1 Clustering

Many problems in science require the identification of "clusters" in data. For instance, suppose we are a modern-day marketing analyst and we want to identify groups of people with similar interests so that we know what types of advertisements to show them. The logical extreme is a personalized advertising model where each individual is shown a unique set of ads. However, this requires aggregating and storing information about all individuals, which presents profound privacy issues, storage requirements, and the difficult problem of trying to build a personalized ad profile from potentially very scant information about the person.

The traditional approach is to, instead, create "market segments" or clusters, each of which represents a stereotypical person or a category of people who have similar interests and therefore can each be shown the same set of ads, e.g., stereotypes like "soccer moms," "blue-collar laborers," "tree-hugging, aura-sensing hippies," etc. This reduces the task of choosing which ads to show some particular person to the problem of identifying which segment said person falls into, based on whatever information we know about them, e.g., which websites they visit, what zip-code they live in, what they buy at Target, etc. Traditionally, market segments were derived by subjective insights, but increasingly they are created using algorithms to mine large data sets on human behavior. This can be done, for instance, by measuring a vector of interests $\vec{\theta}$ for each of n individuals—these interests could be represented by real values, integers, categorical variables, or any mix of these—whom we then partition into k disjoint groups such that the similarity of individuals within each group is greater than the similarity of individuals in different groups.

This approach is simply a spatial clustering problem,[1] where the goal is to find "dense" clusters of points in some metric space, where "similarity" becomes a measure of inverse-distance (more similar = smaller distance). But, what if we only have social network data? That is, instead of having a vector of interests for each person, we instead have the set of their pairwise "friendships"? An empirical fact called "homophily" says that the set of interests of some u and v, given that (u, v) is a friendship link, will tend to be similar.[2] Now, the problem of finding groups of people with similar interests is reduced to the problem of finding groups of people in the social network, where the group has more within-group connections than connections to other groups.

[1]This is a deep and old field, with methods coming out of statistics and machine learning and applications ranging through every natural, biological and social science.

[2]Homophily is a powerful principle and empirically comes close to a fundamental truth in sociology. "Birds of a feather flock together" and all that. What's less clear is, given that two people are friends and share some interests, to what degree they came to be similar because they are friends versus they are friends because they share interests.

1.1 Network clustering

There are many ways we might try to identify such "dense" clusters within a network. For instance, we might apply one of the many algorithms for `Graph Partitioning`, in which we are given a graph $G = (V, E)$ and an integer $k > 1$, and our task is to return a partitioning of V into k disjoint sets $V_1, \ldots, V_k$ such that the sets have equal size and the number of edges with endpoints in different sets is minimized.[3] Generalizations of this problem allow both edges and vertices to be weighted, and now we seek to minimize the weight of the *cut* induced by the partition.[4] For $k \geq 3$, graph partitioning is known to be NP-Hard; only for $k = 2$ (the graph bisection problem), is a polynomial-time algorithm known to exist.

Alternatively, we might have no preference on k and instead want the graph G to tell us what k leads to a "good" decomposition of the graph into dense groups. This is an example of an *unsupervised* learning problem because we seek to understand something about how the data is naturally organized. Because k is unspecified, the algorithm must encode some notion of what it means to be a "good" cluster that allows us to compare partitions with different numbers of groups; this notion is generally represented by a *score function*, which we'll denote f. A score function takes as input a graph and a partition of its vertices and returns a scalar value.

We will cover one popular unsupervised way to identify these clusters. First, however, we need to review a little about the mathematics of graph partitions, because finding a good partition means searching the space of all partitions of G to find the one that maximizes f.

1.1.1 Graph partitions

A graph partition is a division of a network with $n = |V|$ vertices into k non-empty groups (clusters), such that every vertex v belongs to one and only one group (cluster).[5] For a given value of k, the number of possible such partitions is given by the Stirling numbers of the second kind

$$S(n, k) = \left\{ \begin{array}{c} n \\ k \end{array} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^j \binom{k}{j} (k - j)^n \ . \qquad (1)$$

[3]To handle the fact that $|V|$ might not be evenly divisible by k, a small "imbalance" term ϵ is usually allowed.

[4]Graph partitioning crops up in load-balancing problems for parallel computation, in which we want to put most of the inter-node computation within a group of densely connected machines. Since we know how many groups of these machines we have, we know k *a priori*.

[5]This kind of clustering is sometimes call "hard clustering," in which vertices can be in one and only one clusters. In contrast, "soft" or "mixed membership" clustering allows vertices to be members of multiple groups, e.g., with membership being represented by probabilities.

For instance, there are $S(4,2) = 7$ possible partitions of a network with $n = 4$ nodes (indexed as 1,2,3,4) partitioned into $k = 2$ groups:

$$\{1\}\{234\},\ \{2\}\{134\},\ \{3\}\{124\},\ \{4\}\{123\},\ \{12\}\{34\},\ \{13\}\{24\},\ \{14\}\{23\}\ .$$

The number of all possible partitions of a network with n vertices into k non-empty groups is given by the nth Bell number, defined as $B_n = \sum_{k=1}^{n} S(n,k)$, which grows super-exponentially with n.[6] That is, the universe of all possible partitions of even a moderately sized network is very very big and an exhaustive maximization of f is rarely feasible. As a result, most network clustering algorithms use some kind of heuristic for estimating the maximum of f.[7]

1.2 The modularity score

One very popular choice for f is the Newman-Girvan *modularity function* Q, introduced by Mark Newman and Michelle Girvan in 2004, which is conventionally defined for simple, unweighted networks, but can be generalized easily to weighted multi-graphs.

Let k_i denote the degree of vertex i (please excuse the overloading of k), c_i denote the index of the group containing vertex i, and define a function $\delta(c_i, c_j) = 1$ if vertices i and j are placed in the same group (i.e., if $c_i = c_j$) and 0 otherwise. The modularity score of a partition is defined as

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

where we let $m = |E|$. This summation ranges over all pairs of vertices, but only accumulates value when the pair i, j are in the same group. That is, it counts only the internal edges of a group identified by the given partition. Crucially, the inner term counts these internal edges but subtracts their *expected number* under a random graph with the same degree sequence. A random graph is a model where edges occur with some probability; in this case, we use something called the "configuration model," which says that if two vertices have degrees k_i and k_j, then the probability that they are connected is exactly $k_i k_j / 2m$ and edges occur independently.[8]

[6] The first 20 Bell numbers (starting at $n = 0$) are 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322, 1382958545, 10480142147, 82864869804, 682076806159, 5832742205057.

[7] The fact that the search space is very large does not necessarily imply that it is computationally hard to find the point corresponding to the maximum of a function f over that space. Recall that for n numbers, there are $n!$ possible permutations and only two of these are perfectly sorted; however, finding a perfectly sorted permutation can always be accomplished in $O(n \ln n)$ time, i.e. sorting is in P.

[8] The configuration model is similar to the Erdős-Rényi graph in Problem Set 3, except that instead of each edge existing independently with probability p, edge exist independently conditioned on vertices having the degrees specified by the specified degree sequence $\{k_1, k_2, \ldots, k_n\}$.

Equivalently, and perhaps more intuitively, the modularity score can be rewritten as a summation over the structure of the modules themselves

$$Q = \sum_i \left[\frac{e_i}{m} - \left(\frac{d_i}{2m} \right)^2 \right] \ , \tag{3}$$

where now i indexes groups, e_i counts the number of edges within the ith group and d_i is the total degree of vertices in the ith module. Thus, the first term e_i/m measures the fraction of edges within group i and $(d_i/2m)^2$ is the expected fraction under the configuration model. A "good" partition—with Q closer to unity—represents groups with many more internal connections than expected at random; in contrast a "bad" partition—with Q closer to zero—represents groups with no more internal connections that we expect at random. Thus, Q measures the cumulative deviation of the internal densities of the identified modules relative to a simple random graph model.[9]

The following simple example illustrates how the modularity function scores two partitions on the same vertices. Consider G with $n = 6$ vertices and $m = 7$ edges, arranged so that we have two triangles connected by a single edge (u, v) (Figure 1). Now consider two partitions, one in which we place each triangle in a group by itself, and the other where we do the same but then move u into the same group as v.

These partitions induce two distinct 2×2 "block" matrices M, where M_{ij} counts the number of edges with one endpoint in group i and the other in group j (and where we allow double counting for edges with both end points in group i):

M_{good}	red	blue
red	6	1
blue	1	6

M_{bad}	red	blue
red	8	2
blue	2	2

The diagonals of M give $2e_i$, the ith column (or row) sum gives d_i, and the matrix sum gives $2m$.[10] Reading values from these matrices, the first partition has $e_i = 3$ and $d_i = 7$ (for both modules), which yields $Q = 5/14 = 0.357$; the second has $e_1 = 4$ and $d_1 = 10$ while $e_2 = 1$ and $d_2 = 4$, which yields $Q = 6/49 = 0.122$. So, the modularity function does a good job in this case of giving a higher score to the partition that separates the two triangles.

[9] I should point out that a high value of Q only indicates the presence of significant deviations from the "null model" of the random graph. A large deviation could mean two things: (i) the network exhibits genuine modular structure in an otherwise random graph or (ii) the network exhibits other non-random structural patterns that are not predicted by a random graph.

[10] Counting half-edges makes the math a little simpler but introduces these extra factors of 2. Equivalently, we could count only the full connections from vertices of type i to vertices of type j, which is the same as the above definition of M except that all the diagonal elements are half as big.

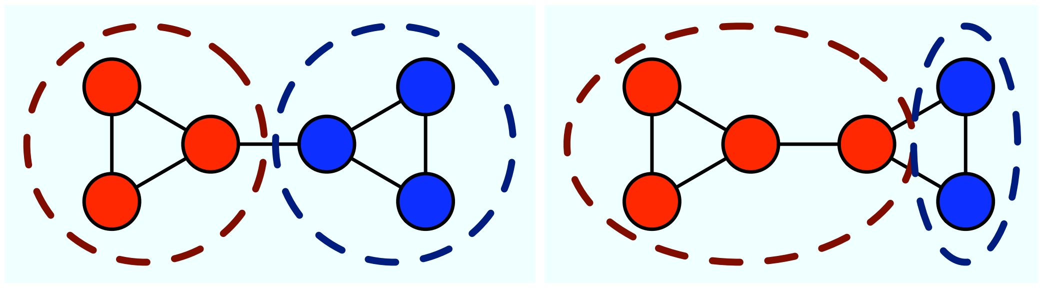

Figure 1: Two partitions of a simple network with modular structure. The modularity score of the first partition is $Q_{\text{good}} = 0.357$, while the score of the second is $Q_{\text{bad}} = 0.122$.

Recall that there are B_n possible partitions of a network with n vertices (in this case, $B_6 = 203$) and we would like to find the one partition that maximizes the modularity function Q. As with graph partitioning, maximizing Q is NP-hard.[11] However, despite this being true for general G, a number of heuristics perform well on most social and biological graphs.

1.3 A greedy agglomerative algorithm

One of those heuristics is the Clauset-Newman-Moore ("CNM") greedy agglomerative algorithm (from 2004). This algorithm runs in time $O(V \log^2 V)$ when the input graph is nice and $O(EV \log V)$ when it is not. (We'll define "nice" later.) In practice, the algorithm can cluster a network with millions or tens of millions of vertices in a few minutes to a few hours, making it suitable for large social networks.

The algorithm begins by placing each vertex is in a group by itself, i.e., a partition with $|V|$ singleton groups. This partition has a modularity score of

$$Q_0 = -\frac{1}{4}\sum_{i=1}^{n}\left(\frac{k_i}{m}\right)^2 \quad .$$

That is, each singleton has $e_i = 0$ and thus the modularity score is the sum of the negative terms. Because each k_i and m are positive values, Q_0 must be negative. (It may seem counter-intuitive that the expected number of edges within a singleton cluster is non-zero. Recall, however, that we subtract the number of self-loops expected under the configuration model, which is technically a multigraph model that allows self-loops, and these occur with probability proportional to $k_i \times k_i$.)

[11]See Brandes et al., "On Finding Graph Clusterings with Maximum Modularity." In *Graph-Theoretic Concepts in Computer Science, LNCS* **4769**, 121–132 (2007).

The algorithm then proceeds in steps. At each step, the algorithm picks a pair of groups in the existing partition and agglomerates (merges) them; if there is only one group left, no pair can be chosen and the algorithm halts. Each time it much pick a pair, it makes the greedy choice. To do this, it considers the incremental change in the modularity ΔQ_{ij} from agglomerating the ith and jth groups, and then chooses the pair with the largest positive ΔQ_{ij}. Because there are V initial groups, and the number of groups in the partition decreases by one at each step, this algorithm will stop after $V - 1$ such agglomerations.

(The history of these pairwise agglomerations can be thought of as a "dendrogram," which is a kind of binary tree that gives the partition structure at each step of the algorithm, with earlier partitions being lower in the tree. If this dendrogram is balanced, it will have logarithmic depth, indicating that most of the groups at each stage in the algorithm are of similar size. If the dendrogram is unbalanced, it will have linear depth, indicating that a few groups at each stage are very large while most are very small. The balance of the dendrogram will become important in our running-time analysis.)

Here's pseudo-code for the greedy-agglomerative algorithm:

```
Greedy-Agglomeration(G) {
   P = partition of |V| singletons            // initialization
   Q = Modularity(G,P)                        //
   while P has more than 1 group {
      for all pairs of groups, compute dQ_{ij}
      find maximum dQ_{ij}                    // make greedy choice
      P = Merge(P,i,j)                        // merge groups i and j
      Q += dQ_{ij}                            // update Q
      if Q > bestQ {                          // track maximum modularity partition
         bestP = P                            //
         bestQ = Q                            //
}}}
```

Here, `Modularity(G,P)` is a function that computes the modularity of a particular partition P given the graph G, and `Merge(P,i,j)` takes a partition P and the indices of two of its groups i and j, and returns a new partition where i and j have been merged.

1.3.1 A naïve implementation

A naïve implementation of this greedy agglomerative algorithm would represent the set of ΔQ_{ij} values as a matrix, recompute it from the original graph G at each step, and search over the set for the maximum value. This is computationally very slow, leading to a best-case performance of $\Theta(V^3)$: we pass through the outer loop $V - 1$ times; if computing any particular ΔQ_{ij} takes constant time, then we have to compute $\Theta(k^2)$ of these to get the ΔQ matrix on k groups each time through the loop (in fact, it takes $O(V)$ time to compute a ΔQ_{ij}; do you see why?); and it takes no more time to identify the maximum of that set. Thus, it takes $\sum_{k=1}^{V} k^2 = \Theta(V^3)$ time total. Further, it takes $\Theta(V^2)$ space to store the ΔQ matrix, so this is a fairly inefficient algorithm if the input graph G is sparse.

1.3.2 A more efficient implementation

If G is sparse, then we can do much better by being smarter about the data structures we use. First, note that merging two groups that are not connected by an edge can never increase the modularity. Thus, instead of considering all pairwise merges, we only need to consider the potential mergers between groups that are connected by at least one edge. Initially, because each vertex is in its own group, there are E potential merges rather than V^2 as in the naïve version. As the algorithm progresses, the number of these potential merges decreases, potentially very quickly.

In the sparse case, i.e., if $E = O(V)$, we can use a sparse-matrix data structure to efficiently store the ΔQ_{ij} value for each pair of groups i, j connected by at least one edge, and this matrix will also be sparse. That is, rather than store the adjacency matrix (as either a full matrix or a sparse matrix), we can simply store the clustered network corresponding to the current partition P, where nodes now represent entire groups and an edge (i, j) is weighted by the change in modularity ΔQ_{ij} induced by merging groups i and j. If we store these adjacencies using a balanced binary tree data structure, we can find or insert elements in $O(\log V)$ time.

But, we also need to efficiently find the maximum-weight edge ΔQ_{ij}. So, we also store the edges in a max-heap H, so that the largest value can be found in constant time. Because a merge operation will modify some edge weights, we have to keep these two sets linked so that changes on the clustered graph are reflected in the max-heap. That is, if we change the weight of some edge in the sparse matrix, we need to be able to quickly find its corresponding entry in H, update it, and then restore the heap property. This can be done by *linking* these data structures (see Chapter 14: Augmenting Data Structures in the text), so that elements in one structure store pointers to the corresponding element in the other structure.

Finally, we need a way to efficiently update the remaining weights ΔQ_{xy} values *after* we merge some pair $i, j \neq x, y$. A little algebra allows us to derive update equations for these modification, and these equations (see below) only require that we maintain a vector a_i storing the $d_i/2m$ for each of the groups.

In the initial partition, the edge weights are simply:

$$\Delta Q_{ij} = \begin{cases} \frac{1}{2m} - \frac{k_i k_j}{4m^2} & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} .$$

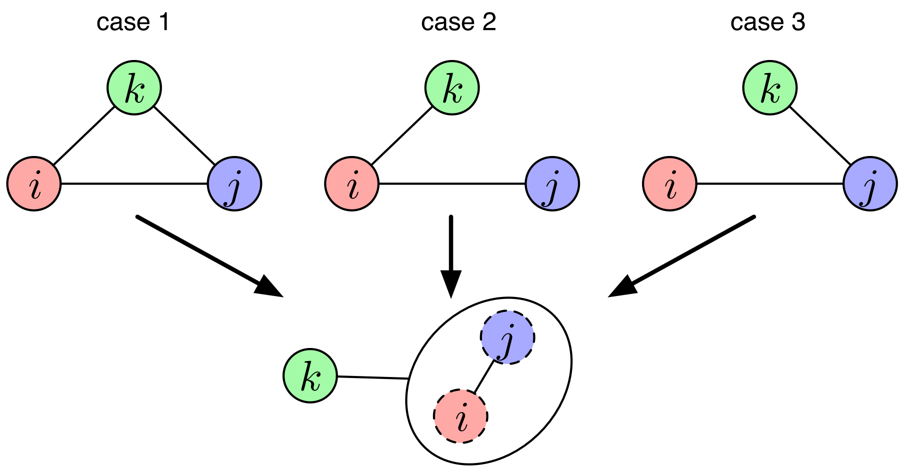

Figure 2: The three possible cases for connectivity between a pair of communities i and j that are to be merged, and a third, connected community k. Each case yields the same result, but we apply a different update equation ΔQ_{jk} to determine the change in modularity for a subsequent merge between j and k.

and $a_i = k_i/2m$.

When we merge two groups i and j, any groups that are not adjacent to either i or j will not change their values of ΔQ. Thus, when we merge, we only need to update a few elements of the sparse matrix and their corresponding entries in the max-heap. For instance, if we merge i and j, labeling the combined group j, then we only need to update the jth row and column, and remove the ith row and column altogether from ΔQ (do you see why?). To update the values in the jth row and column, we use the following update equation:

$$\Delta Q'_{jk} = \begin{cases} \Delta Q_{ik} + \Delta Q_{jk} & \text{if } k \text{ is connected to both } i \text{ and } j \\ \Delta Q_{ik} - 2a_j a_k & \text{if } k \text{ is connected to } i \text{ but not } j \\ \Delta Q_{jk} - 2a_i a_k & \text{if } k \text{ is connected to } j \text{ but not } i \end{cases} .$$

Figure 2 shows the three cases visually, where by convention we say that we insert community i into community j (i.e., we give the newly merged community the name j). Each case yields the same resulting connectivity, with j and k being connected, but the three cases tell us how to calculate the weight for the j, k merge from the weights on the original i, j, k edges. The update equations are derived by applying Eq. (3) to each of the three cases. (At-home exercise: derive these update equations.)

1.3.3 Running time analysis

To analyze how long this version of the algorithm takes, let us denote the degrees of i and j in the partition—i.e., the numbers of neighboring groups—as $|i|$ and $|j|$, respectively. The first operation in a step of the algorithm is to update the jth row. To implement the first update equation (a group that neighbors both i and j), we insert the elements of the ith row into the jth row, summing them wherever an element exists in both columns. Since we store the rows as balanced binary trees, each of these $|i|$ insertions takes $O(\log |j|) \leq O(\log V)$ time. We then update the other elements of the jth row, of which there are at most $|i| + |j|$, according to the second and third update equations (groups that neighbor either i or j, but not both). In the kth row, we update a single element, taking $O(\log |k|) \leq O(\log V)$ time, and there are at most $|i| + |j|$ values of k for which we have to do this. All of this thus takes $O((|i| + |j|) \log V)$ time.

We also have to update the overall max-heap H. Since we have changed the maximum element on at most $|i| + |j|$ rows (one for each neighbor of i and j), we need to do at most $|i| + |j|$ updates of H, each of which takes $O(\log V)$ time, for a total of $O((|i| + |j|) \log V)$.[12]

Finally, the update $a'_j = a_j + a_i$ (and $a_i = 0$) is trivial and can be done in constant time.

Since each join takes $O((|i| + |j|) \log V)$ time, the total running time is at most $O(\log V)$ times the sum over all nodes of the dendrogram of the degrees of the corresponding groups. Let us make the worst-case assumption that the degree of a group is the sum of the degrees of all the vertices in the original network comprising it. In that case, each vertex of the original network contributes its degree to all of the groups it is a part of, along the path in the dendrogram from it to the root.

If the dendrogram has depth d, then there are at most d nodes in this path, and since the total degree of all the vertices is $2E$, we have a running time of $O(Ed \log V)$. In the worst case, the dendrogram is unbalanced, $d = V$ and the running time is $O(EV \log V)$; in the best case, the dendrogram is balanced, $d = \log V$ and the running time is $O(E \log^2 V)$, which is $O(V \log^2 V)$ on a sparse graph.

1.3.4 Some results: Amazon co-purchasing network

Here's an example of running this algorithm on some real data: a co-purchasing or "recommender" network from `amazon.com`. Amazon sells a variety of products, particularly books and music, and as part of their web sales operation they list for each item A a number of other items most frequently purchased by buyers of A. We can represent this information as a directed network in which vertices represent items and there is a edge from item A to another item B if B was frequently purchased by buyers of A. For simplicity, we'll ignored the direction of the edges. The

[12] This could be improved slightly using a Fibonacci heap instead of a binary heap.

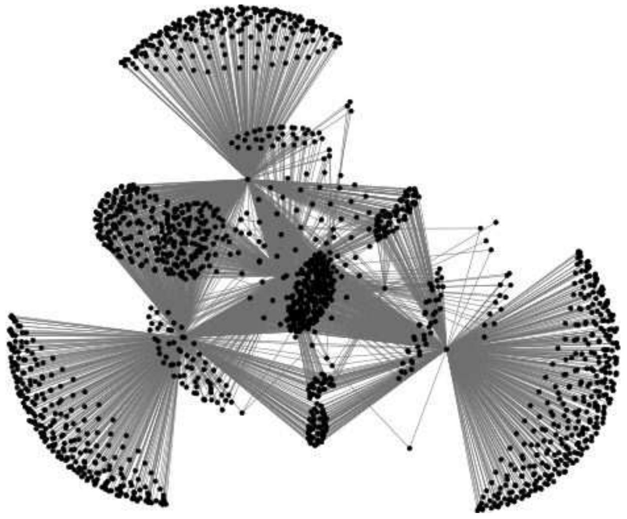

Figure 3: A visualization of the group structure at maximum modularity. Note that the some major groups have a large number of "satellite" groups connected only to them (top, lower left, lower right). Also, some pairs of major groups have sets of smaller groups that act as "bridges" between them (e.g., between the lower left and lower right, near the center).

network data is the set of items listed on the Amazon web site in August 2003, plus the connections. We'll focus on the largest component of the network, which has $409,687$ items and $2,464,630$ edges.

The dendrogram produced by the algorithm is far too large to visualize well, but Fig. 3 illustrates the modularity over the course of the algorithm as vertices are joined into larger and larger groups. The maximum value is $Q = 0.745$, which corresponds to a partition with 1684 groups, containing an average of 243 items each. Fig. 2 gives a visualization of the clustered network, including the major groups, smaller "satellite" groups connected to them, and "bridge" groups that connect two major groups with each other.

Looking at the largest groups in the network, we find that they tend to consist of items (books, music) in similar genres or on similar topics. Table 1, gives informal descriptions of the ten largest groups, which account for about 87% of the entire network. The remainder is generally divided into small, densely connected groups that represent highly specific co-purchasing habits, e.g., major works of science fiction (162 items), music by John Cougar Mellencamp (17 items), and books about (mostly female) spies in the American Civil War (13 items). Not bad.[13]

[13]This material is mostly drawn from my 2004 article "Finding community structure in very large networks," by AC, M.E.J. Newman and C. Moore, *Physical Review E* **70**, 066111 (2004). In the interest of full disclosure, I also

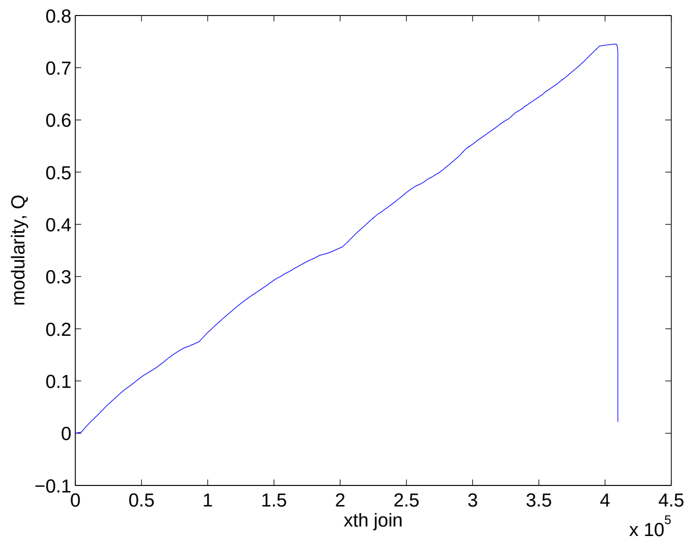

Figure 4: The modularity Q over the course of the algorithm (the x-axis shows the number of joins). Its maximum value is $Q = 0.745$, where the partition consists of 1684 groups.

2 For Next Time

1. Guest lecture by Prof. John Black on Wednesday, March 13

recently wrote a paper on some of the problems with interpreting the results of modularity maximization algorithms for real-world networks; see "The performance of modularity maximization in practical contexts," by B.H. Good, Y.-A. de Montjoye and AC, *Physical Review E* **81**, 046106 (2010).

Rank	Size	Description
1	114538	general interest: politics; art/literature; general fiction; human nature; technical books; how things, people, computers, societies work, etc.
2	92276	the arts: videos, books, DVDs about the creative and performing arts
3	78661	hobbies and interests I: self-help; self-education; popular science fiction, popular fantasy; leisure; etc.
4	54582	hobbies and interests II: adventure books; video games/comics; some sports; some humor; some classic fiction; some western religious material; etc.
5	9872	classical music and related items
6	1904	children's videos, movies, music and books
7	1493	church/religious music; African-descent cultural books; homoerotic imagery
8	1101	pop horror; mystery/adventure fiction
9	1083	jazz; orchestral music; easy listening
10	947	engineering; practical fashion

Table 1: The 10 largest groups in the `amazon.com` network, which account for 87% of the vertices in the network.