Nils Gehlenborg (gehlenbo@u.washington.edu)

CSE 527 (Fall 2003) - Lecture Notes

October 22, 2003.

The goal is to introduce the CAST (cluster affinity search technique) clustering algorithm as described in A. Ben-Dor, R. Shamir and Z. Yakhini, *"Clustering Gene Expression Patterns"*, Journal of Computational Biology, Vol. 6, No. 3/4 (1999), 281-297.

Lecture notes from a previous lecture on CAST given by Larry Ruzzo can be found on http://www.cs.washington.edu/education/courses/cse527/01au/ (10/25/2001).

Graphs

Graphs consist of

- *vertices* and
- *edges* connecting the vertices.

Not all vertices have to be connected to other vertices and there might be no edges at all. Formally, a graph G is defined by a set of vertices V and a set of egdes E, short $G(V,E)$.

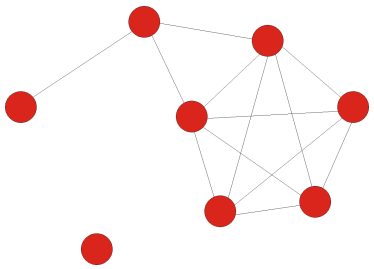

Figure 1: A graph.

We can use graphs to model simple, pairwise relations. For instance, a graph could represent the association between pairs of genes, where vertices represent genes and edges represent the association between them (similar expression level, same regulatory sequences, ...).

Cliques

On graphs we can define *cliques*. A *k-clique* is a set of k vertices, all directly connected to each other. We can use graphs and cliques to find clusterings. We create a graph from our data (representing associations between genes, persons, ...) and then set out to find all cliques in that graph. Since graph theory has been around for a long time, there exist algorithms which perform that task. However, the problem is that the number of possible cliques in a graph is growing exponentially. The general formula for the number of k-cliques in a graph with n vertices is

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\ &\geq \left(\frac{n}{k}\right)^k.\end{aligned}$$

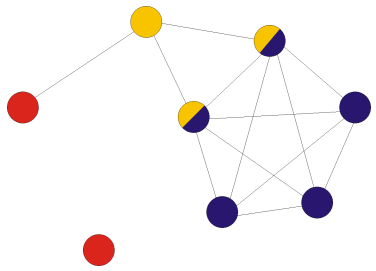

Figure 2: Two (overlapping) cliques in the graph from Figure 1. A 5-clique (blue) and a 3-clique (yellow) are highlighted. Note that two-colored vertices belong to both of the highlighted cliques and that that each k-clique contains $(k-1)$-cliques.

A simple and straightforward approach to clique-finding is a brute force algorithm that tries all possible cliques. There exist also faster clique-finding algorithms, but unfortunately there is no clique-finding algorithm that runs in polynomial time. Clique-finding is *NP*-complete and thus there will be no polynomial time algorithm except $P = NP$, which is quite unlikely.

Polynomial vs exponential growth and asymptotic analysis

Algorithms with an exponential runtime *may* perform better than a polynomial time algorithm, but only up to a distinct problem size. While algorithms with polynomial runtime are usually practicable, algorithms with exponential runtime rarely are.
If we want to analyze the runtime of an algorithm in the worst case, we employ *asymptotic analysis*. Asymptotic analysis helps us to determine how the runtime of an algorithm changes as a function of problem size. Commonly this is expressed using the *big-O notation*.

Definition $f(n) = O(g(n))$ iff there is a constant c such that $|f(n)| \leq c \cdot g(n)$ for all sufficiently large n. n is the problem size.

Examples for the big-O notation are given below.

- $2 \cdot n^2 = O(n^2)$
- $100 \cdot n^2 + 100 \cdot n + 100 = O(n^2)$
- $2^{2n} = O(2^{2n})$
- $n^2 = O(2^{2n})$, but $2^{2n} \neq O(n^2)$!

The bounds given by the big-O notation help us to quickly identify promising algorithm candidates.

Finding cliques in noisy data

Clustering based on cliques is highly sensitive to noise in the data. If we want to do clustering based on cliques anyways, we have to find a way to overcome the problems caused by noisy data. The following describes the problem.

- Given a graph H that is a collection of (large) cliques, we corrupt each edge/non-edge with a probability of $\alpha < 0.5$. That is to simulate noise[1].
- We call the resulting graph G.
- Find an approximate H' given G as input.

This problem is even harder than clique-finding, as we have to find cliques that are corrupted by noise. To solve this, we have to make a simplifying assumption described in the next section.

Finding *disjoint* cliques in noisy data

To simplify the search for cliques, we will only consider disjoint cliques, i.e. cliques that do not overlap. The problem is the following.

- Given a graph H that is a *disjunct* collection of (large) cliques, we corrupt each edge/non-edge with a probability of $\alpha < 0.5$.
- We call the resulting graph G.
- Find an approximate H' given G as input.
- We are successful if $|H' \oplus G| \leq |H \oplus G|$.[2]

If we didn't consider noise, the problem of finding disjoint cliques in a graph is easy. However, the noise gives us a hard time, but we are still able get pretty good results.
The paper of Ben-Dor et al. features several figures that show that large cliques are better reconstructible under these conditions than small ones, since there is more information available about them.

Main results

Ben-Dor et al. found that for all $\alpha \leq 0.5$, $\varepsilon > 0$ and $\delta > 0$ there is an algorithm A and a constant c (dependent on α, ε and δ) such that for all clique graphs H with disjoint cliques of minimum size at least $n\varepsilon$, A successfully recovers H' from the α-corrupted version G of H with probability $> 1-\delta$, running in time $O(n^2 \log(n)^c)$. A is deterministic.
Here it becomes clear why we can't reliably reconstruct small cliques. Ben-Dor et al. found that for "reasonable" choices of parameters like $\varepsilon = 0.1$ and $\alpha = 0.25$ the analysis yields $c < 600$. If we consider the runtime for $c = 600$,

$$O(n^2 \log(n)^{600}),$$

we find that this is by no means practical. The results are nevertheless interesting, as the analysis is probably pessimistic (thus safe) and the intuition for the algorithm is valuable[3].

[1] By randomly switching the edges in the graph, we assume an independent noise distribution over the data. But for instance, in microarray data there is usually a systematic error. Removing and adding edges between vertices with the same probability seems to be an oversimplification as well.

[2] $\oplus$ measures the distance between two graphs, i.e. the number of edge removals and insertions required to transform the one into the other.

[3] "The purpose of theory is insight, not theorems." (Google says that this is due to Richard Hamming as paraphrased by Steve Johnson).

Key ideas for the algorithm

Suppose we know k elements $v_1, v_2, \ldots, v_k$ of a clique. We call these elements a *core*. Given another vertex, we want to know if it is in the same clique. If so, the vertex is neighbor of $k(1-\alpha)$ core members, if not, it is neighbor of $k\alpha$ core members. $\alpha < 0.5 < 1-\alpha$, so the vertex is added to the clique if it is a neighbor of more than half of the vertices of the current core. The failure probability declines exponentially with k.
A rough outline of the algorithm looks like this:

- Find core.
- Test all other points if they are part of a clique.

But how do we find a core? As a brute force approach is too slow (it would have to check all subsets of size $O(\log(n))$), a more subtle solution is to try subsets of size $O(\log(n)/\log(\log(n)))$ to classify a sample size of $O(log(n))$.

A practical heuristic

1. $Copen :=$ the unassigned vertex of maximal average affinity
2. repeat until no change
   - $U :=$ unassigned vertex of maximal affinity to $Copen$
     if affinity $> threshold$, then add U to $Copen$
   - if none, then $V :=$ vertex in $Copen$ of minimal average affinity
     if affinity $< threshold$, then remove V from $Copen$
3. close $Copen$ and start at 1.

Unlike in a greedy algorithm, the removal of vertices in 2. allows the algorithm to correct "misclassifications". A final pass as described below takes place when all clusters have been closed.

1. repeat until no change or iteration limit reached
   - move each element to the cluster to which it has maximum affinity

The user is not able to (or has not to, depending on the point of view) define the expected number of clusters directly. However, by choosing the value of $threshold$, the user defines the number of clusters indirectly. For very high values of $threshold$ each data point will end up in its own cluster, while for very low values all data points will form a single cluster.
If we have biological background knowledge about the data, we might initialize the algorithm with cores based on that knowledge (similarly as we would initialize a k-means algorithm with cluster centers based on prior knowledge). Note that the removal of vertices from cliques in the 2. step or the final pass might destroy the initial cores!