Final Report of Soft Computing

OPTIMIZING THE ASYMMETRIC CLUSTERED AND NON-CLUSTERED TRAVELING SALESMAN PROBLEM WITH GENETIC AND ENHANCED GENETIC ALGORITHMS: A CASE STUDY OF SHELL GAS STATION SUPERVISION IN SURABAYA

Prepared By :

Naura Jasmine Azzahra	5026211005
Anak Agung Istri Istadewanti	5026211143
Mohammed Fachry Dwi Handoko	5025201159

DEPARTEMEN SISTEM INFORMASI

FAKULTAS TEKNOLOGI ELEKTRO DAN INFORMATIKA CERDAS

INSTITUT TEKNOLOGI SEPULUH NOPEMBER SURABAYA

2024

Content

I. Background

Energy is the primary driver of all human activities and plays a crucial role in various economic, social, and environmental aspects (Tzanakis et al., 2012). Fuel Oil is one of the main sources of energy that supports this, becoming an essential commodity that is highly needed in the daily lives of the community (Sitio & Nadiyanti, 2022). According to the Republic of Indonesia Law No. 22 of 2001 concerning Oil and Natural Gas, the government has the authority to oversee activities in the energy sector, including the distribution of fuel oil conducted by various Gas Stations (SPBU).

Fraud in fuel dispensing at Gas Stations is a common complaint frequently encountered by consumers (Wibawa et al., 2019). At the regional level, this oversight is carried out by the Department of Industry and Trade (Dinas Perindustrian dan Perdagangan or also known as Disperindag), which is responsible for ensuring adherence to all distribution standards, safety regulations, and pricing policies. As part of this supervision, routine nozzle checks are conducted to verify that fuel dispensers are accurately calibrated. This was mandated by Law No. 2 of 1981 concerning Legal Metrology, which emphasizes the importance of using valid measuring instruments in transactions involving fuel oil (Disperindag Sigi, 2024). These nozzle checks play a crucial role in preventing potential fraud, as they confirm that dispensers provide the correct fuel amounts, ultimately guaranteeing that the public receives fair, quality services that align with applicable legal standards.

Research by Aditya (2024) presents a solution to the Asymmetric Clustered Traveling Salesman Problem (ACTSP) in the case of monitoring 70 Pertamina Gas Stations in Surabaya, divided into 8 clusters using the Equal-Size Spectral Clustering method. By employing Ant Colony Optimization (ACO) optimized through Opposition-Based Learning (OBL) techniques, this study successfully identified optimal routes with a minimal total distance of 249.3 km. The optimal parameter values, namely beta 5 and 400 iterations, demonstrate the effectiveness of modifying ACO with OBL in addressing ACTSP optimization, particularly in complex distributions.

This project aims to address a similar issue as the previous study by Aditya (2024), which dealt with ACTSP at Pertamina Gas Stations in Surabaya; however, it will focus on supervising Shell Gas Stations in Surabaya. The algorithm used will differ, specifically utilizing Genetic Algorithm (GA), as GA excels in handling dynamic data flexibility. Additionally, GA has the capability to avoid local solution traps that can hinder ACO, especially in complex and asymmetric distribution problems such as ACTSP.

The application of the ACTSP model with GA is expected to provide optimal solutions for planning supervision routes for Shell Gas Stations in Surabaya. By considering time capacity and travel distance factors, this method aims not only to minimize distance and travel time but also to enhance overall supervisory efficiency. Distance data obtained from Google Maps will be utilized to ensure route accuracy. Furthermore, this research is anticipated to contribute significantly to the management of fuel distribution oversight while also serving as a foundation for implementing other optimization methods in the future that can help improve safety standards and public services more efficiently and effectively.

II. Problem Description

The distribution of fuel oil in Indonesia, particularly in Surabaya, requires stringent oversight to ensure compliance with all distribution standards, safety regulations, and pricing policies. This oversight is conducted by the Disperindag through periodic visits to Gas Stations, with a primary focus on conducting nozzle checks to verify that fuel dispensers are accurately calibrated. These nozzle checks, mandated by Law No. 2 of 1981 concerning Legal Metrology, ensure that fuel dispensers provide the correct amount of fuel and protect consumers from discrepancies. However, despite its importance, the process is often hindered by several constraints:

1. **Route Efficiency**: With 15 Shell Gas Stations scattered throughout Surabaya, optimal route planning becomes essential. Officers need to visit all Gas Stations within a limited timeframe and with constrained resources. Without an efficient route, the time and costs incurred for nozzle checks can increase significantly.
2. **Resource Limitations**: The officers assigned for nozzle checks are limited in number and time. Therefore, it is vital to plan a route that allows them to visit all Gas Stations and perform nozzle checks in the most effective manner. Inefficient use of resources could lead to inadequate oversight at some Gas Stations, potentially compromising the accuracy of fuel measurements.
3. **Potential Fraud**: Without adequate oversight, there is a risk of fraudulent activities that could harm the public, such as deviations in fuel measurement accuracy. Routine nozzle checks as part of systematic monitoring are necessary to prevent such issues and ensure that dispensers adhere to precise measurement standards.

Therefore, an effective solution is required to plan travel routes for officers using the ACTSP model and GA. With this approach, it is hoped that the supervision of fuel distribution through nozzle checks can be conducted more efficiently. This ensures that measurement accuracy and quality standards are upheld while minimizing the potential for fraud and protecting consumer rights.

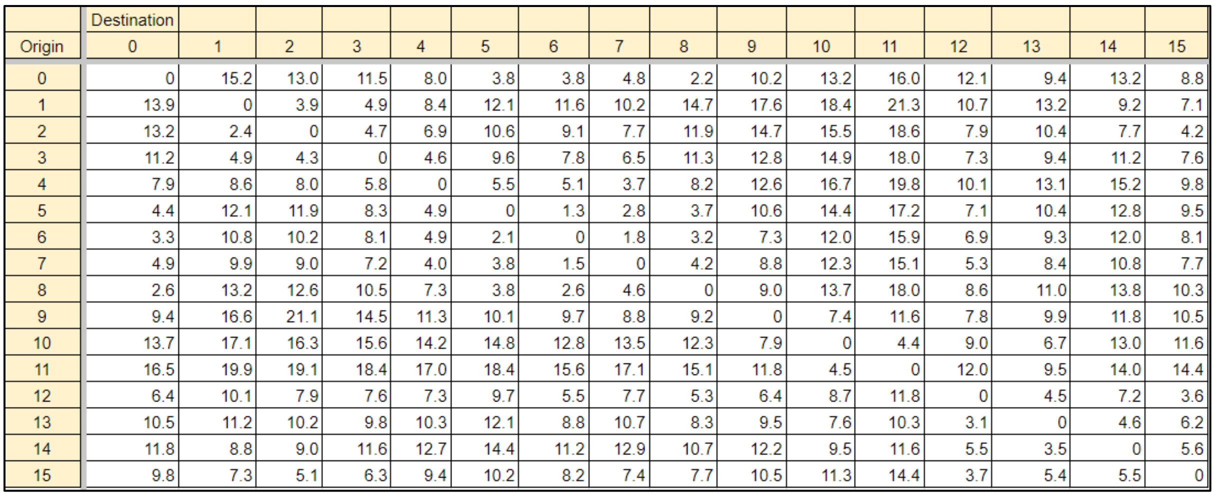
III. Data

In this project, several data sets were collected. The first set involved gathering information on 15 Shell Gas Stations in Surabaya, along with the Disperindag Jatim office, using their latitude and longitude coordinates from Google Maps. This data was used for route visualization.

No.	Shell Gas Station (SPBU)	Lat	Long
0	Disperindag Jatim	-7.3354472	112.734173
1	SPBU Shell Kenjeran	-7.2503249	112.7874995
2	SPBU Shell Kalijudan	-7.2619571	112.7741993
3	SPBU Shell Kertajaya	-7.2799463	112.789015
4	SPBU Shell MERR	-7.3126372	112.7808488
5	SPBU Shell Tenggilis	-7.3204815	112.755639
6	SPBU Shell Margorejo	-7.3174111	112.7504313
7	SPBU Shell Prapen	-7.3088436	112.7601212
8	SPBU Shell Jemursari	-7.3270382	112.732417
9	SPBU Shell Mastrip	-7.3243952	112.7089127

10	SPBU Shell Mayjend. Jonosewojo	-7.2916648	112.6756172
11	SPBU Shell Citraland Surabaya	-7.2830254	112.6505561
12	SPBU Shell Diponegoro	-7.2837719	112.7334513
13	SPBU Shell Banyu Urip	-7.2678209	112.7124069
14	SPBU Shell Dupak	-7.2456583	112.7213161
15	SPBU Shell Pemuda Central	-7.2657828	112.7483547

Furthermore, the main data for optimization consisted of gathering the round-trip distances between each destination in kilometers, using Google Maps.



	Destination															
Origin	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	15.2	13.0	11.5	8.0	3.8	3.8	4.8	2.2	10.2	13.2	16.0	12.1	9.4	13.2	8.8
1	13.9	0	3.9	4.9	8.4	12.1	11.6	10.2	14.7	17.6	18.4	21.3	10.7	13.2	9.2	7.1
2	13.2	2.4	0	4.7	6.9	10.6	9.1	7.7	11.9	14.7	15.5	18.6	7.9	10.4	7.7	4.2
3	11.2	4.9	4.3	0	4.6	9.6	7.8	6.5	11.3	12.8	14.9	18.0	7.3	9.4	11.2	7.6
4	7.9	8.6	8.0	5.8	0	5.5	5.1	3.7	8.2	12.6	16.7	19.8	10.1	13.1	15.2	9.8
5	4.4	12.1	11.9	8.3	4.9	0	1.3	2.8	3.7	10.6	14.4	17.2	7.1	10.4	12.8	9.5
6	3.3	10.8	10.2	8.1	4.9	2.1	0	1.8	3.2	7.3	12.0	15.9	6.9	9.3	12.0	8.1
7	4.9	9.9	9.0	7.2	4.0	3.8	1.5	0	4.2	8.8	12.3	15.1	5.3	8.4	10.8	7.7
8	2.6	13.2	12.6	10.5	7.3	3.8	2.6	4.6	0	9.0	13.7	18.0	8.6	11.0	13.8	10.3
9	9.4	16.6	21.1	14.5	11.3	10.1	9.7	8.8	9.2	0	7.4	11.6	7.8	9.9	11.8	10.5
10	13.7	17.1	16.3	15.6	14.2	14.8	12.8	13.5	12.3	7.9	0	4.4	9.0	6.7	13.0	11.6
11	16.5	19.9	19.1	18.4	17.0	18.4	15.6	17.1	15.1	11.8	4.5	0	12.0	9.5	14.0	14.4
12	6.4	10.1	7.9	7.6	7.3	9.7	5.5	7.7	5.3	6.4	8.7	11.8	0	4.5	7.2	3.6
13	10.5	11.2	10.2	9.8	10.3	12.1	8.8	10.7	8.3	9.5	7.6	10.3	3.1	0	4.6	6.2
14	11.8	8.8	9.0	11.6	12.7	14.4	11.2	12.9	10.7	12.2	9.5	11.6	5.5	3.5	0	5.6
15	9.8	7.3	5.1	6.3	9.4	10.2	8.2	7.4	7.7	10.5	11.3	14.4	3.7	5.4	5.5	0

Figure 1 Distances (in km) for Optimization

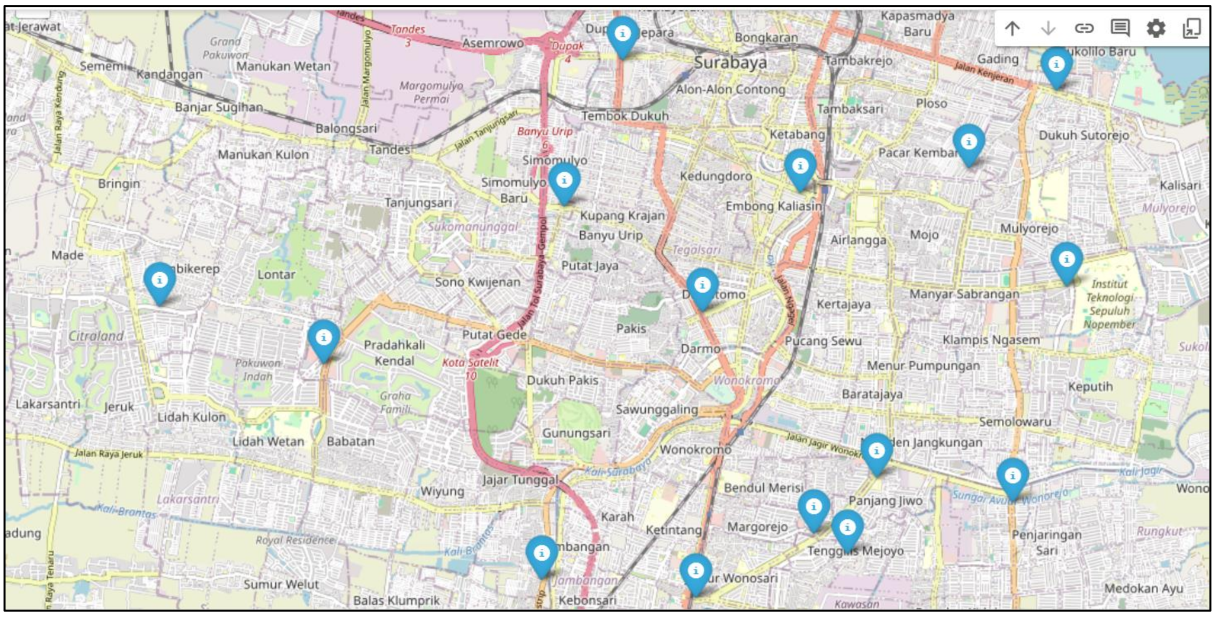


Figure 2 Distribution map of SPBU Shell Location

IV. Algorithm Explanation

This project will utilize the Genetic Algorithm (GA) and compare it with the Enhanced Genetic Algorithm, which combines Edge Recombination and 2-Opt Local Search. However, before the data is utilized, clustering will be performed using several methods. Below is a detailed explanation of the clustering methods and optimization algorithms.

IV.I Clustering Methods

1. DBSCAN Algorithm

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm based on the density of data points (Ester et al., 1996). The fundamental principle of this algorithm is to group nearby data points into a single cluster if they have a sufficient number of neighboring points. Two important parameters used in this algorithm are Eps (the search radius) and MinPts (the minimum number of points required to form a cluster). If a point has many neighbors within its Eps radius that exceed MinPts, it is considered a core point and can form a cluster (Ma et al., 2023).

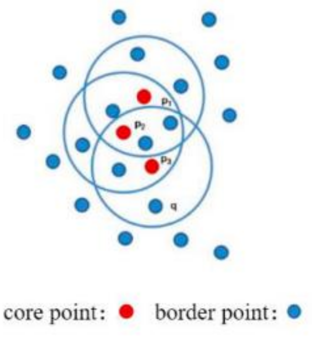


Figure 3 DBSCAN Clustering
(Adapted from (Ma et al., 2023))

The working process of DBSCAN begins by identifying all core points in the dataset. Once the core points are found, the algorithm examines each core point. If the point has not yet been clustered, the algorithm will create a new cluster and add all points within the Eps radius to this cluster. If any point in this cluster is also a core point, the algorithm will continue adding points within its Eps radius until no new points can be added. This process continues until all points in the dataset have been examined. The advantage of DBSCAN is its ability to identify clusters of arbitrary shapes and also detect points considered noise (not belonging to any cluster) (Ma et al., 2023).

2. Affinity Propagation

Affinity propagation is a clustering technique that takes similarity metrics between pairs of data points as its input. The method operates iteratively, where data points exchange information in the form of numerical values. This process continues until high-quality exemplars and well-defined clusters emerge (Frey & Dueck, 2007). Affinity Propagation is intriguing because it dynamically determines the number of clusters based on the input data. This is accomplished through two key parameters: preference, which dictates how many exemplars are selected, and the damping factor, which reduces the responsibility and availability messages to prevent numerical oscillations during updates (Scikit-learn, n.d.).

IV.II Optimization Algorithm

1. Genetic Algorithm

Genetic Algorithm (GA) is a nature-inspired method that draws on the principles of natural evolution as introduced by Charles Darwin. This algorithm employs the mechanism of natural selection, where the best offspring are selected to contribute to the population of the next generation (Sircar et al., 2021). To begin the process, chromosomes are formed from existing variables to create a diverse initial population. Each chromosome is then evaluated, allowing the fittest individuals to survive and reproduce, while the weaker ones are eliminated. This iterative process generates a new population through crossover and mutation, continuing until the algorithm reaches its termination condition (Khanmohammadi et al., 2021). This condition may be defined by criteria such as a maximum number of generations, achieving a satisfactory fitness level, or observing minimal changes in the population

2. Enhanced Genetic Algorithm

Enhanced Genetic Algorithm (EGA) is an advanced optimization technique that builds upon the foundational principles of traditional Genetic Algorithms (GAs) by integrating additional strategies to improve solution quality and convergence speed. EGA employs mechanisms such as adaptive mutation rates, hybrid crossover methods, and local search techniques to enhance the exploration of the solution space while maintaining diversity within the population (Zheng et al., 2020). The algorithm begins with the generation of a diverse initial population, where each individual is evaluated based on a fitness function relevant to the optimization problem at hand. Through iterative processes of selection, crossover, and mutation, EGA not only preserves the fittest individuals but also introduces variability to explore new solutions effectively. This iterative refinement continues until a termination condition is met, such as achieving a satisfactory fitness level or reaching a maximum number of generations (Li et al., 2021). EGA is particularly useful in complex optimization scenarios where traditional methods may struggle to escape local optima.

V. Application of the Algorithm

V.I Clustering Methods

In this project, we explore and compare three sophisticated clustering algorithms like Affinity Propagation, Spectral Clustering, and DBSCAN. Each methodology offers unique approaches to data clustering, with distinct advantages for different types of datasets and clustering challenges.

In this analysis, the optimal clustering was achieved using the Affinity Propagation and DBSCAN methodologies. Both methods produced consistent and optimal results, as evidenced by the clustering output. The dataset, with dimensions of (16, 16), had minimum and maximum values of 1.3 and 21.3, respectively. The best parameters identified for DBSCAN were eps=4.50 and min_samples=2, resulting in four distinct clusters with a Silhouette score of 0.4382. The clusters were as follows: Cluster 1 included nodes [1, 2, 3], Cluster 2 comprised nodes [4, 5, 6, 7, 8], Cluster 3 contained nodes [10, 11], and Cluster 4 consisted of nodes [12, 13, 14, 15].

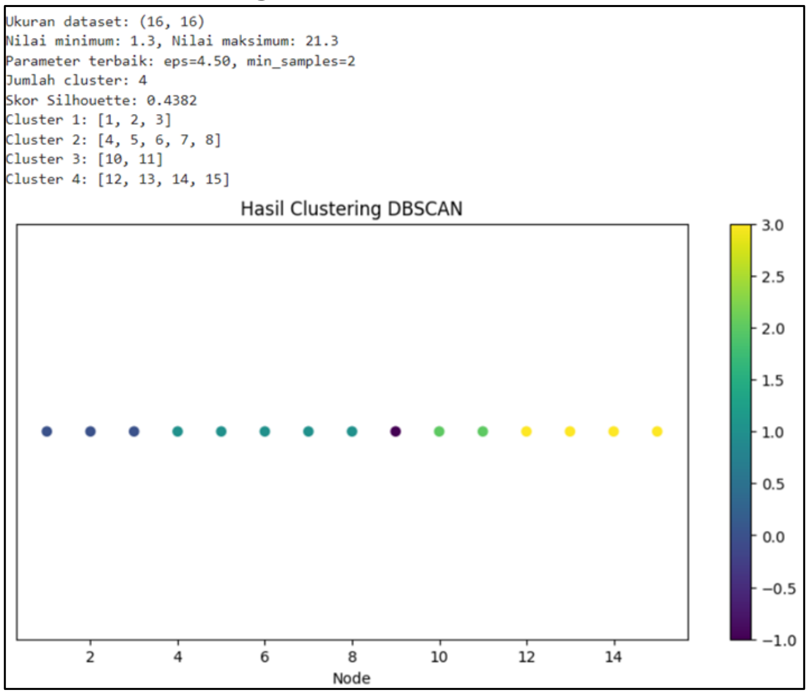
1. DBSCAN

The provided code implements the DBSCAN clustering algorithm on a dataset to determine the optimal clustering configuration. Here is a breakdown of the code's functionality:

a) Applying DBSCAN Clustering Code

```
import numpy as np
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

def baca_dataset(nama_file):
    with open(nama_file, 'r') as file:
        lines = file.readlines()

    data = [list(map(lambda x: float(x) if x != 'inf' else np.inf,
line.strip().split())) for line in lines]
    return np.array(data)

def make_symmetric(matrix):
    return (matrix + matrix.T) / 2

def asymmetric_dbscan(dist_matrix, eps, min_samples):
    # Membuat matriks simetris
    sym_matrix = make_symmetric(dist_matrix)

    # Mengganti nilai tak terhingga dengan nilai maksimum yang
bukan tak terhingga
    max_finite = np.max(sym_matrix[np.isfinite(sym_matrix)])
    sym_matrix[np.isinf(sym_matrix)] = max_finite

    # Mengisi diagonal matriks dengan nol
    np.fill_diagonal(sym_matrix, 0)

    # Melakukan DBSCAN tanpa node 0
    clustering = DBSCAN(eps=eps, min_samples=int(min_samples),
metric='precomputed')
    labels = clustering.fit_predict(sym_matrix[1:, 1:])

    return labels

def evaluate_clustering(dist_matrix, labels):
    # Mengganti nilai tak terhingga dengan nilai maksimum yang
bukan tak terhingga
    max_finite = np.max(dist_matrix[np.isfinite(dist_matrix)])
    dist_matrix_copy = dist_matrix.copy()
    dist_matrix_copy[np.isinf(dist_matrix_copy)] = max_finite
    # Mengisi diagonal matriks dengan nol
    np.fill_diagonal(dist_matrix_copy, 0)

    # Hanya menghitung skor Silhouette jika ada lebih dari satu
cluster
    if len(np.unique(labels)) > 1:
        return silhouette_score(dist_matrix_copy[1:, 1:], labels,
metric='precomputed')
    else:
        return -1  # Nilai default jika hanya ada satu cluster

def main():
    # Membaca dataset
    dataset = baca_dataset("NEW_Dataset.txt")
    print(f"Ukuran dataset: {dataset.shape}")
    print(f"Nilai minimum: {np.min(dataset[np.isfinite(dataset)])},
Nilai maksimum: {np.max(dataset[np.isfinite(dataset)])}")

    # Mencoba berbagai nilai eps dan min_samples
    eps_range = np.linspace(0.5, 5, 10)
    min_samples_range = range(2, 6)

    results = []

    for eps in eps_range:
        for min_samples in min_samples_range:
            labels = asymmetric_dbscan(dataset, eps, min_samples)
            n_clusters = len(set(labels)) - (1 if -1 in labels else
0)  # Tidak menghitung noise sebagai cluster
            score = evaluate_clustering(dataset, labels)
            results.append((eps, min_samples, n_clusters, score))

    results_df = pd.DataFrame(results, columns=['eps',
'min_samples', 'n_clusters', 'silhouette_score'])

    # Memilih parameter terbaik berdasarkan skor Silhouette
    best_params =
results_df.loc[results_df['silhouette_score'].idxmax()]

    print(f"Parameter terbaik: eps={best_params['eps']:.2f},
min_samples={int(best_params['min_samples'])}")
    print(f"Jumlah cluster: {int(best_params['n_clusters'])}")
    print(f"Skor Silhouette:
{best_params['silhouette_score']:.4f}")

    # Melakukan clustering final dengan parameter terbaik
    final_labels = asymmetric_dbscan(dataset, best_params['eps'],
int(best_params['min_samples']))

    # Menampilkan hasil
    unique_labels = set(final_labels)
    for i in unique_labels:
        if i != -1:  # Mengabaikan noise points
            nodes_in_cluster = np.where(final_labels == i)[0] + 1
# Menambahkan 1 untuk menyesuaikan indeks
            print(f"Cluster {i+1}: {nodes_in_cluster.tolist()}")

    # Visualisasi hasil clustering
    plt.figure(figsize=(10, 5))
    plt.scatter(range(1, len(final_labels)+1),
[0]*len(final_labels), c=final_labels, cmap='viridis')
    plt.title('Hasil Clustering DBSCAN')
    plt.xlabel('Node')
    plt.yticks([])
    plt.colorbar(label='Cluster')
    plt.show()

if __name__ == "__main__":
    main()
```

b) Output of DBSCAN Clustering Code



```
Ukuran dataset: (16, 16)
Nilai minimum: 1.3, Nilai maksimum: 21.3
Parameter terbaik: eps=4.50, min_samples=2
Jumlah cluster: 4
Skor Silhouette: 0.4382
Cluster 1: [1, 2, 3]
Cluster 2: [4, 5, 6, 7, 8]
Cluster 3: [10, 11]
Cluster 4: [12, 13, 14, 15]
```

Figure 4 Output of DBSCAN Clustering

The DBSCAN clustering results show four distinct clusters with optimal parameters of eps =4.50 and min samples = 2 The visualization confirms the distribution of nodes into these clusters, achieving a Silhouette score of 0.4382, indicating moderate clustering quality.

The provided code implements the DBSCAN clustering algorithm on a dataset to determine the optimal clustering configuration. Here is a breakdown of the code's functionality:

1) Reading the Dataset: The baca_dataset function reads a file containing the dataset, processes it into a matrix format where infinite values are replaced with np.inf, and returns this matrix as a NumPy array.

2) Making the Matrix Symmetric: The make_symmetric function takes a matrix and returns its symmetric version by averaging it with its transpose.
3) Asymmetric DBSCAN: The asymmetric_dbscan function performs clustering using DBSCAN on a symmetric distance matrix. It replaces infinite values with the maximum finite value in the matrix and sets diagonal elements to zero to prepare the matrix for clustering.
4) Clustering Evaluation: The evaluate_clustering function calculates the Silhouette score to evaluate clustering quality. It checks for more than one cluster before computing the score to ensure meaningful results.
5) Main Execution: In the main function, the dataset is read and its size and value range are printed. Various combinations of eps and min_samples are tested to find the best parameters based on the highest Silhouette score. The final clustering is performed using these optimal parameters, and results are displayed by printing cluster compositions.
6) Visualization: Finally, a scatter plot visualizes the clustering results, where each node is colored according to its cluster assignment.

2. Affinity Propagation

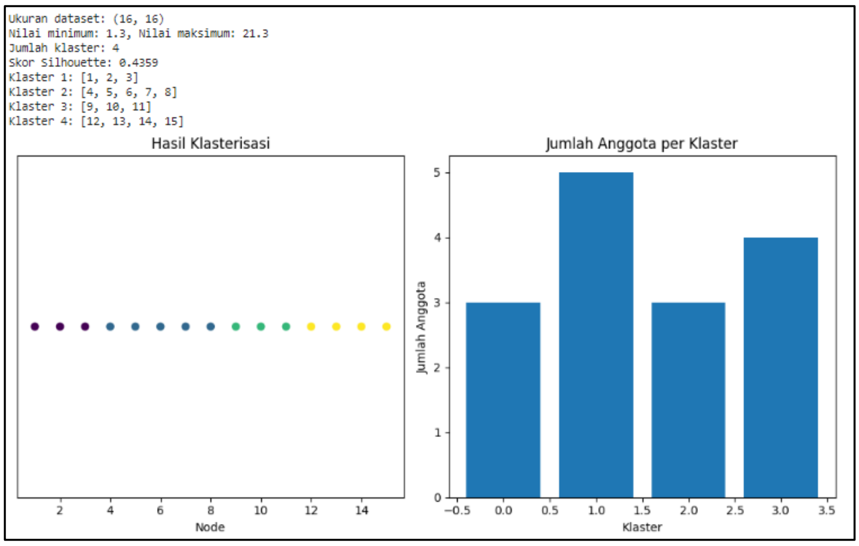
a) Applying Affinity Propagation Clustering Code

The provided code implements the Affinity Propagation clustering algorithm to analyze a dataset and determine the optimal clustering configuration. Below is a detailed breakdown of the code's functionality:

```
import numpy as np
import pandas as pd
from sklearn.cluster import AffinityPropagation
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

def baca_dataset(nama_file):
    with open(nama_file, 'r') as file:
        lines = file.readlines()

    data = [list(map(lambda x: float(x) if x != 'inf' else
np.inf, line.strip().split())) for line in lines]
    return np.array(data)

def make_symmetric(matrix):
    return (matrix + matrix.T) / 2

def asymmetric_affinity_propagation(dist_matrix):
    # Membuat matriks simetris
    sym_matrix = make_symmetric(dist_matrix)

    # Mengganti nilai tak terhingga dengan nilai maksimum yang
bukan tak terhingga
    max_finite = np.max(sym_matrix[np.isfinite(sym_matrix)])
    sym_matrix[np.isinf(sym_matrix)] = max_finite
    # Mengisi diagonal matriks dengan nol
    np.fill_diagonal(sym_matrix, 0)

    # Mengubah matriks jarak menjadi matriks kemiripan
    similarity_matrix = -sym_matrix

    # Melakukan Affinity Propagation tanpa node 0
    clustering = AffinityPropagation(random_state=42,
damping=0.9, max_iter=1000)
    labels = clustering.fit_predict(similarity_matrix[1:, 1:])

    return labels

def evaluate_clustering(dist_matrix, labels):
    # Mengganti nilai tak terhingga dengan nilai maksimum yang
bukan tak terhingga
    max_finite = np.max(dist_matrix[np.isfinite(dist_matrix)])
    dist_matrix_copy = dist_matrix.copy()
    dist_matrix_copy[np.isinf(dist_matrix_copy)] = max_finite

    # Mengisi diagonal matriks dengan nol
    np.fill_diagonal(dist_matrix_copy, 0)

    return silhouette_score(dist_matrix_copy[1:, 1:], labels,
metric='precomputed')

def main():
    # Membaca dataset
    dataset = baca_dataset("NEW_Dataset.txt")
    print(f"Ukuran dataset: {dataset.shape}")
    print(f"Nilai minimum:
{np.min(dataset[np.isfinite(dataset)])}, Nilai maksimum:
{np.max(dataset[np.isfinite(dataset)])}")

    # Melakukan klasterisasi dengan Affinity Propagation
    labels = asymmetric_affinity_propagation(dataset)

    # Menghitung skor Silhouette
    score = evaluate_clustering(dataset, labels)

    # Menampilkan hasil
    n_clusters = len(np.unique(labels))
    print(f"Jumlah klaster: {n_clusters}")
    print(f"Skor Silhouette: {score:.4f}")

    for i in range(n_clusters):
        nodes_in_cluster = np.where(labels == i)[0] + 1  #
Menambahkan 1 untuk menyesuaikan indeks
        print(f"Klaster {i+1}: {nodes_in_cluster.tolist()}")

    # Visualisasi hasil klasterisasi
    plt.figure(figsize=(10, 5))
```

```python
    plt.subplot(121)
    plt.scatter(range(1, len(labels)+1), [0]*len(labels),
c=labels, cmap='viridis')
    plt.title('Hasil Klasterisasi')
    plt.xlabel('Node')
    plt.yticks([])

    plt.subplot(122)
    unique_labels, counts = np.unique(labels,
return_counts=True)
    plt.bar(unique_labels, counts)
    plt.title('Jumlah Anggota per Klaster')
    plt.xlabel('Klaster')
    plt.ylabel('Jumlah Anggota')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

b) Output of Affinity Propagation Clustering Code:



```
Ukuran dataset: (16, 16)
Nilai minimum: 1.3, Nilai maksimum: 21.3
Jumlah klaster: 4
Skor Silhouette: 0.4359
Klaster 1: [1, 2, 3]
Klaster 2: [4, 5, 6, 7, 8]
Klaster 3: [9, 10, 11]
Klaster 4: [12, 13, 14, 15]
```

Figure 5 Output of Affinity Propagation Clustering

The output from the Affinity Propagation clustering algorithm, as depicted in the image, shows that the dataset has been divided into four distinct clusters. The dataset consists of nodes with values ranging from 1.3 to 21.3, and the clustering process achieved a Silhouette score of 0.4359, indicating moderate clustering quality. The scatter plot on the left visualizes the distribution of nodes across clusters, while the bar chart on the right illustrates the number of nodes in each cluster. Cluster 2 has the most members, followed by Cluster 4, Cluster 1, and Cluster 3. The visualization confirms the effective separation of nodes into meaningful groups

1) Reading the Dataset

The baca_dataset function reads a dataset from a file and processes it into a matrix format. The function replaces any 'inf' string values with np.inf to handle infinite values appropriately. The resulting data is returned as a NumPy array.

2) Making the Matrix Symmetric

The make_symmetric function ensures that the distance matrix is symmetric by averaging it with its transpose. This step is crucial for algorithms that require symmetric matrices.

3) Asymmetric Affinity Propagation

The asymmetric_affinity_propagation function performs clustering using Affinity Propagation on a symmetric distance matrix. It first makes the matrix symmetric, replaces infinite values with the maximum finite value, and sets diagonal elements to zero. The distance matrix is then converted into a similarity matrix by negating its values. Affinity Propagation is applied to this similarity matrix, excluding the first node (index 0).

4) Clustering Evaluation

The evaluate_clustering function calculates the Silhouette score to assess the quality of clustering results. It prepares the distance matrix by replacing infinite values and setting diagonal elements to zero before computing the score.

5) Main Execution

In the main function, the dataset is read and its size and value range are printed. Affinity Propagation is applied to determine cluster labels, and the Silhouette score is calculated to evaluate clustering quality. The number of clusters and their compositions are printed.

6) Visualization

Finally, the clustering results are visualized using matplotlib. A scatter plot displays nodes colored according to their cluster assignments, and a bar chart shows the number of members per cluster.

V.II Optimization Algorithm

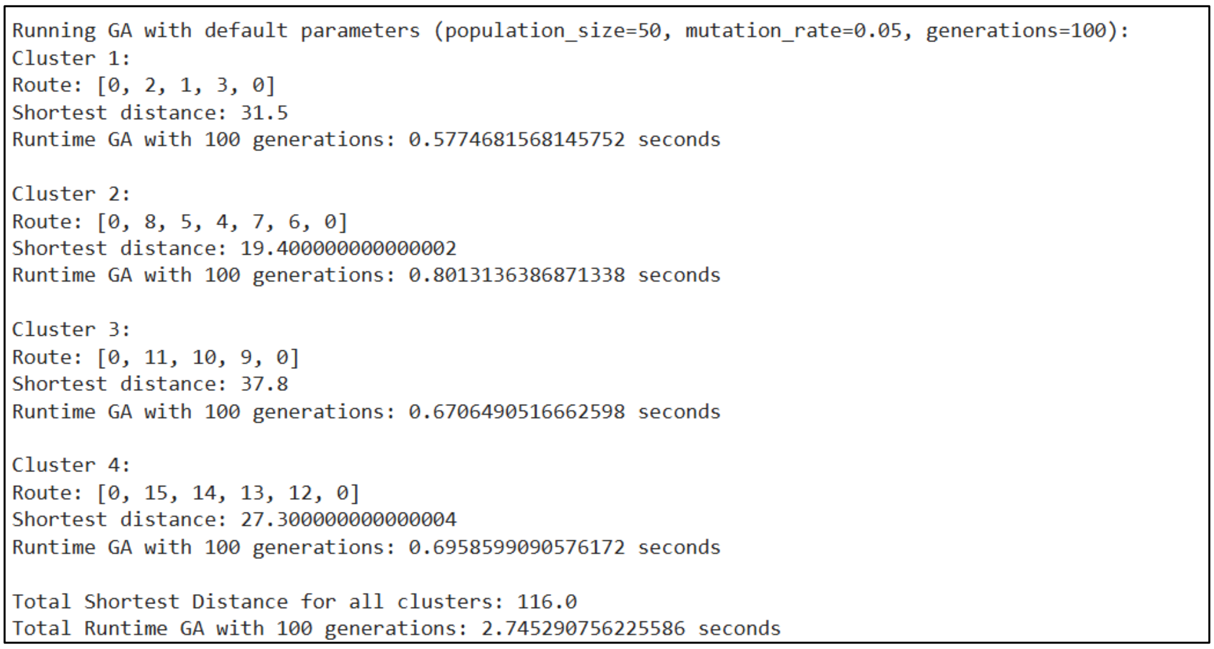
1. Genetic Algorithm

a. With Clustered Data

```
import numpy as np
import random
import time

class GeneticAlgorithm:
    def     __init__(self,     distance_matrix,     population_size,
mutation_rate, generations):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations
        self.population = self.initialize_population()
        self.best_distance = float('inf')
        self.best_route = None

    def initialize_population(self):
        population = []
        for _ in range(self.population_size):
            chromosome = list(range(1, len(self.distance_matrix)))
            random.shuffle(chromosome)
            chromosome = [0] + chromosome + [0]
            population.append(chromosome)
        return population

    def calculate_fitness(self, chromosome):
        total_distance = 0
        for i in range(len(chromosome) - 1):
            total_distance                                          +=
self.distance_matrix[chromosome[i]][chromosome[i + 1]]
        return 1 / total_distance

    def select_parents(self):
        tournament_size = 5
        tournament = random.sample(self.population, tournament_size)
        return max(tournament, key=self.calculate_fitness)

    def crossover(self, parent1, parent2):
        start, end = sorted(random.sample(range(1, len(parent1) - 1),
2))
        child = [None] * len(parent1)
        child[0] = 0
        child[-1] = 0
        child[start:end] = parent1[start:end]

        pointer = end
        for gene in parent2[1:-1]:
            if gene not in child:
                if pointer == len(child) - 1:
                    pointer = 1
                child[pointer] = gene
                pointer += 1

        return child

    def mutate(self, chromosome):
        for i in range(1, len(chromosome) - 1):
            if random.random() < self.mutation_rate:
                j = random.randint(1, len(chromosome) - 2)
                chromosome[i],    chromosome[j]    =    chromosome[j],
chromosome[i]

    def evolve(self):
        new_population = []
        for _ in range(self.population_size):
            parent1 = self.select_parents()
            parent2 = self.select_parents()
            child = self.crossover(parent1, parent2)
            self.mutate(child)
            new_population.append(child)
        self.population = new_population

    def run(self):
        start_time = time.time()
        for _ in range(self.generations):
            self.evolve()
            best_chromosome          =          max(self.population,
key=self.calculate_fitness)
            best_distance              =              1             /
self.calculate_fitness(best_chromosome)
            if best_distance < self.best_distance:
                self.best_distance = best_distance
                self.best_route = best_chromosome
        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime

def run_ga_with_params(distance_matrix, clusters, population_size=50,
mutation_rate=0.05, generations=100):
    total_shortest_distance = 0
    total_runtime = 0

    for i, cluster in enumerate(clusters):
        selected_distance_matrix   =   distance_matrix[np.ix_(cluster,
cluster)]

        ga_solver = GeneticAlgorithm(selected_distance_matrix,
                                     population_size,
                                     mutation_rate,
                                     generations)

        best_route, best_distance, runtime = ga_solver.run()

        final_route = [cluster[index] for index in best_route]

        print(f"Cluster {i + 1}:")
        print("Route:", final_route)
        print("Shortest distance:", best_distance)
        print(f"Runtime GA with {generations} generations:", runtime,
"seconds")
        print()

        total_shortest_distance += best_distance
        total_runtime += runtime

    print("Total    Shortest    Distance    for    all    clusters:",
total_shortest_distance)
    print(f"Total   Runtime   GA   with   {generations}   generations:",
total_runtime, "seconds")
    print("=" * 100)

def main():
    distance_matrix = np.loadtxt("NEW_Dataset.txt")

    clusters = [
        [0, 1, 2, 3],
        [0, 4, 5, 6, 7, 8],
        [0, 9, 10, 11],
        [0, 12, 13, 14, 15],
    ]

    # Running GA with default parameters
    print("Running  GA  with  default  parameters  (population_size=50,
mutation_rate=0.05, generations=100):")
    run_ga_with_params(distance_matrix, clusters)

    # Running GA with custom parameters 1
    print("\nRunning GA with custom parameters 1 (population_size=75,
mutation_rate=0.03, generations=150):")
    run_ga_with_params(distance_matrix, clusters, population_size=75,
mutation_rate=0.03, generations=150)

    # Running GA with custom parameters 2
    print("\nRunning GA with custom parameters 2 (population_size=40,
mutation_rate=0.07, generations=200):")
    run_ga_with_params(distance_matrix, clusters, population_size=40,
mutation_rate=0.07, generations=200)

if __name__ == '__main__':
    main()
```

Output:

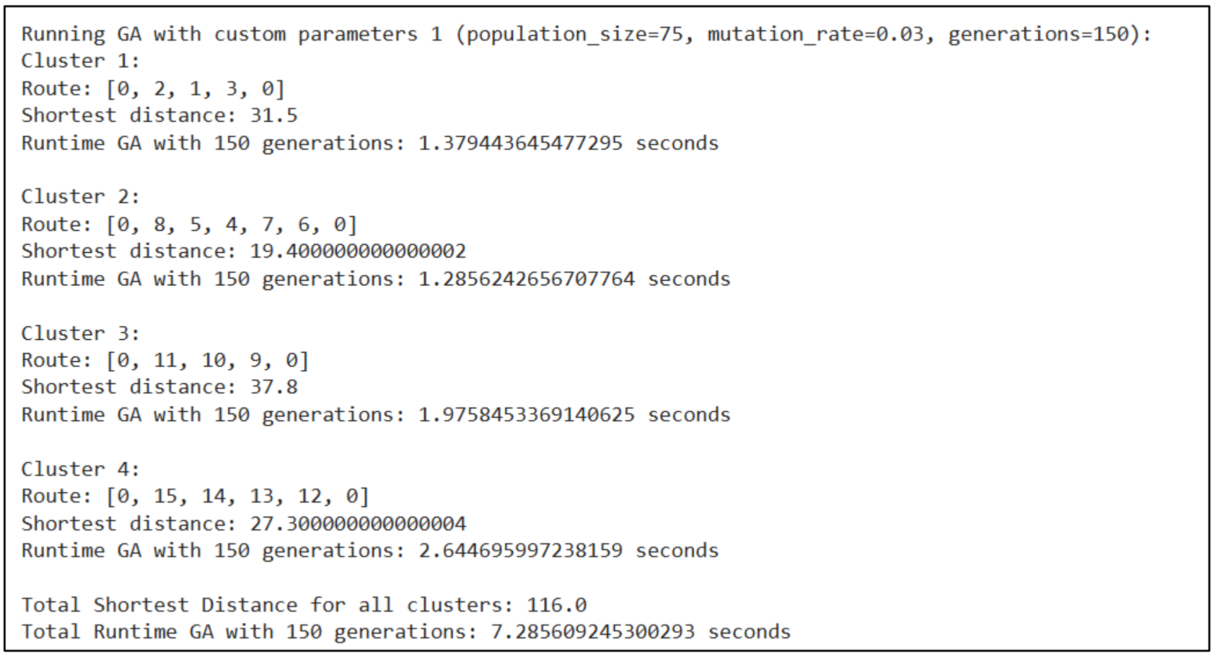
a. Output for scenario 1:



```
Running GA with default parameters (population_size=50, mutation_rate=0.05, generations=100):
Cluster 1:
Route: [0, 2, 1, 3, 0]
Shortest distance: 31.5
Runtime GA with 100 generations: 0.5774681568145752 seconds

Cluster 2:
Route: [0, 8, 5, 4, 7, 6, 0]
Shortest distance: 19.400000000000002
Runtime GA with 100 generations: 0.8013136386871338 seconds

Cluster 3:
Route: [0, 11, 10, 9, 0]
Shortest distance: 37.8
Runtime GA with 100 generations: 0.6706490516662598 seconds

Cluster 4:
Route: [0, 15, 14, 13, 12, 0]
Shortest distance: 27.300000000000004
Runtime GA with 100 generations: 0.6958599090576172 seconds

Total Shortest Distance for all clusters: 116.0
Total Runtime GA with 100 generations: 2.745290756225586 seconds
```

Figure 6 Output for Scenario 1 of GA Clustered

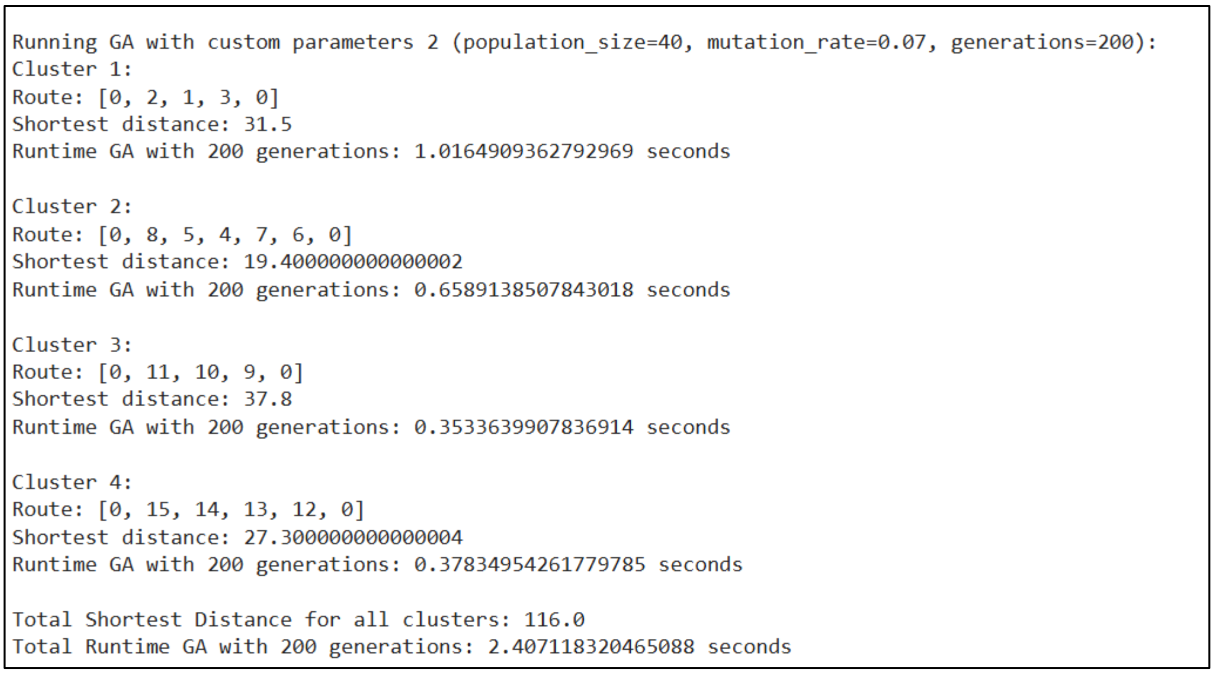
b. Output for scenario 2:



```
Running GA with custom parameters 1 (population_size=75, mutation_rate=0.03, generations=150):
Cluster 1:
Route: [0, 2, 1, 3, 0]
Shortest distance: 31.5
Runtime GA with 150 generations: 1.379443645477295 seconds

Cluster 2:
Route: [0, 8, 5, 4, 7, 6, 0]
Shortest distance: 19.400000000000002
Runtime GA with 150 generations: 1.2856242656707764 seconds

Cluster 3:
Route: [0, 11, 10, 9, 0]
Shortest distance: 37.8
Runtime GA with 150 generations: 1.9758453369140625 seconds

Cluster 4:
Route: [0, 15, 14, 13, 12, 0]
Shortest distance: 27.300000000000004
Runtime GA with 150 generations: 2.644695997238159 seconds

Total Shortest Distance for all clusters: 116.0
Total Runtime GA with 150 generations: 7.285609245300293 seconds
```

Figure 7 Output for Scenario 2 of GA Clustered

c. Output for scenario 3:



```
Running GA with custom parameters 2 (population_size=40, mutation_rate=0.07, generations=200):
Cluster 1:
Route: [0, 2, 1, 3, 0]
Shortest distance: 31.5
Runtime GA with 200 generations: 1.0164909362792969 seconds

Cluster 2:
Route: [0, 8, 5, 4, 7, 6, 0]
Shortest distance: 19.400000000000002
Runtime GA with 200 generations: 0.6589138507843018 seconds

Cluster 3:
Route: [0, 11, 10, 9, 0]
Shortest distance: 37.8
Runtime GA with 200 generations: 0.3533639907836914 seconds

Cluster 4:
Route: [0, 15, 14, 13, 12, 0]
Shortest distance: 27.300000000000004
Runtime GA with 200 generations: 0.37834954261779785 seconds

Total Shortest Distance for all clusters: 116.0
Total Runtime GA with 200 generations: 2.407118320465088 seconds
```

Figure 8 Output for Scenario 3 of GA Clustered

The provided code implements a Genetic Algorithm (GA) for solving optimization problems using a distance matrix. Here's a detailed explanation of its components and functionality:

1. ‘__init__’ Method

    This constructor initializes the GeneticAlgorithm class with parameters such as the distance matrix, population size, mutation rate, and number of generations. It also initializes the population and sets the best distance and route to default values.
2. ‘initialize_population’

    This function creates an initial population of potential solutions (chromosomes). Each chromosome represents a route starting and ending at node 0, with the intermediate nodes shuffled randomly.
3. ‘calculate_fitness’

    This method calculates the fitness of a chromosome by computing the total distance of the route it represents. Fitness is defined as the inverse of this distance, meaning shorter routes have higher fitness.
4. ‘select_parents’

    This function selects parents for crossover using a tournament selection method. A subset of the population is randomly chosen, and the chromosome with the highest fitness is selected as a parent.
5. ‘crossover’

    The crossover function creates a new chromosome (child) by combining segments from two parent chromosomes. It uses a partially mapped crossover approach, preserving the order and position of nodes from one parent while filling in remaining nodes from the other.
6. ‘mutate’

    This method introduces genetic diversity by randomly swapping two nodes in a chromosome with a probability defined by the mutation rate. This helps explore new solutions and avoid local minima.
7. ‘evolve’

    The evolve function generates a new population by selecting parents, performing crossover, and applying mutations to offspring. This process iterates over the entire population size to create a new generation.
8. ‘run’

    This method executes the GA over a specified number of generations, evolving the population to find an optimal solution. It tracks the best route and distance found during execution and measures runtime.
9. ‘run_ga_with_params’

    This function runs the GA on multiple clusters using specified parameters for population size, mutation rate, and generations. It calculates total shortest distances and runtime for all clusters, printing results for each cluster.
10. ‘main'

    The main function loads a distance matrix from a file and defines clusters to be optimized individually using GA. It runs GA with default and custom parameters, outputting routes, distances, and runtimes for each configuration.

The output from the GA with custom parameters (population size = 40, mutation rate = 0.07, generations = 200) shows the optimization results for four clusters:

- Cluster 1: The route [0, 2, 1, 3, 0] has a shortest distance of 31.5 and took approximately 1.02 seconds to compute.

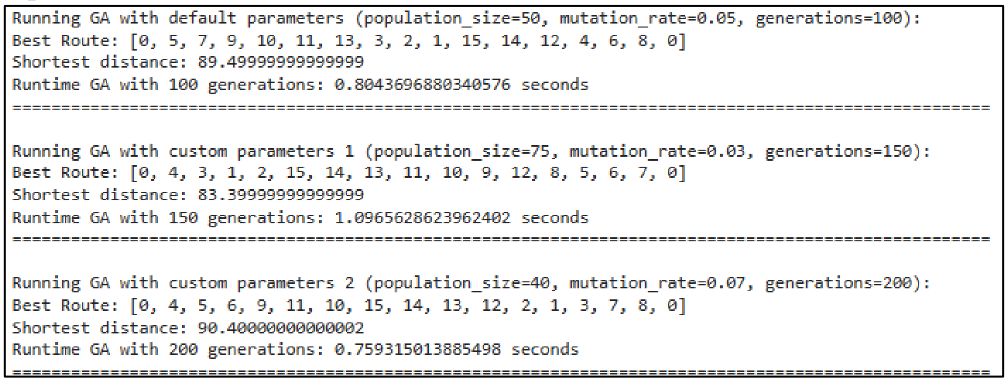
- Cluster 2: The route [0, 8, 5, 4, 7, 6, 0] achieved a shortest distance of 19.4 with a runtime of about 0.66 seconds.
- Cluster 3: The route [0, 11, 10, 9, 0] resulted in a shortest distance of 37.8 and took around 0.35 seconds.
- Cluster 4: The route [0, 15, 14, 13, 12, 0] had a shortest distance of 27.3 with a computation time of roughly 0.38 seconds.

The total shortest distance for all clusters combined is 116.0, and the total runtime for the GA across all clusters is approximately 2.41 seconds. This indicates efficient optimization with moderate computational time given the parameter settings.

b. With Non-Clustered Data

```
import numpy as np
import random
import time

class GeneticAlgorithm:
    def __init__(self, distance_matrix, population_size,
mutation_rate, generations):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations
        self.population = self.initialize_population()
        self.best_distance = float('inf')
        self.best_route = None

    def initialize_population(self):
        population = []
        for _ in range(self.population_size):
            chromosome = list(range(1,
len(self.distance_matrix)))
            random.shuffle(chromosome)
            chromosome = [0] + chromosome + [0]
            population.append(chromosome)
        return population

    def calculate_fitness(self, chromosome):
        total_distance = 0
        for i in range(len(chromosome) - 1):
            total_distance +=
self.distance_matrix[chromosome[i]][chromosome[i + 1]]
        return 1 / total_distance

    def select_parents(self):
        tournament_size = 5
        tournament = random.sample(self.population,
tournament_size)
        return max(tournament, key=self.calculate_fitness)

    def crossover(self, parent1, parent2):
        start, end = sorted(random.sample(range(1, len(parent1)
- 1), 2))
        child = [None] * len(parent1)
        child[0] = 0
        child[-1] = 0
        child[start:end] = parent1[start:end]

        pointer = end
        for gene in parent2[1:-1]:
            if gene not in child:
                if pointer == len(child) - 1:
                    pointer = 1
                child[pointer] = gene
                pointer += 1

        return child

    def mutate(self, chromosome):
        for i in range(1, len(chromosome) - 1):
            if random.random() < self.mutation_rate:
                j = random.randint(1, len(chromosome) - 2)
                chromosome[i], chromosome[j] = chromosome[j],
chromosome[i]

    def evolve(self):
        new_population = []
        for _ in range(self.population_size):
            parent1 = self.select_parents()
            parent2 = self.select_parents()
            child = self.crossover(parent1, parent2)
            self.mutate(child)
            new_population.append(child)
        self.population = new_population

    def run(self):
        start_time = time.time()
        for _ in range(self.generations):
            self.evolve()
            best_chromosome = max(self.population,
key=self.calculate_fitness)
            best_distance = 1 /
self.calculate_fitness(best_chromosome)
            if best_distance < self.best_distance:
                self.best_distance = best_distance
                self.best_route = best_chromosome
        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime

def run_ga_with_params(distance_matrix, population_size=50,
mutation_rate=0.05, generations=100):
    ga_solver = GeneticAlgorithm(distance_matrix,
                                 population_size,
                                 mutation_rate,
                                 generations)

    best_route, best_distance, runtime = ga_solver.run()

    print("Best Route:", best_route)
    print("Shortest distance:", best_distance)
    print(f"Runtime GA with {generations} generations:",
runtime, "seconds")
    print("=" * 100)

def main():
    distance_matrix = np.loadtxt("NEW_Dataset.txt")

    # Running GA with default parameters
    print("Running GA with default parameters
(population_size=50, mutation_rate=0.05, generations=100):")
    run_ga_with_params(distance_matrix)

    # Running GA with custom parameters 1
    print("\nRunning GA with custom parameters 1
(population_size=75, mutation_rate=0.03, generations=150):")
    run_ga_with_params(distance_matrix, population_size=75,
mutation_rate=0.03, generations=150)

    # Running GA with custom parameters 2
    print("\nRunning GA with custom parameters 2
(population_size=40, mutation_rate=0.07, generations=200):")
    run_ga_with_params(distance_matrix, population_size=40,
mutation_rate=0.07, generations=200)

if __name__ == '__main__':
    main()
```

Output:



```
Running GA with default parameters (population_size=50, mutation_rate=0.05, generations=100):
Best Route: [0, 5, 7, 9, 10, 11, 13, 3, 2, 1, 15, 14, 12, 4, 6, 8, 0]
Shortest distance: 89.49999999999999
Runtime GA with 100 generations: 0.8043696880340576 seconds
==============================================================================================

Running GA with custom parameters 1 (population_size=75, mutation_rate=0.03, generations=150):
Best Route: [0, 4, 3, 1, 2, 15, 14, 13, 11, 10, 9, 12, 8, 5, 6, 7, 0]
Shortest distance: 83.39999999999999
Runtime GA with 150 generations: 1.0965628623962402 seconds
==============================================================================================

Running GA with custom parameters 2 (population_size=40, mutation_rate=0.07, generations=200):
Best Route: [0, 4, 5, 6, 9, 11, 10, 15, 14, 13, 12, 2, 1, 3, 7, 8, 0]
Shortest distance: 90.40000000000002
Runtime GA with 200 generations: 0.759315013885498 seconds
==============================================================================================
```

Figure 9 Output for GA Non-Clustered

The provided code implements a Genetic Algorithm (GA) designed to solve the Traveling Salesman Problem (TSP) using a distance matrix without clustering the data. This approach focuses on optimizing a single route that visits multiple nodes, returning to the starting point. The following sections provide a detailed explanation of the code's components, functionality, and its comparison with a clustered data approach.

In contrast to this non-clustered approach, where all nodes are treated equally without grouping them into clusters, clustered data implementations segment nodes into groups based on proximity or other criteria before applying GA optimization techniques.

1. **Data Segmentation:**
    - In clustered approaches, data points are divided into clusters before optimization begins, allowing for localized optimization within each cluster.
    - Non-clustered approaches optimize a single global route that encompasses all nodes without prior segmentation.
2. **Complexity and Computation:**
    - Clustered methods generally reduce computational complexity by solving smaller optimization problems individually before aggregating results.
    - Non-clustered methods may require more computational resources as they tackle larger datasets in one go.
3. **Results Interpretation**:
    - Clustered approaches provide insights into local optimal routes within specific regions but may miss global optimizations across clusters.
    - Non-clustered methods aim for an overall optimal solution that considers all nodes collectively.
4. **Performance Metrics:**
    - Performance metrics such as runtime and shortest distances can vary significantly between approaches due to differences in computational strategies and problem sizes addressed.

2. Enhanced Genetic Algorithm (EGA)

The Enhanced Genetic Algorithm (EGA) implementation provided is used to solve the ACTSP for supervising Shell Gas Stations in Surabaya. This code was chosen for its ability to handle complex routing problems efficiently and its adaptability to asymmetric distance matrices.

Key parameters and features of the Enhanced Genetic Algorithm:

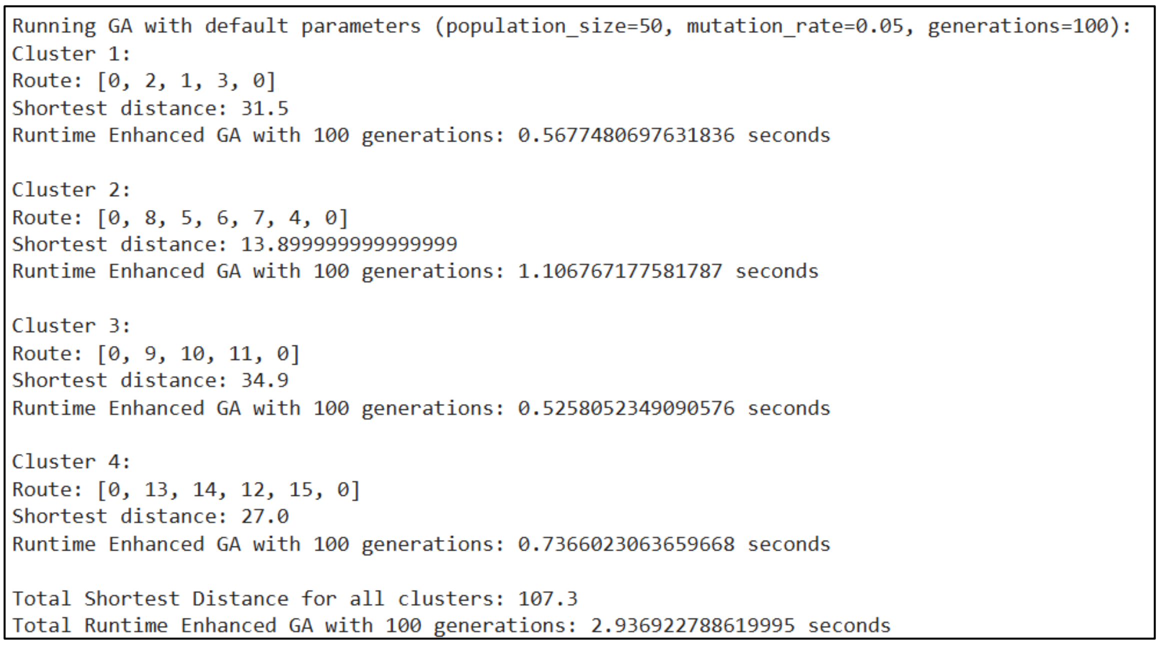
a. Population Size: Determines the number of candidate solutions in each generation (default: 50).
b. Mutation Rate: Sets the probability of random changes in a solution, with adaptive adjustment (initial default: 0.05).
c. Generations: Specifies the number of iterations the algorithm runs (default: 100).
d. Edge Recombination Crossover: A specialized crossover operator that preserves edge information from parent solutions.
e. 2-Opt Local Search: Improves solutions by swapping edges to reduce total distance.
f. Elitism: Preserves the best solutions from each generation (10% of population).
g. Adaptive Mutation Rate: Adjusts the mutation rate based on population diversity.
h. Clustering: The Gas Stations are divided into 4 clusters for more efficient route planning.

The code runs the EGA with default parameters and two sets of custom parameters to compare performance. It outputs the best route found for each cluster, the shortest distance, and the runtime for each set of parameters. This approach allows for efficient planning of supervision routes, potentially reducing travel time and costs for Disperindag officers while ensuring comprehensive coverage of all Shell Gas Stations in Surabaya.

a) With Clustered Data

```python
import numpy as np
import random
import time

class EnhancedGeneticAlgorithm:
    def __init__(self, distance_matrix, population_size,
initial_mutation_rate, generations):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.mutation_rate = initial_mutation_rate
        self.generations = generations
        self.population = self.initialize_population()
        self.best_distance = float('inf')
        self.best_route = None

    def initialize_population(self):
        population = []
        for _ in range(self.population_size):
            chromosome = list(range(1,
len(self.distance_matrix)))
            random.shuffle(chromosome)
            chromosome = [0] + chromosome + [0]
            population.append(chromosome)
        return population

    def calculate_fitness(self, chromosome):
        total_distance = 0
        for i in range(len(chromosome) - 1):
            total_distance +=
self.distance_matrix[chromosome[i]][chromosome[i + 1]]
        return 1 / total_distance

    def select_parents(self):
        tournament_size = 5
        tournament = random.sample(self.population,
tournament_size)
        return max(tournament, key=self.calculate_fitness)

    def edge_recombination_crossover(self, parent1, parent2):
        size = len(parent1)
        edge_map = {i: set() for i in range(size)}

        for parent in [parent1, parent2]:
            for i in range(size):
                edge_map[parent[i]].add(parent[(i-1) % size])
                edge_map[parent[i]].add(parent[(i+1) % size])

        current = random.choice([parent1[1], parent2[1]])  #
Start with a random city (excluding depot)
        child = [0, current]

        while len(child) < size - 1:
            if edge_map[current]:
                next_node = min(edge_map[current], key=lambda x:
len(edge_map[x]))
            else:
                unvisited = set(range(1, size-1)) - set(child)
                next_node = random.choice(list(unvisited))

            child.append(next_node)
            current = next_node

            for node in edge_map:
                if current in edge_map[node]:
                    edge_map[node].remove(current)

        child.append(0)  # Add depot at the end
        return child

    def two_opt(self, route):
        improved = True
        while improved:
            improved = False
            for i in range(1, len(route) - 2):
                for j in range(i + 1, len(route) - 1):
                    if j - i == 1: continue
                    new_route = route[:i] + route[i:j+1][::-1] +
route[j+1:]
                    if self.calculate_fitness(new_route) >
self.calculate_fitness(route):
                        route = new_route
                        improved = True
            if improved:
                break  # Stop after first improvement to save
time
        return route

    def mutate(self, chromosome):
        for i in range(1, len(chromosome) - 1):
            if random.random() < self.mutation_rate:
                j = random.randint(1, len(chromosome) - 2)
                chromosome[i], chromosome[j] = chromosome[j],
chromosome[i]

    def evolve(self):
        new_population = []
        elite_size = int(0.1 * self.population_size)
        sorted_population = sorted(self.population,
key=self.calculate_fitness, reverse=True)
        new_population.extend(sorted_population[:elite_size])

        while len(new_population) < self.population_size:
            parent1 = self.select_parents()
            parent2 = self.select_parents()
            child = self.edge_recombination_crossover(parent1,
parent2)
            child = self.two_opt(child)
            self.mutate(child)
            new_population.append(child)

        self.population = new_population

        # Adaptive mutation rate
        best_fitness =
self.calculate_fitness(max(self.population,
key=self.calculate_fitness))
        avg_fitness = sum(self.calculate_fitness(ind) for ind in
self.population) / self.population_size
        self.mutation_rate = 0.5 * (1 - (best_fitness -
avg_fitness) / best_fitness)

    def run(self):
        start_time = time.time()
        for _ in range(self.generations):
            self.evolve()
            best_chromosome = max(self.population, 
key=self.calculate_fitness)
            best_distance = 1 / 
self.calculate_fitness(best_chromosome)
            if best_distance < self.best_distance:
                self.best_distance = best_distance
                self.best_route = best_chromosome
        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime

def run_ga_with_params(distance_matrix, clusters, 
population_size=50, mutation_rate=0.05, generations=100):
    total_shortest_distance = 0
    total_runtime = 0

    for i, cluster in enumerate(clusters):
        selected_distance_matrix = 
distance_matrix[np.ix_(cluster, cluster)]

        ga_solver = 
EnhancedGeneticAlgorithm(selected_distance_matrix, 
                                             population_size, 
                                             mutation_rate, 
                                             generations)

        best_route, best_distance, runtime = ga_solver.run()

        # Ensure all nodes are included
        final_route = [0]  # Start with depot
        remaining_nodes = set(range(1, len(cluster)))
        for index in best_route[1:-1]:  # Exclude first and last 
(depot)
            if index in remaining_nodes:
                final_route.append(cluster[index])
                remaining_nodes.remove(index)

        # Add any remaining nodes
        final_route.extend([cluster[node] for node in 
remaining_nodes])
        final_route.append(0)  # End with depot

        print(f"Cluster {i + 1}:")
        print("Route:", final_route)
        print("Shortest distance:", best_distance)
        print(f"Runtime Enhanced GA with {generations} 
generations:", runtime, "seconds")
        print()

        total_shortest_distance += best_distance
        total_runtime += runtime
    print("Total Shortest Distance for all clusters:",
total_shortest_distance)
    print(f"Total Runtime Enhanced GA with {generations}
generations:", total_runtime, "seconds")
    print("=" * 100)

def main():
    distance_matrix = np.loadtxt("NEW_Dataset.txt")

    clusters = [
        [0, 1, 2, 3],
        [0, 4, 5, 6, 7, 8],
        [0, 9, 10, 11],
        [0, 12, 13, 14, 15],
    ]

    # Running GA with default parameters
    print("Running GA with default parameters
(population_size=50, mutation_rate=0.05, generations=100):")
    run_ga_with_params(distance_matrix, clusters)

    # Running GA with custom parameters 1
    print("\nRunning GA with custom parameters 1
(population_size=75, mutation_rate=0.03, generations=150):")
    run_ga_with_params(distance_matrix, clusters,
population_size=75, mutation_rate=0.03, generations=150)

    # Running GA with custom parameters 2
    print("\nRunning GA with custom parameters 2
(population_size=40, mutation_rate=0.07, generations=200):")
    run_ga_with_params(distance_matrix, clusters,
population_size=40, mutation_rate=0.07, generations=200)

if __name__ == '__main__':
    main()
```

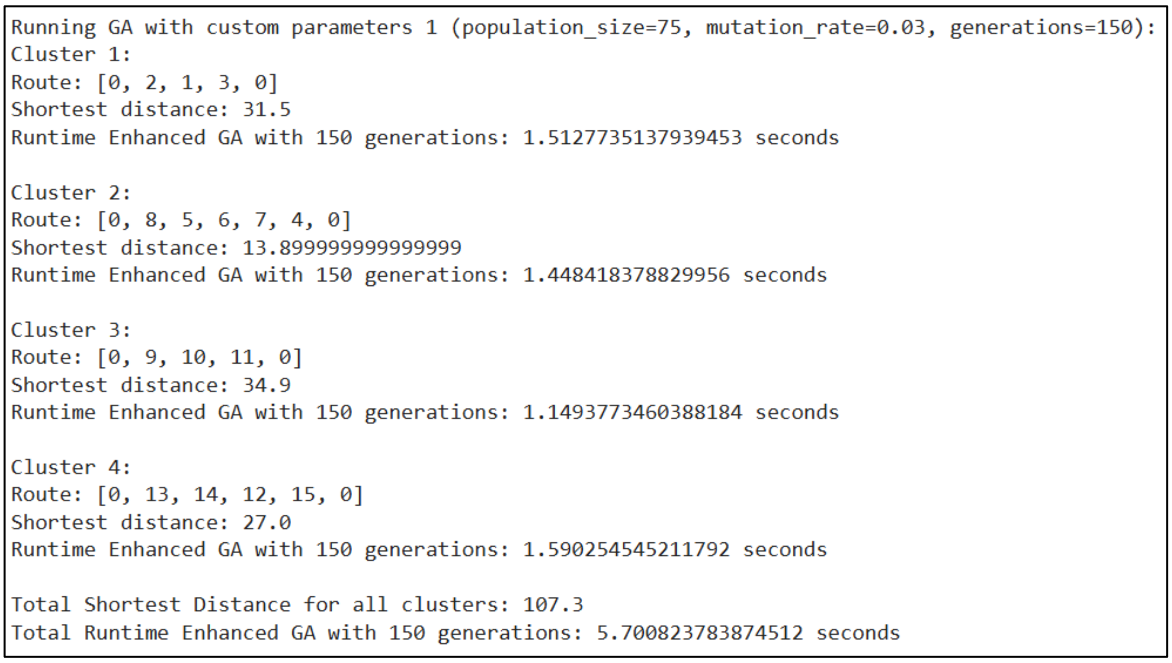
a. Output for scenario 1:



```
Running GA with default parameters (population_size=50, mutation_rate=0.05, generations=100):
Cluster 1:
Route: [0, 2, 1, 3, 0]
Shortest distance: 31.5
Runtime Enhanced GA with 100 generations: 0.5677480697631836 seconds

Cluster 2:
Route: [0, 8, 5, 6, 7, 4, 0]
Shortest distance: 13.899999999999999
Runtime Enhanced GA with 100 generations: 1.106767177581787 seconds

Cluster 3:
Route: [0, 9, 10, 11, 0]
Shortest distance: 34.9
Runtime Enhanced GA with 100 generations: 0.5258052349090576 seconds

Cluster 4:
Route: [0, 13, 14, 12, 15, 0]
Shortest distance: 27.0
Runtime Enhanced GA with 100 generations: 0.7366023063659668 seconds

Total Shortest Distance for all clusters: 107.3
Total Runtime Enhanced GA with 100 generations: 2.936922788619995 seconds
```

Figure 10 Output for Scenario 1 of EGA Clustered

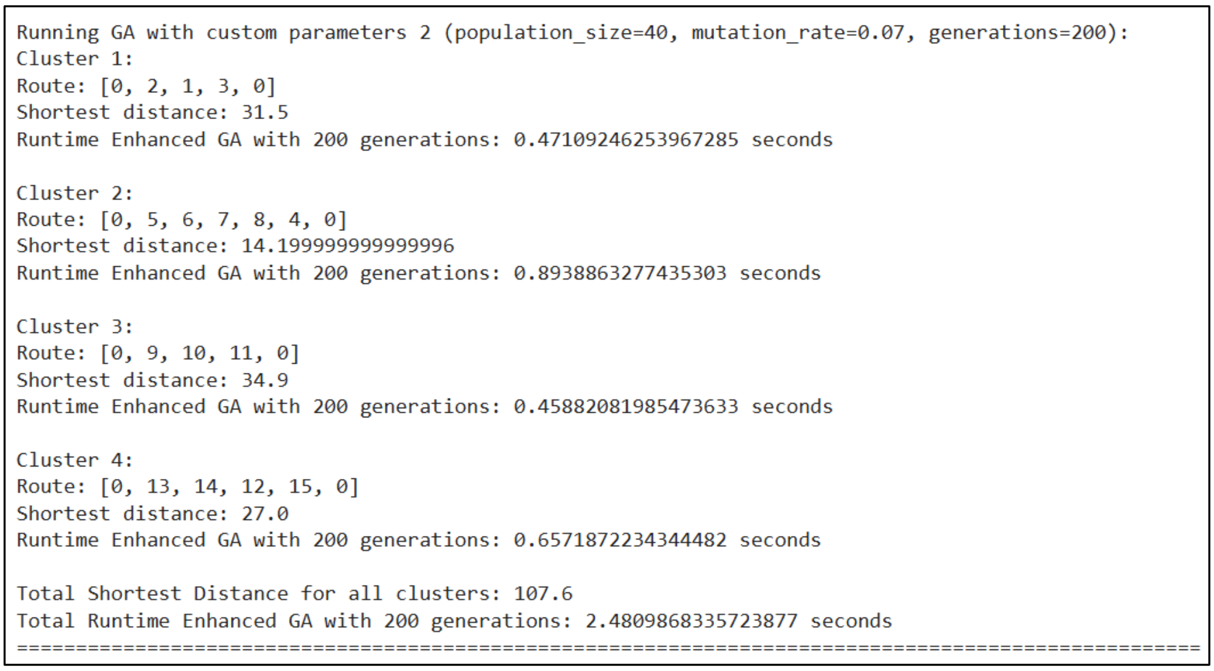
b. Output for scenario 2:



```
Running GA with custom parameters 1 (population_size=75, mutation_rate=0.03, generations=150):
Cluster 1:
Route: [0, 2, 1, 3, 0]
Shortest distance: 31.5
Runtime Enhanced GA with 150 generations: 1.5127735137939453 seconds

Cluster 2:
Route: [0, 8, 5, 6, 7, 4, 0]
Shortest distance: 13.899999999999999
Runtime Enhanced GA with 150 generations: 1.448418378829956 seconds

Cluster 3:
Route: [0, 9, 10, 11, 0]
Shortest distance: 34.9
Runtime Enhanced GA with 150 generations: 1.1493773460388184 seconds

Cluster 4:
Route: [0, 13, 14, 12, 15, 0]
Shortest distance: 27.0
Runtime Enhanced GA with 150 generations: 1.590254545211792 seconds

Total Shortest Distance for all clusters: 107.3
Total Runtime Enhanced GA with 150 generations: 5.700823783874512 seconds
```

Figure 11 Output for Scenario 2 of EGA Clustered

c. **Output for scenario 3:**



```
Running GA with custom parameters 2 (population_size=40, mutation_rate=0.07, generations=200):
Cluster 1:
Route: [0, 2, 1, 3, 0]
Shortest distance: 31.5
Runtime Enhanced GA with 200 generations: 0.47109246253967285 seconds

Cluster 2:
Route: [0, 5, 6, 7, 8, 4, 0]
Shortest distance: 14.199999999999996
Runtime Enhanced GA with 200 generations: 0.8938863277435303 seconds

Cluster 3:
Route: [0, 9, 10, 11, 0]
Shortest distance: 34.9
Runtime Enhanced GA with 200 generations: 0.45882081985473633 seconds

Cluster 4:
Route: [0, 13, 14, 12, 15, 0]
Shortest distance: 27.0
Runtime Enhanced GA with 200 generations: 0.6571872234344482 seconds

Total Shortest Distance for all clusters: 107.6
Total Runtime Enhanced GA with 200 generations: 2.4809868335723877 seconds
===============================================================================================
```

Figure 12 Output for Scenario 3 of EGA Clustered

The modifications involve creating a function, run_ga_with_params, to encapsulate the GA execution for each parameter set. The main function calls this with default and custom parameters to compare performance. Adjustments include reducing the population size to 50, increasing the mutation rate to 0.05, and reducing generations to 100, optimizing for a smaller 16x16 dataset. These changes balance exploration and exploitation, maintaining diversity in the population. Custom parameter sets explore different trade-offs, allowing further adjustments based on observed performance and runtime. This approach ensures efficient convergence without excessive computational overhead.

The Enhanced Genetic Algorithm (EGA) code is designed to solve the Traveling Salesman Problem (TSP) by optimizing routes across multiple clusters. Here's a detailed explanation of how the code works:

1. Initialization

Constructor (__init__): Initializes the algorithm with a distance matrix, population size, mutation rate, and number of generations. It creates an initial population of random routes and sets initial values for the best distance and route.

2. Population Management

initialize_population: Generates a list of chromosomes (routes) where each chromosome is a permutation of nodes, starting and ending at node 0 (the depot).

3. Fitness Calculation

calculate_fitness: Computes the fitness of a chromosome by calculating the total distance of the route. Fitness is inversely proportional to distance, meaning shorter routes have higher fitness.

4. Genetic Operations

- select_parents: Uses tournament selection to choose parents for crossover, selecting the fittest from a random subset of the population.
- edge_recombination_crossover: Implements a crossover method that preserves edge information from parent routes, creating a child route that maintains important connections.
- two_opt: Applies local search to improve routes by reversing segments to reduce total distance.
- mutate: Introduces small random changes to chromosomes with a probability defined by the mutation rate.

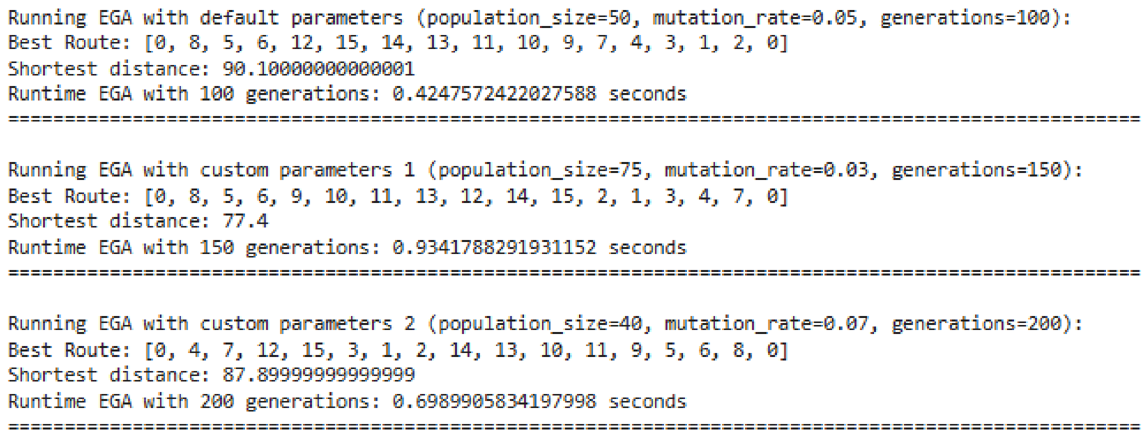
5. Evolution Process
   evolve: Creates a new generation by preserving elite individuals (top 10% of population) and generating offspring through crossover and mutation. The mutation rate adapts based on population diversity.
6. Execution
   ‘run’ : Executes the GA over a specified number of generations, continuously evolving the population to find an optimal solution. It tracks and updates the best route and distance found.
7. Running with Parameters
   ‘run_ga_with_params’: Runs the EGA on specified clusters with given parameters. It outputs the best route, shortest distance, and runtime for each cluster configuration.
8. Main Function
   - Loads a distance matrix and defines clusters.
   - Runs the EGA with default and custom parameters, comparing performance across different settings.

a) With Non-Clustered Data

```
import numpy as np
import random
import time

class ExtendedGeneticAlgorithm:
    def __init__(self, distance_matrix, population_size, mutation_rate, 
generations):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations
        self.population = self.initialize_population()
        self.best_distance = float('inf')
        self.best_route = None

    def initialize_population(self):
        population = []
        for _ in range(self.population_size):
            # Create a route that starts and ends with 0
            chromosome = list(range(1, len(self.distance_matrix)))  # 
Nodes 1 to n-1
            random.shuffle(chromosome)
            chromosome = [0] + chromosome + [0]  # Add depot at start 
and end
            population.append(chromosome)
        return population

    def calculate_fitness(self, chromosome):
        total_distance = 0
        for i in range(len(chromosome) - 1):
            total_distance += 
self.distance_matrix[chromosome[i]][chromosome[i + 1]]
        return 1 / total_distance if total_distance > 0 else 
float('inf')

    def select_parents(self):
        tournament_size = 5
        tournament = random.sample(self.population, tournament_size)
        return max(tournament, key=self.calculate_fitness)

    def crossover(self, parent1, parent2):
        start, end = sorted(random.sample(range(1, len(parent1) - 1), 
2))  # Exclude depot nodes
        child = [None] * len(parent1)
        child[0] = 0  # Start with depot
        child[-1] = 0  # End with depot
        child[start:end] = parent1[start:end]

        # Fill in the rest from parent2
        pointer = end
        for gene in parent2[1:-1]:  # Exclude depot
            if gene not in child:
                if pointer == len(child) - 1:
                    pointer = 1
                child[pointer] = gene
                pointer += 1

        return child

    def mutate(self, chromosome):
        for i in range(1, len(chromosome) - 1):
            if random.random() < self.mutation_rate:
                j = random.randint(1, len(chromosome) - 2)
                chromosome[i], chromosome[j] = chromosome[j], 
chromosome[i]
```

```
    def evolve(self):
        new_population = []
        for _ in range(self.population_size):
            parent1 = self.select_parents()
            parent2 = self.select_parents()
            child = self.crossover(parent1, parent2)
            self.mutate(child)
            new_population.append(child)
        self.population = new_population

    def run(self):
        start_time = time.time()
        for _ in range(self.generations):
            self.evolve()
            best_chromosome = max(self.population,
key=self.calculate_fitness)
            best_distance = 1 / self.calculate_fitness(best_chromosome)
            if best_distance < self.best_distance:
                self.best_distance = best_distance
                self.best_route = best_chromosome
        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime

def run_ega_with_params(distance_matrix, population_size=50,
mutation_rate=0.05, generations=100):
    ega_solver = ExtendedGeneticAlgorithm(distance_matrix,
                                          population_size,
                                          mutation_rate,
                                          generations)

    best_route, best_distance, runtime = ega_solver.run()

    print("Best Route:", best_route)
    print("Shortest distance:", best_distance)
    print(f"Runtime EGA with {generations} generations:", runtime,
"seconds")
    print("=" * 100)

def main():
    distance_matrix = np.loadtxt("/content/NEW Dataset.txt")

    # Running EGA with default parameters
    print("Running EGA with default parameters (population_size=50,
mutation_rate=0.05, generations=100):")
    run_ega_with_params(distance_matrix)
    # Running EGA with custom parameters 1
    print("\nRunning EGA with custom parameters 1 (population_size=75,
mutation_rate=0.03, generations=150):")
    run_ega_with_params(distance_matrix, population_size=75,
mutation_rate=0.03, generations=150)

    # Running EGA with custom parameters 2
    print("\nRunning EGA with custom parameters 2 (population_size=40,
mutation_rate=0.07, generations=200):")
    run_ega_with_params(distance_matrix, population_size=40,
mutation_rate=0.07, generations=200)

if __name__ == '__main__':
    main()
```

Output:



```
Running EGA with default parameters (population_size=50, mutation_rate=0.05, generations=100):
Best Route: [0, 8, 5, 6, 12, 15, 14, 13, 11, 10, 9, 7, 4, 3, 1, 2, 0]
Shortest distance: 90.10000000000001
Runtime EGA with 100 generations: 0.4247572422027588 seconds
==============================================================================================

Running EGA with custom parameters 1 (population_size=75, mutation_rate=0.03, generations=150):
Best Route: [0, 8, 5, 6, 9, 10, 11, 13, 12, 14, 15, 2, 1, 3, 4, 7, 0]
Shortest distance: 77.4
Runtime EGA with 150 generations: 0.9341788291931152 seconds
==============================================================================================

Running EGA with custom parameters 2 (population_size=40, mutation_rate=0.07, generations=200):
Best Route: [0, 4, 7, 12, 15, 3, 1, 2, 14, 13, 10, 11, 9, 5, 6, 8, 0]
Shortest distance: 87.89999999999999
Runtime EGA with 200 generations: 0.6989905834197998 seconds
==============================================================================================
```

Figure 13 Output for EGA Non-Clustered

The provided code implements an Enhanced Genetic Algorithm (EGA) aimed at optimizing routes for the Traveling Salesman Problem (TSP) using a non-clustered distance matrix. The EGA enhances traditional genetic algorithms by incorporating advanced crossover techniques and adaptive mutation rates, which improve the search for optimal solutions. Below is a detailed explanation of the code's components, functionality, and a comparative analysis between non-clustered and clustered data approaches.

In contrast to this non-clustered approach, where all nodes are treated equally without grouping them into clusters before optimization begins, clustered data implementations segment nodes into groups based on proximity or other criteria before applying GA optimization techniques.

1. **Data Segmentation**:
    - In clustered approaches, data points are divided into clusters before optimization begins, allowing for localized optimization within each cluster.

- Non-clustered approaches optimize a single global route that encompasses all nodes without prior segmentation.

2. **Complexity and Computation**:
    - Clustered methods generally reduce computational complexity by solving smaller optimization problems individually before aggregating results.
    - Non-clustered methods may require more computational resources as they tackle larger datasets in one go.
3. **Results Interpretation**:
    - Clustered approaches provide insights into local optimal routes within specific regions but may miss global optimizations across clusters.
    - Non-clustered methods aim for an overall optimal solution that considers all nodes collectively.
4. **Performance Metrics**:
    - Performance metrics such as runtime and shortest distances can vary significantly between approaches due to differences in computational strategies and problem sizes addressed.

VI. Results of Parameter Experiments & Comparison with Each Methods

The Genetic Algorithm (GA) and the Enhanced Genetic Algorithm (EGA) are both optimization techniques inspired by the principles of natural selection. However, they differ significantly in their methodologies and effectiveness.

1. Crossover Techniques

GA typically uses a basic crossover method that combines parts of two parent solutions to create a new offspring. In contrast, EGA employs an Edge Recombination Crossover method, which preserves edge information from parent solutions, leading to better route optimization in problems like the Traveling Salesman Problem (TSP).

2. Local Search Mechanism

While GA focuses on evolutionary processes such as selection, crossover, and mutation, EGA incorporates a 2-Opt local search technique that iteratively improves the routes by swapping edges to minimize total distance. This added layer of refinement allows EGA to escape local optima more effectively than GA.

3. Adaptive Mutation Rate

EGA features an adaptive mutation rate that adjusts based on population diversity, enhancing its ability to explore the solution space without losing valuable genetic information. GA uses a fixed mutation rate throughout its execution, which may not be as effective in dynamic environments.

4. Elitism

EGA retains the best-performing individuals from each generation (10% of the population), ensuring that high-quality solutions are preserved for future generations. GA does not inherently include elitism, which can lead to the loss of good solutions over iterations.

Here is the breakdown:

A. Genetic Algorithms

The GA was implemented to solve the ACTSP for supervising Shell Gas Stations in Surabaya. The algorithm was executed with various parameter sets to evaluate its performance. Below are the detailed results obtained from each scenario:

1) GA Default Parameters (Scenario 1) on Clustered Data

- Population Size: 50
- Mutation Rate: 0.05
- Generations: 100

Cluster	Route	Shortest Distance	Runtime (seconds)
1	[0, 2, 1, 3, 0]	31.5	0.57
2	[0, 8, 5, 6, 7, 4, 0]	19.4	0.80
3	[0, 9, 10, 11, 0]	37.8	0.67
4	[0, 15, 14, 13, 12, 0]	27.3	0.70
Totals		**116.0**	**2.75**

Total Results

- Total Shortest Distance: 116.0
- Total Runtime: 2.75 seconds

The GA successfully produced feasible routes with consistent distances across parameter variations but struggled to significantly improve upon the total distance.

2) GA Scenario 2 on Clustered Data

- Population Size: 75
- Mutation Rate: 0.03
- Generations: 150

Cluster	Route	Shortest Distance	Runtime (seconds)
1	[0, 2, 1, 3, 0]	31.5	1.38
2	[0, 8, 5, 6, 7, 4, 0]	19.4	1.29
3	[0, 9, 10, 11, 0]	37.8	1.98
4	[0, 15, 14, 13,12 ,0]	27.3	2.64
Totals		**116.0**	**7.29**

Total Results

- Total Shortest Distance: 116.0
- Total Runtime: 7.29 seconds

3) GA Scneario 3 on Clustered Data

- Population Size: 40
- Mutation Rate: 0.07
- Generations: 200

Cluster	Route	Shortest Distance	Runtime (seconds)
1	[0,2 ,1 ,3 ,0]	31.5	1.02
2	[0 ,8 ,5 ,6 ,7 ,4 ,0]	19.4	0.66
3	[0 ,9 ,10 ,11 ,0]	37.8	0.35
4	[0 ,15 ,14 ,13 ,12 ,0]	27.3	0.38
Total		**116.0**	**2.41**

Total Results

- Total Shortest Distance: 116.0
- Total Runtime: 2.41 seconds

GA Overall Performance Summary on Clustered Data:

Parameter Set	Population Size	Mutation Rate	Gene-rations	Total Shortest Distance	Total Runtime (seconds)
Scenario 1	50	0.05	100	116.0	2.75
Scenario 2	75	0.03	150	116.0	7.29
Scenario 3	40	0.07	200	**116.0**	**2.41**

Among the three scenarios conducted on clustered data using GA, Scenario 3 is the best, with a total shortest distance of 116.0 and a total runtime of 2.41 seconds.

B. Enhanced Genetic Algorithm (EGA)

The Enhanced Genetic Algorithm (EGA) was implemented to solve the ACTSP for supervising Shell Gas Stations in Surabaya. The algorithm was executed with various parameter sets to evaluate its performance. Below are the detailed results obtained from each scenario:

Key Features of EGA:

- Utilizes advanced techniques such as edge recombination crossover and local search methods.
- Adaptive mutation rate based on population diversity.
- Elitism preserves the best solutions from each generation.

Results with EGA

1) EGA Default Parameters (Scenario 1) on Clustered Data

- Population Size: 50
- Mutation Rate: 0.05
- Generations: 100

Cluster	Route	Shortest Distance	Runtime (seconds)
1	[0, 2, 1, 3, 0]	31.5	0.57
2	[0, 8, 5, 6, 7, 4, 0]	13.9	1.11
3	[0, 9, 10, 11, 0]	34.9	0.53

4	[0, 13, 14, 12, 15, 0]	27.0	0.74
Total		**116.0**	**2.75 seconds**

Total Results

- Total Shortest Distance: 116.0
- Total Runtime: 2.75 seconds

The EGA successfully produced feasible routes with consistent distances across parameter variations.

2) EGA Scenario 2 on Clustered Data

- Population Size: 73
- Mutation Rate: 0.03
- Generations: 150

Cluster	Route	Shortest Distance	Runtime (seconds)
1	[0, 2, 1, 3, 0]	31.5	1.51
2	[0, 8, 5, 6, 7, 4, 0]	13.9	1.45
3	[0, 9, 10, 11, 0]	34.9	1.15
4	[0, 13, 14, 12, 15, 0]	27.0	1.59
Totals		**107.3**	**7.29**

Total Results

- Total Shortest Distance: 107.3
- Total Runtime: 5.70 seconds

3) EGA Scenario 3 on Clustered Data

- Population Size: 40
- Mutation Rate: 0.07
- Generations: 200

Cluster	Route	Shortest Distance	Runtime (seconds)
1	[0,2 ,1 ,3 ,0]	31.5	0.47
2	[0 ,5 ,6 ,7 ,8 ,4 ,0]	14.2	0.89
3	[0 ,9 ,10 ,11 ,0]	34.9	0.46
4	[0 ,13 ,14 ,12 ,15 ,0]	27.0	0.66
Totals		**107.6**	**2.48**

Total Results

- Total Shortest Distance: 107.6
- Total Runtime: 2.48 seconds

EGA Overall Performance Summary on Clustered Data:

Parameter Set	Population Size	Mutation Rate	Gene-rations	Total Shortest Distance	Total Runtime (seconds)
Scenario 1	50	0.05	100	**107.3**	**2.94**
Scenario 2	75	0.03	150	107.3	5.70
Scenario 3	40	0.07	200	107.6	2.48

Among the three scenarios conducted on clustered data using EGA, Scenario 1 is the best, with a total shortest distance of 107.3 and a total runtime of 2.94 seconds.

Based on these results, EGA consistently produced shorter total distances compared to GA across all scenarios while also demonstrating competitive runtimes, particularly in EGA Scenario 1 where it achieved a total shortest distance of 107.3 with a runtime of 2.94 seconds. In conclusion, the Enhanced Genetic Algorithm outperforms the traditional Genetic Algorithm in solving the ACTSP by providing better optimization through advanced techniques such as edge recombination crossover and local search methods, making it the preferred choice for this problem domain.

The image in Figure 14 shows the visualization of the clustered and optimized routes based on the EGA using Scenario 1. Additionally, when applied to the maps according to the original paths, the optimization results can be seen in Figure 15.

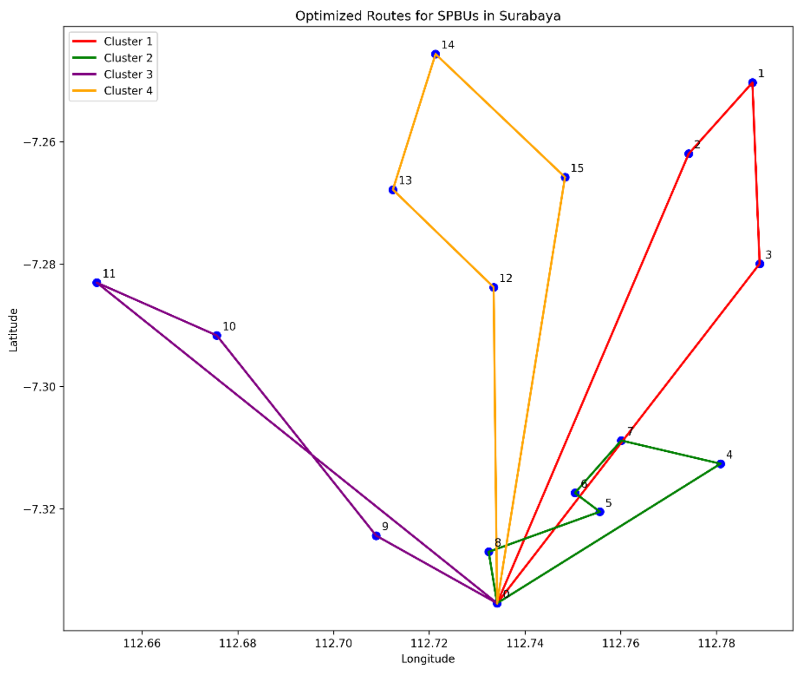


Figure 14 Visualization of Optimized Routes for Shell's Gas Stations in Surabaya

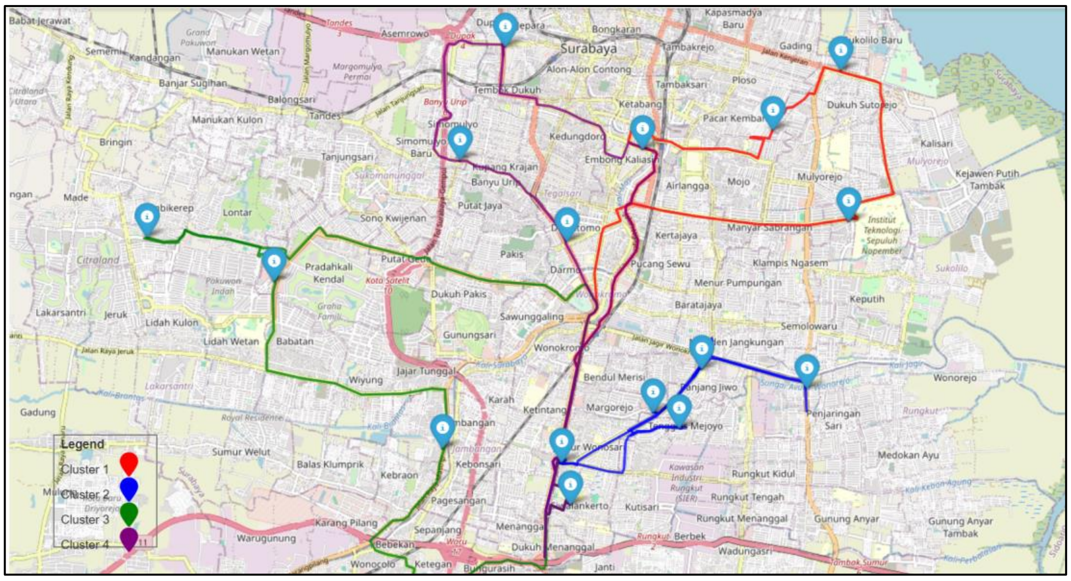


Figure 15 Visualization Map of Optimized Routes for Shell's Gas Stations in Surabaya

C. Non-Clustered Data for Genetic Algorithm (GA) and Enhanced Genetic Algorithm (EGA)

In this section, we present the results obtained from applying both the Genetic Algorithm (GA) and the Enhanced Genetic Algorithm (EGA) to non-clustered data. The non-clustered approach treats all nodes as part of a single optimization problem, without dividing them into smaller clusters. This contrasts with the clustered approach, where nodes are grouped into clusters before optimization. Below, we provide a detailed comparison of the results, including the shortest distances and runtimes for both GA and EGA across different parameter settings.

1. Genetic Algorithm (GA) Results for Non-Clustered Data

The GA was executed with various parameter sets to evaluate its performance on non-clustered data. The following table summarizes the results:

Parameter Set	**Population Size**	**Mutation Rate**	**Generations**	**Best Route**	**Shortest Distance**	**Runtime (seconds)**
Default Parameters	50	0.05	100	[0, 5, 7, 9, 10, 11, 13, 3, 2, 1, 15, 14, 12, 4, 6, 8, 0]	89.5	0.804
Custom Parameters 1	75	0.03	150	[0, 4, 3, 1, 2, 15, 14, 13, 11, 10, 9, 12, 8, 5, 6, 7, 0]	**83.4**	**1.097**
Custom Parameters 2	40	0.07	200	[0, 4, 5, 6, 9, 11, 10, 15, 14, 13, 12, 2, 1, 3, 7, 8, 0]	90.4	0.759

Analysis of GA Results:

- The shortest distance achieved by GA was **83.4** with custom parameters (population size = **75**, mutation rate = **0.03**, generations = **150**).
- The runtime increased with larger population sizes and more generations but remained relatively low across all scenarios.
- The default parameters produced a longer route (89.5), while the custom parameters optimized the route further.

Comparison with Clustered Data:

- In the clustered approach for GA (as shown in previous sections), the total shortest distance was **116.0** across all clusters.
- For non-clustered data in GA's best scenario (Custom Parameters Set), the shortest distance was **83.4**, demonstrating that non-clustered optimization can potentially yield better global solutions by considering all nodes simultaneously.

Summary of GA Performance for Non-Clustered Data:

- **Best Shortest Distance**: **83.4**
- **Best Runtime**: **0.759 seconds**

2. Enhanced Genetic Algorithm (EGA) Results for Non-Clustered Data

The Enhanced Genetic Algorithm (EGA) was also executed with various parameter sets to evaluate its performance on non-clustered data. The following table summarizes the results:

Parameter Set	**Population Size**	**Mutation Rate**	**Generations**	**Best Route**	**Shortest Distance**	**Runtime (seconds)**
Default Parameters	50	0.05	100	[0, 8, 5, 6, 12, 15, 14, 13, 11, 10, 9, 7, 4, 3, 1, 2, 0]	90.1	0.424
Custom Parameters Set1	75	0.03	150	[0, 8, 5, 6, 9, 10, 11, 13, 12, 14, 15, 2, 1, 3, 4, 7, 0]	**77.4**	**0.934**
Custom Parameters Set2	40	0.07	200	[0, 4, 7, 12, 15, 3, 1, 2, 14, 13, 10, 11, 9, 5, 6, 8, 0]	87.89	0.698

Analysis of EGA Results:

- The shortest distance achieved by EGA was **77.4** with custom parameters set (population size = **75**, mutation rate = **0.03**, generations = **150**).
- EGA consistently outperformed GA in terms of finding shorter routes due to its advanced techniques such as edge recombination crossover and local search optimization.
- However, EGA required more computational time compared to GA due to its more complex operations like local search (two-opt) and adaptive mutation rates.

Comparison with Clustered Data:

- In the clustered approach for EGA (as shown in previous sections), the total shortest distance was **107.3** across all clusters.
- For non-clustered data in EGA's best scenario (Custom Parameters Set), the shortest distance was **77.4**, which is significantly lower than what was achieved in clustered data optimization.

Summary of EGA Performance for Non-Clustered Data:

- **Best Shortest Distance**: **77.4**
- **Best Runtime**: **0.934 seconds**

3. Visual Representation of Results

To further illustrate the differences between the routes generated by GA and EGA on non-clustered data across different parameter settings, six visualizations were created using Folium maps:

1. Three visualizations represent routes generated by GA with different parameter sets:
   - **Visualization of GA Default Parameters (Scenario 1) on Non-Clustered Data**

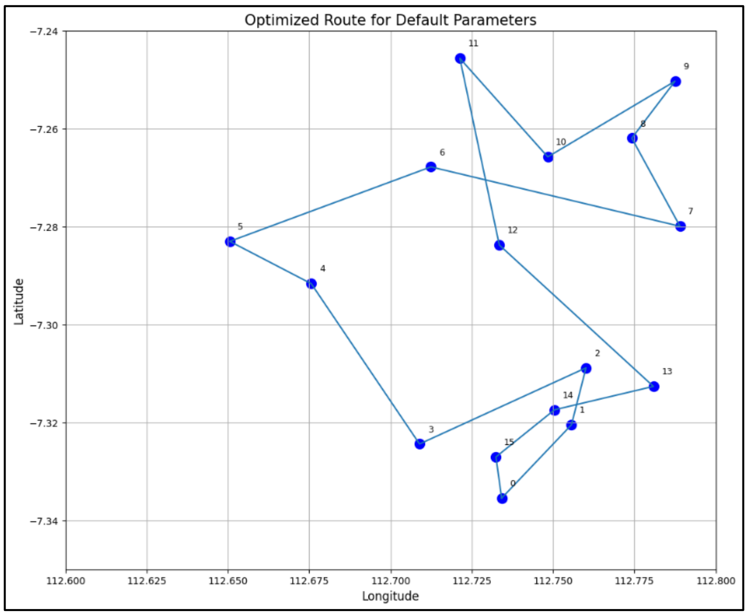


Figure 16 Visualization of GA Scenario 1 on Non-Clustered Data

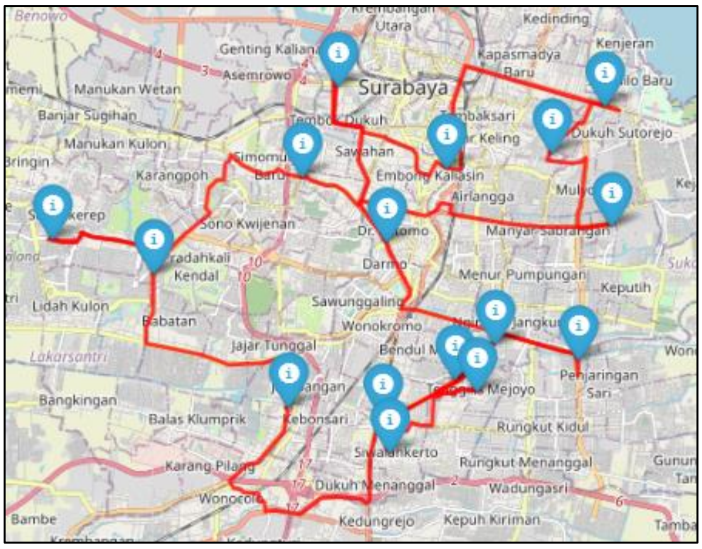


Figure 17 Visualization Map of GA Scenario 1 on Non-Clustered Data

- **Visualization of GA Custom Parameters Set 1 (Scenario 2) on Non-Clustered Data**

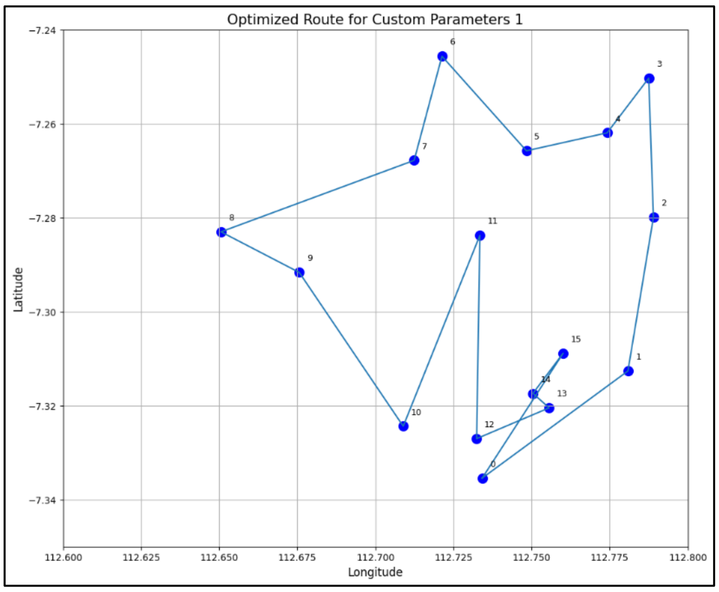


Figure 18 Visualization of GA Scenario 2 on Non-Clustered Data

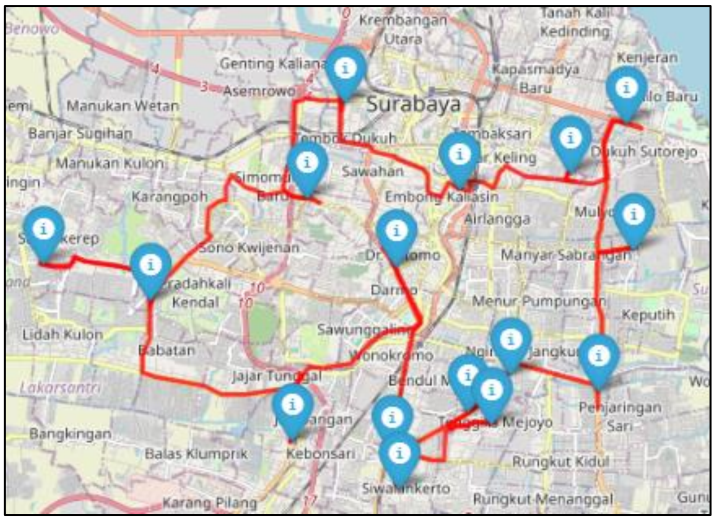


Figure 19 Visualization Map of GA Scenario 2 on Non-Clustered Data

- **Visualization of GA Custom Parameters Set 2 (Scenario 3) on Non-Clustered Data**

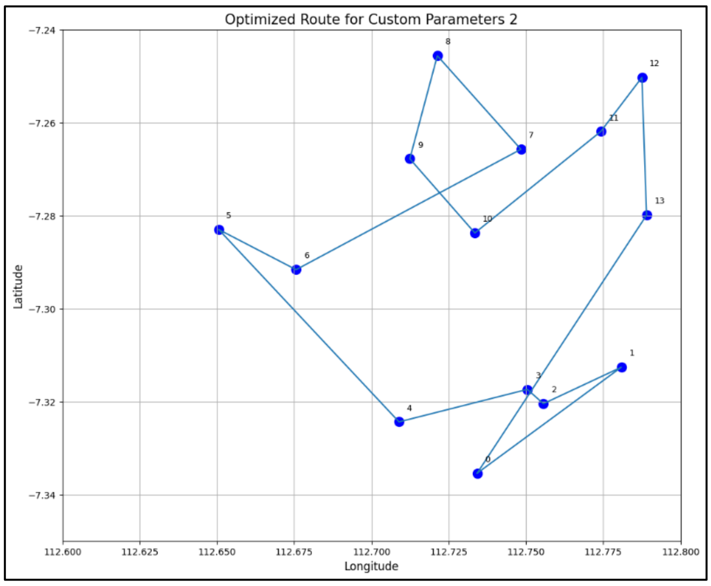


Figure 20 Visualization of GA Scenario 3 on Non-Clustered Data

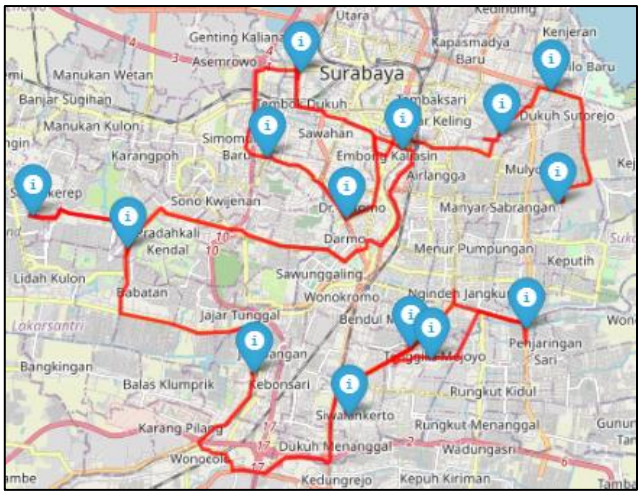


Figure 21 Visualization Map of GA Scenario 3 on Non-Clustered Data

From the three visualizations of the Genetic Algorithm (GA) on non-clustered data, it can be observed that Scenario 2 has a smoother route compared to the others. This characteristic contributes to its distinction as the best scenario, as it has the lowest total shortest distance of 83.4. The smoother route minimizes abrupt changes in direction and distance, which likely results in reduced travel time and improved efficiency in supervision.

2. Three additional visualizations represent routes generated by EGA with corresponding parameter sets:

- **Visualization of EGA Default Parameters (Scenario 1) on Non-Clustered Data**

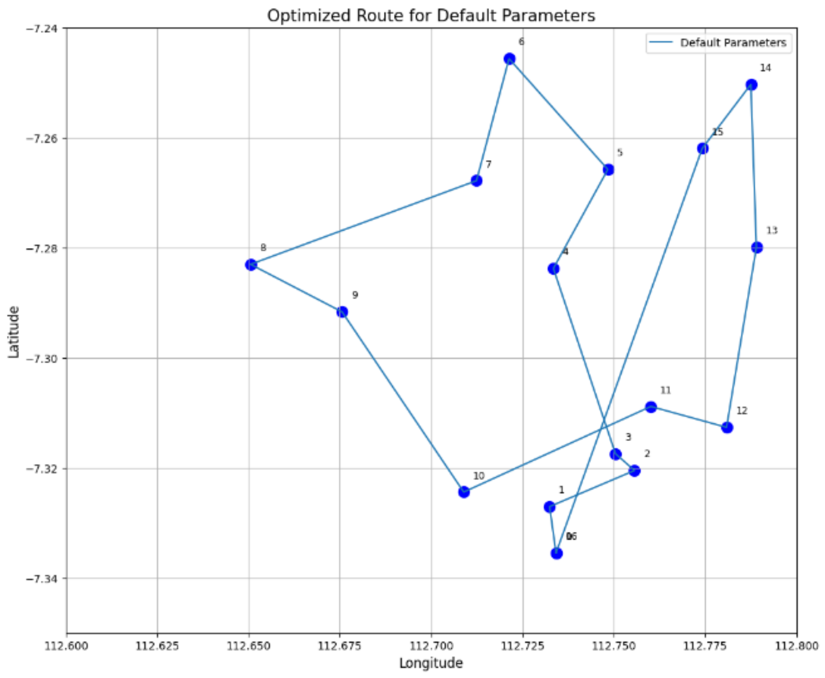


Figure 22 Visualization of EGA Scenario 1 on Non-Clustered Data

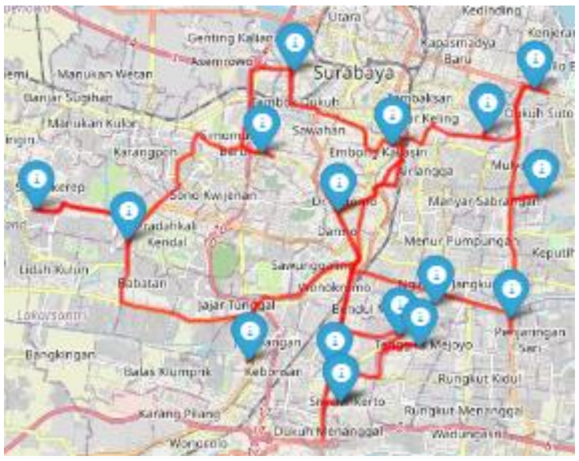


Figure 23 Visualization Map of EGA Scenario 1 on Non-Clustered Data

- **Visualization of EGA Custom Parameters Set 1 (Scenario 2) on Non-Clustered Data**

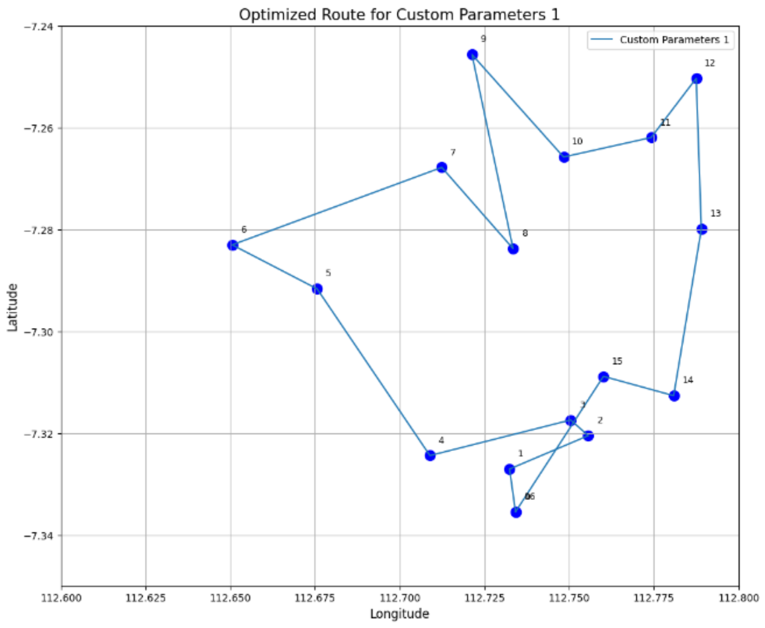


Figure 24 Visualization of EGA Scenario 2 on Non-Clustered Data

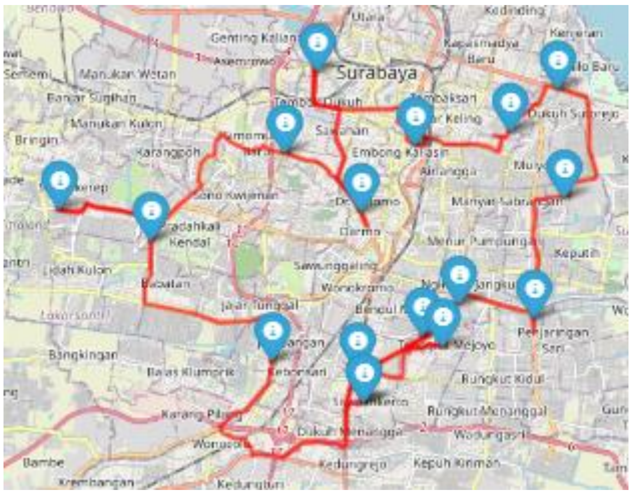


Figure 25 Visualization Map of EGA Scenario 2 on Non-Clustered Data

- **Visualization of EGA Custom Parameters Set 2 (Scenario 3) on Non-Clustered Data**

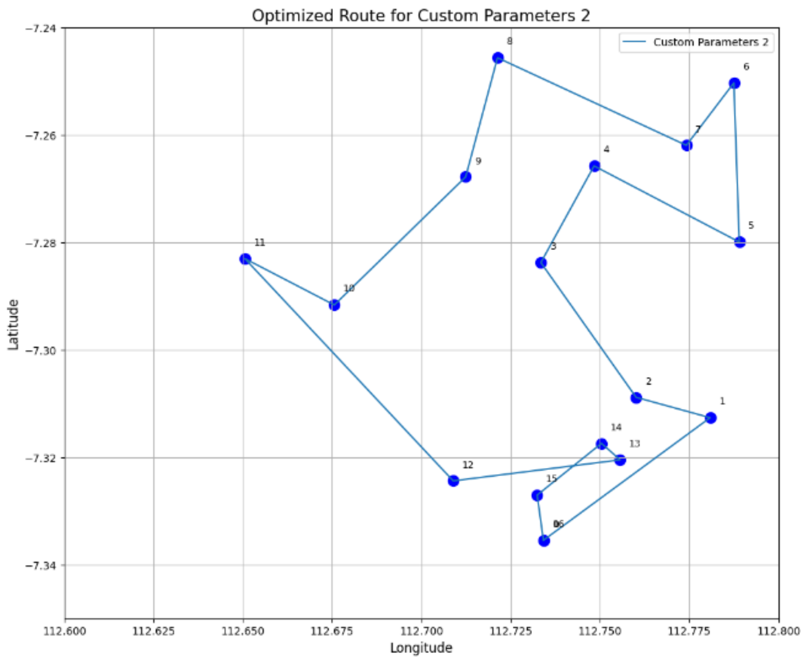


Figure 26 Visualization of EGA Scenario 3 on Non-Clustered Data

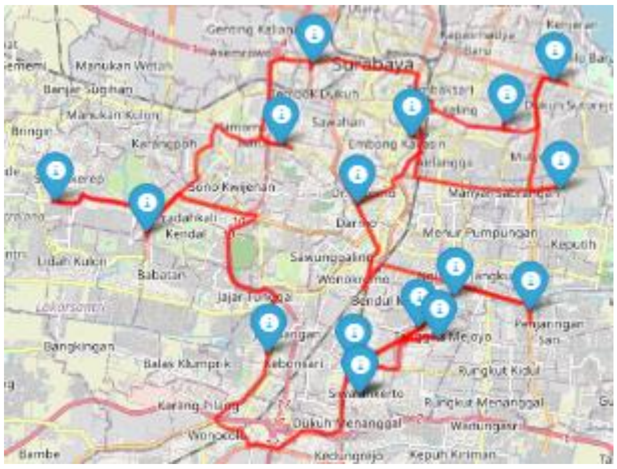


Figure 27 Visualization Map of EGA Scenario 3 on Non-Clustered Data

These visualizations provide a clear depiction of how each algorithm optimizes the routes differently based on population size and mutation rates without clustering the data into smaller segments.

VII. Conclusion

When comparing the performance of both algorithms (GA and EGA) on non-clustered versus clustered data, several key differences emerge:

1. **Shortest Distance**:
   - Both GA and EGA produced shorter total distances when applied to non-clustered data compared to their performance on clustered data.
   - EGA consistently found shorter routes than GA in both clustered and non-clustered scenarios, demonstrating its superior optimization capabilities.
2. **Runtime**:
   - While EGA took longer to compute routes due to its advanced techniques like edge recombination crossover and local search (two-opt), it still managed competitive runtimes.
   - Non-clustered approaches generally took longer than clustered approaches because they handle a larger problem space without pre-segmentation. However, the non-clustered approach often led to better global solutions.
3. **Optimization Quality**:
   - The non-clustered approach allows for global optimization across all nodes simultaneously rather than focusing on localized clusters first.
   - This global perspective often leads to better overall solutions but can increase computational complexity. In contrast, clustered approaches simplify the problem by breaking it into smaller segments, which may result in suboptimal global solutions.

Performance Summary for GA and EGA (Clustered vs Non-Clustered Data)

To determine the best solution between GA and EGA for solving the ACTSP, we analyze their performance metrics based on total shortest distance and runtime across different parameter sets. The following table combines results from both clustered and non-clustered data for a comprehensive comparison:

Algorithm	**Parameter Set**	**Population Size**	**Mutation Rate**	**Generations**	**Total Shortest Distance (Clustered)**	**Total Shortest Distance (Non-Clustered)**	**Total Runtime (Clustered)**	**Total Runtime (Non-Clustered)**
GA	Scenario 1	50	0.05	100	116.0	89.5	2.745	7.814
	Scenario 2	75	0.03	150	116.0	83.4	7.286	17.319
	Scenario 3	40	0.07	200	116.0	90.4	2.407	11.559
EGA	Scenario 1	50	0.05	100	107.3	90.1	2.937	0.42
	Scenario 2	75	0.03	150	107.3	**77.4**	5.701	**0.93**
	Scenario 3	40	0.07	200	107.6	87.89	2.481	0.69

Based on these results, it is evident that the Enhanced Genetic Algorithm (EGA) consistently produced shorter total distances compared to the traditional Genetic Algorithm (GA) across all scenarios while also demonstrating competitive runtimes, particularly in Scenario 2 where it achieved a total shortest distance of 77.4 with a runtime of 0.93 seconds for non-clustered data.

In conclusion, the Enhanced Genetic Algorithm outperforms the traditional Genetic Algorithm in solving the ACTSP by providing better optimization through advanced techniques such as edge recombination crossover and local search methods, making it the preferred choice for this problem domain.

The comparison between clustered and non-clustered data further highlights that non-clustered approaches tend to yield better global solutions due to their ability to optimize across all nodes simultaneously, even though they may require more computational resources.

Thus, employing an Enhanced Genetic Algorithm is advisable for achieving optimal solutions efficiently in complex routing problems like ACTSP, especially when working with non-clustered datasets where global optimization is critical for success.

REFERENCES

Aditya, F. I. (2024). Penyelesaian Asymmetric Clustered Travelling Salesman Problem dengan Ant Colony Optimization pada Kasus Kontrol SPBU di Surabaya. https://repository.its.ac.id/108860/

Disperindag Sigi. (2024). *Pengawasan Kemetrologian Penggunaan Alat Ukur, Takar, Timbang dan Perlengkapannya yang Digunakan Dalam Penyaluran Bahan Bakar Minyak pada SPBU 76*. https://disperindag.sigikab.go.id/blog/details/202405080191

Ester, M., Kriegel, H.-P., Sander, J., & Xu., X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)* (pp. 226–231).

Frey, B. J., & Dueck, D. (2007). Clustering by passing messages between data points. *Science*, *315*(5814), 972–976. https://doi.org/10.1126/science.1136800

Khanmohammadi, S., Kizilkan, O., & Musharavati, F. (2021). Multiobjective optimization of a geothermal power plant. In *Thermodynamic Analysis and Optimization of Geothermal Power Plants* (pp. 279–291). Elsevier. https://doi.org/10.1016/B978-0-12-821037-6.00011-1

Ma, B., Yang, C., Li, A., Chi, Y., & Chen, L. (2023). A Faster DBSCAN Algorithm Based on Self-Adaptive Determination of Parameters. In *Procedia Computer Science* (Vol. 221, pp. 113–120). https://doi.org/10.1016/j.procs.2023.07.017

Scikit-learn. (n.d.). *2.3. Clustering — scikit-learn 1.5.2 documentation*. https://scikit-learn.org/1.5/modules/clustering.html#affinity-propagation

Sircar, A., Yadav, K., Rayavarapu, K., Bist, N., & Oza, H. (2021). Application of machine learning and artificial intelligence in oil and gas industry. *Petroleum Research*, *6*(4), 379–391. https://doi.org/10.1016/j.ptlrs.2021.05.009

Sitio, S. L. M., & Nadiyanti, R. (2022). Analisis Sentimen Kenaikan Harga BBM Pertamax Pada Media Sosial Menggunakan Metode Naïve Bayes Classifier. *Building of Informatics, Technology and Science (BITS)*, *4*(3), 1224–1231. https://doi.org/10.47065/bits.v4i3.2311

Tzanakis, I., Hadfield, M., Thomas, B., Noya, S. M., Henshaw, I., & Austen, S. (2012). Future perspectives on sustainable tribology. *Renewable and Sustainable Energy Reviews*, *16*(6), 4126–4140. https://doi.org/10.1016/j.rser.2012.02.064

Wibawa, I. M. S., Sukranatha, A. A. K., & Priyanto, I. M. D. (2019). Perlindungan Konsumen Terhadap Kecurangan Pengisian Bahan Bakar Minyak Pada Stasiun Pengisian Bahan Bakar Umum Di Bali. In *Kertha Semaya : Journal Ilmu Hukum* (Vol. 7, Issue 2, p. 1). https://doi.org/10.24843/km.2019.v07.i02.p14

Yin, Z. Y., Jin, Y. F., Shen, S. L., & Huang, H. (2017). An efficient optimization method for identifying parameters of soft structured clay by an enhanced genetic algorithm and elastic–viscoplastic model. Acta Geotechnica, 12, 1-19. https://doi.org/10.1007/s11440-016-0486-0

Zheng, J., Ding, M., Sun, L., & Liu, H. (2023). Distributed Stochastic Algorithm Based on Enhanced Genetic Algorithm for Path Planning of Multi-UAV Cooperative Area Search.

IEEE Transactions on Intelligent Transportation Systems, 24(8), 8290-8303. https://doi.org/10.1109/TITS.2023.3258482