11/7: Lecture 18. Clustering Heuristics 1

Wednesday, November 08, 2006
8:49 AM

Today we will talk about clustering and partitioning in graphs, and sometimes in data sets. Partitioning usually refers to the problem of dividing a graph into a small number of pieces. Clustering often refers to the problem of dividing a graph into many small pieces, which may overlap. It also refers to the problem of finding a single cluster around a particular vertex.

What constitutes a good cluster is an ill-defined concept. But, if we have to create a criterion, then it is good to use one of them from our early lectures on partitioning. If C is a set of vertices in an unweighted graph, we might use the ratio of a cut:

$$\frac{|\partial(C)|}{|C|}$$

where $|\partial(C)|$ = # edges leaving C
$|C|$ = # verts in C

A better measure, which makes especially good sense in a weighted graph, is the conductance of the cut (also called the normalized cut objective):

$$\frac{wt(\partial(C))}{vol(C)}$$

$wt(\partial(C))$ ← sum of weight of edges leaving C

$vol(C)$ ← sum of weighted degrees of vertices in C

Our goal in both cases is to find sets C in which these quantities are low.

If we divide the vertices of a graph into sets $C_1, \ldots, C_k$

then we could measure the sum of these quality measures:

$$\sum_{i=1}^{k} \frac{wt(\partial(C_i))}{vol(C_i)}$$

Single-link clustering:

Let's begin by looking at a very simple clustering heuristic, called single-link clustering. We usually think of applying it to point sets in a vector space. But it really can be applied on any weighted graph. Let x_1, …, x_n be the points. Initially, the algorithm creates one cluster for each point, so cluster C_i = {x_i}.

The algorithm then successively merges clusters together, until the desired number is obtained (say k). The rule for picking which pair of clusters to merge is simple:

Find the points x_i and x_j in different clusters minimizing the distance between x_i and x_j. Then, merge their clusters.

We just repeatedly apply this rule until there are only k clusters left.

We just repeatedly apply this rule until there are only k clusters left.

This clustering algorithm has three nice properties. The first is that it produces a hierarchy of clusters, so that if you unwind the process, you can find sub-clusters inside each cluster. The second is that if you have some performance criteria in mind, as opposed to just a number of clusters, you can merge clusters until that criteria is met. The third is that it does produce some guarantee.

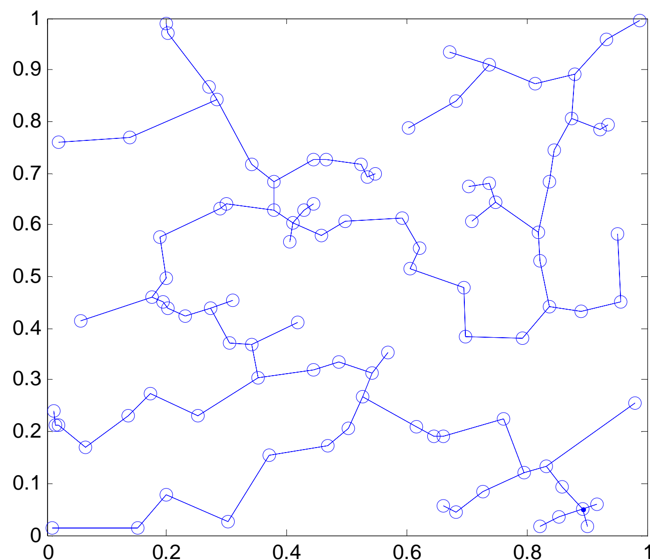
Let C_1, …, C_k be the k clusters output by the algorithm. If we define the distance between a pair of clusters to be the least distance between a pair of points, one from each cluster, then this algorithm maximizes the minimum over pairs of clusters of their distance. That is, given any different arrangment into k clusters, D_1, …, D_k, there will be an x_i and x_j in different clusters (say x_i is in D_a and x_j is in D_b), such that dist(x_i, x_j) is at most the distance between each pair of clusters C_1, …, C_k.

To see why this is true, let's consider the case in which all pairs of distances d(x_i, x_j) are distinct, and show that in the clustering D_1, …, D_k there has to be a pair that is closer. If D_1, …, D_k is a different clustering than C_1, …, C_k, then there must be some cluster C_a such that some of its vertices are in a cluster D_b and some of its other vertices are in D_c. Say that x_i is in D_b and x_j is in D_c. So, The distance between D_b and D_c is at most dist(x_i, x_j). But, every pair of vertices in different clusters in C_1, …, C_k must be further apart than x_i and x_j, as otherwise the algorithm would have chosen to merge that pair of vertices before merging x_i and x_j, and it didn't.

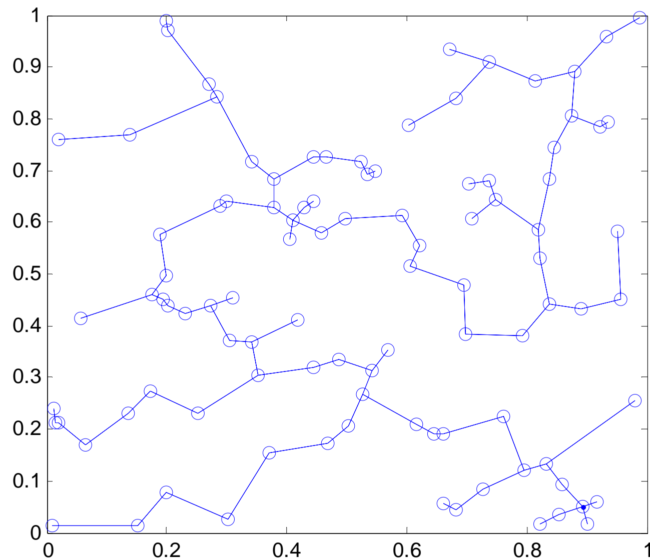
In graph theoretic terms, this algorithm is computing a Minimum Spanning Tree. To make a graph of this problem, we create n vertices, and set the weight of the edge between node i and node j to be dist(x_i,x_j). A minimum spanning tree is a spanning tree on which the sum of the weights is minimum. One can show that the pairs of vertices that are merged by this algorithm correspond exactly to the edges in a minimum spanning tree.

Let's look at a picture. Here are a bunch of points
in the plane, and the tree of edges that get merged.

```
>> load nov7.mat
>> primdemo
```

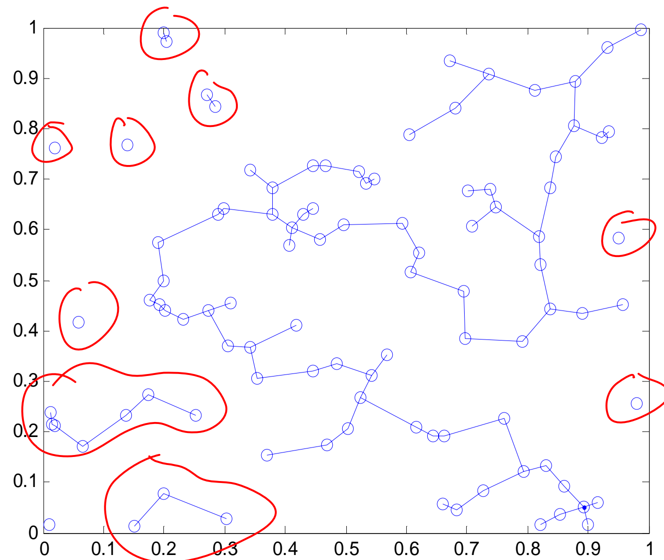


```
>> load nov7.mat
>> primdemo
```



Here's what happens if we just wanted to make 10 clusters. We get them by deleting the 10 longest edges in this tree.

I've circled the clusters (except the big one).



So, this isn't all that impressive an algorithm. But, we have to start somewhere. Now, let's see some better ones.

Personalized Page Rank Vectors

These appeared in the original paper of Brin and Page. Recall that the ordinary page-rank vector is the vector v that satisfies

$$v = \alpha \mathbb{1} + (1-\alpha) v \cdot W$$

where W is the walk-matrix. Recall that the α1 term represents a probability α of jumping to a random node in the graph.

When constructing the personalized page-rank vector, we pick a particular start node s, and we replace the jump to a random node with a jump back to s. Abusing notation by writing s for the vector that is all zeros, with a one at s, this gives the equation

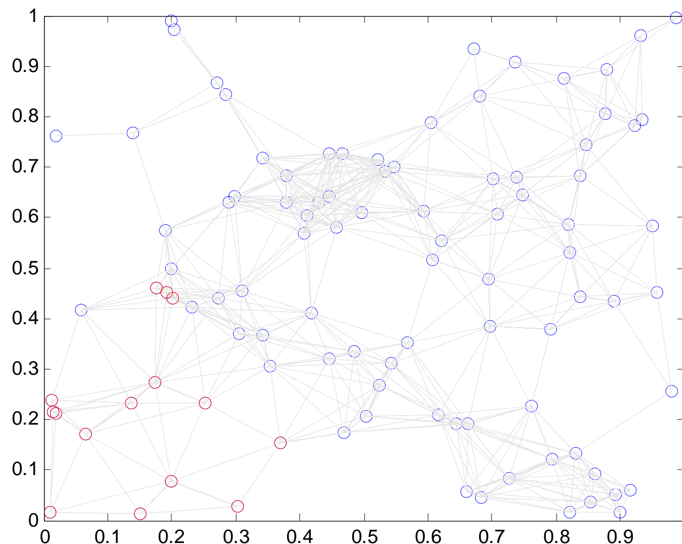
$$v = \alpha s + (1-\alpha) v \cdot W$$

This equation has the solution:

$$v = \alpha s + \alpha \sum_{t=1}^{\infty} (1-\alpha)^t (s W^t)$$

The nodes that have the highest values in this vector can be viewed as those that are most interesting in terms of s.

If you run the code prdemo, Matlab will compute this vector, and then successively color the nodes in order of their rank in this vector (each time you press a key). To do this experiment, I made the weight of an edge the reciprocal of the distance between the corresponding points. Here is what it looked like after 15 or so clicks.



By the way, the edges you see in light grey were drawn between each pair of points at distance less than 0.2.

In a recent paper "Local graph partitioning using PageRank vectors", Andersen,

Chung and Lang proved that these vectors can be used to find cuts in graphs as good as those found by the spectral method. They also construct an algorithm that very quickly approximates the personal PageRank vector, while looking at only a few times more vertices than are in the cluster that it outputs.
(of course, you can find this paper by Googling it)

Markov Cluster Algorithm

The Markov Cluster Algorithm (MCL) of Stijn van Dongen is an exciting new technique for dividing a graph into many small pieces. To explain it, let me give the matlab code, and then say what it does in words:

```
while and( i < maxiters, sum(sum(abs(m - m^2))) > 10^(-6)),
      d = diag(1./sum(m));
      m = m * d;
      m = m ^ 2;
      m = m .^ r;
      i = i + 1;
end
```

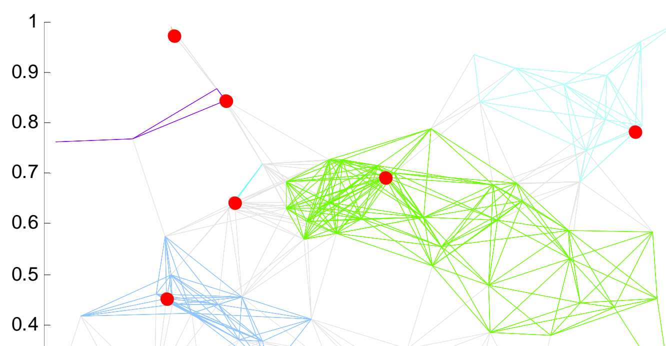
The first two lines take the matrix m and scale it so that all its column sums are 1.
The next line squares the matrix (after which the column sums are still 1).
The third line raises each entry of the matrix to the rth power.

We then repeat this process until it converges, detected here by testing for m being very close to its square.

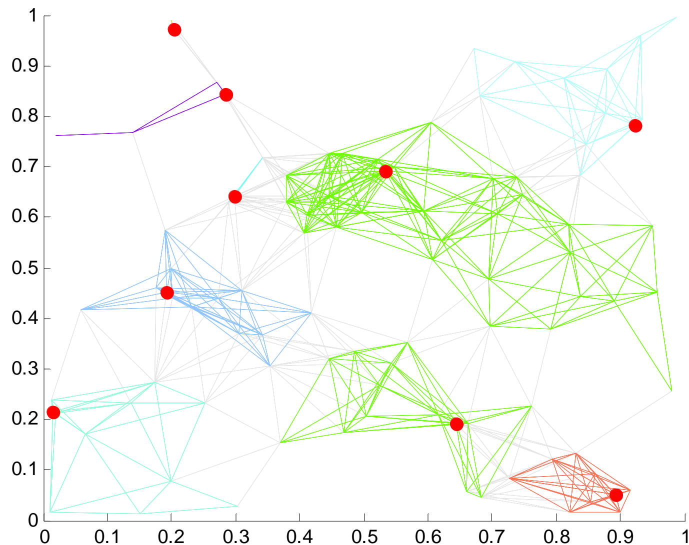
If r = 1, then this algorithm just simulates a random walk, and when it is done the graph will look like the walk matrix on a clique.

When r > 1, it tends to converge to a matrix with block structure, and these blocks reveal the clusters. In particular, most of the entries of the matrix m go to zero, and the connected components of the rest essentially give the clusters. I say "essentially" because there is a better way to define the clusters, and the clusters can overlap. But, we'll see that in the next lecture.

The larger r is, the more clusters we tend to find. Here are some examples.

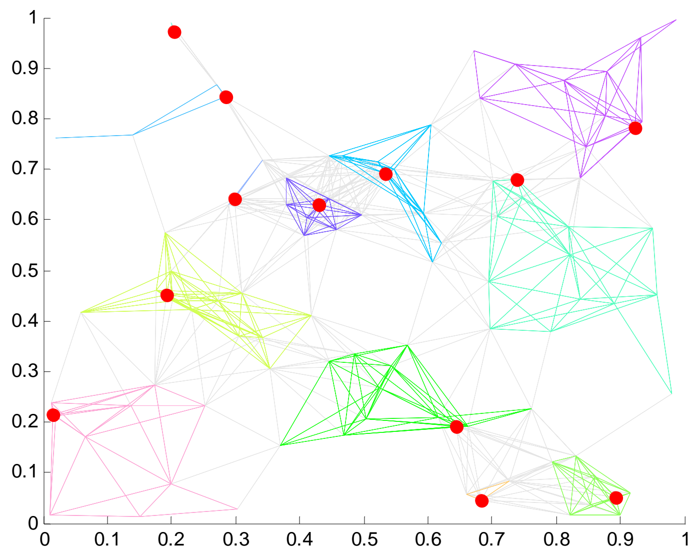


```
>> m = mcl(a,1.25);
>> plotmcl(a,xy,m)
```

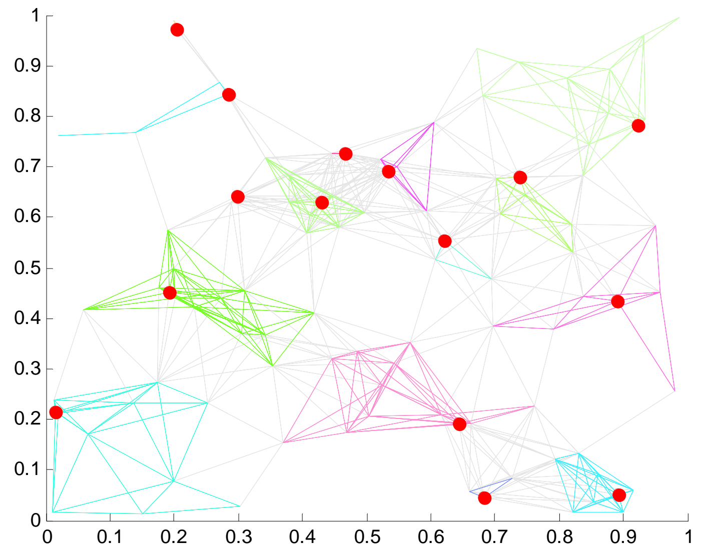


Again, the underlying graph is the graph in which nodes at distance less that 0.2 are connected, and the weight is the reciprocal of their distance. Edges within a cluster are all given the same color. The red nodes are identified as cluster centers, which we will see again next lecture. Let's finish by raising r, and looking at pictures.

```
>> m = mcl(a,1.3);
>> plotmcl(a,xy,m)
```



```
>> m = mcl(a,1.4);
>> plotmcl(a,xy,m)
```




```
>> m = mcl(a,1.5);
>> plotmcl(a,xy,m)
```