# Comparison of Clustring Algorithms

When I studying Python, this is a homework for me to analyze the performance of these two clustering methods on various subsets of our county-level cancer risk data set in the USA.

In particular, we will compare these two clustering methods in three areas:

**Efficiency** - Which method computes clusterings more efficiently?

**Automation** - Which method requires less human supervision to generate reasonable clusterings?

**Quality** - Which method generates clusterings with less error?

**The methods of clustering and related information are showed below**

# Algorithmic Thinking
# Luay Nakhleh

## Clustering and the Closest Pair Problem
### The Divide-and-Conquer Algorithmic Technique

## 1 Clustering

**Definition 1** *A clustering of a set $P$ of points into $k$ clusters is a partition of $P$ into sets $C_1, \ldots, C_k$, such that*

- $\forall 1 \leq i \leq k,\ C_i \subseteq P$,
- $\forall 1 \leq i \leq k,\ C_i \neq \emptyset$,
- $\forall 1 \leq i, j \leq k, i \neq j,\ C_i \cap C_j = \emptyset$, *and*
- $\cup_{i=1}^{k} C_i = P$.

Fig. 1 shows a clustering of 10 points into two clusters.

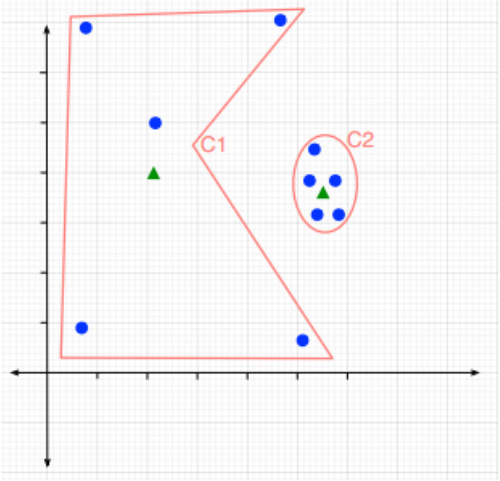



Figure 1: Two clusters C1 and C2 on the set of points (blue solid circles) and their centers are shown (green triangle).

We define the center of a cluster $C_u$ as

$$center(C_u) = \frac{1}{|C_u|} \sum_{p_i \in C_u} (x_i, y_i).$$

For example, if $C_u = \{p_1, p_7, p_9\}$, with $p_1 = (1, 2)$, $p_2 = (4, 6)$, and $p_3 = (4, 4)$, then

$$center(C_u) = \frac{1}{3}((1, 2) + (4, 6) + (4, 4)) = \frac{1}{3}(9, 12) = (3, 4).$$

Fig. 1 shows the centers of the two clusters C1 and C2.

While many clusterings of the points in $P$ exist, a desired property is that the partition results in clusters with higher similarity of points within a cluster than of points between clusters. Algorithms **HierarchicalClustering** and **KMeansClustering** below are two heuristics for generating clustering with this desired property. In both algorithms, we will use $k$ to denote the number of clusters.

**A word on implementation.** While $P$ is defined as a set in both clustering algorithms, it is more convenient to implement it using a list, since each point can be accessed directly in the list. Further, both algorithms return a set $C$ of clusters; here, $C$ is a set of elements, where each element is a set of points, and $C$ satisfies the properties in Definition 1.

**Algorithm 1: HierarchicalClustering.**

**Input**: A set $P$ of points whose $i$th point, $p_i$, is a pair $(x_i, y_i)$; $k$, the desired number of clusters.
**Output**: A set $C$ of $k$ clusters that provides a clustering of the points in $P$.

$n \leftarrow |P|$;
Initialize $n$ clusters $C = \{C_1, \ldots, C_n\}$ such that $C_i = \{p_i\}$;
**while** $|C| > k$ **do**
  $(C_i, C_j) \leftarrow \text{argmin}_{C_i, C_j \in C, i \neq j} d_{C_i, C_j}$;
  $C \leftarrow C \cup \{C_i \cup C_j\}$;
  $C \leftarrow C \setminus \{C_i, C_j\}$;
**return** $C$;

**Algorithm 2: KMeansClustering.**

**Input**: A set $P$ of points whose $i$th point, $p_i$, is a pair $(x_i, y_i)$; $k$, the desired number of clusters; $q$, a number of iterations.
**Output**: A set $C$ of $k$ clusters that provides a clustering of the points in $P$.

$n \leftarrow |P|$;
Initialize $k$ centers $\mu_1, \ldots, \mu_k$ to initial values (each $\mu$ is a point in the 2D space);
**for** $i \leftarrow 1$ **to** $q$ **do**
  Initialize $k$ empty sets $C_1, \ldots, C_k$;
  **for** $j = 0$ **to** $n - 1$ **do**
    $\ell \leftarrow \text{argmin}_{1 \leq f \leq k} d_{p_j, \mu_f}$;
    $C_\ell \leftarrow C_\ell \cup \{p_j\}$;
  **for** $f = 1$ **to** $k$ **do**
    $\mu_f = center(C_f)$
**return** $\{C_1, C_2, \ldots, C_k\}$;

## 2 2D Points, the Euclidian Distance, and Cluster Error

Both algorithms, **HierarchicalClustering** and **KMeansClustering**, make use of a distance measure, $d$. In the case of **HierarchicalClustering**, $d_{C_i, C_j}$ is the distance between the two clusters $C_i$ and $C_j$. In the case of **KMeansClustering**, $d_{p_j, \mu_f}$ is the distance between the point $p_j$ and center $\mu_f$ of cluster $C_f$. But, how is this distance measure $d$ defined?

In our case, we will only deal with points in the 2D space, such that each point $p_i$ is given by two features: its horizontal (or, $x$) and vertical (or, $y$) coordinates, so that $p_i = (x_i, y_i)$. One natural way to quantify the distance between two points $p_i$ and $p_j$ in this case is the standard Euclidian distance:

$$d_{p_i, p_j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

While this distance measure applies directly to compute $d_{p_j,\mu_f}$ in **KMeansClustering**, how does it apply to computing $d_{C_i,C_j}$ in **HierarchicalClustering**? By computing the distance between their centers:

$$d_{C_i,C_j} \equiv d_{center(C_i),center(C_j)}.$$

In our discussion above, we assumed a known number of clusters, $k$, and sought a clustering with that number of clusters. However, in general, the number of clusters is unknown. One way to determine the number of clusters is to vary the value of $k$, such that $k = 1, 23, \ldots$, and for for each value of $k$ to "inspect" the quality of the clusters obtained. One measure of such quality is the error of a cluster, which reflects how tightly packed around the center the cluster's points are, and is defined for cluster $C_i$ as

$$error(C_i) = \sum_{p \in C_i} (d_{p,center(C_i)})^2.$$

To illustrate, consider the the two clusters in Fig. 1. Cluster C1 has a larger *error* value than C2 and, indeed, compared to C2, it is hard to argue that the points in C1 form a cluster.

## 3 The Closest Pair Problem

Now that we have defined the distance between points and clusters, we need an algorithm that finds, among a set of clusters, two clusters that are closest to each other (in the case of **HierarchicalClustering**), or, among a set of centers, a closest center to a given point (in the case of **KMeansClustering**). In this Module, we will approach this task by solving the Closest Pair problem, defined as follows:

- **Input:** A set $P$ of (distinct) points and a distance measure $d$ defined on every two points in $P$.
- **Output:** A pair of distinct points in $P$ that are closest to each other under the distance measure $d$.

Notice that the solution of the problem might not be unique (that is, more than a single pair of points might be closest), in which case we are interested in an arbitrary one of those pairs with the smallest pairwise distance.

A simple brute-force algorithm can solve this problem, as given by the pseudo-code of Algorithm **SlowClosestPair**. Notice the notation we use for finding the minimum of two tuples $\min\{(d_1, p_1, q_1), (d_2, p_2, q_2)\}$, which returns the tuple that has the smallest first element (that is, it returns tuple $(d_1, p_1, q_1)$ if $d_1 < d_2$, and $(d_2, p_2, q_2)$ otherwise). In case the two tuples have the same first element, one of them is returned arbitrarily.

**Algorithm 3: SlowClosestPair.**

**Input**: A set $P$ of ($\geq 2$) points whose $i$th point, $p_i$, is a pair $(x_i, y_i)$.
**Output**: A tuple $(d, i, j)$ where $d$ is the smallest pairwise distance of points in $P$, and $i, j$ are the indices of two points whose distance is $d$.

```
(d, i, j) ← (∞, −1, −1);
foreach p_u ∈ P do
    foreach p_v ∈ P (u ≠ v) do
        (d, i, j) ← min{(d, i, j), (d_{p_u,p_v}, u, v)};   // min compares the first element of each tuple
return (d, i, j);
```

Can we do better than **SlowClosestPair** in terms of running time? We will now consider a divide-and-conquer algorithm for this problem, **FastClosestPair**.

**Algorithm 4: FastClosestPair.**

**Input**: A set $P$ of ($\geq 2$) points whose $i$th point, $p_i$, is a pair $(x_i, y_i)$, sorted in nondecreasing order of their horizontal ($x$) coordinates.

**Output**: A tuple $(d, i, j)$ where $d$ is the smallest pairwise distance of the points in $P$, and $i, j$ are the indices of two points whose distance is $d$.

$n \leftarrow |P|$;
**if** $n \leq 3$ **then**
    $(d, i, j) \leftarrow$ **SlowClosestPair**$(P)$;
**else**
    $m \leftarrow \lfloor n/2 \rfloor$;
    $P_\ell \leftarrow \{p_i : 0 \leq i \leq m-1\}$; $P_r \leftarrow \{p_i : m \leq i \leq n-1\}$; // $P_\ell$ and $P_r$ are also sorted
    $(d_\ell, i_\ell, j_\ell) \leftarrow$ **FastClosestPair**$(P_\ell)$;
    $(d_r, i_r, j_r) \leftarrow$ **FastClosestPair**$(P_r)$;
    $(d, i, j) \leftarrow \min\{(d_\ell, i_\ell, j_\ell), (d_r, i_r + m, j_r + m)\}$;
    $mid \leftarrow \frac{1}{2}(x_{m-1} + x_m)$; // center line of strip
    $(d, i, j) \leftarrow \min\{(d, i, j),$ **ClosestPairStrip**$(P, mid, d)\}$;
**return** $(d, i, j)$;

**Algorithm 5: ClosestPairStrip.**

**Input**: A set $P$ of points whose $i$th point, $p_i$, is a pair $(x_i, y_i)$; $mid$ and $w$, both of which are real numbers.

**Output**: A tuple $(d, i, j)$ where $d$ is the smallest pairwise distance of points in $P$ whose horizontal ($x$) coordinates are within $w$ from $mid$.

Let $S$ be a list of the set $\{i : |x_i - mid| < w\}$;
Sort the indices in $S$ in nondecreasing order of the vertical ($y$) coordinates of their associated points;
$k \leftarrow |S|$;
$(d, i, j) \leftarrow (\infty, -1, -1)$;
**for** $u \leftarrow 0$ **to** $k-2$ **do**
    **for** $v \leftarrow u+1$ **to** $\min\{u+3, k-1\}$ **do**
        $(d, i, j) \leftarrow \min\{(d, i, j), (d_{P_{S[u]}, P_{S[v]}}, S[u], S[v])\}$;
**return** $(d, i, j)$;

**The content below is my answer to this project task.**

# My Work

First, here is a Class Cluster to store the different clusters we generate.

In [1]:
```
"""
Cluster class for Module 3
"""

import math


class Cluster:
    """
    Class for creating and merging clusters of counties
    """

    def __init__(self, fips_codes, horiz_pos, vert_pos, population,
risk):
        """
        Create a cluster based the models a set of counties' data
        """
        self._fips_codes = fips_codes
        self._horiz_center = horiz_pos
        self._vert_center = vert_pos
        self._total_population = population
        self._averaged_risk = risk

    def __repr__(self):
        """
        String representation assuming the module is "alg_cluster".
        """
        rep = "Cluster("
        rep += str(self._fips_codes) + ", "
        rep += str(self._horiz_center) + ", "
        rep += str(self._vert_center) + ", "
        rep += str(self._total_population) + ", "
        rep += str(self._averaged_risk) + ")"
        return rep

    def fips_codes(self):
        """
        Get the cluster's set of FIPS codes
        """
        return self._fips_codes
```

```
    def horiz_center(self):
        """
        Get the averged horizontal center of cluster
        """
        return self._horiz_center

    def vert_center(self):
        """
        Get the averaged vertical center of the cluster
        """
        return self._vert_center

    def total_population(self):
        """
        Get the total population for the cluster
        """
        return self._total_population

    def averaged_risk(self):
        """
        Get the averaged risk for the cluster
        """
        return self._averaged_risk

    def copy(self):
        """
        Return a copy of a cluster
        """
        copy_cluster = Cluster(set(self._fips_codes), self._horiz_c
enter, self._vert_center,
                               self._total_population, self._averag
ed_risk)
        return copy_cluster

    def distance(self, other_cluster):
        """
        Compute the Euclidean distance between two clusters
        """
        vert_dist = self._vert_center - other_cluster.vert_center()
        horiz_dist = self._horiz_center - other_cluster.horiz_cente
r()
        return math.sqrt(vert_dist ** 2 + horiz_dist ** 2)

    def merge_clusters(self, other_cluster):
        """
        Merge one cluster into another
        The merge uses the relatively populations of each
        cluster in computing a new center and risk

        Note that this method mutates self
        """
        if len(other_cluster.fips_codes()) == 0:
            return self
        else:
```

```
            self._fips_codes.update(set(other_cluster.fips_codes())
)

            # compute weights for averaging
            self_weight = float(self._total_population)
            other_weight = float(other_cluster.total_population())
            self._total_population = self._total_population + other
_cluster.total_population()
            self_weight /= self._total_population
            other_weight /= self._total_population

            # update center and risk using weights
            self._vert_center = self_weight * self._vert_center + o
ther_weight * other_cluster.vert_center()
            self._horiz_center = self_weight * self._horiz_center +
other_weight * other_cluster.horiz_center()
            self._averaged_risk = self_weight * self._averaged_risk
+ other_weight * other_cluster.averaged_risk()
            return self

    def cluster_error(self, data_table):
        """
        Input: data_table is the original table of cancer data used
in creating the cluster.

        Output: The error as the sum of the square of the distance
from each county
        in the cluster to the cluster center (weighted by its popul
ation)
        """
        # Build hash table to accelerate error computation
        fips_to_line = {}
        for line_idx in range(len(data_table)):
            line = data_table[line_idx]
            fips_to_line[line[0]] = line_idx

        # compute error as weighted squared distance from counties
to cluster center
        total_error = 0
        counties = self.fips_codes()
        for county in counties:
            line = data_table[fips_to_line[county]]
            singleton_cluster = Cluster(set([line[0]]), line[1], li
ne[2], line[3], line[4])
            singleton_distance = self.distance(singleton_cluster)
            total_error += (singleton_distance ** 2) * singleton_cl
uster.total_population()
        return total_error
```

Then I apply **2 closest-pair methods** and **2 clustering mathod**

In [2]:
```
"""
```

```
We will implement five functions:

slow_closest_pair(cluster_list)
fast_closest_pair(cluster_list)
closest_pair_strip(cluster_list, horiz_center, half_width)
hierarchical_clustering(cluster_list, num_clusters)
kmeans_clustering(cluster_list, num_clusters, num_iterations)

where cluster_list is a 2D list of clusters in the plane
"""


######################################################
# Code for closest pairs of clusters

def min_pair(pair_a, pair_b):
    """
    get the min distance pair
    """
    if pair_a[0] <= pair_b[0]:
        return pair_a
    return pair_b


def slow_closest_pair(cluster_list):
    """
    Compute the distance between the closest pair of clusters in a
list (slow)

    Input: cluster_list is the list of clusters

    Output: tuple of the form (dist, idx1, idx2) where the centers
of the clusters
    cluster_list[idx1] and cluster_list[idx2] have minimum distance
dist.
    """
    min_dis = float('inf')
    closest_pair = (min_dis, 0, 0)
    for cluster in cluster_list:
        index_c = cluster_list.index(cluster)
        for other_cluster in cluster_list[cluster_list.index(cluste
r) + 1:]:
            dis = cluster.distance(other_cluster)
            if dis < closest_pair[0]:
                closest_pair = (dis, index_c, cluster_list.index(ot
her_cluster))

    return closest_pair


def closest_pair_strip(cluster_list, horiz_center, half_width):
    """
    Helper function to compute the closest pair of clusters in a ve
rtical strip
```

```
    Input: cluster_list is a list of clusters produced by fast_clos
est_pair
    horiz_center is the horizontal position of the strip's vertical
center line
    half_width is the half the width of the strip (i.e; the maximum
horizontal distance
    that a cluster can lie from the center line)

    Output: tuple of the form (dist, idx1, idx2) where the centers
of the clusters
    cluster_list[idx1] and cluster_list[idx2] lie in the strip and
have minimum distance dist.
    """
    # don't go out of [horiz_l, horiz_r]
    cluster_list_m = []
    horiz_l = max(cluster_list[0].horiz_center(), horiz_center - ha
lf_width)
    horiz_r = min(cluster_list[-1].horiz_center(), horiz_center + h
alf_width)
    # horiz_l = horiz_center - half_width
    # horiz_r = horiz_center + half_width
    for cluster in cluster_list:
        if horiz_l <= cluster.horiz_center() <= horiz_r:
            cluster_list_m.append(cluster)

    closest_pair = (float('inf'), -1, -1)
    cluster_list_m.sort(key=lambda clstr: clstr.vert_center())
    lenth = len(cluster_list_m)
    for index_i in range(lenth - 1):
        for index_j in range(index_i + 1, min(index_i + 4, lenth)):
            dis = cluster_list_m[index_i].distance(cluster_list_m[i
ndex_j])
            real_index_i = cluster_list.index(cluster_list_m[index_
i])
            real_index_j = cluster_list.index(cluster_list_m[index_
j])
            closest_pair = min_pair(closest_pair, (dis, min(real_in
dex_i, real_index_j),
                                                   max(real_index_i
, real_index_j)))

    return closest_pair


def fast_closest_pair(cluster_list):
    """
    Compute the distance between the closest pair of clusters in a
list (fast)

    Input: cluster_list is list of clusters SORTED such that horizo
ntal positions of their
    centers are in ascending order
```

```
    Output: tuple of the form (dist, idx1, idx2) where the centers
of the clusters
    cluster_list[idx1] and cluster_list[idx2] have minimum distance
dist.
    """
    # print(cluster_list)
    if len(cluster_list) < 2:
        return float('inf'), -1, -1
    if len(cluster_list) == 2:
        return cluster_list[0].distance(cluster_list[1]), 0, 1

    mid_num = len(cluster_list) // 2
    closest_pair_l = fast_closest_pair(cluster_list[:mid_num])
    closest_pair_r = fast_closest_pair(cluster_list[mid_num:])
    closest_pair_r = (closest_pair_r[0], closest_pair_r[1] + mid_nu
m, closest_pair_r[2] + mid_num)
    closest_pair = min_pair(closest_pair_l, closest_pair_r)

    mid_horiz = (cluster_list[mid_num - 1].horiz_center() + cluster
_list[mid_num].horiz_center()) / 2
    closest_pair = min_pair(closest_pair, closest_pair_strip(cluste
r_list, mid_horiz, closest_pair[0]))

    return closest_pair


##############################################################################
###
# Code for hierarchical clustering


def hierarchical_clustering(cluster_list, num_clusters):
    """
    Compute a hierarchical clustering of a set of clusters
    Note: the function may mutate cluster_list

    Input: List of clusters, integer number of clusters
    Output: List of clusters whose length is num_clusters
    """
    cluster_list_copy = []
    for cluster in cluster_list:
        cluster_list_copy.append(cluster.copy())
    cluster_list_copy.sort(key=lambda clstr: clstr.horiz_center())
    while len(cluster_list_copy) > num_clusters:
        closest_pair = fast_closest_pair(cluster_list_copy)
        cluster_list_copy[closest_pair[1]].merge_clusters(cluster_l
ist_copy[closest_pair[2]])

        cluster_t = cluster_list_copy[closest_pair[1]]
        cluster_list_copy.remove(cluster_list_copy[closest_pair[2]]
)
        cluster_list_copy.remove(cluster_list_copy[closest_pair[1]]
)
        insert_index = -1
```

```
        for cluster in cluster_list_copy[closest_pair[1]:]:
            if cluster_t.horiz_center() < cluster.horiz_center():
                insert_index = cluster_list_copy.index(cluster)
                break
        if insert_index == -1:
            cluster_list_copy.append(cluster_t)
        else:
            cluster_list_copy.insert(insert_index, cluster_t)
    return cluster_list_copy


######################################################################
###
# Code for k-means clustering


def deep_copy(cluster):
    """
    deepcopy written by myself, not from importing copy module
    """
    fips_codes = set(cluster.fips_codes())
    horiz_center = cluster.horiz_center()
    vert_center = cluster.vert_center()
    population = cluster.total_population()
    risk = cluster.averaged_risk()
    return Cluster(fips_codes, horiz_center, vert_center, populatio
n, risk)


def kmeans_clustering(cluster_list, num_clusters, num_iterations):
    """
    Compute the k-means clustering of a set of clusters
    Note: the function may not mutate cluster_list

    Input: List of clusters, integers number of clusters and number
of iterations
    Output: List of clusters whose length is num_clusters
    """

    # position initial clusters at the location of clusters with la
rgest populations
    cluster_list_copy = []
    for cluster in cluster_list:
        cluster_list_copy.append(cluster.copy())
    cluster_list_copy.sort(key=lambda clstr: clstr.total_population
(), reverse=True)
    cluster_list_ans = []
    for index in range(num_clusters):
        cluster_list_ans.append(cluster_list_copy[index])
    for _ in range(num_iterations):
        cluster_list_ans_after = []
        for index in range(num_clusters):
            cluster_list_ans_after.append(Cluster(set([]), 0.0, 0.0
, 0.0, 0.0))
```

```
        for cluster in cluster_list_copy:
            chose_num = -1
            min_dis = float('inf')
            for center in cluster_list_ans:
                dis = center.distance(cluster)
                if dis < min_dis:
                    min_dis = dis
                    chose_num = cluster_list_ans.index(center)
            cluster_list_ans_after[chose_num].merge_clusters(cluste
r)

        for index in range(num_clusters):
            cluster_list_ans[index] = deep_copy(cluster_list_ans_af
ter[index])

    return cluster_list_ans
```

In [3]:
```
# Test Data

LIST_A = [Cluster({'1'}, 0.05, 0.11, 1, 0),
          Cluster({'2'}, 0.26, 0.92, 1, 0),
          Cluster({'3'}, 0.34, 0.57, 1, 0),
          Cluster({'4'}, 0.35, 0.15, 1, 0),
          Cluster({'5'}, 0.6, 0.41, 1, 0),
          Cluster({'6'}, 0.89, 0.28, 1, 0)]

print(slow_closest_pair(LIST_A))
print(fast_closest_pair(LIST_A))
print(hierarchical_clustering(LIST_A, 2))
print(kmeans_clustering(LIST_A, 2, 4))
```

```
(0.3026549190084311, 0, 3)
(0.3026549190084311, 0, 3)
[Cluster({'2'}, 0.26, 0.92, 1, 0), Cluster({'3', '4', '6', '5', '1
'}, 0.44599999999999995, 0.304, 5, 0.0)]
[Cluster({'6', '4', '5', '1'}, 0.4724999999999999, 0.2375, 4.0, 0.
0), Cluster({'2', '3'}, 0.30000000000000004, 0.745, 2.0, 0.0)]
```

Then to answer the Question related to these methods.

**Question 1**: How about the different **time efficiency** of fast_closest_pair and slow_closest_pair?

The **slow_closest_pair**: O(n^2)

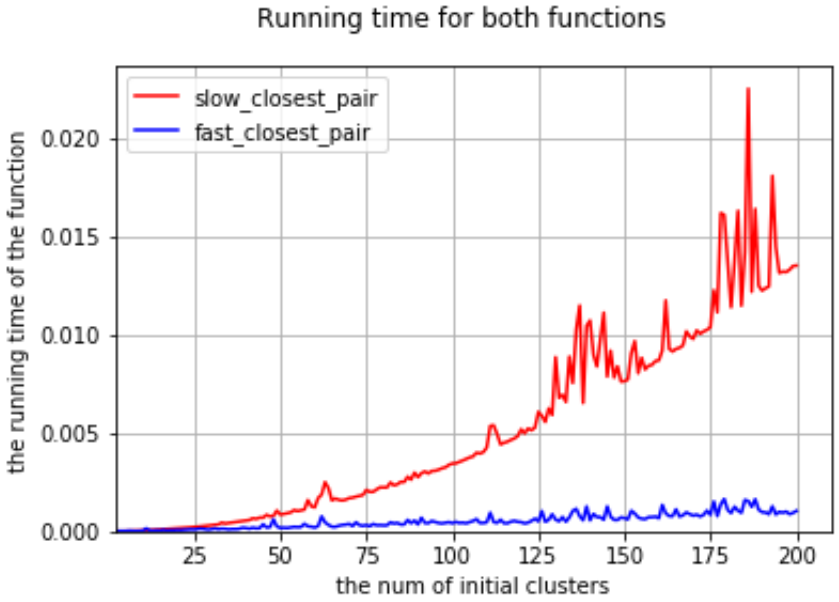
The **fast_closest_pair** use binary search and T(n) = 2T(n/2) + O(n) and T(2) = T(1) = O(1). Therefore, the time efficiency of fast_closest_pair is O(nlogn)

And I generated random data to confirm this conclusion as below.

In [4]:
```
%matplotlib inline
```

In [5]:
```
"""
Time efficency of fast_closest_pair and slow_closest_pair?
"""

import time
import random
import matplotlib.pyplot as plt


def gen_random_clusters(num_clusters):
    random_clusters = []
    for _ in range(num_clusters):
        horiz_center = random.random() * 2 - 1
        vert_center = random.random() * 2 - 1
        random_clusters.append(Cluster(set([]), horiz_center, vert_
center, 1.0, 0))
    # print(random_clusters)
    return random_clusters


def gen_time():
    slow_time = []
    fast_time = []
    for num in range(2, 201):
        random_clusters = gen_random_clusters(num)

        start = time.clock()
        slow_closest_pair(random_clusters)
        slow_time.append(time.clock() - start)

        start = time.clock()
        fast_closest_pair(random_clusters)
        fast_time.append(time.clock() - start)

    return slow_time, fast_time


def draw_plot():
    slow_time, fast_time = gen_time()
    plt.plot(list(range(2, 201)), slow_time, '-r', label="slow_clos
est_pair")
    plt.plot(list(range(2, 201)), fast_time, '-b', label="fast_clos
est_pair")
    plt.xlabel("the num of initial clusters")
    plt.ylabel("the running time of the function")
    plt.suptitle("Running time for both functions")
    plt.grid()
    plt.xlim(2)
    plt.ylim(0)
    plt.legend(loc="upper left")
```

```
    plt.show()


draw_plot()
```




Then to visualize the clusters

In [6]:
```
"""
Some provided code for plotting the clusters using matplotlib
"""

import math
import matplotlib.pyplot as plt

# URLS for various important datasets
DIRECTORY = "http://commondatastorage.googleapis.com/codeskulptor-assets/"
MAP_URL = DIRECTORY + "data_clustering/USA_Counties.png"

# Define colors for clusters.  Display a max of 16 clusters.
COLORS = ['Aqua', 'Yellow', 'Blue', 'Fuchsia', 'Black', 'Green', 'Lime', 'Maroon', 'Navy', 'Olive', 'Orange', 'Purple',
          'Red', 'Brown', 'Teal']


# Helper functions

def circle_area(pop):
    """
    Compute area of circle proportional to population
    """
    return math.pi * pop / (200.0 ** 2)
```

```
def plot_clusters(data_table, cluster_list, draw_centers=False):
    """
    Create a plot of clusters of counties
    """

    fips_to_line = {}
    for line_idx in range(len(data_table)):
        fips_to_line[data_table[line_idx][0]] = line_idx

    # Load map image
    map_img = plt.imread('USA_Counties.png')

    # Scale plot to get size similar to CodeSkulptor version
    ypixels, xpixels, bands = map_img.shape
    DPI = 60.0  # adjust this constant to resize your plot
    xinch = xpixels / DPI
    yinch = ypixels / DPI
    fig = plt.figure(figsize=(xinch, yinch))
    implot = plt.imshow(map_img)

    # draw the counties colored by cluster on the map
    if not draw_centers:
        for cluster_idx in range(len(cluster_list)):
            cluster = cluster_list[cluster_idx]
            cluster_color = COLORS[cluster_idx % len(COLORS)]
            for fips_code in cluster.fips_codes():
                line = data_table[fips_to_line[fips_code]]
                plt.scatter(x=[line[1]], y=[line[2]], s=circle_area
(line[3]), lw=1,
                            facecolors=cluster_color, edgecolors=cl
uster_color)

    # add cluster centers and lines from center to counties
    else:
        for cluster_idx in range(len(cluster_list)):
            cluster = cluster_list[cluster_idx]
            cluster_color = COLORS[cluster_idx % len(COLORS)]
            for fips_code in cluster.fips_codes():
                line = data_table[fips_to_line[fips_code]]
                plt.scatter(x=[line[1]], y=[line[2]], s=circle_area
(line[3]), lw=1,
                            facecolors=cluster_color, edgecolors=cl
uster_color, zorder=1)
        for cluster_idx in range(len(cluster_list)):
            cluster = cluster_list[cluster_idx]
            cluster_color = COLORS[cluster_idx % len(COLORS)]
            cluster_center = (cluster.horiz_center(), cluster.vert_
center())
            for fips_code in cluster.fips_codes():
                line = data_table[fips_to_line[fips_code]]
                plt.plot([cluster_center[0], line[1]], [cluster_cen
ter[1], line[2]], cluster_color, lw=1, zorder=2)
        for cluster_idx in range(len(cluster_list)):
            cluster = cluster_list[cluster_idx]
```

```
            cluster_color = COLORS[cluster_idx % len(COLORS)]
            cluster_center = (cluster.horiz_center(), cluster.vert_
center())
            cluster_pop = cluster.total_population()
            plt.scatter(x=[cluster_center[0]], y=[cluster_center[1]
], s=circle_area(cluster_pop), lw=2,
                        facecolors="none", edgecolors="black", zord
er=3)

    fig.savefig('ppt1.png', ppi=1000)
    return plt.figure()
```

In [7]:

```
"""
Example code for creating and visualizing
cluster of county-based cancer risk data
"""

import math
import urllib.request as urllib2
import csv


######################################################
# Code to load data tables

# URLs for cancer risk data tables of various sizes
# Numbers indicate number of counties in data table

DATA_3108_URL = open('unifiedCancerData_3108.csv');
DATA_896_URL = open('unifiedCancerData_896.csv');
DATA_290_URL = open('unifiedCancerData_290.csv');
DATA_111_URL = open('unifiedCancerData_111.csv');


def load_data_table(data_url):
    """
    Import a table of county-based cancer risk data
    from a csv format file
    """
    data_file = urllib2.urlopen(data_url)
    data = data_file.read()
    data_lines = data.split('\n')
    print("Loaded", len(data_lines), "data points")
    data_tokens = [line.split(',') for line in data_lines]
    return [[tokens[0], float(tokens[1]), float(tokens[2]), int(tok
ens[3]), float(tokens[4])]
            for tokens in data_tokens]
```

load 3108 cancer data

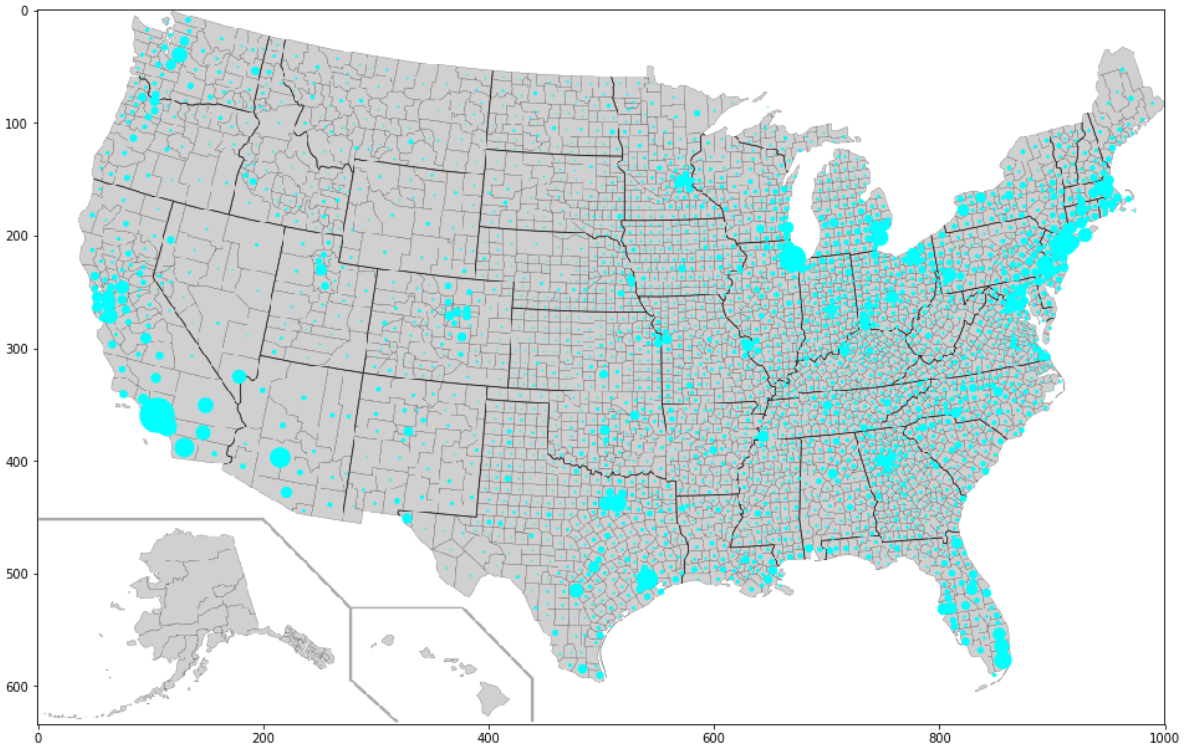
In [8]:
```
dataReader = csv.reader(DATA_3108_URL)
dataData = list(dataReader)
for line in range(len(dataData)):
    dataData[line][0] = str(dataData[line][0])
    dataData[line][1] = float(dataData[line][1])
    dataData[line][2] = float(dataData[line][2])
    dataData[line][3] = int(dataData[line][3])
    dataData[line][4] = float(dataData[line][4])

data_table = dataData

singleton_list = []
for line in data_table:
    singleton_list.append(Cluster(set([line[0]]), line[1], line[2],
line[3], line[4]))

singleton_list.sort(key=lambda clstr: clstr.horiz_center())
```

compute a list of clusters and plot a list of clusters by **hierarchical clustering method** (Using minutes of time)

In [9]:
```
cluster_list = hierarchical_clustering(singleton_list, 1)
print("Displaying", len(cluster_list), "hierarchical clusters")
# plot_clusters(data_table, cluster_list, Ture)
plot_clusters(data_table, cluster_list, False)  #hide cluster cente
rs
```

```
Displaying 1 hierarchical clusters
```

Out[9]: <Figure size 432x288 with 0 Axes>

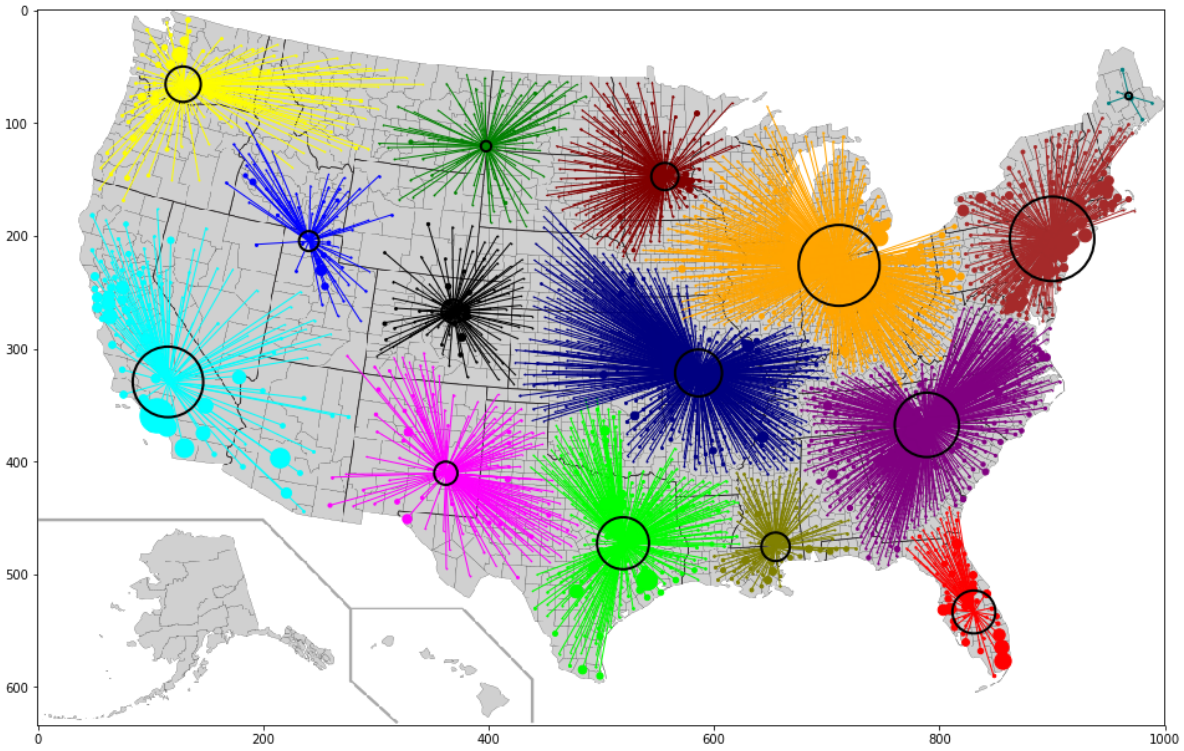



<Figure size 432x288 with 0 Axes>

In [10]:
```
cluster_list = hierarchical_clustering(singleton_list, 15)
print("Displaying", len(cluster_list), "hierarchical clusters")
# plot_clusters(data_table, cluster_list, False)
plot_clusters(data_table, cluster_list, True)  #add cluster centers
```

```
Displaying 15 hierarchical clusters
```

Out[10]:
```
<Figure size 432x288 with 0 Axes>
```



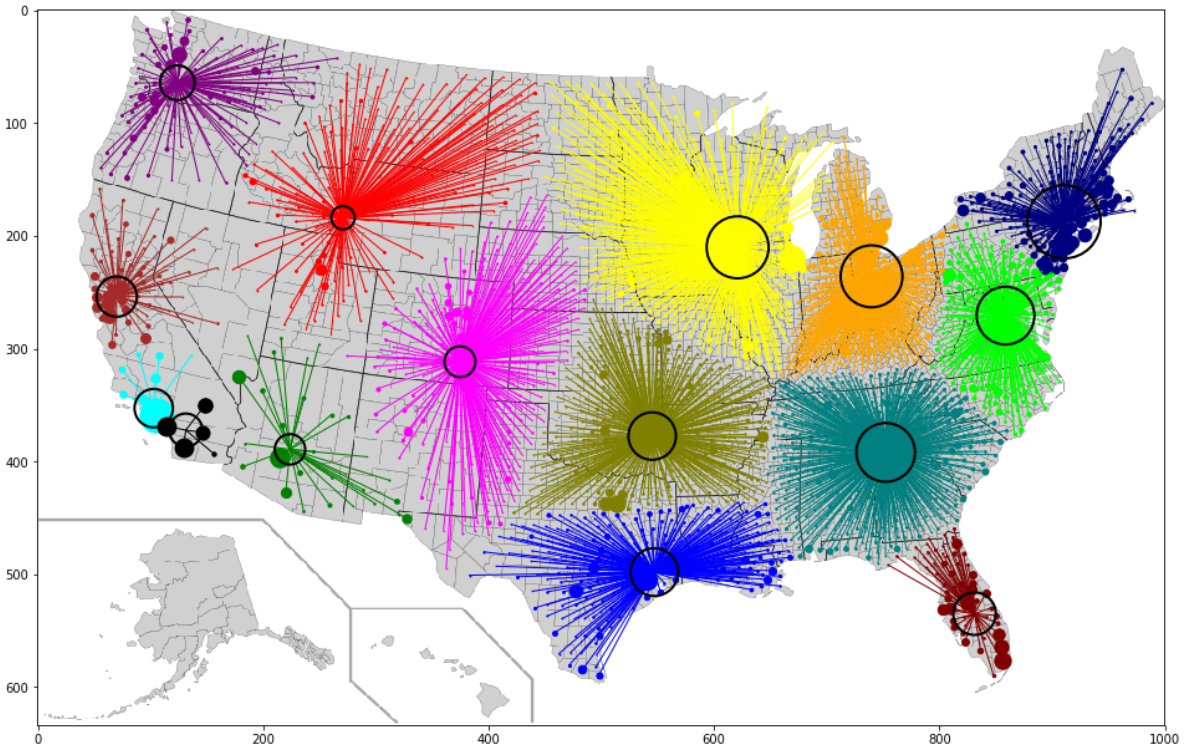

```
<Figure size 432x288 with 0 Axes>
```

compute a list of clusters and plot a list of clusters by **k-means clustering method**

In [11]:
```
cluster_list = kmeans_clustering(singleton_list, 15, 5)
print("Displaying", len(cluster_list), "k-means clusters")
# plot_clusters(data_table, cluster_list, False)
plot_clusters(data_table, cluster_list, True)  #add cluster centers
```

```
Displaying 15 k-means clusters
```

Out[11]:
```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

To compare the time efficiency of 2 clustering method:

Suppose the number of counties is **n**, the number of clusters output is **m**

The running time of **the hierarchical method is O((n^2)*log(n))**

and the running time of **the k means method is O(nm) and 0<m≤n.**

So hierarchical method is much slower than other one.

Let's confirm this conclusion (Use 111 data set).

In [12]:
```
dataReader = csv.reader(DATA_111_URL)
dataData = list(dataReader)
for line in range(len(dataData)):
    dataData[line][0] = str(dataData[line][0])
    dataData[line][1] = float(dataData[line][1])
    dataData[line][2] = float(dataData[line][2])
    dataData[line][3] = int(dataData[line][3])
    dataData[line][4] = float(dataData[line][4])

data_table = dataData

singleton_list = []
for line in data_table:
    singleton_list.append(Cluster(set([line[0]]), line[1], line[2],
line[3], line[4]))

singleton_list.sort(key=lambda clstr: clstr.horiz_center())
```

In [13]:
```
%%timeit -n 100
cluster_list = hierarchical_clustering(singleton_list, 9)
```

```
57.7 ms ± 2.05 ms per loop (mean ± std. dev. of 7 runs, 100 loops
each)
```

In [14]:
```
%%timeit -n 100
cluster_list = kmeans_clustering(singleton_list, 9, 5)
```

```
5.12 ms ± 64 µs per loop (mean ± std. dev. of 7 runs, 100 loops ea
ch)
```

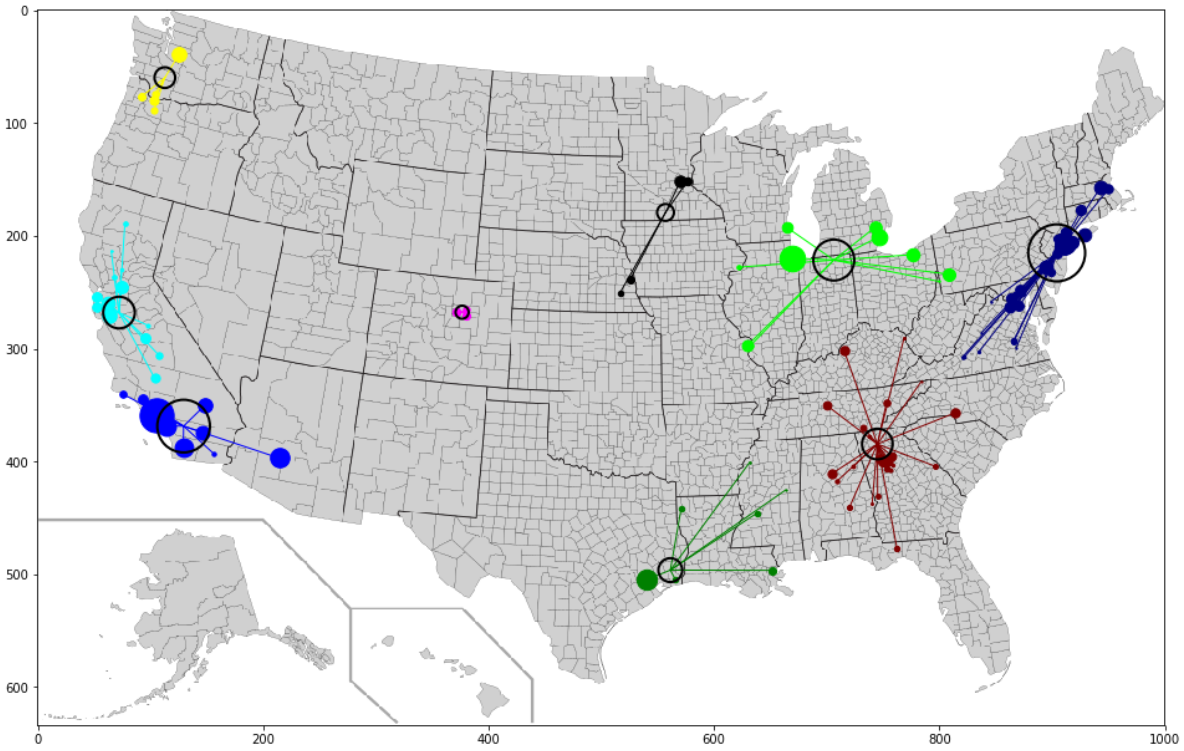
**Question 2**: Which clustering method is with less error and requires less supervision from human?

Let's use 111 data set

In [15]:
```
cluster_list = hierarchical_clustering(singleton_list, 9)
print("Displaying", len(cluster_list), "hierarchical clusters")
# plot_clusters(data_table, cluster_list, False)
plot_clusters(data_table, cluster_list, True)  #add cluster centers
```

```
Displaying 9 hierarchical clusters
```

Out[15]:
```
<Figure size 432x288 with 0 Axes>
```

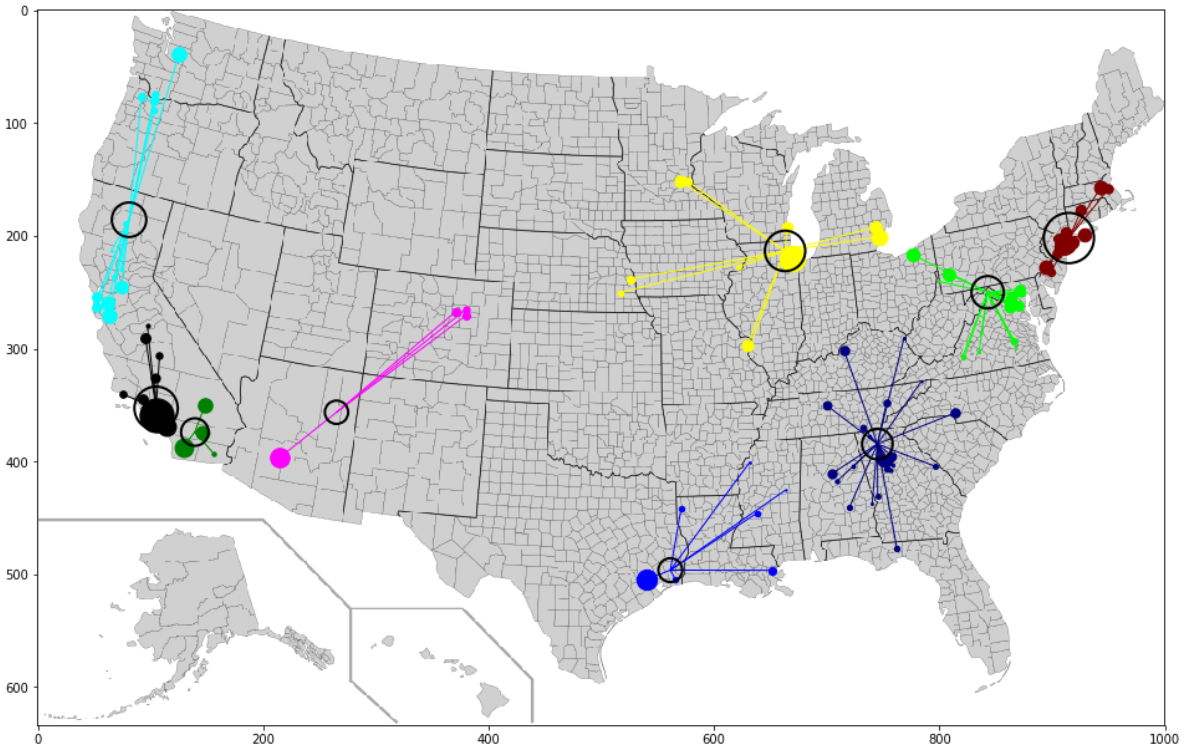



```
<Figure size 432x288 with 0 Axes>
```

In [16]:
```
cluster_list = kmeans_clustering(singleton_list, 9, 5)
print("Displaying", len(cluster_list), "k-means clusters")
# plot_clusters(data_table, cluster_list, False)
plot_clusters(data_table, cluster_list, True)  #add cluster centers
```

```
Displaying 9 k-means clusters
```

Out[16]:
```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

As we can seen, in the graph k-means method generated, the purple cluster and blue cluster in the left is **worse** than the counterpart in the graph that hierarchical method generated. (some points are too far to each other and it is better not to cluster them togather)

Why this happened? This is because k-means method chooses counties have largest population initially and, unluckily, some counties which are great choices to be a center just have small population.

**Question 3**: Is hierarchical method always more precise than k-means method?

One way to make this concept more precise is to formulate a mathematical measure of the error associated with a cluster. Given a cluster C, itserror is the sum of the squares of the distances from each county in the cluster to the cluster's center, weighted by each county's population. If pi is the position of the county and wi is its population, the cluster's error is:

$$error(C) = \sum_{p_i \in C} w_i (d_{p_i c})^2$$

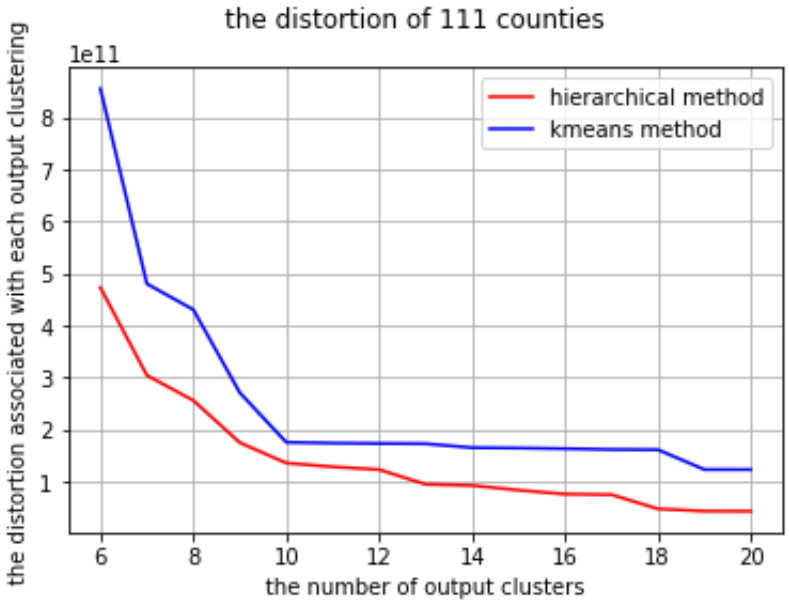
where c is the center of the cluster C. The Cluster class includes a method cluster_error(data_table) that takes a Cluster object and the original data table associated with the counties in the cluster and computes the error associated with a given cluster.

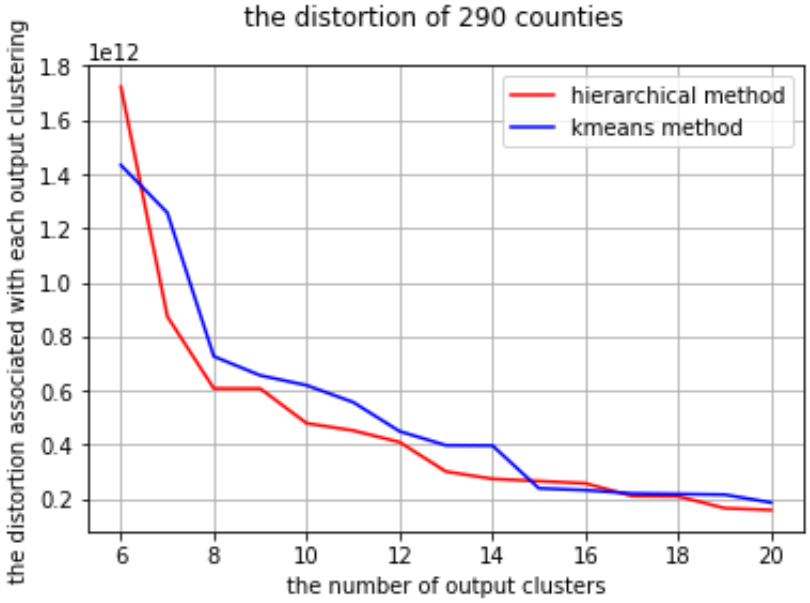
Given a list of clusters L, the distortion of the clustering is the sum of the errors associated with its clusters.

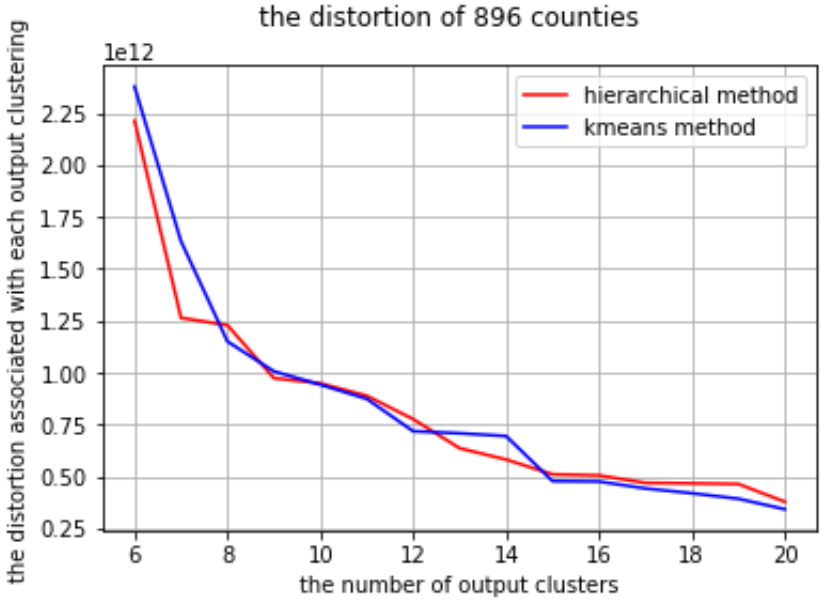
$$distortion(L) = \sum_{C \in L} error(C).$$

In [17]:
```
"""
compute distortion
"""

import matplotlib.pyplot as plt
import csv

DATA_TABLE = []
DATA_URL = ['unifiedCancerData_111.csv',
            'unifiedCancerData_290.csv',
            'unifiedCancerData_896.csv']

for index in range(3):
    with open(DATA_URL[index]) as datafile:
        database = csv.reader(datafile)
        data_table = []
        for row in database:
            data_table.append([row[0], float(row[1]), float(row[2])
, int(row[3]), float(row[4])])
        DATA_TABLE.append(data_table)


def read_data(data_num):
    cluster_list = []
    for row in DATA_TABLE[data_num]:
        cluster_list.append(Cluster(set([row[0]]), row[1], row[2],
row[3], row[4]))

    cluster_list.sort(key=lambda clstr: clstr.horiz_center())
    return cluster_list
```

```
def compute_distortion(cluster_list, data_table):
    distortion = 0
    for cluster in cluster_list:
        distortion += cluster.cluster_error(data_table)
    return distortion

def run_example():
    cluster_lists = []
    for idx in range(3):
        cluster_lists.append(read_data(idx))

        # hierarchical method
        cluster_list_pre = cluster_lists[idx]
        hierarchical_distortion = []
        for num_clusters in range(20, 5, -1):
            cluster_list_pre = hierarchical_clustering(cluster_list
_pre, num_clusters)
            hierarchical_distortion.append(compute_distortion(clust
er_list_pre, DATA_TABLE[idx]))
        hierarchical_distortion.reverse()

        # kmeans method
        kmeans_distortion = []
        for num_clusters in range(6, 21):
            kmeans_distortion.append(compute_distortion(kmeans_clus
tering(cluster_lists[idx], num_clusters, 5),
                                                        DATA_TABLE[
idx]))

        # plot
        plt.plot(list(range(6, 21)), hierarchical_distortion, '-r',
label='hierarchical method')
        plt.plot(list(range(6, 21)), kmeans_distortion, '-b', label
='kmeans method')
        plt.xlabel('the number of output clusters')
        plt.ylabel('the distortion associated with each output clus
tering')
        plt.suptitle('the distortion of ' + DATA_URL[idx][18:21] +
' counties')
        plt.grid()
        plt.legend(loc="upper right")
        plt.savefig(str(idx))
        plt.show()


run_example()
```


the distortion of 111 counties
1e11
hierarchical method
kmeans method
the distortion associated with each output clustering
the number of output clusters


the distortion of 290 counties
1e12
hierarchical method
kmeans method
the distortion associated with each output clustering
the number of output clusters


the distortion of 896 counties
1e12
hierarchical method
kmeans method
the distortion associated with each output clustering
the number of output clusters

So **only in 111 county data set**, hierarchical method consistently produce lower distortion, which means hierarchical method is precise only in 111 data set.

# Conclusion

For time efficienty: k-means method is O(nm),0<m<=n; hierarchical method is O(n^2*logn). The k-means method is much more efficient to cluster counties.

However, hierarchical method is more precise to cluster counties than k-means one when the data is not large.

Therefore, we'd better use hierarchical method when we process data below 200 sets, and use k-means method to process larger data.