Reach(X)$ and $Arc(X, Y)$, which we can do by MapReduce as explained in Section 2.3.7. Then, we have to group the result by Y and eliminate duplicates, a process that can be done by another MapReduce job as in Section 2.3.8.

How many rounds this process requires depends on how far from v is the furthest node that v can reach. In many social-network graphs, the diameter is small, as discussed in the box on "Six Degrees of Separation." If so, computing reachability in parallel, using MapReduce or another approach is feasible. Few rounds of computation will be needed and the space requirements are not greater than the space it takes to represent the graph.

However, there are some graphs where the number of rounds is a serious impediment. For instance, in a typical portion of the Web, it has been found that most pages reachable from a given page are reachable by paths of length 10–15. Yet there are some pairs of pages such that the first reaches the second, but only through paths whose length is measured in the hundreds. For instance, blogs are sometimes structured so each response is reachable only through the comment to which it responds. Running arguments lead to long paths with no way to 'shortcut" around that path. Or a tutorial on the Web, with 50 chapters, may be structured so you can only get to Chapter i through the page for Chapter $i - 1$.

Interestingly, the transitive closure can be computed much faster in parallel than can strict reachability. By a recursive-doubling technique, we can double the length of paths we know about in a single round. Thus, on a graph of diameter d, we need only $\log_2 d$ rounds, rather than d rounds. If $d = 6$, the difference is not important, but if $d = 1000$, $\log_2 d$ is about 10, so there is a hundredfold reduction in the number of rounds. The problem, as discussed above, is that while we can compute the transitive closure quickly, we must compute many more facts than are needed for a reachability computation on the same graph, and therefore the space requirements for transitive closure can greatly exceed the space requirements for reachability. That is, if all we want is the set $Reach(v)$, we can compute the transitive closure of the entire graph, and then throw away all pairs that do not have v as their first component. But we cannot throw away all those pairs until we are done. During the computation of the transitive closure, we could wind up computing many facts $Path(x, y)$, where neither x nor y is reachable from v, and even if they are reachable from v, we may not need to know x can reach y.

Assuming the graph is small enough that we can compute the transitive closure in its entirety, we still must be careful how we do so using MapReduce or another parallelism approach. The simplest recursive-doubling approach is to start the the relation $Path(X, Y)$ equal to the arc relation $Arc(X, Y)$. Suppose that after i rounds, $Path(X, Y)$ contains all pairs (x, y) such that there is a path from x to y of length at most 2^i. Then if we join $Path$ with itself at the next round, we shall discover all those pairs (x, y) such that there is a path

from x to y of length at most $2 \times 2^i = 2^{i+1}$. The recursive-doubling query in SQL is

```
SELECT DISTINCT p1.X, p2.Y
FROM Path p1, Path p2
WHERE p1.Y = p2.X;
```

After computing this query, we get all pairs connected by a path of length between 2 and 2^{i+1}, assuming $Path$ contains pairs connected by paths of length between 1 and 2^i. If we take the union of the result of this query with the Arc relation itself, then we get all paths of length between 1 and 2^{i+1} and can use the union as the $Path$ relation in the next round of recursive doubling. The query itself can be implemented by two MapReduce jobs, one to do the join and the other to do the union and eliminate duplicates. As we observed for the parallel reachability computation, the methods of Sections 2.3.7 and 2.3.8 suffice. The union, discussed in Section 2.3.6 doesn't really require a MapReduce job of its own; it can be combined with the duplicate-elimination.

If a graph has diameter d, then after $\log_2 d$ rounds of the above algorithm $Path$ contains all pairs (x, y) connected by a path of length at most d; that is, it contains all pairs in the transitive closure. Unless we already know d, one more round will be necessary to verify that no more pairs can be found, but for large d, this process takes many fewer rounds than the breadth-first search that we used for reachability.

However, the above recursive-doubling method does a lot of redundant work. An example should make the point clear.

Example 10.26: Suppose the shortest path from x_0 to x_{17} is of length 17; in particular, let there be a path $x_0 \to x_1 \to \cdots \to x_{17}$. We shall discover the fact $Path(x_0, x_{17})$ on the fifth round, when $Path$ contains all pairs connected by paths of length up to 16. The same path from x_0 to x_{17} will be discovered 16 times when we join $Path$ with itself. That is, we can take the fact $Path(x_0, x_{16})$ and combine it with $Path(x_{16}, x_{17})$ to obtain $Path(x_0, x_{17})$. Or we can combine $Path(x_0, x_{15})$ with $Path(x_{15}, x_{17})$ to discover the same fact, and so on. □

10.7.5 Smart Transitive Closure

A variant of recursive doubling that avoids discovering the same path more than once is called *smart* transitive closure. Every path of length greater than 1 can be broken into a *head* whose length is a power of 2, followed by a *tail* whose length is no greater than the length of the head.

Example 10.27: A path of length 13 has a head consisting of the first 8 arcs, followed by a tail consisting of the last 5 arcs. A path of length 2 is a head of length 1 followed by a tail of length 1. Note that 1 is a power of 2 (the 0th power), and the tail will be as long as the head whenever the path itself has a length that is a power of 2. □

To implement smart transitive closure in SQL, we introduce a relation $Q(X, Y)$ whose function after the ith round is to hold all pairs of nodes (x, y) such that the shortest path from x to y is of length exactly 2^i. Also, after the ith round, $Path(x, y)$ will be true if the shortest path from x to y is of length at most $2^{i+1} - 1$. Note that this interpretation of $Path$ is slightly different from the interpretation of $Path$ in the simple recursive-doubling method given in Section 10.7.4.

Initially, set both Q and $Path$ to be copies of the relation Arc. After the ith round, assume that Q and $Path$ have the contents described in the previous paragraph. Note that for the round $i = 1$, the initial values of Q and $Path$ initially satisfy the conditions as described for $i = 0$. On the $(i + 1)$st round, we do the following:

1. Compute a new value for Q by joining it with itself, using the SQL query

```
SELECT DISTINCT q1.X, q2.Y
FROM Q q1, Q q2
WHERE q1.Y = q2.X;
```

2. Subtract $Path$ from the relation Q computed in step (1). Note that step (1) discovers all paths of length 2^{i+1}. But some pairs connected by these paths may also have shorter paths. The result of step (2) is to leave in Q all and only those pairs (u, v) such that the shortest path from u to v has length exactly 2^{i+1}.

3. Join $Path$ with the new value of Q computed in (2), using the SQL query

```
SELECT DISTINCT Q.X, Path.Y
FROM Q, Path
WHERE Q.Y = Path.X
```

   At the beginning of the round $Path$ contains all (y, z) such that the shortest path from y to z has a length up to $2^{i+1} - 1$ from y to z, and the new value of Q contains all pairs (x, y) for which the shortest path from x to y is of length 2^{i+1}. Thus, the result of this query is the set of pairs (x, y) such that the shortest path from x to y has a length between $2^{i+1} + 1$ and $2^{i+2} - 1$, inclusive.

4. Set the new value of $Path$ to be the union of the relation computed in step (3), the new value of Q computed in step (1), and the old value of $Path$. These three terms give us all pairs (x, y) whose shortest path is of length $2^{i+1} + 1$ through $2^{i+2} - 1$, exactly 2^{i+1}, and 1 through $2^{i+1} - 1$, respectively. Thus, the union gives us all shortest paths up to length $2^{i+2} - 1$, as required by the inductive hypothesis about what is true after each round.

Path Facts Versus Paths

We should be careful to distinguish between a path, which is a sequence of arcs, and a path fact, which is a statement that there exists a path from some node x to some node y. The path fact has been shown typically as $Path(x, y)$. Smart transitive closure discovers each path only once, but it may discover a path fact more than once. The reason is that often a graph will have many paths from x to y, and may even have many different paths from x to y that are of the same length.

Not all paths are discovered independently by smart transitive closure. For instance, if there are arcs $w \to x \to y \to z$ and also arcs $x \to u \to z$, then the path fact $Path(w, z)$ will be discovered twice, once by combining $Path(w, y)$ with $Path(y, z)$ and again when combining $Path(w, u)$ with $Path(u, z)$. On the other hand, if the arcs were $w \to x \to y \to z$ and $w \to v \to y$, then $Path(w, z)$ would be discovered only once, when combining $Path(w, y)$ with $Path(y, z)$.

Each round of the smart transitive-closure algorithm uses steps that are joins, aggregations (duplicate eliminations), or unions. A round can thus be implemented as a short sequence of MapReduce jobs. Further, a good deal of work can be saved if these operations can be combined, say by using the more general patterns of communication permitted by a workflow system (see Section 2.4.1).

10.7.6 Transitive Closure by Graph Reduction

A typical directed graph such as the Web contains many strongly connected components (SCC's). We can collapse an SCC to a single node as far as computing the transitive closure is concerned, since all the nodes in an SCC reach exactly the same nodes. There is an elegant algorithm for finding the SCC's of a graph in time linear in the size of the graph, due to J.E. Hopcroft and R.E. Tarjan. However, this algorithm is inherently sequential, based on depth-first search, and so not well suited to parallel impelementation on large graphs.

We can find most of the SCC's in a graph by some random node selections followed by two breadth-first searches. Moreover, the larger an SCC is, the more likely it is to be collapsed early, thus reducing the size of the graph quickly. The algorithm for reducing SCC's to single nodes is as follows. Let G be the graph to be reduced, and let G' be G with all the arcs reversed.

1. Pick a node v from G at random.
2. Find $N_G(v, \infty)$, the set of nodes reachable from v in G.
3. Find $N_{G'}(v, \infty)$, the set of nodes that v reaches in the graph G' that has the arcs of G reversed. Equivalently, this set is those nodes that reach v

in G.

4. Construct the SCC S containing v, which is $N_G(v, \infty) \cap N_{G'}(v, \infty)$. That is, v and u are in the same SCC of G if and only if v can reach u and u can reach v.

5. Replace SCC S by a single node s in G. To do so, delete all nodes in S from G and add s to the node set of G. Delete from G all arcs one or both ends of which are in S. Then, add to the arc set of G an arc $s \to x$ whenever there was an arc in G from any member of S to x. Finally, add an arc $x \to s$ if there was an arc from x to any member of S.

We can iterate the above steps a fixed number of times. We can instead iterate until the graph becomes sufficiently small, or we could examine all nodes v in turn and not stop until each node is in an SCC by itself; i.e.,

$$N_G(v, \infty) \cap N_{G'}(v, \infty) = \{v\}$$

for all remaining nodes v. If we make the latter choice, the resulting graph is called the *transitive reduction* of the original graph G. The transitive reduction is always acyclic, since if it had a cycle there would remain an SCC of more than one node. However, it is not necessary to reduce to an acyclic graph, as long as the resulting graph has sufficiently few nodes that it is feasible to compute the full transitive closure of this graph; that is, the number of nodes is small enough that we can deal with a result whose size is proportional to the square of that number of nodes.

While the transitive closure of the reduced graph is not exactly the same as the transitive closure of the original graph, the former, plus the information about what SCC each original node belongs to is enough to tell us anything that the transitive closure of the original graph tells us. If we want to know whether $Path(u, v)$ is true in the original graph, find the SCC's containing u and v. If one or both of these nodes have never been combined into an SCC, then treat that node as an SCC by itself. If u and v belong to the same SCC, then surely u can reach v. If they belong to different SCC's s and t, respectively, determine whether s reaches t in the reduced graph. If so, then u reaches v in the original graph, and if not, then not.

Example 10.28: Let us reconsider the "bowtie" picture of the Web from Section 5.1.3. The number of nodes in the part of the graph examined was over 200 million; surely too large to deal with data of size proportional to the square of that number. There was one large set of nodes called "the SCC" that was regarded as the center of the graph. Since about one node in four was in this SCC, it would be collapsed to a single node as soon as any one of its members was chosen at random. But there are many other SCC's in the Web, even though they were not shown explicitly in the "bowtie." For instance, the in-component would have within it many large SCC's. The nodes in one of these SCC's can reach each other, and can reach some of the other nodes in the

in-component, and of course they can reach all the nodes in the central SCC. The SCC's in the in- and out-components, the tubes, and other structures can all be collapsed, leading to a far smaller graph. □

10.7.7 Approximating the Sizes of Neighborhoods

In this section we shall take up the problem of computing the neighborhood profile for each node of a large graph. A variant of the problem, which yields to the same technique, is to find the size of the reachable set for each node v, the set we have called $N(v, \infty)$. Recall that for a graph of a billion nodes, it is totally infeasible to compute the neighborhoods for each node, even using a very large cluster of machines. However, even if we want only a count of the nodes in each neighborhood, we need to remember the nodes found so far as we explore the graph, or else we shall not know whether a new node found is one we have seen already.

On the other hand, it is not so hard to find an approximation to the size of each neighborhood, using the Flajolet-Martin technique discussed in Section 4.4.2. Recall that this technique uses some large number of hash functions; in this case, the hash functions are applied to the nodes of the graph. The important property of the bit string we get when we apply hash function h to node v is the "tail length" – the number of 0's at the end of the string. For any set of nodes, an estimate of the size of the set is 2^R, where R is the length of the longest tail for any member of the set. Thus, instead of storing all the members of the set, we can instead record only the value of R for that set. Of course, there are many hash functions, so we need to record values of R for each hash function.

Example 10.29: If we use a hash function that produces a 64-bit string, then six bits are all that are needed to store each value of R. For instance, if there are a billion nodes, and we want to estimate the size of the neighborhood for each, we can store the value of R for 20 hash functions for each node in 15 gigabytes. □

If we store tail lengths for each neighborhood, we can use this information to compute estimates for the larger neighborhoods from our estimates for the smaller neighborhoods. That is, suppose we have computed our estimates for $|N(v, d)|$ for all nodes v, and we want to compute estimates for the neighborhoods of radius $d + 1$. For each hash function h, the value of R for $N(v, d + 1)$ is the largest of:

1. The tail of v itself and
2. The values of R associated with h and $N(u, d)$, where $v \to u$ is an arc of the graph.

Notice that it doesn't matter whether a node is reachable through only one successor of v in the graph, or through many different successors. We get

the same estimate in either case. This useful property was the same one we exploited in Section 4.4.2 to avoid having to know whether a stream element appeared once or many times in the stream.

We shall now describe the complete algorithm, called *ANF* (Approximate Neighborhood Function). We choose K hash functions $h_1, h_2, \ldots, h_k$. For each node v and radius d, let $R_i(v,d)$ denote the maximum tail length of any node in $N(v,d)$ using hash function h_i. To initialize, let $R_i(v,0)$ be the length of the tail of $h_i(v)$, for all i and v.

For the inductive step, suppose we have computed $R_i(v,d)$ for all i and v. Initialize $R_i(v,d+1)$ to be $R_i(v,d)$, for all i and v. Then, consider all arcs $x \to y$ in the graph, in any order. For each $x \to y$, set $R_i(x,d+1)$ to the larger of its current value and $R_i(y,d)$. Observe that the fact we can consider the arcs in any order can provide a big speedup in the case that we can store the R_i's in main memory, but the set of arcs is so large it must be stored on disk. We can stream all the disk blocks containing arcs one at a time, thus using only one disk access per iteration per disk block used for arc storage. This advantage is similar to the one we observed in Section 6.2.1, where we pointed out how frequent-itemset algorithms like A-priori could take advantage of reading market-basket data in a stream, where each disk block was read only once for each round.

To estimate the size of $N(v,d)$, combine the values of the $R_i(v,d)$ for $i = 1, 2, \ldots, k$, as discussed in Section 4.4.3. That is, group the R's into small groups, take the average, and take the median of the averages.

Another improvement to the ANF Algorithm can be had if we are only interested in estimating the sizes of the reachable sets, $N(v,\infty)$. Then, it is not necessary to save $R_i(v,d)$ for different radii d. We can maintain one value $R_i(v)$ for each hash function h_i and each node v. When on any round, we consider arc $x \to y$, we simply assign

$$R_i(x) := \max\big(R_i(x), R_i(y)\big)$$

We can stop the iteration when at some round no value of any $R_i(v)$ changes. Or if we know d is the diameter of the graph, we can just iterate d times.

10.7.8 Exercises for Section 10.7

Exercise 10.7.1 : For the graph of Fig. 10.9, which we repeat here as Fig. 10.22:

(a) If the graph is represented as a directed graph, how many arcs are there?

(b) What are the neighborhood profiles for nodes A and B?

(c) What is the diameter of the graph?

(d) How many pairs are in the transitive closure? *Hint*: Do not forget that there are paths of length greater than zero from a node to itself in this graph.

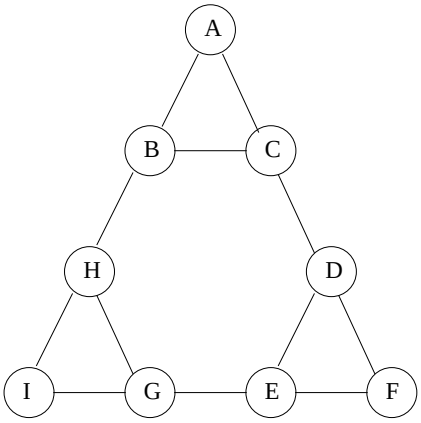

Figure 10.22: A graph for exercises on neighborhoods and transitive closure

(e) If we compute the transitive closure by recursive doubling, how many rounds are needed?

Exercise 10.7.2: The smart transitive closure algorithm breaks paths of any length into head and tail of specific lengths. What are the head and tail lengths for paths of length 7, 8, and 9?

Exercise 10.7.3: Consider the running example of a social network, last shown in Fig. 10.21. Suppose we use one hash function h which maps each node (capital letter) to its ASCII code. Note that the ASCII code for A is 01000001, and the codes for $B, C, \ldots$ are sequential, $01000010, 01000011, \ldots$.

(a) Using this hash function, compute the values of R for each node and radius 1. What are the estimates of the sizes of each neighborhood? How do the estimates compare with reality?

(b) Next, compute the values of R for each node and radius 2. Again compute the estimates of neighborhood sizes and compare with reality.

(c) The diameter of the graph is 3. Compute the value of R and the size estimate for the set of reachable nodes for each of the nodes of the graph.

(d) Another hash function g is one plus the ASCII code for a letter. Repeat (a) through (c) for hash function g. Take the estimate of the size of a neighborhood to be the average of the estimates given by h and g. How close are these estimates?

10.8 Summary of Chapter 10

- *Social-Network Graphs*: Graphs that represent the connections in a social network are not only large, but they exhibit a form of locality, where small

subsets of nodes (communities) have a much higher density of edges than the average density.

- *Communities and Clusters*: While communities resemble clusters in some ways, there are also significant differences. Individuals (nodes) normally belong to several communities, and the usual distance measures fail to represent closeness among nodes of a community. As a result, standard algorithms for finding clusters in data do not work well for community finding.

- *Betweenness*: One way to separate nodes into communities is to measure the betweenness of edges, which is the sum over all pairs of nodes of the fraction of shortest paths between those nodes that go through the given edge. Communities are formed by deleting the edges whose betweenness is above a given threshold.

- *The Girvan-Newman Algorithm*: The Girvan-Newman Algorithm is an efficient technique for computing the betweenness of edges. A breadth-first search from each node is performed, and a sequence of labeling steps computes the share of paths from the root to each other node that go through each of the edges. The shares for an edge that are computed for each root are summed to get the betweenness.

- *Communities and Complete Bipartite Graphs*: A complete bipartite graph has two groups of nodes, all possible edges between pairs of nodes chosen one from each group, and no edges between nodes of the same group. Any sufficiently dense community (a set of nodes with many edges among them) will have a large complete bipartite graph.

- *Finding Complete Bipartite Graphs*: We can find complete bipartite graphs by the same techniques we used for finding frequent itemsets. Nodes of the graph can be thought of both as the items and as the baskets. The basket corresponding to a node is the set of adjacent nodes, thought of as items. A complete bipartite graph with node groups of size t and s can be thought of as finding frequent itemsets of size t with support s.

- *Graph Partitioning*: One way to find communities is to partition a graph repeatedly into pieces of roughly similar sizes. A cut is a partition of the nodes of the graph into two sets, and its size is the number of edges that have one end in each set. The volume of a set of nodes is the number of edges with at least one end in that set.

- *Normalized Cuts*: We can normalize the size of a cut by taking the ratio of the size of the cut and the volume of each of the two sets formed by the cut. Then add these two ratios to get the normalized cut value. Normalized cuts with a low sum are good, in the sense that they tend to divide the nodes into two roughly equal parts, and have a relatively small size.

- *Adjacency Matrices*: These matrices describe a graph. The entry in row i and column j is 1 if there is an edge between nodes i and j, and 0 otherwise.

- *Degree Matrices*: The degree matrix for a graph has d in the ith diagonal entry if d is the degree of the ith node. Off the diagonal, all entries are 0.

- *Laplacian Matrices*: The Laplacian matrix for a graph is its degree matrix minus its adjacency matrix. That is, the entry in row i and column i of the Laplacian matrix is the degree of the ith node of the graph, and the entry in row i and column j, for $i \neq j$, is -1 if there is an edge between nodes i and j, and 0 otherwise.

- *Spectral Method for Partitioning Graphs*: The lowest eigenvalue for any Laplacian matrix is 0, and its corresponding eigenvector consists of all 1's. The eigenvectors corresponding to small eigenvalues can be used to guide a partition of the graph into two parts of similar size with a small cut value. For one example, putting the nodes with a positive component in the eigenvector with the second-smallest eigenvalue into one set and those with a negative component into the other is usually good.

- *Simrank*: One way to measure the similarity of nodes in a graph with several types of nodes is to start a random walker at one node and allow it to wander, with a fixed probability of restarting at the same node. The distribution of where the walker can be expected to be is a good measure of the similarity of nodes to the starting node. This process must be repeated with each node as the starting node if we are to get all-pairs similarity.

- *Triangles in Social Networks*: The number of triangles per node is an important measure of the closeness of a community and often reflects its maturity. We can enumerate or count the triangles in a graph with m edges in $O(m^{3/2})$ time, but no more efficient algorithm exists in general.

- *Traingle Finding by MapReduce*: We can find triangles in a single round of MapReduce by treating it as a three-way join. Each edge must be sent to a number of reducers proportional to the cube root of the total number of reducers, and the total computation time spent at all the reducers is proportional to the time of the serial algorithm for triangle finding.

- *Neighborhoods*: The neighborhood of radius d for a node v in a directed or undirected graph is the set of nodes reachable from v along paths of length at most d. The neighborhood profile of a node is the sequence of neighborhood sizes for all distances from 1 upwards. The diameter of a connected graph is the smallest d for which the neighborhood of radius d for any starting node includes the entire graph.

- *Transitive Closure*: A node v can reach node u if u is in the neighborhood of v for some radius. The transitive closure of a graph is the set of pairs of nodes (v, u) such that v can reach u.

- *Computing Transitive Closure*: Since the transitive closure can have a number of facts equal to the square of the number of nodes of a graph, it is infeasible to compute transitive closure directly for large graphs. One approach is to find strongly connected components of the graph and collapse them each to a single node before computing the transitive closure.

- *Transitive Closure and MapReduce*: We can view transitive closure computation as the iterative join of a path relation (pairs of nodes v and u such that u is known to be reachable from v) and the arc relation of the graph. Such an approach requires a number of MapReduce rounds equal to the diameter of the graph.

- *Transitive Closure by Recursive Doubling*: An approach that uses fewer MapReduce rounds is to join the path relation with itself at each round. At each round, we double the length of paths that are able to contribute to the transitive closure. Thus, the number of needed rounds is only the base-2 logarithm of the diameter of the graph.

- *Smart Transitive Closure*: While recursive doubling can cause the same path to be considered many times, and thus increases the total computation time (compared with iteratively joining paths with single arcs), a variant called smart transitive closure avoids discovering the same path more than once. The trick is to require that when joining two paths, the first has a length that is a power of 2.

- *Approximating Neighborhood Sizes*: By using the Flajolet-Martin technique for approximating the number of distinct elements in a stream, we can find the neighborhood sizes at different radii approximately. We maintain a set of tail lengths for each node. To increase the radius by 1, we examine each edge (u, v) and for each tail length for u we set it equal to the corresponding tail length for v if the latter is larger than the former.

10.9 References for Chapter 10

Simrank comes from [8]. An alternative approach in [11] views similarity of two nodes as the propability that random walks from the two nodes will be at the same node. [3] combines random walks with node classification to predict links in a social-network graph. [16] looks at the efficiency of computing simrank as a personalized PageRank.

The Girvan-Newman Algorithm is from [6]. Finding communities by searching for complete bipartite graphs appears in [9].

Normalized cuts for spectral analysis were introduced in [13]. [19] is a survey of spectral methods for finding clusters, and [5] is a more general survey of community finding in graphs. [10] is an analysis of communities in many networks encountered in practice.

Counting triangles using MapReduce was discussed in [15]. The method descirbed here is from [1], which also gives a technique that works for any subgraph. [17] discusses randomized algorithms for triangle finding.

The ANF Algorithm was first investigated in [12]. [4] gives an additional speedup to ANF.

The Smart Transitive-Closure Algorithm was discovered by [7] and [18] independently. Implementation of transitive closure using MapReduce or similar systems is discussed in [2].

An open-source C++ implementation of many of the algorithms described in this chapter can be found in the SNAP library [14].

1. F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances by map-reduce,"

`http://ilpubs.stanford.edu:8090/1020`

2. F.N. Afrati and J.D. Ullman, "Transitive closure and recursive Datalog implemented on clusters," in *Proc. EDBT* (2012).

3. L. Backstrom and J. Leskovec, "Supervised random walks: predicting and recommending links in social networks," *Proc. Fourth ACM Intl. Conf. on Web Search and Data Mining* (2011), pp. 635–644.

4. P. Boldi, M. Rosa, and S. Vigna, "HyperANF: approximating the neighbourhood function of very large graphs on a budget," *Proc. WWW Conference* (2011), pp. 625–634.

5. S. Fortunato, "Community detection in graphs," *Physics Reports* **486**:3–5 (2010), pp. 75–174.

6. M. Girvan and M.E.J. Newman, "Community structure in social and biological networks," *Proc. Natl. Acad. Sci.* **99** (2002), pp. 7821–7826.

7. Y.E. Ioannidis, "On the computation of the transitive closure of relational operators," *Proc. 12th Intl. Conf. on Very Large Data Bases*, pp. 403–411.

8. G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," *Proceedings of the eighth ACM SIGKDD international conference on Knowledge Discovery and Data Mining* (2002), pp. 538–543.

9. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tompkins, "Trawling the Web for emerging cyber-communities, *Computer Networks* **31**:11–16 (May, 1999), pp. 1481–1493.

10. J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney, "Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters," `http://arxiv.org/abs/0810.1355`.

11. S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: a versatile graph matching algorithm and its application to schema matching, *Proc. Intl. Conf. on Data Engineering* (2002), pp. 117–128.

12. C.R. Palmer, P.B. Gibbons, and C. Faloutsos, "ANF: a fast and scalable tool for data mining in massive graphs," *Proc. Eighth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (2002), pp. 81–90.

13. J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. on Pattern Analysis and Machine Intelligence*," **22**:8 (2000), pp. 888–905.

14. Stanford Network Analysis Platform, `http://snap.stanford.edu`.

15. S. Suri and S. Vassilivitskii, "Counting triangles and the curse of the last reducer," *Proc. WWW Conference* (2011).

16. H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," *ICDM* 2006, pp. 613–622.

17. C.E. Tsourakakis, U. Kang, G.L. Miller, and C. Faloutsos, "DOULION: counting triangles in massive graphs with a coin," *Proc. Fifteenth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (2009).

18. P. Valduriez and H. Boral, "Evaluation of recursive queries using join indices," *Expert Database Conf.* (1986), pp. 271–293.

19. U. von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing* bf17:4 (2007), 2007, pp. 395–416.