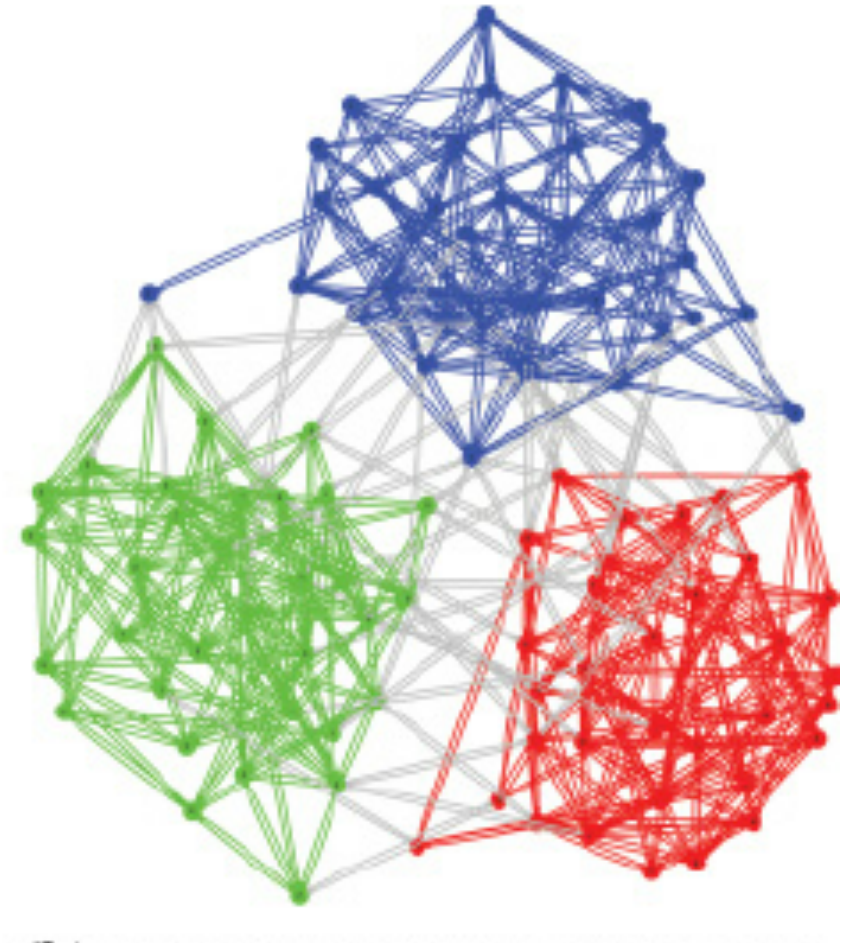
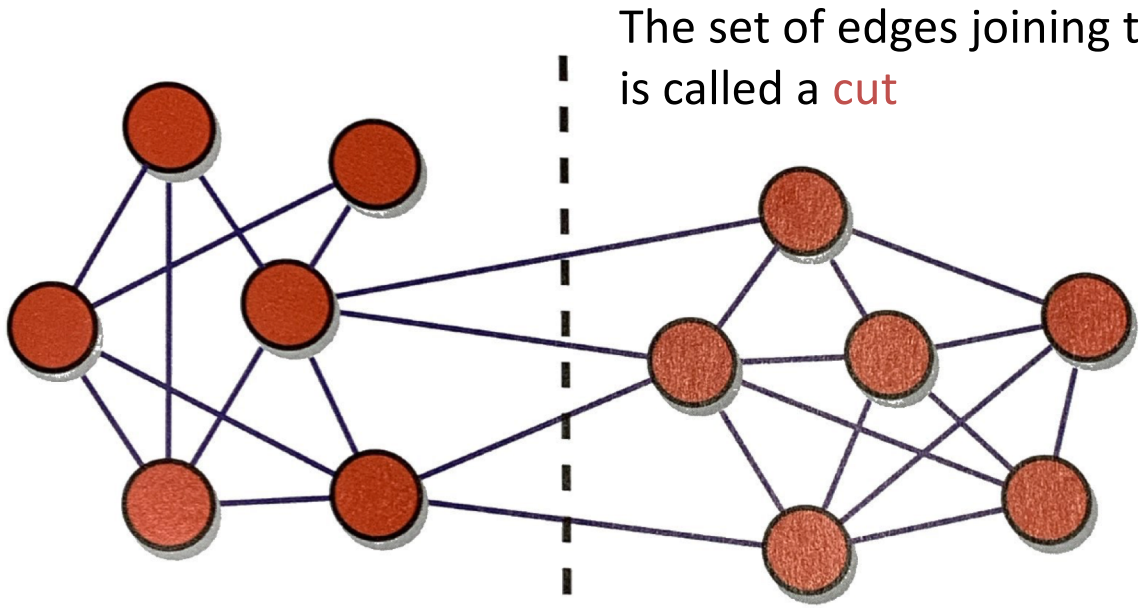
Network Partitioning and Community Detection

Network Partitioning



Goal of Partitioning: Optimally Separate Groups of Nodes

The partitioning problem consists of finding a division of a network into a given number of subnetworks, or clusters, of given sizes, such that the total number of links connecting nodes in different clusters is minimized.



Partitioning seeks to find the way to group nodes so that the cut joining the groups is minimized. Often called the minimum cut problem.

Kernighan-Lin Algorithm for Graph Bisection

The goal here is to divide the network into two pieces of equal size (plus 1 if node number is odd), called graph bisection. The algorithm easily generalizes to partitions with more than 2 clusters.

Begin with an arbitrary bisection P of the graph into clusters A and B. At each iteration, do the following:

1. For each pair of nodes $i \in A$ and $j \in B$, compute the variation in cut size between the current partition and the one obtained by swapping *i* and *j*.

2. The pair of nodes $i^*$and j^* yielding the largest decrease in the cut size is selected and swapped. This pair of nodes is locked; they will not be touched again during this iteration.

3. Repeat steps 1 and 2 until no more swaps of unlocked nodes yields a decrease in the cut size. This yields a new partition P' that is used as the starting configuration for the next iteration.

Stop iterating when the new partition fails to improve on the previous one.

The Fiedler Method

Uses the smallest non-zero eigenvalue of the graph Laplacian to partition a graph.

Recall that if *G* is a simple undirected graph with *n* nodes, then the graph Laplacian matrix is $L = D - A$ where D is the main diagonal matrix of A.
If G is connected then the eigenvalues of L satisfy:

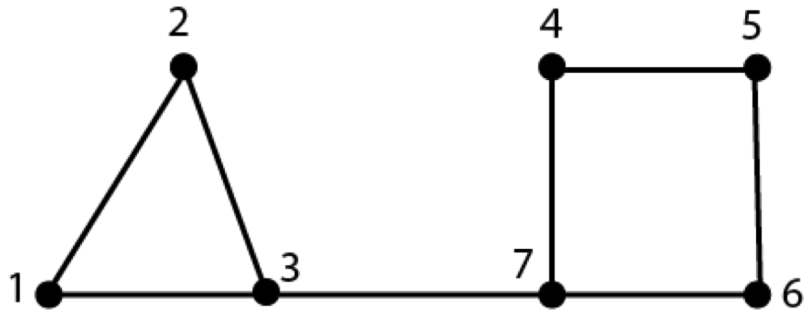
$$0 = \lambda_1 < \lambda_2 \leq \lambda_3 \ldots \leq \lambda_n$$

The Fiedler value or algebraic connectivity of G is the first non-zero eigenvalue, λ_2.

The Fiedler vector is the eigenvector corresponding to the Fiedler value.

The idea is that nodes corresponding to positive elements of the Fiedler vector form one community, while those corresponding to negative elements form the other. If there is a 0 element, then put that node into either community.

The Fiedler Method



The eigenvalues of *L* are 0, 0.36, 2, 2.28, 3, 3.59, 4.78

The Fiedler value is 0.36 and the Fiedler vector is

$$\vec{v} = (0.48, 0.48, 0.31, -0.35, -0.42, -0.35, -0.15)$$

Nodes with positive entries: 1, 2,3
Nodes with negative entries: 4, 5, 6, 7

First community={1, 2, 3} Second community={4,5,6,7}

This agrees with our expectation; we removed the minimum number of edges in the partition, while keeping similar number of nodes in the subgraphs. This is called a minimum cut.

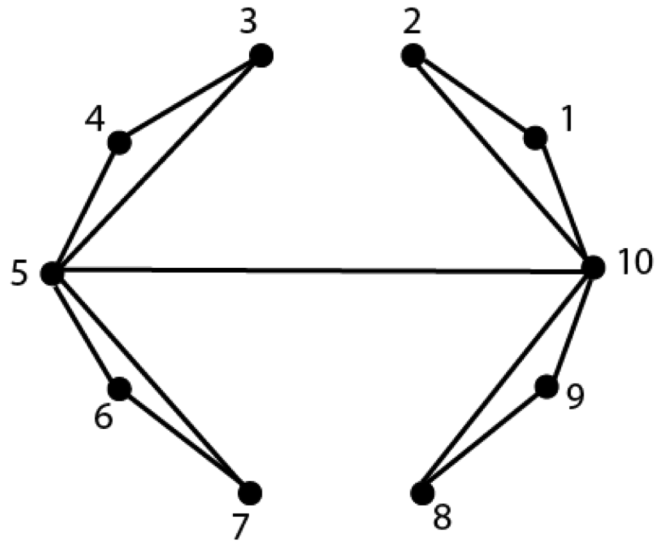
Graphs with More Modules

In general, we can look for the biggest jump in the eigenvalues of *L*. If the biggest jump in value is between λ_4 and λ_5, then the most reasonable description of the graph is that it has 4 modules. This jump between eigenvalues is called the **spectral gap**.

In our last example, the eigenvalues of *L* were 0, 0.36, 2, 2.28, 3, 3.59, 4.78

The largest spectral gap is between $\lambda_2 = 0.36$ and $\lambda_3 = 2$, indicating two modules.

Graphs with More Modules



Eigenvalues of *L*: 0, 0.2, 1, 1, 3, 3, 3, 3, 5, 6.8

Clear large spectral gap between λ_4 and λ_5. Indicating 4 modules in the graph.

Information on how to form these modules is contained in the three eigenvectors associated with the first 3 non-zero eigenvalues,

$$\vec{v}_2, \vec{v}_3, \vec{v}_4$$

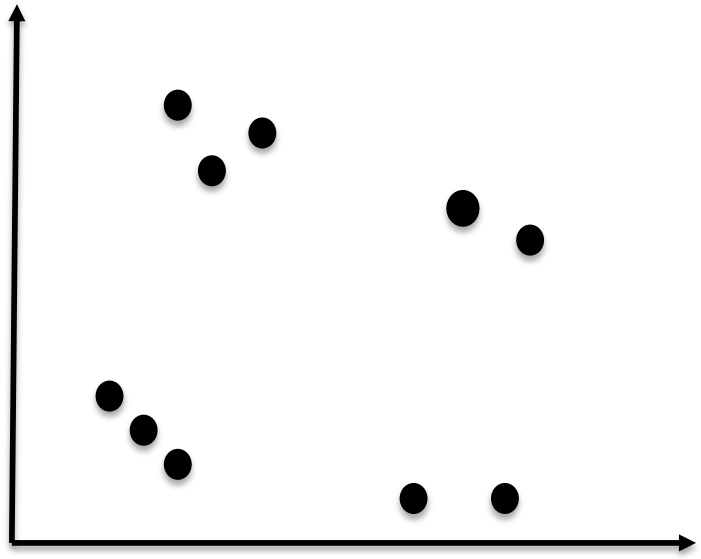
Each of these has dimension equal to the number of nodes, which is 10.

Graphs with More Modules

The first element of each vector corresponds to node 1. Collect them Into a vector $\vec{w}_1$. Do the same for the second element, forming $\vec{w}_2$. Repeating, there will be $n = 10$ such vectors, each with 3 elements. Plot these in 3-D space with tails at the origin, and the vectors point to *n* points in $\mathbb{R}^3$.

Those points that "cluster together" should correspond to nodes in the same module. But how to do this clustering analysis?

Sketch in 2-D



K-means Clustering

Here K is the number of clusters you want to put the "data" into.
In our example, we want to find 4 modules in the graph, so the "data" should be put into 4 clusters.

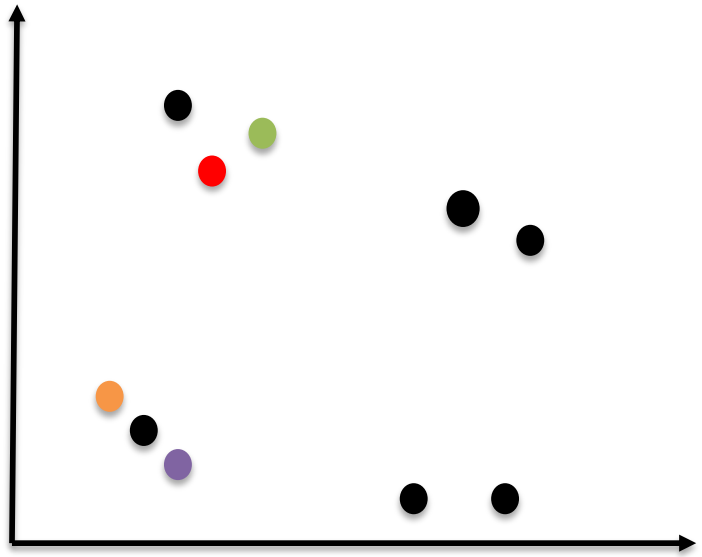
Step 1: Initialize by picking 4 of the data points, and calling each a centroid: c_1, c_2, c_3, c_4.

Step 2: Assign each other data point to be in the cluster with the closest centroid.

Step 3: Recalculate the centroids for the 4 existing clusters and repeat steps 2 and 3.

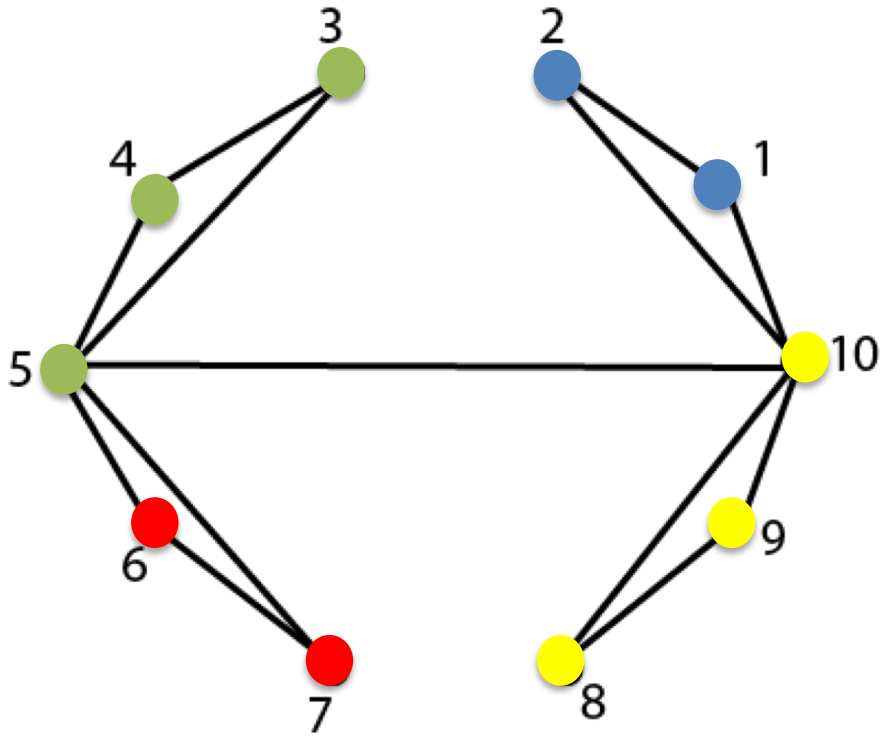
Continue iterating until none of the data points change clusters.

Colors are different initial centroids, chosen randomly



Outcome with Our Example

Different colors indicate different modules

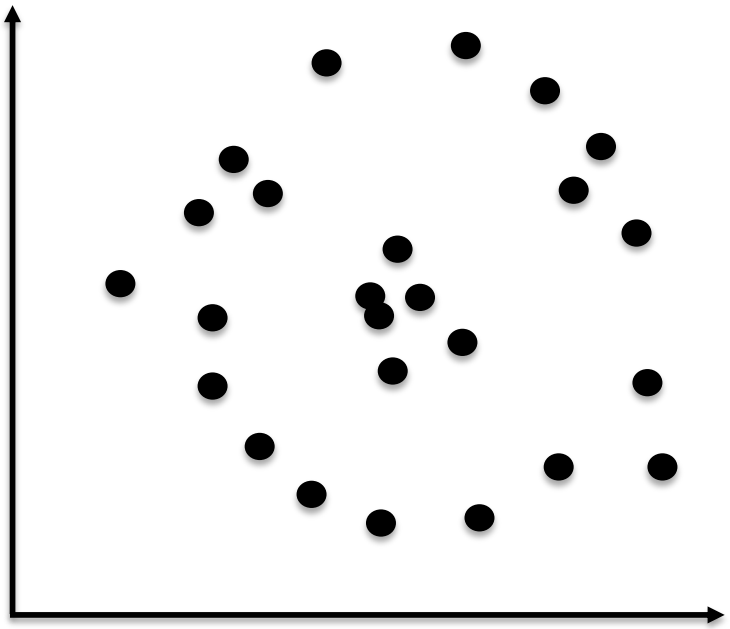


Choosing different initial means could have produced somewhat different results. What would another possible result look like?

This approach could be called Spectral Community Detection since it uses eigenvalue and eigenvector information.

Clustering Data

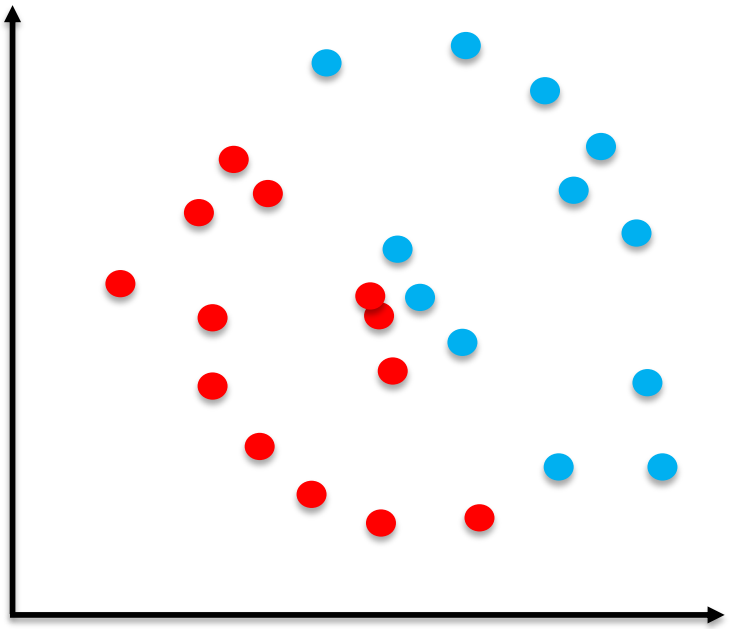
What if we don't have a network, but just data points?



There are clearly 2 clusters here, the inside points and those on the ring

K-means Clustering Can Easily Fail

Using K-means clustering, specifying 2 clusters (so using 2 centroids) we get the following:

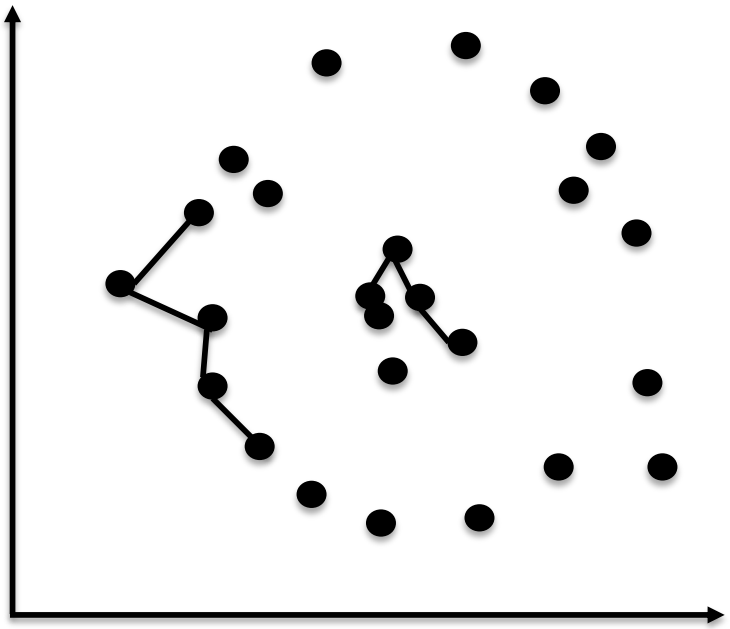


or something like this, depending on which choice is made for the initial centroids

Can We Use a Spectral Approach?

To do this, we first need a network, not just data. The easiest thing to do is to construct a *k*-nearest neighbors graph. Treat each data point as a node in a graph, then connect it with an edge to each of its k nearest neighbors (in the potentially high dimensional space in which the data exist). Typically, one uses *k* anywhere from 5 to 10.

A few edges shown for $k = 2$

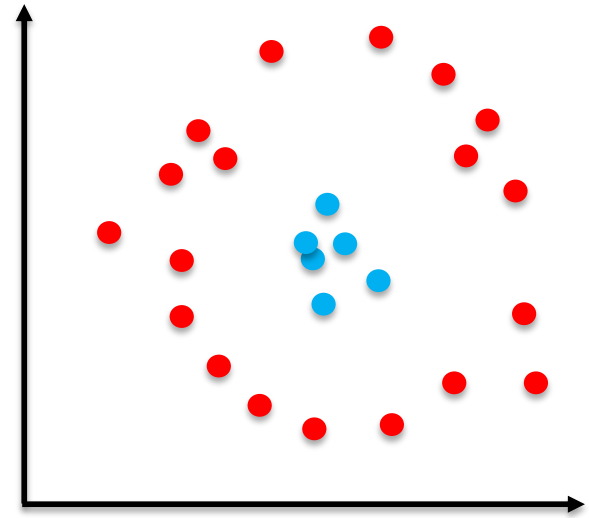


In this example, the nodes on the outside will form a connected component and those on the inside will form another.

An example of Spectral Clustering

Since we are only separating the data into two components, we can use Fiedler partitioning. That is:

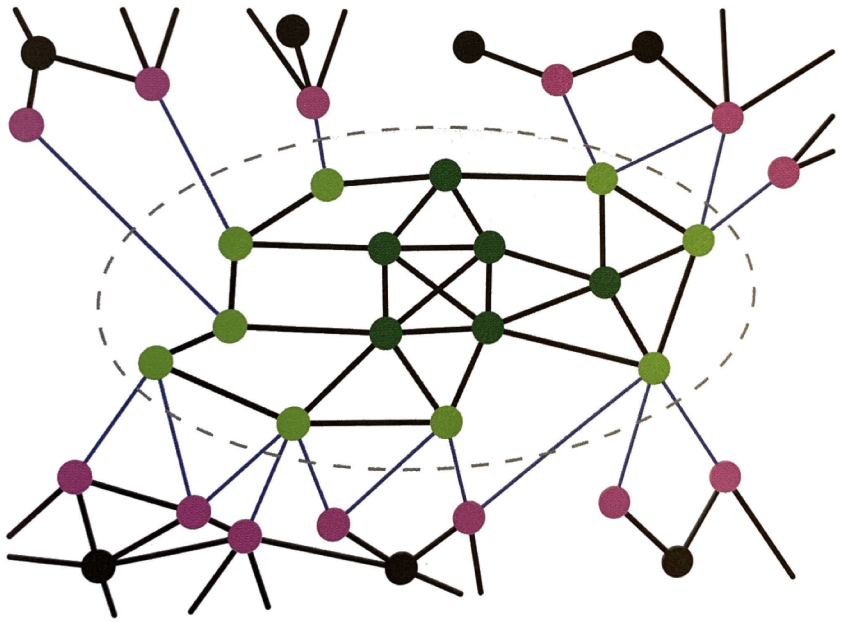
(1) Using the graph, construct the adjacency matrix *A*
(2) Compute the graph Laplacian *L*
(3) Find the eigenvalues of *L*
(4) Find the Fiedler value and Fiedler vector
(5) Use the Fiedler vector to partition the graph into two modules
(6) Cluster the data according to which module they are in



If we wanted to split into more than 2 modules, then we would have used the approach used earlier that involved more than one eigenvector of *L*.

What is a Community?

Graph partitioning separates nodes, but the elements of each cluster may have little interconnectivity. A community should have **cohesion**.



Internal degree of a node in a community: number of neighbors inside the community

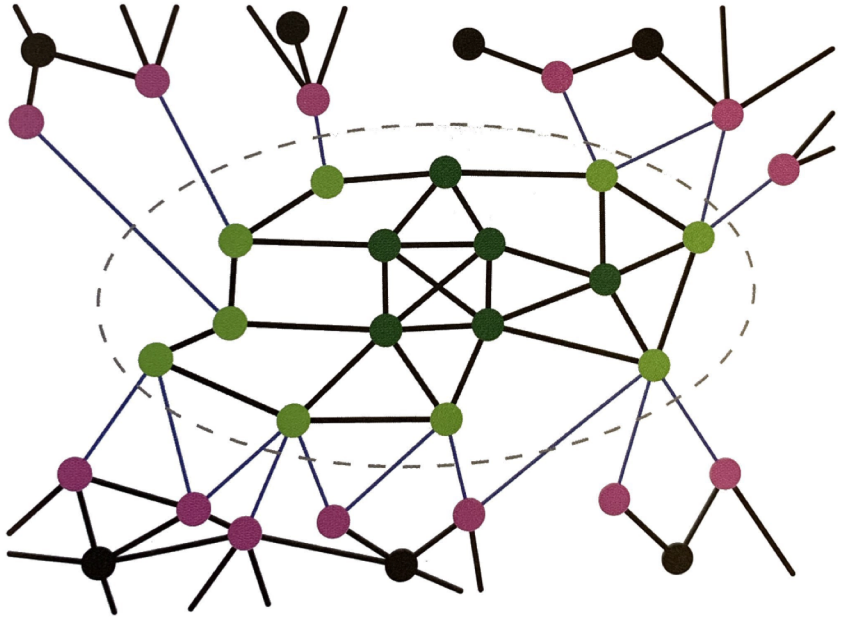
External degree of a node in a community: number of neighbors outside the community

Community degree: the sum of the degrees of the nodes in the community

Internal link density: ratio of the number of internal links and the maximum number of links that could exist between community nodes. This is just the density of the community subnetwork.

What is a Community?

A strong community is a subnetwork such that each node has more neighbors in the subnetwork than in the rest of the network.

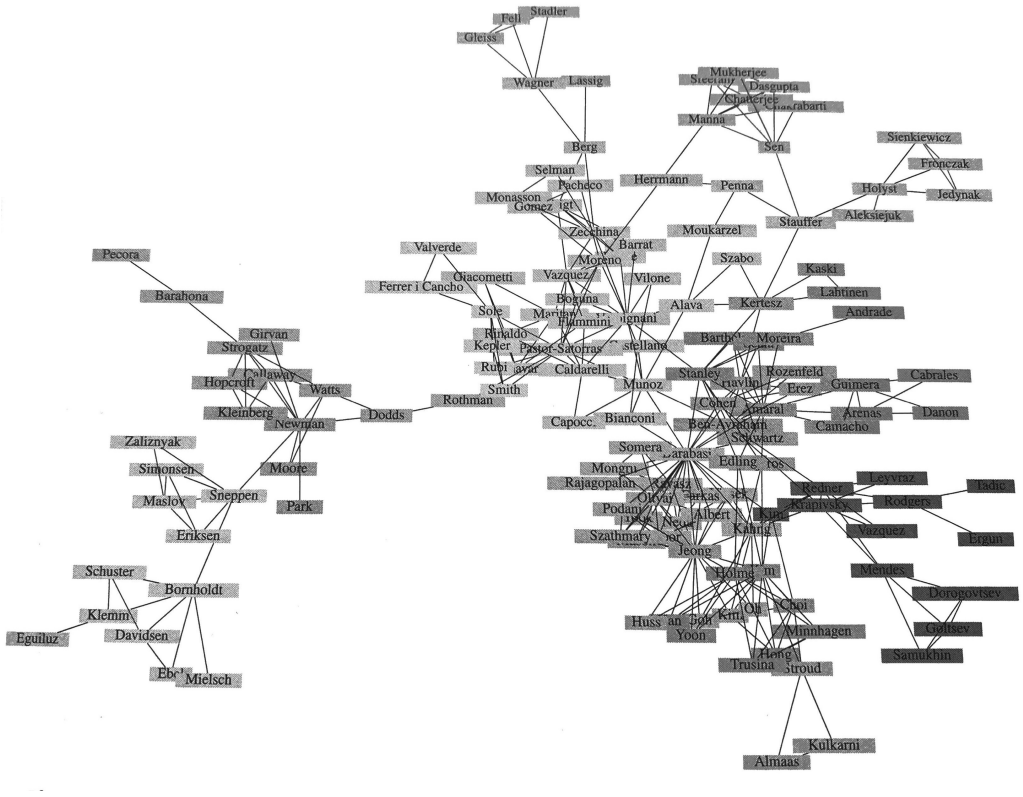


A weak community is a subnetwork such that the sum of the internal degrees of all nodes exceeds the sum of their external degrees.

All strong communities are weak communities, but not vice versa.

Where are the Communities Here?

Easley and
Kleinberg 2010



Coauthorship network of researchers working on networks

Data Clustering

Community detection is a special version of the more general problem of **data clustering**, where data elements are grouped into clusters based on some notion of similarity.

For graphs, a natural measure of similarity of nodes is structural equivalence, which means the similarity or overlap between neighbors of the two nodes.

The similarity S_{ij} of a pair of nodes *i* and *j* via structural equivalence is:

$$S_{ij} = \frac{\text{number of neighbors shared by i and j}}{\text{total number of distinct neighbors of i and j}}$$

Example: if the neighbors of *i* and *j* are $\{v_1, v_2, v_3\}$ and $\{v_1, v_2, v_4, v_5\}$, then

$$S_{ij} = \frac{2}{5} = 0.4.$$

If two nodes shared no neighbors, then $S_{ij} = 0$. If all neighbors are shared, then $S_{ij} = 1$.

Data Clustering

Similarity between two groups of nodes, G_1 and G_2:

Measure the similarity of all pairs of nodes $\{i, j\}$ with $i \in G_1$ and $j \in G_2$.

The group similarity is then $S_{G_1 G_2} = \Phi(S_{ij})$ where the function Φ could be $\max_{i,j} S_{ij}$, or $\min_{i,j} S_{ij}$, or average $\langle S_{ij} \rangle$.

An agglomerative clustering technique iteratively merges groups of nodes based on their similarity.

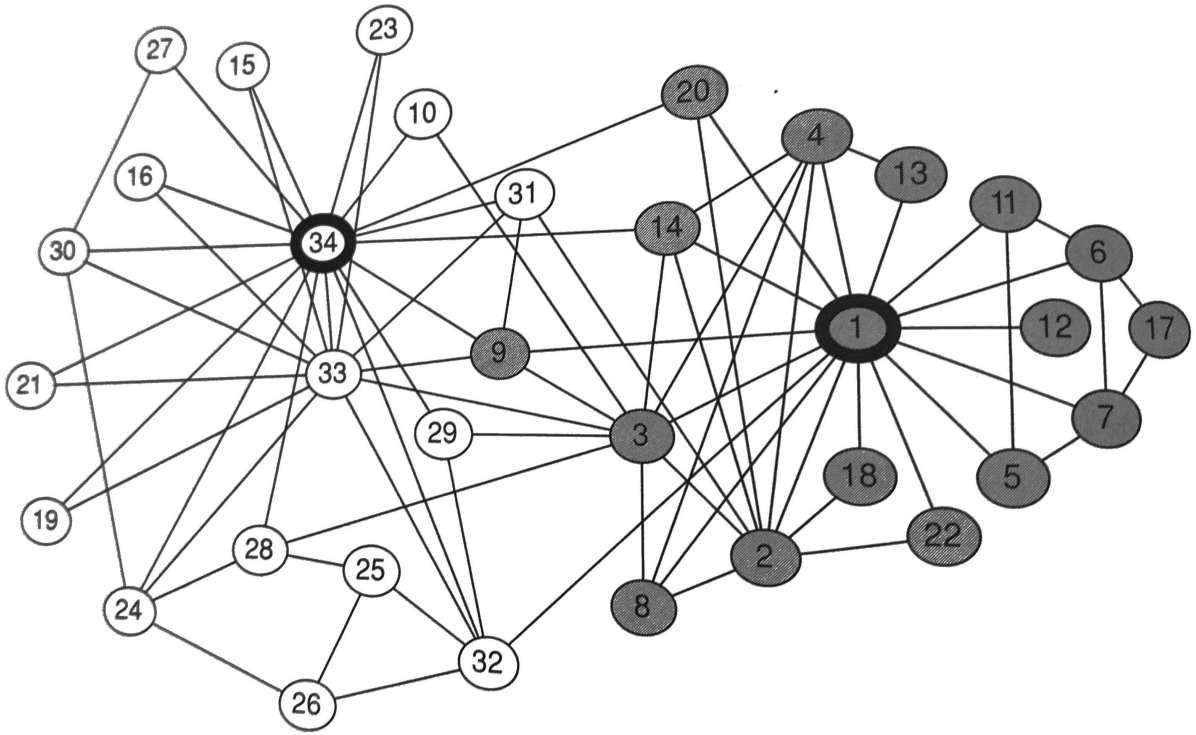
Start with a trivial partition of *N* groups (each node is a group).

At each iteration, find the group pair with the greatest similarity and merge those into a single group.

Repeat until all nodes are in the same group.

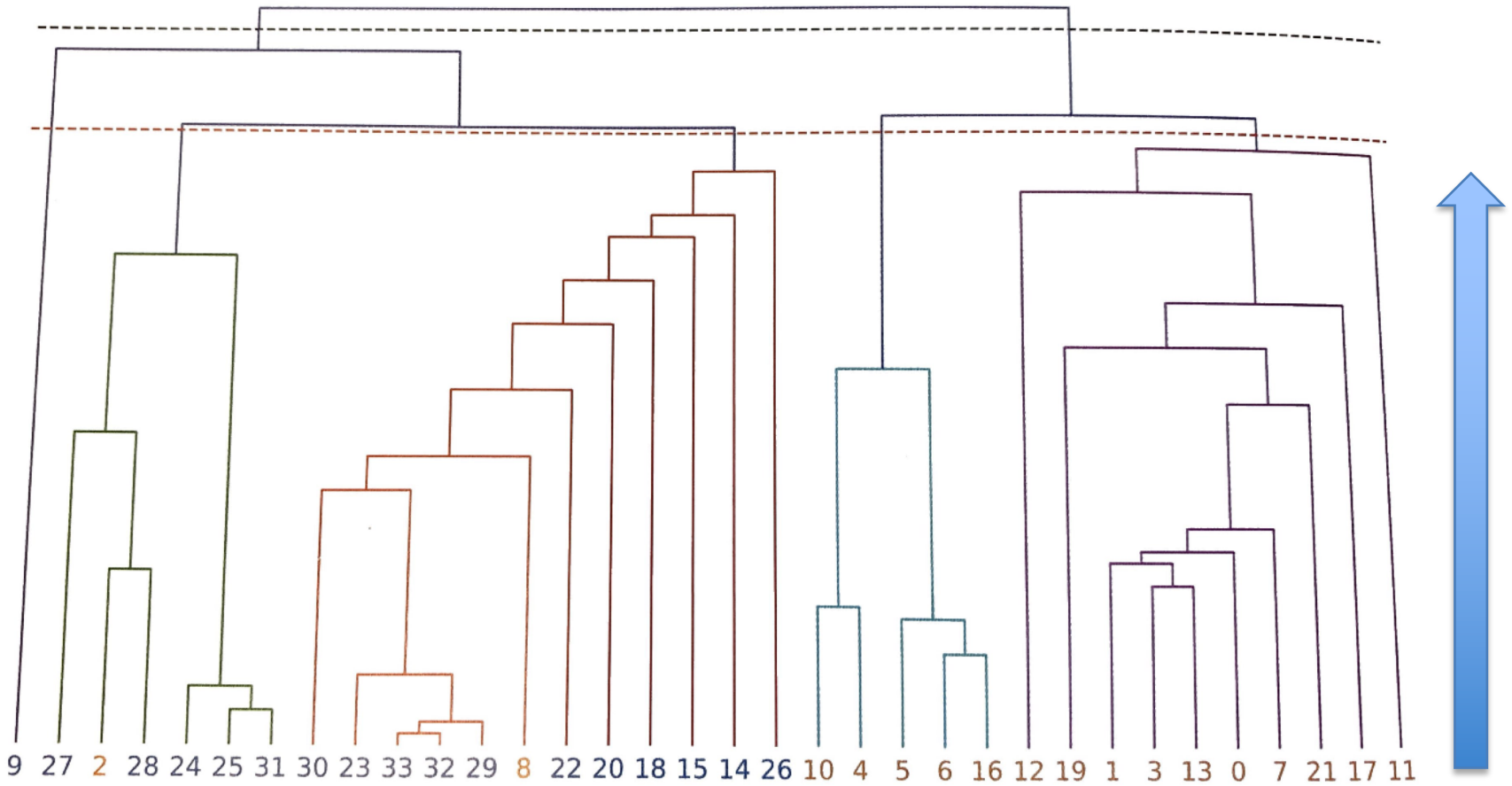
How Does This Agglomerative Algorithm Do With Zachery's Karate Club?

Easley and Kleinberg 2010



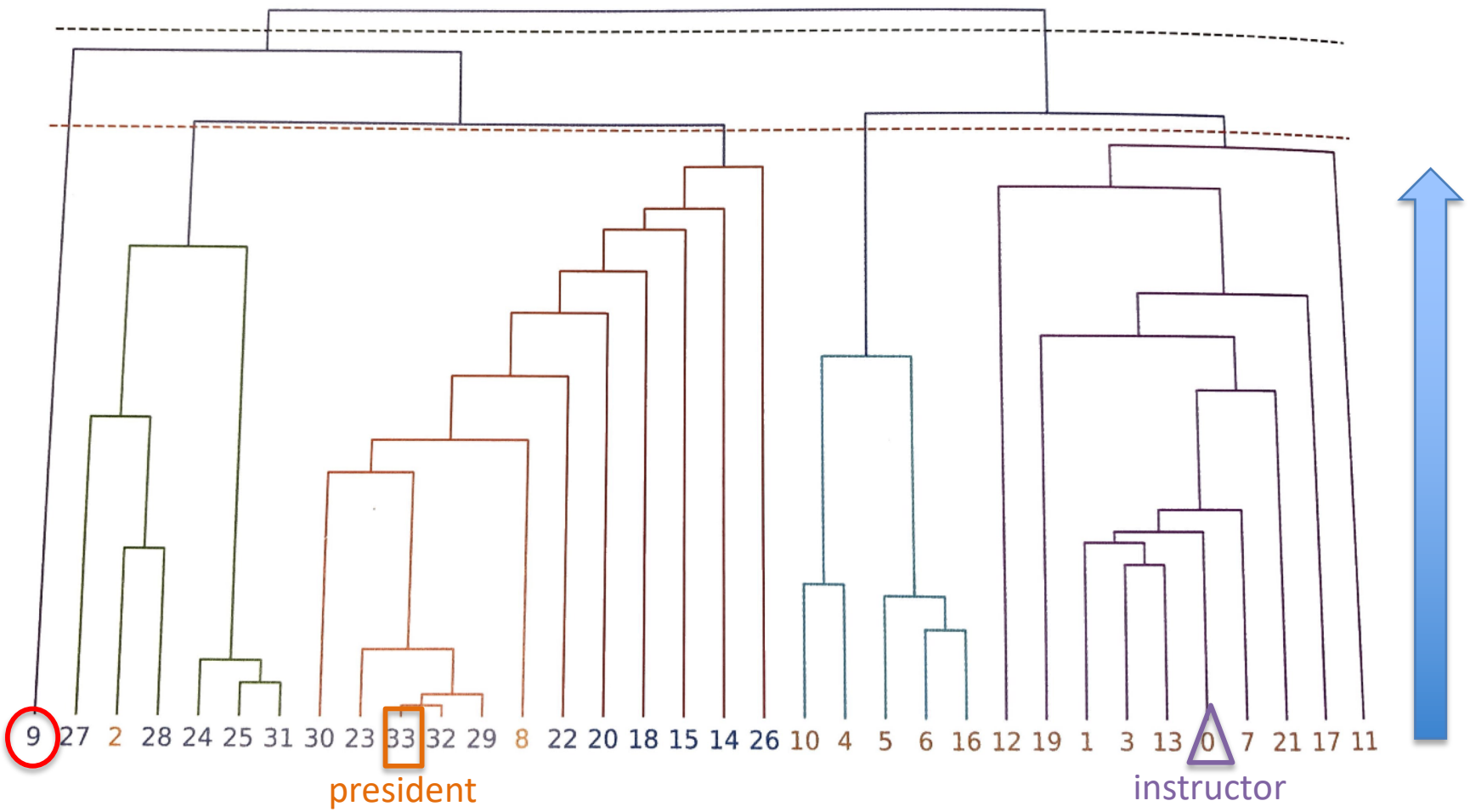
Members of a karate club that ultimately split into two clubs following a dispute. One group sided with the club president (node 34) and the other with the club instructor (node 1).

Hierarchical Clustering of Zachery's Karate Club



Dendrogram shows *N* partitions of the network. Each dashed slice yields a partition. The top slice produces a partition with 2 clusters. The lower slice produces a partition with 5 clusters. The algorithm is agglomerative: it builds the dendrogram from the bottom up.

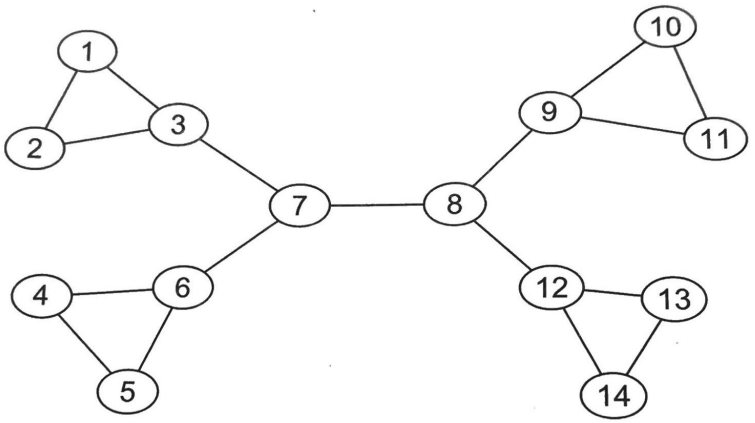
Hierarchical Clustering of Zachery's Karate Club



The algorithm has a hard time finding a cluster for node 9, so it stays isolated from the other clusters until almost the end. When looking at a partition into two clusters, node 9 is put in the group with the club president. But the person actually went into the group with the club instructor. [Note that numbering here starts at 0, while in network graph it started at 1.]

Divisive Methods for Network Partitioning

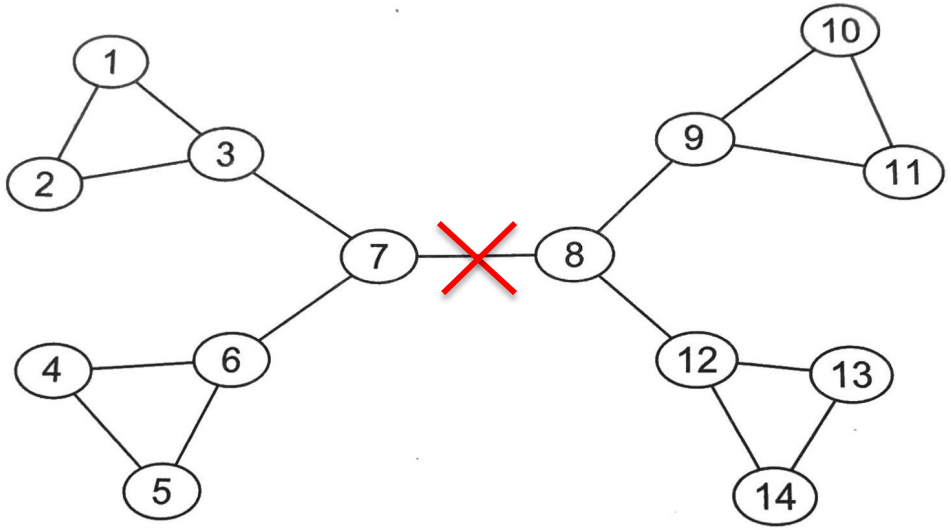
Easley and
Kleinberg 2010



Divisive methods erase edges to divide up the network

Divisive Methods for Network Partitioning

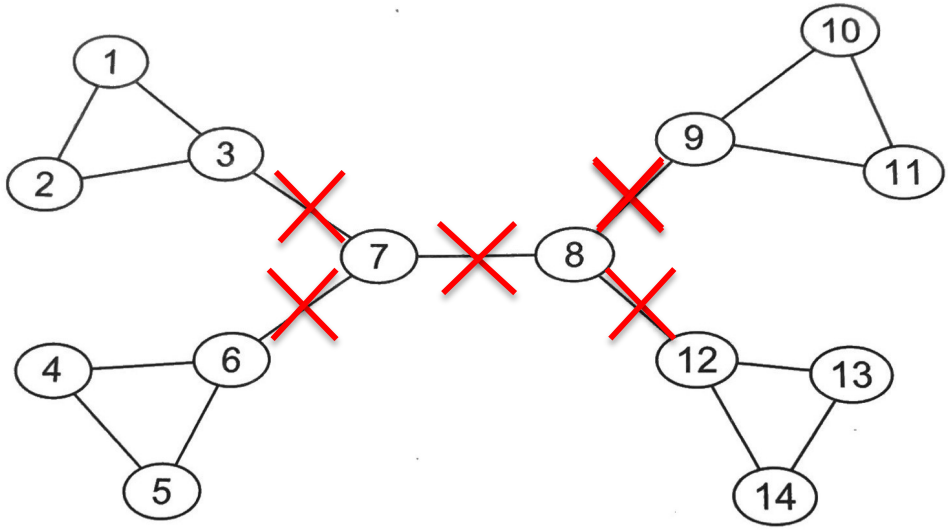
Easley and
Kleinberg 2010



Iteration 1: remove central link

Divisive Methods for Network Partitioning

Easley and
Kleinberg 2010



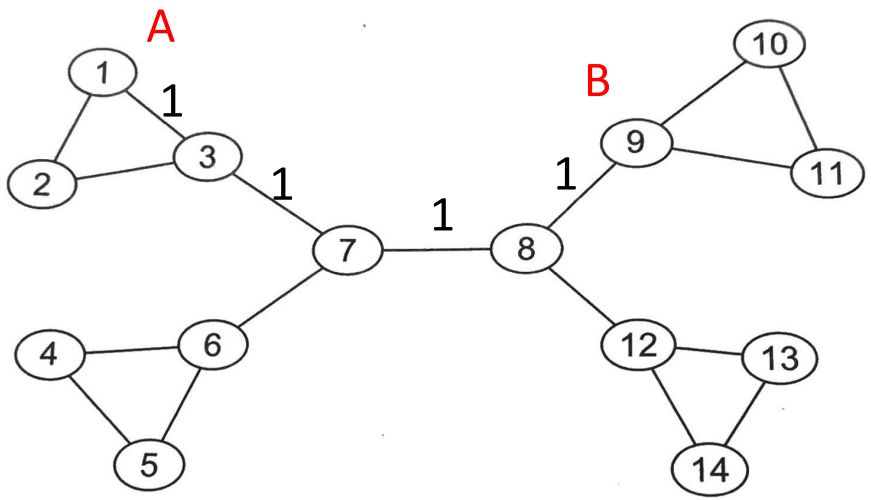
Subsequent iterations: remove links to other clusters

The Girvan-Newman Divisive Method

Basic idea: Iteratively delete the edges containing the most “traffic”.

Consider each pair of nodes, A and B. Find the shortest path(s) between them. For each edge on such a path add one unit of flow. If there are two shortest paths, then edges on each get ½ unit of flow, etc.

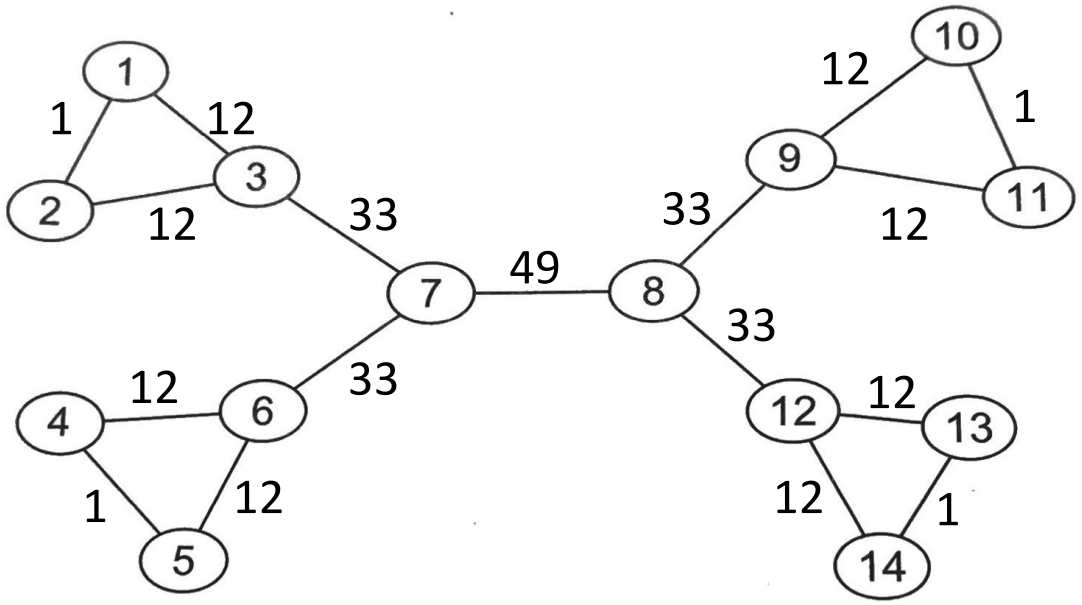
Easley and Kleinberg 2010



The Girvan-Newman Divisive Method

Sum the flows at each edge for all node pairs. This gives the betweenness of each edge (previously we discussed betweenness of nodes).

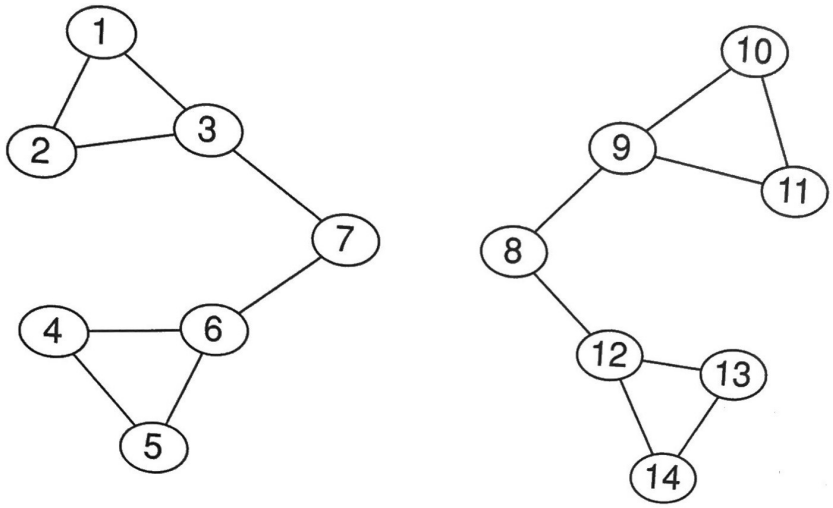
Easley and Kleinberg 2010



The Girvan-Newman Divisive Method

Next, remove the edge with the greatest betweenness (or edges if there is a tie).

Easley and Kleinberg 2010

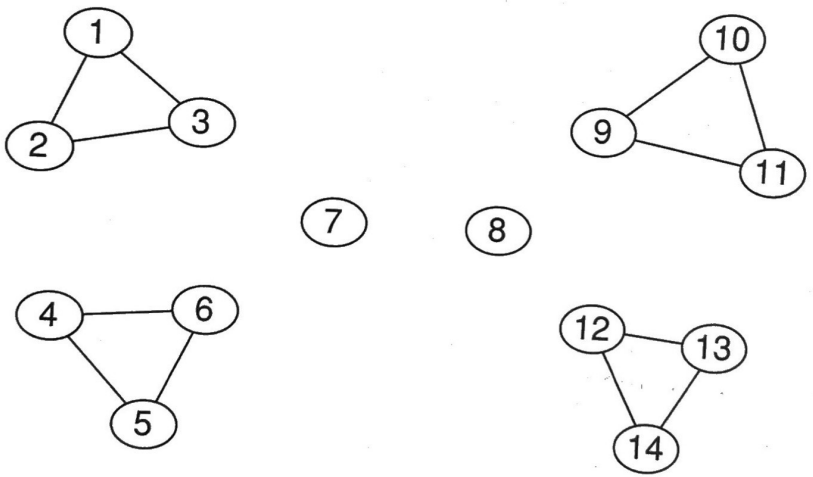


This is the first level of partitioning of the graph.

The Girvan-Newman Divisive Method

Recalculate the betweenness, then iterate again, removing the edge(s) with greatest betweenness.

Easley and Kleinberg 2010

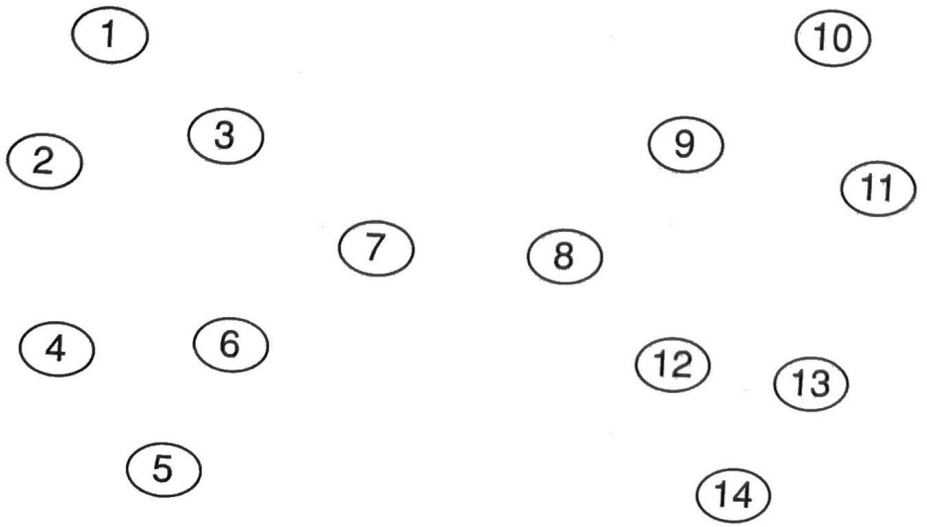


This is the second level of partitioning.

The Girvan-Newman Divisive Method

Repeat until all edges have been removed, each time recalculating the betweenness of all remaining edges.

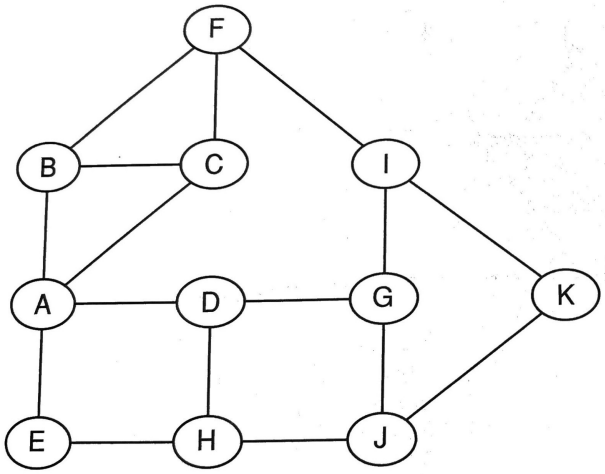
Easley and Kleinberg 2010



Computing Edge Betweenness

Is there a good way to count all the shortest paths between nodes? Yes, there is a clever process based on breadth-first search.

Easley and Kleinberg 2010



Perform a breadth-first search of the graph, starting at A

Determine number of geodesic paths from A to each other node

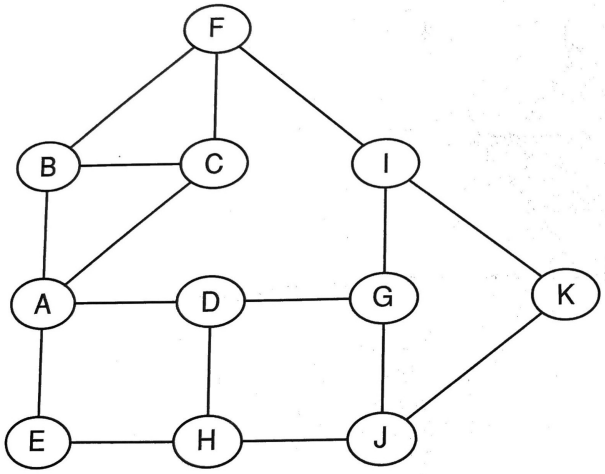
Based on these numbers, determine the amount of flow from A to all other nodes that use each edge

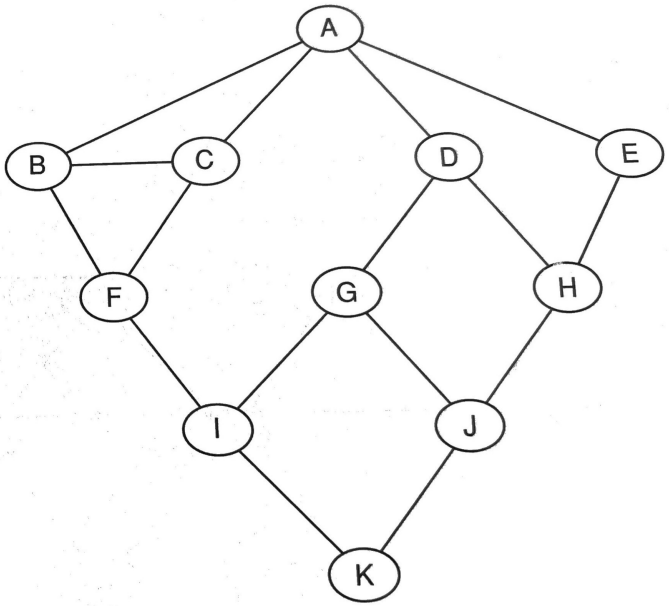
Do this for all other nodes and sum up flows.

Breadth-First Search

Rearrange graph into layers, with A at the top.

Easley and Kleinberg 2010





distance=1

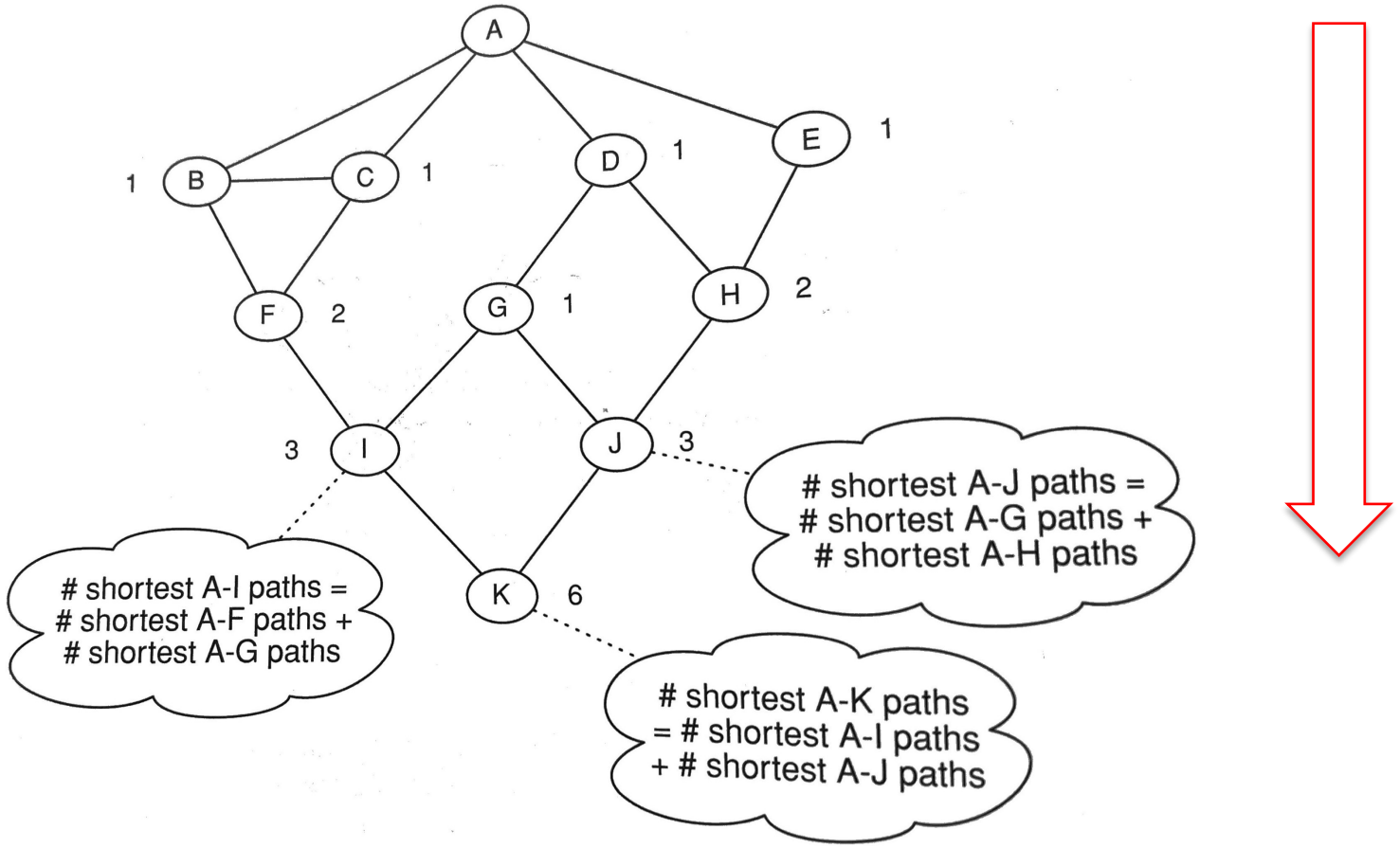
distance=2

distance=3

distance=4

Counting Shortest Paths

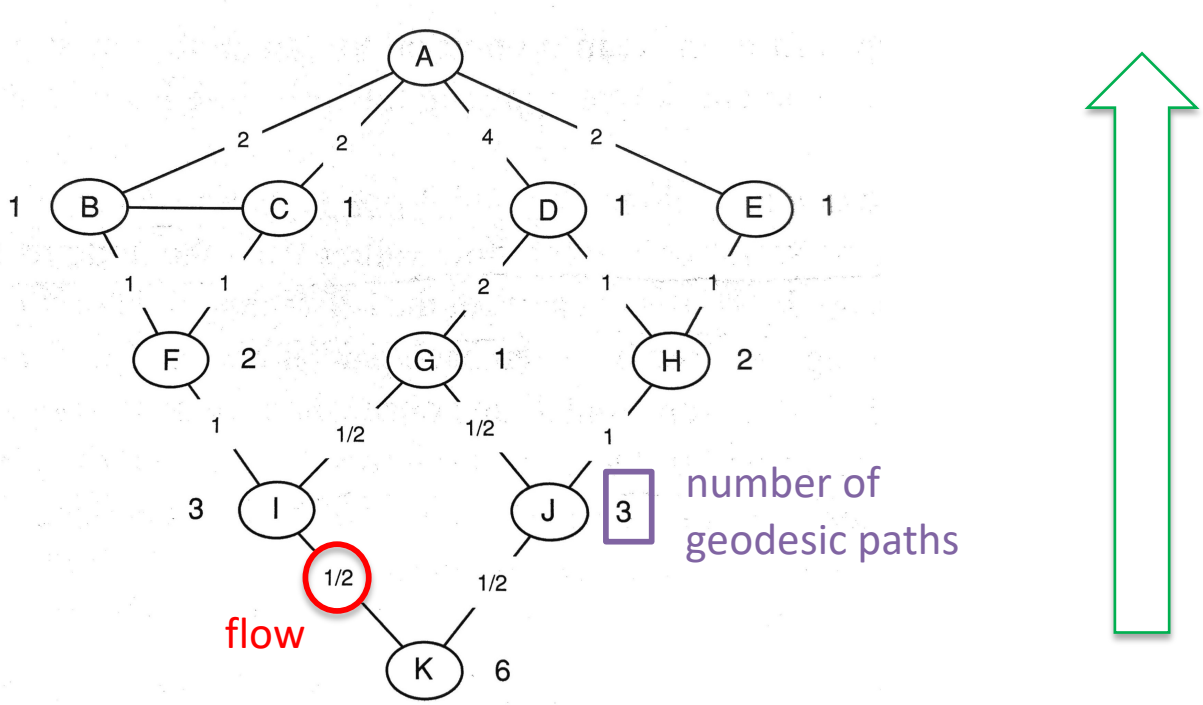
Easley and Kleinberg 2010



Add up number of shortest paths, moving downward through the network.

Determining Flow Values

Easley and
Kleinberg 2010



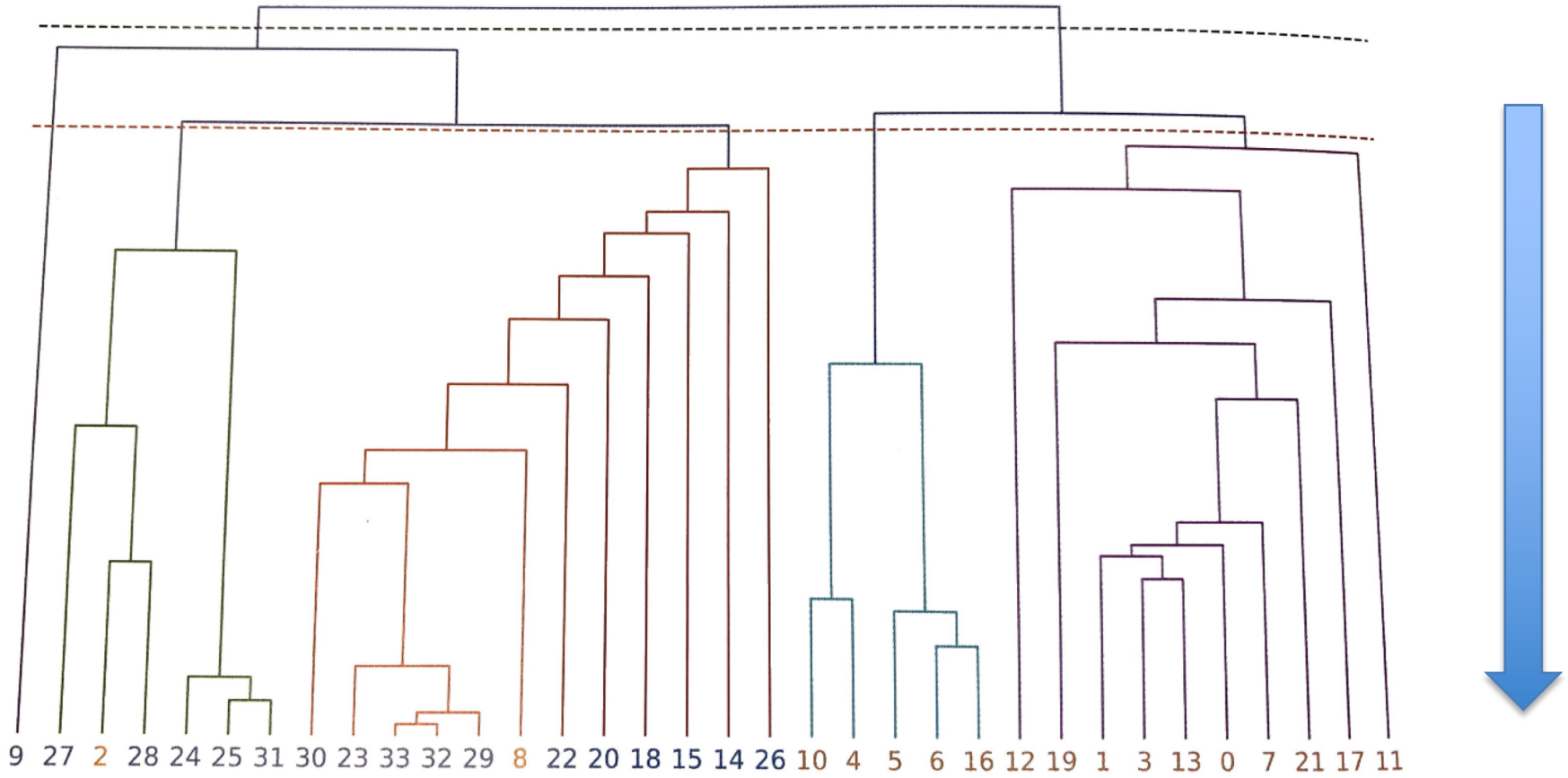
From bottom, K gets 1 unit of flow, divided equally along I-K and J-K

I gets 1 unit, plus the half unit passing through I to K, so 1 ½ total.
Edge F-I should get twice as much of this than G-I. So F-I gets 1 unit, G-I gets ½, and both get one unit for free.

Continue working up through the graph to get all flows associated with node A.

Now repeat, centering on node B. Add flows together. Continue repeating and summing. This gives betweenness values for all edges.

This Method is also Hierarchical, but Builds the Dendrogram from the Top Down



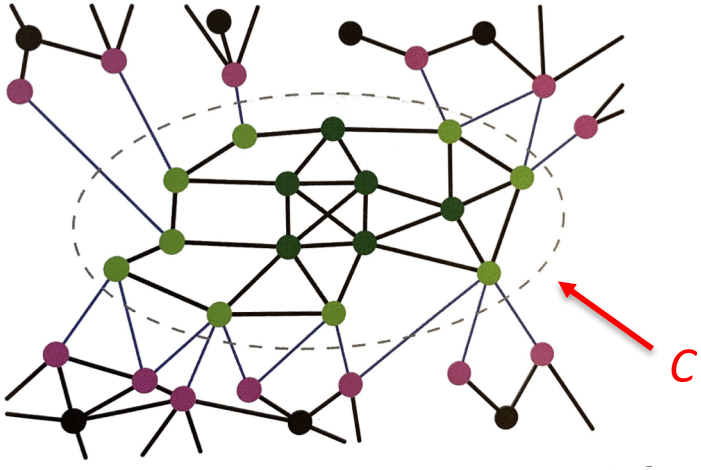
Dendrogram shows *N* partitions of the network. Each horizontal cut yields a partition. The top cut produces a partition with 2 clusters. The lower cut produces a partition with 5 clusters. This is divisive: the number of communities grows as the algorithm is iterated.

Modularity of a Partition

How appropriate is our partition of the network into communities? Are the communities really cohesive? What metric can be used to know?

Modularity of a Partition

If L is the total number of links in the graph, then the total number of stubs in the graph is $2L$. Suppose a community *C* has total degree of k_C. This is also the total number of stubs attached to nodes in *C*. If you randomly pick a stub in the graph, what is the probability that it will be in *C*? $\frac{k_C}{2L}$



For a random link to connect two nodes in *C*, the probability is $\approx \frac{k_C}{2L} \cdot \frac{k_C}{2L} = \frac{k_C^2}{4L^2}$

Since there are a total of *L* links in the graph, the expected number that are in *C* is $\frac{k_C^2}{4L}$

If the actual number of links in *C* is L_C, then actual – expected number of links in *C* is $L_C - \frac{k_C^2}{4L}$. The partition modularity can then be defined as

$$Q = \frac{1}{L}\sum_C \left(L_C - \frac{k_C^2}{4L}\right)$$

Larger Q values give more cohesive communities.

Modularity Optimization

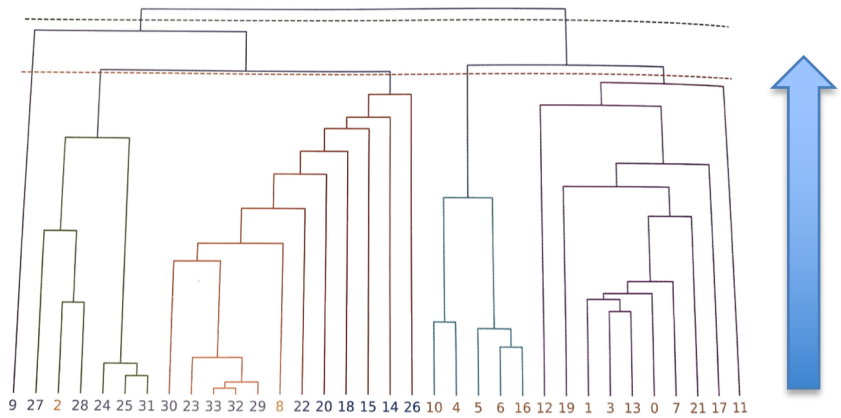
The modularity can be computed for any partition of the network, regardless of how the partition was arrived at. Partition modularity can also be used as the basis for partitioning the network. A simple example of such a modularity optimization algorithm is the one described below:

Start with a partition in which each node is its own community.

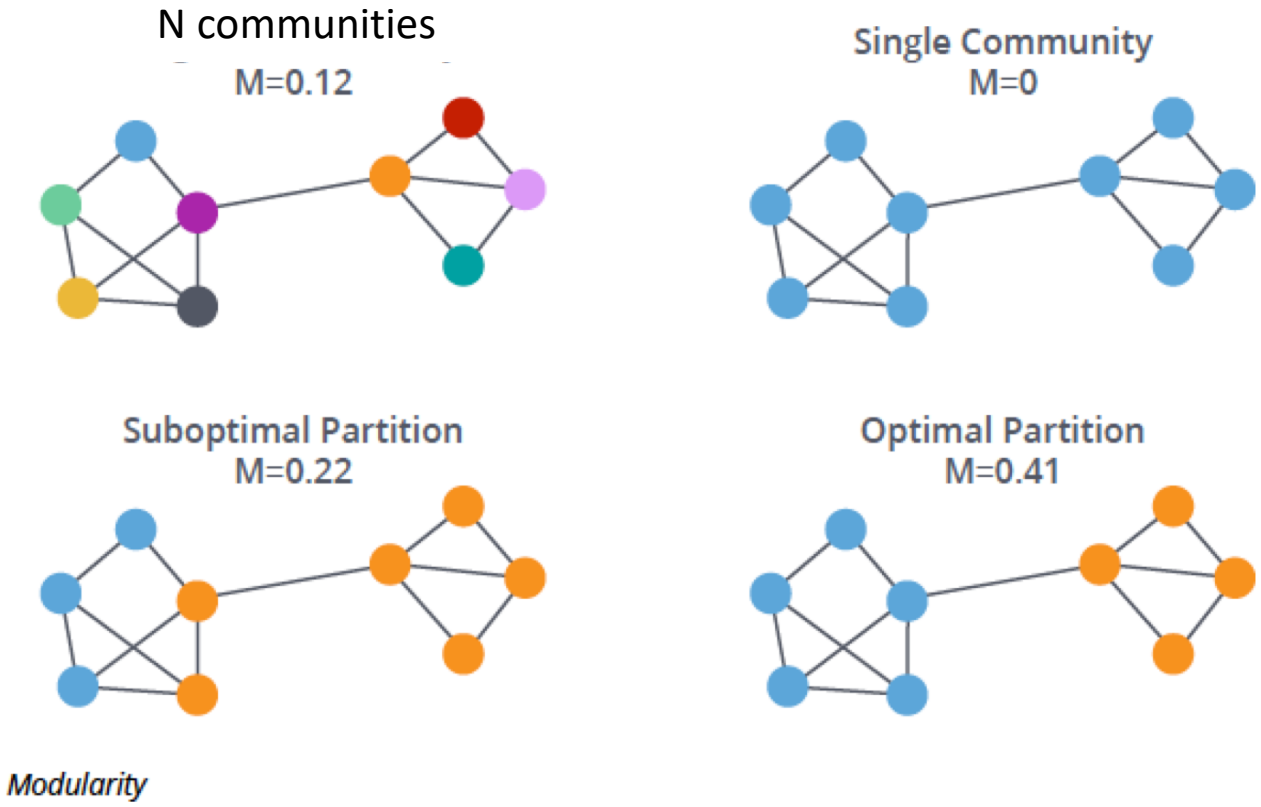
1. Merge the pair of communities that gives the largest increase in modularity.

2. Repeat step 1 until there is a single community containing all the nodes.

This will again yield a sequence of *N* partitions, forming a dendrogram. It works from the bottom up, so is an agglomerative algorithm.



Modularity Optimization



As the algorithm is iterated, the modularity first increases, then later decreases to 0. The “optimal partition” is typically taken to be the one where the modularity peaks.

Label Propagation

This is a simple and fast community detection method based on the idea that neighbors usually belong to the same community. That is, most links are internal to a community.

Start with a partition of singletons. Each node is given a different label. Then iterate over the following two steps.

1. Sweep over all nodes in random order: each node takes the label shared by the majority of its neighbors. If there is no unique majority, one of the majority labels is picked at random.

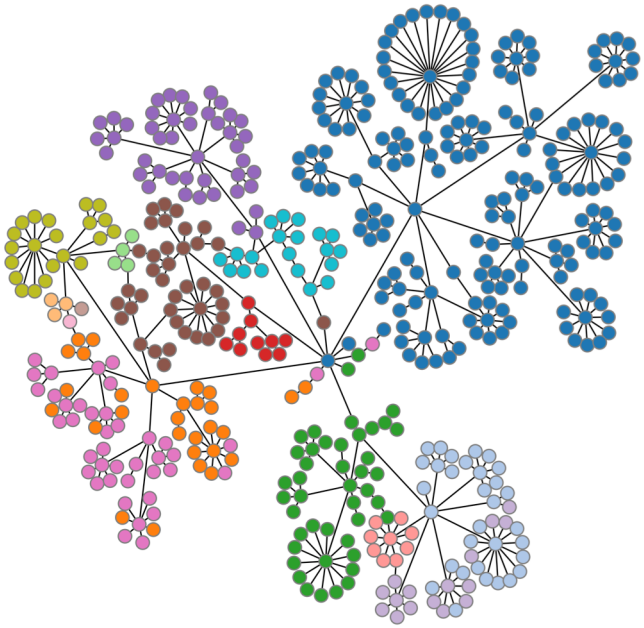
2. If every node has the majority label of it neighbors the system has reached a steady state, so stop. Else, repeat step 1.

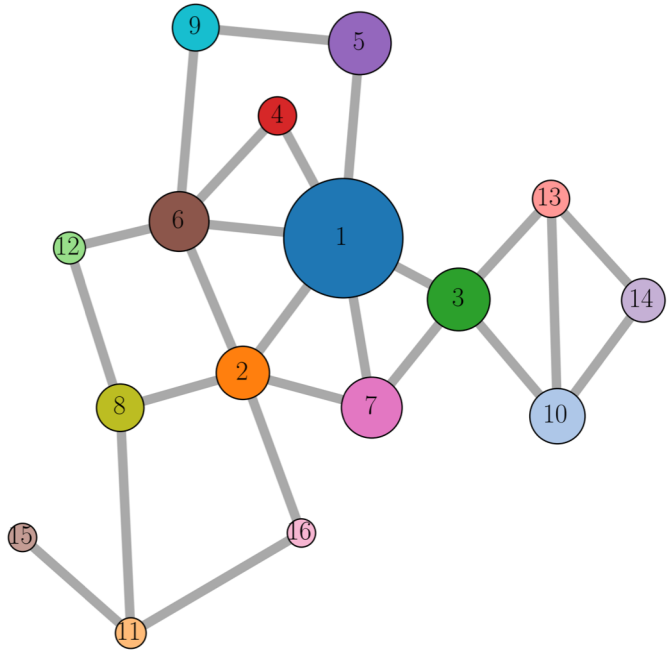
Communities are defined as groups of nodes with the same labels at steady state.

Is this an agglomerative or divisive algorithm?

Course Graining

Once you have partitioned the network into communities you can create a course grained network in which each node is a community of the original network. The edges then reflect links between communities.
This can be a useful way to visualize the network interactions on a macro scale.





The End