CSE 527
Lecture 7
Non-Hierarchical Clustering

Lecturer: Larry Ruzzo
Notes: Daniel Grossman (grossman@cs)

October 25, 2001

1 Brief Discussion of Graph Theory

A **graph** is defined a collection of n **vertices** (or **nodes**) and m **edges** connecting them. Formally, $G = (V, E)$, where E is a set of unordered pairs from V. Figure **??** shows an example graph: edge $1, 2 \in E$ but edge $1, 3 \notin E$. There are multiple ways of representing a graph in computer memory, from which we select the **adjacency matrix** for our discussion. In an adjacency matrix $A[n, n]$, $A_{ij} = 1$ if edge ij exists in G. The main diagonal is sometimes populated with 0s and sometimes with 1s; it makes no difference as long as the graph has no unit-length cycles. Here is the adjacency matrix corresponding to our example graph (notice that it is symmetric by definition):

$$\begin{bmatrix} . & 1 & 2 & 3 & 4 & 5 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 1 & 0 & 0 \\ 3 & 0 & 1 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \\ 5 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

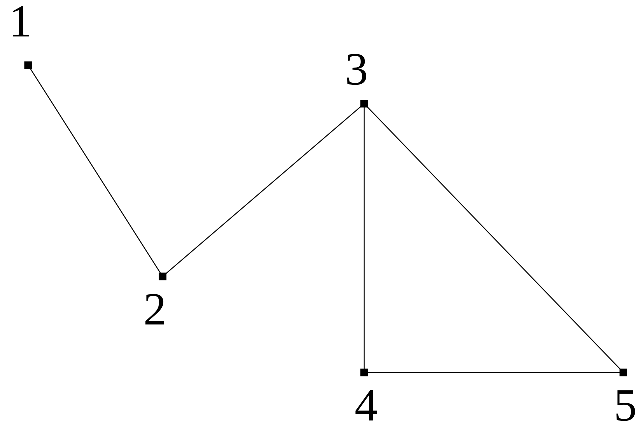

Figure 1: Example Graph

Cliques The clique is an important concept in graph theory. Also known as a **complete graph**, it is defined as a graph where every vertex is adjacent to every other. In computational biology we use cliques as a method of abstracting pairwise relationships such as protein-protein interaction or gene similarity. In the latter case, we might want to establish an edge between the vertices representing two genes if those two genes have, say, similar expression profiles over several timepoints of an experiment. If we were to find a clique in such a graph, we would have an important finding. A goal might be to find the largest clique in the graph.

Finding k-Cliques A useful question to then ask would be, *is there a k-clique in the graph*? I.e. is there at least one subgraph of at least size k whose vertices are completely connected to each other? Here is an algorithm which would allow us to solve this problem:

`Examine all subsets of size` k `and determine if any is a clique.`

From a logistical standpoint, the computer should have an easy time checking whether a graph is a clique: in the adjacency matrix a clique is represented simply by a subarray containing only 1s. The problem with the algorithm is that the number of subsets of size k in the graph is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} > \left(\frac{n}{k}\right)^k$$

which is far too many to check one by one. For example, in a graph of 100 vertices, if we would like to search for 10-cliques, we have

$$\binom{100}{10} > \left(\frac{100}{10}\right)^{10} = 10^{10}$$

subgraphs to examine! Considering that typical gene expression datasets contain *thousands* of genes (nodes) and that investigators are interested in cliques of several dozen, this algorithm would require an impractical amount of time.

Here is a more specific example of the k-clique problem: Given graph G of size n, is there a clique of size $n/20$ (i.e. 5% of the graph)? Since the clique size is a function of the problem input size, our naive algorithm would require more than $20^{(n/20)}$ subgraph examinations, meaning that the amount of work we would have to do is exponential in the size of the input.

In fact, no known algorithm for solving this problem can be proven to require an amount of work that is *less* than exponentially related to the size of the input. The problem is **NP-complete**, meaning that it lies in a category of problems for which no **polynomial-time** (i.e. requiring only an amount of work that is polynomially related to the problem input size) algorithms are known. If any problem in this category were shown to be solvable via a P-time (polynomial-time) algorithm, then that algorithm could be used to solve all the rest of the NP-complete problems. Since computer scientists have been looking unsuccessfully for such P-time algorithms for several decades, it is clear that we must take a different approach to answer our biological clustering questions.

2 Solving Clustering Problems

While the general theoretical problem of identifying cliques may be intractable, we can solve a simpler problem. Given a graph H that is a **clique graph** — i.e. a union of disjoint cliques (see Figure **??**) — that represents the true state of whatever similarity/interaction concept we are trying to discover, let us define G, the graph of data that we actually measure, to be the true graph (H) plus noise, where the noise

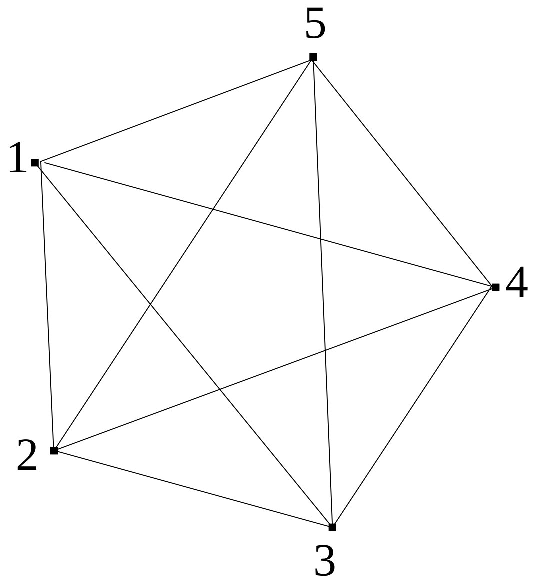

Figure 2: Clique Graph

is defined by a constant $\alpha < 1/2$ such that all edges and non-edges are switched on or turned off in G with probability α.

Given the true graph H it would be trivial to find the cliques in our data. Unfortunately, our experimental methods measure G. So, can we estimate H given G? Is there an algorithm to reconstruct H? In general, we've already shown the answer to be no. A happy medium that is often acceptable in practice is constructing a clique graph H' such that

$$|H' \oplus G| \leq |H \oplus G|$$

with probability $\geq 1 - \delta$. That is, we can create a clique graph that shares more edges with the observed graph than does the true clique graph. Most importantly, we can do this partial reconstruction of the true graph H in polynomial time.

2.1 Example

Let us consider a simple example problem. We have the following adjacency matrix A for the true graph H:

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

We have the following parameters for our algorithm:

- n = size of the graph
- α = an upper bound on the error rate in the data
- δ = the probability of failure of our algorithm

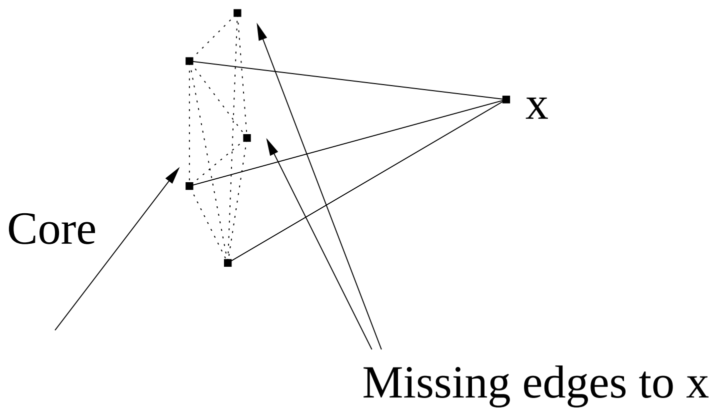

Figure 3: Core Graph

- γ = the size of the smallest clique we are looking to be able to reconstruct

Because $\alpha < 1/2$, we expect to see more ones than zeroes in the clique regions of G's adjacency matrix, and vice versa in the non-clique regions. Of course, we would never expect to be able to recover cliques of such small size as those in A above which is why we impose the γ limit. We do not stipulate that our algorithm must be able to find even very small cliques, which could easily be corrupted more than 50% simply by chance.

2.2 Practicality

Before we continue on to a discussion of our algorithm for this simpler problem, we pause to discuss why we bother to work on this problem at all. Would we realistically expect to find easy to resolve clique graphs in the real world, even in the absence of noise? The answer, of course, is that in biological problems, the relationships we are trying to learn are so complex that such clique graphs would indeed not exist. However, solving our simpler problem will help us find heuristics, answers, and insights for solving the real-life systems, as discussed in Ben-Dor, Shamir, & Yakhini [JCB '99], the paper in which the current algorithm was presented.

The bottom line of the "simpler problem" discussion is that Ben-Dor *et al.* arrive at an $O(n^2(\log n)^l$ algorithm for any fixed choice of $\alpha, \delta, \&\gamma$, where the constants depend heavily on those fixed values. For example, for $\gamma = .1, \alpha = .25, l$ turns out to be 600. So, while Ben-Dor's algorithm is polynomial in the size of the input, it is impractical. However, the theorem governing the validity of the algorithm's output is correct. Moreover, time bounds in theoretical papers are often conservative, meaning that the algorithm could perhaps work in practice for some non-trivial problems.

2.3 Brief Look at the Algorithm's Inner Workings

Omitting several key components of the approach, we focus on the key insight in the Ben-Dor algorithm, the **core**. Suppose H has only two cliques, one containing 90% of the data, and the other, 10%. We define a core as a set of vertices all located in one clique. Now consider some other vertex x not in the core. What do we expect of x? If it is in the same clique as the core, then (in the noisy graph G) the expected fraction of edges between x and the core is $1 - \alpha$. Conversely, if x is not in the same clique as the core, then the expected fraction of edges connecting the two is α. See Figure **??**.

So, given a core, we classify all vertices x based on what fraction of its possible edges to the core actually exist. If x has greater than $1/2$ of its edges connected to the core, then we decide that x is in the same clique. If less than 1/2 of the possible edges exist in G, we conclude that the vertex lies outside the clique. (Note that if our core is small to begin with, our shared-edges test does not have much statistical power.)

Suppose we start with a core of size $c \log n$ vertices. What is the chance of a vertex classification error if $\alpha = 0.1$? It is equivalent to

- the probability of having > 0.5 connectedness to the core when we expect 0.1, or
- the probability of having < 0.5 connectedness to the core when we expect 0.9.

Therefore, by picking an appropriate value of c, we can assure that we only misclassify some constant number of edges.

2.4 Generating the Core

How do we generate the core to begin with? There is the brute force option by which we analyze all $\binom{n}{\log n}$ cores of size $\log n$, but this set of cores is too large. Instead, we perform the following sampling procedure that seems to work quite well in practice:

```
Sample d log n vertices from G (probably picks vertices from both cliques)
   Sample d' log log n vertices from the first sample
   Partition these into cliques
   By brute force on these small vertex sets, find potential members of each core
Build these small cores into larger size cores using the original vertex samples
```

2.5 The Practical Algorithm

Is there a real heuristic practical algorithm to which the above approach leads us? Yes. It is based on the key *core attraction* concept.

First we pick a threshold t such that we add the current vertex under consideration to the current component under consideration (C) if the similarity of the vertex to the componet is greater than t. Later on we might want to remove early-placed members of the component is they no longer have high affinity to the resulting cluster.

We define $a(x)$ to be the average **affinity** of node x to the current cluster. Now ...

```
Repeat until everything clustered (freezing the cluster results each loop):
{
   Pick the vertex with the highest a(x) to members of C and add it to C.
   Repeat until no more changes (or until we feel like stopping):
   {
      For each vertex x ∉ C, if a(x) > t then add x to C
      For each x ∈ C, if a(x) < t then remove x from C
   }
}
```

At the end of the algorithm we perform one final pass during which every point is compared to all of the clusters and moved to the cluster with which it has the highest affinity. This corrects for local optima that may have been encountered along the way.

What does the practical algorithm have to do with the theoretical algorithm? Not much, aside from the concept of *affinity to cores*. We do not want to perform the brute force core enumeration, so instead we just start somewhere and reserver the right to remove points from clusters as the algorithm progresses.