UI – Zadanie 3(c)

Hredzhev Oleksandr

ID: 123560

Problem

The task is to write a code that will generate 20 points on a field -5000 - 5000, then generate another 20,000 points, each of which is generated with a random deviation from -100 to 100 from a random (already existing) point on the field (and does not leave beyond its borders). Next, you need to create an agglomerative clustering algorithm that will additionally calculate the medoid and centroid for each cluster and then analyze the success of the clusters.

Optimizations:

* I use the combinations function from the itertools library to more efficiently calculate the distances between pairs of points and write them into a matrix.

*Also, to speed up the functionality, after each cycle the matrix is reduced by 1 column and 1 row

Below I will show how these algorithms are implemented in code:

```
def generate(points_count):
    # Generating initial 20 unique points
    num_initial_points = 20
    points = np.random.uniform(low=-5000, high=5000, size=(0, 2))

    while len(np.unique(points, axis=0)) < num_initial_points:
        new_point = np.random.uniform(low=-5000, high=5000, size=(1, 2))

        if not np.any(np.all(np.isclose(points, new_point), axis=1)):
            points = np.vstack([points, new_point])

    # Creating an additional 20,000 points considering the given conditions
    for _ in range(points_count):
        # Randomly choosing one of the generated points
        selected_point = points[np.random.randint(0, len(points))]

        # Determining distances to edges
        distance_to_edge_x = 5000 - abs(selected_point[0])
        distance_to_edge_y = 5000 - abs(selected_point[1])

        while True:
            # Adjusting the interval if the point is close to the edge
            if 100 > distance_to_edge_x > -100:
                x_offset = np.random.uniform(-distance_to_edge_x,
distance_to_edge_x)
            else:
                x_offset = np.random.uniform(-100, 100)
            if 100 > distance_to_edge_y > -100:
                y_offset = np.random.uniform(-distance_to_edge_y,
distance_to_edge_y)
            else:
                y_offset = np.random.uniform(-100, 100)

            # Adding a new point in the two-dimensional space considering the
offsets
            new_point = selected_point + [x_offset, y_offset]

            # Ensuring the new point stays within the boundaries [-5000,
5000]
            new_point = np.clip(new_point, -5000, 5000)
```

```
            # Checking if the point 'new_point' already exists in the
'points' array
            if not np.any(np.all(np.isclose(points, new_point), axis=1)):
                # If not, adding the new point and exiting the loop
                points = np.vstack([points, new_point])
                break
```

Here the initial (20) and additional (the number is entered from the console) generation of points takes place

```
def agglomerative_clustering(points, k):
    n = len(points)

    current_size = n

    # Computing the distance matrix
    distances = cdist(points, points)

    # List to store clusters
    clusters = [[i] for i in range(n)]

    while len(clusters) > k:
        min_dist = np.inf
        merge_indices = (0, 0)

        # Using current_size to limit the matrix size
        for i, j in combinations(range(current_size), 2):
            if distances[i, j] < min_dist:
                min_dist = distances[i, j]
                merge_indices = (i, j)

        clusters[merge_indices[0]] += clusters[merge_indices[1]]
        del clusters[merge_indices[1]]

        # Calling merge_clusters to update distance matrix after merging
        distances = merge_clusters(distances, merge_indices)
        current_size -= 1  # Decreasing the current matrix size

    # Assigning labels based on clusters
    labels = np.zeros(len(points), dtype=int)
    for i, cluster in enumerate(clusters):
        labels[cluster] = i

    return labels
```

Agglomerative clustering mechanism and reference to function that changes matrix size. The algorithm searches for the smallest value in the matrix and stores the value of the row and column at the intersection of which this number is located, then the cluster representation of our field changes, cluster j is transferred to cluster i in the array, and then the matrix reduction function is called.

```
def merge_clusters(distances, cluster_indices):

    new_row_column = np.maximum(distances[:, cluster_indices[0]],
distances[:, cluster_indices[1]])
    new_row_column = np.expand_dims(new_row_column, axis=1)
```

```
    # Writing values of the new row into the matrix
    distances[cluster_indices[0], :] = new_row_column.T  # Overwriting the
row
    distances[:, cluster_indices[0]] = new_row_column[:, 0]  # Overwriting
the column

    # Setting a value of 0 at the intersection of the new row and column
    distances[cluster_indices[0], cluster_indices[0]] = 0

    # Deleting the row and column
    distances = np.delete(distances, cluster_indices[1], axis=0)  # Deleting
the row
    distances = np.delete(distances, cluster_indices[1], axis=1)  # Deleting
the column

    return distances
```

A matrix reduction function that removes row j and replaces row i, keeping the largest distance values from i and j

```
def cluster_evaluation(points, labels, method):
    for i in np.unique(labels):
        # Extract points belonging to the current cluster
        cluster_points = points[labels == i]

        # Calculate the center based on the specified method
        if method == 'centroid':
            center = np.mean(cluster_points, axis=0)
        elif method == 'medoid':
            center = np.median(cluster_points, axis=0)

        # Calculate distances from the center to each point in the cluster
        distances = [euclidean_distance(center, point) for point in
cluster_points]

        # Calculate the average distance from the center to points in the
cluster
        avg_distance = np.mean(distances)

        # Check if the average distance meets a certain threshold (500 in
this case)
        if avg_distance <= 500:
            print(f"Cluster {i} is successful. Average {method} distance:
{avg_distance:.2f}")
        else:
            print(f"Cluster {i} is unsuccessful. Average {method} distance:
{avg_distance:.2f}")
```

Function that calculates centroids and medoids, and evaluates the success of the cluster

```
def visualize_clusters(points, labels):
    for i in np.unique(labels):
        # Plotting individual points within each cluster
        cluster_points = points[labels == i]
        plt.scatter(cluster_points[:, 0], cluster_points[:, 1], s=20,
label=f'Cluster {i}')

    for i in np.unique(labels):
```

```
        # Plotting centroids for each cluster
        cluster_points = points[labels == i]
        centroid = np.mean(cluster_points, axis=0)
        plt.scatter(centroid[0], centroid[1], marker='*', color='black',
s=50, label=f'Centroid {i}')

    for i in np.unique(labels):
        # Plotting medoids for each cluster
        cluster_points = points[labels == i]
        medoid_point = medoid(cluster_points)  # Assuming there's a 'medoid'
function available
        plt.scatter(medoid_point[0], medoid_point[1], marker='*',
color='red', s=50, label=f'Medoid {i}')

    plt.title('Agglomerative Clustering')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.show()
```

Calculating centroids and medoids for visualization (here I call a function that calculates medoids, rather than using a ready-made function)

```
def medoid(cluster_points):
    min_sum_distance = np.inf  # Initializing the minimum sum of distances to
infinity
    medoid_point = None  # Initializing the medoid point as None

    # Iterating through each point in the cluster
    for point in cluster_points:
        # Calculating the sum of distances from 'point' to all other points
in the cluster
        sum_distance = np.sum([euclidean_distance(point, other) for other in
cluster_points])

        # Updating the minimum sum of distances and medoid point if a smaller
sum is found
        if sum_distance < min_sum_distance:
            min_sum_distance = sum_distance
            medoid_point = point

    return medoid_point  # Returning the point that minimizes the sum of
distances (the medoid)
```

Function that calculates medoids

```
if __name__ == "__main__":
    number_of_points = int(input("Enter the number of points: "))
    points = generate(number_of_points)

    k = int(input("Enter the number of clusters: "))

    # Performing agglomerative clustering
    labels = agglomerative_clustering(points, k)

    # Evaluation based on medoid distance for each cluster
    cluster_evaluation(points, labels, method='medoid')
```

```
    # Calculating centroids for each cluster and evaluating them
    cluster_evaluation(points, labels, method='centroid')

    # Visualizing the clusters
    visualize_clusters(points, labels)
```

Main function calling operations in the desired order

Results and statistics

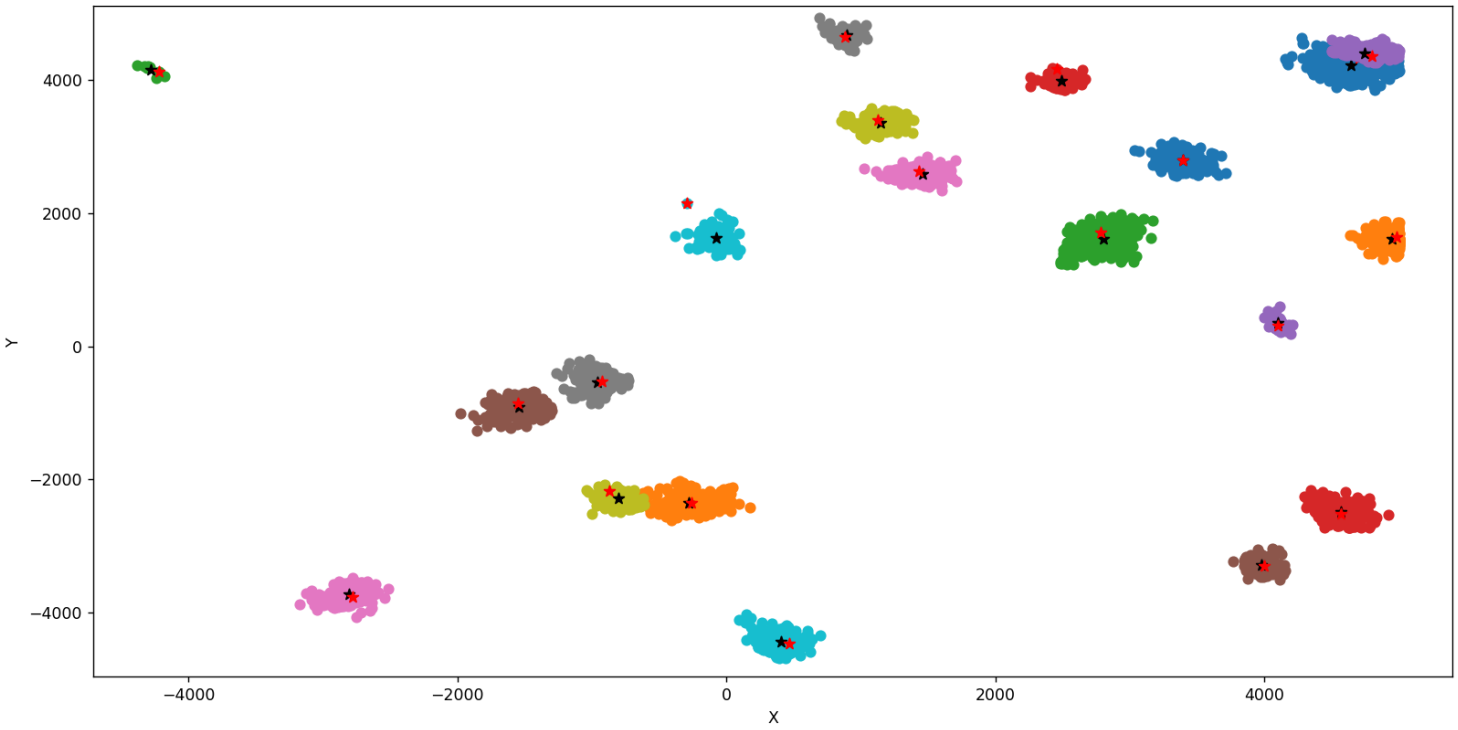
**It makes no sense to provide statistics on program execution by time, since it is quite large, and measuring time will affect the execution time of the program, which will further worsen the result. I can say that during the execution of the program, with each new cycle it speeds up, as soon as half of the cycles are completed, the cycles are executed twice as fast as at the beginning, etc. This is because the matrix is reduced by one row and column (this is a rough calculation, but it shows the approximate speedup).

Below are graphical representations of some solutions

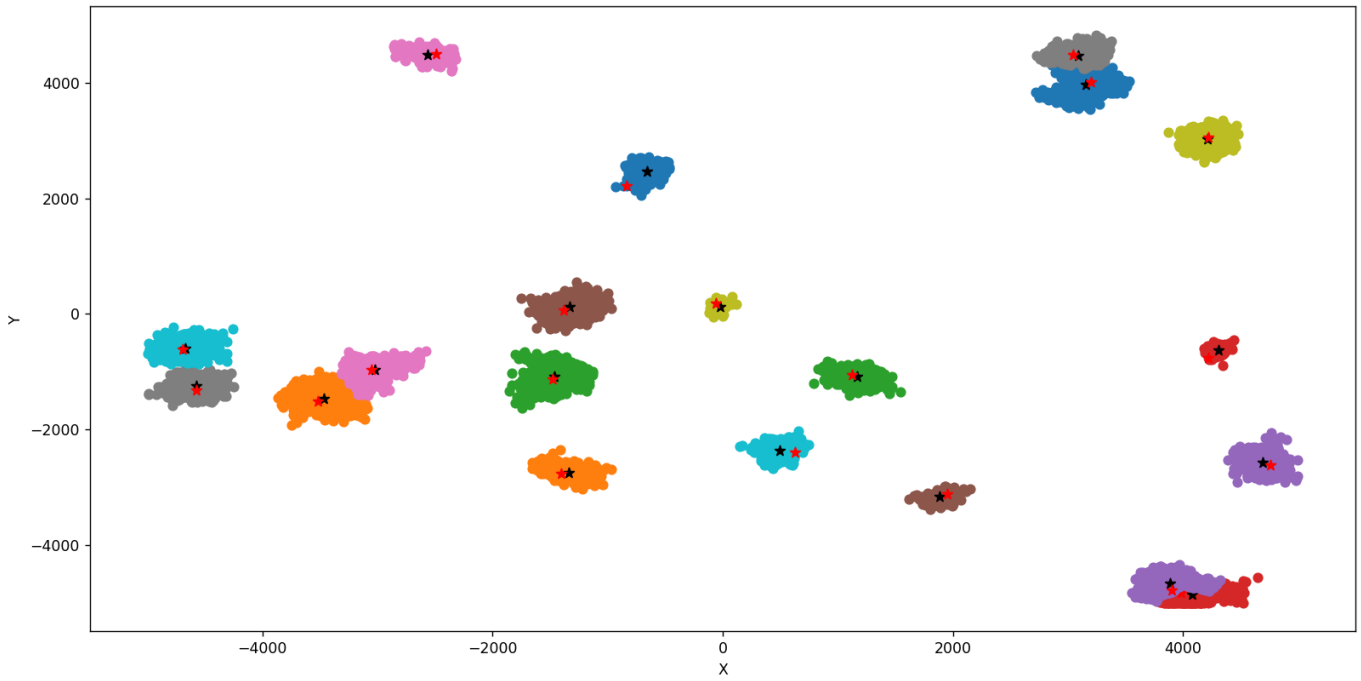
(points/clusters///unsuccessful clusters)

number of starting points = 20

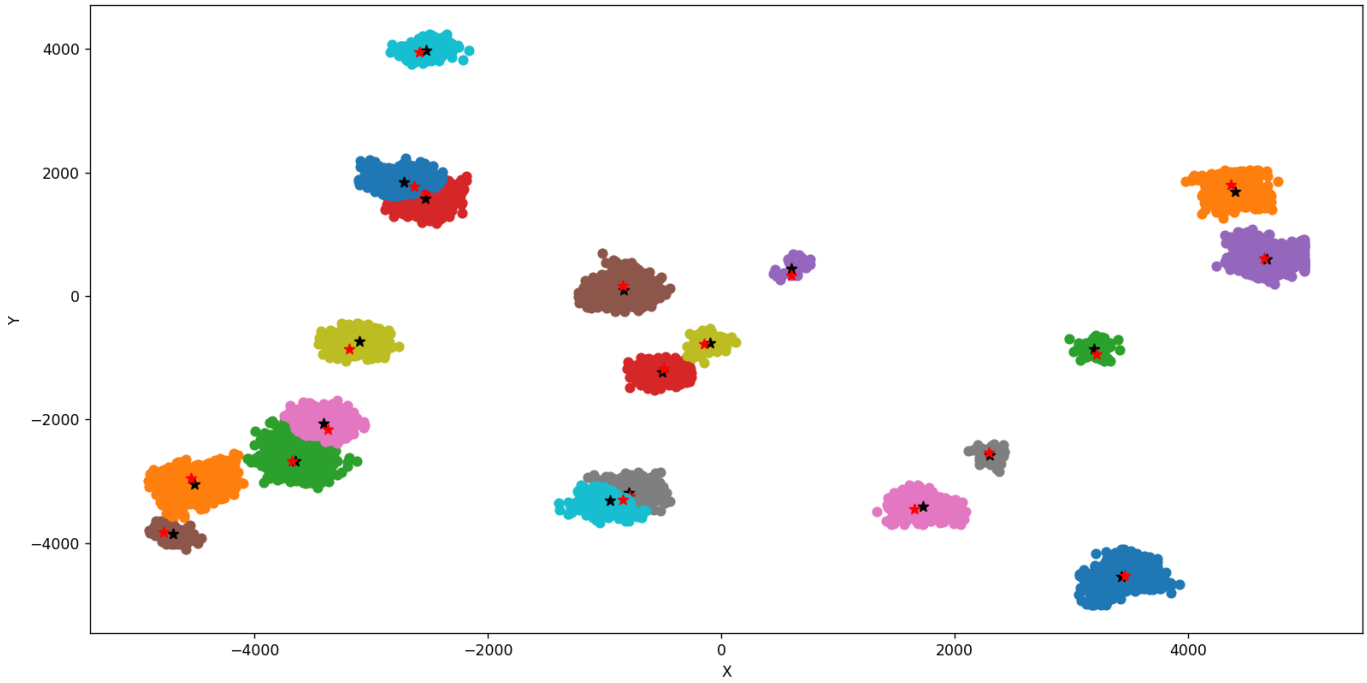
black * - centroids, red * - medoids



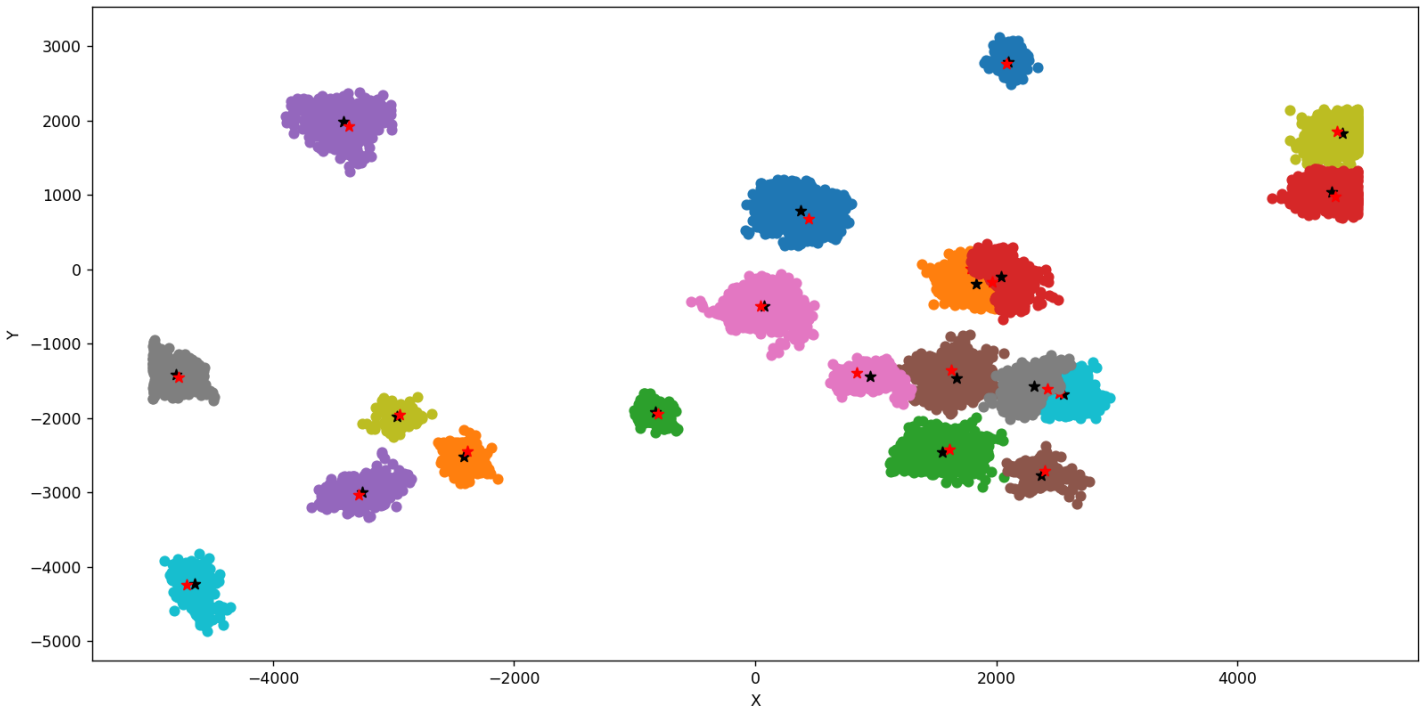
5000 / 20 /// 0



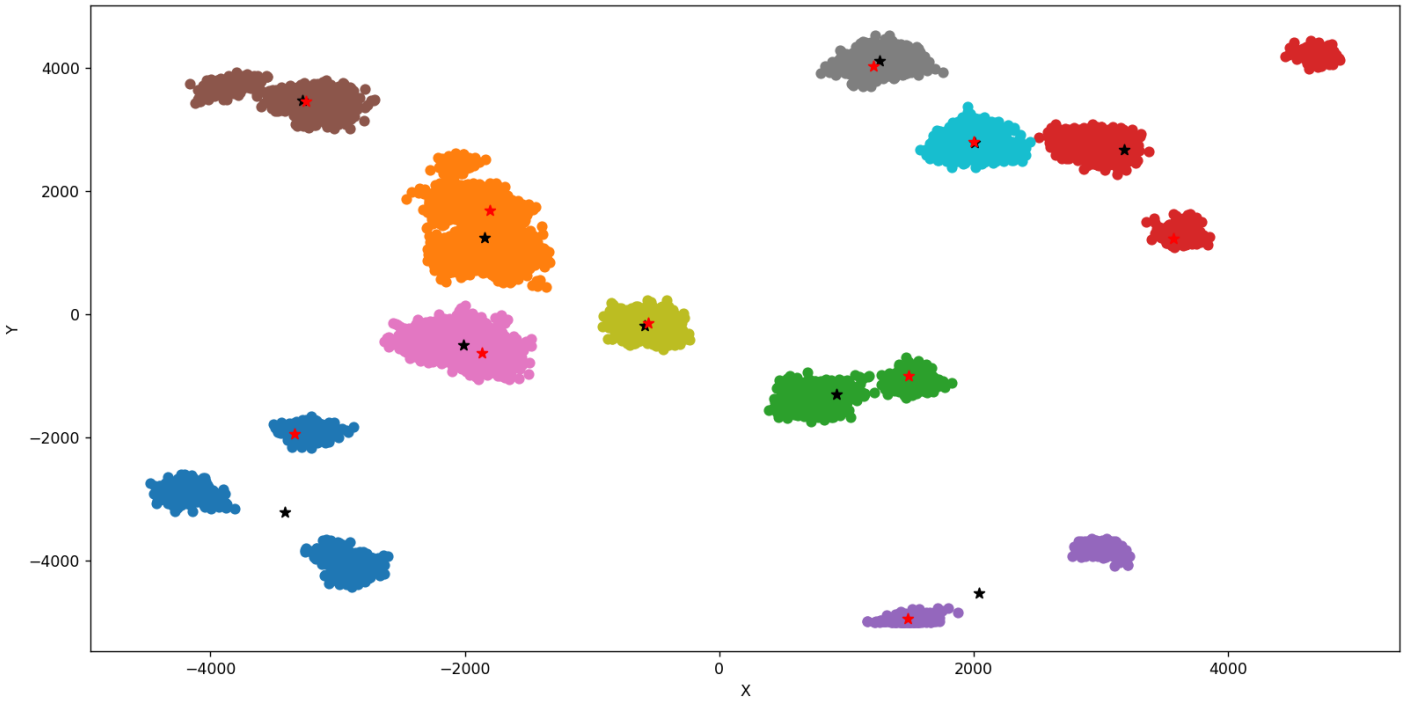
10000 / 20 /// 0



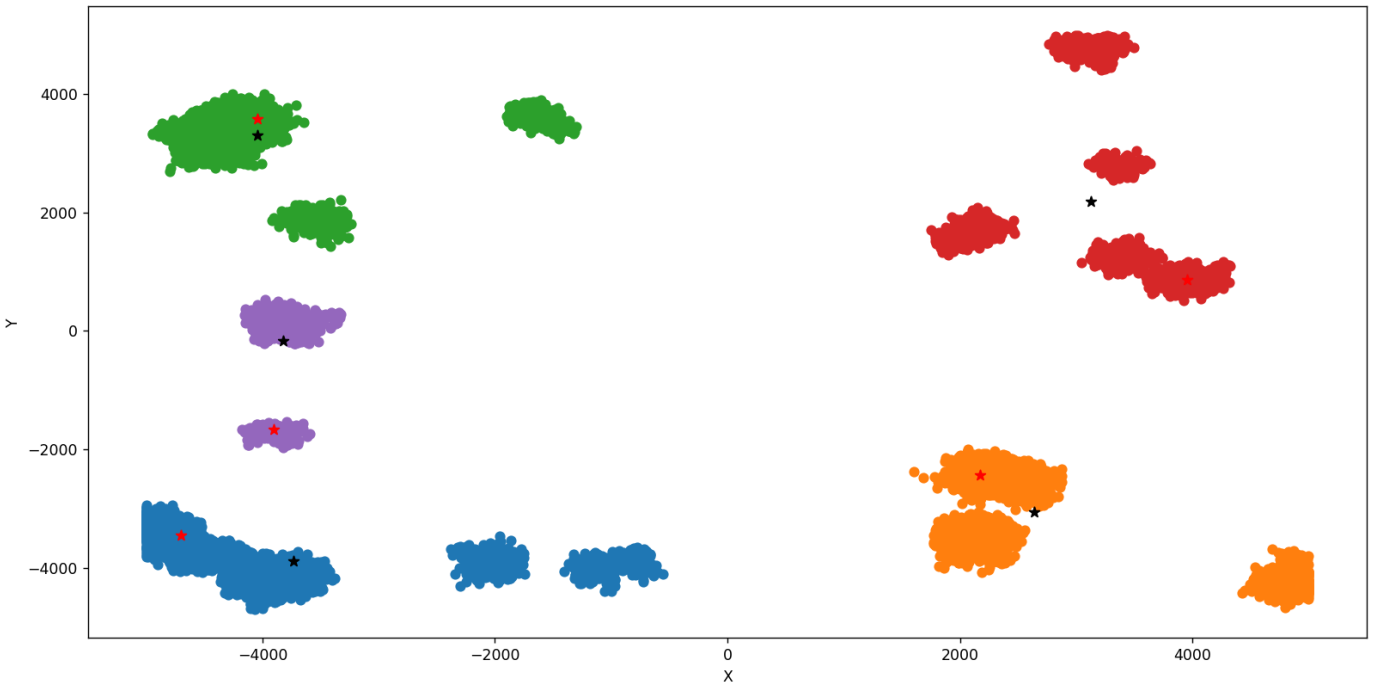
15000 / 20 /// 0



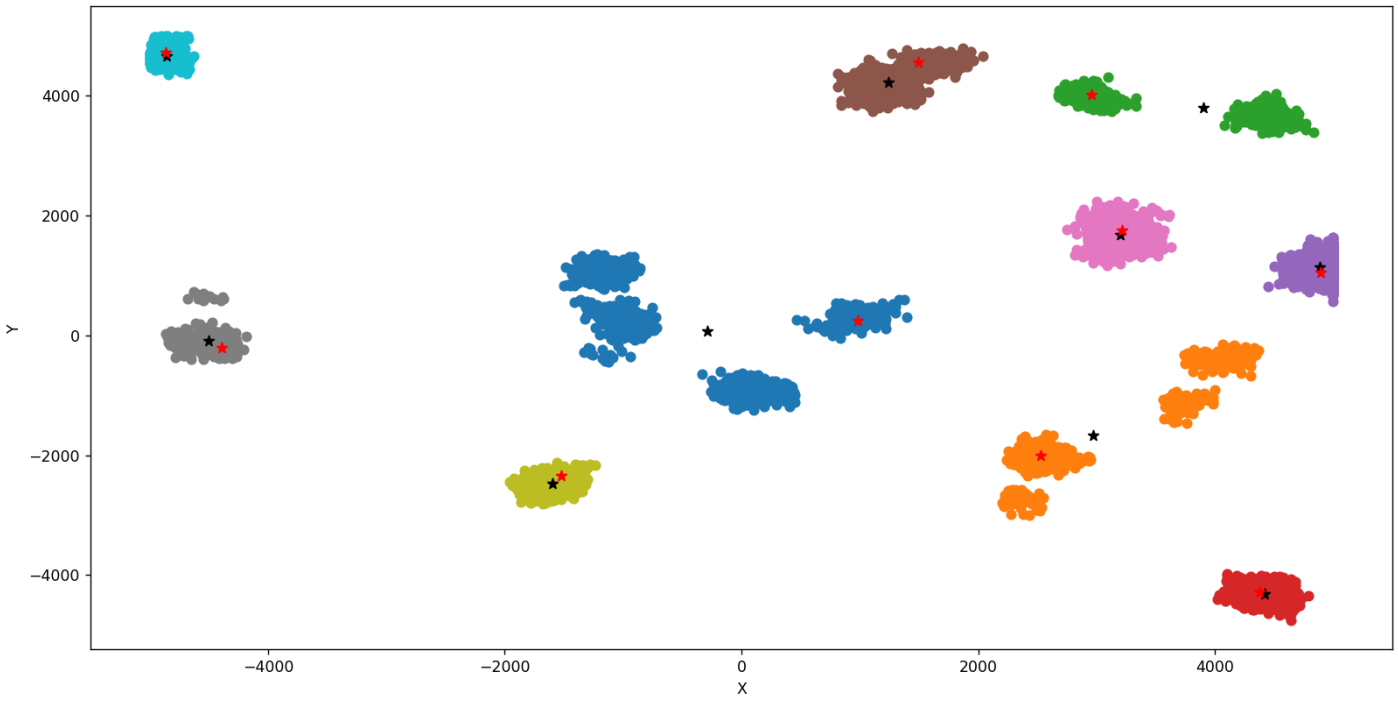
20000 / 20 /// 0



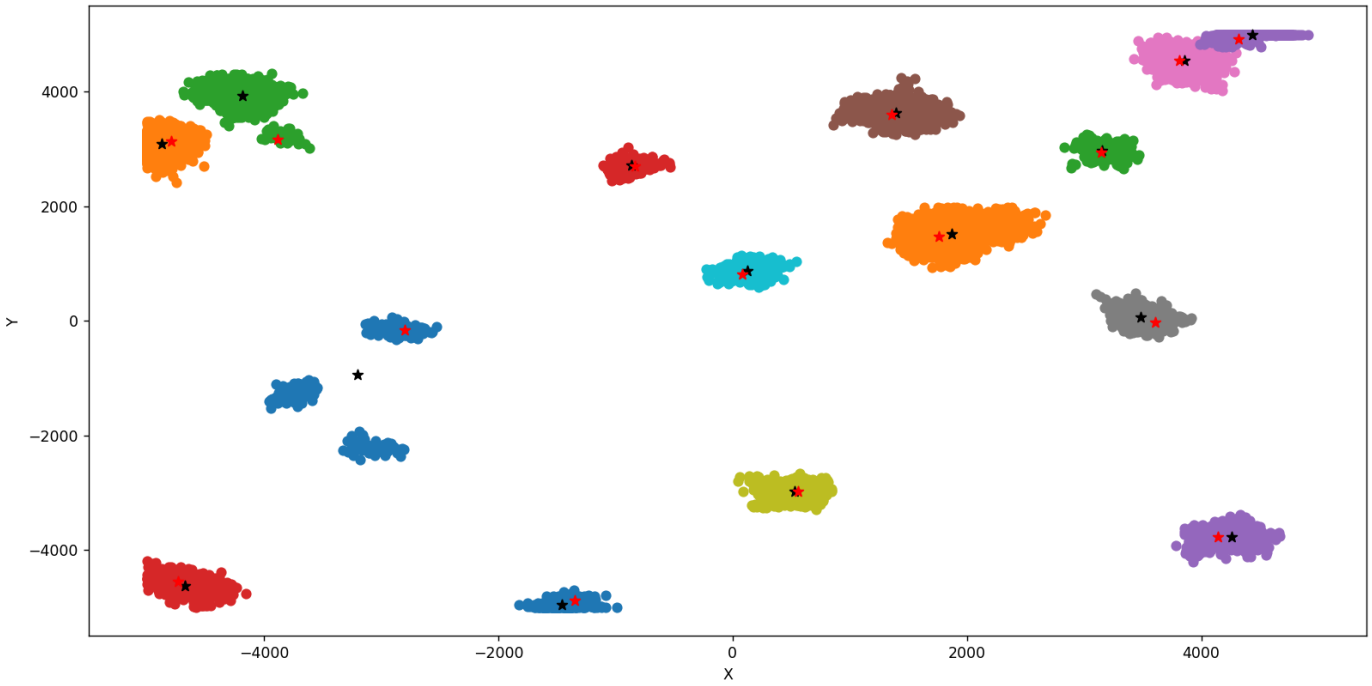
20000 / 10 /// 5



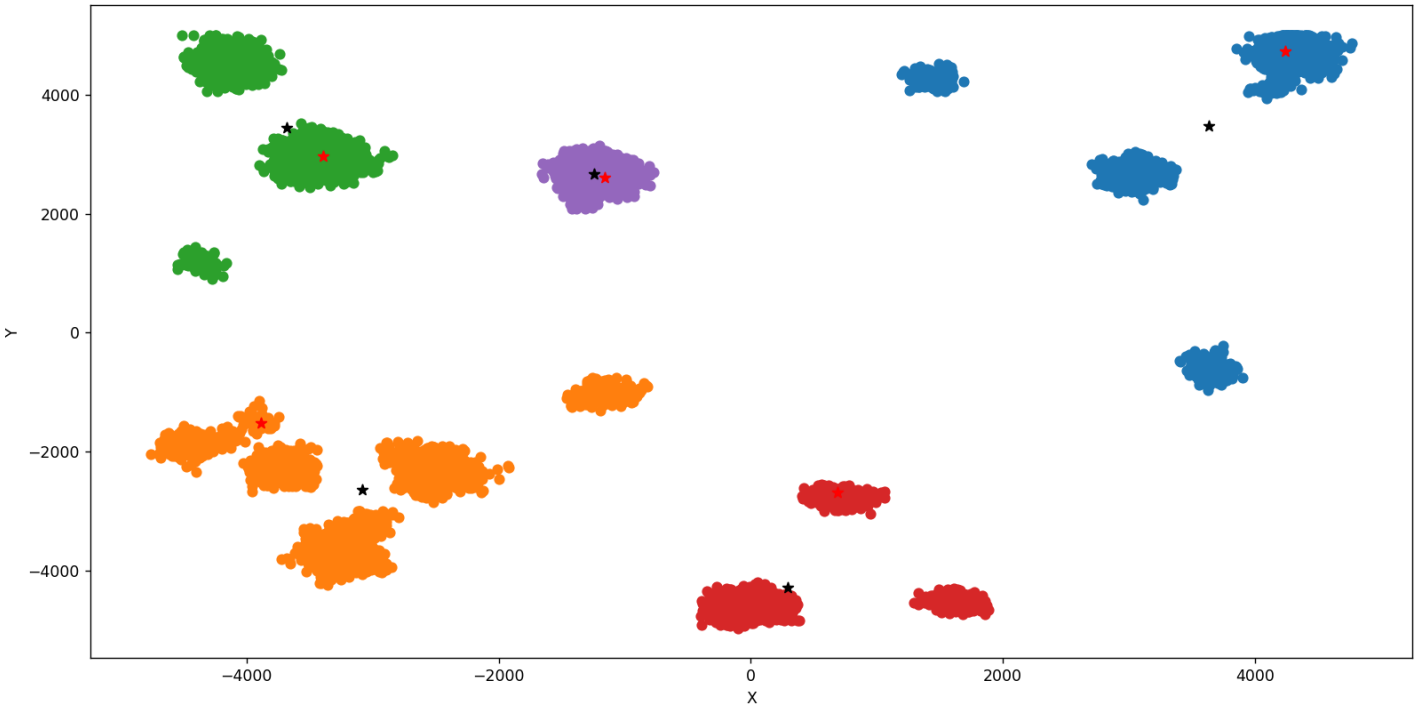
20000 / 5 //// 5



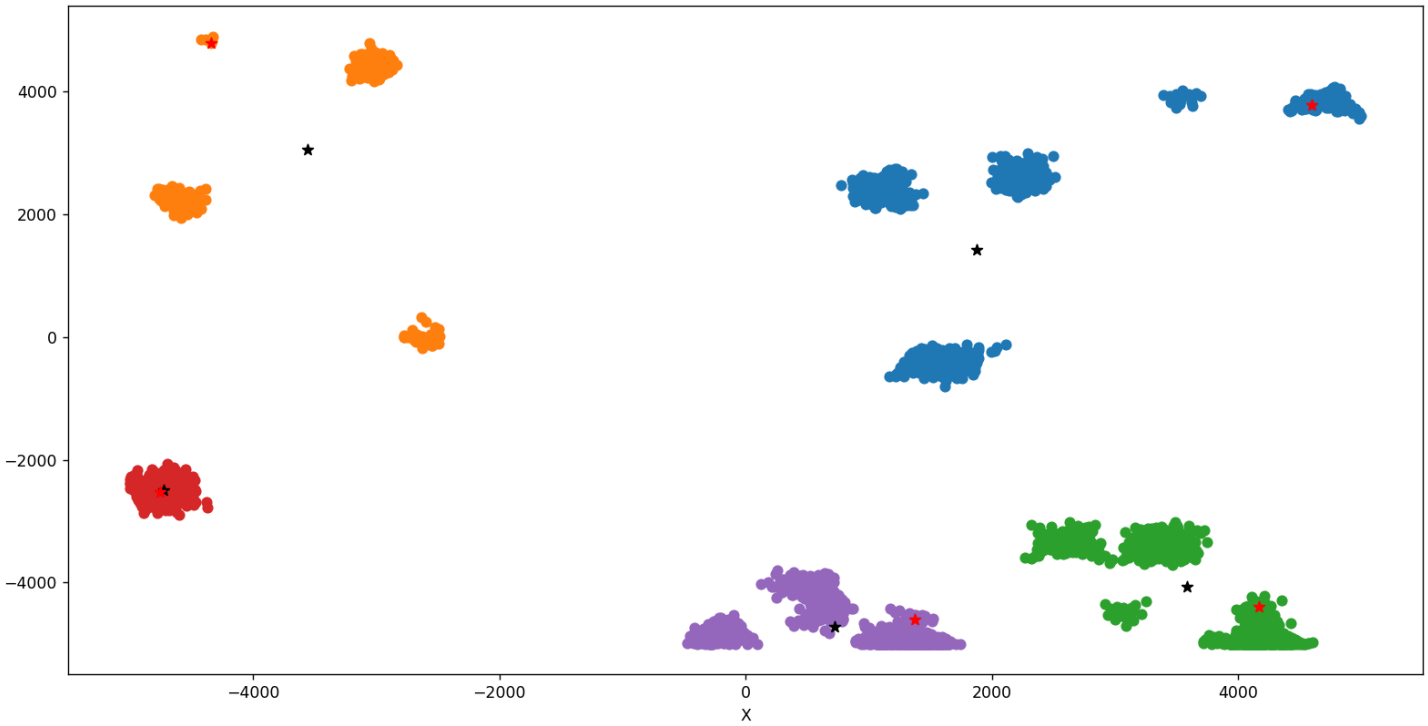
10000 / 10 /// 3



15000 / 15 /// 1

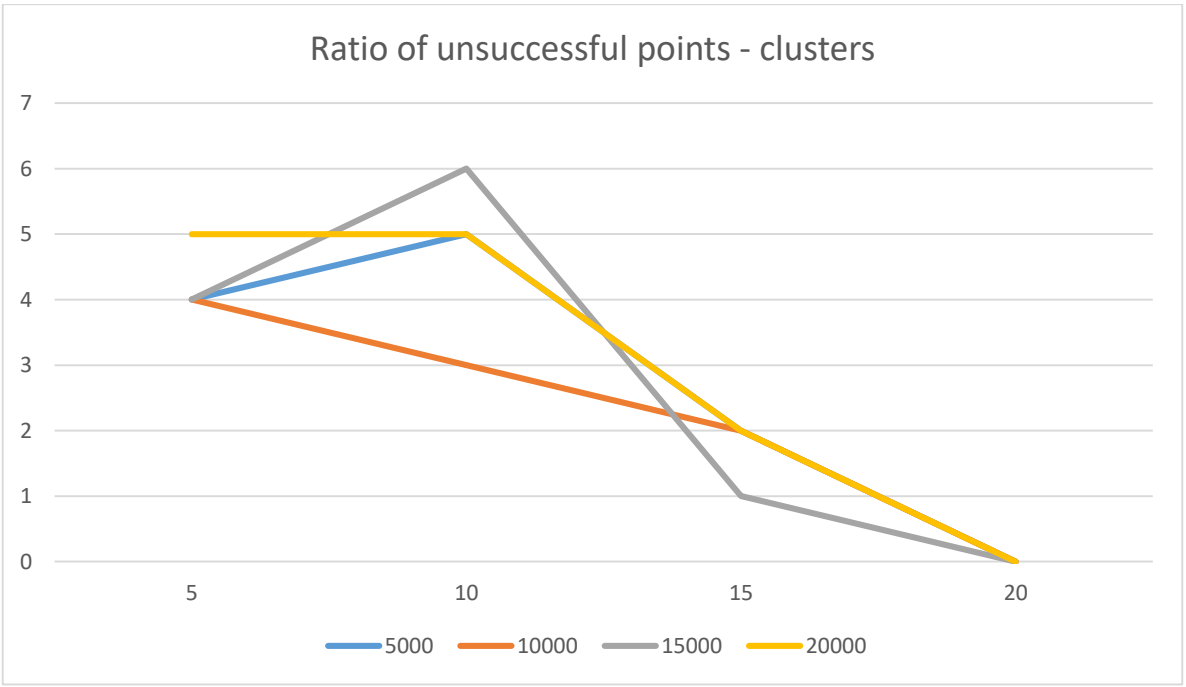


15000 / 5 /// 4



5000 / 5 /// 4

This table shows a comparison of how many clusters were unsuccessful for a certain number of points



points\clusters	5	10	15	20
5000	4	5	2	0
10000	4	3	2	0
15000	4	6	1	0
20000	5	5	2	0

Speculation about my results

***Looking at the statistics, and taking into account the randomness of cluster generation, we can say that the number of unsuccessful clusters mainly depends on the ratio to the starting points, since they are the cores of the generated array of points. The more starting points and the fewer output clusters, the greater the probability of unsuccessful clustering.