Agglomerative Clustering
Project Documentation

Anton Dmitriev
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
`xdmitriev@stuba.sk`

Contents

1. Introduction
This project implements clustering algorithms to divide 2D space into groups of nearby points, called clusters. Initially, points are randomly generated, then the goal is to group these points into clusters with minimum distance from their centers, ensuring clustering efficiency.

2. Task
We have a 2D space that has X and Y dimensions, ranging from −5000 to +5000. Fill this 2D space with 20 points, with each point having a randomly chosen position using the X and Y coordinates. Each point has unique coordinates (i.e., there should not be multiple points in exactly the same location).

After generating 20 random points, generate another 20,000 points, but these points will not be generated completely randomly, but in the following way:

1. Randomly select one of all the points generated so far in 2D space (not just the first 20). If a point is too close to the edge, reduce the corresponding interval given in the next two steps.
2. Generate a random number `X_offset` in the interval from −100 to +100
3. Generate a random number `Y_offset` in the interval from −100 to +100
4. Add a new point to the 2D space that will have the coordinates as the randomly selected point in step 1, with these coordinates shifted by `X_offset` and `Y_offset`

Task is to program a clusterer for 2D space that analyzes the 2D space with all its points and divides this space into k clusters. Implement different versions of the clusterer, specifically with the following algorithms:

- Agglomerative clustering, where the center is a centroid
- Agglomerative clustering, where the centroid is a medoid

Evaluate the success/error rate of your clusterer. Consider a successful clusterer to be one in which no cluster has an average distance of points from the center greater than 500.

3. Algorithm
The overall process of agglomerative clustering involves starting with each point as an individual cluster and iteratively merging the closest clusters until the stopping condition is met. The algorithm ensures that clusters are formed such that the distance between points and their respective cluster centers is minimized.

3.1. Point Generation

3.1.1. Initial Points
The initial set of points is generated randomly within the specified bounds, ensuring that each point has unique coordinates.

```python
# generation.py
def generate_initial_points():
    points = set()

    while len(points) < initial_points_num:
        x = random.randint(lower_bound, upper_bound)
        y = random.randint(lower_bound, upper_bound)
        points.add((x, y))

    return np.array(list(points))
```

3.1.2. Additional Points

Additional points are generated by selecting existing points and adding random offsets within a specified range.

```python
# generation.py
def generate_additional_points(initial_points, additional_points_num):
    points = initial_points.tolist()

    for _ in range(additional_points_num):
        it = random.randint(0, len(points) - 1)
        base_point = points[it]

        x_offset_lower = max(-offset_range, lower_bound - base_point[0])
        x_offset_upper = min(offset_range, upper_bound - base_point[0])
        y_offset_lower = max(-offset_range, lower_bound - base_point[1])
        y_offset_upper = min(offset_range, upper_bound - base_point[1])

        x_offset = random.randint(int(x_offset_lower), int(x_offset_upper))
        y_offset = random.randint(int(y_offset_lower), int(y_offset_upper))

        new_point = [base_point[0] + x_offset, base_point[1] + y_offset]
        points.append(new_point)

    return np.array(points)
```

3.2. Clustering Steps

The following explanation of algorithm steps is exactly identical for both `centroids.py` file and `medoids.py` file.

1. **Initialization:** Each point is treated as its own cluster.

```python
num_points = len(points)
clusters = {i: [i] for i in range(num_points)}
centroids = {i: points[i].astype(float) for i in range(num_points)}
active_clusters = list(clusters.keys())
```

2. **Iterations:** Algorithm continues to merge clusters until only one cluster remains or another stopping condition is met

```python
iteration = 0

while len(active_clusters) > 1:
  iteration += 1
```

2. **Distance Calculation:** Calculate the distances between all current clusters.

```python
tree = KDTree(centroid_coords)

k = min(10, len(active_clusters))
distances, indices = tree.query(centroid_coords, k=k)
```

3. **Finding Pair:** To merge clusters, algorithm identifies closest pair of clusters based on the minimum distance between them.

```python
min_distance = np.inf
min_pair = None

for idx, (dist_list, ind_list) in enumerate(zip(distances, indices)):
  i = index_to_id[idx]

  for dist, neighbor_idx in zip(dist_list[1:], ind_list[1:]):
    if neighbor_idx != idx:
      j = index_to_id[neighbor_idx]
      break
    else:
      continue

    if dist < min_distance:
      min_distance = dist
      min_pair = (i, j)
```

3. **Cluster Merging:** Once the pair is identified, algorithm merges clusters and updates list of active clusters.

```python
i, j = min_pair

print(f"\rIteration {iteration}, Unified clusters {i} and {j}, Distance
{min_distance:.2f}, Time: {iteration_time:.2f}s",end="")

clusters[i].extend(clusters[j])
active_clusters.remove(j)
del clusters[j]
del centroids[j]
```

4. **Center Update:** Recalculate the center of the new merged cluster.

3.3. Termination Condition

The algorithm stops under any of the following conditions:

- **Condition 1:** No valid pairs are found.

```python
if min_pair is None:
  print(f"\rIteration {iteration}, No valid pairs found", end="")
  break
```

- **Condition 2:** The minimum distance between clusters exceeds `max_distance`.

```python
if min_distance > max_distance:
  print(f"\rIteration {iteration}, Min Distance ({min_distance:.2f}) > Max Distance
({max_distance})", end="")
  break
```

- **Condition 3:** Only one cluster remains.

3.4. Center Differences

3.4.1. Centroid

The centroid of a cluster is the arithmetic mean of all points in the cluster:

```
# centroids.py
centroids[i] = np.mean(points[clusters[i]], axis=0)
```

3.4.2. Medoid

The medoid is an actual point within the cluster that has the minimum total distance to all other points in the cluster:

```
# medoids.py
cluster_point_indices = clusters[i]
cluster_points = points[cluster_point_indices]

distances_matrix = np.linalg.norm(cluster_points[:, np.newaxis] -
cluster_points[np.newaxis, :], axis=2)
sum_distances = np.sum(distances_matrix, axis=1)
min_index = np.argmin(sum_distances)
medoids[i] = cluster_points[min_index]
```

3.5. Clusters Number

Instead of specifying a hard number of clusters `k`, I introduced a configurable parameter `max_distance` to determine when to stop the clustering process. The algorithm stops merging clusters when the minimum distance between any pair of clusters exceeds `max_distance`.

```
if min_distance > max_distance:
    print(f"\rIteration {iteration}, Min Distance ({min_distance:.2f}) > Max Distance
({max_distance})", end="")
    break
```

3.5.1. Optimal Value

To find the ideal value, I conducted experiments by varying its value and observing the success rate of the clustering.

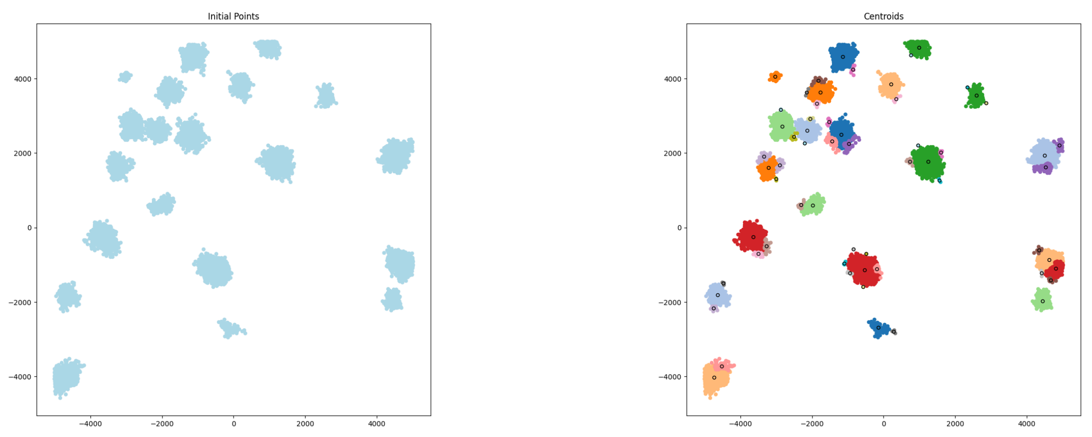

Figure 1: Experiment with max_distance = 300

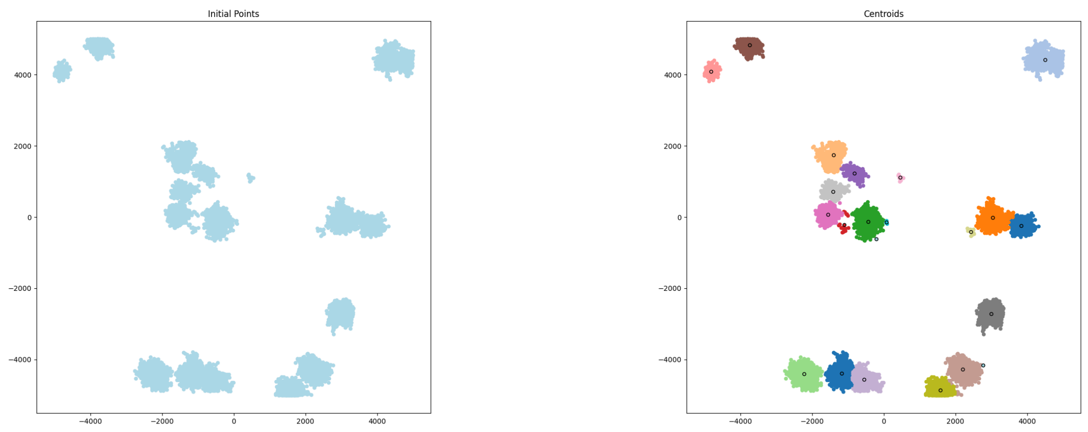

Figure 2: Experiment with max_distance = 500

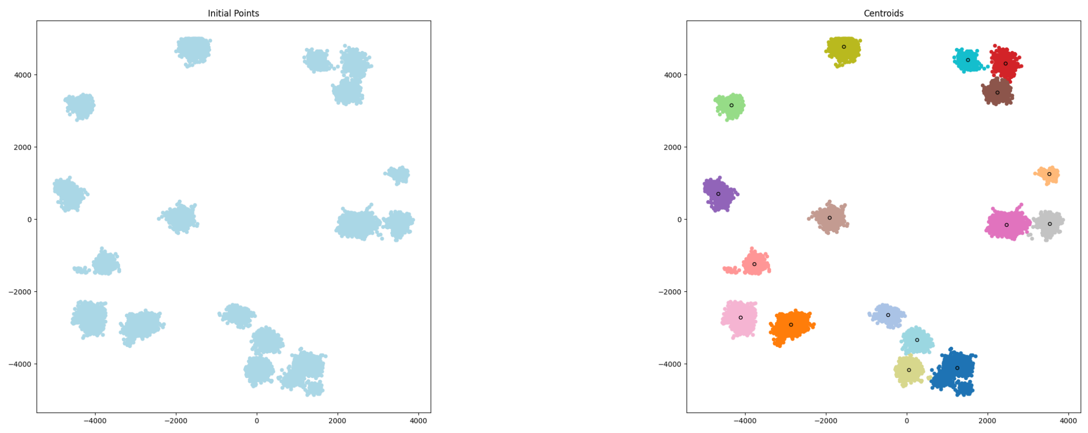

Figure 3: Experiment with max_distance = 800

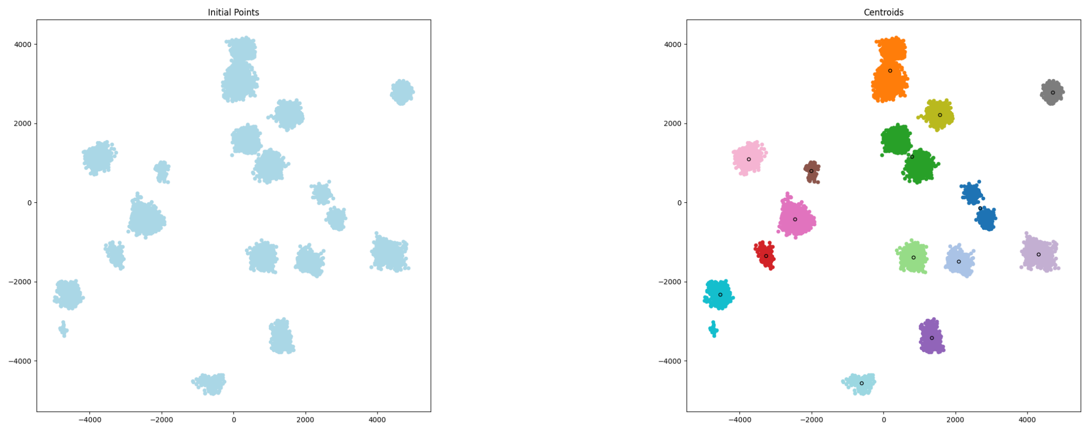

Figure 4: Experiment with max_distance = 1000

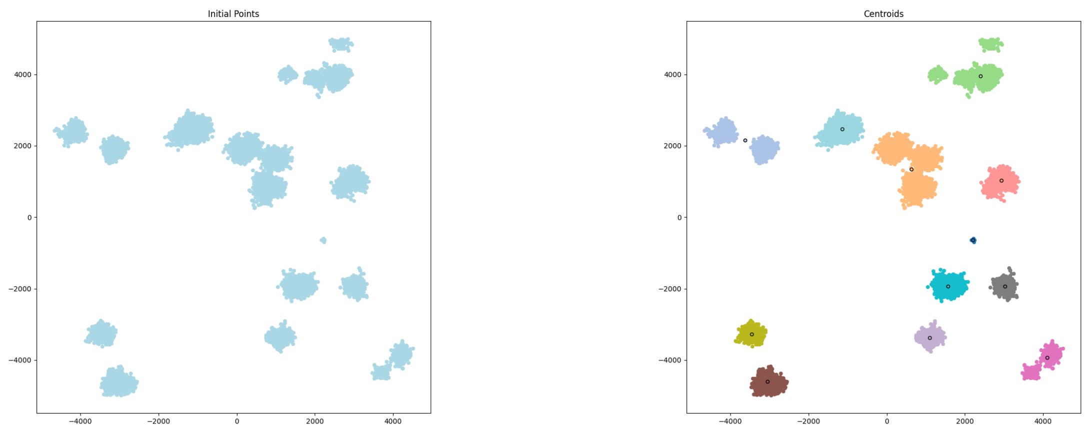

Figure 5: Experiment with max_distance = 1300

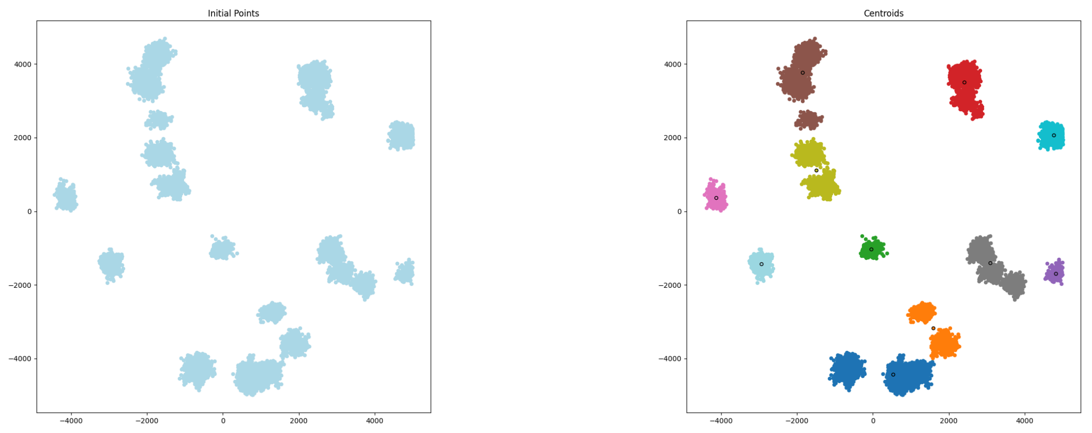

Figure 6: Experiment with max_distance = 1500

Maximum Distance	Number of Clusters	Success Rate
300	61	100%
500	22	100%
800	17	100%
1000	15	93.34%
1300	12	83.33%
1500	11	63.64%

Table 1: Experimental Results

By analyzing the results, I identified that a maximum distance of 800 provides 100% success rate, meeting the task's criteria and leaving sufficient number of output clusters.

4. Optimization

4.1. Distance Matrix

Initially, I attempted to optimize the algorithm by using a 2-dimensional distance matrix to store the distances between all pairs of points. This approach has quadratic time complexity `O(n²)` for both filling the matrix and updating it during each iteration. With 20,000 points, this method became computationally impractical.

4.2. K-D Tree

To overcome the performance bottleneck, I switched to using a K-D Tree data structure for efficient nearest neighbor searches. The K-D Tree significantly speed up the computation by reducing the time complexity of querying nearest clusters to approximately `O(n log n)`.

With the K-D Tree implementation, the algorithm's execution time was reduced to approximately 1,200 seconds (20 minutes) on standard `CPython`.

4.2.1. Overview

A K-D Tree (k-dimensional tree) is a binary search tree that stores data points in k-dimensional space. It is a space-partitioning data structure that organizes points to allow efficient queries, such as nearest neighbor searches and range searches.

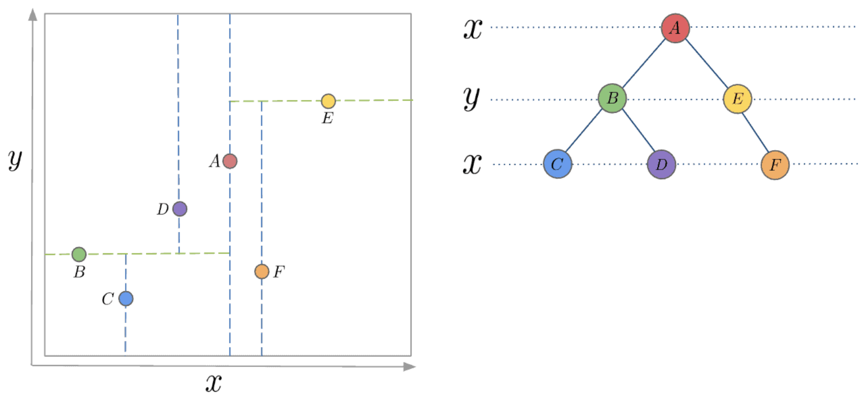

Figure 7: K-D Tree

4.2.2. Tree Structure

In a K-D Tree, each node represents a hyperplane that divides the space into two parts. The tree alternates between splitting dimensions at each level, for example, splitting along the X-axis at one level and the Y-axis at the next. This recursive partitioning results in a balanced tree structure that significantly speeds up search operations.

```python
def kd_tree(points, depth=0):

    # Select the axis based on depth so that axis cycles through all valid values
    k = len(points[0])
    axis = depth % k

    points.sort(key=lambda x: x[axis])

    # Choose median as pivot element
    median_index = len(points) // 2

    # Create node and construct subtree
    return Node(
        point=points[median_index],
        left=kd_tree(points[:median_index], depth + 1),
        right=kd_tree(points[median_index + 1:], depth + 1)
    )
```

4.2.3. Search

When performing a nearest neighbor search, the algorithm traverses the tree to find the region containing the query point. It then backtracks, exploring other branches of the tree only if they could contain a point closer than the current best. Branches that cannot contain closer points are pruned, reducing unnecessary computations. This pruning of branches that cannot contain closer points greatly reduces the number of distance calculations required.

4.2.4. Application

In the context of the clustering algorithm, the K-D Tree is rebuilt at each iteration to reflect the updated cluster centers. This dynamic update ensures that the nearest neighbor search always uses relevant data during the run of the algorithm. By using K-D Tree, the algorithm efficiently determines nearest clusters without computing all pairwise distances, which significantly improves performance.

4.3. Improvement

The switch to K-D Tree provided significant performance gains:

- **Time Complexity:** Redused from `O(n²)` to approximately `O(n log n)`.
- **Execution Time:** Reduced from an unaffordable large to an average of 1,200 seconds (20 minutes).
- **Resource Utilization:** Algorithm can be executed using standard `CPython` without relying on `PyPy`, allowing compatibility with essential graphical visualization libraries.

5. Evaluation

The evaluation of the clustering algorithm is crucial to determine whether it meets the project's success criteria. The primary requirement is that no cluster should have an average distance of its points from the center greater than 500.

Evaluation process involves calculating the average distance of points from the cluster center for each cluster. This is achieved by computing the Euclidean distance between each point in the cluster and the cluster's center (centroid or medoid) and then averaging these distances.

```python
# evaluation.py
def evaluate_clustering(points, clusters, centers):
    total_clusters = len(clusters)
    successful_clusters = 0
    cluster_average_distances = {}

    for cluster_id, point_indices in clusters.items():
        cluster_points = points[point_indices]
        center = centers[cluster_id]
        distances = np.linalg.norm(cluster_points - center, axis=1)
        average_distance = np.mean(distances)
        cluster_average_distances[cluster_id] = average_distance

        if average_distance <= max_average_distance:
            successful_clusters += 1

    success_percentage = (successful_clusters / total_clusters) * 100

    print(f"Success rate {success_percentage:.2f}%")
```

6. Visualization

The following figures show a visual representation of the clustering algorithm using the optimal parameter `max_distance = 800`. This parameter was determined to ensure 100% success in ensuring that the average distance between points within each cluster remains below a given threshold. The visualization illustrates the original point distribution, the resulting clusters using centroids, and the resulting clusters using medoids.

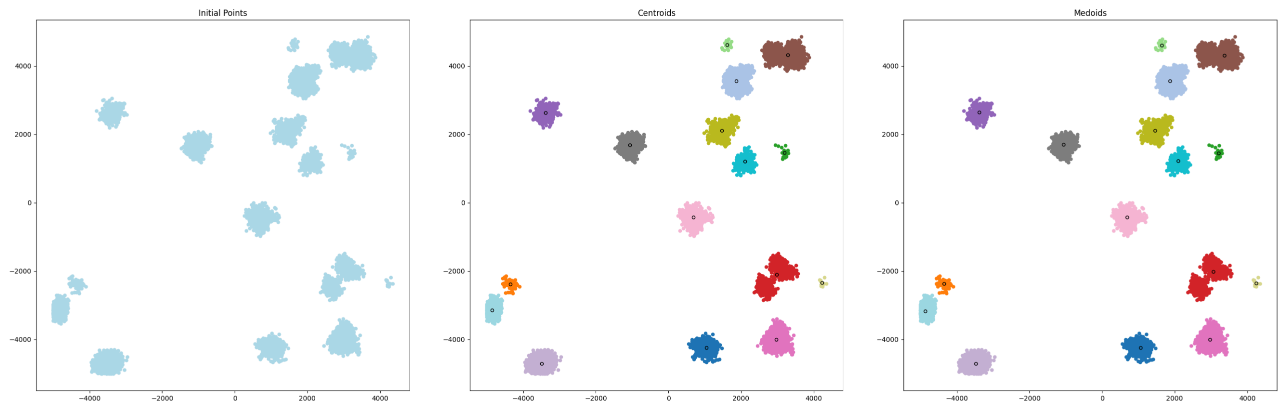

Figure 8: Algorithm Run №1

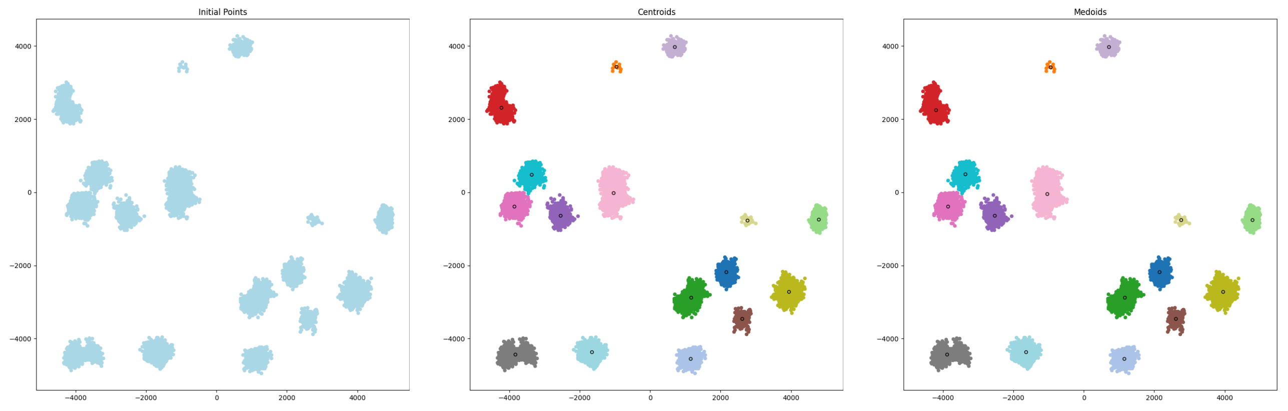

Figure 9: Algorithm Run №2

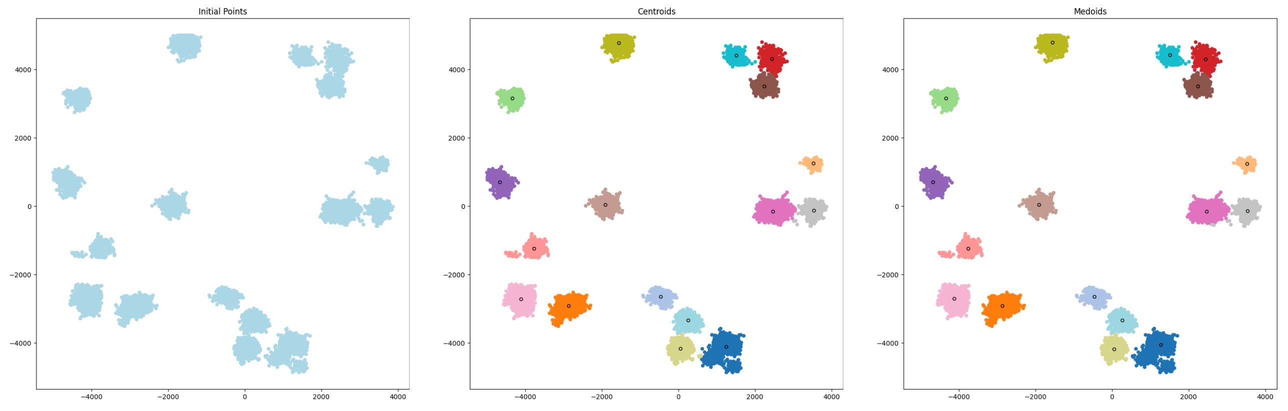

Figure 10: Algorithm Run №3

Each set of results consistently achieved a 100% success rate, confirming the reliability and effectiveness of the algorithm when configured with the optimal parameter.

7. Conclusion

In this project, agglomerative clustering algorithms using both centroids and medoids as cluster centers were successfully implemented. By optimizing the algorithm using K-D Tree data structures and introducing a tunable stopping parameter `max_distance`, efficient clustering was achieved, satisfying the specified success criteria and making it possible to process large data sets in reasonable time.