Components, Partitioning and Clustering

Vincenzo Bonifaci

March 28, 2017

1 Cliques, density, and k-cores

See the slides.

2 Independent paths and k-connectivity

Two paths between u and v are node-independent (or *node-disjoint*) if they do not share any node except for u and v.

Two paths between u and v are edge-independent (or *edge-disjoint*) if they do not share any edge. If two paths between u and v are node-independent, they are also edge-independent, but the reverse is not true.

The number of independent paths between two nodes is called the *connectivity* of the two vertices. It can be thought as a measure of how strongly connected the two nodes are.

A *node cut set* for u and v is a set of nodes whose removal disconnects u from v. Similarly, an *edge cut set* for u and v is a set of edges whose removal disconnects u from v. A *minimum cut set* for u and v is the smallest cut set that disconnects u and v. A minimum cut set need not be unique.

Theorem 2.1 (Menger's Theorem)**.** *If there is no cut set of size less than k between a given pair of nodes, then there are at least k independent paths between the same nodes.*

Note that the theorem is true for both situations: 1) comparing edge cut sets to edge indepedent paths, and 2) comparing node cut sets to node independent paths.

Therefore, the size of the minimum node (respectively, edge) cut set that disconnects a given pair of nodes in a network is equal to the node (respectively, edge) connectivity of the same nodes.

Menger's Theorem is closely related to the maximum flow / minimum cut theorem, which applies to networks where the edges have weights (often also called *capacities*). In this case, a minimum edge cut set is an edge cut set with the smallest sum of weights that separates a given pair of nodes.

Theorem 2.2 (Max-flow/min-cut Theorem)**.** *The maximum flow between a given pair of nodes in a network is equal to the sum of the weights on the edges of the minimum edge cut set that separates the same two nodes.*

A maximum flow in a weighted graph (with rational weights) can be computed relatively efficiently. For example, in time $O(m^2n)$ with the Edmonds-Karp algorithm, or in time $O(mF)$, where F is the maximum flow value, with the Ford-Fulkerson algorithm.

The maximum flow algorithms can, in particular, be used to determine the edge connectivity between any two vertices s and t: apply the algorithm to a network where the capacity of every edge is 1; the value of the maximum flow between s and t will be the value of the edge connectivity between s and t.

To determine vertex connectivity, we can use the following reduction to the edge connectivity problem. Assume that the graph is directed (if it is not, first replace each edge by two opposite directed edges). Then replace every node with two nodes separated by a directed edge; all original incoming edges connect to the first of these two, and all outgoing edges to the second. The value of the edge connectivity between s and t in the new digraph is equal to the value of the node connectivity in the original graph.

Exercise 2.1. Prove that the edge and vertex connectivity between two nodes of a simple connected graph (that is, without repeated edges) can both be computed in time $O(mn)$. Hint: use the Ford-Fulkerson algorithm.

3 Graph bipartitioning

In the graph bipartitioning problem we want to divide a graph into two parts of prescribed sizes n_1 and n_2. The goal is to minimize the number of edges cut by the resulting partition. This problem is NP-hard, so here we discuss some heuristics.

3.1 Local search: the Kernighan-Lin heuristic

A popular approach is the *Kernighan-Lin* heuristic, which is a *local-search* type algorithm. Let $e(S, T)$ be the size of a partition (S, T) (our objective function). This is the algorithm:

1. Start with an arbitrary partition of the nodes into two sets S, T of size n_1, n_2 respectively (for example, a random one). Initially, all nodes are unmarked. A new "round" of the algorithm starts. Let $S^* := S$, $T^* := T$.

2. Find the unmarked pair $(u, v) \in S \times T$, such that swapping u and v *decreases the cut size the most*. Then swap u and v:
$$S := S \setminus \{u\} \cup \{v\},$$
$$T := T \setminus \{v\} \cup \{u\}.$$
(Note: we perform the swap even if the decrease is *negative*, that is, it is actually an *increase* in cut size). Also, mark u and v.

3. If $e(S, T) < e(S^*, T^*)$, set $S^* := S$, $T^* := T$.

4. If there are more unmarked pairs, go back to step 2. When no swappable unmarked pair exists, the round ends. Go back to 1., but instead of starting with a random partition, start with (S^*, T^*).

5. The algorithm ends when $e(S^*, T^*)$ stops decreasing, or after a maximum number r of rounds.

How do we implement the algorithm efficiently? Let's consider these values for $(u, v) \in S \times T$: $\delta_S(u)$ (the degree of u within S), $\delta_T(u)$, $\delta_S(v)$, $\delta_T(v)$, and A_{uv} (the entry of the adjacency matrix corresponding to the pair (u, v)). We call $\delta_S(u)$ the *internal degree* of node u, while $\delta_T(u)$ is its *external degree*. For $v \in T$, $\delta_S(v)$ is the external degree and $\delta_T(v)$ is the internal degree. We can show the following.

Lemma 3.1. *The decrease in cut size when swapping the unmarked pair $(u, v) \in S \times T$ is exactly*

$$\delta_T(u) - \delta_S(u) + \delta_S(v) - \delta_T(v) - 2A_{uv}.$$

Proof. Let C be the cost due to all edges between S and T that are not incident to u or v. Then

$$\text{cost}_{\text{old}} = C + \delta_T(u) + \delta_S(v) - A_{uv},$$

while

$$\text{cost}_{\text{new}} = C + \delta_S(u) + \delta_T(v) + A_{uv}.$$

So, $\text{cost}_{\text{old}} - \text{cost}_{\text{new}} = \delta_T(u) - \delta_S(u) + \delta_S(v) - \delta_T(v) - 2A_{uv}$. □

Thus, to implement step 2 of the algorithm, we keep track of the difference between external degree and internal degree of each node:

$$\Delta(z) \stackrel{\text{def}}{=} \begin{cases} \delta_T(z) - \delta_S(z) & \text{if } z \in S, \\ \delta_S(z) - \delta_T(z) & \text{if } z \in T. \end{cases}$$

The Lemma says that if we swap u and v, the decrease in the objective function is

$$\text{cost}_{\text{old}} - \text{cost}_{\text{new}} = \Delta(u) + \Delta(v) - 2A_{uv}. \tag{1}$$

The Δ values can be computed at the beginning of each round in $O(n)$ time for each node ($O(n^2)$ total). After each swap, we need to update the Δ values of the (remaining) unmarked nodes, using the fact that, if $z \in S$ is unmarked,

$$\Delta_{\text{new}}(z) = \begin{cases} \Delta_{\text{old}}(z) + 2A_{uz} - 2A_{vz} & \text{if } z \in S, \\ \Delta_{\text{old}}(z) - 2A_{uz} + 2A_{vz} & \text{if } z \in T. \end{cases}$$

Recomputing each such value costs $O(1)$ time; so we use $O(n-k)$ time after k pairs have been swapped, since we only recompute the values for the $O(n-k)$ unmarked nodes. In one round, the total update time grows as

$$n - 1 + n - 2 + \ldots + 2 + 1 = O(n^2).$$

Every round we swap at most $n/2$ pairs. Each time, to identify an optimal pair, we evaluate the expression in Equation (1) for all $O(n_1 n_2) = O(n^2)$ unmarked pairs. This gives a total time of $O(n^3)$ per round, or $O(rn^3)$ in total for r rounds. With an additional trick, one can get a "typical case" bound of $O(rn^2 \log n)$, although in the worst case it is still $O(rn^3)$.

The Kernighan-Lin heuristic often produces good quality solutions in practice, but it has no quality guarantee and it may be quite slow.

3.2 Spectral partitioning

Another popular approach to partitioning uses the matrix properties of the graph Laplacian – in particular, the second smallest eigenpair. This is why this approach is called *spectral partitioning*.

Note that, if $c : V \to \{S, T\}$ is the function that assigns every node its "side" (either S or T), we can write $e(S,T) = \frac{1}{2} \sum_{i,j:\, c_i \neq c_j} A_{ij}$. For $i \in V$, we introduce variables

$$s_i = \begin{cases} +1 \text{ if } c_i = S \\ -1 \text{ if } c_i = T. \end{cases}$$

Note that any such vector s has Euclidean length $\sqrt{n}$, since $\sum_{i \in V} s_i^2 = n$.

Also, observe that

$$\frac{1}{2}(1 - s_i s_j) = \begin{cases} 1 \text{ if } c_i \neq c_j \\ 0 \text{ if } c_i = c_j. \end{cases}$$

So we can write

$$e(S,T) = \frac{1}{4} \sum_{i,j \in V} A_{ij}(1 - s_i s_j).$$

We can rewrite

$$\sum_{i,j} A_{ij} = \sum_i \deg(i) = \sum_i \deg(i) s_i^2 = \sum_{i,j} \deg(i) \delta_{ij} s_i s_j,$$

where $\deg(i)$ is the degree of i, and δ_{ij} is the Kronecker symbol: $\delta_{ij} = 1$ if $i = j$, $\delta_{ij} = 0$ if $i \neq j$. Therefore

$$e(S,T) = \frac{1}{4} \sum_{i,j} (\deg(i)\delta_{ij} - A_{ij}) s_i s_j = \frac{1}{4} \sum_{i,j} L_{ij} s_i s_j,$$

where $L_{ij} = \deg(i)\delta_{ij} - A_{ij}$ is the (i,j)-th element of the graph Laplacian matrix. In matrix form,

$$e(S,T) = \frac{1}{4} s^\top L s.$$

In other words, our goal is to minimize the quadratic form $s^\top L s$, subject to $s_i \in \{-1, +1\}$ for all i, and $\sum_{i \in V} s_i = n_1 - n_2$. In general this is NP-hard to optimize exactly. We will take the following approach: instead of ranging in the set $\{-1, +1\}^n$, we first allow the vector s to be *any vector in* $\mathbb{R}^n$

of total length $\sqrt{n}$ (we will later see how to get back to the original domain). That is, we allow $s \in \mathbb{R}^n$, but we require

$$\sum_{i \in V} s_i^2 = n.$$

We also still require that $\sum_i s_i = n_1 - n_2$ as before, which can also be expressed as

$$\mathbf{1}^\top s = n_1 - n_2.$$

We can solve the relaxed problem optimally, by the method of *Lagrangian multipliers*. The theory of Lagrangian multipliers tells use that if $s \in \mathbb{R}^n$ is an optimal solution of the relaxed problem, then there exist two "multipliers" λ, $\mu \in \mathbb{R}$ such that, for all i,

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

This implies (after redefining $\mu \leftarrow \mu/2$)

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

which in matrix notation is

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

Recall that $\mathbf{1}$ is always an eigenvector of L, with eigenvalue 0, so multiplying both sides by $\mathbf{1}^\top$ gives

$$0 = \lambda \mathbf{1}^\top s + \mu \mathbf{1}^\top \mathbf{1} = \lambda(n_1 - n_2) + \mu n,$$

equivalent to

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

If we define the vector

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

then the Lagrangian condition tells us that

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

so λ *must be* an eigenvalue of L, and x its associated eigenvector! Notice also that

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

so x should be orthogonal to $\mathbf{1}$ (so λ cannot be the zero eigenvalue of L!). Other than this, it seems that we can choose any eigenpair (λ, x) of L, but which one gives the smallest cut? Note that our objective function satisfies

$$e(S,T) = \frac{1}{4} s^\top L s = \frac{1}{4} x^\top L x = \frac{1}{4}\lambda x^\top x.$$

On the other hand,

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

and hence

$$e(S,T) = \frac{n_1 n_2}{n}\lambda.$$

Since n_1, n_2 and n are fixed, it means that the smaller the λ, the better. We know that $\lambda_1 = 0$ and $0 \leq \lambda_2 \leq \ldots \leq \lambda_n$. We cannot pick λ_1 because then x would not be orthogonal to $\mathbf{1}$. So the optimal choice is to pick λ_2 and its associated eigenvector, v_2. That is, we pick x proportional to v_2, but scaled to satisfy $x^\top x = 4n_1 n_2/n$. This is indeed the optimal solution to the relaxed problem. In terms of the vector s,

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

At this point we know the optimal solution of the relaxed problem. However, remember that what we really wanted is to have $s_i \in \{-1, +1\}$. A heuristic to do so is to make the inner product of the binary vector with the "desired" (but non-binary) vector as large as possible: that is, maximize

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

This is achieved by setting $s_i = +1$ for the n_1 nodes with largest x_i value and $s_i = -1$ for the other n_2 nodes.

There is also another natural solution that we can consider: above we assumed that $|S| = n_1$, $|T| = n_2$, but the symmetric situation is equally valid, therefore we can also assign $s_i = +1$ for the n_2 nodes with largest x_i value and $s_i = -1$ for the other n_1 nodes. Clearly, among these solutions we prefer the one with the smallest cut value. In summary, our heuristic is:

1. Compute the second eigenpair (λ_2, v_2) of the graph Laplacian.
2. Each element of v_2 yields a score for a node of G. Sort the scores.
3. One candidate partition puts the n_1 nodes with largest score in S and the others in T.
4. The other candidate partition puts the n_1 nodes with smallest score in S and the others in T.
5. Among the two partitions above, return the one with the lower cut value.

3.3 Bipartitioning without prescribed sizes

Above we assumed that we know the sizes (n_1, n_2) of the partition that we are looking for. However, often one does not know n_1 and n_2 in advance. In this case, it does not make sense to just minimize $e(S,T)$ – most likely we would get a single node in one of the two partitions. Instead, it is more appropriate to minimize a value normalized by the partition sizes (to avoid excessive imbalance). One possibility is the *cut ratio*, also called *sparsity* of the cut:

$$\sigma(S,T) = \frac{e(S,T)}{|S| \cdot |T|}.$$

Another similar possibility is the *conductance* of the partition:

$$\phi(S,T) = \frac{e(S,T)}{\min(\mathrm{vol}(S), \mathrm{vol}(T))},$$

where $\mathrm{vol}(U)$ is the sum of the degrees of the nodes in a subset $U \subseteq V$. The *conductance of a graph* G is the minimum conductance of a partition of G:

$$\phi_G = \min_{(S,T)} \phi(S,T),$$

where (S,T) ranges over all possible nontrivial partitions of V.

Finding a partition with minimum conductance is still an NP-hard problem. However, it admits a guaranteed approximation algorithm, based on a spectral partitioning approach similar to the one we discussed for the case of prescribed sizes. Here is the algorithm:

1. Compute the second eigenpair (λ_2, v_2) of the *normalized* graph Laplacian[1].
2. Each element of v_2 yields a score for a node of G. Sort the scores to obtain an ordering $v_1, \ldots, v_n$ of the nodes of G.
3. Consider the $n-1$ cuts of the form

   $$S = \{v_1, \ldots, v_i\}, T = \{v_{i+1}, \ldots, v_n\}$$

   for $i = 1, \ldots, n-1$.
4. Return the minimum conductance cut among those $n-1$ cuts.

We omit the proof of the following result (for a proof, see F.Chung, ICCM 2007, Theorem 1).

Theorem 3.2. *The partition (S,T) found by the above algorithm satisfies*

$$\phi_G \leq \phi(S,T) \leq 2\phi_G^{1/2}.$$

[1]The normalized Laplacian $\mathcal{L}$ is obtained by normalizing each row and column of L by the square root of the degree of the corresponding node. It has entries $\mathcal{L}_{ij} = -1/\sqrt{\deg(i)\deg(j)}$ for $(i,j) \in E$, and $\mathcal{L}_{ii} = 1$ on the diagonal.

4 Clustering and community detection

4.1 Modularity

Let c_i be the class (category) or type of node i. The total number of edges running between nodes of the same type is

$$\sum_{(i,j)\in E} \delta(c_i, c_j) = \frac{1}{2} \sum_{i,j\in V} A_{ij}\delta(c_i, c_j),$$

where $\delta(c_i, c_j) = 1$ if $c_i = c_j$ and 0 otherwise. We compare this quantity with the expected number of edges between nodes if edges are placed at random. This is

$$\frac{1}{2}\sum_{i,j} \frac{k_i k_j}{2m}\delta(c_i, c_j),$$

where k_i is the degree of i. The normalized difference between the above two quantities is a measure of the assortative mixing of the network, called the *modularity* of the network:

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

The modularity is a real number between -1 and $+1$ (why?). A positive modularity implies associative mixing (nodes of the same kind tend to connect more with each other); a negative modularity implies dissociative mixing (nodes of the same kind tend to connect less with each other).

4.2 Girvan-Newman

The Girvan-Newman method is a divisive hierarchical clustering algorithm. It produces a sequence of partitions of the nodes of the graph, where each partition is a refinement of the previous partition. At the end of the sequence it returns the best partition seen so far.

1. Initially, all nodes have the same type.
2. Compute the modularity. If it is higher than the best modularity computed so far, store this value (and the corresponding partition).
3. Find the edge(s) of highest betweenness.
4. Remove it (or them) from the graph. If the graph splits into multiple components, this is a level of regions in the partitioning of the graph: label the components with different types (nodes in the same component get the same type, nodes in different components get different types).
5. If edges are still present, go back to point 2.
6. At the end, return the partition with highest modularity.

4.3 Greedy Modularity Maximization

Since the Girvan-Newman method is based on recomputing the betweennesses of all edges many times, it can be quite slow. Newman proposed a faster algorithm based on agglomerative hierarchical clustering.

1. Initially, each node has a distinct type.
2. Compute the modularity. If it is higher than the best modularity computed so far, store this value (and the corresponding partition).
3. Join the pair of communities that results in the largest increase (or smallest decrease) of modularity. Update the type labels accordingly.
4. If there is still more than one type, go back to point 2.
5. At the end, return the partition with highest modularity.

The algorithm can be implemented faster than Girvan-Newman (see Newman's 2004 paper for details; implementing this algorithm and testing it on real network could make for an interesting project).

4.4 Spectral Modularity Maximization

We can apply a spectral approach to modularity maximization assuming two communities. If $s_i \in \{-1, +1\}$ is the variable indicating where node i belongs to community 1 or community 2, notice that $(s_i s_j + 1)/2$ equals 1 when i and j are in the same group and 0 otherwise, therefore $\delta(c_i, c_j) = (s_i s_j + 1)/2$. Define

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

so the modularity can be written as

$$Q = \frac{1}{4m} \sum_{i,j} B_{ij}(s_i s_j + 1) = \frac{1}{4m} \sum_{i,j} B_{ij} s_i s_j + 0.$$

In matrix terms,

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

where the $n \times n$ matrix B has elements B_{ij}. B is called the *modularity matrix*.

The problem is again hard given the binary nature of the s vector. So we relax the binary constrain to allow $s \in \mathbb{R}^n$, but we keep the constraint $s^\top s = n$.

Using the technique of Lagrangian multipliers, we find there exists β such that for each i,

$$\frac{\partial}{\partial s_i} \left(\sum_{j,k} B_{jk} s_j s_k + \beta(n - \sum_j s_j^2) \right) = 0.$$

That is, $\sum_j B_{ij}s_j = \beta s_i$, which in vector form is $Bs = \beta s$. So s is an eigenvector of B. The corresponding modularity value is

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

so for maximum modularity we want to have β as large as possible. In other words we want β to be the largest (most positive) eigenvalue of the matrix B, and the optimal solution to the relaxed problem is its associated eigenvector u_1.

Again, we cannot really set $s = u_1$ in the original problem, because of the binary constraints $s_i \in \{-1, +1\}$. But, heuristically, we can maximize $s^\top u_1 = \sum_i s_i(u_1)_i$, i.e. we pick $s_i = +1$ if $(u_1)_i > 0$ and $s_i = -1$ if $(u_1)_i < 0$.

A potential issue with the efficiency of this method is that the B matrix is *not* sparse (differently from the Laplacian). Indeed, B could have n^2 nonzero entries. So, if we apply the power method blindly, we incur a high cost per iteration. But, the adjacency matrix A *is* sparse, and note that B differs from A just by a very special term (that has rank 1):

$$B = A - \frac{\boldsymbol{k}\boldsymbol{k}^\top}{2m},$$

where $\boldsymbol{k}$ is the vector of node degrees. So we can still compute a matrix-vector product Bx in $O(m+n)$ operations, by first computing the number $\boldsymbol{k}^\top x$ (in $O(n)$ time), and then computing

$$Bx = Ax - \frac{\boldsymbol{k}(\boldsymbol{k}^\top x)}{2m}$$

for a cost of $O(m+n)$. Using the power method, we can then compute u_1, in the following way. Note that u_1 is the eigenvector associated to the *most positive* eigenvalue of B, which may or may not be the dominant one (the dominant eigenvalue is either the most positive, or the most negative). After applying the power method once, we obtain a dominant eigenpair (β^*, u^*). If $\beta^* \geq 0$, then $u_1 = u^*$. Otherwise, we can compute u_1 by applying the power method to the matrix $B - \beta^* I$.

Exercise 4.1. Prove that, if $\beta^* < 0$, then u_1 is a dominant eigenvector of $B - \beta^* I$.