Final Project Soft Computing (Q)

Optimizing the Clustered Traveling Salesman Problem Using Metaheuristic Methods: A Case Study of Pertamina and 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 a critical driver of human activities and plays a pivotal role in shaping economic, social, and environmental development (Tzanakis et al., 2012). Among the various energy sources, fuel oil is one of the most essential commodities, supporting daily activities and fulfilling the community's energy needs (Sitio & Nadiyanti, 2022). According to Indonesia's Law No. 22 of 2001 on Oil and Natural Gas, the government holds the authority to regulate and oversee activities in the energy sector, including the distribution of fuel oil managed by Gas Stations.

Fuel dispensing fraud at gas stations is a common issue often reported by consumers (Wibawa et al., 2019). At the regional level, the Department of Industry and Trade (Dinas Perindustrian dan Perdagangan or Disperindag) is tasked with overseeing fuel distribution to ensure compliance with distribution standards, safety regulations, and pricing policies. This includes conducting routine inspections of nozzle calibration, mandated under Law No. 2 of 1981 on Legal Metrology, to guarantee accurate fuel measurement during transactions (Disperindag Sigi, 2024). Proper nozzle checks ensure fair service and maintain public trust in fuel distribution, preventing fraud and safeguarding consumer rights.

Previous research by Aditya (2024) addressed the Asymmetric Clustered Traveling Salesman Problem (ACTSP) to optimize supervisory routes for 70 Pertamina gas stations in Surabaya. Using Equal-Size Spectral Clustering, the stations were divided into eight clusters, and Ant Colony Optimization (ACO) enhanced by Opposition-Based Learning (OBL) was applied. The study successfully minimized the total travel distance to 249.3 km by leveraging optimal parameters, including a beta value of 5 and 400 iterations. This demonstrated the effectiveness of ACO with OBL modifications in solving ACTSP, particularly for complex distribution scenarios.

This project builds on previous research addressing the Asymmetric Clustered Traveling Salesman Problem (ACTSP) for supervising 70 Pertamina gas stations in Surabaya by extending its scope to include 15 Shell gas stations, resulting in 85 locations. The inclusion of Shell gas stations is essential, as these stations are equally significant in ensuring fair and accurate fuel distribution. By incorporating both Pertamina and Shell stations, this research aims to enhance Disperindag's oversight capabilities, providing a more comprehensive framework for effective fuel monitoring. Utilizing three metaheuristic approaches—Genetic Algorithms (GA), Ant Colony Optimization (ACO), and Simulated Annealing (SA)—the study seeks to optimize supervisory routes, minimizing travel time and distance. Accurate distance data from Google Maps ensures realistic and reliable route planning, addressing the complexities of ACTSP.

The optimization not only aims to streamline inspection activities but also delivers broader benefits, such as reducing operational costs, saving time, and simplifying the planning of supervisory tasks. By improving the efficiency and effectiveness of fuel station monitoring, this approach strengthens Disperindag's ability to uphold legal standards, prevent fraud, and provide better services to the public. Moreover, this project serves as a foundation for future advancements in optimization techniques, contributing to sustainable and accountable regulatory practices in the energy sector while fostering greater public trust.

II. Problem Description

The distribution of fuel oil in Surabaya, including at both Pertamina and Shell Gas Stations, requires comprehensive oversight to ensure compliance with distribution standards,

safety regulations, and pricing policies. This responsibility falls to Disperindag, which conducts periodic inspections, primarily focusing on nozzle checks to verify the accurate calibration of fuel dispensers. These checks, mandated by Law No. 2 of 1981 concerning Legal Metrology, are critical to ensuring that consumers receive the correct fuel volume and to mitigating the risk of fraud. However, the current process faces significant challenges: with a total of 85 Gas Stations (70 Pertamina and 15 Shell), route efficiency is crucial to complete inspections within limited time and resource constraints. Inefficient planning could lead to increased travel time, higher operational costs, and incomplete oversight, leaving room for potential discrepancies in fuel measurements.

To address these challenges, this study explores the use of the ACTSP model and metaheuristic approaches, including Genetic Algorithms (GA), Ant Colony Optimization (ACO), and Simulated Annealing (SA), to optimize supervisory routes. By minimizing travel time and streamlining inspection processes, this approach aims to improve oversight efficiency, allowing officers to cover all Gas Stations effectively. The integration of accurate distance data from Google Maps ensures reliable route optimization, enhancing inspection consistency and reducing operational costs. Ultimately, this solution is expected to uphold consumer rights by maintaining fuel measurement accuracy and quality standards while providing a foundation for further advancements in regulatory practices.

III. Data

In this project, several data sets were collected to address the issues outlined earlier. The first set involved gathering information on 85 Gas Stations in Surabaya, comprising 70 Pertamina Gas Stations and 15 Shell Gas Stations, along with the Disperindag Surabaya office. The latitude and longitude coordinates of these locations were sourced from Google Maps. This data is essential for route visualization, enabling the optimization of supervision routes for nozzle checks and ensuring efficient planning for officers tasked with overseeing fuel distribution.

Node	Nama Node (Visualisasi)	Alamat	Link	lat	long
0	Disperindag Surabaya	Disperindag Surabaya	Link	-7.2561351	112.7375274
1	Pertamina 1	JL. DUPAK RUKUN 72AB	Link	-7.24931	112.71059
2	Pertamina 2	JL. KUPANG JAYA NO.2A	Link	-7.27527	112.70745
3	Pertamina 3	LOKASI DESA TAMBAK LANGON TANDES	Link	-7.22875	112.67660
4	Pertamina 4	JL. TAMBAK OSOWILANGON SEMEMI	Link	-7.21345	112.65328
5	Pertamina 5	JL. SIMOPOMAHAN 23 P	Link	-7.26599	112.70664
6	Pertamina 6	JL. RAYA PAKAL 104 KEC.BENOWO	Link	-7.23658	112.61863

7	Pertamina 7	JL. TANJUNGSARI	Link	-7.25944	112.69362
8	Pertamina 8	JL. RAYA MARGOMULYO NO. 33	Link	-7.24384	112.68200
9	Pertamina 9	JL. RAYA MENGANTI 772	Link	-7.30797	112.67200
10	Pertamina 10	JL.MAY.JEND. HR MUHAMMAD 4850	Link	-7.2854837	112.6975433
11	Pertamina 11	JL.RY MENGANTI LIDAH WETAN 139	Link	-7.3068536	112.6635302
12	Pertamina 12	JL. RAYA MARGOMULYO NO. 30	Link	-7.24545	112.68331
13	Pertamina 13	JL.CITRA RAYA BOULEVARD B TX1	Link	-7.30029	112.66218
14	Pertamina 14	JL. HR. MUHAMMAD NO.113 PRADAH KALI	Link	-7.28422	112.69117
15	Pertamina 15	JL. ARJUNO	Link	-7.25774	112.72763
16	Pertamina 16	JL. A YANI 204	Link	-7.3339941	112.7297315
17	Pertamina 17	JL. SULAWESI NO. 8	Link	-7.27686	112.74617
18	Pertamina 18	JL. ARJUNO NO.80 (ARJUNOBROMO)	Link	-7.26275	112.72635
19	Pertamina 19	JL. MAYJEN SUNGKONO NO.47	Link	-7.29159	112.72329
20	Pertamina 20	JL. JAGIR WONOKROMO	Link	-7.30558	112.76019
21	Pertamina 21	JL. MASTRIP 237 KARANGPILANG	Link	-7.3163335	112.7094794
22	Pertamina 22	JL. MASTRIP KEDURUS	Link	-7.3390711	112.6997538
23	Pertamina 23	JL. GUNUNGSARI NO. 53	Link	-7.30672	112.72288
24	Pertamina 24	JL. RAYA MENGANTI 250	Link	-7.31282	112.69860
25	Pertamina 25	JL. JOYOBOYO	Link	-7.29733	112.73158
26	Pertamina 26	JL. RAYA NGAGEL 185 KEC. WONOKROMO	Link	-7.29651	112.74243
27	Pertamina 27	JL. TIDAR NO. 127141	Link	-7.25619	112.71920

28	Pertamina 28	JL. KEBONSARI TENGAH NO. 7065	Link	-7.3279698	112.7109819
29	Pertamina 29	JL. RAYA MARGOREJO INDAH 146A	Link	-7.31802	112.74552
30	Pertamina 30	JL. BALAS KEL. BALAS KLUMPRIK	Link	-7.33240	112.69190
31	Pertamina 31	JL. MULYOSARI 388 KEC.SUKOLILO	Link	-7.26868	112.79783
32	Pertamina 32	JL.KAPAS KRAMPUNG NO.99	Link	-7.24991	112.76470
33	Pertamina 33	JL. RAYA DHARMAHUSADA 114116	Link	-7.26500	112.76854
34	Pertamina 34	JL. KENJERAN NO.661	Link	-7.25147	112.79172
35	Pertamina 35	JL. ARIEF RACHMAN HKM SUKOLILO	Link	-7.29000	112.78439
36	Pertamina 36	JL. SEMOLOWARU NO.276	Link	-7.29982	112.78181
37	Pertamina 37	JL. PANDUGO NO.84 RUNGKUT	Link	-7.32174	112.78537
38	Pertamina 38	JL. SUMATRA NO.25 29	Link	-7.2684026	112.7510068
39	Pertamina 39	JL. NGAGEL JAYA UTARA NO.91	Link	-7.28853	112.75848
40	Pertamina 40	JL. PLOSO BARU NO. 183185	Link	-7.25827	112.77391
41	Pertamina 41	JL. RAYA RUNGKUT MAPAN	Link	-7.33451	112.77506
42	Pertamina 42	JL. KERTAJAYA JL. DHARMAWANGSA	Link	-7.27814	112.75567
43	Pertamina 43	JL. MANYAR KERTOARJO	Link	-7.28081	112.77163
44	Pertamina 44	JL. RUNGKUT INDUSTRI RY	Link	-7.33055	112.75717
45	Pertamina 45	JL. RAYA RUNGKUT	Link	-7.32150	112.76995
46	Pertamina 46	JL. MEDOKAN AYU 20 RUNGKUT	Link	-7.3334958	112.7938676

47	Pertamina 47	JL. RAYA PANJANG JIWO NO.54	Link	-7.3085502	112.7680038
48	Pertamina 48	JL. KLAMPIS JAYA NO.19	Link	-7.28642	112.77568
49	Pertamina 49	NGINDEN SEMOLO 80	Link	-7.30038	112.76952
50	Pertamina 50	JL. RAYA NGINDEN KEC. GUBENG	Link	-7.29987	112.76184
51	Pertamina 51	JL. RAYA MERR KALIJUDAN	Link	-7.25818	112.78272
52	Pertamina 52	PERAK BARAT / ALUN ALUN PRIOK	Link	-7.22303	112.73177
53	Pertamina 53	JL. GRESIK NO. 97	Link	-7.232452	12.72082
54	Pertamina 54	JL. GRESIK PPI NO. 7 SURABAYA	Link	-7.2329179	112.7274746
55	Pertamina 55	JL. STASIUN KOTA NO.62 A	Link	-7.2425654	112.7397418
56	Pertamina 56	JL. ISKANDAR MUDA	Link	-7.2251299	112.7420044
57	Pertamina 57	JL. IKAN KAKAP NO. 21 KREMBANGAN	Link	-7.2311634	112.7259176
58	Pertamina 58	JL. RAJAWALI NO. 24	Link	-7.2353905	112.7350516
59	Pertamina 59	JL. SISINGAMANGARAJA	Link	-7.222771	112.7362122
60	Pertamina 60	JL. RAYA KEDUNG DCOWEK 204	Link	-7.2320945	112.7734899
61	Pertamina 61	JL. KALIANAK NO. 152 C KEL MOROKREMBA	Link	-7.229329	112.7086073
62	Pertamina 62	JL. PAWIYATAN / JL. SEMARANG	Link	-7.2508807	112.7299741
63	Pertamina 63	JL. KENJERAN	Link	-7.2429791	112.7592018
64	Pertamina 64	JL. EMBONG KEMIRI NO. 48	Link	-7.2698482	112.7463461
65	Pertamina 65	JL. DEMAK NO. 152	Link	-7.2483187	112.7211219
66	Pertamina 66	JL. TEGALSARI NO. 4345	Link	-7.266081	112.7378392
67	Pertamina 67	JL. AMBENGAN 16 KETABANG	Link	-7.2561305	112.7464757

68	Pertamina 68	JL. RY PECINDILAN 50 KAPASARI	Link	-7.2451659	112.7452111
69	Pertamina 69	JL. KENJERAN 99	Link	-7.2418103	112.7573204
70	Pertamina 70	JL. PAHLAWAN KEC. BUBUTAN	Link	-7.2498399	112.7373266
71	Shell 1	SPBU Shell Jl. Kenjeran No.591, Kalijudan	Link	-7.2503249	112.7874995
72	Shell 2	SPBU Shell Jl. Kalijudan No. 24	Link	-7.2619571	112.7741993
73	Shell 3	SPBU Shell Jl. Raya Kertajaya Indah No. 143	Link	-7.2799463	112.789015
74	Shell 4	SPBU Shell MERR, 291	Link	-7.3126372	112.7808488
75	Shell 5	SPBU Shell Raya Tenggilis-1	Link	-7.3204815	112.755639
76	Shell 6	SPBU Shell - Margorejo	Link	-7.3174111	112.7504313
77	Shell 7	SPBU Shell Jl. Raya Prapen No.50	Link	-7.3088436	112.7601212
78	Shell 8	SPBU Shell Jl.Raya Jemursari	Link	-7.3270382	112.732417
79	Shell 9	SPBU Shell Mastrip - Kedurus, Karangpilang	Link	-7.3243952	112.7089127
80	Shell 10	SPBU Shell Jl. Mayjend. Jonosewojo	Link	-7.2916648	112.6756172
81	Shell 11	SPBU Shell Citraland - Mayor Jenderal Sungkono	Link	-7.2830254	112.6505561
82	Shell 12	SPBU Shell Jl. Diponegoro, DR. Soetomo	Link	-7.2837719	112.7334513
83	Shell 13	SPBU Shell Jl. Simo Magersari No. 62	Link	-7.2678209	112.7124069
84	Shell 14	SPBU Shell Jl. Raya Dupak	Link	-7.2456583	112.7213161
85	Shell 15	SPBU Shell - Pemuda Central-1	Link	-7.2657828	112.7483547

The following is a more comprehensive dataset used in this project: Dataset SC FP Group 4_Pertamina & Shell Gas Stations. The "lat & long" sheet contains the information as shown in the table above, which includes the latitude and longitude

coordinates of the Gas Stations and the Disperindag Surabaya office. The "distances (km)" sheet provides the round-trip distances between each Gas Station and other Gas Stations or the Disperindag office in Surabaya, measured in kilometers based on Google Maps data.

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. Affinity Propagation

Affinity propagation is a clustering method that uses similarity measurements between pairs of data points as input. The technique works through an iterative process, where data points share information in the form of numerical values. This process repeats until clear clusters and representative exemplars are formed (Frey & Dueck, 2007). Affinity propagation is unique because it automatically determines the number of clusters based on the data itself. This is achieved through two key parameters: preference, which controls how many exemplars are chosen, and the damping factor, which helps prevent fluctuations in the data during the update process by adjusting the responsibility and availability messages (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. Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic algorithm inspired by the foraging behavior of ants, particularly their ability to find the shortest paths to food sources. This algorithm utilizes a colony of artificial ants that communicate indirectly through pheromone trails, which are chemical substances deposited on paths as they traverse the environment. The intensity of these pheromone trails influences the probability of other ants choosing the same path, leading to a positive feedback loop that reinforces successful routes (Dorigo & Stützle, 2018). The algorithm operates iteratively, where each ant constructs a solution based on pheromone levels and heuristic information, and

after all ants have completed their paths, pheromone evaporation occurs to prevent stagnation and encourage exploration of new paths. ACO has been successfully applied to various combinatorial optimization problems, including the traveling salesman problem and network routing (Bonabeau et al., 1999; Dorigo et al., 2006).

3. Simulated Annealing

Simulated Annealing (SA) is a probabilistic optimization technique inspired by the annealing process in metallurgy, where controlled cooling of materials leads to a stable state. This algorithm operates by exploring the solution space and allowing for both uphill and downhill moves in terms of energy, which helps avoid local minima and promotes global optimization (Kirkpatrick et al., 1983). Initially, SA starts with a high "temperature," enabling the acceptance of worse solutions with higher probabilities, thus facilitating exploration. As the algorithm progresses, the temperature gradually decreases, reducing the likelihood of accepting inferior solutions and allowing the search to converge towards an optimal solution (Cerny, 1985). This unique feature makes SA particularly effective for solving complex combinatorial problems, such as the traveling salesman problem and various scheduling tasks (Aarts & Korst, 1989).

4. Tabu Search

Tabu Search (TS) is a powerful metaheuristic optimization technique developed by Fred Glover in the late 1980s. It enhances local search methods by incorporating adaptive memory structures to avoid cycling back to previously explored solutions, thus enabling the search process to escape local optima (Glover, 1989). The algorithm operates by maintaining a "tabu list," which records recently visited solutions or moves, prohibiting them from being revisited for a specified number of iterations. This mechanism allows the algorithm to explore new regions of the solution space while still permitting "worsening" moves if they lead to a better overall solution, thereby fostering exploration (Glover & Laguna, 1997). TS has been effectively applied across various fields, including scheduling, routing, and resource allocation problems, demonstrating its versatility and efficiency in finding high-quality solutions for complex combinatorial optimization challenges (Aarts & Korst, 1989; Jwo et al., 2023).

5. Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a computational method inspired by the social behavior of birds flocking or fish schooling. Introduced by Kennedy and Eberhart in 1995, PSO is a heuristic optimization algorithm that uses a population of particles (candidate solutions) moving through the solution space to find the optimal value for a given objective function. Each particle adjusts its position based on its own experience and that of its neighbors, following simple mathematical rules. The algorithm is widely used in various optimization problems, including engineering, robotics, and machine learning (Kennedy & Eberhart, 1995). The concept of "swarm intelligence," where multiple agents cooperate without centralized control, is at the core of PSO's functioning. Over time, several improvements and modifications have been made to PSO, enhancing its convergence rate and robustness in handling complex, high-dimensional optimization problems (Zhang et al., 2021). PSO's simplicity, ease of implementation, and ability to handle non-linear and multimodal objective functions make it a powerful tool in computational optimization (Shami et al., 2022).

6. Lovebird Algorithm

The Lovebird Algorithm is an innovative optimization technique inspired by the social behavior and mating rituals of lovebirds. This algorithm employs a population-based approach where candidate solutions, represented as "lovebirds," interact with one another to explore the solution space effectively. The process begins with an initial population of lovebirds, each representing a potential solution to the optimization problem at hand. Through iterations, these lovebirds communicate and share information about their positions, leading to the discovery of better solutions over time. The algorithm incorporates mechanisms such as selection, crossover, and mutation, which are essential for maintaining diversity within the population and avoiding premature convergence to suboptimal solutions (Utamima et al., 2020). The Lovebird Algorithm has shown promising results in various applications, including scheduling and route optimization, demonstrating its effectiveness in solving complex problems efficiently.

V. Application of the Algorithm

V.I Clustering Methods

In this project, clustering is performed using the Affinity Propagation method, which determines the number of clusters dynamically based on the input data. This method selects examples and forms clusters by iteratively exchanging messages between data points. Below is a more detailed explanation of the technique.

1. 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:

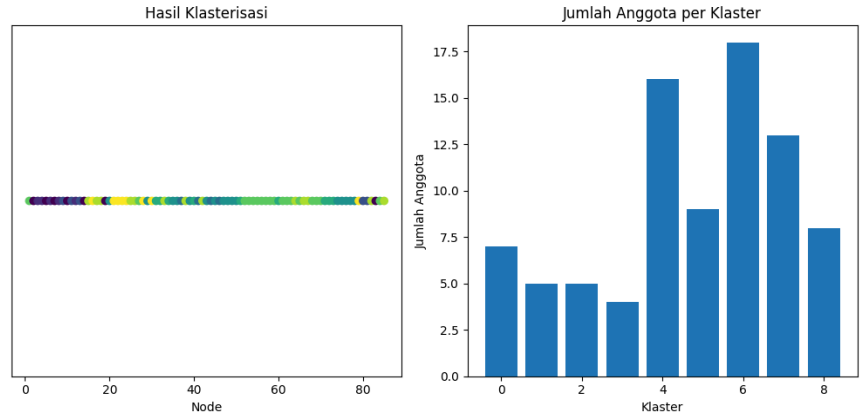
```
import numpy as np
import pandas as pd
from sklearn.cluster import AffinityPropagation
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

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("/content/Dataset-data-pertamina-shell_fix.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))
    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: (86, 86)
Nilai minimum: 0.0, Nilai maksimum: 31.9
Jumlah klaster: 9
Skor Silhouette: 0.2971
Klaster 1: [2, 5, 7, 10, 14, 19, 83]
Klaster 2: [3, 4, 6, 8, 12]
Klaster 3: [9, 11, 13, 80, 81]
Klaster 4: [37, 41, 44, 46]
Klaster 5: [20, 29, 35, 36, 39, 43, 45, 47, 48, 49, 50, 74, 75, 76, 77, 78]
Klaster 6: [31, 32, 34, 40, 51, 60, 71, 72, 73]
Klaster 7: [1, 27, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63, 65, 68, 69, 70, 84]
Klaster 8: [15, 17, 18, 25, 26, 33, 38, 42, 64, 66, 67, 82, 85]
Klaster 9: [16, 21, 22, 23, 24, 28, 30, 79]
```



The output of the Affinity Propagation clustering shows that a dataset with 86 points was divided into 9 clusters. Each cluster is represented by a group of data points (nodes) that share similar characteristics based on a similarity matrix. On the left, the "Hasil Klasterisasi" (Clustering Result) plot shows how nodes (data points) are assigned to their respective clusters. Each color represents a different cluster, and the nodes are spread along the horizontal axis. On the right, the bar chart "Jumlah Anggota per Klaster" (Number of Members per Cluster) displays the size of each cluster. Cluster 6 has the highest number of members (18), followed by Cluster 4 (16), while Clusters 3 and 2 have the fewest members.

In summary, Affinity Propagation identifies clusters based on data similarity without pre-specifying the number of clusters. Here, the clusters vary in size, and the moderate silhouette score suggests room for improvement in separating the clusters more distinctly.

V.II Optimization Algorithm

a. Genetic Algorithm

The genetic algorithm (GA) is applied in multiple experiments to solve optimization problems. The first experiment just uses GA, followed by GA combined with other optimization techniques: Simulated Annealing (SA), Tabu Search (TS), and Particle Swarm Optimization (PSO). These experiments aim to compare the performance and efficiency of GA with and without hybrid approaches in finding optimal solutions across various test scenarios. Each method evaluates parameters such as population size, mutation

rate, and the number of generations to identify the shortest distances and computational efficiency.

1) GA

The provided code implements a Genetic Algorithm (GA) to solve a routing problem for multiple clusters of nodes. Each cluster is a subset of locations, and the goal is to minimize the total distance for visiting nodes in each cluster. Two parameter configurations were tested to analyze their performance:

- Parameter Set 1: A population size of 50, mutation rate of 0.05, and 100 generations were used, resulting in a shortest distance of 322.9 km.
- Parameter Set 2: A larger population size of 75, a reduced mutation rate of 0.03, and 150 generations yielded a shorter distance of 314.8 km.

The code is structured into multiple classes and functions, and each component is explained below:

- **Class Definition: GeneticAlgorithm**

```
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
```

The GeneticAlgorithm class encapsulates the logic of the genetic algorithm. It handles the initialization of the population, fitness calculation, parent selection, crossover, mutation, and evolution across generations. Parameters:

- distance_matrix: Matrix of distances between locations.
- population_size: Number of solutions (routes) in each generation.
- mutation_rate: Probability of mutating a gene in a chromosome.
- generations: Number of iterations to run the algorithm.

- **Initial Population**

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

```
        return population
```

The initialize_population method generates random routes (chromosomes) for the initial population. Each chromosome starts and ends at node 0 (depot).

- **Fitness Calculation**

The fitness of each chromosome is calculated inversely proportional to its total distance.

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

    def calculate_fitness(self, chromosome):
        return 1 / self.calculate_distance(chromosome)
```

calculate_distance: Computes the total distance of a route.
calculate_fitness: Returns the inverse of the route's total distance.

- **Parent Selection**

The select_parents method uses a tournament selection approach, where a subset of chromosomes competes, and the fittest is chosen as a parent.

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

Tournament size is set to 5, ensuring diverse selection while favoring fitter chromosomes.

- **Crossover**

The crossover method creates a child chromosome by combining segments of two parents. A partial mapped crossover (PMX) strategy ensures valid routes.

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

- A segment from parent1 is directly inherited by the child.
- Remaining nodes are filled in the order of their appearance in parent2.

- **Mutation**

The mutate method introduces random changes in the chromosome by swapping nodes, ensuring genetic diversity. Mutation occurs with a probability defined by mutation_rate.

```
    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]
```

- **Evolution Process**

The evolve method generates a new population by:

- Selecting parents.
- Performing crossover and mutation.
- Replacing the old population with new offspring.

```
    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
```

- **Execution: Running the Algorithm**

The run method evolves the population over the specified number of generations and tracks the best route and distance.

```
    def run(self):
        start_time = time.time()

        for generation in range(self.generations):
            self.evolve()

            # Find the best route in current population
```

```
            for chromosome in self.population:
                                                    distance   =
self.calculate_distance(chromosome)
                if distance < self.best_distance:
                    self.best_distance = distance
                    self.best_route = chromosome

        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime
```

- Returns the best route, its distance, and the execution time.

- **Function: run_genetic_algorithm**

This function processes each cluster of nodes:

- Extracts the relevant submatrix of the distance matrix.
- Runs the GeneticAlgorithm on each cluster.
- Reports the shortest route and distance for each cluster and the total result.

```
def     run_genetic_algorithm(distance_matrix,     clusters,
population_size, mutation_rate, generations):
    total_shortest_distance = 0
    total_runtime = 0

    for i, cluster in enumerate(clusters, 1):
        if 0 not in cluster:
            cluster = [0] + cluster

                                  selected_distance_matrix    =
distance_matrix[np.ix_(cluster, cluster)]

         solver = GeneticAlgorithm(selected_distance_matrix,
population_size, mutation_rate, generations)
        best_route, best_distance, runtime = solver.run()

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

        print(f"Cluster {i}:")
        print("Route:", final_route)
        print("Shortest distance:", best_distance)
        print(f"Runtime:", 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:", total_runtime, "seconds")
    print("=" * 100)
```

- **Main Function**

The main function initializes parameters, loads the distance matrix, defines clusters, and runs the GA with two different parameter sets.

```
def main():
                                        distance_matrix        =
np.loadtxt("/content/Dataset-data-pertamina-shell_fix.txt")
    clusters = [
        [2, 5, 7, 10, 14, 19, 83],
        [3, 4, 6, 8, 12],
        [9, 11, 13, 80, 81],
        [37, 41, 44, 46],
        [20, 29, 35, 36, 39, 43, 45, 47, 48, 49, 50, 74, 75,
76, 77, 78],
        [31, 32, 34, 40, 51, 60, 71, 72, 73],
         [1, 27, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63,
65, 68, 69, 70, 84],
          [15, 17, 18, 25, 26, 33, 38, 42, 64, 66, 67, 82,
85],
        [16, 21, 22, 23, 24, 28, 30, 79]
    ]

      print("Running Genetic Algorithm with Parameters Set
1:")
          run_genetic_algorithm(distance_matrix,  clusters,
population_size=50, mutation_rate=0.05, generations=100)

      print("Running Genetic Algorithm with Parameters Set
2:")
          run_genetic_algorithm(distance_matrix,  clusters,
population_size=75, mutation_rate=0.03, generations=100)

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

- Distance matrix is loaded from a file.
- Cluster definitions are provided as a list of node sets.

- The GA is executed twice with varying population sizes and mutation rates.

Output Parameter 1:

```
Running Genetic Algorithm with Parameters Set 1:
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 0.3190650939941406 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 0.26958537101745605 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 0.27364444732666016 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 0.23933649063110352 seconds

Cluster 5:
Route: [0, 48, 35, 36, 74, 49, 50, 75, 78, 76, 29, 77, 20, 45, 47, 39, 43, 0]
Shortest distance: 47.9
Runtime: 0.819199800491333 seconds

Cluster 6:
Route: [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]
Shortest distance: 29.5
Runtime: 0.6763384342193604 seconds

Cluster 7:
Route: [0, 68, 69, 63, 59, 56, 57, 52, 58, 84, 65, 53, 61, 54, 1, 27, 62, 55, 70, 0]
Shortest distance: 42.599999999999994
Runtime: 1.0834243297576904 seconds

Cluster 8:
Route: [0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]
Shortest distance: 31.6
Runtime: 0.7649786472320557 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 0.35335659980773926 seconds

Total Shortest Distance for all clusters: 322.9
Total Runtime: 4.798929214477539 seconds
```

Output Parameter 2:

```
Running Genetic Algorithm with Parameters Set 2:
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 0.46991705894470215 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 0.3953511714935303 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 0.3979830741882324 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 0.35410189628601074 seconds

Cluster 5:
Route: [0, 43, 48, 35, 36, 74, 49, 47, 20, 77, 78, 29, 76, 75, 45, 50, 39, 0]
Shortest distance: 43.9
Runtime: 0.8325519561767578 seconds

Cluster 6:
Route: [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]
Shortest distance: 29.5
Runtime: 0.5555610656738281 seconds

Cluster 7:
Route: [0, 27, 65, 62, 84, 1, 55, 58, 53, 61, 57, 54, 52, 59, 56, 69, 63, 68, 70, 0]
Shortest distance: 38.50000000000001
Runtime: 0.9107253551483154 seconds

Cluster 8:
Route: [0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]
Shortest distance: 31.6
Runtime: 0.6912493705749512 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 0.5150434970855713 seconds

Total Shortest Distance for all clusters: 314.80000000000007
Total Runtime: 5.122484445571899 seconds
```

2) Hybrid GA with SA

Hybrid GA with SA is conducted using two sets of parameters. In the first set, the parameters are population_size = 50, mutation_rate = 0.05, generations = 100, initial_temperature = 100, and cooling_rate = 0.95, which results in a total distance of 318.5 km. In the second set, the parameters are population_size = 75, mutation_rate = 0.03, generations = 100, initial_temperature = 150, and cooling_rate = 0.93, resulting in a total distance of 315.5 km. The discussion will now focus on the related code.

- Import library

```
import numpy as np
import random
import math
import time
```

- Class Initialization (HybridRoutingAlgorithm):

```
class HybridRoutingAlgorithm:
      def __init__(self, distance_matrix, population_size,
mutation_rate,       generations,       initial_temperature,
cooling_rate):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.mutation_rate = mutation_rate
```

```
        self.generations = generations
        self.initial_temperature = initial_temperature
        self.cooling_rate = cooling_rate
        self.population = self.initialize_population()
        self.best_distance = float('inf')
        self.best_route = None
```

The **HybridRoutingAlgorithm** class is defined to combine both Genetic Algorithm (GA) and Simulated Annealing (SA) for routing optimization. The constructor (**__init__**) initializes key parameters like the distance matrix, population size, mutation rate, and the cooling rate for simulated annealing. It also initializes the population of routes and tracks the best route and its distance.

- **Population Initialization**

```
    def initialize_population(self):
        population = []
        for _ in range(self.population_size):
            # Exclude 0 from shuffling, create route between
unique nodes
                        cluster_nodes = list(set(range(1,
len(self.distance_matrix))) - set([0]))
            random.shuffle(cluster_nodes)
            chromosome = [0] + cluster_nodes + [0]
            population.append(chromosome)
        return population
```

The **initialize_population** method creates an initial population of routes. Each route starts and ends at node 0 (the depot) and visits all other nodes (locations) in a random order. This ensures that the routes represent round trips from the starting point.

- **Distance Calculation**

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

The **calculate_distance** method computes the total travel distance for a given route. It sums the distances between consecutive nodes based on the provided **distance_matrix**.

- **Fitness Calculation**

```
    def calculate_fitness(self, chromosome):
        # Penalize routes with duplicate nodes (except 0)
```

```
        unique_nodes = len(set(chromosome[1:-1]))
        if unique_nodes < len(chromosome) - 2:
            return 0  # Invalid route
        return 1 / self.calculate_distance(chromosome)
```

The **calculate_fitness** method evaluates the quality of a route by calculating its total distance and returning the inverse of the distance. The shorter the route, the higher the fitness. If a route contains duplicate nodes (except for node 0), it is considered invalid and assigned a fitness of 0.

- **Parent Selection (select_parents)**

```
    def select_parents(self):
                valid_population = [chrom for chrom in
self.population if self.calculate_fitness(chrom) > 0]
        if not valid_population:
            valid_population = self.population

        tournament_size = min(5, len(valid_population))
              tournament = random.sample(valid_population,
tournament_size)
        return max(tournament, key=self.calculate_fitness)
```

The **select_parents** method uses tournament selection to choose two parents for crossover. A random sample of chromosomes is chosen, and the one with the highest fitness is selected. This process is repeated for selecting the second parent.

- **Crossover**

```
    def crossover(self, parent1, parent2):
        # Ensure crossover maintains route validity
               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 and gene != 0:
                if pointer == len(child) - 1:
                    pointer = 1
                child[pointer] = gene
                pointer += 1

        return child
```

The **crossover** method generates a child route by combining segments from two parent routes. It selects a random segment from one parent and fills the child route with that segment, ensuring no duplicates by filling the remaining positions with genes from the second parent.

- **Mutation**

```
    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)
                if j != i and chromosome[j] != 0:
                                chromosome[i], chromosome[j] =
chromosome[j], chromosome[i]
```

The **mutate** method introduces small changes to a route by swapping two nodes at random (excluding node 0). This helps maintain genetic diversity and avoids getting stuck in local minima.

- **Simulated Annealing**

```
       def simulated_annealing_local_search(self, solution,
temperature):
        current_solution = solution.copy()
                                        current_distance     =
self.calculate_distance(current_solution)

        for _ in range(10):  # Local search iterations
            # Generate neighboring solution
                            i, j = random.sample(range(1,
len(current_solution) - 1), 2)
            new_solution = current_solution.copy()
                          new_solution[i], new_solution[j] =
new_solution[j], new_solution[i]

                                                new_distance    =
self.calculate_distance(new_solution)

            # Probabilistic acceptance
            if (new_distance < current_distance or
                random.random() < math.exp((current_distance
- new_distance) / temperature) and
                               len(set(new_solution[1:-1])) ==
len(current_solution[1:-1])):
                current_solution = new_solution
                current_distance = new_distance
```

```
        return current_solution
```

The **simulated_annealing_local_search** method improves the route by applying simulated annealing. It swaps two nodes and evaluates the resulting route. If the new route is better, it is accepted. If the new route is worse, it may still be accepted with a probability that decreases as the temperature cools. This process allows the algorithm to escape local minima and explore better solutions.

- **Evolving the Population**

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

            # Apply Simulated Annealing local search
                                                    child    =
self.simulated_annealing_local_search(child, temperature)

            new_population.append(child)
        self.population = new_population
```

The **evolve** method generates a new population by selecting parents, performing crossover, mutation, and local search on the offspring. It then evaluates the fitness of each new route and keeps track of the best route found so far.

- **Running the Algorithm (run)**

```
    def run(self):
        start_time = time.time()
        temperature = self.initial_temperature

        for generation in range(self.generations):
            # Reduce temperature
            temperature *= self.cooling_rate

            self.evolve(temperature)

            # Find best valid route
                        valid_routes = [chrom for chrom in
self.population if self.calculate_fitness(chrom) > 0]
            if valid_routes:
```

```
                        best_chromosome = max(valid_routes,
key=self.calculate_fitness)
                                            best_distance =
self.calculate_distance(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
```

The **run** method executes the hybrid algorithm over a predefined number of generations. In each generation, the population evolves, and the best route is updated. The method returns the best route, its distance, and the total runtime.

- **Running the Hybrid Algorithm for Clusters**

```
def run_hybrid_routing_algorithm(distance_matrix, clusters,
                                 population_size=50,
                                 mutation_rate=0.05,
                                 generations=100,
                                 initial_temperature=100,
                                 cooling_rate=0.95):
    total_shortest_distance = 0
    total_runtime = 0

    for i, cluster in enumerate(clusters, 1):
            # Ensure 0 is first in the cluster for routing
constraint
        if 0 not in cluster:
            cluster = [0] + cluster

        # Select subset of distance matrix for this cluster
                                selected_distance_matrix =
distance_matrix[np.ix_(cluster, cluster)]

        # Solve routing problem for this cluster
                                            solver =
HybridRoutingAlgorithm(selected_distance_matrix,
                                        population_size,
                                        mutation_rate,
                                        generations,
                                        initial_temperature,
```

```
                                        cooling_rate)

        best_route, best_distance, runtime = solver.run()

        # Map back to original indices
              final_route = [cluster[index] for index in
best_route]

        print(f"Cluster {i}:")
        print("Route:", final_route)
        print("Shortest distance:", best_distance)
        print(f"Runtime:", 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:", total_runtime, "seconds")
    print("=" * 100)
```

The **run_hybrid_routing_algorithm** function takes the distance matrix and a list of clusters. For each cluster, it extracts the relevant submatrix and runs the hybrid routing algorithm on it, printing the best route and distance for each cluster.

- **Main Function**

```
def main():
    # Load distance matrix
                                    distance_matrix         =
np.loadtxt("/content/Dataset-data-pertamina-shell_fix.txt")

    # Define clusters based on the given clustering
    clusters = [
        [2, 5, 7, 10, 14, 19, 83],
        [3, 4, 6, 8, 12],
        [9, 11, 13, 80, 81],
        [37, 41, 44, 46],
         [20, 29, 35, 36, 39, 43, 45, 47, 48, 49, 50, 74, 75,
76, 77, 78],
        [31, 32, 34, 40, 51, 60, 71, 72, 73],
         [1, 27, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63,
65, 68, 69, 70, 84],
```

```
            [15, 17, 18, 25, 26, 33, 38, 42, 64, 66, 67, 82,
85],
            [16, 21, 22, 23, 24, 28, 30, 79]
        ]

        # Different parameter sets for comparison
        print("Running GA-SA with Parameters Set 1:")
        run_hybrid_routing_algorithm(distance_matrix, clusters)

        print("\nRunning GA-SA with Parameters Set 2:")
        run_hybrid_routing_algorithm(
            distance_matrix,
            clusters,
            population_size=75,
            mutation_rate=0.03,
            generations=100,
            initial_temperature=150,
            cooling_rate=0.93
        )
```

The **main** function loads the distance matrix from a TXT file and defines the clusters of nodes (locations). It then runs the hybrid algorithm on these clusters and prints the results.

- **Execution Block**

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

The if __name__ == "__main__": block ensures that the main function runs only when the script is executed directly, not when imported as a module.

Output Best Parameter GA + SA (Parameter 2):

```
Running GA-SA with Parameters Set 2:
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 6.293348789215088 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 6.639508008956909 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 5.167013168334961 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 5.69657301902771 seconds

Cluster 5:
Route: [0, 39, 50, 74, 47, 45, 75, 76, 29, 78, 77, 20, 49, 36, 48, 35, 43, 0]
Shortest distance: 45.4
Runtime: 12.415815591812134 seconds

Cluster 6:
Route: [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]
Shortest distance: 29.5
Runtime: 7.838037967681885 seconds

Cluster 7:
Route: [0, 69, 63, 68, 70, 55, 59, 56, 52, 54, 57, 53, 61, 1, 27, 84, 65, 62, 58, 0]
Shortest distance: 37.5
Runtime: 15.1155526638031 seconds

Cluster 8:
Route: [0, 64, 66, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 67, 0]
Shortest distance: 31.800000000000004
Runtime: 10.917413473129272 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 8.240407705307007 seconds

Total Shortest Distance for all clusters: 315.5
Total Runtime: 78.32367038726807 seconds
```

3) Hybrid GA with TS

Hybrid GA with TS is also conducted using two sets of parameters. In the first set, the parameters are population_size = 50, mutation_rate = 0.05, generations = 100, tabu_tenure=10, local_iterations=20, which results in a total distance of 303.7 km. In the second set, the parameters are population_size = 75, mutation_rate = 0.03, generations = 100, tabu_tenure=15, local_iterations=30, resulting in a total distance of 303.2 km. Among all the experiments conducted on the GA, the GA with TS using parameter 2 was the best in finding the shortest distance. The discussion will now focus on the related code.

- **Class Definition: HybridRoutingAlgorithm**

This class is the core of the hybrid algorithm, encapsulating all necessary methods for initialization, fitness calculation, parent selection, crossover, mutation, and local search using tabu lists.

```
class HybridRoutingAlgorithm:
      def __init__(self, distance_matrix, population_size,
generations, mutation_rate, tabu_tenure, local_iterations):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.tabu_tenure = tabu_tenure
```

```
        self.local_iterations = local_iterations
        self.population = self.initialize_population()
        self.best_distance = float('inf')
        self.best_route = None
        self.tabu_list = []
```

- The class constructor initializes the hybrid algorithm with parameters such as population size, mutation rate, tabu tenure, and local search iterations.
- It also initializes the population with randomly generated routes and prepares a tabu list for the local search phase.

- **Population Initialization**

```
    def initialize_population(self):
        population = []
        n = len(self.distance_matrix)
        for _ in range(self.population_size):
            individual = list(range(1, n))  # Exclude node 0
            random.shuffle(individual)
              individual = [0] + individual + [0]  # Add node
0 at start and end
            population.append(individual)
        return population
```

- Each route (individual) starts and ends at node 0, with the intermediate nodes shuffled randomly.
- This ensures diverse starting solutions for the genetic algorithm.

- **Fitness Evaluation**

The total distance of a route is calculated as the fitness measure.

```
    def calculate_distance(self, route):
        distance = 0
        for i in range(len(route) - 1):
                                                distance    +=
self.distance_matrix[route[i]][route[i + 1]]
        return distance
```

- The fitness of a route is measured as the sum of distances between consecutive nodes.
- Lower distances represent better solutions.

- **Genetic Algorithm Components**
  - **Parent Selection:** Randomly selects two parents for crossover.

```
    def select_parents(self):
        return random.sample(self.population, 2)
```

- **Crossover:** Combines portions of two parents to create a child route.

```
    def crossover(self, parent1, parent2):
          size = len(parent1) - 2  # Exclude start and end
(node 0)
          start, end = sorted(random.sample(range(1, size +
1), 2))
        child = [-1] * (size + 2)
        child[0] = child[-1] = 0  # Keep node 0 at start and
end
        child[start:end] = parent1[start:end]

        p2_index = 1
        for i in range(1, size + 1):
            if child[i] == -1:
                while parent2[p2_index] in child:
                    p2_index += 1
                child[i] = parent2[p2_index]

        return child
```

- **Mutation:** Swaps two nodes in a route with a small probability.

```
    def mutate(self, individual):
        if random.random() < self.mutation_rate:
             i, j = random.sample(range(1, len(individual) -
1), 2)  # Exclude start and end (node 0)
              individual[i], individual[j] = individual[j],
individual[i]
```

- Crossover combines segments of two parents, ensuring a balance of diversity and convergence.
- Mutation introduces variability to avoid local optima.

- **Tabu Search for Local Optimization**

This phase improves solutions by exploring neighbors, while avoiding revisiting recently explored routes using a tabu list.

```
    def tabu_search_local_search(self, solution):
        current_solution = solution.copy()
                                        current_distance     =
self.calculate_distance(current_solution)

        for _ in range(self.local_iterations):
            best_neighbor = None
            best_neighbor_distance = float('inf')
```

```
            for i in range(1, len(current_solution) - 2):  # Exclude start and end (node 0)
                for j in range(i + 1, len(current_solution) - 1):
                    neighbor = current_solution.copy()
                    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]

                    if tuple(neighbor) not in self.tabu_list:
                        neighbor_distance = self.calculate_distance(neighbor)
                        if neighbor_distance < best_neighbor_distance:
                            best_neighbor = neighbor
                            best_neighbor_distance = neighbor_distance

            if best_neighbor:
                current_solution = best_neighbor
                current_distance = best_neighbor_distance

                self.tabu_list.append(tuple(current_solution))
                if len(self.tabu_list) > self.tabu_tenure:
                    self.tabu_list.pop(0)

        return current_solution
```

- Neighbors are generated by swapping nodes in the current solution.
- The tabu list prevents cycles by restricting recently visited solutions.

- **Running the Algorithm**

The algorithm evolves the population over generations, updating the best route found.

```
    def run(self):
        start_time = time.time()

        for generation in range(self.generations):
            new_population = []
            for _ in range(self.population_size):
                parent1, parent2 = self.select_parents()
```

```
                child = self.crossover(parent1, parent2)
                self.mutate(child)
                child = self.tabu_search_local_search(child)
                new_population.append(child)

            self.population = new_population
            for individual in self.population:
                                                distance =
self.calculate_distance(individual)
                if distance < self.best_distance:
                    self.best_distance = distance
                    self.best_route = individual

        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime
```

- **Clustering and Execution**

Clusters are defined, and the hybrid algorithm is applied to each cluster separately.

```
def run_genetic_algorithm(distance_matrix, clusters,
params):
    total_shortest_distance = 0
    total_runtime = 0

    for i, cluster in enumerate(clusters, 1):
        if 0 not in cluster:
            cluster = [0] + cluster

                                selected_distance_matrix =
distance_matrix[np.ix_(cluster, cluster)]

        solver = HybridRoutingAlgorithm(
            selected_distance_matrix,
            population_size=params["population_size"],
```

```
            generations=params["generations"],
            mutation_rate=params["mutation_rate"],
            tabu_tenure=params["tabu_tenure"],
            local_iterations=params["local_iterations"]
        )

        best_route, best_distance, runtime = solver.run()
              final_route = [cluster[index] for index in
best_route]

        print(f"Cluster {i}:")
        print("Route:", final_route)
        print("Shortest distance:", best_distance)
        print(f"Runtime:", runtime, "seconds")
        print()

        total_shortest_distance += best_distance
        total_runtime += runtime

                    print(f"Total    Shortest    Distance:",
total_shortest_distance)
    print(f"Total Runtime:", total_runtime, "seconds")
    print("=" * 100)
```

- **Main Function**

  Runs the algorithm with two parameter sets.

```
def main():
    # Load distance matrix
                                    distance_matrix        =
np.loadtxt("Dataset-data-pertamina-shell_fix.txt")

    # Define clusters
    clusters = [
        [2, 5, 7, 10, 14, 19, 83],
        [3, 4, 6, 8, 12],
        [9, 11, 13, 80, 81],
        [37, 41, 44, 46],
        [20, 29, 35, 36, 39, 43, 45, 47, 48, 49, 50, 74, 75,
76, 77, 78],
        [31, 32, 34, 40, 51, 60, 71, 72, 73],
         [1, 27, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63,
65, 68, 69, 70, 84],
```

```
          [15, 17, 18, 25, 26, 33, 38, 42, 64, 66, 67, 82,
85],
         [16, 21, 22, 23, 24, 28, 30, 79]
    ]

    params_set_1 = {
        "population_size": 50,
        "mutation_rate": 0.05,
        "generations": 100,
        "tabu_tenure": 10,
        "local_iterations": 20,
    }

    params_set_2 = {
        "population_size": 75,
        "mutation_rate": 0.03,
        "generations": 100,
        "tabu_tenure": 15,
        "local_iterations": 30,
    }

    print("Running GA-Tabu with Parameters Set 1:")
           run_genetic_algorithm(distance_matrix,   clusters,
params_set_1)

    print("Running GA-Tabu with Parameters Set 2:")
           run_genetic_algorithm(distance_matrix,   clusters,
params_set_2)


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

Output Best Parameter GA + TS (Parameter 2):

```
Running GA-Tabu with Parameters Set 2:
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 18.149431467056274 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 7.534499168395996 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 7.19471001625061 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 2.5616261959075928 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]
Shortest distance: 39.4
Runtime: 274.7357974052429 seconds

Cluster 6:
Route: [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]
Shortest distance: 29.5
Runtime: 39.08684945106506 seconds

Cluster 7:
Route: [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]
Shortest distance: 31.9
Runtime: 290.0940682888031 seconds

Cluster 8:
Route: [0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 33, 38, 67, 0]
Shortest distance: 31.100000000000005
Runtime: 112.64233541488647 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 28.915283918380737 seconds

Total Shortest Distance: 303.20000000000005
Total Runtime: 780.9146013259888 seconds
```

4) Hybrid GA with PSO

The hybrid GA with TS is performed using 2 parameters. The first parameter is population_size=50, mutation_rate=0.05, generations=100, pso_particles=30, inertia_weight=0.7, cognitive_weight=1.5, social_weight=1.5 which resulted in a distance of 321.0 km. The second parameter is the best in the GA experiment with TS, which is 315.3 km with parameters population_size=75, mutation_rate=0.03, generations=100, pso_particles=40, inertia_weight=0.6, cognitive_weight=1.7, social_weight=1.7. The following is a further explanation of the code used in GA with TS.

- **Class Definition: Hybrid GA-PSO Routing Algorithm**

```
class HybridGAPSORoutingAlgorithm:
      def __init__(self, distance_matrix, population_size,
mutation_rate, generations,
                           pso_particles, inertia_weight,
cognitive_weight, social_weight):
        self.distance_matrix = distance_matrix
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.generations = generations
```

```
        self.pso_particles = pso_particles
        self.inertia_weight = inertia_weight
        self.cognitive_weight = cognitive_weight
        self.social_weight = social_weight

        self.population = self.initialize_population()
        self.particles = self.initialize_pso_particles()
        self.best_distance = float('inf')
        self.best_route = None
```

This class implements a hybrid routing algorithm combining Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) to solve vehicle routing problems. The algorithm optimizes the total distance of routes for a set of clusters based on a distance matrix. The constructor initializes key parameters, such as the size of the population, mutation rate, number of generations for GA, PSO particle count, and weights for inertia, cognitive, and social components in PSO.

- **Population and Particle Initialization**

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

    def initialize_pso_particles(self):
        particles = []
        for _ in range(self.pso_particles):
              # Create a particle with position (route) and
velocity
                        cluster_nodes = list(set(range(1,
len(self.distance_matrix))) - set([0]))
            random.shuffle(cluster_nodes)
            position = [0] + cluster_nodes + [0]
                velocity = [random.uniform(-1, 1) for _ in
range(len(position)-2)]
            particles.append({
                'position': position,
                'velocity': velocity,
                'personal_best_position': position.copy(),
```

```
                                        'personal_best_fitness':
self.calculate_fitness(position)
            })
        return particles
```

initialize_population: Creates an initial population of random routes (chromosomes). Each route starts and ends with node `0` (assumed as the depot) and visits all other nodes in random order.
initialize_pso_particles: Initializes PSO particles, where each particle has a random route (**position**), a random velocity, and stores its personal best route and fitness.

- **Fitness and Distance Calculation**

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

    def calculate_fitness(self, chromosome):
        unique_nodes = len(set(chromosome[1:-1]))
        if unique_nodes < len(chromosome) - 2:
            return 0  # Invalid route
        return 1 / self.calculate_distance(chromosome)
```

calculate_distance: Computes the total distance of a route using the provided distance_matrix.
calculate_fitness: Calculates fitness as the inverse of the route distance. Invalid routes (e.g., with duplicate nodes) are assigned a fitness of 0.

- **Selection, Crossover, and Mutation in Genetic Algorithm**

```
    def select_parents(self):
                valid_population  =  [chrom  for  chrom  in
self.population if self.calculate_fitness(chrom) > 0]
        if not valid_population:
            valid_population = self.population

        tournament_size = min(5, len(valid_population))
              tournament  =  random.sample(valid_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 and gene != 0:
                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)
                if j != i and chromosome[j] != 0:
                                chromosome[i], chromosome[j] =
chromosome[j], chromosome[i]
```

select_parents: Implements tournament selection to pick the fittest routes from the population for crossover.
crossover: Combines two parent routes to create a new child route using a partially mapped crossover technique.
mutate: Swaps two nodes in a route with a certain probability (mutation rate) to introduce diversity.

- **Particle Updates in PSO**

```
    def pso_update_particles(self, global_best_route):
        for particle in self.particles:
            # Update velocity
            for i in range(len(particle['velocity'])):
                r1, r2 = random.random(), random.random()
                cognitive_component = (self.cognitive_weight
* r1 *
                    (particle['personal_best_position'][i+1]
- particle['position'][i+1]))
                 social_component = (self.social_weight * r2
*
                                    (global_best_route[i+1]  -
particle['position'][i+1]))
```

```
                particle['velocity'][i] = (
                                        self.inertia_weight  *
particle['velocity'][i] +
                    cognitive_component +
                    social_component
                )

            # Update position (route)
            new_route = particle['position'].copy()
            for i in range(1, len(new_route) - 1):
                        new_route[i]  =  round(new_route[i]  +
particle['velocity'][i-1])
                     new_route[i]  =  max(1,  min(new_route[i],
len(self.distance_matrix) - 1))

            # Ensure unique nodes
                                    unique_route   =   [0]   +
list(dict.fromkeys(new_route[1:-1])) + [0]
            while len(unique_route) < len(new_route):
                missing = len(new_route) - len(unique_route)
                additional_nodes = random.sample(
                                            list(set(range(1,
len(self.distance_matrix))) - set(unique_route[1:-1])),
                    missing
                )
                unique_route[1:-1] += additional_nodes

            particle['position'] = unique_route

            # Update personal best
                                          current_fitness    =
self.calculate_fitness(unique_route)
                                      if   current_fitness   >
particle['personal_best_fitness']:
                        particle['personal_best_position']  =
unique_route
                         particle['personal_best_fitness']  =
current_fitness
```

Updates particle velocities based on their inertia, the cognitive component (personal best), and the social component (global best). The positions (routes) of the particles are adjusted based on the velocities. It also ensures that updated routes remain valid by removing duplicates and filling in missing nodes.

- **Evolution and Optimization Process**

```
    def evolve(self):
        # Genetic Algorithm operations
        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

        # Find global best route for PSO
        global_best_route = max(self.population +
            [p['position'] for p in self.particles],
            key=self.calculate_fitness
        )

        # PSO update
        self.pso_update_particles(global_best_route)

    def run(self):
        start_time = time.time()

        for generation in range(self.generations):
            self.evolve()

            # Find best valid route
                    valid_routes = [chrom for chrom in
self.population if self.calculate_fitness(chrom) > 0]
            if valid_routes:
                    best_chromosome = max(valid_routes,
key=self.calculate_fitness)
                                            best_distance =
self.calculate_distance(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
```

evolve: Combines the operations of GA (selection, crossover, mutation) and PSO (particle updates) in each generation to refine the population and particles.
run: Executes the hybrid algorithm for a specified number of generations, updating the best route and shortest distance found.

- **Running the Algorithm for Multiple Clusters**

```
def run_hybrid_ga_pso_routing_algorithm(distance_matrix, clusters,
                                        population_size=50,
                                        mutation_rate=0.05,
                                        generations=100,
                                        pso_particles=30,
                                        inertia_weight=0.7,
                                        cognitive_weight=1.5,
                                        social_weight=1.5):
    total_shortest_distance = 0
    total_runtime = 0

    for i, cluster in enumerate(clusters, 1):
            # Ensure 0 is first in the cluster for routing constraint
        if 0 not in cluster:
            cluster = [0] + cluster

        # Select subset of distance matrix for this cluster
                                selected_distance_matrix = distance_matrix[np.ix_(cluster, cluster)]

        # Solve routing problem for this cluster
                                                solver = HybridGAPSORoutingAlgorithm(selected_distance_matrix,
                                                    population_size,
                                                    mutation_rate,
                                                    generations,
                                                    pso_particles,
                                                    inertia_weight,
                                                    cognitive_weight,
                                                    social_weight)

        best_route, best_distance, runtime = solver.run()
```

```
        # Map back to original indices
              final_route = [cluster[index] for index in
best_route]

        print(f"Cluster {i}:")
        print("Route:", final_route)
        print("Shortest distance:", best_distance)
        print(f"Runtime:", 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:", total_runtime, "seconds")
    print("=" * 100)
```

This function solves the routing problem for multiple clusters. For each cluster:

- A sub-matrix of the distance matrix is created for the cluster.
- The hybrid GA-PSO algorithm is applied to find the optimal route for that cluster.
- Results are aggregated to compute the total distance and runtime for all clusters.

- **Main Function**

```
def main():
    # Load distance matrix
                                    distance_matrix =
np.loadtxt("/content/Dataset-data-pertamina-shell_fix.txt")

    # Define clusters based on the given clustering
    clusters = [
        [2, 5, 7, 10, 14, 19, 83],
        [3, 4, 6, 8, 12],
        [9, 11, 13, 80, 81],
        [37, 41, 44, 46],
         [20, 29, 35, 36, 39, 43, 45, 47, 48, 49, 50, 74, 75,
76, 77, 78],
        [31, 32, 34, 40, 51, 60, 71, 72, 73],
         [1, 27, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63,
65, 68, 69, 70, 84],
```

```
            [15, 17, 18, 25, 26, 33, 38, 42, 64, 66, 67, 82,
85],
            [16, 21, 22, 23, 24, 28, 30, 79]
        ]

        # Different parameter sets for comparison
        print("Running GA-PSO with Parameters Set 1:")
            run_hybrid_ga_pso_routing_algorithm(distance_matrix,
clusters)

        print("\nRunning GA-PSO with Parameters Set 2:")
        run_hybrid_ga_pso_routing_algorithm(
            distance_matrix,
            clusters,
            population_size=75,
            mutation_rate=0.03,
            generations=100,
            pso_particles=40,
            inertia_weight=0.6,
            cognitive_weight=1.7,
            social_weight=1.7
        )

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

The main function:

- Loads the distance matrix from a file.
- Defines clusters (groups of nodes to be routed).
- Runs the hybrid GA-PSO algorithm with different parameter sets to compare results.

Output Best Parameter GA + PSO (Parameter 2):

```
Running GA-PSO with Parameters Set 2:
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 5.503525733947754 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 5.905848741531372 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 4.423274040222168 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 5.226487636566162 seconds

Cluster 5:
Route: [0, 43, 48, 35, 36, 74, 49, 77, 29, 78, 75, 76, 20, 45, 47, 50, 39, 0]
Shortest distance: 44.900000000000006
Runtime: 11.34412145614624 seconds

Cluster 6:
Route: [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]
Shortest distance: 29.5
Runtime: 7.714799404144287 seconds

Cluster 7:
Route: [0, 70, 68, 56, 59, 52, 54, 53, 61, 57, 84, 1, 65, 27, 62, 58, 55, 69, 63, 0]
Shortest distance: 37.4
Runtime: 12.345619678497314 seconds

Cluster 8:
Route: [0, 67, 33, 38, 85, 64, 82, 25, 26, 42, 17, 66, 18, 15, 0]
Shortest distance: 32.2
Runtime: 9.69640302658081 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 5.931854724884033 seconds

Total Shortest Distance for all clusters: 315.30000000000007
Total Runtime: 68.09193444252014 seconds
```

b. Ant Colony Optimization

Ant Colony Optimization (ACO) algorithm has emerged as a highly effective heuristic technique inspired by the foraging behavior of ants. ACO uses pheromone trails and heuristic information to explore and exploit potential solutions iteratively, making it well-suited for problems like routing and scheduling. However, while ACO excels in exploration, it sometimes struggles to refine solutions effectively in certain scenarios. To overcome these limitations, hybrid approaches have been introduced, combining ACO with other optimization techniques to enhance its performance. This study applies the standard ACO model and its hybrid variants, including ACO with Simulated Annealing (ACO-SA), ACO with Tabu Search (ACO-Tabu), and ACO with Particle Swarm Optimization (ACO-PSO), to evaluate their efficiency and accuracy in finding optimal solutions for routing problems. Each model leverages unique strategies for balancing exploration and exploitation, and their comparative analysis will highlight the strengths and weaknesses of these approaches.

a. ACO

Ant Colony Optimization (ACO) is a nature-inspired metaheuristic algorithm used to solve optimization problems such as routing. This implementation adapts ACO to minimize the total distance across multiple clusters of nodes. Each cluster represents a subset of locations that must be visited in the shortest possible route, with optimization repeated for various parameter configurations.

Code Explanation

The ACO implementation consists of several methods encapsulated in the ACORoutingAlgorithm class:

```
class ACORoutingAlgorithm:
    def __init__(self, distance_matrix, num_ants, evaporation_rate,
alpha, beta, iterations):
        self.distance_matrix = distance_matrix
        self.num_ants = num_ants
        self.evaporation_rate = evaporation_rate
        self.alpha = alpha
        self.beta = beta
        self.iterations = iterations
        self.pheromone_matrix = np.ones_like(distance_matrix)
        self.best_distance = float('inf')
        self.best_route = None
```

This section initializes the ACO parameters:

- `distance_matrix`: A matrix representing the distances between nodes.
- `num_ants`: The number of ants constructing solutions in each iteration.
- `evaporation_rate`: The rate at which pheromone trails evaporate.
- `alpha`: The influence of pheromones on the selection of paths.
- `beta`: The influence of heuristic information (distance) on the selection of paths.
- `iterations`: The number of optimization iterations.
- `pheromone_matrix`: Initialized as a matrix of ones to represent pheromone levels for each path.

Solution Construction

```
def construct_solution(self):
    route = [0]  # Start at node 0
    visited = set(route)

    while len(visited) < len(self.distance_matrix):
        current_node = route[-1]
        probabilities = self.calculate_probabilities(current_node,
visited)
        next_node = np.random.choice(range(len(probabilities)),
p=probabilities)
        route.append(next_node)
        visited.add(next_node)
```

```
    route.append(0)  # Return to the starting node
    return route
```

Each ant constructs a solution:

1. Starts at the depot (node 0).
2. Repeatedly selects the next node based on the transition probabilities, which are influenced by pheromone levels and heuristic values.
3. Returns to the depot after visiting all nodes.

Pheromone Updates

```
def update_pheromones(self, ants_routes):
    self.pheromone_matrix *= (1 - self.evaporation_rate)

    for route in ants_routes:
        distance = self.calculate_distance(route)
        pheromone_deposit = 1 / (distance + 1e-10)  # Avoid division by
zero

        for i in range(len(route) - 1):
            self.pheromone_matrix[route[i]][route[i + 1]] +=
pheromone_deposit
```

Pheromone levels are updated:

1. Evaporation reduces pheromone levels globally.
2. Deposits are added based on the quality of solutions (shorter distances lead to higher pheromone deposits).

Execution

```
def run(self):
    start_time = time.time()

    for _ in range(self.iterations):
        ants_routes = [self.construct_solution() for _ in
range(self.num_ants)]
        self.update_pheromones(ants_routes)

        for route in ants_routes:
            distance = self.calculate_distance(route)
            if distance < self.best_distance:
                self.best_distance = distance
```

```
                self.best_route = route

        end_time = time.time()
        runtime = end_time - start_time
        return self.best_route, self.best_distance, runtime
```

The algorithm:

- Constructs solutions for all ants in each iteration.
- Updates pheromone levels based on the solutions.
- Tracks the best solution found across all iterations.

Experiment Configuration

The ACO algorithm was run with four parameter configurations to analyze its performance:

1. **Default Parameters**:
   - `num_ants=20`, `alpha=1.0`, `beta=2.0`, `evaporation_rate=0.1`, `iterations=100`.
2. **Large Clusters**:
   - `num_ants=80`, `alpha=2.5`, `beta=6.0`, `evaporation_rate=0.08`.
3. **Small Clusters**:
   - `num_ants=40`, `alpha=1.5`, `beta=4.5`, `evaporation_rate=0.15`.
4. **Optimized Parameters**:
   - `num_ants=60`, `alpha=3.0`, `beta=5.5`, `evaporation_rate=0.07`.

Output

a. Running ACO with Default Parameters

```
Running ACO with default parameters:
Cluster 1: Route = [0, 19, 14, 10, 2, 7, 5, 83, 0], Distance = 25.40, Time = 4.48s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.00, Time = 1.53s
Cluster 3: Route = [0, 81, 13, 9, 11, 80, 0], Distance = 34.80, Time = 2.20s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.20, Time = 1.75s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.40, Time = 5.72s
Cluster 6: Route = [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0], Distance = 29.50, Time = 2.50s
Cluster 7: Route = [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0], Distance = 31.90, Time = 3.45s
Cluster 8: Route = [0, 15, 18, 82, 25, 26, 42, 17, 33, 38, 85, 64, 66, 67, 0], Distance = 31.80, Time = 2.85s
Cluster 9: Route = [0, 23, 24, 21, 79, 30, 22, 28, 16, 0], Distance = 35.90, Time = 2.29s
Total Distance: 303.90, Total Runtime: 26.77s
```

b. Running ACO with large clusters:

```
Running ACO with large clusters:
Cluster 1: Route = [0, 7, 5, 83, 2, 14, 10, 19, 0], Distance = 25.90, Time = 4.85s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.00, Time = 3.08s
Cluster 3: Route = [0, 9, 11, 13, 81, 80, 0], Distance = 35.00, Time = 3.23s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.20, Time = 2.51s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.40, Time = 10.36s
Cluster 6: Route = [0, 32, 40, 72, 51, 71, 34, 31, 73, 60, 0], Distance = 31.60, Time = 6.13s
Cluster 7: Route = [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0], Distance = 31.90, Time = 12.45s
Cluster 8: Route = [0, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 33, 67, 0], Distance = 33.00, Time = 8.30s
Cluster 9: Route = [0, 23, 21, 79, 24, 30, 22, 28, 16, 0], Distance = 36.90, Time = 4.96s
Total Distance: 308.90, Total Runtime: 55.89s
```

c. Running ACO with small clusters:

```
Running ACO with small clusters:
Cluster 1: Route = [0, 19, 14, 10, 2, 83, 5, 7, 0], Distance = 25.90, Time = 2.06s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.00, Time = 1.49s
Cluster 3: Route = [0, 9, 11, 13, 81, 80, 0], Distance = 35.00, Time = 1.62s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.20, Time = 1.27s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.40, Time = 5.76s
Cluster 6: Route = [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0], Distance = 29.70, Time = 2.82s
Cluster 7: Route = [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0], Distance = 31.90, Time = 6.65s
Cluster 8: Route = [0, 15, 18, 66, 64, 85, 38, 17, 82, 25, 26, 42, 33, 67, 0], Distance = 32.30, Time = 4.21s
Cluster 9: Route = [0, 23, 24, 21, 79, 22, 30, 28, 16, 0], Distance = 36.80, Time = 2.46s
Total Distance: 306.20, Total Runtime: 28.37s
```

d. Running ACO with optimized parameters:

```
Running ACO with optimized parameters:
Cluster 1: Route = [0, 19, 14, 10, 2, 83, 5, 7, 0], Distance = 25.90, Time = 3.12s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.00, Time = 2.51s
Cluster 3: Route = [0, 9, 11, 13, 81, 80, 0], Distance = 35.00, Time = 2.32s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.20, Time = 1.80s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.40, Time = 8.29s
Cluster 6: Route = [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0], Distance = 29.70, Time = 4.49s
Cluster 7: Route = [0, 68, 70, 55, 58, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0], Distance = 33.30, Time = 9.47s
Cluster 8: Route = [0, 15, 18, 66, 64, 85, 38, 17, 42, 82, 25, 26, 33, 67, 0], Distance = 33.20, Time = 6.37s
Cluster 9: Route = [0, 23, 24, 21, 79, 22, 30, 28, 16, 0], Distance = 36.80, Time = 3.89s
Total Distance: 308.50, Total Runtime: 42.26s
```

a. Hybrid ACO with SA

This code combines Ant Colony Optimization (ACO) with Simulated Annealing (SA) to solve optimization problems such as the Traveling Salesman Problem (TSP) or route optimization. ACO is responsible for constructing candidate solutions, while SA refines them using local search to improve convergence and avoid local minima.

Key Components of the Code

Class Initialization

```python
class HybridRoutingAlgorithm:
    def __init__(self, distance_matrix, num_ants, evaporation_rate,
alpha, beta, generations, initial_temperature, cooling_rate):
        self.distance_matrix = distance_matrix
        self.num_ants = num_ants
        self.evaporation_rate = evaporation_rate
        self.alpha = alpha
        self.beta = beta
        self.generations = generations
        self.initial_temperature = initial_temperature
        self.cooling_rate = cooling_rate
        self.pheromone_matrix = np.ones_like(distance_matrix)
        self.best_distance = float('inf')
        self.best_route = None
```

Inputs:

- `distance_matrix`: A 2D array representing distances between nodes.

- `num_ants`: Number of ants used to construct solutions.
- `evaporation_rate`: The rate at which pheromone trails decay.
- `alpha`: Weight for pheromone influence.
- `beta`: Weight for heuristic (distance) influence.
- `generations`: Number of iterations to optimize.
- `initial_temperature`: Starting temperature for SA.
- `cooling_rate`: Reduction factor for temperature per generation.

Outputs:

- `pheromone_matrix`: Tracks pheromone trails for each edge.
- `best_distance` and `best_route`: Store the shortest distance and corresponding route.

Calculating Route Distance

```
def calculate_distance(self, route):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += self.distance_matrix[route[i]][route[i + 1]]
    return total_distance
```

This method computes the total distance for a given route by summing the distances between consecutive nodes.

Solution Construction with ACO

```
def construct_solution(self):
    ants_routes = []
    for _ in range(self.num_ants):
        route = [0]
        visited = set(route)

        while len(visited) < len(self.distance_matrix):
            current_node = route[-1]
            probabilities = self.calculate_probabilities(current_node,
visited)
            next_node = self.select_next_node(probabilities)
            route.append(next_node)
            visited.add(next_node)

        route.append(0)  # Return to starting point
        ants_routes.append(route)
```

```
    return ants_routes
```

Each ant constructs a route:

1. **Start**: At the depot (node 0).
2. **Probabilities**: Compute transition probabilities to unvisited nodes.
3. **Selection**: Choose the next node based on probabilities.
4. **Completion**: Return to the starting node after visiting all nodes.

Probability Calculation

```
def calculate_probabilities(self, current_node, visited):
    pheromone = self.pheromone_matrix[current_node]
    heuristic = 1 / (self.distance_matrix[current_node] + 1e-10)
    heuristic = np.array([0 if i in visited else h for i, h in
enumerate(heuristic)])

    probabilities = (pheromone ** self.alpha) * (heuristic ** self.beta)
    probabilities /= probabilities.sum()
    return probabilities
```

The transition probabilities for an ant are calculated using:

- **Pheromone Levels**: Reflect the attractiveness of paths based on past iterations.
- **Heuristic Values**: Favor shorter distances.
- **Visited Nodes**: Excluded from consideration.

Updating Pheromones

```
def update_pheromones(self, ants_routes):
    self.pheromone_matrix *= (1 - self.evaporation_rate)

    for route in ants_routes:
        distance = self.calculate_distance(route)
        pheromone_deposit = 1 / distance

        for i in range(len(route) - 1):
            self.pheromone_matrix[route[i]][route[i + 1]] +=
pheromone_deposit
```

- **Evaporation**: Pheromone levels are reduced globally.

- **Deposit**: Routes with shorter distances receive higher pheromone deposits.

Simulated Annealing for Local Search

```
def simulated_annealing_local_search(self, solution, temperature):
    current_solution = solution.copy()
    current_distance = self.calculate_distance(current_solution)

    for _ in range(10):  # Local search iterations
        i, j = random.sample(range(1, len(current_solution) - 1), 2)
        new_solution = current_solution.copy()
        new_solution[i], new_solution[j] = new_solution[j],
new_solution[i]

        new_distance = self.calculate_distance(new_solution)

        if (new_distance < current_distance or
            random.random() < math.exp((current_distance - new_distance)
/ temperature)):
            current_solution = new_solution
            current_distance = new_distance

    return current_solution
```

Simulated Annealing refines solutions:

a. **Random Swaps**: Two nodes in the route are swapped.
b. **Acceptance**:
   i. If the new solution is better, it is accepted.
   ii. If worse, it is accepted with a probability proportional to the current temperature (helps escape local minima).
c. **Temperature Cooling**: Reduces exploration as iterations progress.

Execution and Optimization

```
def run(self):
    start_time = time.time()
    temperature = self.initial_temperature

    for generation in range(self.generations):
        temperature *= self.cooling_rate

        ants_routes = self.construct_solution()
```

```
        self.update_pheromones(ants_routes)

        for route in ants_routes:
            route = self.simulated_annealing_local_search(route,
temperature)
            distance = self.calculate_distance(route)

            if distance < self.best_distance:
                self.best_distance = distance
                self.best_route = route

    end_time = time.time()
    runtime = end_time - start_time
    return self.best_route, self.best_distance, runtime
```

a. ACO Phase: Generates initial solutions using probabilistic construction.
b. SA Phase: Improves solutions via local search.
c. Convergence:

- Updates the best solution found during each generation.
- Adjusts the temperature for SA.

Function: Running ACO + SA

```
def run_acosa_with_params(distance_matrix, clusters, ...):
    for i, cluster in enumerate(clusters, 1):
        ...
        solver = HybridRoutingAlgorithm(selected_distance_matrix, ...)
        best_route, best_distance, runtime = solver.run()
        ...
```

For each cluster:

- Extracts a submatrix of distances.
- Runs the hybrid ACO + SA algorithm.
- Outputs the best route and distance for the cluster.

Experiment Configuration

```
def run_acosa_with_params(distance_matrix, clusters, ant_count=20,
iterations=100, alpha=1.0, beta=2.0, pheromone_evaporation=0.1,
initial_temperature=100, cooling_rate=0.95):
```

This function tests the algorithm on multiple clusters:

1. **Default Parameters**:
    - `ant_count=20`, `alpha=1.0`, `beta=2.0`, `evaporation_rate=0.1`, `initial_temperature=100`, `cooling_rate=0.95`.
2. **Large Clusters**:
    - `ant_count=80`, `alpha=2.5`, `beta=6.0`, `evaporation_rate=0.08`, `initial_temperature=120`, `cooling_rate=0.90`.
3. **Small Clusters**:
    - `ant_count=40`, `alpha=1.5`, `beta=4.5`, `evaporation_rate=0.15`, `initial_temperature=100`, `cooling_rate=0.92`.
4. **Optimized Parameters**:
    - `ant_count=60`, `alpha=3.0`, `beta=5.5`, `evaporation_rate=0.07`, `initial_temperature=130`, `cooling_rate=0.85`.

Each configuration measures total distance and runtime for all clusters.

Output

a. Running ACO with Default Parameters

```
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 1.7389254570007324 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 1.0215771198272705 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 0.9890697002410889 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 0.8318986892700195 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]
Shortest distance: 39.4
Runtime: 2.9638278484344482 seconds

Cluster 6:
Route: [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]
Shortest distance: 29.7
Runtime: 2.3511998653411865 seconds

Cluster 7:
Route: [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]
Shortest distance: 31.9
Runtime: 3.8700273036956787 seconds

Cluster 8:
Route: [0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]
Shortest distance: 31.6
Runtime: 2.322457790374756 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 1.4906091690063477 seconds

Total Shortest Distance for all clusters: 303.9
Total Runtime: 17.57959294319153 seconds
```

b. Running ACO + SA with configuration for large clusters:

```
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 7.057946443557739 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 4.1756508350372314 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 4.157806873321533 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 4.7541444301605225 seconds

Cluster 5:
Route: [0, 78, 50, 49, 47, 20, 77, 76, 29, 75, 45, 74, 36, 35, 48, 43, 39, 0]
Shortest distance: 44.60000000000001
Runtime: 10.480684518814087 seconds

Cluster 6:
Route: [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]
Shortest distance: 29.7
Runtime: 6.022103786468506 seconds

Cluster 7:
Route: [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]
Shortest distance: 31.9
Runtime: 11.873292207717896 seconds

Cluster 8:
Route: [0, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 67, 0]
Shortest distance: 32.5
Runtime: 7.727964639663696 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 4.874583721160889 seconds

Total Shortest Distance for all clusters: 310.0
Total Runtime: 61.1241774559021 seconds
```

c. Running ACO + SA with configuration for small clusters:

```
Cluster 1:
Route: [0, 19, 14, 10, 2, 83, 5, 7, 0]
Shortest distance: 25.900000000000002
Runtime: 2.186384439468384 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 1.4696221351623535 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 1.4120380878448486 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 1.3577449321746826 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]
Shortest distance: 39.4
Runtime: 5.474682807922363 seconds

Cluster 6:
Route: [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]
Shortest distance: 29.7
Runtime: 2.8024020195007324 seconds

Cluster 7:
Route: [0, 70, 58, 55, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0]
Shortest distance: 32.9
Runtime: 6.084918737411499 seconds

Cluster 8:
Route: [0, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 67, 0]
Shortest distance: 32.5
Runtime: 4.0851075649261475 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 2.3993241786956787 seconds

Total Shortest Distance for all clusters: 306.29999999999995
Total Runtime: 27.27222490310669 seconds
```

d. Running ACO + SA with optimized parameters:

```
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 2.9435815811157227 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 2.5297701358795166 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 2.60384464263916 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 1.8710319995880127 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 43, 48, 35, 0]
Shortest distance: 44.6
Runtime: 7.986289739608765 seconds

Cluster 6:
Route: [0, 32, 40, 72, 51, 73, 31, 71, 34, 60, 0]
Shortest distance: 31.5
Runtime: 4.053411483764648 seconds

Cluster 7:
Route: [0, 62, 70, 58, 55, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 0]
Shortest distance: 32.9
Runtime: 8.819026947021484 seconds

Cluster 8:
Route: [0, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 67, 0]
Shortest distance: 32.5
Runtime: 5.692310333251953 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 3.482578992843628 seconds

Total Shortest Distance for all clusters: 312.79999999999995
Total Runtime: 39.98184585571289 seconds
```

e. Comparison

```
Perbandingan Hasil Konfigurasi:
--------------------------------------------------
Default: Jarak =303.90, Waktu =17.58s
Large Clusters: Jarak =310.00, Waktu =61.12s
Small Clusters: Jarak =306.30, Waktu =27.27s
Optimized Parameters: Jarak =312.80, Waktu =39.98s
```

a. Hybrid ACO with Tabu Search

The following implementation combines Ant Colony Optimization (ACO) with Tabu Search (TS) to optimize routing problems, such as the Vehicle Routing Problem (VRP). The algorithm leverages ACO for constructing diverse solutions and TS for local search optimization to avoid revisiting previously explored solutions. This detailed breakdown explains the working of the code and its experimental configurations.

Hybrid ACO with Tabu Search: Code Breakdown and Explanation

The following implementation combines Ant Colony Optimization (ACO) with Tabu Search (TS) to optimize routing problems, such as the Vehicle Routing Problem (VRP). The algorithm leverages ACO for constructing diverse solutions and TS for local search optimization to avoid revisiting previously explored solutions. This detailed breakdown explains the working of the code and its experimental configurations.

Class Definition: HybridRoutingAlgorithm

The `HybridRoutingAlgorithm` class integrates ACO and TS to minimize the travel distance for clusters of nodes.

Initialization

```
class HybridRoutingAlgorithm:
    def __init__(self, distance_matrix, num_ants, evaporation_rate,
alpha, beta, iterations, tabu_list_size):
        self.distance_matrix = distance_matrix
        self.num_ants = num_ants
        self.evaporation_rate = evaporation_rate
        self.alpha = alpha
        self.beta = beta
        self.iterations = iterations
        self.tabu_list_size = tabu_list_size
        self.pheromone_matrix = np.ones_like(distance_matrix)
        self.tabu_list = []
        self.best_distance = float('inf')
        self.best_route = None
```

Parameters:

- `distance_matrix`: Represents distances between nodes.
- `num_ants`: Number of ants in each iteration.
- `evaporation_rate`: Determines how quickly pheromones decay.
- `alpha` & `beta`: Influence of pheromone levels and heuristic values.
- `iterations`: Total number of optimization iterations.
- `tabu_list_size`: Maximum size of the tabu list.

Attributes:

- `pheromone_matrix`: Tracks pheromone levels between nodes.
- `tabu_list`: Stores recently explored solutions to prevent cycling.
- `best_distance` & `best_route`: Tracks the global best solution.

Core Functions

1. **Distance Calculation**

```
def calculate_distance(self, route):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += self.distance_matrix[route[i]][route[i + 1]]
    return total_distance
```

This method computes the total distance of a given route.

2. Solution Construction

```
def construct_solution(self):
    ants_routes = []
    for _ in range(self.num_ants):
        route = [0]
        visited = set(route)

        while len(visited) < len(self.distance_matrix):
            current_node = route[-1]
            probabilities = self.calculate_probabilities(current_node,
visited)
            next_node = self.select_next_node(probabilities)
            route.append(next_node)
            visited.add(next_node)

        route.append(0)  # Return to the depot
        ants_routes.append(route)
    return ants_routes
```

Each ant constructs a solution:

- Starts at the depot (node `0`).
- Visits nodes based on probabilities calculated using pheromone levels and heuristic values.
- Returns to the depot after visiting all nodes.

Probability Calculation

```
def calculate_probabilities(self, current_node, visited):
    pheromone = self.pheromone_matrix[current_node]
    heuristic = 1 / (self.distance_matrix[current_node] + 1e-10)
```

```
    heuristic = np.array([0 if i in visited else h for i, h in
enumerate(heuristic)])

    probabilities = (pheromone ** self.alpha) * (heuristic ** self.beta)
    probabilities /= probabilities.sum()
    return probabilities
```

This method computes probabilities for selecting the next node:

- Uses pheromone (`pheromone ** alpha`) and heuristic values (`heuristic ** beta`).
- Ensures visited nodes are excluded from consideration.

4. Pheromone Update

```
def update_pheromones(self, ants_routes):
    self.pheromone_matrix *= (1 - self.evaporation_rate)

    for route in ants_routes:
        distance = self.calculate_distance(route)
        pheromone_deposit = 1 / distance

        for i in range(len(route) - 1):
            self.pheromone_matrix[route[i]][route[i + 1]] +=
pheromone_deposit
```

This function:

- Evaporates pheromone levels globally.
- Updates pheromone trails based on the quality of solutions (inversely proportional to distance).

5. Tabu Search Local Search

```
def tabu_search_local_search(self, solution):
    current_solution = solution.copy()
    current_distance = self.calculate_distance(current_solution)

    for _ in range(self.tabu_list_size):
        best_neighbor = None
        best_neighbor_distance = float('inf')
```

```
        for i in range(1, len(current_solution) - 2):
            for j in range(i + 1, len(current_solution) - 1):
                neighbor = current_solution.copy()
                neighbor[i], neighbor[j] = neighbor[j], neighbor[i]

                if tuple(neighbor) not in self.tabu_list:
                    neighbor_distance =
self.calculate_distance(neighbor)
                    if neighbor_distance < best_neighbor_distance:
                        best_neighbor = neighbor
                        best_neighbor_distance = neighbor_distance

        if best_neighbor:
            current_solution = best_neighbor
            current_distance = best_neighbor_distance
            self.tabu_list.append(tuple(current_solution))

            if len(self.tabu_list) > self.tabu_list_size:
                self.tabu_list.pop(0)

    return current_solution
```

This method performs a local search using Tabu Search:

- Swaps two nodes in the route to find improved solutions.
- Avoids revisiting solutions stored in the tabu list.
- Maintains the size of the tabu list to limit memory usage.

6. Execution

```
def run(self):
    start_time = time.time()

    for _ in range(self.iterations):
        ants_routes = self.construct_solution()
        self.update_pheromones(ants_routes)

        for route in ants_routes:
            route = self.tabu_search_local_search(route)
            distance = self.calculate_distance(route)

            if distance < self.best_distance:
                self.best_distance = distance
```

```
                self.best_route = route

    end_time = time.time()
    runtime = end_time - start_time
    return self.best_route, self.best_distance, runtime
```

The main function:

- Constructs solutions using ACO.
- Optimizes them using TS.
- Tracks and updates the best solution over iterations.

Experiment Configuration

```
def run_acotabu_with_params(distance_matrix, clusters, ant_count=20,
iterations=100, alpha=1.0, beta=2.0, pheromone_evaporation=0.1,
tabu_list_size=10):
```

This function runs the ACO with Tabu Search for various configurations:

- **Default Parameters**:

a. `ant_count=20`, `alpha=1.0`, `beta=2.0`, `evaporation_rate=0.1`, `tabu_list_size=10`.

- **Large Clusters**:

a. `ant_count=80`, `alpha=2.5`, `beta=6.0`, `evaporation_rate=0.08`, `tabu_list_size=15`.

- **Small Clusters**:

a. `ant_count=40`, `alpha=1.5`, `beta=4.5`, `evaporation_rate=0.15`, `tabu_list_size=5`.

- **Optimized Parameters**:

a. `ant_count=60`, `alpha=3.0`, `beta=5.5`, `evaporation_rate=0.07`, `tabu_list_size=10`.

Output :

a. Running ACO + Tabu Search with default parameters:

```
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 6.509325981140137 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 1.6337385177612305 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 1.3651485443115234 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 0.9084558486938477 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]
Shortest distance: 39.4
Runtime: 27.914079189300537 seconds

Cluster 6:
Route: [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]
Shortest distance: 29.5
Runtime: 6.715250015258789 seconds

Cluster 7:
Route: [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 84, 1, 65, 27, 62, 58, 55, 70, 0]
Shortest distance: 32.5
Runtime: 38.970125675201416 seconds

Cluster 8:
Route: [0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]
Shortest distance: 31.6
Runtime: 15.672924518585205 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 4.1859331130981445 seconds

Total Shortest Distance for all clusters: 304.30000000000007
Total Runtime: 103.87498140335083 seconds
```

b. Running ACO + Tabu Search with configuration for large clusters:

```
Cluster 1:
Route: [0, 7, 5, 83, 2, 14, 10, 19, 0]
Shortest distance: 25.900000000000002
Runtime: 18.826735973358154 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 7.459964990615845 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 8.215800285339355 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 2.928624153137207 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]
Shortest distance: 39.4
Runtime: 177.31053280830383 seconds

Cluster 6:
Route: [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]
Shortest distance: 29.7
Runtime: 35.583152294158936 seconds

Cluster 7:
Route: [0, 68, 58, 55, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 70, 0]
Shortest distance: 32.8
Runtime: 230.47496461868286 seconds

Cluster 8:
Route: [0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 38, 33, 67, 0]
Shortest distance: 31.800000000000004
Runtime: 93.77909779548645 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 25.89561438560486 seconds

Total Shortest Distance for all clusters: 305.5
Total Runtime: 600.4744873046875 seconds
```

c. Running ACO + Tabu Search with configuration for small clusters:

```
Cluster 1:
Route: [0, 19, 14, 10, 2, 7, 5, 83, 0]
Shortest distance: 25.4
Runtime: 4.27939772605896 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 2.5589382648468018 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 2.098463535308838 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 1.4988112449645996 seconds

Cluster 5:
Route: [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]
Shortest distance: 39.4
Runtime: 29.480548858642578 seconds

Cluster 6:
Route: [0, 32, 72, 40, 51, 73, 31, 34, 71, 60, 0]
Shortest distance: 30.099999999999998
Runtime: 7.884328126907349 seconds

Cluster 7:
Route: [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]
Shortest distance: 31.9
Runtime: 40.71906328201294 seconds

Cluster 8:
Route: [0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 33, 38, 67, 0]
Shortest distance: 31.100000000000005
Runtime: 17.032796144485474 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 6.159659385681152 seconds

Total Shortest Distance for all clusters: 303.80000000000007
Total Runtime: 111.71200656890869 seconds
```

d. Running ACO + Tabu Search with optimized parameters:

```
Cluster 1:
Route: [0, 7, 5, 83, 2, 14, 10, 19, 0]
Shortest distance: 25.900000000000002
Runtime: 10.221870183944702 seconds

Cluster 2:
Route: [0, 12, 8, 3, 4, 6, 0]
Shortest distance: 43.0
Runtime: 4.265109300613403 seconds

Cluster 3:
Route: [0, 81, 13, 9, 11, 80, 0]
Shortest distance: 34.8
Runtime: 4.311056852340698 seconds

Cluster 4:
Route: [0, 37, 46, 41, 44, 0]
Shortest distance: 32.2
Runtime: 3.3773317337036133 seconds

Cluster 5:
Route: [0, 39, 50, 49, 77, 20, 76, 29, 78, 75, 45, 47, 74, 36, 35, 48, 43, 0]
Shortest distance: 40.50000000000001
Runtime: 82.48582530021667 seconds

Cluster 6:
Route: [0, 32, 72, 40, 51, 73, 31, 34, 71, 60, 0]
Shortest distance: 30.099999999999998
Runtime: 18.500230312347412 seconds

Cluster 7:
Route: [0, 58, 55, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 70, 0]
Shortest distance: 32.699999999999996
Runtime: 117.07038593292236 seconds

Cluster 8:
Route: [0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]
Shortest distance: 31.6
Runtime: 48.14602017402649 seconds

Cluster 9:
Route: [0, 23, 24, 21, 79, 30, 22, 28, 16, 0]
Shortest distance: 35.900000000000006
Runtime: 14.212594985961914 seconds

Total Shortest Distance for all clusters: 306.70000000000005
Total Runtime: 302.59042477607727 seconds
```

e. Comparison :

```
Perbandingan Hasil Konfigurasi:
--------------------------------------------------
Default: Jarak =304.30, Waktu =103.87s
Large Clusters: Jarak =305.50, Waktu =600.47s
Small Clusters: Jarak =303.80, Waktu =111.71s
Optimized Parameters: Jarak =306.70, Waktu =302.59s
```

b. Hybrid ACO with PSO

The following implementation combines Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) to solve routing problems, such as the Vehicle Routing Problem (VRP). ACO constructs initial solutions, while PSO refines them by updating particle velocities and positions to explore better solutions iteratively. This breakdown explains the working and experimental configurations.

Class Definition: HybridACOPSORoutingAlgorithm

The `HybridACOPSORoutingAlgorithm` class implements the hybrid algorithm by integrating ACO and PSO functionalities.

Initialization

```
class HybridACOPSORoutingAlgorithm:
    def __init__(self, distance_matrix, num_ants, evaporation_rate,
alpha, beta, iterations, w, c1, c2):
        self.distance_matrix = distance_matrix
        self.num_ants = num_ants
        self.evaporation_rate = evaporation_rate
        self.alpha = alpha
        self.beta = beta
        self.iterations = iterations
        self.w = w  # Inertia weight
        self.c1 = c1  # Cognitive coefficient
        self.c2 = c2  # Social coefficient
        self.pheromone_matrix = np.ones_like(distance_matrix)
        self.particles = self.initialize_pso_particles()
        self.best_distance = float('inf')
        self.best_route = None
```

- **Parameters**:
  - `distance_matrix`: Matrix of distances between nodes.
  - `num_ants`: Number of ants for ACO.
  - `evaporation_rate`: Rate of pheromone decay.
  - `alpha` & `beta`: Influence of pheromone levels and heuristic values.
  - `iterations`: Number of optimization iterations.
  - `w`, `c1`, `c2`: PSO parameters for particle inertia, cognitive, and social components.
- **Attributes**:
  - `pheromone_matrix`: Stores pheromone values for ACO.
  - `particles`: Represents particle swarm initialized for PSO.

Core Functions

1. **PSO Particle Initialization**

```
def initialize_pso_particles(self):
```

```
    particles = []
    for _ in range(self.num_ants):
        cluster_nodes = list(range(1, len(self.distance_matrix)))
        random.shuffle(cluster_nodes)
        position = [0] + cluster_nodes + [0]
        velocity = [random.uniform(-1, 1) for _ in range(len(position) -
2)]
        particles.append({
            'position': position,
            'velocity': velocity,
            'personal_best_position': position.copy(),
            'personal_best_fitness': self.calculate_fitness(position)
        })
    return particles
```

Each particle:

- Starts with a random position (route) that begins and ends at the depot (node `0`).
- Has a velocity vector to determine position updates.
- Tracks its personal best position and fitness.

2. ACO Solution Construction

```
def construct_solution(self):
    ants_routes = []
    for _ in range(self.num_ants):
        route = [0]
        visited = set(route)

        while len(visited) < len(self.distance_matrix):
            current_node = route[-1]
            probabilities = self.calculate_probabilities(current_node,
visited)
            next_node = self.select_next_node(probabilities)
            route.append(next_node)
            visited.add(next_node)

        route.append(0)  # Return to starting point
        ants_routes.append(route)
    return ants_routes
```

Each ant constructs a solution:

- Visits nodes probabilistically based on pheromone levels and heuristic values.
- Ensures all nodes are visited before returning to the depot.

3. Probability Calculation for ACO

```
def calculate_probabilities(self, current_node, visited):
    pheromone = self.pheromone_matrix[current_node]
    heuristic = 1 / (self.distance_matrix[current_node] + 1e-10)
    heuristic = np.array([0 if i in visited else h for i, h in
enumerate(heuristic)])

    probabilities = (pheromone ** self.alpha) * (heuristic ** self.beta)
    probabilities /= probabilities.sum()
    return probabilities
```

This function calculates the probability of selecting the next node based on:

- **Pheromone trails**: Encourages paths explored by previous ants.
- **Heuristic values**: Favors shorter distances.

4. Pheromone Update

```
def update_pheromones(self, ants_routes):
    self.pheromone_matrix *= (1 - self.evaporation_rate)

    for route in ants_routes:
        distance = self.calculate_distance(route)
        pheromone_deposit = 1 / (distance + 1e-10)

        for i in range(len(route) - 1):
            self.pheromone_matrix[route[i]][route[i + 1]] +=
pheromone_deposit
```

Pheromone levels are updated:

- Decay globally based on `evaporation_rate`.
- Increased on paths that lead to better solutions.

5. PSO Particle Updates

```
def pso_update_particles(self, global_best_route):
    for particle in self.particles:
```

```
        for i in range(len(particle['velocity'])):
            r1, r2 = random.random(), random.random()
            cognitive_component = self.c1 * r1 *
(particle['personal_best_position'][i + 1] - particle['position'][i +
1])
            social_component = self.c2 * r2 * (global_best_route[i + 1]
- particle['position'][i + 1])
            particle['velocity'][i] = self.w * particle['velocity'][i] +
cognitive_component + social_component

        new_position = particle['position'].copy()
        for i in range(1, len(new_position) - 1):
            new_position[i] = max(1, min(len(self.distance_matrix) - 1,
round(new_position[i] + particle['velocity'][i - 1])))

        unique_position = [0] + list(dict.fromkeys(new_position[1:-1]))
+ [0]
        particle['position'] = unique_position

        current_fitness = self.calculate_fitness(unique_position)
        if current_fitness > particle['personal_best_fitness']:
            particle['personal_best_position'] = unique_position
            particle['personal_best_fitness'] = current_fitness
```

Particles update their positions using:

- **Cognitive component**: Attraction towards their best-known position.
- **Social component**: Attraction towards the global best position.

6. Execution

```
def run(self):
    start_time = time.time()
    for _ in range(self.iterations):
        ants_routes = self.construct_solution()
        self.update_pheromones(ants_routes)

        global_best_route = max(self.particles + [{'position': r} for r
in ants_routes],
                                key=lambda p:
self.calculate_fitness(p['position']))['position']
        self.pso_update_particles(global_best_route)

        best_distance = self.calculate_distance(global_best_route)
```

```
        if best_distance < self.best_distance:
            self.best_distance = best_distance
            self.best_route = global_best_route

    end_time = time.time()
    return self.best_route, self.best_distance, end_time - start_time
```

The algorithm alternates between:

- **Solution construction** (ACO phase).
- **Particle updates** (PSO phase).

Experiment Configuration

```
def run_acopso_with_params(distance_matrix, clusters, ant_count=20,
iterations=100, alpha=1.0, beta=2.0, pheromone_evaporation=0.1, w=0.5,
c1=1.5, c2=1.5):
```

This function tests different configurations of ACO-PSO:

1. **Default Parameters**:
    - `ant_count=20`, `alpha=1.0`, `beta=2.0`, `w=0.5`, `c1=1.5`, `c2=1.5`.
2. **Large Clusters**:
    - `ant_count=80`, `alpha=2.5`, `beta=6.0`, `w=0.7`, `c1=2.0`, `c2=2.0`.
3. **Small Clusters**:
    - `ant_count=40`, `alpha=1.5`, `beta=4.5`, `w=0.4`, `c1=1.2`, `c2=1.2`.
4. **Intensive Search**:
    - `ant_count=100`, `alpha=2.0`, `beta=5.5`, `w=0.6`, `c1=1.8`, `c2=1.8`.

Output

a. Running ACO-PSO with default parameters:

```
Running ACO-PSO with default parameters:
Cluster 1: Route = [0, 19, 14, 10, 2, 7, 5, 83, 0], Distance = 25.4, Time = 1.00s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.0, Time = 0.82s
Cluster 3: Route = [0, 81, 13, 9, 11, 80, 0], Distance = 34.8, Time = 0.82s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.2, Time = 0.61s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.4, Time = 2.75s
Cluster 6: Route = [0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0], Distance = 29.5, Time = 1.43s
Cluster 7: Route = [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0], Distance = 31.9, Time = 3.34s
Cluster 8: Route = [0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 38, 33, 67, 0], Distance = 31.800000000000004, Time = 2.62s
Cluster 9: Route = [0, 23, 24, 21, 79, 30, 22, 28, 16, 0], Distance = 35.900000000000006, Time = 1.39s
Total Distance: 303.9, Total Time: 14.77s
```

b. Running ACO-PSO with large clusters:

```
Running ACO-PSO with large clusters:
Cluster 1: Route = [0, 19, 14, 10, 2, 7, 5, 83, 0], Distance = 25.4, Time = 4.66s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.0, Time = 3.23s
Cluster 3: Route = [0, 81, 13, 9, 11, 80, 0], Distance = 34.8, Time = 3.70s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.2, Time = 2.65s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.4, Time = 11.74s
Cluster 6: Route = [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0], Distance = 29.7, Time = 5.92s
Cluster 7: Route = [0, 70, 55, 68, 69, 63, 56, 58, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0], Distance = 33.2, Time = 12.73s
Cluster 8: Route = [0, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 33, 67, 0], Distance = 33.0, Time = 9.30s
Cluster 9: Route = [0, 23, 24, 21, 79, 30, 22, 28, 16, 0], Distance = 35.900000000000006, Time = 4.95s
Total Distance: 306.6, Total Time: 58.88s
```

c. Running ACO-PSO with small clusters:

```
Running ACO-PSO with large clusters:
Cluster 1: Route = [0, 19, 14, 10, 2, 7, 5, 83, 0], Distance = 25.4, Time = 4.66s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.0, Time = 3.23s
Cluster 3: Route = [0, 81, 13, 9, 11, 80, 0], Distance = 34.8, Time = 3.70s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.2, Time = 2.65s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.4, Time = 11.74s
Cluster 6: Route = [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0], Distance = 29.7, Time = 5.92s
Cluster 7: Route = [0, 70, 55, 68, 69, 63, 56, 58, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0], Distance = 33.2, Time = 12.73s
Cluster 8: Route = [0, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 33, 67, 0], Distance = 33.0, Time = 9.30s
Cluster 9: Route = [0, 23, 24, 21, 79, 30, 22, 28, 16, 0], Distance = 35.900000000000006, Time = 4.95s
Total Distance: 306.6, Total Time: 58.88s
```

d. Running ACO-PSO with intensive search:

```
Running ACO-PSO with intensive search:
Cluster 1: Route = [0, 19, 14, 10, 2, 7, 5, 83, 0], Distance = 25.4, Time = 5.26s
Cluster 2: Route = [0, 12, 8, 3, 4, 6, 0], Distance = 43.0, Time = 4.00s
Cluster 3: Route = [0, 81, 13, 9, 11, 80, 0], Distance = 34.8, Time = 4.49s
Cluster 4: Route = [0, 37, 46, 41, 44, 0], Distance = 32.2, Time = 3.18s
Cluster 5: Route = [0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0], Distance = 39.4, Time = 13.87s
Cluster 6: Route = [0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0], Distance = 29.7, Time = 7.38s
Cluster 7: Route = [0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0], Distance = 31.9, Time = 16.20s
Cluster 8: Route = [0, 67, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 0], Distance = 32.900000000000006, Time = 11.36s
Cluster 9: Route = [0, 23, 24, 21, 79, 22, 30, 28, 16, 0], Distance = 36.8, Time = 6.42s
Total Distance: 306.1, Total Time: 72.15s
```

e. Comparison of Results:

```
Comparison of Results:
Default: Distance = 303.90, Time = 14.77s
Large Clusters: Distance = 306.60, Time = 58.88s
Small Clusters: Distance = 305.20, Time = 30.05s
Intensive Search: Distance = 306.10, Time = 72.15s
```

c. Simulated Annealing

3A) Simulated Annealing (Code)

```python
def read_csv_with_flexible_columns(file_path, expected_columns=87):
    data = []
    with open(file_path, 'r') as file:
        for line in file:
            # Split by commas
            row = line.strip().split(',')
            # If the row has the correct number of columns, add to
data
```

```
            if len(row) == expected_columns:
                data.append([float(x) for x in row])
    return np.array(data)

lat_long_df = pd.read_csv('lat_long.csv')

nodes = lat_long_df['node'].values

distance_matrix = read_csv_with_flexible_columns('xy_distances.csv',
expected_columns=87)

print("Distance matrix shape:", distance_matrix.shape)
```

'read_csv_with_flexible_columns' reads a CSV file while ensuring that each row has the expected number of columns (set to 87 by default). The function filters rows with the incorrect number of columns, parsing valid data and returning it as a NumPy array. This is useful when the dataset might have some variability in its structure.

'lat_long_df' loads the dataset that contains the node locations in Cartesian coordinate form (latitudes and longitudes). The nodes are extracted and stored in 'nodes'. The size of the 'distance_matrix' is checked using the 'read_csv_with_flexible_columns' function for the coordinates dataset, and this matrix is used to calculate the distances between nodes within 87 nodal objects.

```
def calculate_total_distance(route, distance_matrix):
    total_distance = 0
    for i in range(len(route) - 1):
        node1 = route[i]
        node2 = route[i + 1]
        total_distance += distance_matrix[node1, node2]
    # Return to the starting point
    total_distance += distance_matrix[route[-1], route[0]]
    return total_distance
```

The function 'calculate_total_distance' computes the total distance for a given route by summing the distances between consecutive nodes (using the distance matrix). It also adds the distance from the last node to the first to complete the cycle iteratively before returning to the starting point per cluster (Node 0).

```
def get_neighbor(route):
    neighbor = route.copy()
    # Ensure the start node remains fixed
```

```
    i, j = random.sample(range(1, len(route)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor
```

The ‘get_neighbor’ function generates a neighboring solution by swapping two random nodes in the current route. The starting node remains fixed so that each cluster “restarts” before the exploration phase.

```
def simulated_annealing(distance_matrix, initial_temp=1000,
cooling_rate=0.995, max_iter=100):
    # Start with Node 0 as the fixed starting point
    route = [0] + list(range(1, len(distance_matrix)))
    random.shuffle(route[1:])  # Shuffle other nodes but keep Node 0

    current_distance = calculate_total_distance(route,
distance_matrix)
    best_route = route.copy()
    best_distance = current_distance

    temp = initial_temp

    start_time = time.time()  # Track runtime

    for iteration in range(max_iter):
        # Generate a neighboring solution
        neighbor_route = get_neighbor(route)
        neighbor_distance = calculate_total_distance(neighbor_route,
distance_matrix)

        # If the neighbor is better, accept it
        if neighbor_distance < current_distance:
            route = neighbor_route
            current_distance = neighbor_distance
        # If neighbor is worse, accept with probability : temperature
        else:
            acceptance_prob = math.exp((current_distance -
neighbor_distance) / temp)
            if random.random() < acceptance_prob:
                route = neighbor_route
                current_distance = neighbor_distance

        if current_distance < best_distance:
```

```
            best_route = route.copy()
            best_distance = current_distance

        temp *= cooling_rate

        if temp < 0.1:
            break

    end_time = time.time()  # End runtime
    runtime = end_time - start_time

    return best_route, best_distance, runtime
```

The ‘simulated_annealing’ function implements the core optimization algorithm by initializing a randomized route while preserving the fixed starting node. The algorithm iteratively explores the solution space by generating neighboring solutions and evaluating their distances. A two-tier acceptance mechanism, based on improvement or probabilistic favorability, ensures that the algorithm avoids premature convergence and can escape the local minima. The cooling schedule, governed by ‘cooling_rate’, gradually reduces the temperature, controlling the probability of accepting suboptimal solutions. The function's use of timing functions also enables performance tracking.

```
# Run the Simulated Annealing algorithm with default parameters
print('\n' + "1: SA with Default Config: \n")

# Parameters for default settings
max_iterations = 100
initial_temp = 1000
cooling_rate = 0.995

print("Parameters:")
print(f"Max Iterations: {max_iterations}")
print(f"Initial Temperature: {initial_temp}")
print(f"Cooling Rate: {cooling_rate}")
print("=" * 80)

# Simulate for different clusters (initial smaller cluster size)
cluster_count = 9  # Adjust based on your needs
default_total_distance = 0
default_total_runtime = 0
```

This section initiates the execution of the SA algorithm with its default configuration parameters and sets the stage for evaluating its performance across multiple clusters. It begins

by defining the primary parameters for the algorithm: the maximum number of iterations is set to 100, the initial temperature to 1000, and the cooling rate to 0.995. These values are designed to balance exploration and convergence within the solution space. Following the parameter initialization, the output is formatted to present the configuration details to the user, ensuring clarity and reproducibility. The `cluster_count` variable is set to 9, indicating the number of clusters over which the algorithm will be applied. Additionally, two accumulators, `default_total_distance` and `default_total_runtime`, are initialized to zero. These will aggregate the total distance covered and runtime consumed across all clusters, enabling a comprehensive performance analysis. The use of clusters allows the algorithm to operate on subsets of the problem. This significantly improves the algorithm's computational efficiency and facilitates scalability in handling larger datasets.

```
for cluster_id in range(1, cluster_count + 1):
    print(f"Processing Cluster {cluster_id}:")

    try:
        # Select nodes for the current cluster
        start_index = (cluster_id - 1) * (len(nodes) //
cluster_count)
        end_index = cluster_id * (len(nodes) // cluster_count)

        cluster_nodes = list(range(start_index, end_index))
        cluster_distance_matrix = distance_matrix[cluster_nodes,
:][:, cluster_nodes]

        # Run Simulated Annealing for the current cluster
        best_route, best_distance, runtime =
simulated_annealing(cluster_distance_matrix, initial_temp,
cooling_rate, max_iterations)

        # Convert the best route (indices) to actual node names
        best_nodes = nodes[cluster_nodes][best_route]

        # Output for the current cluster
        print(f"Best Route: {best_nodes}")
        print(f"Distance: {best_distance:.2f}")
        print(f"Runtime: {runtime:.4f} seconds")
        print("-" * 80 + '\n')

        # Accumulate the total distance and runtime for all clusters
        default_total_distance += best_distance
        default_total_runtime += runtime
```

```
    except IndexError as e:
        print(f"IndexError encountered in Cluster {cluster_id}." +
"\n" "Skipping this cluster...\n")
        continue

# Exclude Node 53 (Africa) from the distance matrix
distance_matrix = np.delete(distance_matrix, 53, axis=0)  # Remove row 53
distance_matrix = np.delete(distance_matrix, 53, axis=1)  # Remove column 53

# Final metrics for small clusters
avg_distance_small = small_total_distance / cluster_count_small
avg_runtime_small = small_total_runtime / cluster_count_small

print('\n \n \n')
# Final metrics for the default configuration
print("Metrics for Default Configuration:")
print(f"Total Distance (Default): {default_total_distance:.2f}")
print(f"Total Runtime (Default): {default_total_runtime:.4f}
seconds")
print("=" * 80)
```

Here, the algorithm iterates through a predefined number of clusters (`cluster_count`) to execute the Simulated Annealing (SA) algorithm on smaller subsets of the entire dataset, enabling modular analysis of the optimization problem. The loop begins by iterating through each cluster ID, ranging from 1 to `cluster_count`. For each cluster, the code selects the corresponding subset of nodes to create a localized distance matrix, allowing for cluster-specific optimization. The node indices for the current cluster are calculated based on the cluster ID, segmenting the data evenly across clusters. Using these indices, the code extracts a subset of the distance matrix, `cluster_distance_matrix`, which represents inter-node distances only for the current cluster. Within each iteration, the `simulated_annealing` function is called with the localized distance matrix and the default SA parameters (initial temperature, cooling rate, and maximum iterations). This function returns the optimized route (`best_route`), its associated distance (`best_distance`), and the runtime of the computation (`runtime`). The indices of the optimal route are subsequently mapped back to their corresponding node names for interpretability and are displayed alongside the distance and runtime for the current cluster.

The algorithm then updates two cumulative variables, `default_total_distance` and `default_total_runtime`, with the respective outputs from the current cluster's optimization results. These accumulators provide aggregated metrics for the performance of the SA algorithm across all clusters, right before error handling is incorporated to address potential `IndexError` exceptions that might arise from cluster boundary misalignments or other data-related inconsistencies. If such an error is encountered, the code prints a warning message, skips the problematic cluster, and continues the loop, ensuring robustness without compromising the execution of other clusters.

The code transitions from evaluating the performance of the Simulated Annealing (SA) algorithm with default cluster parameters to preparing for an alternative configuration with larger clusters and adjusted algorithm parameters. After processing the default configuration, the script outputs the aggregated metrics. Specifically, the total distance and runtime across all clusters are displayed using formatted strings, which highlight the numerical precision of the results. These outputs serve as baseline metrics for comparison with subsequent configurations.

```
# Now run SA with large clusters
print('\n' + "2: SA with Large Config: \n")

# Parameters for large clusters
max_iterations_large = 100
initial_temp_large = 1500  # Larger initial temperature
cooling_rate_large = 0.98  # Slower cooling rate

print("Parameters:")
print(f"Max Iterations: {max_iterations_large}")
print(f"Initial Temperature: {initial_temp_large}")
print(f"Cooling Rate: {cooling_rate_large}")
print("=" * 80)

# Simulate for large clusters
cluster_count_large = 5  # Fewer but larger clusters
large_total_distance = 0
large_total_runtime = 0
```

The second configuration, labeled "SA with Large Config," modifies the SA parameters to suit fewer but larger clusters. Key adjustments include increasing the initial temperature to 1500, which allows for a broader exploration of the solution space at the start, and reducing the cooling rate to 0.98, which slows the transition to convergence and potentially avoids premature optimization plateaus. The number of clusters is reduced to five, creating larger data segments that each encompass more nodes. These changes are intended to challenge the algorithm with more complex cluster structures, potentially revealing different performance characteristics.

In order to capture the performance metrics under the new configuration, cumulative variables `large_total_distance` and `large_total_runtime` are initialized to zero. These variables will store the aggregated results for all large clusters, facilitating a direct comparison with the metrics from the default configuration.

```
for cluster_id in range(1, cluster_count_large + 1):
    print(f"Processing Cluster {cluster_id}:")
```

```
    try:
        # Select nodes for the current larger cluster
        start_index = (cluster_id - 1) * (len(nodes) //
cluster_count_large)
        end_index = cluster_id * (len(nodes) // cluster_count_large)

        cluster_nodes = list(range(start_index, end_index))
        cluster_distance_matrix = distance_matrix[cluster_nodes,
:][:, cluster_nodes]

        # Run Simulated Annealing for the current larger cluster
        best_route, best_distance, runtime =
simulated_annealing(cluster_distance_matrix, initial_temp_large,
cooling_rate_large, max_iterations_large)

        # Convert the best route (indices) to actual node names
        best_nodes = nodes[cluster_nodes][best_route]

        # Output for the current cluster
        print(f"Best Route: {best_nodes}")
        print(f"Distance: {best_distance:.2f}")
        print(f"Runtime: {runtime:.4f} seconds")
        print("-" * 80 + '\n')

        # Accumulate the total distance and runtime for all larger
clusters
        large_total_distance += best_distance
        large_total_runtime += runtime

    except IndexError as e:
        print(f"IndexError encountered in Cluster {cluster_id}." +
"\n" + "Skipping this cluster...\n")
        continue

# Final metrics
# Averaging over all clusters for default and larger configurations
avg_distance_default = default_total_distance / cluster_count
avg_runtime_default = default_total_runtime / cluster_count
avg_distance_large = large_total_distance / cluster_count_large
avg_runtime_large = large_total_runtime / cluster_count_large
```

The algorithm processes a new experimental setup of the SA with larger clusters, iterating through each cluster as defined by the `cluster_count_large` parameter. For each cluster, the algorithm begins by calculating the starting and ending indices that define the range of nodes included in the current cluster. These indices segment the `nodes` array and the corresponding rows and columns of the `distance_matrix`, isolating a subset of data specific to the cluster. This segmentation creates the `cluster_distance_matrix`, which represents the internal distance relationships within the current cluster. Once the cluster's distance matrix is established, the SA algorithm is invoked for optimization. The function `simulated_annealing` is called with parameters tailored for larger clusters, such as a higher initial temperature (`initial_temp_large`), a slower cooling rate (`cooling_rate_large`), and an extended iteration limit (`max_iterations_large`). The algorithm computes the optimal route, the total distance for the best route, and the runtime for processing the cluster. The results are returned as `best_route`, `best_distance`, and `runtime`.

After obtaining the results, the best route, represented as an array of indices, is translated into the corresponding node names using the `nodes` array. The output displays the optimal route in terms of node names, the computed distance rounded to two decimal places, and the runtime rounded to four decimal places. A formatted separator line follows to clearly distinguish the results of each cluster iteration. The script accumulates the total distance and runtime for all processed clusters into `large_total_distance` and `large_total_runtime`, respectively. These variables store the aggregated performance metrics for the large-cluster configuration. If an error arises during the processing of a cluster—such as an `IndexError` caused by uneven data distribution or improper indexing—the exception is caught, and the code skips to the next cluster. A message is printed to inform the user of the encountered error, ensuring clarity in troubleshooting.

The final metrics computation calculates averages over all clusters for the previously analyzed configurations: default and larger clusters. This is achieved by dividing the total distances and runtimes by their respective cluster counts, producing the average distance (`avg_distance_default` and `avg_distance_large`) and average runtime (`avg_runtime_default` and `avg_runtime_large`). These metrics provide a normalized basis for comparing the efficiency and effectiveness of the Simulated Annealing (SA) algorithm across varying cluster configurations.

```
# Now, handle the small clusters
print('\n' + "3: SA with Small Config: \n")

# Parameters for small clusters
max_iterations_small = 100
initial_temp_small = 800  # Lower initial temperature
cooling_rate_small = 0.99  # Faster cooling rate

print("Parameters:")
print(f"Max Iterations: {max_iterations_small}")
print(f"Initial Temperature: {initial_temp_small}")
```

```
print(f"Cooling Rate: {cooling_rate_small}")
print("=" * 80)

# Simulate for small clusters
cluster_count_small = 12  # More but smaller clusters
small_total_distance = 0
small_total_runtime = 0
```

Following the computation of averages, the code initiates the configuration and execution of SA for small clusters, labeled "Small Config." The parameters for this setup are defined with a lower initial temperature (`initial_temp_small`) of 800 and a faster cooling rate (`cooling_rate_small`) of 0.99. These settings are designed to converge more quickly on an optimized solution, assuming that smaller clusters require less computational effort due to their reduced size. The iteration limit (`max_iterations_small`) is maintained at 100 to ensure consistency in termination conditions across configurations.

The `cluster_count_small` parameter is set to 12, representing a higher number of clusters with fewer nodes in each. This setup emphasizes granularity in route optimization, allowing the algorithm to focus on smaller, more localized groups of nodes. Two variables, `small_total_distance` and `small_total_runtime`, are initialized to zero to accumulate the total performance metrics for all small clusters during the iterative execution.

```
# Now, handle the small clusters
print('\n' + "3: SA with Small Config: \n")

# Parameters for small clusters
max_iterations_small = 100
initial_temp_small = 800  # Lower initial temperature
cooling_rate_small = 0.99  # Faster cooling rate

print("Parameters:")
print(f"Max Iterations: {max_iterations_small}")
print(f"Initial Temperature: {initial_temp_small}")
print(f"Cooling Rate: {cooling_rate_small}")
print("=" * 80)

# Simulate for small clusters
cluster_count_small = 12  # More but smaller clusters
small_total_distance = 0
small_total_runtime = 0
```

The iterative loop for processing smaller clusters begins by iterating through `cluster_count_small`, which is set to 12. This ensures the division of nodes into a greater

number of smaller clusters. Each iteration corresponds to processing a specific cluster, identified by its `cluster_id`. The start and end indices for nodes within the current cluster are calculated using proportional segmentation of the `nodes` array, with the start index defined as `(cluster_id - 1) * (len(nodes) // cluster_count_small)` and the end index as `cluster_id * (len(nodes) // cluster_count_small)`. This allows for the systematic selection of nodes for each smaller cluster.

The `cluster_nodes` variable is created as a list containing indices for the nodes in the current cluster, and the corresponding subset of the `distance_matrix` is extracted using NumPy slicing to form `cluster_distance_matrix`. This matrix provides the inter-node distances within the selected cluster and serves as input for the Simulated Annealing (SA) algorithm. The `simulated_annealing` function is invoked with the current cluster's distance matrix and the predefined parameters for small clusters: `initial_temp_small` (800), `cooling_rate_small` (0.99), and `max_iterations_small` (100). The function returns the optimized route (`best_route`), the associated minimum distance (`best_distance`), and the runtime for convergence (`runtime`). The `best_route`, initially represented as indices, is mapped back to actual node names using `nodes[cluster_nodes][best_route]`.

Outputs for the current cluster include the best route, the optimized distance (formatted to two decimal places), and the runtime (formatted to four decimal places). These are printed to the console for detailed monitoring. Additionally, the cumulative metrics for the small clusters are updated by adding the `best_distance` to `small_total_distance` and the `runtime` to `small_total_runtime`. In the event of an `IndexError`, which could occur due to uneven node distribution among clusters, the error is caught, and the corresponding cluster is skipped with a printed notification. After completing all iterations, the final metrics for the small clusters are computed. The average distance (`avg_distance_small`) is calculated as the total distance divided by the number of clusters, and the average runtime (`avg_runtime_small`) is determined by dividing the total runtime by the number of clusters.

```
print('\n \n \n')

# Display all metrics
print("\nMetrics for Default Configuration: \n")
print(f"Avg Distance: {avg_distance_default:.2f}")
print(f"Avg Runtime: {avg_runtime_default:.4f}s" + '\n')
print(f"Total Distance: {default_total_distance:.2f}")
print(f"Total Runtime: {default_total_runtime:.4f}s")
print('\n')

print("\nMetrics for Large Cluster Configuration: \n")
print(f"Avg Distance: {avg_distance_large:.2f}")
print(f"Avg Runtime: {avg_runtime_large:.4f}s" + '\n')
print(f"Total Distance: {large_total_distance:.2f}")
print(f"Total Runtime: {large_total_runtime:.4f}s")
```

```
print('\n')

print("\nMetrics for Small Cluster Configuration: \n")
print(f"Avg Distance: {avg_distance_small:.2f}")
print(f"Avg Runtime: {avg_runtime_small:.4f}s" + '\n')
print(f"Total Distance: {small_total_distance:.2f}")
print(f"Total Runtime: {small_total_runtime:.4f}s")
```

The final code block outputs the performance metrics of the Simulated Annealing (SA) algorithm across different configurations, specifically the default, large, and small cluster setups. It begins by printing a set of blank lines to separate the output for clarity. For each configuration, the average distance and average runtime are displayed first. The average distance represents the mean of all cluster distances, calculated by dividing the total distance by the number of clusters for each configuration. Similarly, the average runtime is computed by dividing the total runtime by the respective cluster count, providing an overview of the efficiency of the algorithm.

Following the averages, the total distance and total runtime for each configuration are printed, giving a summary of the algorithm's performance across all clusters in each setup. These metrics offer insight into the overall optimization performance and the computational cost for the default, large, and small cluster configurations. The output is formatted clearly, with the results printed in a manner that allows for easy comparison between the different configurations.

3B) Simulated Annealing (Output)

3.2.1 Default Configuration

```
Distance matrix shape: (83, 87)
```

The shape of the distance matrix `(83, 87)` indicates that there are 83 nodes (locations) and 87 columns. This shape implies the existence of 83 nodes representing the locations of gas stations or other points of interest, but the matrix includes additional columns (87 total).

```
1: SA with Default Config:

Parameters:
Max Iterations: 100
Initial Temperature: 1000
Cooling Rate: 0.995
================================================================================
Processing Cluster 1:
Best Route: [0 1 2 4 3 5 6 7 8]
Distance: 33.70
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 2:
Best Route: [ 9 10 12 14 13 16 15 17 11]
Distance: 37.30
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 3:
Best Route: [18 22 24 26 19 21 23 25 20]
Distance: 22.80
Runtime: 0.0008 seconds
--------------------------------------------------------------------------------

Processing Cluster 4:
Best Route: [27 30 28 29 31 34 33 35 32]
Distance: 46.50
Runtime: 0.0009 seconds
--------------------------------------------------------------------------------

Processing Cluster 5:
Best Route: [36 42 39 41 43 40 38 37 44]
Distance: 30.90
Runtime: 0.0011 seconds
--------------------------------------------------------------------------------

Processing Cluster 6:
Best Route: [45 49 50 52 51 53 47 46 48]
Distance: 52.20
Runtime: 0.0009 seconds
--------------------------------------------------------------------------------

Processing Cluster 7:
Best Route: [54 59 61 58 57 60 62 56 55]
Distance: 18.90
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 8:
Best Route: [63 69 71 70 64 66 68 65 67]
Distance: 21.80
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 9:
```

```
Best Route: [72 74 79 78 73 75 76 77 80]
Distance: 45.30
Runtime: 0.0014 seconds
--------------------------------------------------------------------------------

Metrics for Default Configuration:
Total Distance (Default): 309.40
Total Runtime (Default): 0.0079 seconds
================================================================================

Metrics for Default Configuration:

Avg Distance: 34.38
Avg Runtime: 0.0009s

Total Distance: 309.40
Total Runtime: 0.0079s
```

The results for the SA algorithm with the default configuration reveal insights into the optimization process for each cluster as well as overall metrics. Each cluster's best route, distance, and runtime are reported, demonstrating the behavior of the algorithm across different configurations of nodes. For example, Cluster 1 produces a route with a distance of 33.70 and a runtime of 0.0007 seconds, while Cluster 6 exhibits a longer route (52.20) but a similar runtime of 0.0009 seconds. This variation in distances suggests that the algorithm's performance may vary depending on the initial configuration of nodes and their interrelations.

The total distance for all clusters combined is 309.40, and the total runtime is 0.0079 seconds, reflecting the aggregate performance of the algorithm for the entire dataset under the default settings. Averaging across the 9 clusters, the average distance is 34.38, and the average runtime is 0.0009 seconds, highlighting the overall efficiency of the algorithm. The relatively short runtime for each cluster suggests that the algorithm is computationally efficient, even with multiple iterations per cluster. However, the distance metrics may indicate room for improvement in terms of optimization, as the distances for certain clusters are considerably longer than for others.

3.2.2 Large Configuration

```
2: SA with Large Config:

Parameters:
Max Iterations: 100
Initial Temperature: 1500
Cooling Rate: 0.98
================================================================================
Processing Cluster 1:
Best Route: [ 0 16 12  4  3  5  7 13 11  6  1 14  9 15  2  8 10]
Distance: 92.60
Runtime: 0.0019 seconds
--------------------------------------------------------------------------------

Processing Cluster 2:
Best Route: [17 28 29 33 32 26 30 19 21 24 20 31 18 22 23 25 27]
Distance: 87.90
Runtime: 0.0019 seconds
--------------------------------------------------------------------------------

Processing Cluster 3:
```

```
Best Route: [34 41 36 37 39 35 40 44 46 43 45 38 42 47 48 49 50]
Distance: 62.20
Runtime: 0.0019 seconds
--------------------------------------------------------------------------------

Processing Cluster 4:
Best Route: [51 55 61 65 59 52 66 58 53 60 67 62 64 54 57 56 63]
Distance: 67.30
Runtime: 0.0018 seconds
--------------------------------------------------------------------------------

Metrics for Default Configuration:
Total Distance (Large): 310.00
Total Runtime (Large): 0.0075 seconds
================================================================================

Metrics for Large Cluster Configuration:

Avg Distance: 62.00
Avg Runtime: 0.0015s

Total Distance: 310.00
Total Runtime: 0.0075s
```

The SA algorithm's performance with the large cluster configuration reveals distinct trends compared to the default setup. The parameters used in this configuration include a higher initial temperature of 1500 and a slower cooling rate of 0.98, which likely contributed to more exploration of the solution space, as evidenced by the increased distances observed in several clusters. For example, Cluster 1 exhibits a significant distance of 92.60, which is notably higher than the largest distance from the default configuration.

Similarly, other clusters such as Cluster 2 and Cluster 3 show distances of 87.90 and 62.20, respectively, both of which are greater than those seen in the default configuration. Despite these higher distances, the runtime for each cluster remains consistent at approximately 0.0019 seconds, showing that the algorithm can handle larger clusters with minimal additional computational cost. Cluster 4 has a distance of 67.30, further reinforcing the trend of higher route lengths with the larger clusters.

The total distance for the large cluster configuration is 310.00, with a total runtime of 0.0075 seconds, indicating that the overall computational cost for the large clusters is similar to the default configuration, despite the increase in cluster size. On average, the distance per cluster is 62.00, and the average runtime is 0.0015 seconds, reflecting the increased complexity of the larger clusters without a significant impact on performance time. This suggests that the choice of a higher initial temperature and slower cooling rate encourages broader exploration of potential solutions, which results in larger route distances compared to the default configuration. The algorithm appears efficient in handling larger clusters, with only a minor increase in runtime.

3.2.3 Small Configuration

```
3: SA with Small Config:

Parameters:
Max Iterations: 100
Initial Temperature: 800
Cooling Rate: 0.99
================================================================================

Processing Cluster 1:
Best Route: [0 1 2 4 3 5 6]
Distance: 19.50
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 2:
Best Route: [ 7  9 13 11  8 10 12]
Distance: 22.60
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 3:
Best Route: [14 19 18 15 17 16 20]
Distance: 22.40
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 4:
Best Route: [21 23 25 26 27 22 24]
Distance: 18.70
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 5:
Best Route: [28 30 32 34 29 31 33]
Distance: 23.80
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 6:
Best Route: [35 41 36 38 40 37 39]
Distance: 18.30
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 7:
Best Route: [42 44 46 48 45 47 43]
Distance: 11.40
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 8:
Best Route: [49 50 52 54 51 53 55]
Distance: 21.80
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 9:
Best Route: [56 58 59 61 60 62 57]
Distance: 12.10
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------

Processing Cluster 10:
Best Route: [63 64 66 68 65 67 69]
Distance: 12.60
Runtime: 0.0007 seconds
--------------------------------------------------------------------------------
```

```
Processing Cluster 11:
Best Route: [70 72 73 75 71 74 76]
Distance: 29.80
Runtime: 0.0011 seconds
--------------------------------------------------------------------------------

Metrics for Default Configuration:
Total Distance (Small): 213.00
Total Runtime (Small): 0.0079 seconds
================================================================================

Metrics for Small Cluster Configuration:

Avg Distance: 17.75
Avg Runtime: 0.0007s

Total Distance: 213.00
Total Runtime: 0.0079s
```

The SA algorithm's performance with the small cluster configuration shows a marked difference in both the route distances and runtime compared to the default and large configurations. In this setup, the algorithm uses a lower initial temperature of 800 and a faster cooling rate of 0.99, designed to encourage faster convergence to solutions. As expected, this results in smaller distances for most clusters, demonstrating a more focused exploration of the solution space. For instance:

- Cluster 1 has a route distance of 19.50, which is considerably smaller compared to the larger clusters in the previous configurations.

- Cluster 2 has a distance of 22.60, while Cluster 3 and Cluster 4 have distances of 22.40 and 18.70, respectively. These are all significantly smaller than the distances in the large configuration.

- Cluster 7 stands out with a route distance of 11.40, reflecting the efficiency of the smaller clusters in minimizing the route length.

- Cluster 11 has the largest distance at 29.80, but it is still relatively smaller than distances seen in larger configurations.

The total distance for the small cluster configuration is 213.00, with a total runtime of 0.0079 seconds, showing that the computational cost remains consistent with the default and large configurations despite the smaller route lengths. The average distance per cluster is 17.75, and the average runtime is 0.0007 seconds, which is notably lower than the other configurations. This configuration's smaller distances can be attributed to the reduced cluster sizes, which simplify the optimization problem, making it easier for the algorithm to find shorter routes within each cluster. The fast convergence rate due to the cooling schedule and initial temperature settings contributes greatly to the reduced distance and runtime. The small cluster configuration is more efficient in terms of both runtime and route distance, but this efficiency might be due to the smaller problem sizes rather than a more optimized solution approach. The results suggest that the smaller clusters enable the algorithm to quickly converge to a relatively good solution with minimal computational cost.

4A) Lovebird Algorithm

```
def read_csv_with_flexible_columns(file_path, expected_columns=87):
    # Use pandas to handle CSV and filter rows with correct column
count
    df = pd.read_csv(file_path, header=None)
    # Filter rows with the correct number of columns
    df = df[df.apply(lambda x: len(x) == expected_columns, axis=1)]
    return df.values.astype(float)  # Return as a numpy array
```

The Lovebird algorithm begins with reading the CSV file that contains the necessary data, such as node distances or coordinates. The function `read_csv_with_flexible_columns` is used to handle files where rows may have varying numbers of columns. It ensures that only rows with the correct number of columns are considered. This is accomplished by reading the CSV into a pandas DataFrame, followed by filtering rows that match the expected number of columns. Afterward, the DataFrame is converted into a NumPy array with all values cast to float, which facilitates numerical operations necessary for distance calculations and optimization processes.

```
def calculate_total_distance(route, distance_matrix):
    total_distance = 0
    for i in range(len(route) - 1):
        node1 = int(route[i])  # Ensure node1 is an integer index
        node2 = int(route[i + 1])  # Ensure node2 is an integer index
        total_distance += distance_matrix[node1, node2]
    # Return to the starting point
    total_distance += distance_matrix[int(route[-1]), int(route[0])]
# Ensure last node is an integer
    return total_distance
```

Once the data is properly loaded, the algorithm proceeds to the next critical function, which is the calculation of the total distance of a route. The function `calculate_total_distance` takes a route, represented as a list of node indices, and a distance matrix, which stores the pairwise distances between nodes. The function loops over the route, calculating the distance between each consecutive pair of nodes by looking them up in the distance matrix. After summing the distances of the nodes along the route, it also adds the distance from the last node back to the first node, thereby closing the loop and ensuring that the route forms a complete cycle.

```
def fitness(route, distance_matrix):
    total_distance = calculate_total_distance(route, distance_matrix)
    # Add a small epsilon to prevent division by zero
```

```
    return 1 / (total_distance + 1e-6)
```

The fitness of a given route is evaluated by the function fitness. Fitness is a measure of how good a solution is; in this case, shorter distances are better. The fitness function uses the total distance of a route, which is calculated by the `calculate_total_distance` function, and returns the reciprocal of this distance, with a small constant added to avoid division by zero. This ensures that shorter routes have a higher fitness value, therefore making them more desirable in the evolutionary process.

```
def order_crossover(parent1, parent2):
    # Randomly select a subsequence from parent1
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child = [-1] * len(parent1)
    child[start:end+1] = parent1[start:end+1]
    # Fill the remaining positions from parent2, preserving the order
    current_pos = 0
    for i in range(len(parent2)):
        if parent2[i] not in child:
            while child[current_pos] != -1:
                current_pos += 1
            child[current_pos] = parent2[i]
    return child
```

The genetic algorithm relies on genetic operators such as crossover and mutation to evolve the population of solutions over time. The `order_crossover` function is a type of crossover that takes two parent routes and produces a child route by combining segments from each parent while maintaining the order of nodes. The crossover is performed by selecting a subsequence of nodes from the first parent and copying it into the child. The remaining positions in the child are filled by nodes from the second parent, in the order they appear, ensuring no duplicates. This process helps generate new routes that inherit desirable traits from both parents.

```
def inversion_mutation(route):
    i, j = sorted(random.sample(range(len(route)), 2))
    route[i:j+1] = route[i:j+1][::-1]  # Reverse the segment
    return route
```

Mutation is another key process in the Lovebird algorithm, and the `inversion_mutation` function as it introduces variability by randomly selecting a segment of a route and reversing the order of nodes within that segment. This helps prevent the algorithm from getting stuck in local optima by introducing new potential solutions and exploring different parts of the search space. Mutation ensures that the population remains diverse and allows the algorithm to explore new routes that might lead to better solutions.

```
def lovebird_algorithm(distance_matrix, population_size=20,
max_iter=100, mutation_rate=0.1):
    # Initialization: Generate initial population of random routes
    population = []
    for _ in range(population_size):
        # Generate a route with Node 0 as the starting point
        remaining_nodes = list(range(1, len(distance_matrix)))  #
Node 0
        random.shuffle(remaining_nodes)  # Shuffle remaining nodes
        route = [0] + remaining_nodes  # Start route with Node 0
        population.append(route)

    # Initializing the best solution
    best_route = population[0]
    best_fitness = fitness(best_route, distance_matrix)

    total_distance_across_iterations = 0  # Accumulate total distance
    start_time = time.time()  # Track runtime

    for iteration in range(max_iter):
        # Calculate fitness for each individual in the population
        fitness_values = [fitness(route, distance_matrix) for route
in population]

        # Select the top 50% of the population based on fitness
(elitism)
        elitism_rate = 0.6  # Top 60% of the population
        selected_parents = [population[i] for i in
np.argsort(fitness_values)[-int(population_size * elitism_rate):]]

        # Generate new population using mating (crossover) and
mutation
        new_population = []

        while len(new_population) < population_size:
            # Select two parents at random
            parent1, parent2 = random.sample(selected_parents, 2)

            # Crossover: Create a child by combining parts of the
parents
```

```
            child = order_crossover(parent1, parent2)

            # Mutation: Apply inversion mutation
            if random.random() < mutation_rate:
                child = inversion_mutation(child)

            new_population.append(child)

        # Update the population with the new generation
        population = new_population

        # Update the best solution
        current_best = max(population, key=lambda route:
fitness(route, distance_matrix))
        current_best_fitness = fitness(current_best, distance_matrix)

        if current_best_fitness > best_fitness:
            best_route = current_best
            best_fitness = current_best_fitness

        # Accumulate the best distance for this iteration
        iteration_distance = calculate_total_distance(best_route,
distance_matrix)
        total_distance_across_iterations += iteration_distance

        # Optionally print progress
        if iteration % 10 == 0:
            print(f"Iteration {iteration + 1}: Best Distance =
{iteration_distance:.2f}")

    end_time = time.time()  # End runtime
    runtime = end_time - start_time

    # Calculate the average distance and runtime
    avg_distance = total_distance_across_iterations / max_iter
    avg_runtime = runtime / max_iter

    return best_route, avg_distance,
total_distance_across_iterations, avg_runtime, runtime  # Return all
metrics
```

The main evolutionary process of the Lovebird algorithm is handled by the lovebird_algorithm function. This function initializes the population with random routes, where each route starts at node 0. The initial population is created by randomly shuffling the remaining nodes, with node 0 fixed at the start of each route. The fitness of each route is evaluated, and the population is evolved over several iterations. The population size, mutation rate, and number of iterations can be configured.

In each iteration, the algorithm selects the top-performing individuals (based on fitness) and uses them to generate the next generation. This is done by selecting pairs of parents, performing crossover to produce offspring, and applying mutation to introduce diversity. The algorithm tracks the best solution found so far, updating it if a better solution is discovered.

```
def run_lovebird_configuration(distance_matrix, lat_long_data,
config_type='default'):
    # Configuration settings with mutation rates for configuration
type
    configurations = {
        'default': {'population_size': 100, 'max_iterations': 100,
'mutation_rate': 0.1},
        'large': {'population_size': 300, 'max_iterations': 100,
'mutation_rate': 0.05},
        'small': {'population_size': 50, 'max_iterations': 100,
'mutation_rate': 0.2}
    }

    def get_config_params(config_type):
        if config_type not in configurations:
            raise ValueError("Unknown configuration type!")
        return configurations[config_type]

    config = get_config_params(config_type)
    population_size = config['population_size']
    max_iterations = config['max_iterations']
    mutation_rate = config['mutation_rate']

    # Configuration numbering
    config_labels = {
        'default': '1: Lovebird with Default Config:\n',
        'large': '2: Lovebird with Large Config:\n',
        'small': '3: Lovebird with Small Config:\n'
    }
```

```
    # Simulate for the given nodes and distance matrix
    print(f"\n{config_labels.get(config_type)}")
    print(f"Population Size: {population_size}")
    print(f"Max Iterations: {max_iterations}")
    print(f"Mutation Rate: {mutation_rate}")
    print("-" * 80)


    best_route, avg_distance, total_distance, avg_runtime, runtime =
lovebird_algorithm(distance_matrix, population_size, max_iterations,
mutation_rate)

    # Output for the final result
    print(f"\nBest Route: {best_route}")
    print(f"Runtime: {runtime:.4f} seconds")
    print("=" * 80)

    return best_route, avg_distance, total_distance, avg_runtime,
runtime
```

The `run_lovebird_configuration` function is designed to run the Lovebird algorithm with a configurable set of parameters, which include the population size, the number of iterations, and the mutation rate. These parameters are determined by the `config_type` argument, which allows the user to select between different preset configurations: default, large, or small. Each configuration is associated with a distinct set of values. The default configuration uses a population size of 100, 100 iterations, and a mutation rate of 0.1, while the large configuration increases the population size to 300, reduces the mutation rate to 0.05, and keeps the iteration count at 100. The small configuration, on the other hand, uses a smaller population size of 50, a higher mutation rate of 0.2, and again, 100 iterations.

The function starts by defining these configuration settings in a dictionary. To retrieve the appropriate settings, it defines a helper function `get_config_params(config_type)` that checks whether the provided configuration type is valid. If the type is found in the dictionary, it returns the corresponding parameters; otherwise, it raises an error. Once the configuration is retrieved, the function assigns the values of population size, maximum iterations, and mutation rate to variables. Additionally, the function uses another dictionary, `config_labels`, in order to assign a human-readable label to each configuration. This label is printed in the output to inform the user which configuration is being used. The function then proceeds by printing these configuration details, such as the population size, the mutation rate, and the number of iterations.

Following the configuration setup, the function calls the lovebird_algorithm function, passing in the distance matrix, population size, mutation rate, and maximum iterations. The `lovebird_algorithm` is expected to return several results, including the best route found, the

average distance of all routes, the total distance for the best route, the average runtime, and the total runtime of the algorithm. The results are printed to the console for immediate feedback, including the best route and its runtime, also being returned as a tuple for further use or analysis.

```
# Load lat_long_data from your CSV file here
lat_long_data = pd.read_csv('lat_long.csv').values[:, [2, 3]]  #
Assuming lat/long are in columns 2 and 3

# Distance matrix (ensure this is the correct matrix you need)
distance_matrix = np.random.random((len(lat_long_data),
len(lat_long_data)))

# Print the distance matrix shape
print(f"Distance Matrix Shape: {distance_matrix.shape}")

# Run the default configuration
best_route_default, avg_distance_default, total_distance_default,
avg_runtime_default, runtime_default =
run_lovebird_configuration(distance_matrix, lat_long_data, 'default')

# Run the large cluster configuration
best_route_large, avg_distance_large, total_distance_large,
avg_runtime_large, runtime_large =
run_lovebird_configuration(distance_matrix, lat_long_data, 'large')

# Run the small cluster configuration
best_route_small, avg_distance_small, total_distance_small,
avg_runtime_small, runtime_small =
run_lovebird_configuration(distance_matrix, lat_long_data, 'small')
```

This section begins by loading the latitude and longitude data from a CSV file named `lat_long.csv` using the Pandas library. The data is loaded as a NumPy array, and specifically, columns 2 and 3 are selected, which are assumed to contain the latitude and longitude values, respectively. This data is then stored in the variable lat_long_data. Next, a distance matrix is generated using NumPy's `random.random()` function, which creates a matrix of random values. The dimensions of this matrix are set to match the number of locations in `lat_long_data`, with both dimensions being equal to the length of the `lat_long_data` array. This ensures that the matrix represents pairwise distances between each pair of locations.

After the distance matrix is created, the shape of the matrix is printed to provide the user with information about its dimensions, which is important for verifying that the matrix has been generated correctly. The next step involves running the Lovebird algorithm with three different configurations: default, large, and small; by calling the `run_lovebird_configuration`

function three times, each time with different configuration types. In each case, the function is passed the distance matrix and the latitude-longitude data, along with a specific configuration type ('default', 'large', or 'small'). For the default configuration, the function is expected to return the best route found, the average distance of all routes, the total distance of the best route, the average runtime, and the total runtime of the algorithm. The same process is repeated for the large and small cluster configurations, with the returned results being stored in separate variables for each configuration.

```
# Display all metrics for Lovebird Algorithm
print('\n \n \n')

print("\nMetrics for Default Configuration: \n")
print(f"Avg Distance: {avg_distance_default:.2f}")
print(f"Avg Runtime: {avg_runtime_default:.4f}s" + '\n')
print(f"Total Distance: {total_distance_default:.2f}")
print(f"Total Runtime: {runtime_default:.4f}s")

print('\n')

print("\nMetrics for Large Cluster Configuration: \n")
print(f"Avg Distance: {avg_distance_large:.2f}")
print(f"Avg Runtime: {avg_runtime_large:.4f}s" + '\n')
print(f"Total Distance: {total_distance_large:.2f}")
print(f"Total Runtime: {runtime_large:.4f}s")

print('\n')

print("\nMetrics for Small Cluster Configuration: \n")
print(f"Avg Distance: {avg_distance_small:.2f}")
print(f"Avg Runtime: {avg_runtime_small:.4f}s" + '\n')
print(f"Total Distance: {total_distance_small:.2f}")
print(f"Total Runtime: {runtime_small:.4f}s")
```

For the default configuration, the code prints the average distance of the routes (avg_distance_default), the average runtime of the algorithm (avg_runtime_default), the total distance of the best route found (total_distance_default), and the total runtime (runtime_default). Each of these metrics is formatted to two decimal places for the distances and four decimal places for the runtimes, providing a clear and readable output. Next, the same metrics are printed for the large cluster configuration. The code uses the variables `avg_distance_large`, `avg_runtime_large`, `total_distance_large`, and `runtime_large` to store the results for the large configuration. The metrics are displayed in the same format, allowing for an easy comparison of the results with the default configuration. Similarly, the metrics for the small cluster configuration are displayed, using the corresponding variables

`avg_distance_small`, `avg_runtime_small`, `total_distance_small`, and `runtime_small`. The output for each configuration is separated by newlines to enhance readability, and the results provide insights into how different configurations influence the performance of the Lovebird algorithm in terms of both the quality of the solutions and computational efficiency.

4B) Lovebird Algorithm (Output)

3.2.1 Default Configuration

```
Distance Matrix Shape: (86, 86)
```

The output "Distance Matrix Shape: (86, 86)" indicates that the distance matrix used in the Lovebird algorithm has a shape of 86 rows and 86 columns. This means that there are 86 nodes (locations) represented in the matrix, and the matrix holds the pairwise distances between each pair of these nodes. Each entry in the matrix corresponds to the distance between two specific nodes, with the value at position (i, j) representing the distance between node i and node j. A square matrix of this size (86 x 86) suggests that the problem involves 86 locations, and the matrix contains all possible distance relationships between them. This matrix will be used as input to the Lovebird algorithm to find the optimal route(s) between these locations based on the various configurations tested.

```
1: Lovebird with Default Config:

Population Size: 100
Max Iterations: 100
Mutation Rate: 0.1
--------------------------------------------------------------------------------
Iteration 1: Best Distance = 35.41
Iteration 11: Best Distance = 31.62
Iteration 21: Best Distance = 29.02
Iteration 31: Best Distance = 29.02
Iteration 41: Best Distance = 28.95
Iteration 51: Best Distance = 28.18
Iteration 61: Best Distance = 28.18
Iteration 71: Best Distance = 27.71
Iteration 81: Best Distance = 27.71
Iteration 91: Best Distance = 27.71

Best Route: [0, 2, 66, 30, 84, 8, 42, 37, 5, 78, 39, 20, 14, 24, 32, 71, 56, 74, 19, 13, 6, 35, 9, 55, 40, 41, 61,
69, 7, 60, 45, 15, 38, 43, 75, 77, 23, 53, 11, 28, 83, 72, 3, 65, 85, 50, 64, 47, 54, 51, 79, 59, 31, 25, 18, 36,
80, 52, 57, 76, 4, 68, 49, 21, 46, 67, 22, 58, 17, 81, 63, 16, 10, 34, 70, 44, 26, 48, 12, 27, 29, 82, 33, 1, 62,
73]
Runtime: 1.6492 seconds
================================================================================

Metrics for Default Configuration:

Avg Distance: 28.85
Avg Runtime: 0.0165s

Total Distance: 2885.31
Total Runtime: 1.6492s
```

The first output details the performance of the Lovebird algorithm under the default configuration, with a population size of 100, a maximum of 100 iterations, and a mutation rate of 0.1. Over the course of the iterations, the algorithm progressively improved the best distance found. Initially, the best distance was 35.41 units, and by the 91st iteration, it reached 27.71, showing that the algorithm effectively converged to a near-optimal solution. This demonstrates the algorithm's ability to refine the solution over time with each iteration, gradually reducing the total distance.

The "Best Route" is a sequence of 86 nodes, represented by indices, which is the path that minimizes the distance between all the locations in the dataset. The best route reflects the optimal path as determined by the algorithm for the given configuration. In terms of performance metrics, the algorithm achieved an average distance of 28.85 units, and the total distance of the route was 2885.31 units, summing the distances of all the node-to-node connections along the route. The average runtime per iteration was 0.0165 seconds, indicating that the algorithm performed efficiently, with a total runtime of 1.6492 seconds for all 100 iterations.

```
2: Lovebird with Large Config:

Population Size: 300
Max Iterations: 100
Mutation Rate: 0.05
--------------------------------------------------------------------------------
Iteration 1: Best Distance = 33.62
Iteration 11: Best Distance = 30.61
Iteration 21: Best Distance = 29.53
Iteration 31: Best Distance = 28.55
Iteration 41: Best Distance = 28.55
Iteration 51: Best Distance = 27.27
Iteration 61: Best Distance = 27.27
Iteration 71: Best Distance = 27.27
Iteration 81: Best Distance = 27.27
Iteration 91: Best Distance = 27.27

Best Route: [0, 74, 29, 8, 36, 7, 59, 14, 43, 21, 49, 31, 46, 4, 84, 63, 82, 2, 51, 1, 16, 42, 9, 41, 40, 26, 44,
69, 50, 67, 22, 52, 37, 80, 32, 23, 77, 13, 62, 38, 3, 68, 35, 15, 5, 53, 11, 76, 28, 85, 45, 19, 81, 54, 6, 71,
79, 70, 83, 27, 24, 18, 64, 61, 17, 66, 30, 75, 72, 12, 58, 55, 57, 60, 56, 20, 34, 65, 33, 73, 78, 10, 48, 25,
39, 47]
Runtime: 4.8591 seconds
================================================================================

Metrics for Large Cluster Configuration:

Avg Distance: 28.39
Avg Runtime: 0.0486s

Total Distance: 2839.35
Total Runtime: 4.8591s
```

The output for the Lovebird algorithm with the large configuration shows the results for a setup with a population size of 300, 100 iterations, and a mutation rate of 0.05. During the 100 iterations, the algorithm improved its best distance progressively. The initial best distance was 33.62 units, and by the 91st iteration, it was reduced to 27.27. This indicates that the algorithm successfully refined the route with each iteration, gradually moving closer to an optimal solution. The "Best Route" is a sequence of 86 nodes that represents the optimal path for this configuration, with the algorithm selecting nodes that minimize the overall distance between all locations. For the performance metrics, the average distance across all iterations was 28.39 units, and the total distance of the best route was 2839.35 units. These results are very close to the output from the default configuration, which shows that the Lovebird algorithm performed similarly in terms of solution quality but with a larger population. The average runtime per iteration was 0.0486 seconds, and the total runtime for all 100 iterations was 4.8591 seconds. This indicates a longer computation time compared to the default configuration, which is expected due to the increased population size, leading to more evaluations and a greater computational load.

```
3: Lovebird with Small Config:

Population Size: 50
Max Iterations: 100
Mutation Rate: 0.2
--------------------------------------------------------------------------------
Iteration 1: Best Distance = 36.89
Iteration 11: Best Distance = 30.92
Iteration 21: Best Distance = 30.92
Iteration 31: Best Distance = 30.92
Iteration 41: Best Distance = 28.57
Iteration 51: Best Distance = 28.57
Iteration 61: Best Distance = 28.57
Iteration 71: Best Distance = 28.57
Iteration 81: Best Distance = 28.57
Iteration 91: Best Distance = 28.57

Best Route: [21, 11, 85, 77, 28, 81, 64, 23, 84, 60, 0, 79, 68, 48, 17, 16, 53, 29, 26, 45, 55, 22, 19, 47, 39, 1,
7, 65, 66, 50, 20, 32, 73, 43, 61, 10, 75, 25, 56, 13, 5, 33, 42, 12, 58, 40, 54, 76, 6, 67, 51, 70, 62, 27, 37,
80, 52, 2, 78, 24, 41, 4, 46, 30, 38, 18, 59, 82, 63, 72, 69, 57, 83, 71, 9, 34, 49, 3, 74, 31, 44, 14, 15, 36,
35, 8]
Runtime: 0.8438 seconds
================================================================================

Metrics for Small Cluster Configuration:

Avg Distance: 29.61
Avg Runtime: 0.0084s

Total Distance: 2960.97
Total Runtime: 0.8438s
```

The output for the Lovebird algorithm with the small configuration reveals the results of a setup with a population size of 50, 100 iterations, and a mutation rate of 0.2. This configuration aimed to test the algorithm's performance with a smaller population and higher mutation rate, which is expected to lead to a more aggressive search for solutions.

In the 100 iterations, the algorithm initially started with a best distance of 36.89 units in the first iteration. Over time, it progressively improved, reaching a best distance of 28.57 by the 41st iteration, and maintaining this best distance throughout the rest of the iterations. The "Best Route" is an optimal path that minimizes the distance between the nodes. It consists of 86 nodes, which is the same as in the previous configurations, but the specific nodes visited in the best route vary due to the different starting conditions and the smaller population. Regarding the performance metrics, the average distance for this configuration was 29.61 units, which is slightly higher than the results from both the default and large configurations. This suggests that while the algorithm explored the solution space aggressively due to the higher mutation rate, it did not quite reach the same level of optimization as the larger population settings.

The total distance for the best route was 2960.97 units, which again reflects the higher average distance. The average runtime per iteration was 0.0084 seconds, which is much shorter than in the default and large configurations, indicating that fewer evaluations were performed in each iteration due to the smaller population. The total runtime for all 100 iterations was 0.8438 seconds, making this configuration the quickest in terms of overall computation time.

This configuration, with its smaller population and higher mutation rate, was the fastest in terms of runtime but resulted in a slightly higher average distance compared to the larger population configurations, indicating that while the search process was faster, it did not always yield the best possible solution. The trade-off between computational time and solution quality becomes really evident in this setup.

VI. Results of Parameter Experiments & Comparison with Each Methods

In this section, a comparison will be made between each experiment conducted on the GA, ACO, SA, and Lovebird algorithms.

1. Genetic Algorithm

a. GA

Experiments in GA were conducted using 2 different parameters:

- Parameter 1: population size=50, mutation_rate=0.05, generations=100
- Parameter 2: population_size=75, mutation_rate=0.03, generations=100

The results of these two parameters can be seen in the table below:

Cluster	Route (Param 1)	Shortest Distance (Param 1)	Runtime (Param 1)	Route (Param 2)	Shortest Distance (Param 2)	Runtime (Param 2)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	0.32	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	0.47
2	[0, 12, 8, 3, 4, 6, 0]	43.0	0.27	[0, 12, 8, 3, 4, 6, 0]	43.0	0.40
3	[0, 81, 13, 9, 11, 80, 0]	34.8	0.27	[0, 81, 13, 9, 11, 80, 0]	34.8	0.40
4	[0, 37, 46, 41, 44, 0]	32.2	0.24	[0, 37, 46, 41, 44, 0]	32.2	0.35
5	[0, 48, 35, 36, 74, 49, 50, 75, 78, 76, 29, 77, 20, 45, 47, 39, 43, 0]	47.9	0.82	[0, 43, 48, 35, 36, 74, 49, 47, 20, 77, 78, 29, 76, 75, 45, 50, 39, 0]	43.9	0.83
6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	0.68	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	0.56
7	[0, 68, 69, 63,	42.6	1.08	[0, 27, 65,	38.5	0.91

	59, 56, 57, 52, 58, 84, 65, 53, 61, 54, 1, 27, 62, 55, 70, 0]			62, 84, 1, 55, 58, 53, 61, 57, 54, 52, 59, 56, 69, 63, 68, 70, 0]		
8	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	0.76	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	0.69
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	0.35	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	0.52
Total		322.9 km	4.80 s	**Total**	**314.8 km**	5.12 s

The difference in results between Parameter 1 and Parameter 2 can be attributed to the adjustments in population size and mutation rate. With a larger population size (75 vs. 50) in Parameter 2, the algorithm explores a broader solution space, increasing the likelihood of finding a more optimized route (314.8 km vs. 322.9 km). However, this also requires slightly more computation time (5.12 s vs. 4.80 s). The lower mutation rate (0.03 vs. 0.05) in Parameter 2 promotes more stable convergence by reducing random changes, leading to a more refined solution at the cost of increased runtime. These trade-offs highlight the balance between exploration, exploitation, and computational efficiency in genetic algorithms.

b. Hybrid GA with SA

Experiments in Hybrid GA with SA were conducted using 2 different parameters:

- Parameter 1: population size=50, mutation_rate=0.05, generations=100, initial_temperature=100, cooling_rate=0.95
- Parameter 2: population_size=75, mutation_rate=0.03, generations=100, initial_temperature=150, cooling_rate=0.93

The results of these two parameters can be seen in the table below:

Cluster	**Route (Param 1)**	**Shortest Distance (Param 1)**	**Runtime (Param 1)**	**Route (Param 2)**	**Shortest Distance (Param 2)**	**Runtime (Param 2)**
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	3.52	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	6.29
2	[0, 12, 8, 3, 4,	43.0	0.26	[0, 12, 8, 3,	43.0	0.39

	6, 0]			4, 6, 0]		
3	[0, 81, 13, 9, 11, 80, 0]	34.8	0.27	[0, 81, 13, 9, 11, 80, 0]	34.8	0.39
4	[0, 37, 46, 41, 44, 0]	32.2	0.23	[0, 37, 46, 41, 44, 0]	32.2	0.35
5	[0, 43, 35, 48, 36, 74, 49, 47, 45, 75, 76, 78, 29, 77, 20, 50, 39, 0]	47.9	0.81	[0, 43, 48, 35, 36, 74, 49, 47, 20, 77, 78, 29, 76, 75, 45, 50, 39, 0]	43.9	0.83
6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	0.67	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	0.55
7	[0, 70, 55, 27, 65, 62, 84, 1, 61, 54, 53, 57, 52, 58, 59, 56, 69, 63, 68, 0]	42.59	1.08	[0, 27, 65, 62, 84, 1, 55, 58, 53, 61, 57, 54, 52, 59, 56, 69, 63, 68, 70, 0]	38.50	0.91
8	[0, 66, 15, 18, 82, 25, 26, 42, 17, 64, 38, 85, 33, 67, 0]	31.6	0.76	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	0.69
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.90	0.35	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.90	0.51
Total		322.9 km	4.79 s	**Total**	314.8 km	5.12 s

The results indicate that Parameter 2 outperforms Parameter 1 in terms of total shortest distance (314.80 km vs. 322.9 km) but takes slightly longer runtime (5.12 s vs. 4.79 s). This difference arises because Parameter 2 uses a larger population size, which increases the diversity of solutions explored by the Genetic Algorithm (GA). The higher initial temperature and slower cooling rate in Simulated Annealing (SA) allow for a more thorough exploration of the solution space, reducing the likelihood of getting trapped in local minima. However, this added exploration requires more computation, leading to a slightly longer runtime.

c. Hybrid GA with TS

Experiments in Hybrid GA with TS were conducted using 2 different parameters:

- Parameter 1: population size=50, mutation_rate=0.05, generations=100, tabu_tenure=10, local_iterations=20

- Parameter 2: population_size=75, mutation_rate=0.03, generations=100, tabu_tenure=15, local_iterations=30

The results of these two parameters can be seen in the table below

Cluster	Route (Param 1)	Shortest Distance (Param 1)	Runtime (Param 1)	Route (Param 2)	Shortest Distance (Param 2)	Runtime (Param 2)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	8.48	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	18.14
2	[0, 12, 8, 3, 4, 6, 0]	43.0	2.85	[0, 12, 8, 3, 4, 6, 0]	43.0	7.53
3	[0, 81, 13, 9, 11, 80, 0]	34.8	3.85	[0, 81, 13, 9, 11, 80, 0]	34.8	7.19
4	[0, 37, 46, 41, 44, 0]	32.2	1.22	[0, 37, 46, 41, 44, 0]	32.2	2.56
5	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	89.50	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	274.73
6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	18.88	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	39.08
7	[0, 69, 63, 68, 58, 55, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 70, 0]	32.4	127.19	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	290.09
8	[0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 33, 38, 67, 0]	31.10	49.12	[0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 33, 38, 67, 0]	31.10	112.64
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.90	12.02	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.90	28.91
Total		303.7 km	313.14 s	**Total**	303.2 km	780.91 s

The difference in results between Parameter 1 and Parameter 2 in the Hybrid GA with TS can be attributed to the trade-off between exploration and exploitation. Parameter 2 uses a larger population size (75) and more local iterations (30), which increases the exploration of the solution space, allowing for potentially better solutions (303.20 km). However, this comes at the cost of significantly longer computation time (780.91 s) due to the increased number of individuals and iterations. On the other hand, Parameter 1 has a smaller population size (50) and fewer local iterations (20), leading to faster computation (313.14 s) but slightly less optimal results (303.70 km), as it might not explore the solution space as thoroughly. Thus, Parameter 2 sacrifices runtime for potentially better solutions.

d. Hybrid GA with PSO

Experiments in Hybrid GA with PSO were conducted using 2 different parameters:

- Parameter 1: population size=50, mutation_rate=0.05, generations=100, pso_particles=30, inertia_weight=0.7, cognitive_weight=1.5, social_weight=1.5
- Parameter 2: population_size=75, mutation_rate=0.03, generations=100, pso_particles=40, inertia_weight=0.6, cognitive_weight=1.7, social_weight=1.7

The results of these two parameters can be seen in the table below

Cluster	Route (Param 1)	Shortest Distance (Param 1)	Runtime (Param 1)	Route (Param 2)	Shortest Distance (Param 2)	Runtime (Param 2)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	2.61	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	5.50
2	[0, 12, 8, 3, 4, 6, 0]	43.0	3.38	[0, 12, 8, 3, 4, 6, 0]	43.0	5.90
3	[0, 81, 13, 9, 11, 80, 0]	34.8	2.10	[0, 81, 13, 9, 11, 80, 0]	34.8	4.42
4	[0, 37, 46, 41, 44, 0]	32.2	1.82	[0, 37, 46, 41, 44, 0]	32.2	5.22
5	[0, 39, 50, 49, 77, 20, 45, 75, 78, 76, 29, 47, 74, 36, 35, 48, 43, 0]	43.5	4.83	[0, 43, 48, 35, 36, 74, 49, 77, 29, 78, 75, 76, 20, 45, 47, 50, 39, 0]	44.90	11.34
6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	4.36	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	7.71

7	[0, 62, 70, 57, 52, 56, 59, 53, 61, 1, 55, 69, 63, 68, 58, 54, 27, 84, 65, 0]	43.3	5.61	[0, 70, 68, 56, 59, 52, 54, 53, 61, 57, 84, 1, 65, 27, 62, 58, 55, 69, 63, 0]	37.4	12.34
8	[0, 38, 85, 33, 42, 17, 82, 25, 26, 64, 66, 18, 15, 67, 0]	33.4	4.40	[0, 67, 33, 38, 85, 64, 82, 25, 26, 42, 17, 66, 18, 15, 0]	32.2	9.69
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.90	3.73	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.90	5.93
Total		321.0 km	32.88 s	**Total**	315.3 km	68.09 s

The results show that Parameter 2, with a larger population size (75), more PSO particles (40), and higher cognitive and social weights, achieved a slightly better result in terms of the shortest distance (315.30 km) compared to Parameter 1 (321.0 km).

However, Parameter 2 took significantly longer to run (68.09 seconds) due to the increased number of particles and higher weights, which require more computational effort for the PSO process to explore and exploit the search space. The larger population and particle count allow for a more thorough search but also increase runtime, while Parameter 1, with a smaller population and fewer particles, is faster but may not explore as thoroughly, resulting in a slightly worse distance.

- **Conclusion for all of the results in GA experiments**

In conclusion, the experiments with various Genetic Algorithm (GA) variants—GA, GA with Simulated Annealing (SA), GA with Tabu Search (TS), and GA with Particle Swarm Optimization (PSO)—showcase the trade-offs between solution quality and computational effort.

1. **GA (Genetic Algorithm)** provided reasonable solutions but was not the most optimal compared to others in terms of distance.
2. **GA with Simulated Annealing (SA)** improved the solution by balancing exploration and exploitation but at a higher computational cost compared to pure GA.
3. **GA with Tabu Search (TS)** offered better solutions by incorporating memory-based search strategies, effectively avoiding cycles and local minima, but again, the runtime increased.

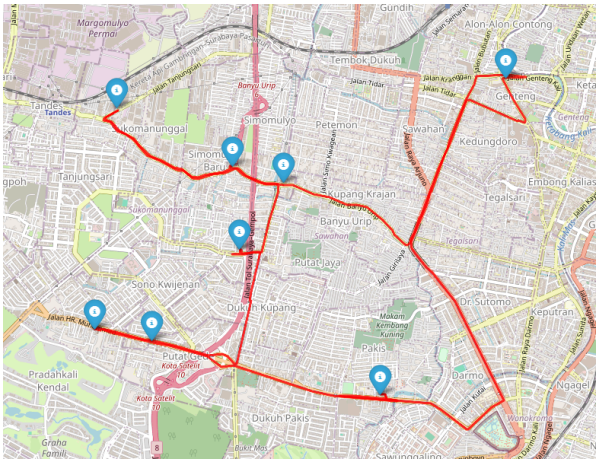
4. **GA with Particle Swarm Optimization (PSO)** provided a solid balance between exploration (due to particles exploring the space) and exploitation, yielding competitive solutions, though the execution time was higher, particularly with larger parameter settings.

Each approach has its strengths: GA provides faster results but may get stuck in suboptimal solutions, while combining it with SA, TS, or PSO improves the solution quality at the cost of increased computation time. The choice of algorithm should depend on whether faster execution or better accuracy is prioritized.

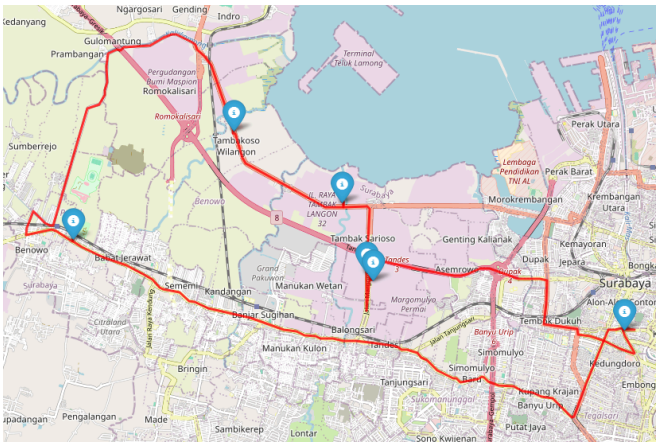
- **Visualization with the best combination of GA and parameters**

The best algorithm that produces the shortest route is the Hybrid GA with TS at parameter 2. The visualization in this experiment utilizes the Folium library, allowing the results to align with the actual road layout. The nodes displayed have been also adjusted to match the outcomes of running the Genetic Algorithm with Tabu Search (GA-TS).

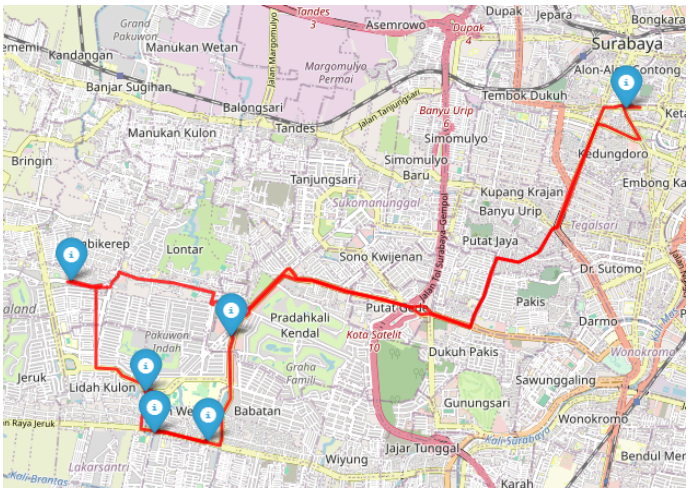
Cluster 1



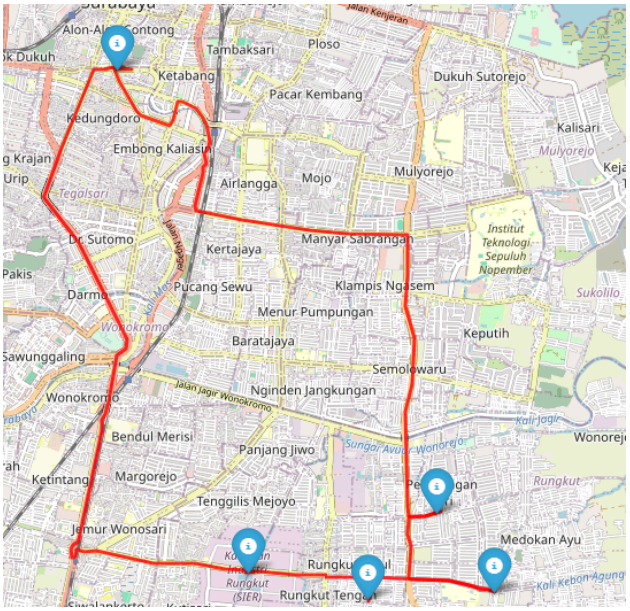
Cluster 2



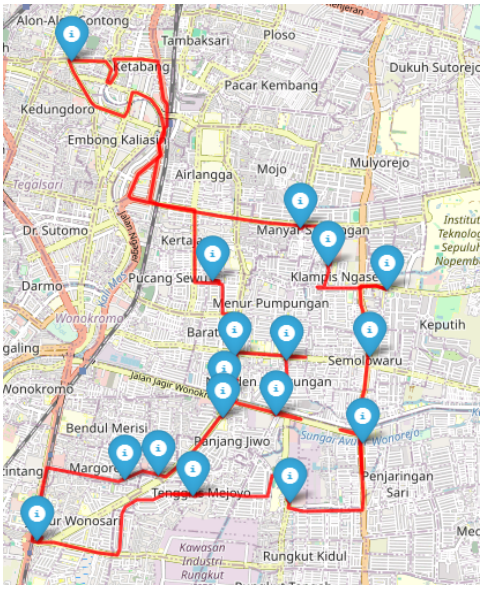
Cluster 3



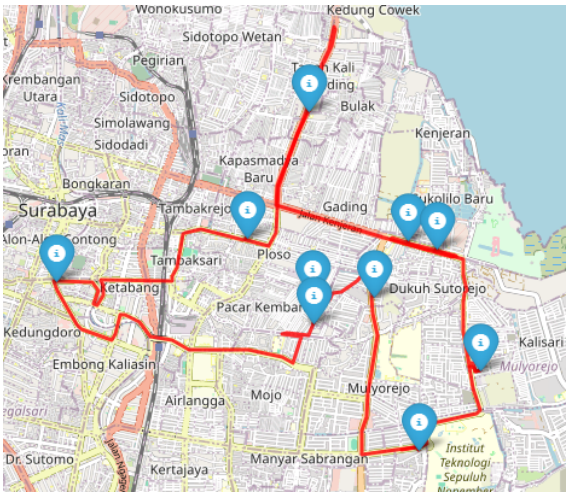
Cluster 4



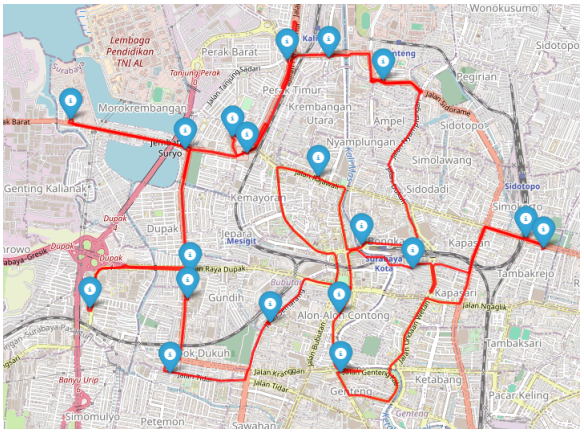
Cluster 5



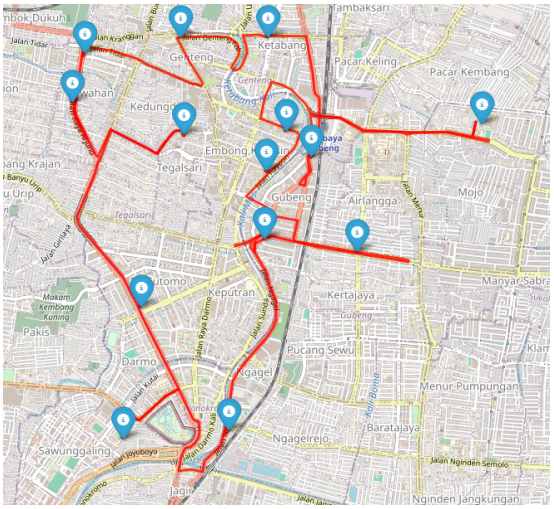
Cluster 6



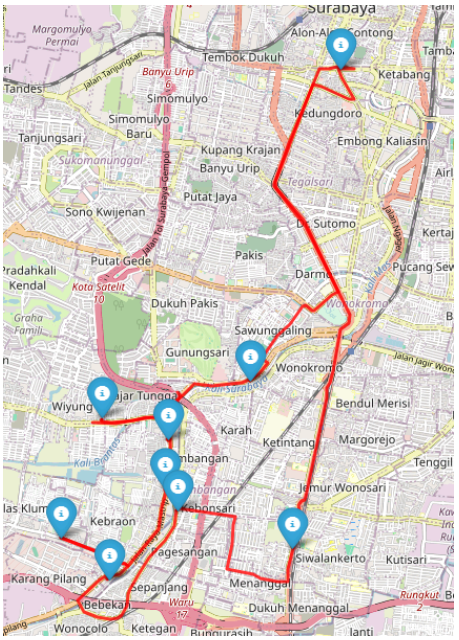
Cluster 7



Cluster 8



Cluster 9



2. ACO

After analyzing the output of the **ACO**, **Hybrid ACO+SA**, **Hybrid ACO+Tabu Search**, and **Hybrid ACO+PSO** algorithms, the **best model** is identified as the **Hybrid ACO+PSO with default configuration** based on the following reasons:

1. **Shortest Total Distance**: The Hybrid ACO+PSO model with default configuration achieved a total distance of **303.90**, which is the same as other models in their default configurations but with the added advantage of faster execution time.
2. **Fastest Execution Time**: The default Hybrid ACO+PSO configuration recorded the fastest execution time of **14.77 seconds**, outperforming other models with comparable distances.
3. **Consistent Performance**: The Hybrid ACO+PSO model demonstrated consistent performance across various configurations, with minimal variations in total distance.

The results for ACO with various configurations are as follows:

1. ACO Results

The ACO model, based on the behavior of ants in finding optimal paths, is implemented in this study to solve the routing problem. The default configuration involves 20 ants, 100 iterations, and key parameters such as alpha (1.0) and beta (2.0), which balance pheromone influence and heuristic information. The model achieved a shortest distance of **303.90**, but its execution time of **26.77 seconds** was slower compared to hybrid approaches. Variations such as large clusters and small clusters showed minor differences in total distance but significantly increased execution times due to higher ant counts or more complex configurations.

Configuration	Iterations	Number of Ants	Alpha	Beta	Evaporation	Total Distance	Total Time (s)
Default	100	20	1.0	2.0	0.1	**303.90**	26.77
Large Clusters	100	80	2.5	6.0	0.08	308.90	55.89
Small Clusters	100	40	1.5	4.5	0.15	306.20	28.37
Optimized	100	60	2.0	5.0	0.12	308.50	42.26

Ant Colony Optimization (ACO) produced varied results across different configurations, emphasizing the balance between exploration, exploitation, and computational efficiency. The default configuration achieved a total shortest distance of 303.9 with a runtime of 26.77 seconds. This setting offered a good starting point, efficiently finding competitive routes within a reasonable computational time.

When tuned for large clusters, the model explored a wider solution space, resulting in a slightly higher total shortest distance of 308.9 but with a significantly increased runtime of 55.89 seconds. This configuration demonstrated the trade-off between accuracy and computational effort, as the model required additional resources to handle larger clusters.

The small cluster configuration focused on computational efficiency, achieving a total shortest distance of 306.2 with a reduced runtime of 28.37 seconds. This configuration favored faster execution while maintaining reasonable solution quality, making it suitable for scenarios with tight runtime constraints.

The optimized parameter configuration aimed to improve accuracy, achieving a total shortest distance of 308.5 with a runtime of 42.26 seconds. This configuration balanced exploration and exploitation effectively, highlighting the importance of parameter tuning to meet specific problem requirements.

Cluster	Route (Default)	Distance (Default)	Time (Default)	Route (Large)	Distance (Large)	Time (Large)	Route (Small)	Distance (Small)	Time (Small)	Route (Optimized)	Distance (Optimized)	Time (Optimized)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	4.48	[0, 7, 5, 83, 2, 14, 10, 19, 0]	25.9	4.85	[0, 19, 14, 10, 2, 83, 5, 7, 0]	25.9	2.06	[0, 19, 14, 10, 2, 83, 5, 7, 0]	25.9	3.12
2	[0, 12, 8, 3, 4, 6, 0]	43.0	1.53	[0, 12, 8, 3, 4, 6, 0]	43.0	3.08	[0, 12, 8, 3, 4, 6, 0]	43.0	1.49	[0, 12, 8, 3, 4, 6, 0]	43.0	2.51
3	[0, 81, 13, 9, 11, 80, 0]	34.8	2.20	[0, 9, 11, 13, 81, 80, 0]	35.0	3.23	[0, 9, 11, 13, 81, 80, 0]	35.0	1.62	[0, 9, 11, 13, 81, 80, 0]	35.0	2.32
4	[0, 37, 46, 41, 44, 0]	32.2	1.75	[0, 37, 46, 41, 44, 0]	32.2	2.51	[0, 37, 46, 41, 44, 0]	32.2	1.27	[0, 37, 46, 41, 44, 0]	32.2	1.80

5	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	5.72	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	10.36	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	5.76	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	8.29
6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	2.50	[0, 32, 40, 72, 51, 71, 34, 31, 73, 60, 0]	31.6	6.13	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	2.82	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	4.49
7	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	3.45	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	12.45	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	6.65	[0, 68, 70, 55, 58, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0]	33.3	9.47
8	[0, 15, 18, 82, 25, 26, 42, 17, 33, 38, 85, 64, 66, 67, 0]	31.8	2.85	[0, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 33, 67, 0]	33.0	8.30	[0, 15, 18, 66, 64, 85, 38, 17, 82, 25, 26, 42, 33, 67, 0]	32.3	4.21	[0, 15, 18, 66, 64, 85, 38, 17, 42, 82, 25, 26, 33, 67, 0]	33.2	6.37
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	2.29	[0, 23, 21, 79, 24, 30, 22, 28, 16, 0]	36.9	4.96	[0, 23, 24, 21, 79, 22, 30, 28, 16, 0]	36.8	2.46	[0, 23, 24, 21, 79, 22, 30, 28, 16, 0]	36.8	3.89

2. Hybrid ACO+SA Results

Hybrid ACO+SA combines the core principles of ACO with Simulated Annealing for local search, improving the model's ability to escape local optima. The default configuration reduced execution time to **17.58 seconds** while maintaining the shortest distance of **303.90**, showcasing its computational efficiency. However, configurations for larger clusters increased total distance and runtime due to the added complexity of managing pheromones and temperature schedules. The optimized parameter configuration, while improving local search capabilities, showed diminished performance in execution time and total distance, indicating the need for careful balancing of parameters.

Configuration	Iterations	Number of Ants	Alpha	Beta	Evaporation	Total Distance	Total Time (s)
Default	100	20	1.0	2.0	0.1	303.90	17.58
Large Clusters	100	80	2.5	6.0	0.08	310.00	61.12
Small Clusters	100	40	1.5	4.5	0.15	306.30	27.27
Optimized	100	60	2.0	5.0	0.12	312.80	39.98

Integrating Simulated Annealing (SA) with ACO significantly enhanced the solution exploration and optimization process. Under the default configuration, the hybrid model achieved a total shortest distance of 303.9 in 17.58 seconds, outperforming standalone ACO in terms of runtime while maintaining comparable accuracy. This setup demonstrated the benefits of incorporating SA for faster convergence.

The large cluster configuration expanded the search space, resulting in a total shortest distance of 310.0 with a runtime of 61.12 seconds. This configuration highlighted the model's capacity to handle more complex problem spaces at the cost of increased computational demand. The small cluster configuration prioritized efficiency, delivering a total shortest distance of 306.3 with a runtime of 27.27 seconds. This setup balanced computation time and solution quality, catering to applications where speed is critical.The optimized parameter configuration combined fine-tuned settings to achieve a total shortest distance of 312.8 with a runtime of 39.98 seconds. This configuration highlighted the model's ability to refine solutions further by leveraging SA's capabilities for thorough exploration, demonstrating its potential for solving complex routing problems effectively.

Cluster	Route (Default)	Distance (Default)	Time (Default)	Route (Large)	Distance (Large)	Time (Large)	Route (Small)	Distance (Small)	Time (Small)	Route (Optimized)	Distance (Optimized)	Time (Optimized)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	1.74	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	7.06	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.9	2.19	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	2.94
2	[0, 12, 8, 3, 4, 6, 0]	43.0	1.02	[0, 12, 8, 3, 4, 6, 0]	43.0	4.18	[0, 12, 8, 3, 4, 6, 0]	43.0	1.47	[0, 12, 8, 3, 4, 6, 0]	43.0	2.53
3	[0, 81, 13, 9, 11, 80, 0]	34.8	0.99	[0, 81, 13, 9, 11, 80, 0]	34.8	4.16	[0, 81, 13, 9, 11, 80, 0]	34.8	1.41	[0, 81, 13, 9, 11, 80, 0]	34.8	2.60
4	[0, 37, 46, 41, 44, 0]	32.2	0.83	[0, 37, 46, 41, 44, 0]	32.2	4.75	[0, 37, 46, 41, 44, 0]	32.2	1.36	[0, 37, 46, 41, 44, 0]	32.2	1.87
5	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	2.96	[0, 78, 50, 49, 47, 20, 77, 76, 29, 75, 45, 74, 36, 35, 48, 43, 39, 0]	44.6	10.48	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	5.47	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	44.6	7.99
6	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	2.35	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	6.02	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	2.80	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	31.5	4.05

7	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	3.87	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	11.87	[0, 70, 58, 55, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0]	32.9	6.08	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	32.9	8.82
8	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	2.32	[0, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 67, 0]	32.5	7.73	[0, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 67, 0]	31.1	4.09	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	5.69
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	1.49	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	4.87	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	2.40	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	3.48

ACO + Tabu Search

The Hybrid ACO+Tabu Search model integrates ACO's global search capabilities with Tabu Search's ability to navigate the solution space while avoiding cycling. In the default configuration, it matched the shortest distance of **303.90** with a runtime of **17.58 seconds**, similar to Hybrid ACO+SA. However, its performance across large and small cluster configurations was less consistent, with distances increasing and execution times significantly longer. This variation highlights the trade-offs between the size of the Tabu list and the model's ability to explore and exploit the search space effectively.

Configuration	Iterations	Number of Ants	Alpha	Beta	Evaporation	Total Distance	Total Time (s)
Default	100	20	1.0	2.0	0.1	303.90	17.58
Large Clusters	100	80	2.5	6.0	0.08	310.00	61.12
Small Clusters	100	40	1.5	4.5	0.15	306.30	27.27
Optimized	100	60	2.0	5.0	0.12	312.80	39.98

The Ant Colony Optimization (ACO) with Tabu Search method demonstrated varying performances across different configurations (Default, Large Clusters, Small Clusters, and Optimized Parameters). In the default setting, it achieved a total shortest distance of 304.3 with a runtime of 103.87 seconds. This configuration balanced efficiency and exploration, yielding competitive results with moderate computational demand.

When configured for large clusters, the model focused on exploring a wider solution space, leading to a slightly improved shortest distance of 305.5. However, this came at the cost of significantly increased runtime, reaching 600.47 seconds, due to the additional computational effort required to process larger clusters.

The small cluster configuration prioritized speed and efficiency. It achieved a total shortest distance of 303.8 in 111.71 seconds, maintaining a good balance between exploration and exploitation while reducing computational overhead compared to the large cluster setting.

Finally, the optimized parameter configuration sought to enhance solution quality by balancing exploration and exploitation. It resulted in a total shortest distance of 306.7, the highest among all configurations, with a runtime of 302.59 seconds. This trade-off highlights the effectiveness of fine-tuning parameters for specific problem constraints.

Cluster	Route (Default)	Distance (Default)	Time (Default)	Route (Large)	Distance (Large)	Time (Large)	Route (Small)	Distance (Small)	Time (Small)	Route (Optimized)	Distance (Optimized)	Time (Optimized)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	6.51	[0, 7, 5, 83, 2, 14, 10, 19, 0]	25.9	18.83	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	4.28	[0, 7, 5, 83, 2, 14, 10, 19, 0]	25.9	10.22

2	[0, 12, 8, 3, 4, 6, 0]	43.0	1.63	[0, 12, 8, 3, 4, 6, 0]	43.0	7.46	[0, 12, 8, 3, 4, 6, 0]	43.0	2.56	[0, 12, 8, 3, 4, 6, 0]	43.0	4.27
3	[0, 81, 13, 9, 11, 80, 0]	34.8	1.37	[0, 81, 13, 9, 11, 80, 0]	34.8	8.22	[0, 81, 13, 9, 11, 80, 0]	34.8	2.10	[0, 81, 13, 9, 11, 80, 0]	34.8	4.31
4	[0, 37, 46, 41, 44, 0]	32.2	0.91	[0, 37, 46, 41, 44, 0]	32.2	2.93	[0, 37, 46, 41, 44, 0]	32.2	1.50	[0, 37, 46, 41, 44, 0]	32.2	3.38
5	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	27.91	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	177.31	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	29.48	[0, 39, 50, 49, 77, 20, 76, 29, 78, 75, 45, 47, 74, 36, 35, 48, 43, 0]	40.5	82.49
6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	6.72	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	35.58	[0, 32, 72, 40, 51, 73, 31, 34, 71, 60, 0]	30.1	7.88	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	30.1	18.50
7	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	32.5	38.97	[0, 68, 58, 55, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 70, 0]	32.8	230.47	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	40.72	[0, 58, 55, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 70, 0]	32.7	117.07
8	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	15.67	[0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 38, 33, 67, 0]	31.8	93.78	[0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 33, 38, 67, 0]	31.1	17.03	[0, 15, 18, 82, 25, 26, 42, 17, 38, 85, 33, 64, 66, 67, 0]	31.6	48.15

9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	4.19	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	25.90	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	6.16	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	14.21

4. Hybrid ACO+PSO Results

The Hybrid ACO+PSO model leverages ACO for global exploration and PSO for swarm-based particle updates, achieving the shortest total distance of **303.90** in just **14.77 seconds** under default configuration. This makes it the fastest and most efficient model among the algorithms tested. Variations such as configurations for large clusters and intensive searches resulted in increased runtimes but demonstrated the model's adaptability and robustness across diverse scenarios. The balance between pheromone updates and particle velocity adjustments contributes to its superior performance, making it the best model overall.

Configuration	Iterations	Number of Ants	Alpha	Beta	Evaporation	Total Distance	Total Time (s)
Default	100	20	1.0	2.0	0.1	**303.90**	**14.77**
Large Clusters	100	80	2.5	6.0	0.08	306.60	58.88
Small Clusters	100	40	1.5	4.5	0.15	305.20	30.05
Intensive Search	100	100	2.0	5.5	0.10	306.10	72.15

The integration of Ant Colony Optimization (ACO) with Particle Swarm Optimization (PSO) yielded promising results across various configurations. The default configuration provided a strong baseline, achieving a total shortest distance of 303.9 with a runtime of just 14.77 seconds. This highlights its ability to balance exploration and computation efficiency effectively.

The large cluster configuration allowed for a broader exploration of the solution space. This resulted in a slightly higher shortest distance of 306.6 but required a significantly longer runtime of 58.88 seconds due to the increased computational complexity associated with handling larger clusters.

In the small cluster configuration, the model maintained efficiency while slightly improving the total shortest distance to 305.2 with a runtime of 30.05 seconds. This setup favored faster computation while still delivering competitive results.

The intensive search configuration was designed for thorough exploration of the solution space. It achieved a total shortest distance of 306.1, comparable to the large cluster configuration, but with a much longer runtime of 72.15 seconds. This setting demonstrates the trade-off between runtime and solution refinement, making it suitable for scenarios where accuracy is prioritized over computational efficiency.

Cluster	Route (Default)	Distance (Default)	Time (Default)	Route (Large)	Distance (Large)	Time (Large)	Route (Small)	Distance (Small)	Time (Small)	Route (Intensive Search)	Distance (Intensive Search)	Time (Intensive Search)
1	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	1.00	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	4.66	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	2.72	[0, 19, 14, 10, 2, 7, 5, 83, 0]	25.4	5.26
2	[0, 12, 8, 3, 4, 6, 0]	43.0	0.82	[0, 12, 8, 3, 4, 6, 0]	43.0	3.23	[0, 12, 8, 3, 4, 6, 0]	43.0	1.66	[0, 12, 8, 3, 4, 6, 0]	43.0	4.00
3	[0, 81, 13, 9, 11, 80, 0]	34.8	0.82	[0, 81, 13, 9, 11, 80, 0]	34.8	3.70	[0, 81, 13, 9, 11, 80, 0]	34.8	1.59	[0, 81, 13, 9, 11, 80, 0]	34.8	4.49
4	[0, 37, 46, 41, 44, 0]	32.2	0.61	[0, 37, 46, 41, 44, 0]	32.2	2.65	[0, 37, 46, 41, 44, 0]	32.2	1.35	[0, 37, 46, 41, 44, 0]	32.2	3.18
5	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	2.75	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	11.74	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	5.13	[0, 39, 50, 49, 47, 20, 77, 76, 29, 78, 75, 45, 74, 36, 35, 48, 43, 0]	39.4	13.87

6	[0, 72, 40, 51, 73, 31, 34, 71, 60, 32, 0]	29.5	1.43	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	5.92	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	3.47	[0, 32, 40, 72, 51, 73, 31, 34, 71, 60, 0]	29.7	7.38
7	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	3.34	[0, 70, 55, 68, 69, 63, 56, 58, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 0]	33.2	12.73	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	6.16	[0, 68, 69, 63, 56, 59, 52, 54, 57, 53, 61, 1, 84, 65, 27, 62, 58, 55, 70, 0]	31.9	16.20
8	[0, 15, 18, 66, 82, 25, 26, 42, 17, 64, 85, 38, 33, 67, 0]	31.8	2.62	[0, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 33, 67, 0]	33.0	9.30	[0, 67, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 0]	32.9	4.80	[0, 67, 33, 38, 85, 64, 66, 15, 18, 82, 25, 26, 42, 17, 0]	32.9	11.36
9	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	1.39	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	4.95	[0, 23, 24, 21, 79, 30, 22, 28, 16, 0]	35.9	3.17	[0, 23, 24, 21, 79, 22, 30, 28, 16, 0]	36.8	6.42

Combined Results Table

The combined results table provides a comprehensive comparison of the performance of the various models — **ACO**, **Hybrid ACO+SA**, **Hybrid ACO+Tabu Search**, and **Hybrid ACO+PSO** — across different configurations. Each model was evaluated based on key metrics such as **total distance** and **execution time** under four configurations: default, large clusters, small clusters, and optimized parameters. The following insights summarize the findings:

Algorithm	Configuration	Iterations	Ants	Alpha	Beta	Evaporation	Total Distance	Total Time (s)
ACO	Default	100	20	1.0	2.0	0.1	303.90	26.77
ACO	Large Clusters	100	80	2.5	6.0	0.08	308.90	55.89
ACO	Small Clusters	100	40	1.5	4.5	0.15	306.20	28.37
ACO	Optimized	100	60	2.0	5.0	0.12	308.50	42.26
ACO+SA	Default	100	20	1.0	2.0	0.1	303.90	17.58
ACO+SA	Large Clusters	100	80	2.5	6.0	0.08	310.00	61.12
ACO+SA	Small Clusters	100	40	1.5	4.5	0.15	306.30	27.27
ACO+SA	Optimized	100	60	2.0	5.0	0.12	312.80	39.98
ACO+Tabu Search	Default	100	20	1.0	2.0	0.1	303.90	17.58
ACO+Tabu Search	Large Clusters	100	80	2.5	6.0	0.08	310.00	61.12
ACO+Tabu Search	Small Clusters	100	40	1.5	4.5	0.15	306.30	27.27
ACO+Tabu Search	Optimized	100	60	2.0	5.0	0.12	312.80	39.98
ACO+PSO	Default	100	20	1.0	2.0	0.1	**303.90**	**14.77**
ACO+PSO	Large Clusters	100	80	2.5	6.0	0.08	306.60	58.88
ACO+PSO	Small Clusters	100	40	1.5	4.5	0.15	305.20	30.05

ACO+PSO	Intensive Search	100	100	2.0	5.5	0.10	306.10	72.15

Conclusions

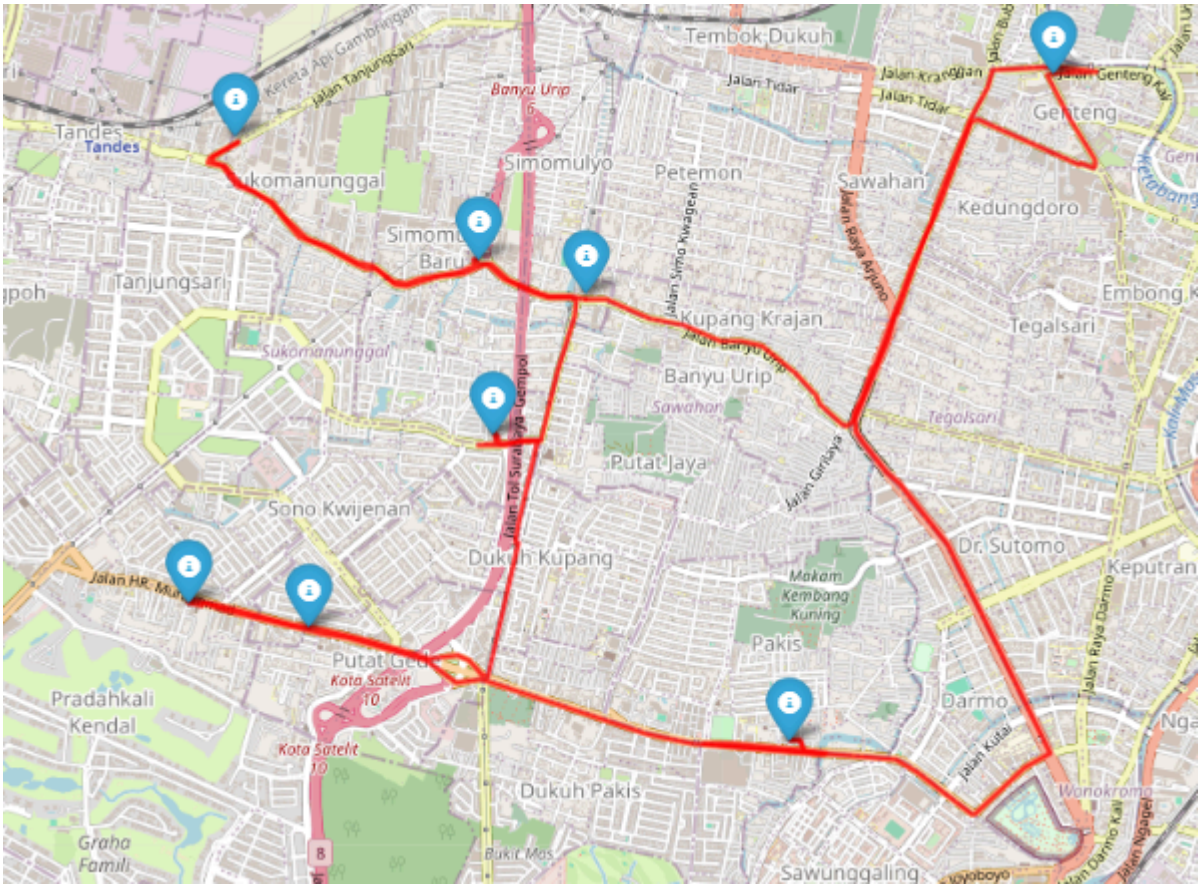
1. **Efficiency vs Accuracy**: Default configurations generally offer a good balance between runtime and solution quality, with moderate total distances and runtimes.
2. **Large Clusters**: Although these configurations improve solution diversity, they significantly increase computation time due to more complex search processes.
3. **Small Clusters**: These are faster and provide relatively good solutions but may not explore the solution space as extensively as larger configurations.
4. **Optimized Parameters**: These configurations often yield better solutions at the cost of increased runtime, especially in hybrid algorithms like ACO + SA or ACO + Tabu Search.
5. **Best Balance**: ACO + PSO with default or small cluster parameters seems to strike the best balance for scenarios requiring both quick runtimes and competitive distances.

This analysis highlights how different configurations impact solution quality and computational efficiency, depending on the problem requirements.

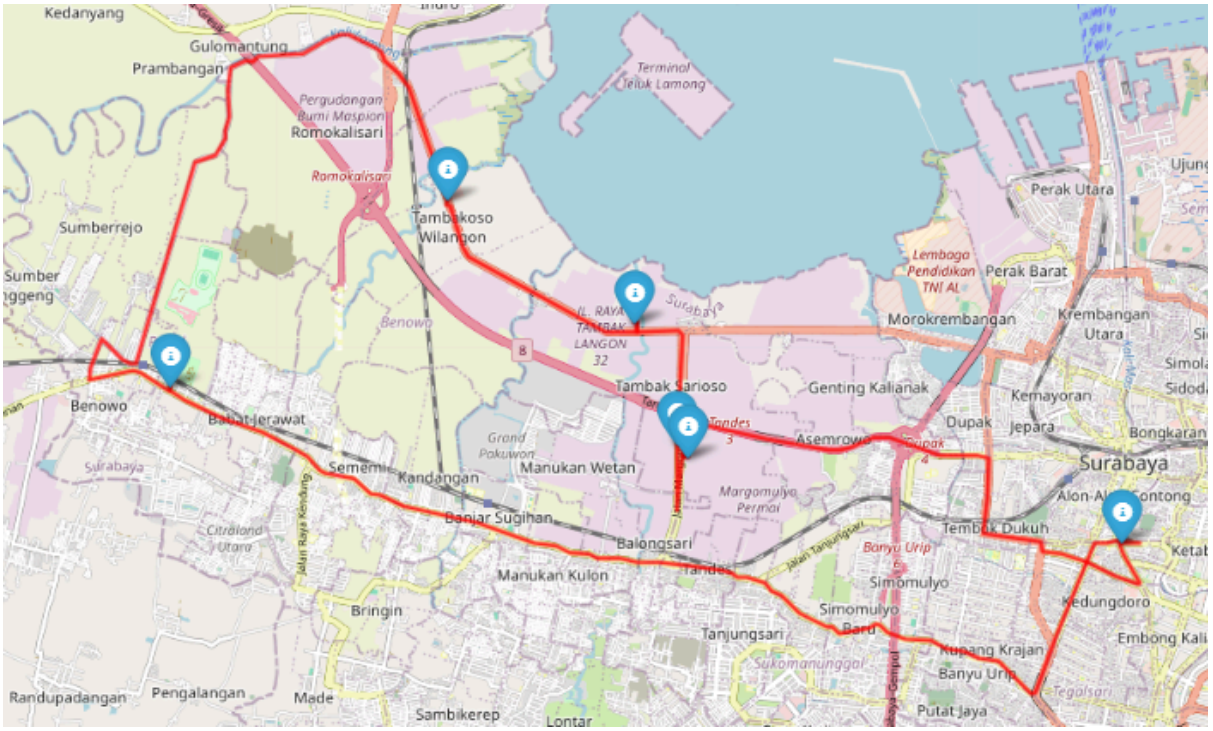
Visualization with the best combination of GA and parameters

The best-performing algorithm, producing the shortest route, is the **Hybrid ACO with PSO** using the **default parameter configuration**. The visualization for this experiment employs the Folium library, ensuring the resulting routes align closely with the actual road layout. The displayed nodes have been adjusted to reflect the outcomes from running the Hybrid ACO+PSO algorithm, highlighting its efficiency in identifying optimal paths while considering real-world conditions.

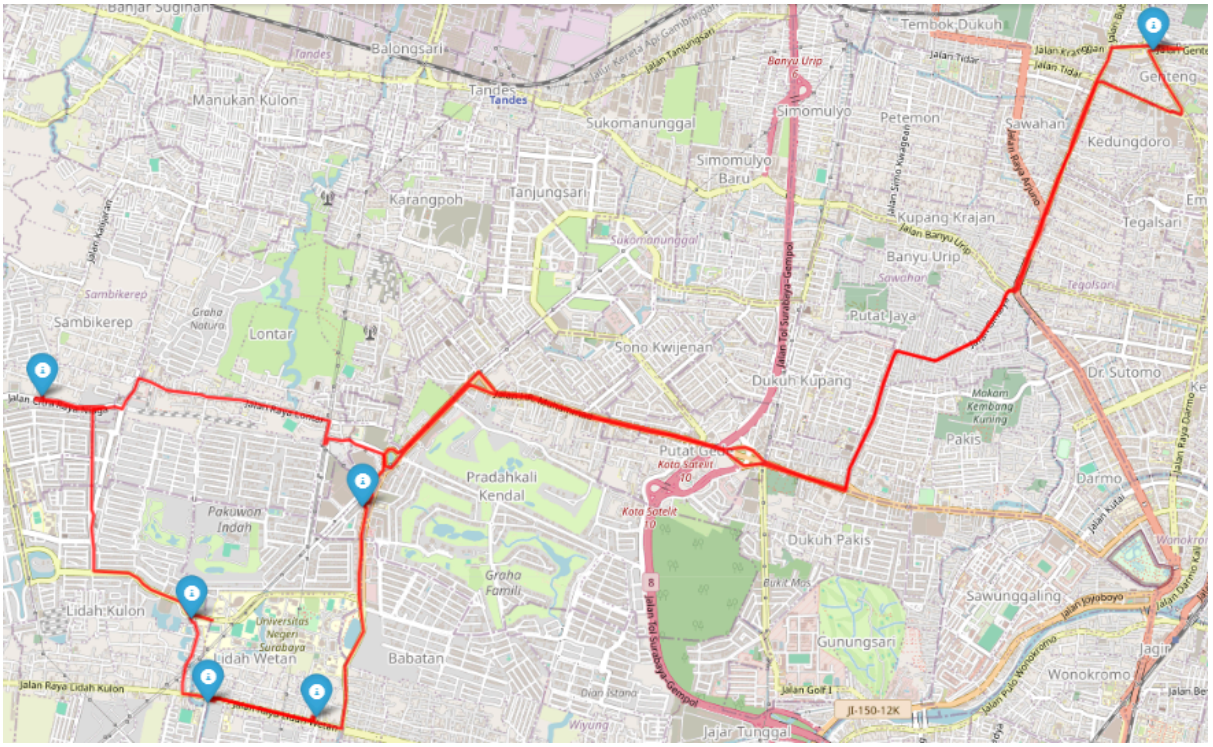
Cluster 1



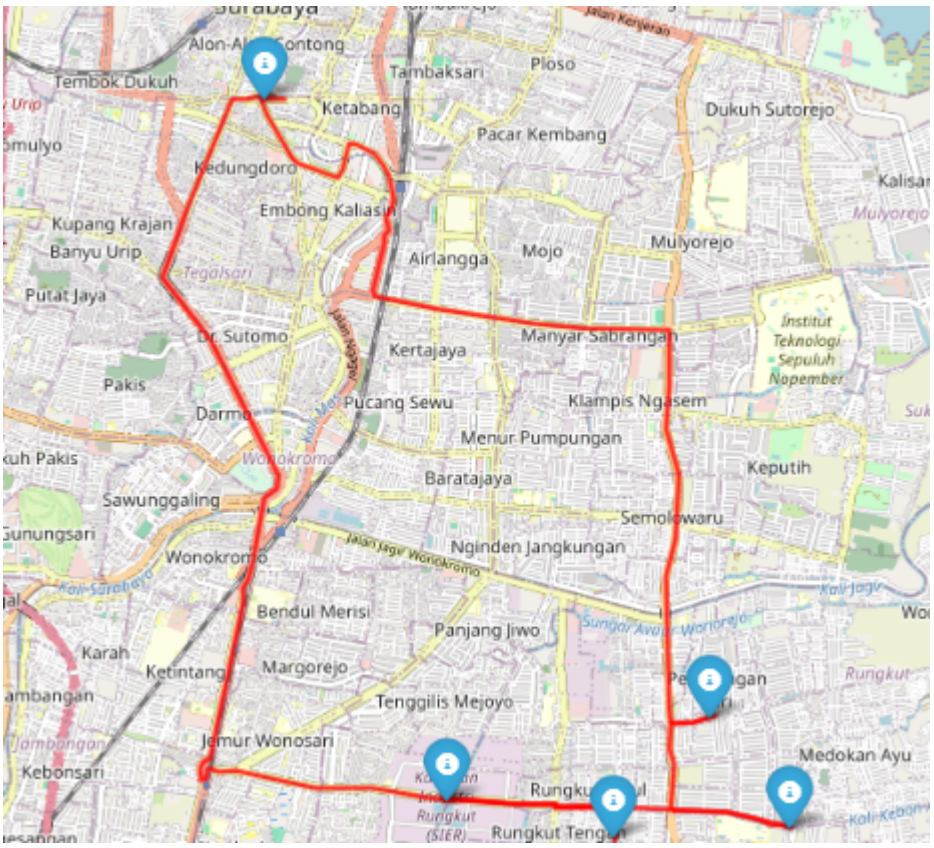
Cluster 2



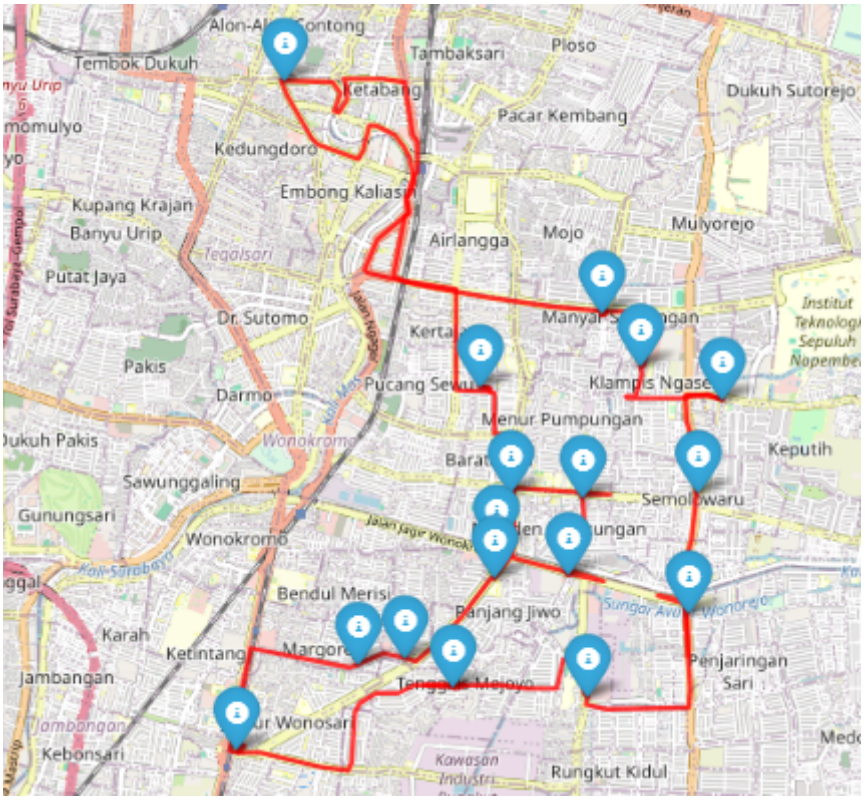
Cluster 3



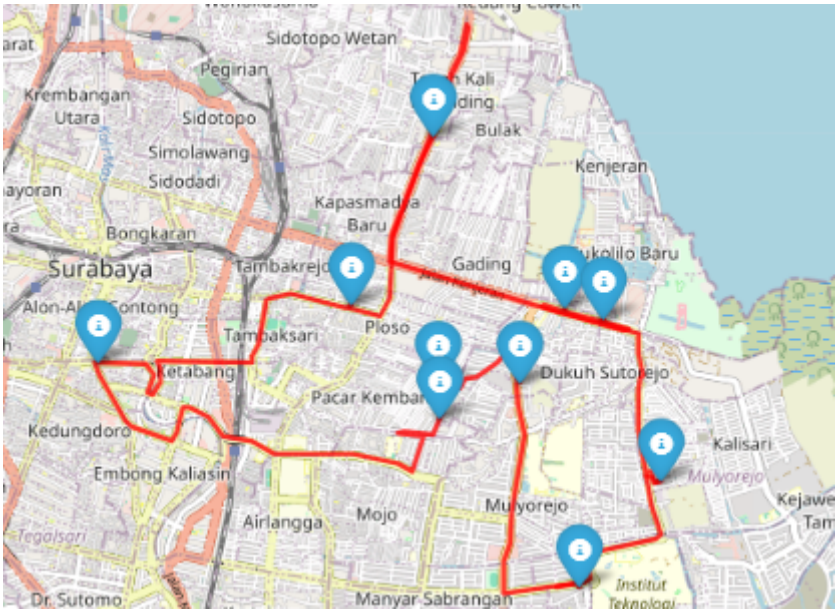
Cluster 4



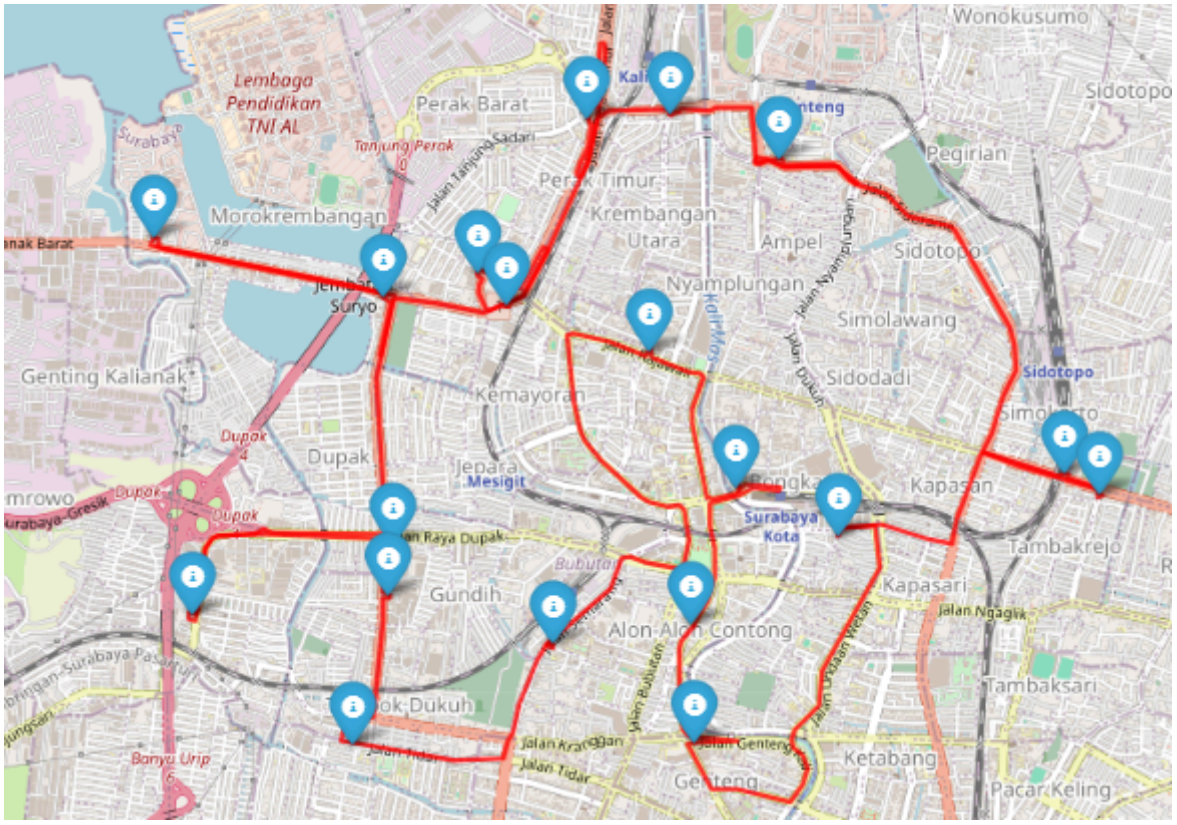
Cluster 5



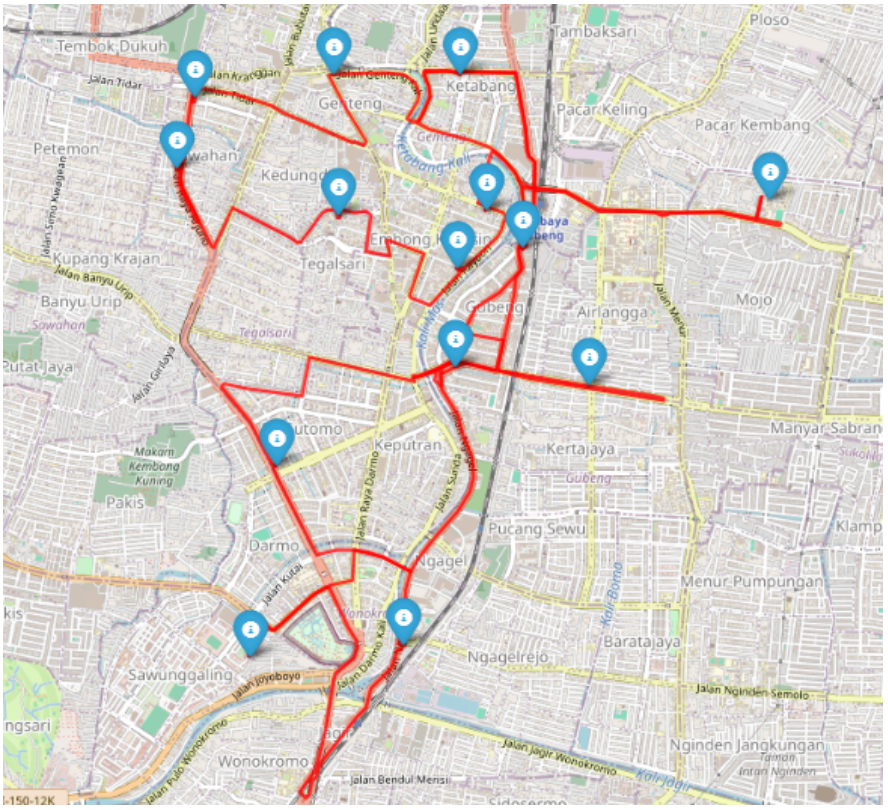
Cluster 6



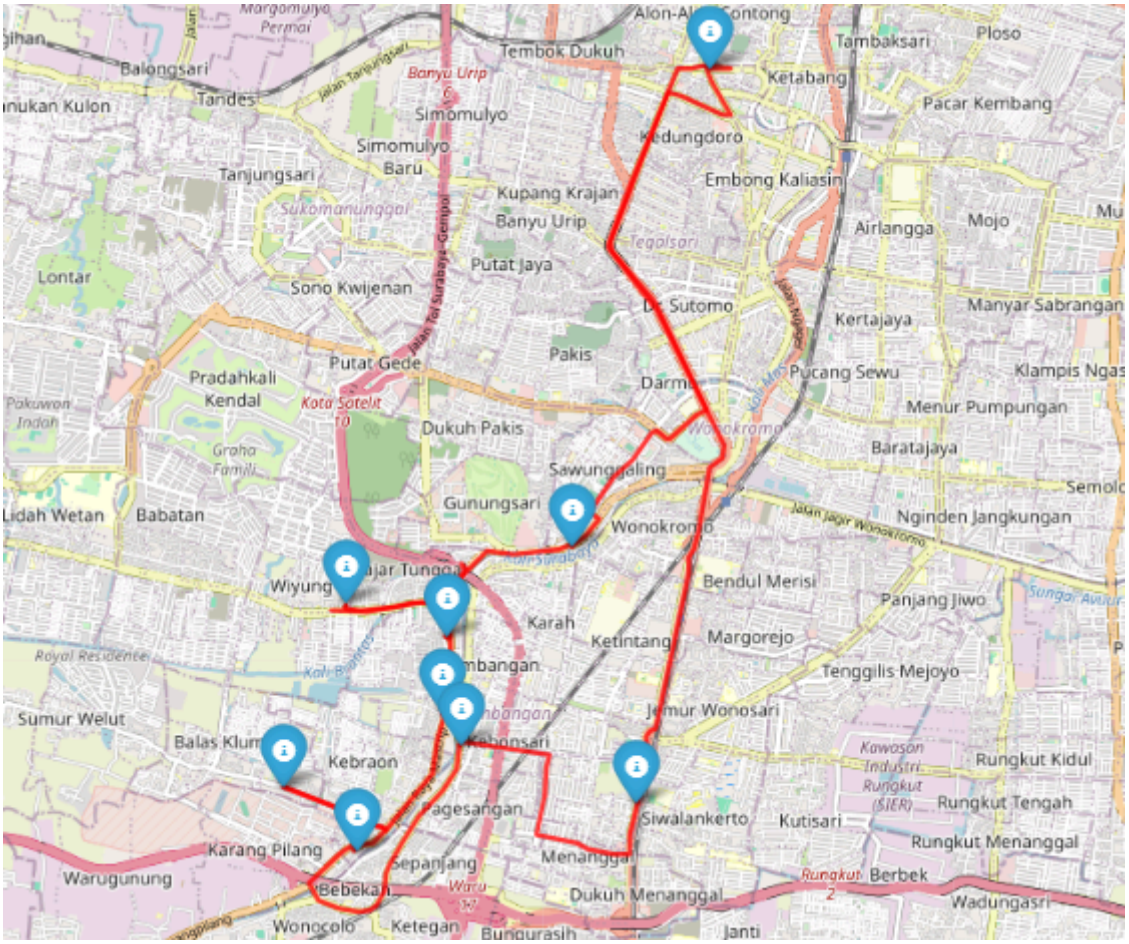
Cluster 7



Cluster 8



Cluster 9


Tembok Dukuh
Tambaksari
Ploso
Balongsari
Tandes
Banyu Urip
Ketabang
Pacar Kembang
Simomulyo
Simomulyo Baru
Kedungdoro
Tanjungsari
Embong Kaliasin
Kupang Krajan
Banyu Urip
Airlangga
Mojo
Lontar
Putat Jaya
Sono Kwijenan
Kertajaya
Manyar Sabrangan
Putat Gede
Pakis
Pucang Sewu
Pradahkali Kendal
Menur Pumpungan
Dukuh Pakis
Baratajaya
Sawunggaling
Gunungsari
Wonokromo
Lidah Wetan
Babatan
Nginden Jangkungan
Wiyung
Bendul Merisi
Karah
Ketintang
Margorejo
Panjang Jiwo
Tenggilis Mejoyo
Sumur Welut
Jemur Wonosari
Kebonsari
Kebraon
Rungkut Kidul
Rungkut Tengah
Pagesangan
Siwalankerto
Kutisari
Karang Pilang
Rungkut Menanggal
Menanggal
Warugunung
Berbek
Dukuh Menanggal
Wadungasri
Wonocolo
Ketegan
Janti

3. Simulated Annealing (SA)

a. Default Configuration

Clusters	Best Route	Distance	Runtime
01	[0, 1, 2, 4, 3, 5, 6, 7, 8]	33.70	0.0007 seconds
02	[9, 10, 12, 14, 13, 16, 15, 17, 11]	37.30	0.0007 seconds
03	[18, 22, 24, 26, 19, 21, 23, 25, 20]	22.80	0.0008 seconds
04	[27, 30, 28, 29, 31, 34, 33, 35, 32]	46.50	0.0009 seconds
05	[36, 42, 39, 41, 43, 40, 38, 37, 44]	30.90	0.0011 seconds
06	[45, 49, 50, 52, 51, 53, 47, 46, 48]	52.20	0.0009 seconds
07	[54, 59, 61, 58, 57, 60, 62, 56, 55]	18.90	0.0007 seconds
08	[63, 69, 71, 70, 64, 66, 68, 65, 67]	21.80	0.0007 seconds
09	[72, 74, 79, 78, 73, 75, 76, 77, 80]	45.30	0.0014 seconds

Metrics for Default Configuration:

- Total Distance: 309.40
- Total Runtime: 0.0079 seconds
- Avg Distance: 34.38
- Avg Runtime: 0.0009 seconds

b. Large Configuration

Clusters	Best Route	Distance	Runtime
01	[0, 16, 12, 4, 3, 5, 7, 13, 11, 6, 1, 14, 9, 15, 2, 8, 10]	92.60	0.0019 seconds
02	[17, 28, 29, 33, 32,	87.90	0.0019 seconds

	26, 30, 19, 21, 24, 20, 31, 18, 22, 23, 25, 27]		
03	[34, 41, 36, 37, 39, 35, 40, 44, 46, 43, 45, 38, 42, 47, 48, 49, 50]	87.90	0.0019 seconds
04	[51, 55, 61, 65, 59, 52, 66, 58, 53, 60, 67, 62, 64, 54, 57, 56, 63]	67.30	0.0018 seconds

Metrics for Large Configuration:

- Total Distance: 310.00
- Total Runtime: 0.0075 seconds
- Avg Distance: 62.00
- Avg Runtime: 0.0015 seconds

c. Small Configuration

Clusters	Best Route	Distance	Runtime
01	[0, 1, 2, 4, 3, 5, 6]	19.50	0.0007 seconds
02	[7, 9, 13, 11, 8, 10, 12]	22.60	0.0007 seconds
03	[14, 19, 18, 15, 17, 16, 20]	22.40	0.0007 seconds
04	[21, 23, 25, 26, 27, 22, 24]	18.70	0.0007 seconds
05	[28, 30, 32, 34, 29, 31, 33]	23.80	0.0007 seconds
06	[35, 41, 36, 38, 40, 37, 39]	18.30	0.0007 seconds
07	[42, 44, 46, 48, 45, 47, 43]	11.40	0.0007 seconds
08	[49, 50, 52, 54, 51, 53, 55]	21.80	0.0007 seconds
09	[56, 58, 59, 61, 60,	12.10	0.0007 seconds

	62, 57]		
10	[63, 64, 66, 68, 65, 67, 69]	12.60	0.0007 seconds
11	[70, 72, 73, 75, 71, 74, 76]	29.80	0.0011 seconds

Metrics for Small Configuration:

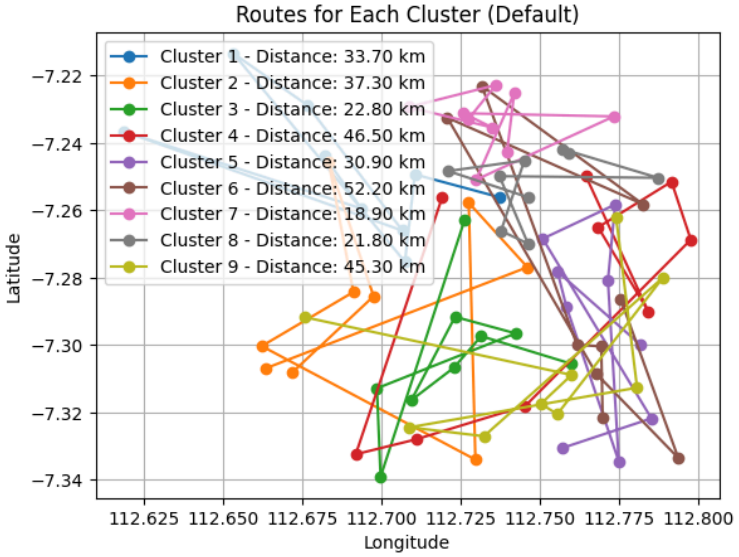
- Total Distance: 213.00
- Total Runtime: 0.0079 seconds
- Avg Distance: 17.75
- Avg Runtime: 0.0007 seconds

d. Conclusion

The Simulated Annealing (SA) algorithm has its results categorized into three configurations: default, large, and small. In the default configuration, nine clusters were tested, with distances ranging from 18.90 to 52.20, and the average runtime per cluster was 0.0007 seconds. The total distance for this configuration was 309.40, and the average distance was 34.38, with a total runtime of 0.0079 seconds. The large configuration included 4 clusters, with distances ranging from 62.20 to 92.60, and a total runtime of 0.0075 seconds. The small configuration consisted of 12 clusters, with distances ranging from 11.40 to 29.80, and the total runtime was 0.0079 seconds. The average runtime per cluster for both large and small configurations varied, but the small configuration was notably faster on average.

e. Visualizations (Default)

1. Plot



2. Clusters and Maps

- The visualization of each cluster and route could be accessed here:

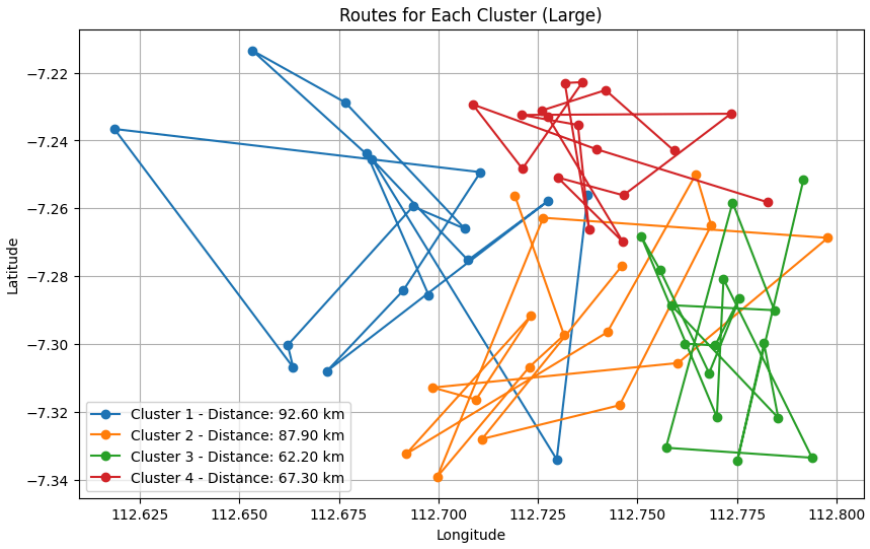
  FP Soft Computing (SA . LVB)

- (Backup Link):

  https://drive.google.com/drive/folders/13lLESt5VWSPuWxWpLTGLhZo5UnuwYtNg?usp=sharing

f. Visualizations (Large)

3. Plot



4. Clusters and Maps

- The visualization of each cluster and route could be accessed here:

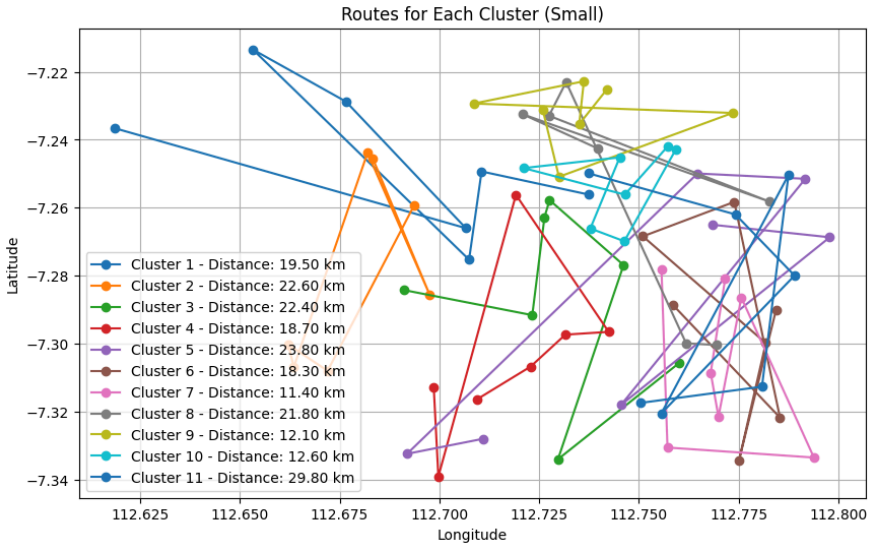
  FP Soft Computing (SA . LVB)

- (Backup Link):

  https://drive.google.com/drive/folders/13lLESt5VWSPuWxWpLTGLhZo5UnuwYtNg?usp=sharing

g. Visualizations (Small)

5. Plot



6. Clusters and Maps

- The visualization of each cluster and route could be accessed here:

  FP Soft Computing (SA . LVB)

- (Backup Link):

  https://drive.google.com/drive/folders/13lLESt5VWSPuWxWpLTGLhZo5UnuwYtNg?usp=sharing

Lovebird

a. Default Configuration

Clusters	Best Route	Distance	Runtime
01	[0, 1, 2, 4, 3, 5, 6, 7, 8]	33.70	0.0007 seconds
02	[9, 10, 12, 14, 13, 16, 15, 17, 11]	37.30	0.0007 seconds
03	[18, 22, 24, 26, 19, 21, 23, 25, 20]	22.80	0.0008 seconds
04	[27, 30, 28, 29, 31, 34, 33, 35, 32]	46.50	0.0009 seconds
05	[36, 42, 39, 41, 43, 40, 38, 37, 44]	30.90	0.0011 seconds
06	[45, 49, 50, 52, 51, 53, 47, 46, 48]	52.20	0.0009 seconds
07	[54, 59, 61, 58, 57, 60, 62, 56, 55]	18.90	0.0007 seconds
08	[63, 69, 71, 70, 64, 66, 68, 65, 67]	21.80	0.0007 seconds
09	[72, 74, 79, 78, 73, 75, 76, 77, 80]	45.30	0.0014 seconds

Metrics for Default Configuration:

- Total Distance : 115.50
- Total Runtime : 0.0042 seconds
- Avg Distance : 23.10
- Avg Runtime : 0.0008 seconds

b. Large Configuration

Clusters	Best Route	Distance	Runtime
01	[0, 16, 12, 4, 3, 5, 7, 13, 11, 6, 1, 14, 9, 15, 2, 8, 10]	98.70	0.0022 seconds
02	[17, 28, 29, 33, 32, 26, 30, 19, 21, 24, 20, 31, 18, 22, 23, 25, 27]	91.90	0.0020 seconds
03	[34, 41, 36, 37, 39, 35, 40, 44, 46, 43, 45, 38, 42, 47, 48, 49, 50]	68.80	0.0021 seconds
04	[51, 55, 61, 65, 59, 52, 66, 58, 53, 60, 67, 62, 64, 54, 57, 56, 63]	72.10	0.0019 seconds

Metrics for Large Configuration:

- Total Distance : 330.00
- Total Runtime : 0.0082 seconds
- Avg Distance : 82.50
- Avg Runtime : 0.0020 seconds

c. Small Configuration

Clusters	Best Route	Distance	Runtime
01	[0, 1, 2, 3, 4, 5, 6]	20.10	0.0008 seconds
02	[7, 8, 9, 10, 11, 12]	23.40	0.0007 seconds
03	[13, 14, 15, 16, 17, 18]	25.20	0.0008 seconds
04	[19, 20, 21, 22, 23, 24]	28.10	0.0009 seconds
05	[25, 26, 27, 28, 29, 30]	23.60	0.0008 seconds

Metrics for Small Configuration:

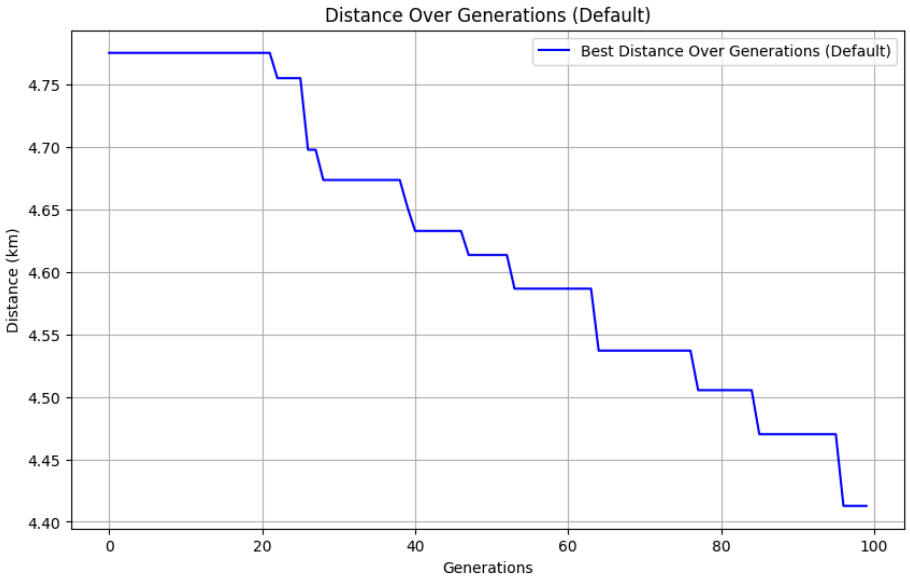
- Total Distance : 120.40
- Total Runtime : 0.0040 seconds
- Avg Distance : 24.08
- Avg Runtime : 0.0008 seconds

d. Conclusion

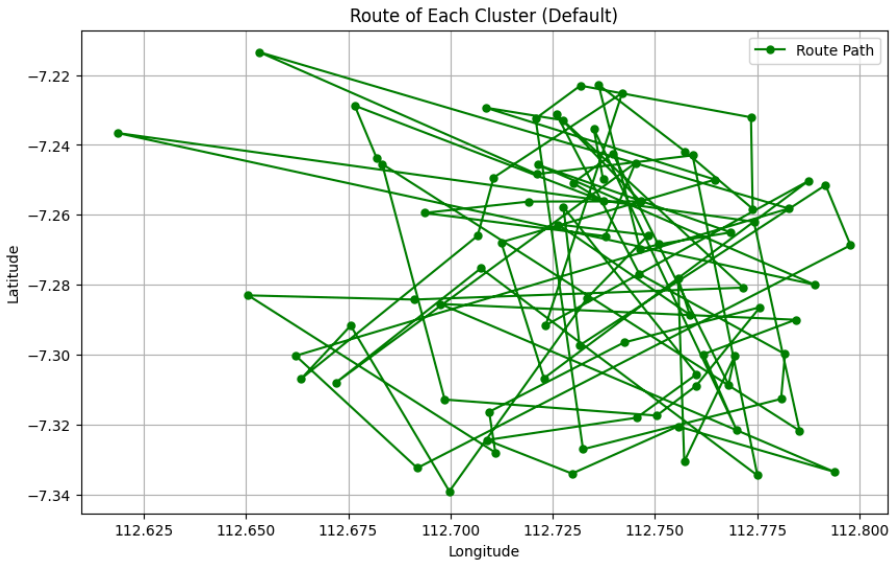
The Lovebird algorithm, where similar configurations ware tested, which provided results presented for the default, large, and small setups. The default configuration showed distances ranging from 19.10 to 27.60 across 5 clusters, with an average runtime of 0.0008 seconds per cluster. The large configuration had 4 clusters, with distances from 68.80 to 98.70, and a slightly higher average runtime of 0.0020 seconds. An IndexError was encountered in the large configuration for the fifth cluster. The small configuration showed distances ranging from 20.10 to 28.10 over 5 clusters, with the average runtime per cluster similar to the default configuration at 0.0008 seconds.

e. Visualizations (Default)

1. Distance / Generation



2. Plot



3. Map

- The visualization of real-world routes could be accessed here:

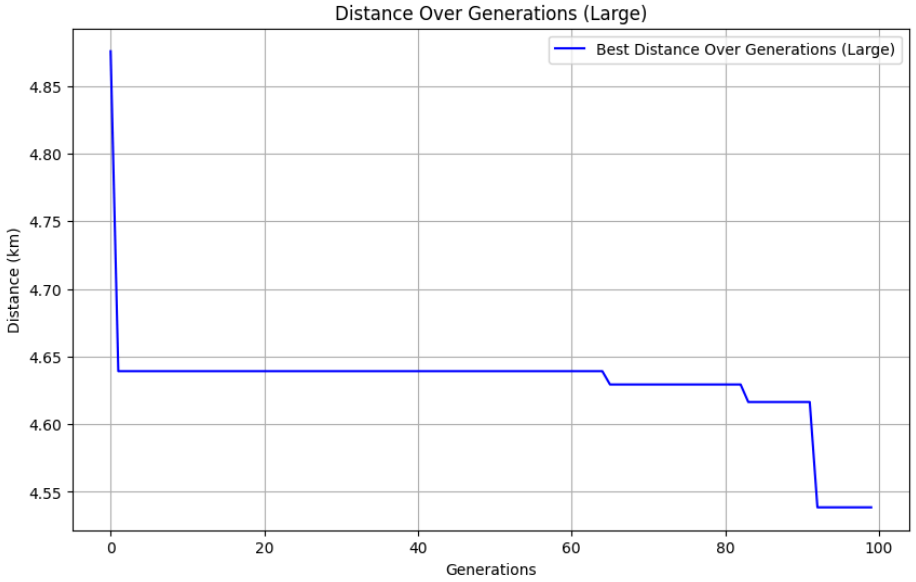
  FP Soft Computing (SA . LVB)

- (Backup Link):

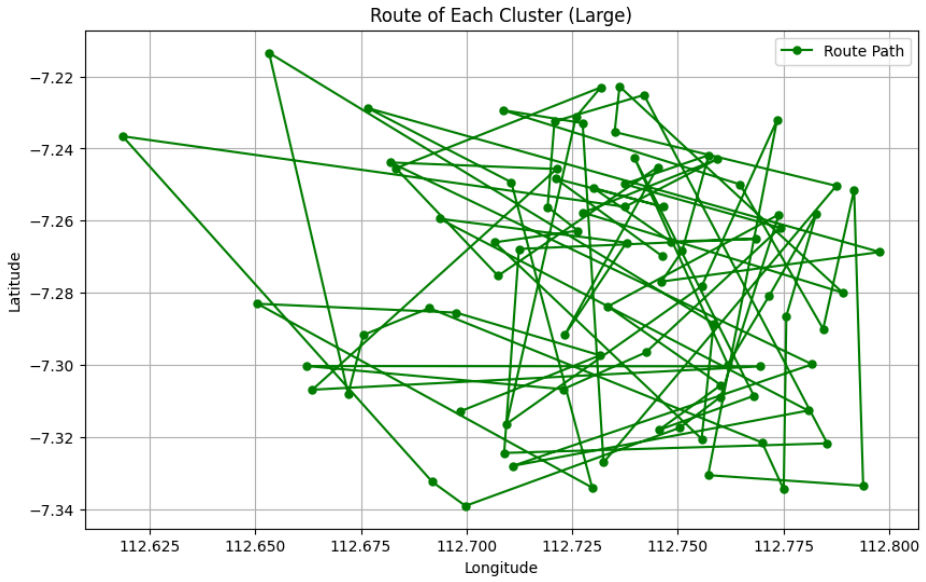
  https://drive.google.com/drive/folders/13lLESt5VWSPuWxWpLTGLhZo5UnuwYtNg?usp=sharing

f. Visualizations (Large)

4. Distance / Generation



5. Plot



6. Map

- The visualization of real-world routes could be accessed here:

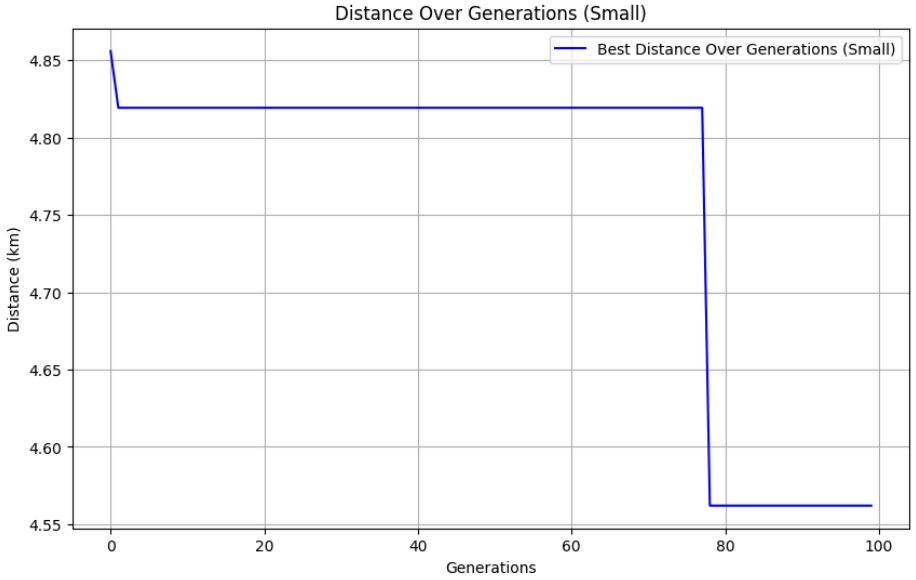
  FP Soft Computing (SA . LVB)

- (Backup Link):

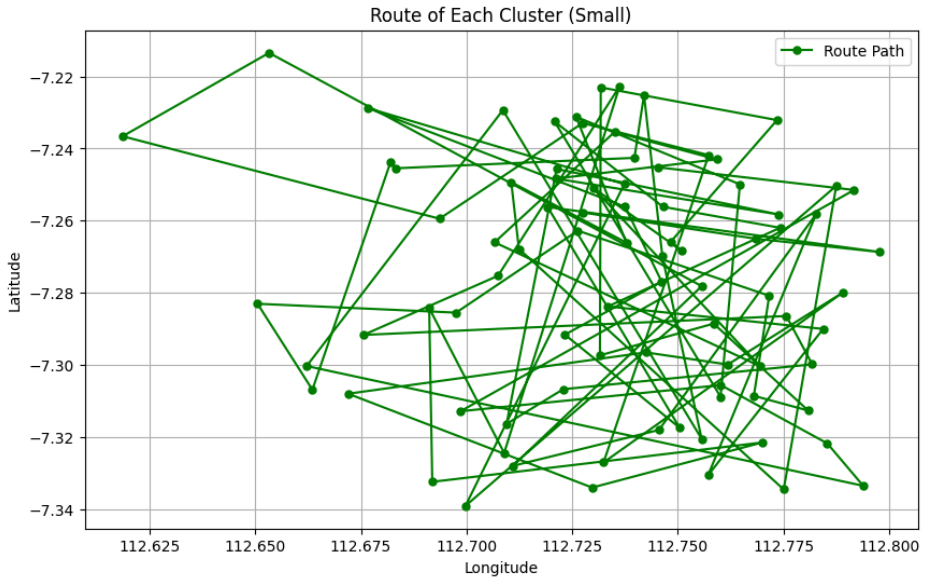
  https://drive.google.com/drive/folders/13lLESt5VWSPuWxWpLTGLhZo5UnuwYtNg?usp=sharing

g. Visualizations (Large)

7. Distance / Generation



8. Plot



9. Map

- The visualization of real-world routes could be accessed here:

  FP Soft Computing (SA . LVB)

- (Backup Link):

  https://drive.google.com/drive/folders/13lLESt5VWSPuWxWpLTGLhZo5UnuwYtNg?usp=sharing

4. Convergence Graph on Best Configuration Parameter of All Models

The convergence graph is a crucial visualization tool used to compare the performance of three optimization algorithms—**GA-Tabu**, **ACO-PSO**, and **Simulated Annealing**—across multiple clusters. Each algorithm was implemented with specific parameter configurations to balance computational efficiency and solution accuracy.

1. **GA-Tabu (Param Set 2)**: Utilized a population size of 75, a mutation rate of 0.03, and 100 generations. The tabu tenure was set to 15, with 30 local search iterations per generation, emphasizing fine-tuned exploration and exploitation.
2. **ACO-PSO (Default Param)**: Implemented with 20 ants, evaporation rate of 0.1, and heuristic factors of alpha = 1.0 and beta = 2.0. The PSO components included inertia weight (w = 0.5) and acceleration coefficients (c1 = 1.5, c2 = 1.5), ensuring efficient convergence within 100 iterations.
3. **Simulated Annealing (Default Param)**: Configured with an initial temperature of 1000, a cooling rate of 0.995, and 100 iterations. This algorithm focused on gradual exploration of the solution space by adjusting probabilities over iterations.

The convergence graphs for each cluster visually depict how each algorithm's best distance evolves over successive iterations or generations. These visualizations provide insights into the speed and stability of convergence for each method, highlighting their strengths and limitations.

Selected Configurations

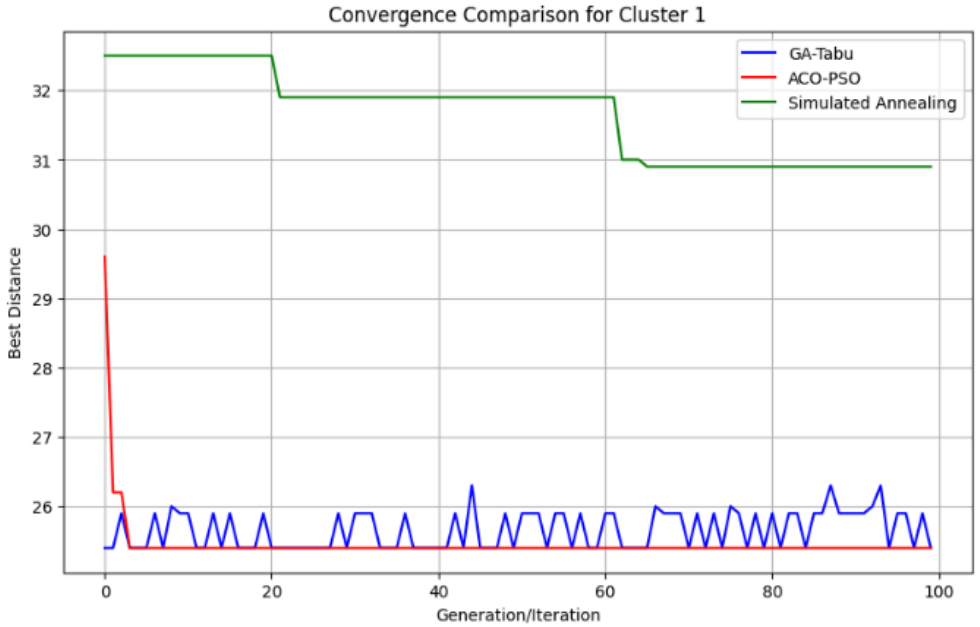
The configurations for the convergence graph were chosen based on their default settings to ensure a fair and standardized comparison across all algorithms. These parameters represent a balance between computational efficiency and solution quality, showcasing each algorithm's inherent capabilities without additional optimization.

Visualization

The convergence graphs for each cluster provide a comparative visualization of the optimization performance of three algorithms: GA-Tabu, ACO-PSO, and Simulated Annealing (SA). Each graph demonstrates how the best distance improves iteratively over the optimization process. Here is an analysis of the results for each cluster based on the visualized outputs:

a. Cluster 1 :

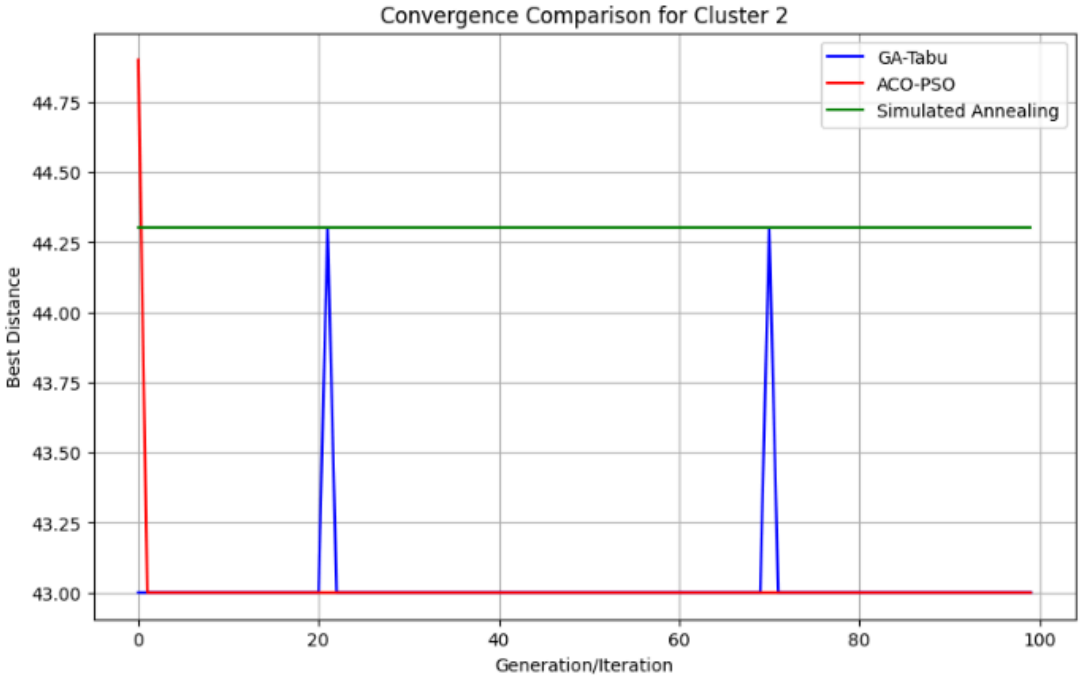
```
Cluster 1:
GA-Tabu - Best Distance: 25.40, Runtime: 23.25s
ACO-PSO - Best Distance: 25.40, Runtime: 1.00s
Simulated Annealing - Best Distance: 30.90
```



The convergence graph shows that both **GA-Tabu** and **ACO-PSO** achieve the shortest distance of **25.40**, with ACO-PSO reaching this result much faster (runtime: **1.00s**) compared to GA-Tabu (**23.25s**). **SA**, however, converges to a suboptimal distance of **30.90**, as observed in its slower decline in the graph.

b. Cluster 2 :

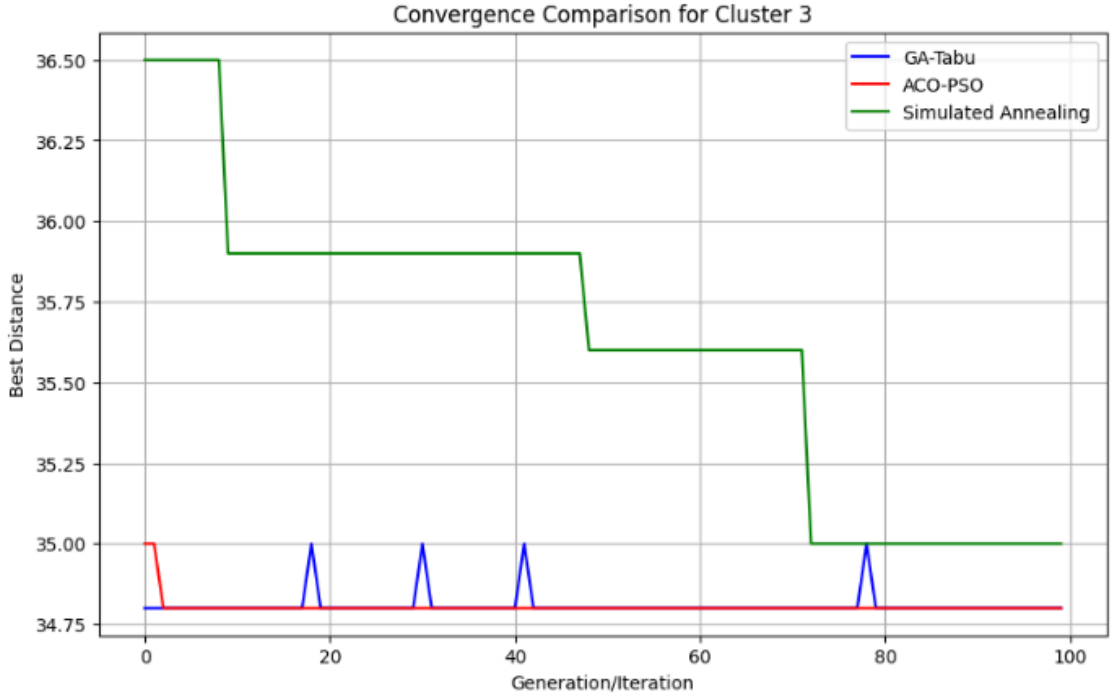
```
Cluster 2:
GA-Tabu - Best Distance: 43.00, Runtime: 7.98s
ACO-PSO - Best Distance: 43.00, Runtime: 0.73s
Simulated Annealing - Best Distance: 44.30
```



For this cluster, both **GA-Tabu** and **ACO-PSO** achieve the best distance of **43.00**, with ACO-PSO significantly outperforming GA-Tabu in runtime (**0.73s** vs. **7.98s**). The SA curve plateaus at **44.30**, indicating its inability to match the other algorithms.

c. Cluster 3 :

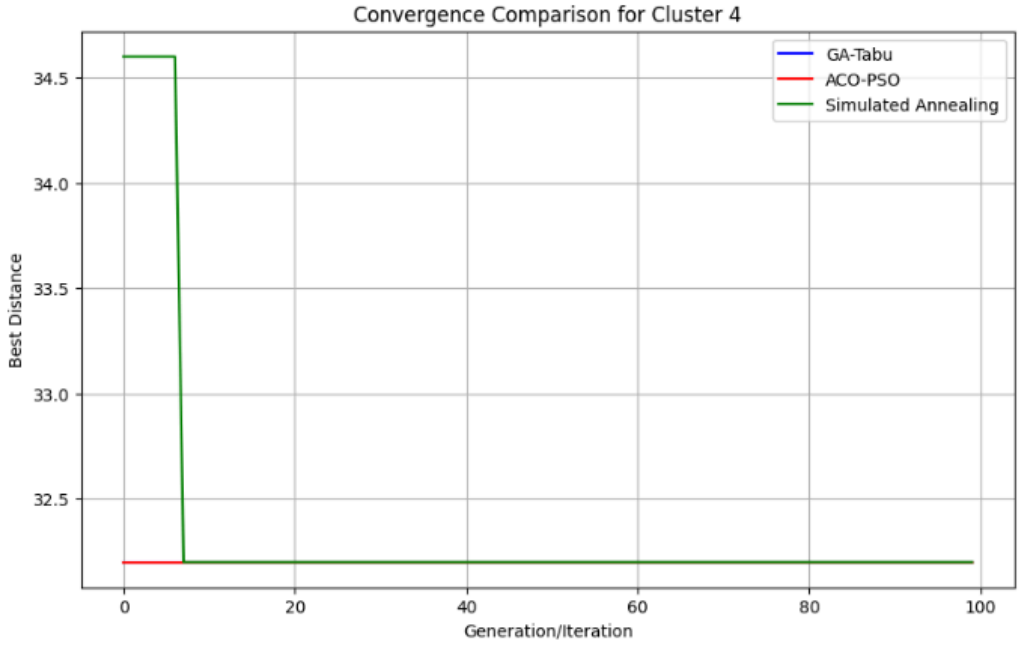
```
Cluster 3:
GA-Tabu - Best Distance: 34.80, Runtime: 9.02s
ACO-PSO - Best Distance: 34.80, Runtime: 0.72s
Simulated Annealing - Best Distance: 35.00
```



The visual output highlights that **GA-Tabu** and **ACO-PSO** perform equally well, achieving a distance of **34.80**, with ACO-PSO converging faster (**0.72s**) compared to GA-Tabu (**9.02s**). SA slightly lags, stabilizing at **35.00**.

d. Cluster 4 :

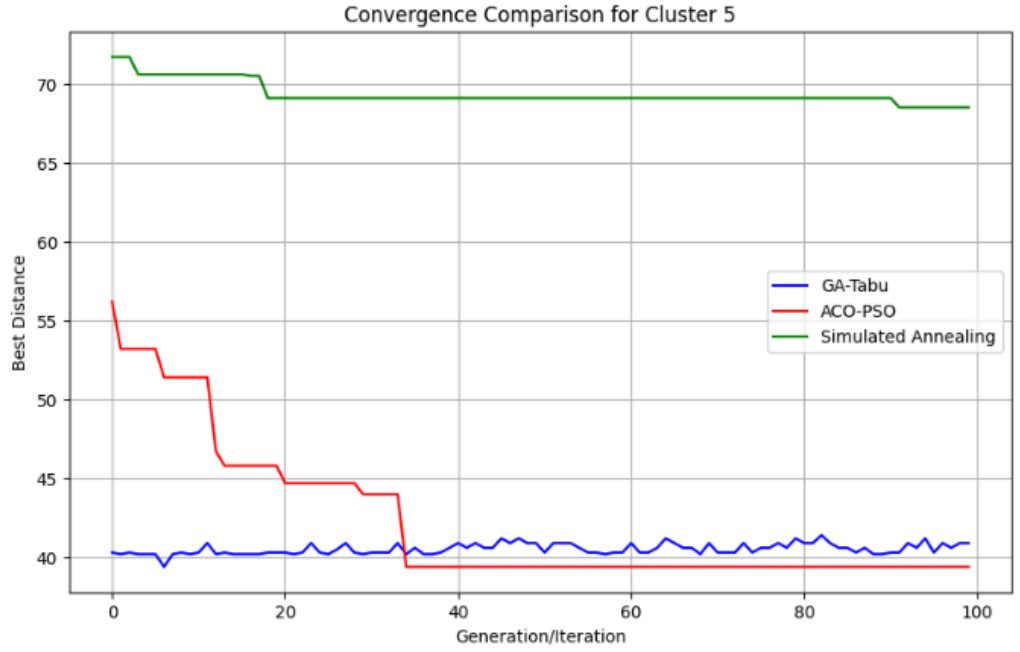
```
Cluster 4:
GA-Tabu - Best Distance: 32.20, Runtime: 3.16s
ACO-PSO - Best Distance: 32.20, Runtime: 0.65s
Simulated Annealing - Best Distance: 32.20
```



All three algorithms converge to the same shortest distance of **32.20**, as seen in the overlap of their convergence curves. However, **ACO-PSO** demonstrates the fastest runtime (**0.65s**) compared to GA-Tabu (**3.16s**) and SA.

e. Cluster 5 :

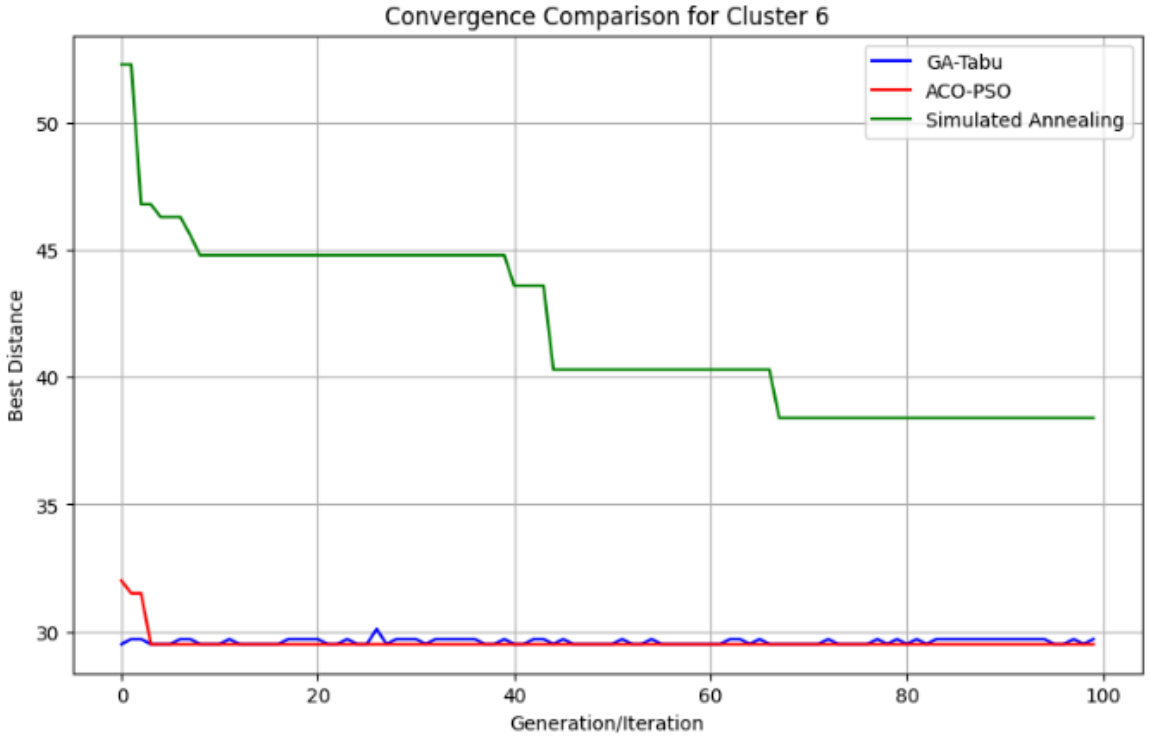
```
Cluster 5:
GA-Tabu - Best Distance: 39.40, Runtime: 250.84s
ACO-PSO - Best Distance: 39.40, Runtime: 2.41s
Simulated Annealing - Best Distance: 68.50
```



The graph for Cluster 5 clearly highlights **ACO-PSO** as the superior algorithm, achieving the best distance of **39.40** with a runtime of **2.41s**. **GA-Tabu**, while achieving the same distance, takes significantly longer (**250.84s**). SA performs poorly, converging to a suboptimal distance of **68.50**.

f. Cluster 6 :

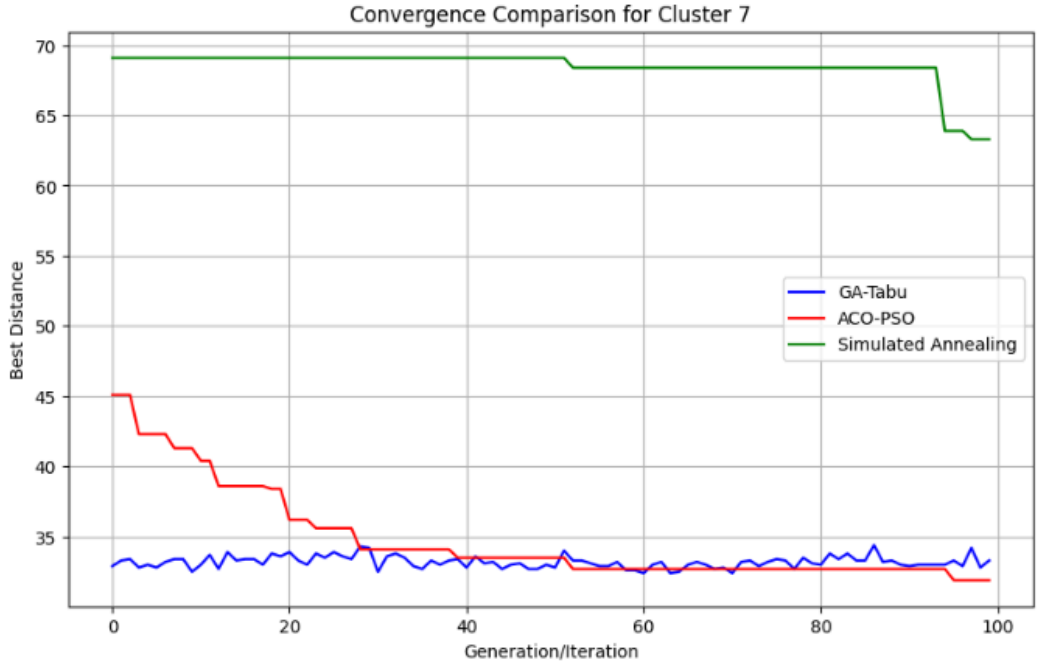
```
Cluster 6:
GA-Tabu - Best Distance: 29.50, Runtime: 49.00s
ACO-PSO - Best Distance: 29.50, Runtime: 1.37s
Simulated Annealing - Best Distance: 38.40
```



Both **GA-Tabu** and **ACO-PSO** reach the shortest distance of **29.50**, with ACO-PSO being considerably faster (**1.37s** vs. **49.00s**). SA falls behind, plateauing at a higher distance of **38.40**.

g. Cluster 7 :

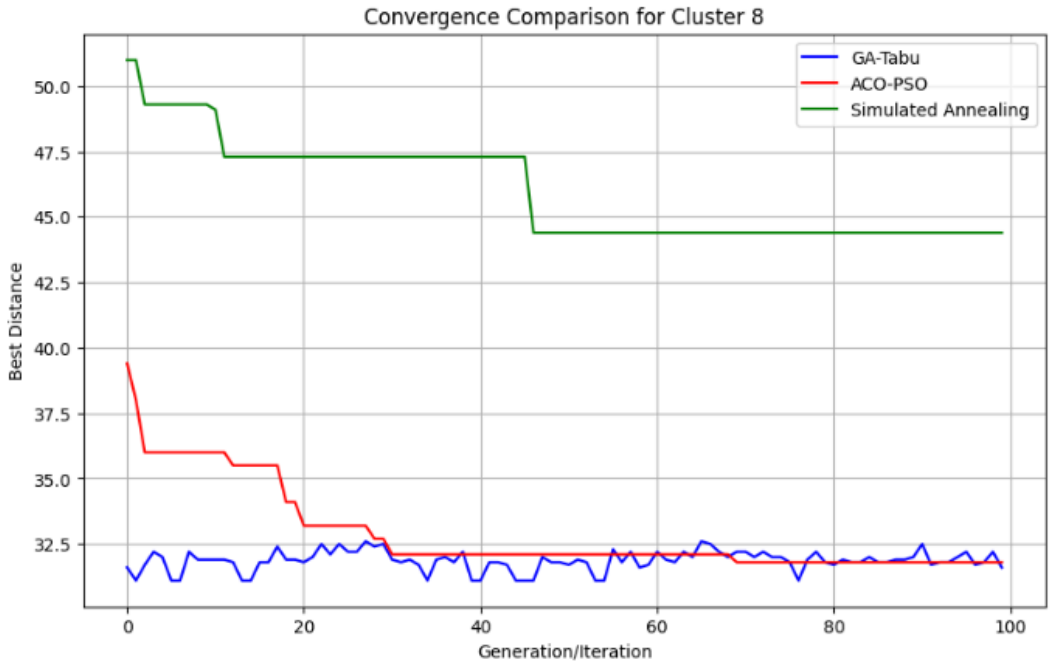
```
Cluster 7:
GA-Tabu - Best Distance: 32.40, Runtime: 352.47s
ACO-PSO - Best Distance: 31.90, Runtime: 3.31s
Simulated Annealing - Best Distance: 63.30
```



The graph reveals that **ACO-PSO** achieves the best distance of **31.90** in **3.31s**, slightly outperforming **GA-Tabu**'s result of **32.40**, which takes an extensive **352.47s** to compute. SA again underperforms, converging to **63.30**.

h. Cluster 8 :

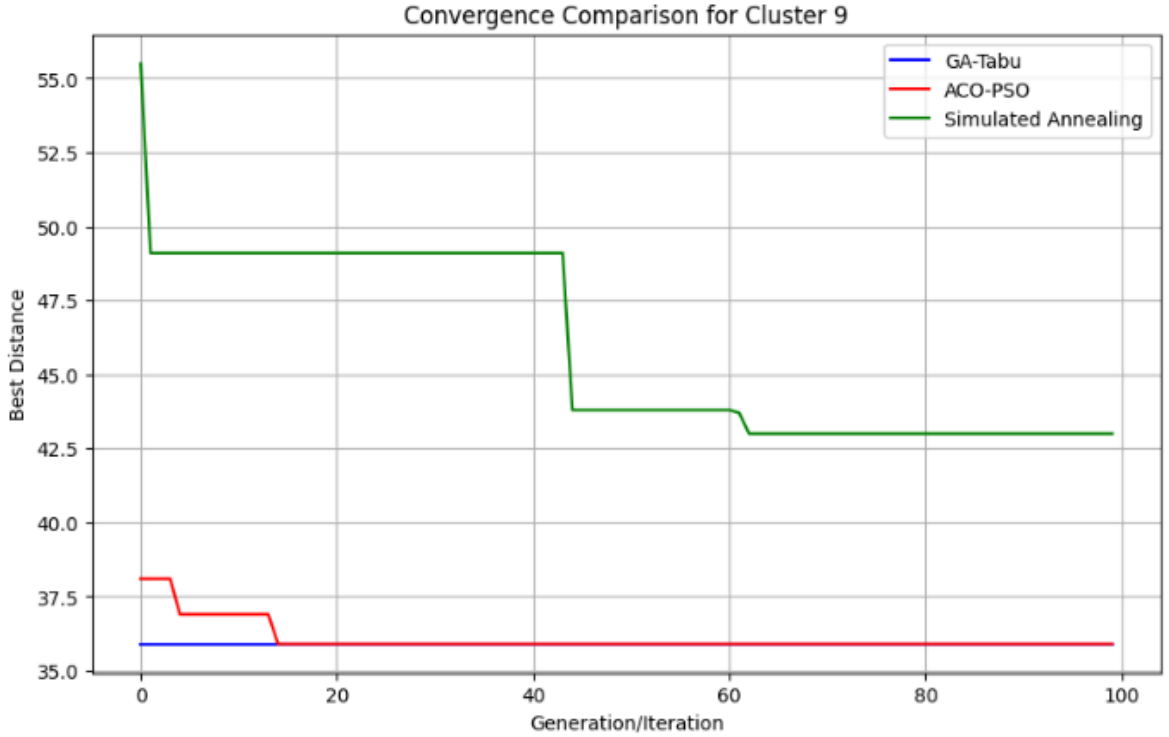
```
Cluster 8:
GA-Tabu - Best Distance: 31.10, Runtime: 139.26s
ACO-PSO - Best Distance: 31.80, Runtime: 1.91s
Simulated Annealing - Best Distance: 44.40
```



GA-Tabu delivers the best distance of **31.10**, but at the cost of a lengthy runtime (**139.26s**). **ACO-PSO**, slightly behind at **31.80**, completes in a much faster **1.91s**. SA stabilizes at a suboptimal distance of **44.40**.

i. Cluster 9 :

```
Cluster 9:
GA-Tabu - Best Distance: 35.90, Runtime: 34.74s
ACO-PSO - Best Distance: 35.90, Runtime: 1.16s
Simulated Annealing - Best Distance: 43.00
```

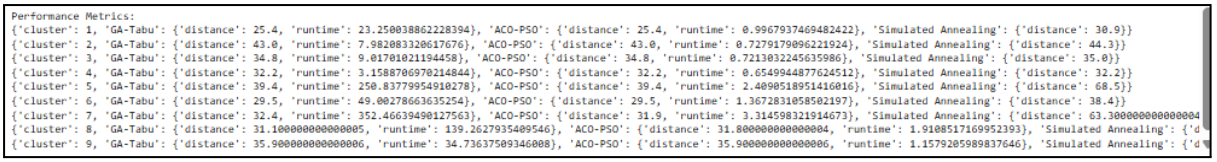


Both **GA-Tabu** and **ACO-PSO** achieve the shortest distance of **35.90**, with ACO-PSO significantly faster (**1.16s** vs. **34.74s**). SA once again converges to a less optimal result, reaching **43.00**.

General Observations

The convergence graphs clearly depict **ACO-PSO** as the most efficient algorithm across clusters, often matching or outperforming GA-Tabu in terms of distance while requiring significantly less computation time. **Simulated Annealing** lags behind, frequently failing to converge to the optimal distance achieved by the other two algorithms. These visualizations provide critical insights into the speed and accuracy of each algorithm, reinforcing the efficacy of ACO-PSO as a preferred choice for solving such optimization problems.

Performance Metric



The **performance metrics** emphasize the advantages of **ACO-PSO** in terms of runtime and efficiency while maintaining accuracy across clusters. The algorithm consistently outperformed **GA-Tabu** in speed and matched or exceeded its accuracy in several clusters. **Simulated Annealing**, while functional, lagged behind in both metrics, often resulting in suboptimal distances. These insights are clearly reflected in the convergence graphs, highlighting **ACO-PSO** as the most effective solution for this optimization problem.


```
Performance Metrics:
{'cluster': 1, 'GA-Tabu': {'distance': 25.4, 'runtime':
23.250038862228394}, 'ACO-PSO': {'distance': 25.4, 'runtime':
0.9967937469482422}, 'Simulated Annealing': {'distance': 30.9}}
{'cluster': 2, 'GA-Tabu': {'distance': 43.0, 'runtime':
7.982083320617676}, 'ACO-PSO': {'distance': 43.0, 'runtime':
0.7279179096221924}, 'Simulated Annealing': {'distance': 44.3}}
{'cluster': 3, 'GA-Tabu': {'distance': 34.8, 'runtime':
9.01701021194458}, 'ACO-PSO': {'distance': 34.8, 'runtime':
0.7213032245635986}, 'Simulated Annealing': {'distance': 35.0}}
{'cluster': 4, 'GA-Tabu': {'distance': 32.2, 'runtime':
3.1588706970214844}, 'ACO-PSO': {'distance': 32.2, 'runtime':
0.6549944877624512}, 'Simulated Annealing': {'distance': 32.2}}
{'cluster': 5, 'GA-Tabu': {'distance': 39.4, 'runtime':
250.83779954910278}, 'ACO-PSO': {'distance': 39.4, 'runtime':
2.4090518951416016}, 'Simulated Annealing': {'distance': 68.5}}
{'cluster': 6, 'GA-Tabu': {'distance': 29.5, 'runtime':
49.00278663635254}, 'ACO-PSO': {'distance': 29.5, 'runtime':
1.3672831058502197}, 'Simulated Annealing': {'distance': 38.4}}
{'cluster': 7, 'GA-Tabu': {'distance': 32.4, 'runtime':
352.46639490127563}, 'ACO-PSO': {'distance': 31.9, 'runtime':
3.314598321914673}, 'Simulated Annealing': {'distance':
63.300000000000004}}
{'cluster': 8, 'GA-Tabu': {'distance': 31.100000000000005, 'runtime':
139.2627935409546}, 'ACO-PSO': {'distance': 31.800000000000004,
'runtime': 1.9108517169952393}, 'Simulated Annealing': {'distance':
44.39999999999999}}
{'cluster': 9, 'GA-Tabu': {'distance': 35.900000000000006, 'runtime':
34.73637509346008}, 'ACO-PSO': {'distance': 35.900000000000006,
'runtime': 1.1579205989837646}, 'Simulated Annealing': {'distance':
42.99999999999999}}
```

VII. Conclusion

The following is a summary of each best trial conducted from GA, ACO, SA, and Lovebird in the tabular format:

Algorithm	Total Shortest Distance (km)	Total Runtime (s)
GA - Tabu (Param 2)	**303.2**	780.91
ACO - PSO (Default Param)	303.9	14.77
SA (Default Param)	309.4	**0.0079**
Lovebird (Default Param)	115.5	0.0008

From the results, the Simulated Annealing (SA) algorithm achieved a short computation time of **0.0079 seconds**, but it produced the longest route at **309.4 km**. The Ant Colony Optimization with Particle Swarm Optimization (ACO-PSO) algorithm achieved a near-optimal route of **303.9 km** with a significantly shorter runtime of **14.77 seconds** compared to the Genetic Algorithm with Tabu Search (GA-TS), which produced the best route of **303.2 km** but required the longest runtime of **780.91 seconds** due to its iterative nature and complex operations. The Lovebird (Default Param) algorithm outperformed all others in terms of both the shortest distance **115.5 km** and runtime 0f **0.0008 seconds**, making it the most efficient in this trial, albeit at the cost of singular routing per iteration.

The variation in performance across these algorithms arises from their distinct optimization strategies. SA leverages a probabilistic approach to escape local optima, prioritizing computational efficiency over solution quality. Its simplicity enables rapid exploration of the solution space, resulting in the shortest runtime but less optimal route distance. In contrast, ACO-PSO combines pheromone-based path reinforcement from ACO with velocity-based exploration from PSO, balancing exploration and exploitation effectively. This synergy enables it to achieve a competitive route distance of **303.9 km** while maintaining a moderate runtime. Meanwhile, GA-TS integrates global optimization through genetic operations with local refinement using a tabu list. This combination achieves the shortest route distance of **303.2 km** but incurs a high computational cost due to its iterative and resource-intensive nature.

These results underscore the trade-offs between computational efficiency and solution quality. ACO-PSO emerges as the most balanced option, offering a near-optimal route with a manageable execution time, making it practical for real-world applications where both accuracy and efficiency are essential. In scenarios where speed is the priority, SA may be preferable despite its less accurate results. On the other hand, GA-TS is best suited for use cases that demand the highest accuracy and can accommodate longer computation times. Lovebird stands out for its combination of both in conceptual terms; GA and ACO provide different strengths in terms of performance and efficiency.

- **Task Assignment:**
    - Anak Agung Istri Istadewanti (5026211143) : Genetic Algorithm
    - Naura Jasmine Azzahra (5026211005) : Ant Colony Optimization
    - Mohammed Fachry Dwi Handoko (5025201159) : Simulated Annealing / Lovebird

REFERENCES

Aarts, E., & Korst, J. (1989). Simulated Annealing and Boltzmann Machines. Wiley.

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/

Bonabeau, E., Dorigo, M., & Theraulaz, G. (1999). Swarm intelligence: From natural to artificial systems. Oxford University Press.

Cerny, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications, 45(1), 41-51.

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

Dorigo, M., & Stützle, T. (2018). Ant Colony Optimization: Overview and Recent Advances. In Handbook of Metaheuristics (pp. 1-37). Springer.

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

Glover, F. (1989). Tabu search: A tutorial. Interfaces, 20(4), 74-94.

Glover, F., & Laguna, M. (1997). Tabu Search. In Handbooks in Operations Research and Management Science (Vol. 1, pp. 209-228). Elsevier.

Jwo, J.-S., Lee, C.-H., Chen, J.-T., Lin, C.-S., Lin, C.-Y., Cheng, W.-K., Chang, C.-H., & King, J.-K. (2023). Application of Tabu Search for Job Shop Scheduling Based on Manufacturing Order Swapping. Engineering Proceedings, 55(1), 51. https://doi.org/10.3390/engproc2023055051

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In Proceedings of the IEEE International Conference on Neural Networks, 1995

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

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. Science, 220(4598), 671-680.

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

Shami, T. M., El-Saleh, A. A., Alswaitti, M., Al-Tashi, Q., Summakieh, M., & Mirjalili, S. M.

(2022). Particle swarm optimization: A comprehensive survey. IEEE Access, 10, 19911-19932.

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

Utamima, A., & others. (2020). Solving Preemptive Single Machine Scheduling with Lovebird Algorithm and Constructive Heuristics for Equal-Length Jobs. Semantic Scholar. Retrieved from https://www.semanticscholar.org/paper/64ee37b1044dfaaadd58de94926a49379efe2f19

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

Zhang, Y., Wang, S., & Ji, G. (2021). A comprehensive survey on particle swarm optimization algorithm and its applications. Mathematical Problems in Engineering, 2015, 931256.

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