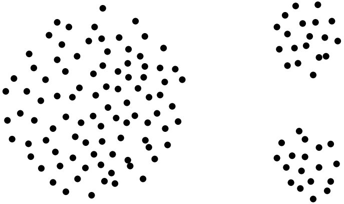
Clustering by Connectivity

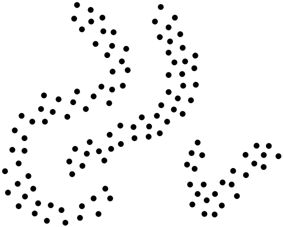
Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

The clusters found by centroid-based clustering (e.g., k-center and k-means) tend to have "ball shapes".



Sometimes clusters may have arbitrary shapes, e.g.:



Why does it make sense to discover such clusters?

Clustering and Unsupervised Learning

Recall that, in classification, we were given a labeled dataset, namely, every point's label was revealed. Finding a good classifier on such datasets is a form of **supervised learning**.

Opposite to this is **unsupervised learning**. Imagine, e.g., in classification, we are given an unlabeled dataset, where we do not know which points have label 0, and which points have label 1. How do we learn a classifier?

A good approach in this scenario is to do clustering. We can treat each cluster as a label, and thereby, get ourselves a "labeled" dataset, from which a classifier can be learned.

Hence, it makes sens to discover clusters of arbitrary shapes — a classification boundary may have an arbitrary shape!

Clustering by Connectivity is a form of clustering that is built on "distance graphs", and deviates significantly from centroid-based clustering. We will discuss two clustering methods under this category:

- Agglomerative clustering — also known as "hierarchical clustering".
- Density-based clustering

Agglomerative Clustering

Given a set P of n objects, the **agglomerative method** works as follows:

1. At the beginning, each object in P forms a cluster by itself.
2. Merge the two clusters that are most similar to each other.
3. Repeat the previous step until only one cluster is left.

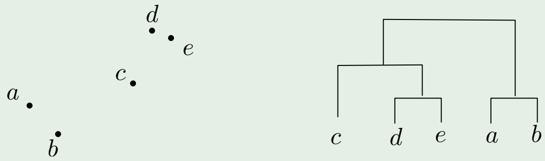
The above framework can be instantiated in many ways depending on how cluster similarity is defined. Specifically, let C_1 and C_2 be two clusters, each being a set of objects. To measure their similarity, we need a function $d(C_1, C_2)$ such that the smaller the function's value, the more similar the two clusters.

Some common definitions for cluster similarity are:

$$\begin{aligned}
d_{min}(C_1, C_2) &= \min_{o_1 \in C_1, o_2 \in C_2} dist(o_1, o_2) \\
d_{max}(C_1, C_2) &= \max_{o_1 \in C_1, o_2 \in C_2} dist(o_1, o_2) \\
d_{mean}(C_1, C_2) &= \frac{1}{|C_1||C_2|} \sum_{o_1 \in C_1, o_2 \in C_2} dist(o_1, o_2)
\end{aligned}$$

Among the three, d_{min} is the most popular—when this function is chosen, the agglomerative framework on the previous slide is known as the **single linkage algorithm**. We will focus on d_{min} in the rest of the lecture.

Example



Execution of the agglomerative method using the d_{min} metric:

1. Initially, 5 clusters: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$.
2. Merging $\{d\}, \{e\} \Rightarrow \{a\}, \{b\}, \{c\}, \{d, e\}$.
3. Merging $\{a\}, \{b\} \Rightarrow \{a, b\}, \{c\}, \{d, e\}$.
4. Merging $\{c\}, \{d, e\} \Rightarrow \{a, b\}, \{c, d, e\}$.
5. Merging $\{c\}, \{d, e\} \Rightarrow \{a, b, c, d, e\}$.

The merging history of the algorithm can be represented as a tree (see above), which is called a **dendrogram**.

Think:

- How many merges are there in total if we have n objects?
- Given a dendrogram, how would you obtain k clusters quickly?

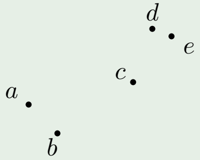
Next, we will explain that a dendrogram can be regarded as a minimum spanning tree. This naturally leads to an algorithm that computes a dendrogram in $O(n^2 \log n)$ time.

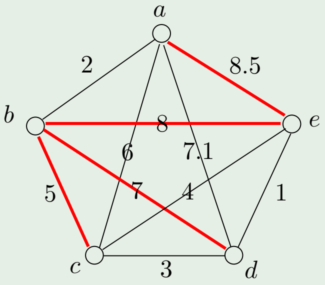
As before, let P be the set of n objects to be clustered. Define a **distance graph** $G(V, E)$ as follows:

- Every vertex of V corresponds to a distinct object in P.
- G is a complete graph, namely, there is an edge between each pair of vertices.
- The edge between vertex o_1 and o_2 carries a **weight** equal to $dist(o_1, o_2)$.

Let T be a set of $n - 1$ edges of G. If T induces no cycles, we say that T is a **spanning tree**. Define $cost(T)$ to be the sum of the weights of all the edges in T.

Example

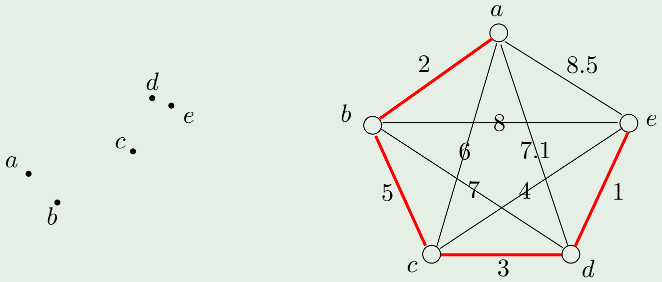




The figure on the right shows the distance graph.
The red edges indicate a spanning tree with cost 28.5.

The agglomerative framework essentially produces a spanning tree.

Example



The figure on the right shows the edges that the agglomerative algorithm uses to produce the dendrogram on Slide 9. Recall that the algorithm picks these edges in ascending order of weight.

Let T^* be a spanning tree of the distance graph G. If for any other spanning tree T, it always holds that $cost(T^*) \leq cost(T)$, we say that T^* is a **minimum spanning tree** (MST) of G.

Lemma

The agglomerative framework returns a minimum spanning tree of G.

Proof: The algorithm works in the same way as the Kruskal's algorithm, which is a well-known algorithm for finding an MST, and runs as follows. At the beginning, initiate an empty set T. At each step, among all the edges e satisfying

- e is not in T yet;
- the addition of e to T does not create a cycle;

add to T the one with the smallest weight. Repeat the step until T has $n - 1$ edges.

Next we will prove that the algorithm indeed finds an MST.

Proof (cont.): Label the edges of T as 1, 2, ..., n in the order they are discovered by the algorithm (i.e., the edge with label i is the i-th one discovered).

Let T^* be an arbitrary MST of G. Let t be the smallest integer such that the edge with label t does not belong to T^*. If t does not exist, then $T = T^*$, and we are done. Otherwise, denote that edge as e. Let S be the set of edges with labels 1, 2, ..., $t - 1$.

Now, add e to T^*, which definitely gives a cycle. In this cycle, at least one edge — say e' — does not belong to S (otherwise, the entire cycle is in S, which is impossible because T has no cycles). Observe that the weight of e' cannot be smaller than that of e: otherwise, Kruskal's algorithm would have used e', instead of e (notice that, the edges with labels 1, 2, ..., $t - 1$ cannot form a cycle with e' because, by definition, all those edges are in T^*).

We now obtain another MST T'^* from T^* by deleting e' and adding e. Repeat the above argument using T'^* — note that when we do so, the value of t increases by 1.

With this, we complete the proof. ☐

Although not required in this course, it is worth mentioning that sub-quadratic time algorithms exist for computing a dendrogram in d-dimensional space where d is a constant (for point objects and Euclidean distance). Specifically, the computation time is

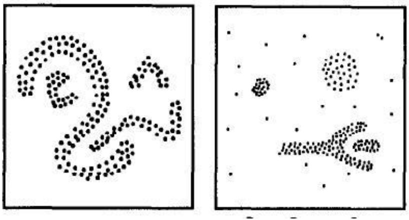
$$O\left(\frac{n^2}{n^{\frac{2}{\lceil d/2 \rceil + 1} - \epsilon}}\right)$$

time, where ϵ can be an arbitrarily small constant. Interested students may refer to:

Pankaj K. Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf: Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs. Discrete & Computational Geometry 6: 407-422 (1991).

Density-Based Clustering

In some applications, clusters can have arbitrary shapes and may need to be separated from **noise**:



(figures from a KDD96 paper titled "A density-based algorithm for discovering clusters in large spatial databases with noise")

We will learn a method called DBSCAN to find such clusters. It serves as a representative of noise-resistant density-based clustering, which works by enforcing two principles:

- The area around a noise point is “sparse”.
- If two points are placed in the same cluster, it should be possible to “walk” from one point to the other by staying only in the “dense” areas.

Parameters and Core Points

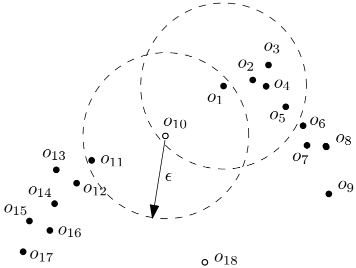
Parameters:

- ϵ: a distance threshold.
- *MinPts*: a constant integer.

$B(p, \epsilon)$: the ball centered at a point with radius ϵ, called the **vicinity area** of p.

P: the set of points to cluster

Core point: a point $p \in P$ such that $B(p, \epsilon)$ covers at least *MinPts* points of P.



$MinPts = 4$
Core points in black

Forming Clusters

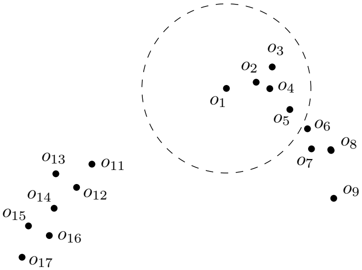
Conceptually, clusters are defined in two steps:

1. Cluster core points.
2. Assign non-core points.

We will explain each step in turn.

Step 1: Cluster core points

This step focuses only on core points.



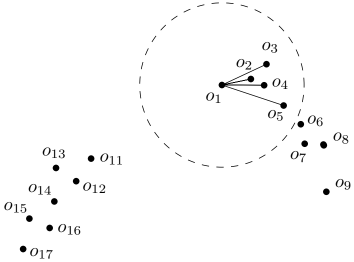
$MinPts = 4$

Core points in black

Step 1: Cluster core points

Connect a core point p to all the points in $B(p, \epsilon)$.

For example, o_1 is connected to 4 points in its vicinity area:

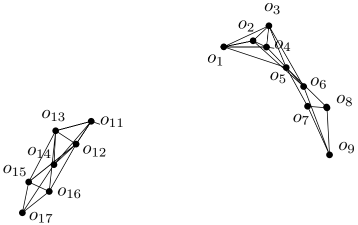


$MinPts = 4$

Core points in black

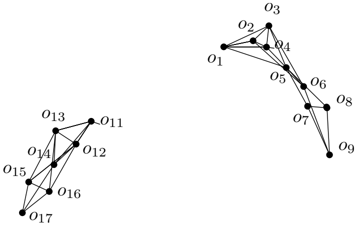
Step 1: Cluster core points

This is the situation after adding all the edges:



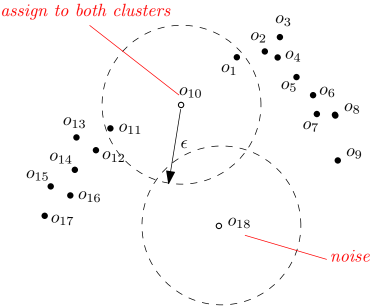
Step 1: Cluster core points

Take each connected component of the resulting a graph as a cluster.



Step 2: Assign non-core points

Every non-core point p is added to the cluster of every core point in $B(p, \epsilon)$. For example, o_{10} is added to two clusters: the cluster of o_1 and the cluster of o_{11}.

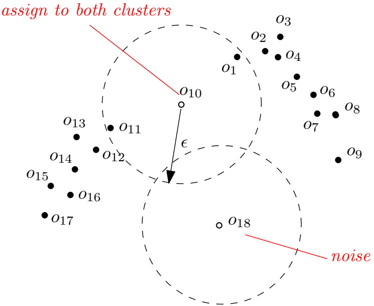


$MinPts = 4$

Each non-core point can be assigned to at most $MinPts - 1 = O(1)$ clusters.

Step 2: Assign non-core points

Final clusters: $\{o_1, o_2, ..., o_9, o_{10}\}$, $\{o_{10}, o_{11}, o_{12}, ..., o_{17}\}$.



$MinPts = 4$

The clustering result is unique.

It is straightforward to obtain the DBSCAN clusters in $O(n^2)$ time, where n is the number of points (think: how), treating d as a constant.

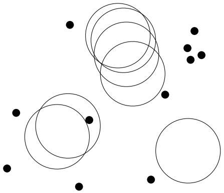
In several textbooks, it is claimed that the time can be improved to $O(n \operatorname{polylog} n)$. Unfortunately, this is unlikely to be possible when the dimensionality d is at least 3, as we explain next.

The following material will not be tested.

Geometry Preliminary 1: Unit-Spherical Emptiness Checking (USEC)

Let S_{pt} be a set of points, and S_{ball} be a set of balls with the same radius, all in data space $\mathbb{R}^d$, where the dimensionality d is a constant.

The objective of USEC is to determine whether there is a point of S_{pt} that is covered by some ball in S_{ball}.



Known results:

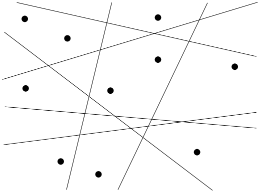
$d = 2$: Solvable in $O(n \log n)$ time.
$d = 3$: Solvable $O((n \log n)^{4/3})$ time.
Big open problem: $o(n^{4/3})$ for $d = 3$?

Common conjecture: no.

Geometry Preliminary 2: Hopcroft

Let S_{pt} be a set of points, and S_{line} be a set of lines, all in data space $\mathbb{R}^2$ (note that the dimensionality is always 2).

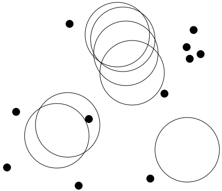
The goal of the Hopcroft's problem is to determine whether there is a point in S_{pt} that lies on some line of S_{line}.



Known results: Solvable in time slightly higher than $O(n^{4/3})$.
Big open problem: $o(n^{4/3})$ possible?
Common conjecture: No.
$\Omega(n^{4/3})$ lower bound known on a broad class of algorithms.

Geometry Preliminary 3: Hopcroft Hardness

We will call a problem X Hopcroft hard if an algorithm solving X in $o(n^{4/3})$ time implies an algorithm solving the Hopcroft's problem in $o(n^{4/3})$ time.



Fact: USEC is Hopcroft hard for $d \geq 5$.

We will prove:

Theorem

The following statements are true about the DBSCAN problem:

- It is Hopcroft hard in any dimensionality $d \geq 5$.
  - Namely, the problem requires $\Omega(n^{4/3})$ time to solve, unless the Hopcroft problem can be settled in $o(n^{4/3})$ time.
- When $d = 3$ (and hence, $d = 4$), the problem requires $\Omega(n^{4/3})$ time to solve, unless the USEC problem can be settled in $o(n^{4/3})$ time.

More specifically, we will prove:

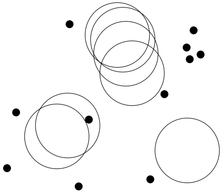
Lemma

For any constant dimensionality d, if we can solve the DBSCAN problem in $T(n)$ time, then we can solve the USEC problem in $T(n) + O(n)$ time.

The theorem is a corollary of this lemma (think: why).

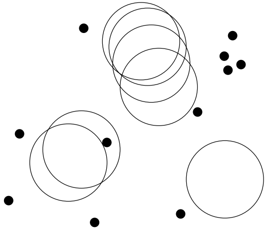
USEC

Let S_{pt} be a set of points, and S_{ball} be a set of balls with the same radius, all in data space $\mathbb{R}^d$, where the dimensionality d is a constant. The objective of USEC is to determine whether there is a point of S_{pt} that is covered by some ball in S_{ball}.



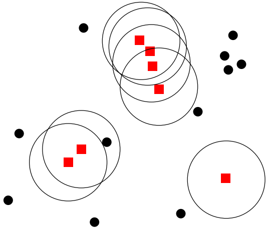
Next, we give a reduction from USEC to DBSCAN. Specifically, given a DBSCAN algorithm A, we show how to solve USEC by using A as a black box.

Using DBSCAN to Solve USEC



Using DBSCAN to Solve USEC

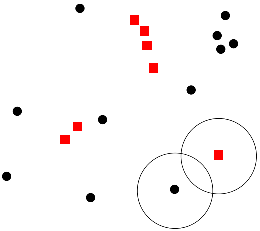
Obtain P as the union of S_{pt} and the set of centers of the balls in S_{ball}.



Using DBSCAN to Solve USEC

Run the DBSCAN algorithm A to cluster P with

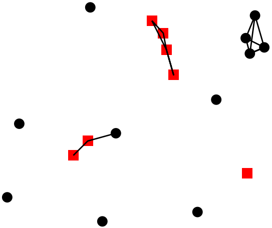
- Set ϵ to the radius of the balls.
- $MinPts = 1$.



Using DBSCAN to Solve USEC

Run the DBSCAN algorithm A to cluster P with

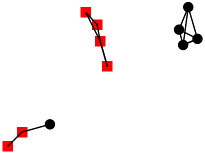
- Set ϵ to the radius of the balls.
- $MinPts = 1$.



Using DBSCAN to Solve USEC

Run the DBSCAN algorithm A to cluster P with

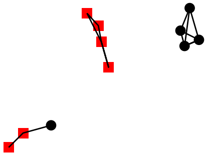
- $\epsilon =$ the radius of the balls.
- $MinPts = 1$.



Using DBSCAN to Solve USEC

Check if any red square and black circle are put in the same cluster.

- If so, say "yes" to USEC.
- Otherwise, say "no".

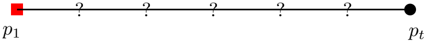


Running time $T(n) + O(n)$.

Using DBSCAN to Solve USEC

Correctness: An original circle covers a point if and only if we say yes.

Proof: The only-if direction is obvious (think: why?). We will focus on proving the if-direction.



A "yes" answer means that there is a sequence of points $p_1, p_2, ..., p_t \in P$ such that (i) p_1 is red and p_t is black, and (ii) $dist(p_i, p_{i+1}) \leq r$ for each $i \in [1, t-1]$. Let k be the smallest $i \in [2, t]$ such that p_i is black. Note that k definitely exists because p_t is black. It thus follows that the ball centered at p_{k-1} covers the point p_k in the original USEC problem. ☐

Recall the single-linkage algorithm we discussed in the previous lecture. There is an inherent connection between single-linkage and DBSCAN.

Think: Suppose that you have computed a dendrogram for single-linkage. How would you use the dendrogram to obtain a DBSCAN clustering with parameterized by $\epsilon > 0$ and $minPts = 1$?