DSC40B: Theoretical Foundations of Data Science II

Lecture 17: *Kruskal's algorithm for MST, and clustering*

Instructor: Yusu Wang

Previously

- Given a weighted undirected graph
  - Prim's algorithm to compute the minimum spanning tree (MST) in $\Theta((V+E)\lg V)$ time
    - It is a greedy algorithm

- Today:
  - Yet another greedy algorithm to compute MST, called Kruskal's algorithm
  - Relation to hierarchical clustering

Kruskal’s Algorithm for MST

Recall the general greedy idea:

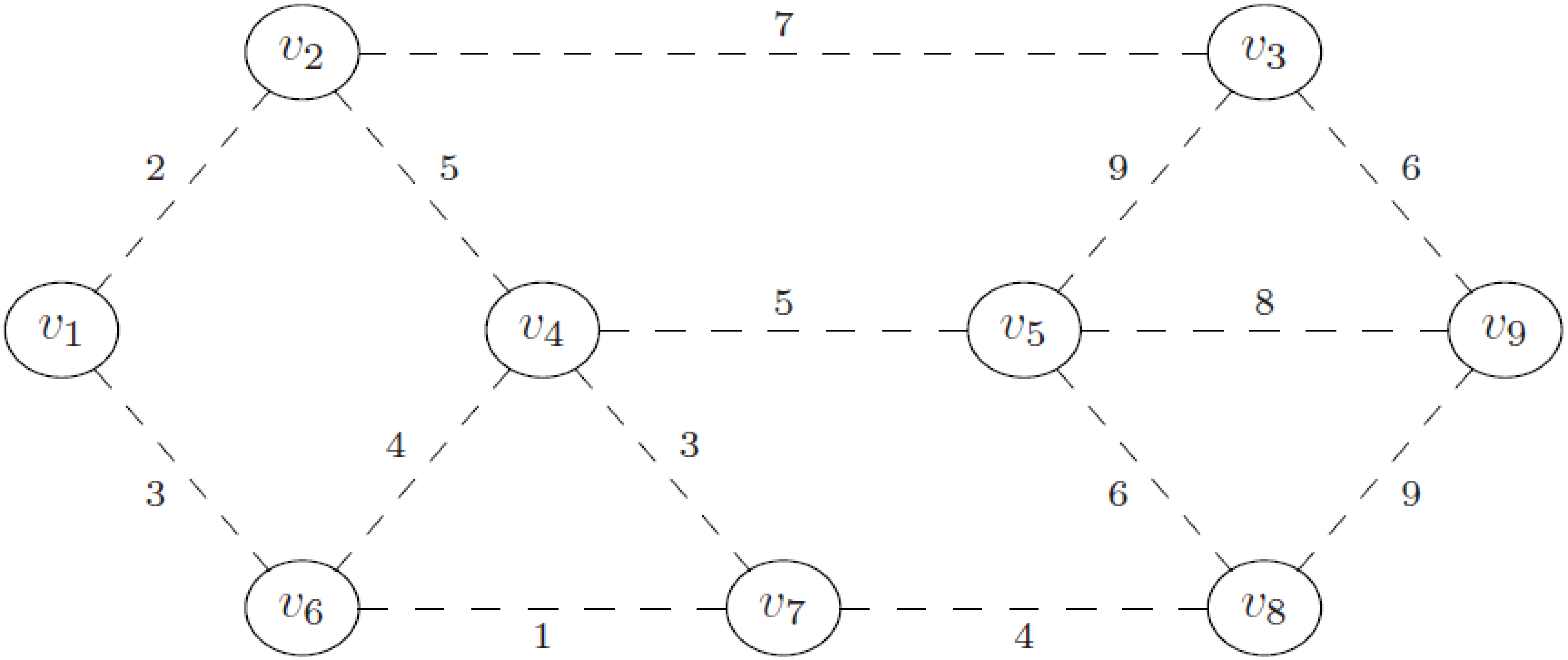
- Input:
  - a weighted undirected graph $G = (V, E)$, with $\omega: E \rightarrow R$
- Output:
  - the set of edges in a MST T of G

- A MST T is $V - 1$ number of edges that connect all nodes, with no cycle.
- Intuitively, we will choose "safe" edges greedily to incrementally build T
  - such that any time, the edges we choose will form a part of some MST

- Last time: Prim's algorithm
  - Starting from any node, it starts to grow a partial tree, by repeatedly choosing the minimum-weight edge to reach an unvisited node, till it connects all graph nodes

- Today: Kruskal's algorithm
  - We will choose the ``safe" edges in a different order, and still grow the tree edge by edge
    - However, during the intermediate stages, what we have may not be a partial tree, could be disconnected.

Example



- What is a "safe" edge to add first?
- What will be the next a "safe" edge to add?

Strategy

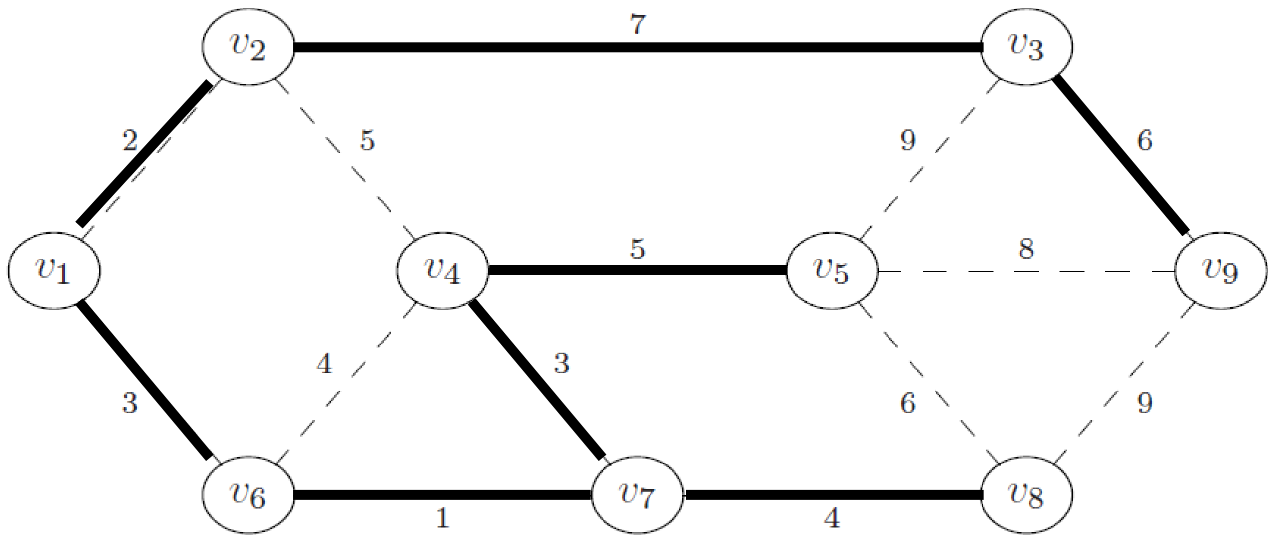
- We will add edges gradually in a greedy manner using smallest weights, while maintaining what we have so far does not have any cycle
  - add edges to tree in ascending order by weight
  - but if an edge creates a cycle, do not add it, and move on to next edge

- How do we check whether adding a new edge $e = (u, v)$ creates a cycle or not?
  - check whether nodes u, v are already connected.

Kruskal’s algorithm (Pseudocode)

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        # if u and v are not already connected
        if ...:
            mst.add_edge(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Example



Kruskal's algorithm (Pseudocode)

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        # if u and v are not already connected
        if ...:
            mst.add_edge(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Checking for connectivity

- Each iteration: need to check if u and v are already connected in current $T = (V, E_{mst})$
- We can do a BFS/DFS on each iteration
  - $\Theta(V + E_{mst}) = \Theta(V)$ each time
  - Expensive!
- Again, remember:
  - If you're computing something once, use a fast algorithm
  - If you're computing it repeatedly, consider a data structure!

Disjoint Set Forests

- Represent a collection of disjoint sets over a set of elements
  - $\{\{1,5,6\},\{2,3\},\{0\},\{4\}\}$
- Two operations:
  - .union(x, y): Union the sets containing x and y
  - .in_same_set(x, y): return True/False if x and y are in the same set
    - Typically, this is implemented by an operation .find(x), which returns the representative of the set containing x.
- In the literature, this is also commonly referred to as the union-find data structure to maintain dynamic disjoint sets

Example

- \>>> *# create a DSF with {{0}, {1}, {2}, {3}, {4}, {5}}*
- \>>> dsf = DisjointSetForest([0, 1, 2, 3, 4, 5])
- \>>> dsf.union(0, 3)
- \>>> dsf.union(1, 4)
- \>>> dsf.union(3, 1)
- \>>> dsf.union(2, 5)
- \>>> *# dsf now represents {{0, 1, 3, 4}, {2, 5}}*
- \>>> dsf.in_same_set(0, 3)
- True
- \>>> dsf.in_same_set(0, 2)
- False

Disjoint Set Forests

- Each operation takes $\Theta(\alpha(n))$ time, where n is the number of objects in the collection

- $\alpha(n)$: inverse Ackermann function
  - It grows very very very slowly.
  - $\alpha(n) = o(\lg n)$
  - While asymptotically, it grows faster than a constant function, effectively in practice, for any number n we can imagine, $\alpha(n)$ is essentially a constant.

Disjoint Set Forest

- Can be used to keep track of connected components (CCs) of a dynamic graph

- Nodes of CCs are disjoint sets
  - Add an edge (u, v): .union(u, v)
  - Check if u and v are connected: .in_same_set(u, v)

- To check if u and v are already connected:
  - BFS/DFS: $\Theta(V)$ each time
  - Disjoin set forest: $\Theta(\alpha(V))$ each time (essentially like constant in practice, but not asymptotically!)

Kruskal’s Algorithm

```python
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Time complexity of Kruskal's algorithm

- Assume graph is connected. Then $E = \Omega(V)$

- Kruskal's algorithm takes $\Theta(E \lg E) = \Theta(E \lg V) time$
  - Dominated by sorting the edges

- Note: if graph is disconnected, then Kruskal's algorithm produces *a minimum spanning forest.*

Kruskal's vs. Prim's

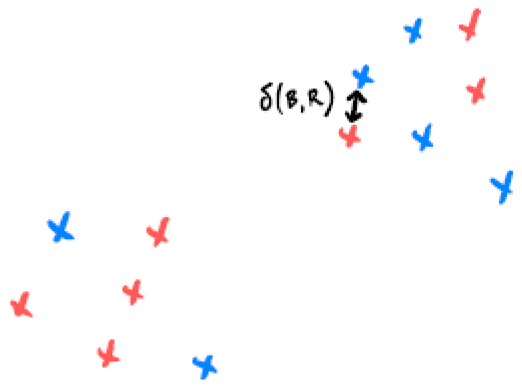
- Time complexity:
  - Prim's:
    - Binary heap: $\Theta(V \lg V + E \lg V)$ (= $\Theta(E \lg V)$ if graph is connected)
    - Fibonacci heap: $\Theta(V \lg V + E)$
  - Kruskal's:
    - $\Theta(V + E \lg V)$(= $\Theta(E \lg V)$ if graph is connected)
  - If graph is dense (e.g, $E = \Theta(V^2)$), then Prim's with Fibonacci heap "wins" in asymptotic time complexity
  - In practice, Fibonacci heaps are hard to implement with high overhead.
    - Kruskal's may be faster for smaller dense graphs

MSTs and (hierarchical) clustering

Clustering problem

- Identify the groups in data
- We frame clustering as a loss minimization problem
  - Input: n data points in R^d
  - Goal: assign each data point a color (red or blue, which is class labels) so that the distance between the closest pair of red and blue points is maximized

input points

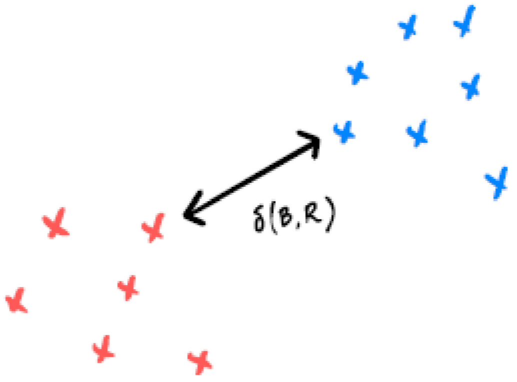


bad clustering

Clustering problem

- Identify the groups in data
- We frame clustering as a loss minimization problem
  - Input: n data points in R^d
  - Goal: assign each data point a color (red or blue, which is class labels) so that the distance between the closest pair of red and blue points is maximized

input points



good clustering

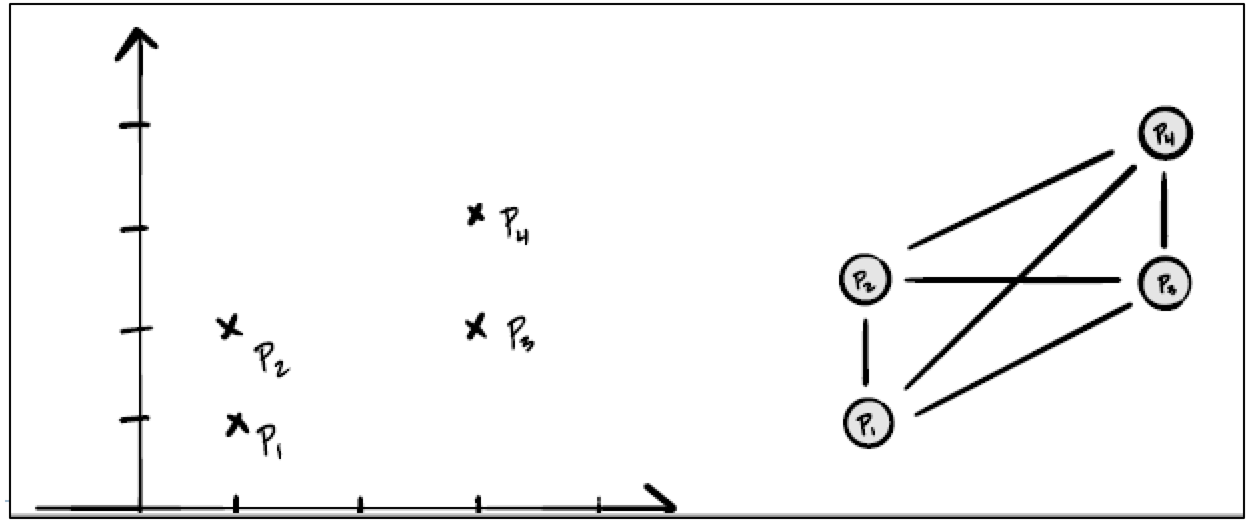
Recall the brute-force solution

- Try all possible assignments
  - there are 2^n possible assignments! (n is the number of input points)
  - highly not efficient!

- Instead, we will convert it to a graph problem

Distance graph

- Given n data points $V = \{ p_1, p_2, \dots, p_n\}$
  - create a complete undirected graph $G = (V, E)$ such that for any $p_i \neq p_j$, there is an edge $(p_i, p_j) \in E$
  - the weight of an edge (p_i, p_j) is $\omega(p_i, p_j) = dist(p_i, p_j)$
- We call this resulting weighted graph the distance graph spanned by V



Clustering

- Given n data points $V = \{ p_1, p_2, \ldots, p_n\}$
- Create distance graph G
- Run either Prim's or Kruskal's Algorithm to compute MST of G, T:

- Delete largest edge in MST, and we obtain two components => clusters!
- In general, if we remove largest $k - 1$ edges in MST
  - then we obtain k clusters (components)

Single linkage clustering

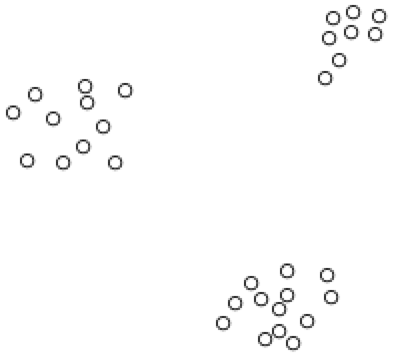
- Alternatively, we can perform Kruskal's algorithm, adding edges in ascending order by weights without forming cycles, and stop till we have a target k number of components (clusters)
  - That is, we terminate early in the Kruskal's algorithm
- Time complexity
  - $\Theta(E \lg V) = \Theta(V^2 \lg V)$ as $E = \Theta(V^2)$ in this case

- This is called the single-linkage clustering algorithm
  - One of the most well-known clustering algorithm.
  - It is popular, although it suffers from the so-called chaining-effect in practice.
  - Variants of it, e.g., average-linkage clustering algorithm and complete-linkage clustering algorithm, are popular as a simple clustering algorithm.

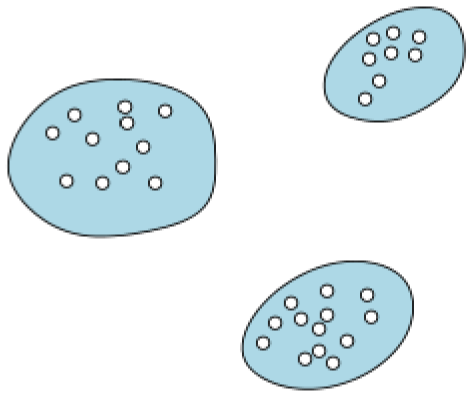
Example of chaining effect

Hierarchical clustering, and single linkage clustering algorithm (Optional)

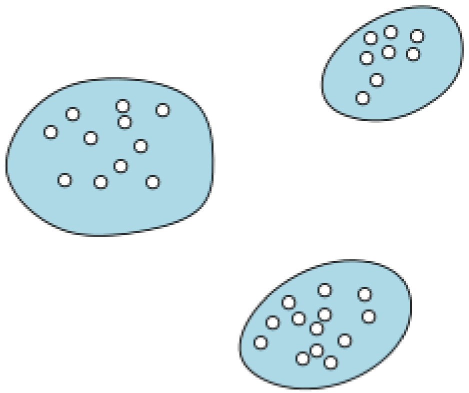
Clustering

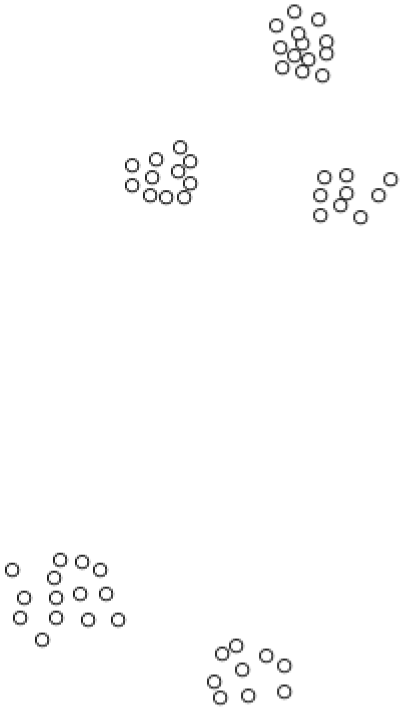


Clustering



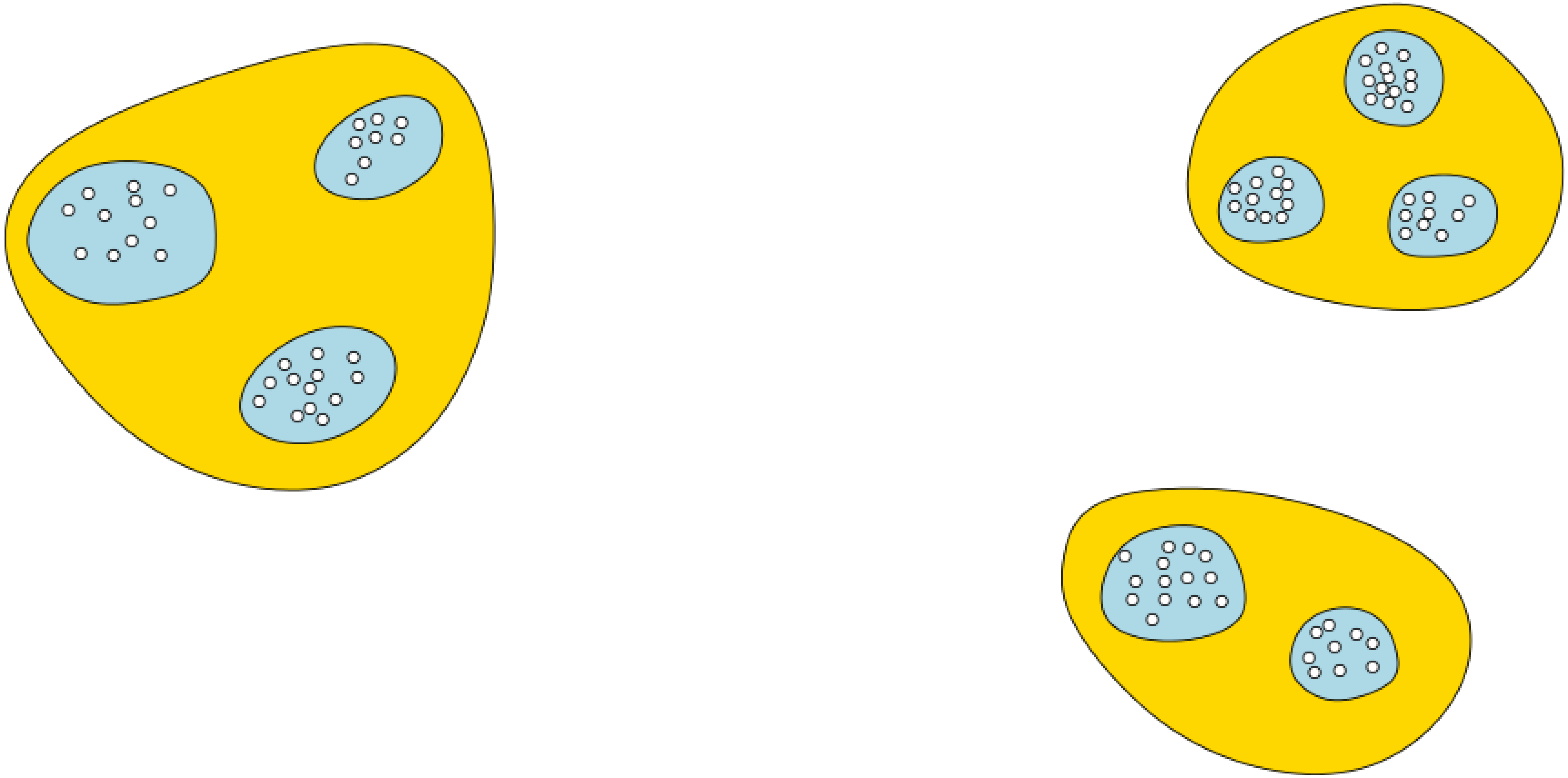
Hierarchical Clustering



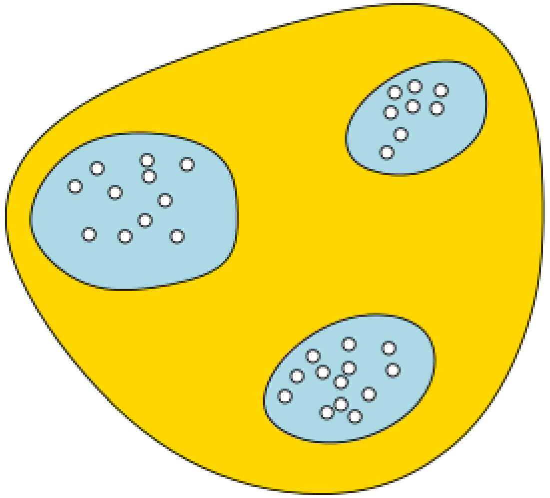


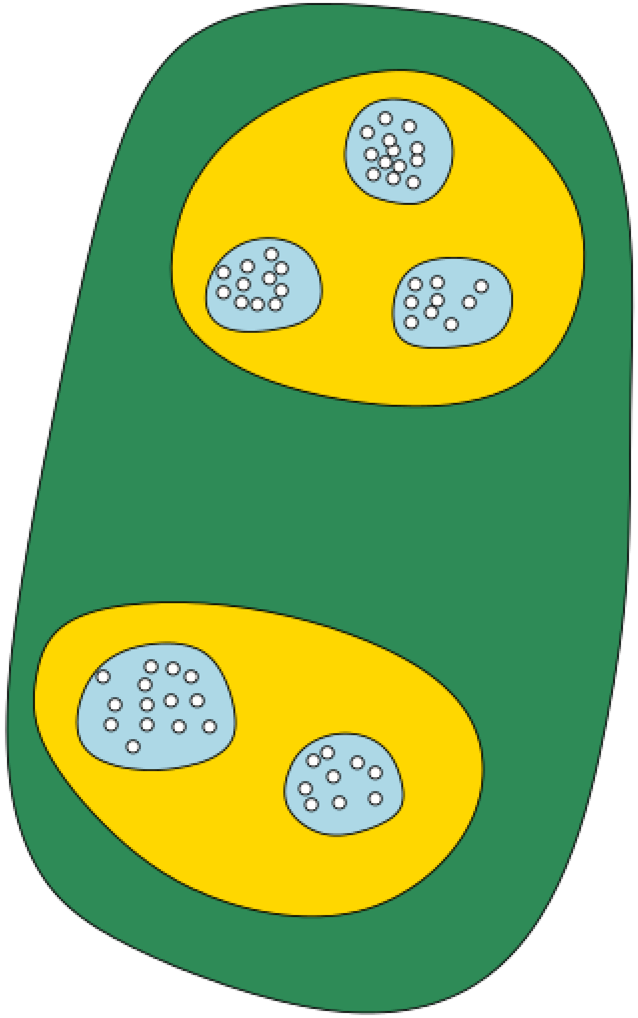
Hierarchical Clustering

Hierarchical Clustering

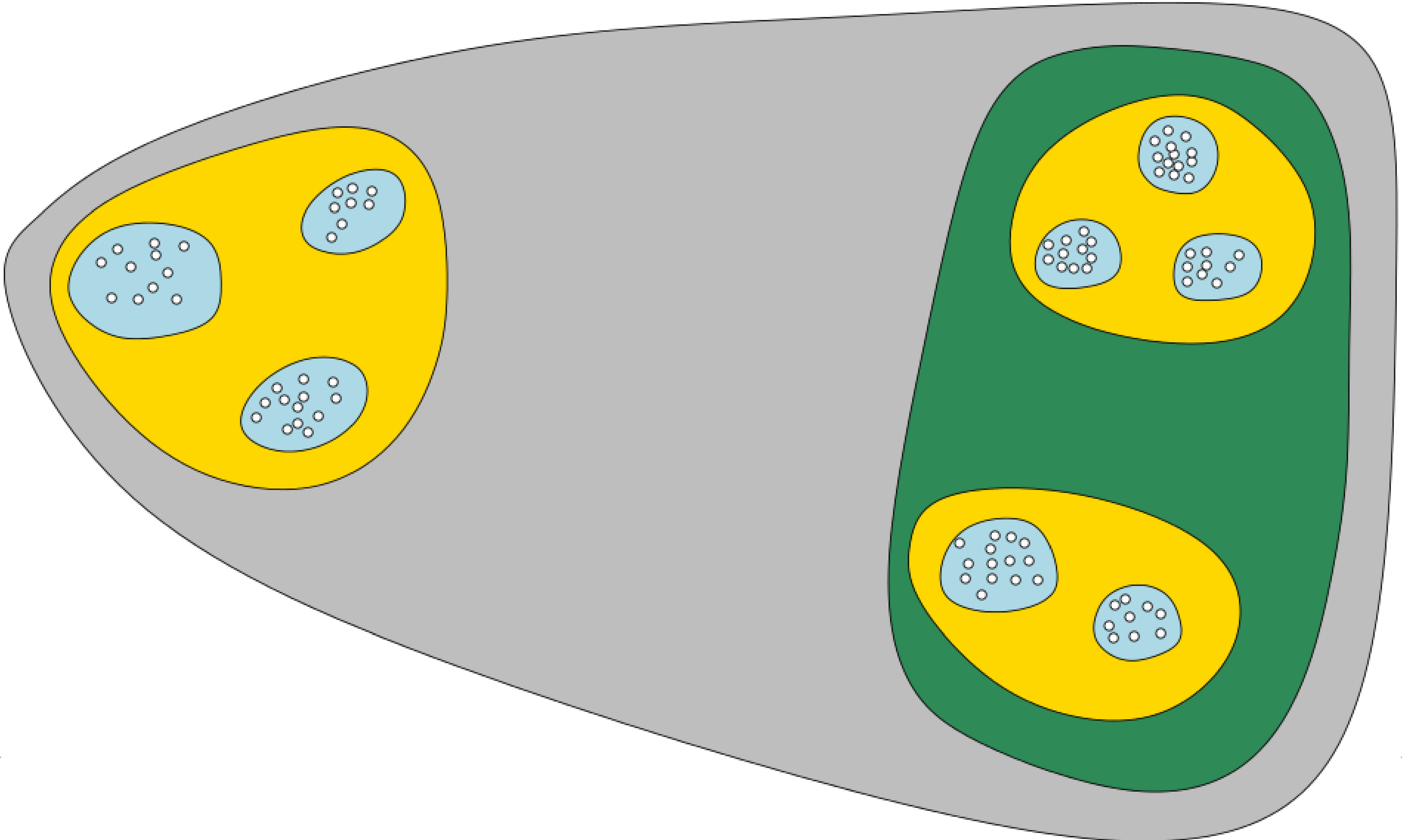


Hierarchical Clustering

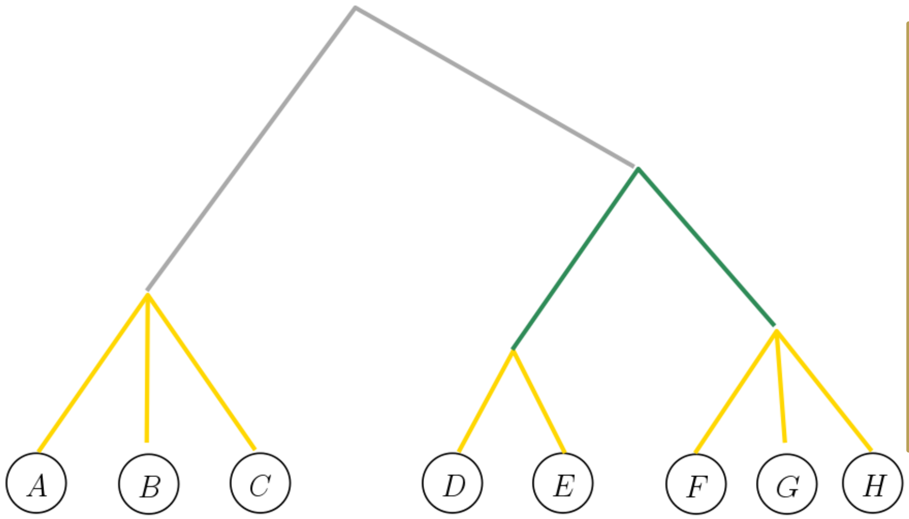




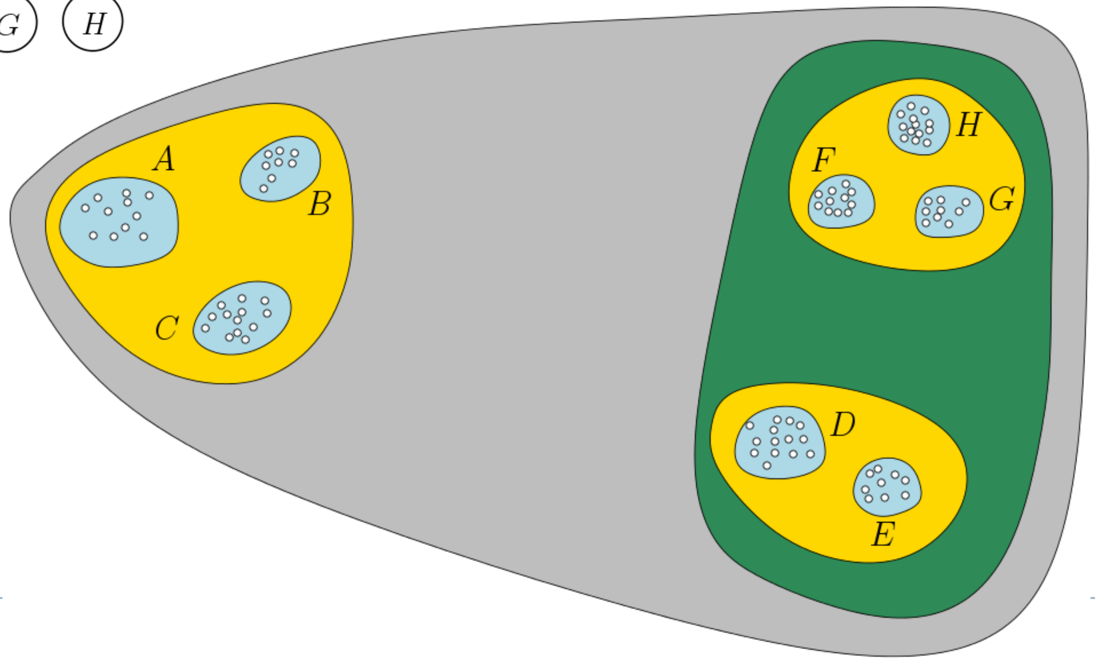
Hierarchical Clustering



Hierarchical Clustering Tree (HCT)



- Each internal tree node indicates a cluster, containing all leaves in subtree
- Ancestor/descendent indicates containment relation
- Height at each tree node corresponds to certain *cost* of the corresponding cluster (e.g, tightness of cluster)



Single Linkage Clustering

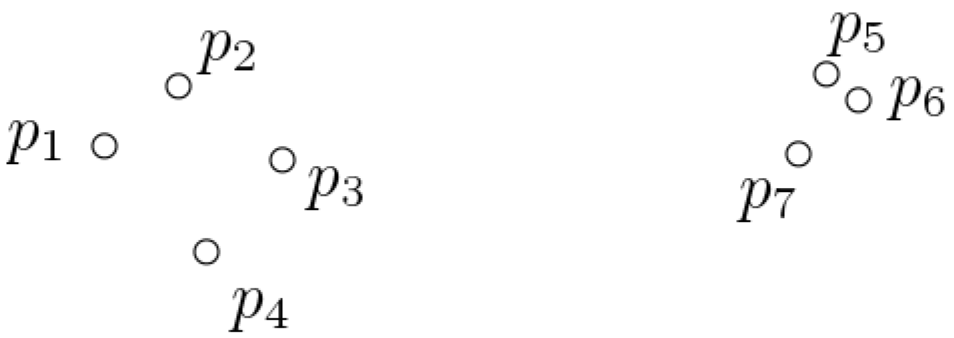
- One of the family of agglomerative clustering methods

- Input: A discrete n-point metric (P, d_P)
- Output: A hierarchical clustering tree T, with points in P being leaves

- Starting with each data point as a single cluster
- Keep merging clusters based on nearest distance between points from their members

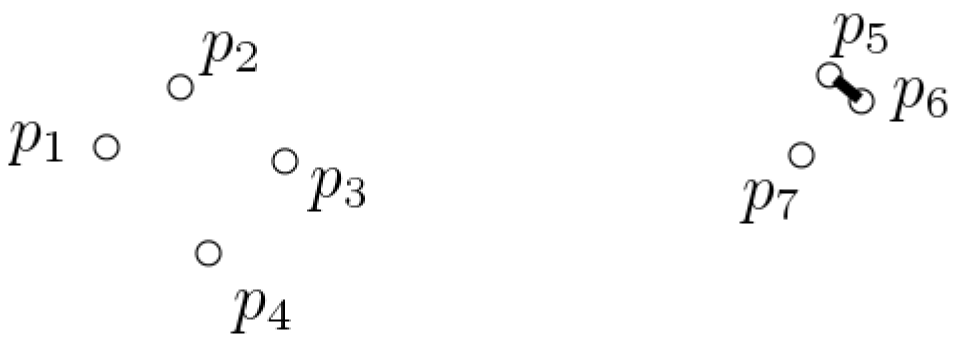
Single Linkage Clustering

- Starting with each data point as a single cluster
- Keep merging clusters based on nearest distance between points from their members



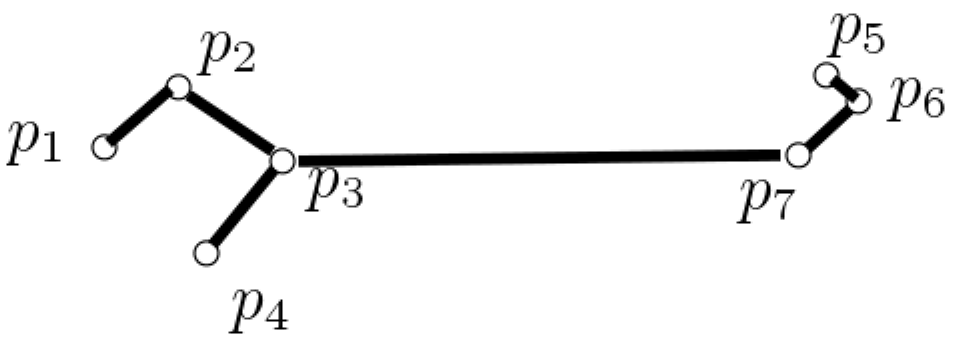
Single Linkage Clustering

- Starting with each data point as a single cluster
- Keep merging clusters based on nearest distance between points from their members



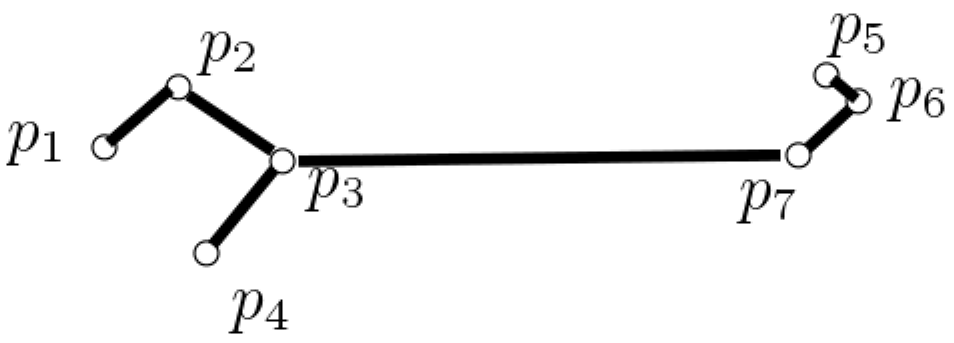
Single Linkage Clustering

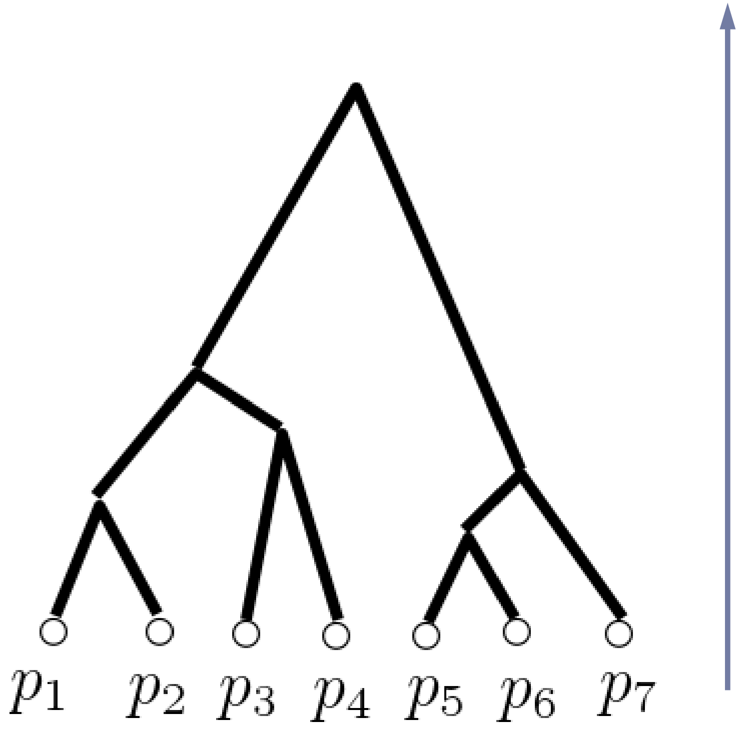
- Starting with each data point as a single cluster
- Keep merging clusters based on nearest distance between points from their members



Single Linkage Clustering

- Starting with each data point as a single cluster
- Keep merging clusters based on nearest distance between points from their members

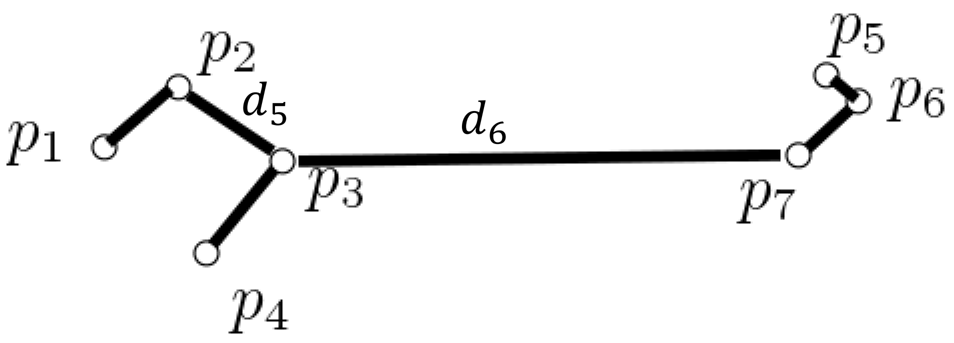


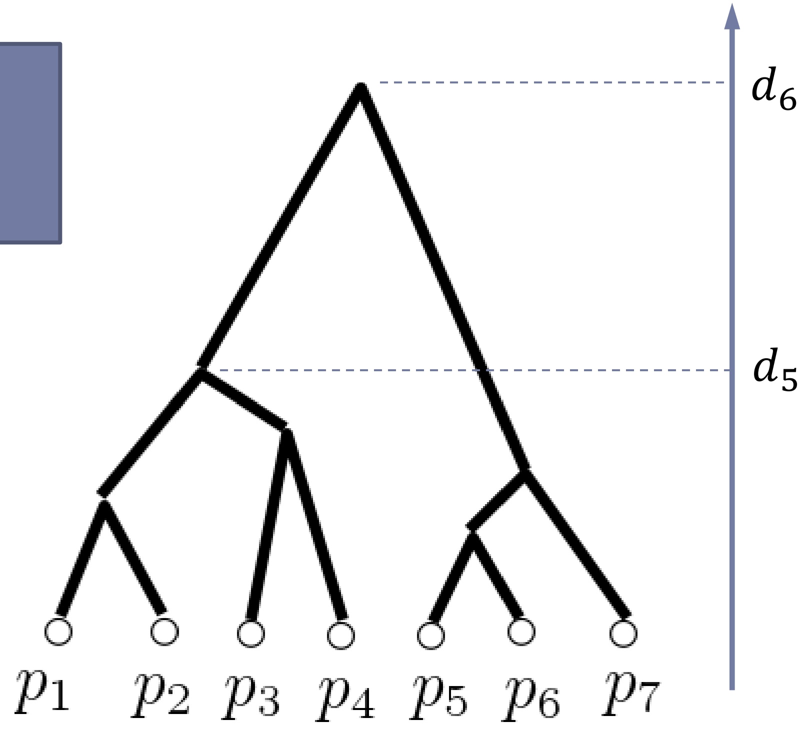


One Example: Single Linkage Clustering

- Starting with each data point as a single cluster
- Keep merging clusters based on nearest distance between points from their members

This procedure is exactly Kruskal's algorithm!





Summary

- MST for weighted undirected graphs
  - Prim's algorithm
  - Kruskal's algorithm
  - Both $\Theta\big((V+E)\lg V\big)$ (which is $\Theta(E\lg V)$ for connected graphs)

  - Kruskal's algorithm can be used to produce single linkage clustering

FIN