FINAL PROJECT SOFT COMPUTING

KELOMPOK 2

ANGGOTA KELOMPOK:

ARVIN AZMI SAVA	(5026211097)
NIKOLAUS VICO CRISTIANTO	(5026211107)
I GUSTI AGUNG JAWA HISWARA	(5026211122)

KELAS
KOMPUTASI LUNAK (A)

DEPARTEMEN SISTEM INFORMASI
FAKULTAS TEKNOLOGI ELEKTRO DAN INFORMATIKA CERDAS
INSTITUT TEKNOLOGI SEPULUH NOPEMBER

SEMESTER GENAP 2024

Daftar Isi

Daftar Isi 2
Latar Belakang 2
Literature Review 3
Tujuan Project & Perbedaan Penelitian Sebelumnya 4
Deskripsi Permasalahan 5
Deskripsi Algoritma yang Digunakan 8
    A. Algoritma Genetika 8
    B. Algoritma Particle Swarm Optimization 9
    C. Algoritma Ant Colony Optimization 10
Hasil dan Perbandingan 12
    A. Hasil Algoritma Genetika 12
    B. Hasil Algoritma Particle Swarm Optimization 13
    C. Hasil Algoritma Ant Colony Optimization 15
Kesimpulan 18
Daftar Pustaka 19
Lampiran 19
    source code Clustering Lloyd 19
    source code Algoritma Genetika 23
    source code Particle Swarm Optimization 31
    source code Ant Colony Optimization 39

Latar Belakang

Pertanian merupakan sektor vital yang memiliki peran penting dalam memenuhi kebutuhan pangan dunia. Seiring dengan pertumbuhan populasi global, tantangan dalam sektor pertanian juga semakin meningkat, termasuk kebutuhan untuk meningkatkan produktivitas, dan efisiensi. Salah satu pendekatan yang sedang dikembangkan untuk mengatasi tantangan ini adalah penggunaan teknologi otomasi kendaraan otonom pertanian, seperti penggunaan drone dan traktor otomatis.

Kendaraan pertanian otonom memainkan peran penting dalam meningkatkan efisiensi, operasional dan mengurangi biaya tenaga kerja (Conesa-Muñoz et al., 2016). Kendaraan tersebut dapat digunakan untuk berbagai tugas seperti penyebaran pestisida, pemupukan, pemetaan lahan, dan panen. Salah satu masalah utama dalam operasional kendaraan ini adalah penentuan rute yang optimal. Penentuan rute optimal sangat penting karena dapat mengurangi waktu operasional, menghemat bahan bakar atau energi, dan meminimalkan keausan kendaraan.

Masalah penentuan rute dalam konteks pertanian sering disebut sebagai Agriculture Vehicle Routing Problem (AVRP). Masalah ini melibatkan penentuan rute terbaik bagi kendaraan pertanian untuk menyelesaikan tugas tertentu di ladang atau lahan pertanian. Rute yang optimal harus mempertimbangkan berbagai faktor seperti jarak tempuh, waktu operasional, energi yang dikonsumsi, dan jumlah perputaran kendaraan. Dalam aplikasi seperti penyemprotan pestisida oleh drone, perputaran yang terlalu sering dapat meningkatkan konsumsi energi dan waktu, sehingga mengurangi efisiensi keseluruhan.

Solusi untuk masalah AVRP tidaklah sederhana karena melibatkan banyak variabel dan kendala. Pendekatan tradisional seringkali tidak efektif karena kompleksitas dan dinamika lingkungan pertanian. Oleh karena itu, metode optimasi yang lebih canggih seperti algoritma genetika, Particle Swarm Optimization (PSO), dan teknik optimasi lainnya digunakan untuk menemukan solusi yang lebih efisien. Metode-metode ini mampu mengeksplorasi ruang solusi yang luas dan menemukan rute yang mendekati optimal dengan mempertimbangkan berbagai faktor yang mempengaruhi operasional kendaraan pertanian. Dengan mengoptimalkan rute kendaraan pertanian, petani dapat mencapai efisiensi yang lebih tinggi, mengurangi biaya operasional, dan meningkatkan produktivitas. Selain itu, solusi yang efisien juga berkontribusi pada keberlanjutan pertanian dengan mengurangi jejak karbon dan penggunaan sumber daya. Oleh karena itu, penelitian dan pengembangan dalam bidang AVRP sangat penting untuk masa depan pertanian yang lebih efisien dan berkelanjutan

Literature Review

Penelitian yang dilakukan (Xu et al., 2024) dengan judul “Collaborative orchard pesticide spraying routing problem with multi-vehicles supported multi-UAVs” bertujuan untuk mengoptimalkan rute kendaraan UAV (Unmanned Aerial Vehicles) dalam penyemprotan pestisida di kebun. UAV adalah pesawat tanpa awak yang digunakan untuk berbagai tugas dalam bidang pertanian. Metode yang diterapkan mencakup penggunaan algoritma Lin-Kernighan-Helsgaun (LKH) dan simulated annealing (SA). Masalah perencanaan rute UAV diformulasikan sebagai masalah Traveling Salesman Problem (TSP) untuk mendapatkan rute awal yang kemudian dioptimalkan menggunakan kombinasi algoritma LKH dan SA. Pada penelitian ini, penulis menyoroti pentingnya distribusi terpusat titik operasi UAV untuk mengurangi panjang jalur operasi dan meningkatkan efisiensi penyemprotan pestisida.

Langkah awal dalam penelitian ini yaitu melakukan clustering terlebih dahulu menggunakan algoritma K-means untuk mengelompokkan titik operasi berdasarkan jumlah UAV yang digunakan. setelah dilakukan clustering, jalur awal UAV dalam setiap cluster direncanakan menggunakan algoritma convex hull. Algoritma LKH memainkan peran penting dalam mengoptimalkan jalur awal yang telah dibentuk oleh metode convex hull dengan memberikan solusi yang mendekati optimal. Sementara itu, algoritma SA digunakan untuk memperbaiki solusi yang ada dengan melakukan pencarian global yang lebih luas untuk menghindari terjebak pada solusi lokal. Hasil awal ini kemudian diuji kelayakannya untuk memastikan jalur kendaraan juga feasible. Dengan cara ini, diperoleh solusi awal untuk rute UAV yang memungkinkan optimalisasi lebih lanjut.

Penelitian ini menekankan pentingnya melakukan clustering sebelum optimasi. Clustering membantu dalam pembagian yang lebih terpusat dan seimbang dari titik operasi UAV. Hal ini penting dilakukan untuk meningkatkan efisiensi dalam aplikasi penyemprotan pestisida, mengurangi konsumsi energi, dan mengoptimalkan penggunaan sumber daya. Kombinasi clustering, LKH, dan SA dalam optimasi jalur UAV dapat secara signifikan meningkatkan efisiensi operasional UAV dalam aplikasi penyemprotan pestisida di kebun sehingga di masa depan dapat memberikan solusi yang lebih berkelanjutan dan hemat energi.

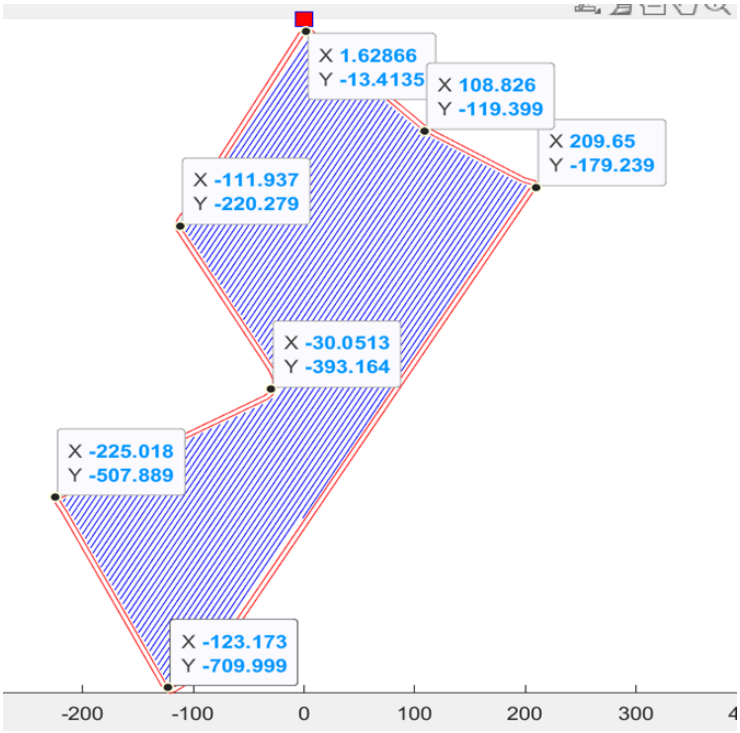
Tujuan Project & Perbedaan Penelitian Sebelumnya

Tujuan dari tugas ini adalah untuk menentukan rute yang optimal bagi kendaraan pertanian otomatis (drone) untuk mendistribusikan pestisida pada lahan pertanian tersebut. Rute yang optimal pada permasalahan ini diukur melalui objective function yang mengukur total jarak tempuh drone dan juga total perputaran drone. Perbedaan tugas final project ini dengan penelitian Xu et al. (2024) adalah pada objective function yang digunakan. Pada penelitian Xu et al. (2024), objective function yang digunakan adalah meminimalkan waktu operasi maksimum UAV untuk meningkatkan efisiensi operasional. Sementara itu, pada tugas final project ini, objective function yang digunakan meminimalkan total jarak tempuh dan total perputaran drone.

Perbedaan lainnya terdapat pada pendekatan yang digunakan untuk menyelesaikan permasalahan. Penelitian Xu et al. (2024) menggunakan kombinasi algoritma K-means dan convex hull untuk menghasilkan cluster awal. Kemudian, untuk optimasi rute UAV pada tiap clusternya menggunakan kombinasi algoritma Lin-Kernighan-Helsgaun (LKH) dan simulated annealing (SA). Sementara itu, pada tugas final project ini untuk multi-level clustering menggunakan pendekatan kombinasi algoritma lloyd dan k-means untuk melakukan clustering lahan pertanian. Kemudian, untuk optimalisasi rute pada tiap cluster dilakukan secara terpisah dengan menggunakan 3 skenario algoritma yang berbeda yaitu algoritma genetika, particle swarm optimization, dan ant colony optimization.

Deskripsi Permasalahan

Pada tugas final project ini, diberikan sebuah dataset koordinat-koordinat garis batas sebuah petak lahan pertanian seperti ditunjukkan pada gambar 1 di bawah ini.

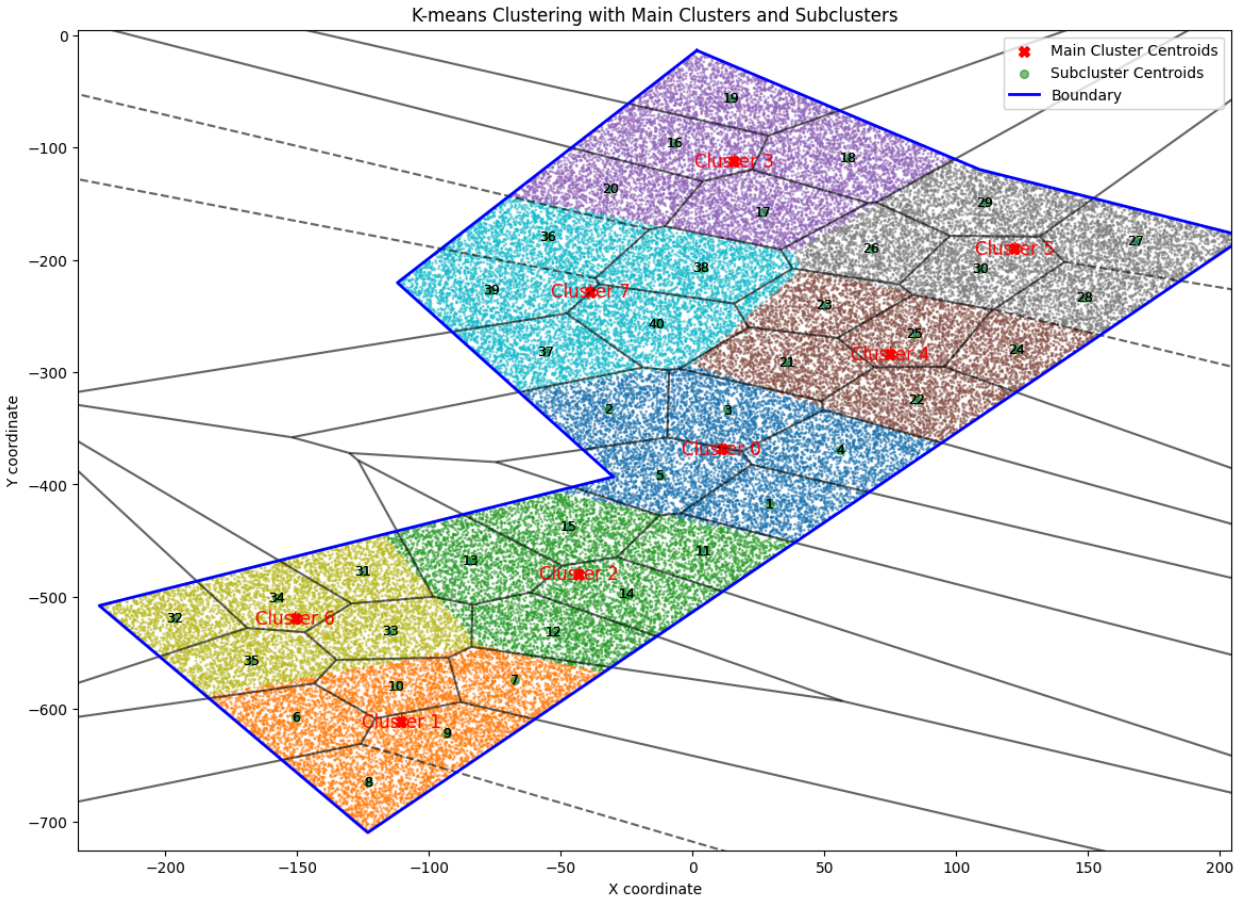


Gambar 1 merupakan plot area lahan berdasarkan batas-batas koordinat

Untuk menentukan rute optimal bagi kendaraan pertanian otomatis seperti drone, salah satu pendekatan yang efektif adalah dengan melakukan clustering terlebih dahulu. Dalam metode ini, lahan pertanian dibagi menjadi beberapa cluster utama berdasarkan koordinat garis batas yang ada. Pada tugas ini, kita akan membagi petak lahan menjadi 8 cluster utama. Kemudian, setiap cluster utama akan dilakukan clustering lagi menjadi 5 sub-cluster yang lebih kecil dan dilakukan optimalisasi rute secara terpisah untuk masing-masing cluster. Optimalisasi rute di setiap cluster bertujuan untuk mengurangi total jarak tempuh dan perputaran drone dalam skala yang lebih kecil dan terfokus.

Setelah rute optimal untuk masing-masing cluster berhasil ditentukan, langkah selanjutnya adalah menggabungkan semua rute cluster tersebut menjadi satu rute keseluruhan. Proses penggabungan ini dilakukan dengan memperhatikan transisi antar cluster untuk

memastikan efisiensi total tetap terjaga. Dengan demikian, metode ini dapat menghasilkan rute distribusi pestisida yang lebih efisien, menghemat sumber daya, dan meningkatkan efektivitas operasional kendaraan pertanian otomatis. Pada tugas ini, kita memanfaatkan kombinasi algoritma lloyd dan k-means untuk melakukan clustering. Pada langkah pertama, digunakan algoritma k-means untuk mengelompokkan data menjadi beberapa cluster utama dan subcluster yang lebih kecil. Setelah itu, digunakan algoritma lloyd untuk mendapatkan titik pusat atau centroid dari subcluster yang diperbaharui secara berulang untuk mendapatkan distribusi yang optimal. Pada gambar 2 di bawah ini merupakan hasil clustering yang telah diterapkan



Gambar 2 hasil clustering menggunakan lloyd

Deskripsi Algoritma yang Digunakan

A. Algoritma Genetika

Algoritma Genetika merupakan metode optimasi yang terinspirasi oleh proses seleksi alam dalam teori evolusi biologis. Algoritma ini dikembangkan oleh John Holland pada tahun 1975 menggunakan mekanisme yang menyerupai genetika dan seleksi alam untuk menemukan solusi optimal dari suatu permasalahan. Adapun konsep yang digunakan pada algoritma genetika yaitu seleksi, crossover, mutasi, dan reproduksi untuk menghasilkan solusi yang lebih optimal. Dalam algoritma genetika, solusi potensial untuk masalah tertentu direpresentasikan sebagai individu dalam populasi.Setiap individu memiliki representasi kromosom yang menggambarkan solusi tersebut.

Tahap awal dari algoritma genetika diawali dengan inisialisasi populasi awal yang terdiri dari sejumlah individu yang dipilih secara acak. Kemudian, setiap individu akan dievaluasi berdasarkan nilai fitness-nya yang mengukur seberapa baik individu tersebut untuk menyelesaikan permasalahan yang diberikan. Individu dengan nilai fitness yang lebih tinggi memiliki peluang lebih besar untuk dipilih dan dikombinasikan dalam proses crossover yang menggabungkan sifat unggul kedua induk untuk menghasilkan individu baru. Setelah itu, akan diterapkan proses mutasi untuk memperkenalkan variasi tambahan pada individu baru. Proses seleksi, crossover, dan mutasi inti diulangi hingga mencapai kondisi terminasi, seperti jumlah generasi tertentu atau tercapainya nilai fitness yang diinginkan.

Dalam permasalahan pada tugas ini, algoritma genetika digunakan untuk menentukan rute optimal sebuah drone pada sebuah ladang pertanian untuk menemukan sebuah rute optimal yang mengunjungi semua subcluster dengan jarak total terpendek. Representasi kromosom dalam permasalahan ini adalah urutan subcluster yang harus dikunjungi oleh drone. Setiap individu dalam populasi mewakili satu solusi potensial berupa rute tertentu. Dalam hal ini, objective function yang digunakan untuk mengevaluasi setiap individu tidak hanya memperhatikan total jarak tempuh drone, tetapi juga total perputaran drone ketika berbelok, karena perputaran yang terlalu sering dapat meningkatkan konsumsi energi dan waktu. Proses seleksi memilih individu-individu dengan nilai objective function terendah untuk dilanjutkan ke tahap crossover, di mana dua rute yang dipilih digabungkan untuk menciptakan rute baru yang diharapkan lebih optimal. Proses mutasi mengubah sebagian urutan subcluster dalam rute untuk

mencegah konvergensi awal ke solusi lokal yang kurang optimal untuk memastikan drone dapat menyelesaikan tugasnya dengan efisien dan efektif.

Solusi optimum yang dihasilkan oleh algoritma genetika untuk penentuan rute drone dalam menyirami pestisida di ladang pertanian adalah rute yang memiliki nilai objective function paling rendah, yang menggabungkan total jarak tempuh dan total perputaran drone. Algoritma genetika mampu mengeksplorasi berbagai kombinasi rute dan menghindari jebakan solusi lokal melalui proses crossover dan mutasi yang memperkenalkan variasi. Dengan demikian, rute optimal yang dihasilkan tidak hanya meminimalkan jarak yang ditempuh oleh drone, tetapi juga mengurangi jumlah perputaran yang diperlukan, sehingga meningkatkan efisiensi operasional drone. Hal ini membuat algoritma genetika menjadi alat yang efektif dan efisien dalam menyelesaikan masalah optimasi rute drone untuk aplikasi pertanian dan masalah optimasi kompleks lainnya.

B. Algoritma Particle Swarm Optimization

Particle Swarm Optimization (PSO) adalah metode optimasi yang terinspirasi oleh perilaku sosial hewan, seperti kawanan burung atau ikan, dalam mencari makanan atau menghindari predator. PSO dikembangkan oleh James Kennedy dan Russell Eberhart pada tahun 1995 dan menggunakan mekanisme yang menyerupai pergerakan kelompok untuk menemukan solusi optimal dari suatu permasalahan. Konsep utama yang digunakan dalam PSO adalah partikel, kecepatan, dan posisi, dimana partikel-partikel bergerak dalam ruang solusi untuk menemukan posisi optimal. Setiap partikel dalam PSO mewakili solusi potensial dan memiliki kecepatan serta posisi yang diperbarui berdasarkan pengalaman pribadi dan pengalaman terbaik dari kawanan.

Tahap awal dari PSO diawali dengan inisialisasi populasi awal yang terdiri dari sejumlah partikel yang dipilih secara acak. Setiap partikel memiliki posisi dan kecepatan awal. Kemudian, setiap partikel dievaluasi berdasarkan nilai fitness-nya, yang mengukur seberapa baik partikel tersebut dalam menyelesaikan permasalahan yang diberikan. Partikel dengan nilai fitness yang lebih tinggi (atau lebih rendah tergantung pada tujuan minimisasi atau maksimisasi) akan mempengaruhi pergerakan partikel lainnya. Posisi dan kecepatan partikel diperbarui menggunakan persamaan yang mempertimbangkan posisi terbaik partikel tersebut (pbest) dan

posisi terbaik yang pernah dicapai oleh seluruh kawanan (gbest). Proses ini diulangi hingga mencapai kondisi terminasi, seperti jumlah iterasi tertentu atau tercapainya nilai fitness yang diinginkan.

Dalam permasalahan pada tugas ini, PSO digunakan untuk menentukan rute optimal sebuah drone pada sebuah ladang pertanian untuk menemukan rute optimal yang mengunjungi semua subcluster dengan jarak total terpendek. Representasi partikel dalam permasalahan ini adalah urutan subcluster yang harus dikunjungi oleh drone. Setiap partikel dalam populasi mewakili satu solusi potensial berupa rute tertentu. Dalam hal ini, objective function yang digunakan untuk mengevaluasi setiap partikel tidak hanya memperhatikan total jarak tempuh drone, tetapi juga total perputaran drone ketika berbelok, karena perputaran yang terlalu sering dapat meningkatkan konsumsi energi dan waktu. Proses pembaruan posisi dan kecepatan partikel mempertimbangkan nilai objective function, dimana partikel-partikel dengan nilai objective function yang lebih rendah akan mempengaruhi pergerakan partikel lainnya menuju solusi yang lebih baik.

Solusi optimum yang dihasilkan oleh PSO untuk penentuan rute drone dalam menyirami pestisida di ladang pertanian adalah rute yang memiliki nilai objective function paling rendah, yang menggabungkan total jarak tempuh dan total perputaran drone. PSO mampu mengeksplorasi berbagai kombinasi rute melalui pergerakan partikel yang dipandu oleh pengalaman pribadi dan kawanan, sehingga menghindari jebakan solusi lokal. Dengan demikian, rute optimal yang dihasilkan tidak hanya meminimalkan jarak yang ditempuh oleh drone, tetapi juga mengurangi jumlah perputaran yang diperlukan, sehingga meningkatkan efisiensi operasional drone. Hal ini membuat PSO menjadi alat yang efektif dan efisien dalam menyelesaikan masalah optimasi rute drone untuk aplikasi pertanian dan masalah optimasi kompleks lainnya.

C. Algoritma Ant Colony Optimization

Ant Colony Optimization (ACO) adalah metode optimasi yang terinspirasi oleh perilaku sosial semut dalam mencari makanan. ACO dikembangkan oleh Marco Dorigo pada awal 1990-an dan menggunakan prinsip-prinsip yang meniru cara semut berinteraksi satu sama lain untuk menemukan jalur terpendek dari sarang ke sumber makanan. Konsep utama yang

digunakan dalam ACO adalah feromon, semut buatan, dan graf solusi, dimana semut-semut buatan bergerak melalui graf solusi untuk menemukan jalur optimal. Setiap semut dalam ACO mewakili solusi potensial dan menandai jalur yang dilaluinya dengan feromon, yang kemudian mempengaruhi pergerakan semut-semut lainnya.

Tahap awal dari ACO diawali dengan inisialisasi populasi semut buatan yang ditempatkan secara acak pada titik-titik awal dalam graf solusi. Setiap semut kemudian melakukan perjalanan melalui graf solusi dengan memilih jalur berdasarkan probabilitas yang dipengaruhi oleh konsentrasi feromon dan jarak antar titik. Setelah setiap semut menyelesaikan perjalanannya, jalur yang ditemukan dievaluasi berdasarkan nilai fitness-nya, yang mengukur seberapa baik jalur tersebut dalam menyelesaikan permasalahan yang diberikan. Semut-semut yang menemukan jalur lebih baik akan menandai jalur tersebut dengan lebih banyak feromon, meningkatkan kemungkinan jalur tersebut dipilih oleh semut-semut berikutnya.

Posisi feromon pada setiap jalur diperbarui menggunakan persamaan yang mempertimbangkan penguapan feromon seiring waktu dan penambahan feromon baru berdasarkan kualitas jalur yang ditemukan. Proses ini diulangi hingga mencapai kondisi terminasi, seperti jumlah iterasi tertentu atau tercapainya nilai fitness yang diinginkan. Dalam permasalahan pada tugas ini, ACO digunakan untuk menentukan rute optimal sebuah drone pada sebuah ladang pertanian untuk menemukan rute optimal yang mengunjungi semua subcluster dengan jarak total terpendek. Representasi semut dalam permasalahan ini adalah urutan subcluster yang harus dikunjungi oleh drone. Setiap semut dalam populasi mewakili satu solusi potensial berupa rute tertentu. Dalam hal ini, fungsi objektif yang digunakan untuk mengevaluasi setiap semut tidak hanya memperhatikan total jarak tempuh drone, tetapi juga total perputaran drone ketika berbelok, karena perputaran yang terlalu sering dapat meningkatkan konsumsi energi dan waktu.

Proses pembaruan feromon mempertimbangkan nilai fungsi objektif, dimana jalur-jalur dengan nilai fungsi objektif yang lebih rendah akan mendapatkan lebih banyak feromon, yang pada gilirannya akan mempengaruhi pergerakan semut-semut lainnya menuju solusi yang lebih baik. Solusi optimum yang dihasilkan oleh ACO untuk penentuan rute drone dalam menyirami pestisida di ladang pertanian adalah rute yang memiliki nilai fungsi objektif paling rendah, yang

menggabungkan total jarak tempuh dan total perputaran drone. ACO mampu mengeksplorasi berbagai kombinasi rute melalui pergerakan semut yang dipandu oleh mekanisme feromon, sehingga menghindari jebakan solusi lokal. Dengan demikian, rute optimal yang dihasilkan tidak hanya meminimalkan jarak yang ditempuh oleh drone, tetapi juga mengurangi jumlah perputaran yang diperlukan, sehingga meningkatkan efisiensi operasional drone. Hal ini membuat ACO menjadi alat yang efektif dan efisien dalam menyelesaikan masalah optimasi rute drone untuk aplikasi pertanian dan masalah optimasi kompleks lainnya.

Hasil dan Perbandingan

Untuk setiap algoritma, akan didapatkan output berupa nilai *objective function*, *runtime*, dan rute optimal. Berikut merupakan hasil untuk setiap algoritma dan perbandingannya.

A. Hasil Algoritma Genetika

Pada algoritma genetika ini dilakukan lima kali percobaan. Adapun nilai parameter *crossover rate* dan *mutation rate* yang digunakan berturut-turut memiliki nilai 0,7 dan 0,01. Sementara itu, nilai parameter jumlah populasi dan generasi diubah-ubah pada setiap percobaan untuk menemukan hasil yang optimum. Pada tabel 1 di bawah ini merupakan beberapa solusi yang dihasilkan dari algoritma genetika.

Tabel 1 Hasil Algoritma Genetika

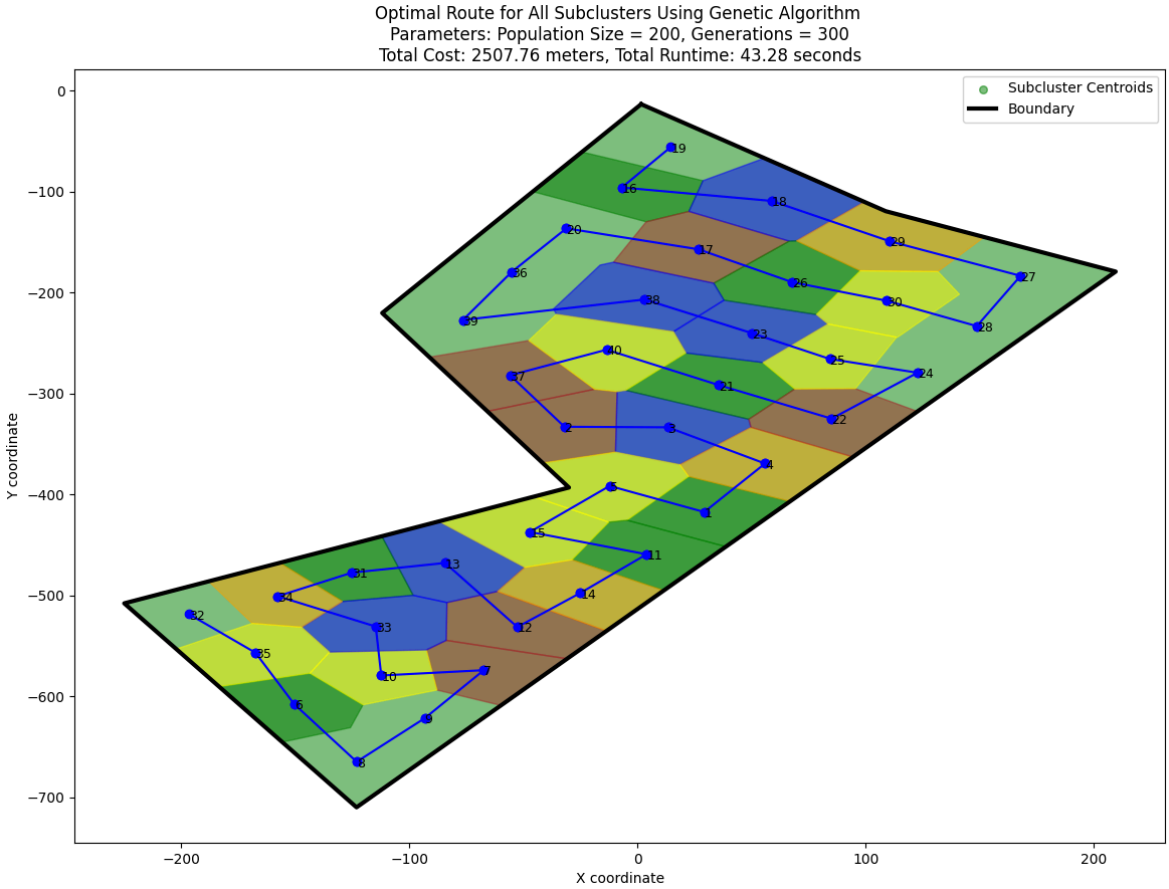
Run	Objective Function	Runtime	Parameter
1	2507,76 meter	43,28 seconds	Populasi 200, generasi 300
2	2518,37 meter	12,45 seconds	populasi 100, generasi 200
3	2669,45 meter	5,57 second	populasi 50, generasi 200
4	2591.84 meter	2,91 second	populasi 50, generasi 100
5	2634, 32 meter	1,72 second	populasi 50, generasi 50

Berdasarkan tabel X di atas, nilai parameter populasi dan generasi memiliki pengaruh terhadap *objective function* beserta *runtime* yang dihasilkan. Dapat disimpulkan bahwa semakin besar nilai parameter populasi dan generasi, maka *runtime* untuk mendapatkan solusinya juga akan semakin lama. Solusi terbaik dari lima percobaan yang telah dilakukan dihasilkan ketika nilai parameter populasi 200 dan generasi 300 dengan nilai *objective function* 2507,76 meter,

serta *runtime* 43,28 detik. Berikut merupakan rute optimal untuk solusi terbaik algoritma genetika.

19 16 18 29 27 28 30 26 17 20 36 39 38 23 25 24 22 21 40 37 2 3 4 1 5 15 11 14 12 13 31 34 33 10 7 9 8 6 35 32

Rute optimal yang dihasilkan kemudian divisualisasikan seperti pada gambar 3 di bawah ini



Gambar 3 Visualisasi jalur solusi terbaik Algoritma Genetika

B. Hasil Algoritma Particle Swarm Optimization

Pada percobaan algoritma *Particle Swarm Optimization* (PSO) dilakukan menggunakan 5 kali iterasi dengan 5 parameter yang berbeda. Parameter yang dibedakan pada setiap percobaan adalah jumlah partikel dan iterasi. Sementara itu, untuk parameter inertia weight, c1, dan c2 akan bernilai tetap dengan nilai berturut-turut 0.9, 1.5, dan 2. Pada tabel 2 di bawah ini merupakan beberapa solusi yang dihasilkan dari algoritma PSO.

Tabel 2 Hasil Algoritma Particle Swarm Optimization

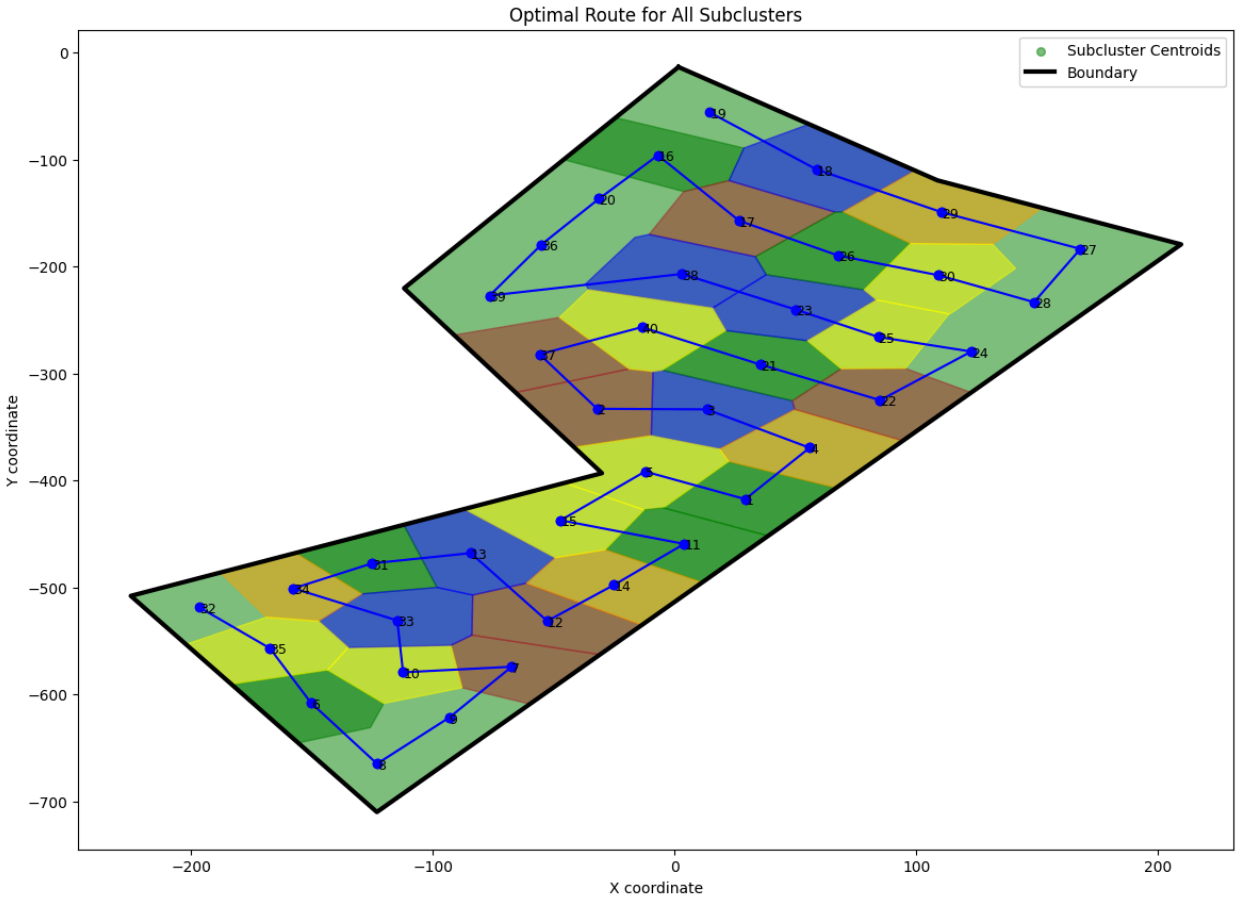
Run	Objective Function	Runtime	Parameter
1	2646.75 Meters	11.82 Seconds	particle = 30, iterations = 200, w = 0.9, c1 = 1.5, c2 = 2
2	2626.01 Meters	40.46 Seconds	particle = 35, iterations = 250, w = 0.9, c1 = 1.5, c2 = 2
3	2524.03 Meters	22.41 Seconds	particle = 40, iterations = 300, w = 0.9, c1 = 1.5, c2 = 2
4	2536.07 Meters	29.85 Seconds	particle = 45, iterations = 350, w = 0.9, c1 = 1.5, c2 = 2
5	2586.12 Meters	37.72 Seconds	particle = 50, iterations = 400, w = 0.9, c1 = 1.5, c2 = 2

Berdasarkan hasil uji diatas, dapat dilihat parameter yang optimal adalah ketika jumlah partikel adalah 40 dan iterasi berjumlah 300. Namun, anehnya yang semestinya dengan jumlah partikel banyak dan jumlah iterasi banyak, seharusnya cost akan semakin optimal tetapi nyatanya tidak seperti itu. Hal ini dikarenakan sudah bukan menjadi rahasia lagi bahwa algoritma PSO akan sangat tidak stabil dalam memecahkan masalah dan dapat dilihat pada problema ini.

Setelah mendapat cost terbaik, maka rute paling optimalnya adalah

19 18 29 27 28 30 26 17 16 20 36 39 38 23 25 24 22 21 40 37 2 3 4 1 5 15 11 14 12 13 31 34 33 10 7 9 8 6 35 32

Pada gambar 4 dibawah ini merupakan visualisasi rute terbaik



Gambar 4 Visualisasi jalur solusi terbaik PSO

C. Hasil Algoritma Ant Colony Optimization

Pada percobaan algoritma *Ant Colony Optimization* (ACO) akan menggunakan 5 kali iterasi dengan 5 parameter yang berbeda yaitu *num_ants* (jumlah semut), *num_iterations*(jumlah iterasi), *alpha* (nilai terbaik oleh maximizer), *beta* (nilai terbaik oleh minimizer), *evaporation rate* (tingkat penguapan). Pada tabel 3 di bawah ini merupakan beberapa solusi yang dihasilkan dari algoritma ACO.

Tabel 3 Hasil Algoritma *Ant Colony Optimization*

Run	*Objective Function*	*Runtime*	Parameter
1	2461.71	9.90	num_ants=100 , num_iterations=200, alpha=0.8, beta=3.0 , evaporation_rate=0.1
2	2461.71	5.55	num_ants=75 , num_iterations=150, alpha=0.8, beta=3.0 , evaporation_rate=0.1

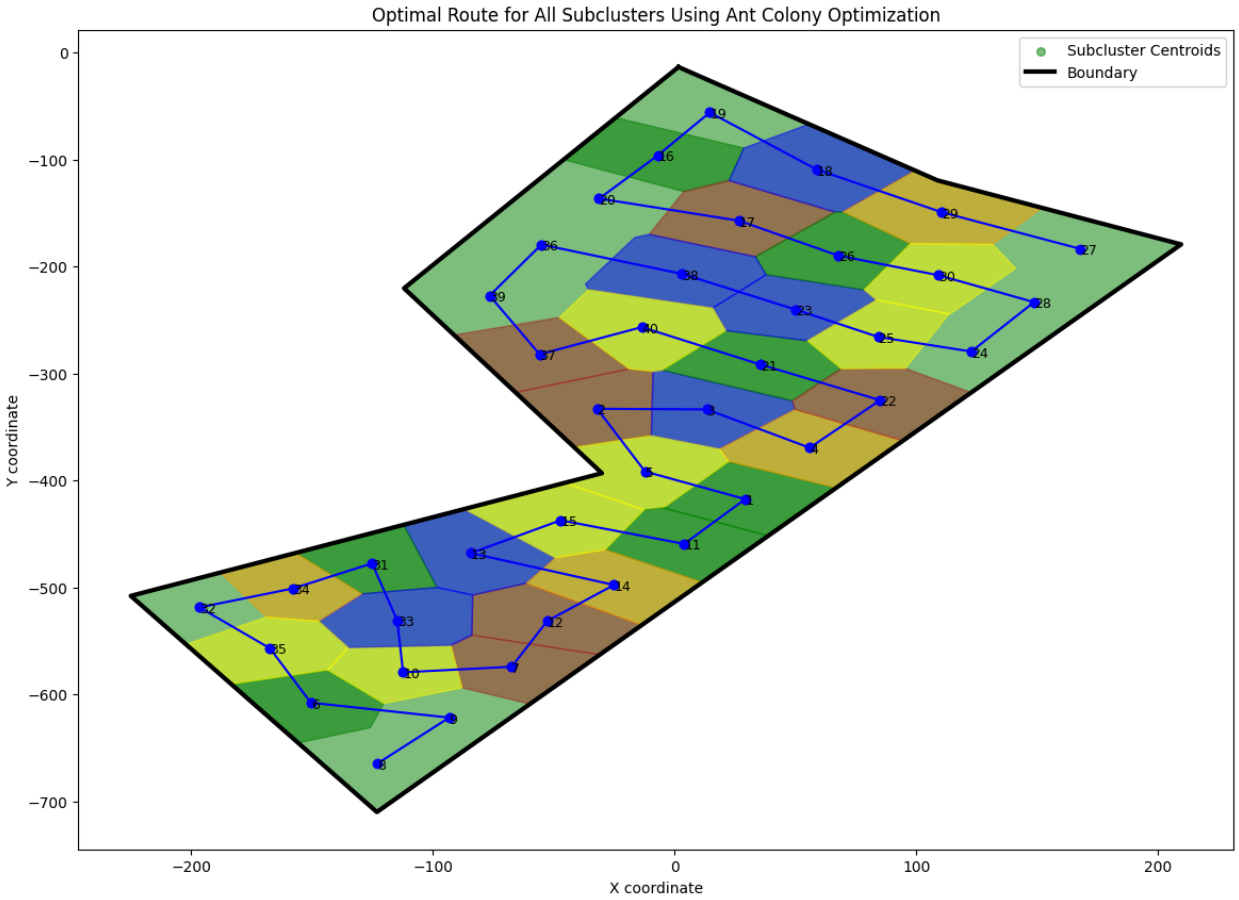
3	2461.71	2.48	num_ants=50 , num_iterations=100, alpha=0.8, beta=3.0 , evaporation_rate=0.1
4	2461.71	0.14	num_ants=20 , num_iterations=15, alpha=0.8, beta=3.0 , evaporation_rate=0.1
5	2518.24	0.09	num_ants=20 , num_iterations=10, alpha=0.8, beta=3.0 , evaporation_rate=0.1

Berdasarkan hasil uji diatas, dapat dilihat parameter yang optimal adalah ketika jumlah semut adalah 20, iterasi berjumlah 15, alpha bernilai 0.8, beta bernilai 3.0 dan evaporasi bernilai 0.1. Setelah mendapat cost terbaik, maka rute paling optimalnya adalah sebagai berikut:

```
27 29 18 19 16 20 17 26 30 28 24 25 23 38 36 39 37 40 21 22  4  3  2  5
  1 11 15 13 14 12  7 10 33 31 34 32 35  6  9  8
```

Dari algoritma ini dapat disimpulkan bahwa semakin sedikit jumlah iterasi maka tingkat efektivitas akan berkurang sehingga diperlukan kombinasi parameter yang tepat agar dapat menghasilkan *best practice.* Selain itu faktor lain seperti *chipset* atau *environment* atau *paltform*yang digunakan untuk menjalankan algoritma bepengaruh dalam *runtime* (waktu perhitungan). Hasil yang didaptkan ini di jalankan pada *Chipset Apple M3 ARM* yang mana sudah di optimalkan untuk kepentingan *machine learning* dan *environment Jupyternotebook* dengan *kernel: Anaconda*. Oleh karena itu hasil mungkin akan berbeda pada perangkat atau e*environment* yang menggunakan *chipset* lainnya meskipun dengan parameter dan algoritma yang sama

Pada gambar 5 dibawah ini merupakan visualisasi rute terbaik algoritma ACO



Gambar 5 Visualisasi jalur solusi terbaik ACO

D. Perbandingan Ketiga Algoritma

Berdasarkan percobaan dengan menggunakan ketiga algoritma tersebut, didapatkan hasil terbaik menggunakan algoritma *Ant Colony Optimization* dengan nilai *objective function* 2461,71 meter. Hasil tersebut didapatkan dengan parameter jumlah semut 20, jumlah iterasi 15, *alpha* 0,8, *beta* 3, *evaporation rate* 0.1. Adapun hasil lengkap *objective function* ketiga algoritma tersebut dapat dilihat pada tabel 4 di bawah ini:

Tabel 4 Perbandingan Nilai *Objective Function* ketiga algoritma

Run	Algoritma Genetika	*Particle Swarm Optimization*	*Ant Colony Optimization*
1	2507,76 meter	2646.75 meter	2461.71 meter
2	2518,37 meter	2626.01 meter	2461.71 meter
3	2669,45 meter	2524.03 meter	2461.71 meter

4	2591.84 meter	2536.07 meter	2461.71 meter
5	2634, 32 meter	2586.12 meter	2461.71 meter

Selain memiliki *objective function* yang unggul, algoritma *Ant Colony Optimization* juga menunjukkan kinerja *runtime* yang lebih cepat dibandingkan dua algoritma yang lainnya. Adapun runtime ketiga algoritma secara keseluruhan bisa dilihat pada tabel 5 di bawah ini:

Tabel 5 Perbandingan *Runtime* ketiga algoritma

Run	Algoritma Genetika	Particle Swarm Optimization	Ant Colony Optimization
1	43,28 seconds	11.82 seconds	9.90 seconds
2	12,45 seconds	40.46 seconds	5.55 seconds
3	5,57 seconds	22.41 seconds	2.48 seconds
4	2,91 seconds	29.85 seconds	0.14 seconds
5	1,72 seconds	37.72 seconds	0.09 seconds

Kesimpulan

Berdasarkan penerapan beberapa algoritma yang telah dilakukan, diperoleh solusi terbaik menggunakan algoritma Ant Colony Optimization dengan nilai objective function 2461,71 meter dan runtime 0,14 detik. Algoritma ini tidak hanya memberikan solusi dengan nilai objective function yang optimal, tetapi juga menunjukkan efisiensi runtime yang signifikan dibandingkan dua algoritma lainnya. Dengan hasil optimal tersebut, rute perjalanan drone untuk mendistribusikan pestisida pada lahan pertanian ini dapat dilakukan dengan seefisien mungkin sehingga meminimalisir penggunaan sumber daya seperti bahan bakar, waktu, dan tenaga kerja. Kedepannya, penelitian ini diharapkan dapat digunakan sebagai referensi untuk mengembangkan sistem optimasi yang lebih kompleks, dengan mempertimbangkan berbagai faktor seperti jenis tanaman, spesifikasi drone, kondisi lahan, dan kebutuhan spesifik pestisida untuk menghemat lebih banyak sumber daya yang diperlukan dalam pengelolaan pertanian otomatis pada berbagai skala lahan.

Daftar Pustaka

Conesa-Muñoz, J., Bengochea-Guevara, J. M., Andujar, D., & Ribeiro, A. (2016). Route planning for agricultural tasks: A general approach for fleets of autonomous vehicles in site-specific herbicide applications. *Computers and Electronics in Agriculture*, *127*, 204–220. https://doi.org/10.1016/j.compag.2016.06.012

Xu, J., Liu, C., Shao, J., Xue, Y., & Li, Y. (2024). Collaborative orchard pesticide spraying routing problem with multi-vehicles supported multi-UAVs. *Journal of Cleaner Production*, *458*, 142429. https://doi.org/10.1016/j.jclepro.2024.142429

Lampiran

source code Clustering Lloyd

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from shapely.geometry import Polygon, Point
from scipy.spatial import Voronoi, voronoi_plot_2d
import pandas as pd

# Tentukan koordinat dari sudut-sudut batas non-square boundary
x1, y1 = 1.62866, -13.4136
x2, y2 = 108.826, -119.399
x3, y3 = 209.65, -179.239
x4, y4 = -123.173, -709.999
x5, y5 = -225.018, -507.889
x6, y6 = -30.0513, -393.164
x7, y7 = -111.937, -220.279

# Buat objek Polygon yang mewakili batas non-square boundary
boundary = Polygon([(x1, y1), (x2, y2), (x3, y3), (x4, y4),(x5, y5),(x6,
y6), (x7, y7)])

# Hasilkan data sintetis untuk cluster utama
n_samples = 60000
```

```
n_main_clusters = 8  # Jumlah cluster utama
n_subclusters = 5  # Jumlah subcluster dalam setiap cluster utama
random_state = 42

points = []

while len(points) < n_samples:
    x = np.random.uniform(boundary.bounds[0], boundary.bounds[2])
    y = np.random.uniform(boundary.bounds[1], boundary.bounds[3])
    point = (x, y)
    if boundary.contains(Point(point)):
        points.append(point)

points = np.array(points)

# Lakukan K-means clustering pada cluster utama
kmeans_main = KMeans(n_clusters=n_main_clusters,
random_state=random_state, n_init=10)
kmeans_main.fit(points)
cluster_labels_main = kmeans_main.labels_

# Hasilkan data sintetis untuk subcluster dalam setiap cluster utama
subclusters_centroid = []
subclusters_label = []
subclusters_points = []
subclusters_centroid_sub_labels = []
subclusters_centroid_main_labels = []
main_labels = []
subcluster_number = []

counter = 1

for i in range(n_main_clusters):
    # Pilih titik-titik yang termasuk dalam cluster utama i
    main_cluster_points = points[cluster_labels_main == i]
    main_cluster_labels = np.repeat(i, len(main_cluster_points))

    # Lakukan K-means clustering pada titik-titik dari cluster utama i
    kmeans_sub = KMeans(n_clusters=n_subclusters,
random_state=random_state, n_init=10)
```

```
    kmeans_sub.fit(main_cluster_points)
    cluster_labels_sub = kmeans_sub.labels_
    centroids_sub = kmeans_sub.cluster_centers_
    centroid_sub_labels = kmeans_sub.predict(centroids_sub)
    centroid_main_labels = np.repeat(i, len(centroids_sub))

    # Tambahkan titik-titik, label, dan centroid subcluster ke dalam
dataset
    subclusters_points.extend(main_cluster_points)
    subclusters_label.extend(cluster_labels_sub)
    main_labels.extend(main_cluster_labels)
    subclusters_centroid.extend(centroids_sub)
    subclusters_centroid_sub_labels.extend(centroid_sub_labels)
    subclusters_centroid_main_labels.extend(centroid_main_labels)

    # Tambahkan penomoran untuk setiap subcluster
    subcluster_number.extend(range(counter, counter + n_subclusters))
    counter += n_subclusters

subclusters_label = np.array(subclusters_label)
subclusters_points = np.array(subclusters_points)
main_labels = np.array(main_labels)
subclusters_centroid = np.array(subclusters_centroid)
subclusters_centroid_sub_labels =
np.array(subclusters_centroid_sub_labels)
subclusters_centroid_main_labels =
np.array(subclusters_centroid_main_labels)
subcluster_number = np.array(subcluster_number)

# Simpan hasil klustering ke dalam DataFrame
df_clusters = pd.DataFrame({
    'MainCluster': subclusters_centroid_main_labels,
    'SubCluster': subcluster_number,
    'x': subclusters_centroid[:, 0],
    'y': subclusters_centroid[:, 1]
})

# Simpan DataFrame ke file CSV
df_clusters.to_csv('clusters.csv', index=False)
```

```
print('Clustering has finished')

# Visualisasi hasil clustering menggunakan Voronoi untuk subclusters
fig, ax = plt.subplots(figsize=(14, 10))

# Plot all points
ax.scatter(points[:, 0], points[:, 1], c=cluster_labels_main,
cmap='tab10', s=1, alpha=0.5)

# Plot centroids of main clusters
ax.scatter(kmeans_main.cluster_centers_[:, 0],
kmeans_main.cluster_centers_[:, 1], c='red', s=50, marker='X', label='Main
Cluster Centroids')
# Plot centroids of subclusters
ax.scatter(subclusters_centroid[:, 0], subclusters_centroid[:, 1],
c='green', s=30, marker='o', label='Subcluster Centroids',alpha=0.5)

# Annotate main cluster centroids with cluster labels
for i, (x, y) in enumerate(kmeans_main.cluster_centers_):
    ax.text(x, y, f'Cluster {i}', fontsize=12, ha='center', va='center',
color='red')

# Annotate subcluster centroids with numbers
for i, (x, y) in enumerate(subclusters_centroid):
    ax.text(x, y, str(subcluster_number[i]), fontsize=9, ha='center',
va='center', color='black')

# Plot Voronoi diagram for subclusters
vor = Voronoi(subclusters_centroid)
voronoi_plot_2d(vor, ax=ax, show_vertices=False, line_colors='black',
line_width=1.5, line_alpha=0.6, point_size=2)

# Plot boundary
x, y = boundary.exterior.xy
ax.plot(x, y, color='blue', linewidth=2, label='Boundary')

# Annotate subclusters with numbers
for i, (x, y) in enumerate(subclusters_centroid):
    ax.text(x, y, str(subcluster_number[i]), fontsize=9, ha='center',
va='center', color='black')
```

```
ax.set_xlabel('X coordinate')
ax.set_ylabel('Y coordinate')
ax.set_title('K-means Clustering with Main Clusters and Subclusters')
ax.legend()
plt.show()
```

source code Algoritma Genetika

```
import matplotlib.colors as mcolors
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from shapely.geometry import Polygon, Point
from scipy.spatial import Voronoi, voronoi_plot_2d
import random
import time

# Load the clusters data
df_clusters =
pd.read_csv('https://raw.githubusercontent.com/arvin0711/Kuliah-Soft-Compu
ting/main/EAS_KL_clusterstering.csv')

# Extract main and subcluster centroids
main_cluster_centroids = df_clusters[['MainCluster', 'x',
'y']].groupby('MainCluster').mean().reset_index()
subclusters_centroid = df_clusters[['SubCluster', 'x', 'y']].values
subcluster_number = df_clusters['SubCluster'].values

# Tentukan koordinat dari sudut-sudut batas non-square boundary
boundary_coords = [
    (1.62866, -13.4136),
    (108.826, -119.399),
    (209.65, -179.239),
    (-123.173, -709.999),
    (-225.018, -507.889),
    (-30.0513, -393.164),
    (-111.937, -220.279)
]
```

```
# Buat objek Polygon yang mewakili batas non-square boundary
boundary = Polygon(boundary_coords)

# Generate a background that looks like a field with multiple colors
x_min, y_min, x_max, y_max = boundary.bounds
x_field, y_field = np.meshgrid(np.linspace(x_min, x_max, 100),
np.linspace(y_min, y_max, 100))
field_colors = np.random.rand(100, 100)

# Create a mask for the field inside the boundary
field_mask = np.ones_like(field_colors)  # Start with all white (1 in
terrain colormap)
for i in range(field_colors.shape[0]):
    for j in range(field_colors.shape[1]):
        point = Point(x_field[i, j], y_field[i, j])
        if boundary.contains(point):
            field_mask[i, j] = field_colors[i, j]

# Function to calculate the total cost of a given tour
def calculate_total_cost(tour, distance_matrix, points):
    total_distance = 0
    total_rotation = 0

    # Calculate total distance
    for i in range(len(tour) - 1):
        total_distance += distance_matrix[tour[i], tour[i+1]]

    # Calculate total rotation
    for i in range(len(tour) - 2):
        forward_vector = points[tour[i+1]] - points[tour[i]]
        backward_vector = points[tour[i+2]] - points[tour[i+1]]
        dot_product = np.dot(forward_vector, backward_vector)
        length_forward = np.linalg.norm(forward_vector)
        length_backward = np.linalg.norm(backward_vector)
        cosine_angle = dot_product / (length_forward * length_backward)
        angle_rad = np.arccos(np.clip(cosine_angle, -1.0, 1.0))
        angle_deg = np.degrees(angle_rad)

        if angle_deg < 15:
            angle_deg = 0
```

```
        total_rotation += angle_deg

    total_cost = (total_distance * 8.8/4 * 0.6) + (total_rotation / 50 *
0.4)
    return total_cost, total_distance, total_rotation

# Genetic Algorithm to solve TSP for a given cluster
def genetic_algorithm_tsp(distance_matrix, points, population_size=200,
generations=300, mutation_rate=0.01, crossover_rate=0.7, start_point=None,
end_point=None):
    num_points = len(distance_matrix)

    # Initialize population
    population = [random.sample(range(num_points), num_points) for _ in
range(population_size)]

    def mutate(tour):
        i, j = random.sample(range(num_points), 2)
        tour[i], tour[j] = tour[j], tour[i]
        return tour

    def crossover(parent1, parent2):
        child = [-1]*num_points
        start, end = sorted(random.sample(range(num_points), 2))
        child[start:end] = parent1[start:end]
        pointer = end
        for i in range(num_points):
            if pointer >= num_points:
                pointer = 0
            if parent2[i] not in child:
                while child[pointer] != -1:
                    pointer += 1
                    if pointer >= num_points:
                        pointer = 0
                child[pointer] = parent2[i]
        return child

    def apply_constraints(tour):
        if start_point is not None and end_point is not None:
```

```
            if tour[0] != start_point:
                tour[tour.index(start_point)], tour[0] = tour[0],
tour[tour.index(start_point)]
            if tour[-1] != end_point:
                tour[tour.index(end_point)], tour[-1] = tour[-1],
tour[tour.index(end_point)]
        return tour

    for _ in range(generations):
        population = sorted(population, key=lambda tour:
calculate_total_cost(tour, distance_matrix, points)[0])
        next_population = population[:population_size//2]
        for _ in range(population_size//2, population_size):
            parent1, parent2 =
random.sample(population[:population_size//2], 2)
            if random.random() < crossover_rate:
                child = crossover(parent1, parent2)
            else:
                child = parent1[:]
            if random.random() < mutation_rate:
                child = mutate(child)
            child = apply_constraints(child)
            next_population.append(child)
        population = next_population

    best_tour = min(population, key=lambda tour:
calculate_total_cost(apply_constraints(tour), distance_matrix, points)[0])
    best_cost, best_distance, best_rotation =
calculate_total_cost(best_tour, distance_matrix, points)

    return best_tour, best_cost, best_distance, best_rotation

# Create a distance matrix for all subclusters
all_points = df_clusters[['x', 'y']].values
distance_matrix = np.linalg.norm(all_points[:, None] - all_points[None,
:], axis=-1)

# Define combined clusters with start and end points
combined_clusters = {
```

```
    '3_5': {'indices': df_clusters[(df_clusters['MainCluster'] == 3) |
(df_clusters['MainCluster'] == 5)]['SubCluster'].values - 1, 'start': 19 -
1, 'end': 20 - 1},
    '4_7': {'indices': df_clusters[(df_clusters['MainCluster'] == 4) |
(df_clusters['MainCluster'] == 7)]['SubCluster'].values - 1, 'start': 36 -
1, 'end': 37 - 1},
    '1_6': {'indices': df_clusters[(df_clusters['MainCluster'] == 1) |
(df_clusters['MainCluster'] == 6)]['SubCluster'].values - 1, 'start': 31 -
1, 'end': 32 - 1},
}

# Define individual clusters with start and end points
individual_clusters = {
    0: {'indices': df_clusters[df_clusters['MainCluster'] ==
0]['SubCluster'].values - 1, 'start': 2 - 1, 'end': 5 - 1},
    2: {'indices': df_clusters[df_clusters['MainCluster'] ==
2]['SubCluster'].values - 1, 'start': 15 - 1, 'end': 13 - 1},
}

# Optimize combined clusters
optimal_routes = {}
total_cost = 0
total_runtime = 0

# Define parameters for the genetic algorithm
population_size = 200
generations = 300
crossover_rate = 0.7

for key, cluster_data in combined_clusters.items():
    subcluster_indices = cluster_data['indices']
    subcluster_distance_matrix =
distance_matrix[np.ix_(subcluster_indices, subcluster_indices)]

    start_time = time.time()
    best_tour, best_cost, best_distance, best_rotation =
genetic_algorithm_tsp(subcluster_distance_matrix,
all_points[subcluster_indices], population_size=population_size,
generations=generations, crossover_rate=crossover_rate,
```

```
start_point=subcluster_indices.tolist().index(cluster_data['start']),
end_point=subcluster_indices.tolist().index(cluster_data['end']))
    end_time = time.time()

    optimal_route = subcluster_indices[best_tour]
    optimal_routes[key] = optimal_route

    total_cost += best_cost
    total_runtime += end_time - start_time

    print(f"Optimal route for Combined Clusters {key}: {optimal_route +
1}")
    print(f"Cost: {best_cost} meters, Distance: {best_distance} meters,
Rotation: {best_rotation} degrees")
    print(f"Runtime: {end_time - start_time} seconds\n")

# Optimize individual clusters
for main_cluster, cluster_data in individual_clusters.items():
    subcluster_indices = cluster_data['indices']
    subcluster_distance_matrix =
distance_matrix[np.ix_(subcluster_indices, subcluster_indices)]

    start_time = time.time()
    best_tour, best_cost, best_distance, best_rotation =
genetic_algorithm_tsp(subcluster_distance_matrix,
all_points[subcluster_indices], population_size=population_size,
generations=generations, crossover_rate=crossover_rate,
start_point=subcluster_indices.tolist().index(cluster_data['start']),
end_point=subcluster_indices.tolist().index(cluster_data['end']))
    end_time = time.time()

    optimal_route = subcluster_indices[best_tour]
    optimal_routes[main_cluster] = optimal_route

    total_cost += best_cost
    total_runtime += end_time - start_time

    print(f"Optimal route for Main Cluster {main_cluster}: {optimal_route
+ 1}")
```

```
    print(f"Cost: {best_cost} meters, Distance: {best_distance} meters,
Rotation: {best_rotation} degrees")
    print(f"Runtime: {end_time - start_time} seconds\n")

# Order of main clusters: combined clusters 3_5 > combined clusters 4_7 >
clusters 0 > clusters 2 > combined clusters 1_6
final_order = ['3_5', '4_7', 0, 2, '1_6']
final_route = []

for key in final_order:
    final_route.extend(optimal_routes[key])

# Visualize the final optimal route
points = subclusters_centroid[:, 1:]
vor = Voronoi(points)

# Create a plot
plt.figure(figsize=(14, 10))

# Plot a solid color field inside the boundary
field_polygon = plt.Polygon(boundary.exterior.coords, closed=True,
color='green', alpha=0.5)
plt.gca().add_patch(field_polygon)

# Plot points
plt.scatter(subclusters_centroid[:, 1], subclusters_centroid[:, 2],
c='green', s=30, marker='o', label='Subcluster Centroids', alpha=0.5)

# Annotate subcluster centroids with numbers
for i, (x, y) in enumerate(subclusters_centroid[:, 1:]):
    plt.text(x, y, str(subcluster_number[i]), fontsize=9, ha='left',
va='center_baseline', color='black')

# Define custom colors for agriculture theme
colors = ['green', 'brown', 'blue', 'orange', 'yellow']  # Add more colors
as needed

# Plot Voronoi regions with agriculture-themed colors
for region in range(len(vor.point_region)):
    region_idx = vor.point_region[region]
```

```
    if -1 in vor.regions[region_idx]:  # skip the infinite region
        continue
    polygon = [vor.vertices[i] for i in vor.regions[region_idx]]
    shape = Polygon(polygon)
    if boundary.intersects(shape):
        intersection = boundary.intersection(shape)
        color = colors[region % len(colors)]
        if intersection.geom_type == 'Polygon':
            plt.fill(*zip(*intersection.exterior.coords), color=color,
alpha=0.5)
        elif intersection.geom_type == 'MultiPolygon':
            for poly in intersection:
                plt.fill(*zip(*poly.exterior.coords), color=color,
alpha=0.5)

# Plot boundary
x, y = boundary.exterior.xy
plt.plot(x, y, color='black', linewidth=3, label='Boundary')

# Example final route plot (replace with your actual route data)
all_points = df_clusters[['x', 'y']].values # Example data
for i in range(len(final_route) - 1):
    plt.plot([all_points[final_route[i]][0],
all_points[final_route[i+1]][0]],
             [all_points[final_route[i]][1],
all_points[final_route[i+1]][1]],
             'bo-')

# Remove the line connecting the last point back to the start
plt.scatter(all_points[:, 0], all_points[:, 1], c='red', s=30)
plt.title("Optimal Route for All Subclusters Using Genetic Algorithm
\nParameters: Population Size = {}, Generations = {}\nTotal Cost: {:.2f}
meters, Total Runtime: {:.2f} seconds".format(population_size,
generations, total_cost, total_runtime))
plt.xlabel("X coordinate")
plt.ylabel("Y coordinate")
plt.legend()
plt.show()

# Convert final route indices to SubCluster numbers
```

```
final_route_subcluster =
df_clusters.iloc[final_route]['SubCluster'].values

# Print final route
print("Final Optimal Route: ", final_route_subcluster)
print("Final Distance: ", calculate_total_cost(final_route,
distance_matrix, all_points)[1])
print(f"Total Cost: {total_cost} meters")
print(f"Total Runtime: {total_runtime} seconds")
```

source code Particle Swarm Optimization

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import pandas as pd
from shapely.geometry import Polygon, Point
from scipy.spatial import Voronoi, voronoi_plot_2d, ConvexHull
import random
import time

# Load the clusters data
df_clusters =
pd.read_csv('https://raw.githubusercontent.com/arvin0711/Kuliah-Soft-Computing/main/EAS_KL_clusterstering.csv')

# Extract main and subcluster centroids
main_cluster_centroids = df_clusters[['MainCluster', 'x',
'y']].groupby('MainCluster').mean().reset_index()
subclusters_centroid = df_clusters[['SubCluster', 'x', 'y']].values
subcluster_number = df_clusters['SubCluster'].values

# Tentukan koordinat dari sudut-sudut batas non-square boundary
boundary_coords = [
    (1.62866, -13.4136),
    (108.826, -119.399),
    (209.65, -179.239),
    (-123.173, -709.999),
    (-225.018, -507.889),
    (-30.0513, -393.164),
    (-111.937, -220.279)
```

```
]

# Buat objek Polygon yang mewakili batas non-square boundary
boundary = Polygon(boundary_coords)

# Generate a background that looks like a field with multiple colors
x_min, y_min, x_max, y_max = boundary.bounds
x_field, y_field = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max,
100))
field_colors = np.random.rand(100, 100, 3)  # Use RGB colors

# Create a mask for the field inside the boundary
field_mask = np.zeros((100, 100, 3))  # Start with all black (0 in all channels)

for i in range(field_colors.shape[0]):
    for j in range(field_colors.shape[1]):
        point = Point(x_field[i, j], y_field[i, j])
        if boundary.contains(point):
            field_mask[i, j] = field_colors[i, j]

# Function to calculate the total cost of a given tour
def calculate_total_cost(tour, distance_matrix, points):
    total_distance = 0
    total_rotation = 0

    # Calculate total distance
    for i in range(len(tour) - 1):
        total_distance += distance_matrix[tour[i], tour[i+1]]

    # Calculate total rotation
    for i in range(len(tour) - 2):
        forward_vector = points[tour[i+1]] - points[tour[i]]
        backward_vector = points[tour[i+2]] - points[tour[i+1]]
        dot_product = np.dot(forward_vector, backward_vector)
        length_forward = np.linalg.norm(forward_vector)
        length_backward = np.linalg.norm(backward_vector)
        cosine_angle = dot_product / (length_forward * length_backward)
        angle_rad = np.arccos(np.clip(cosine_angle, -1.0, 1.0))
        angle_deg = np.degrees(angle_rad)
```

```
        if angle_deg < 15:
            angle_deg = 0

        total_rotation += angle_deg

    total_cost = (total_distance * 8.8/4 * 0.6) + (total_rotation / 50 * 0.4)
    return total_cost, total_distance, total_rotation

# Particle Swarm Optimization to solve TSP for a given cluster
def pso_tsp(distance_matrix, points, population_size=40, iterations=300, w=0.9, c1=1.5, c2=2,
start_point=None, end_point=None):
    num_points = len(distance_matrix)

    # Initialize population (particles)
    def initialize_population():
        population = []
        for _ in range(population_size):
            tour = list(range(num_points))
            random.shuffle(tour)
            if start_point is not None:
                tour.remove(start_point)
                tour.insert(0, start_point)
            if end_point is not None:
                tour.remove(end_point)
                tour.append(end_point)
            population.append(tour)
        return population

    population = initialize_population()
    velocities = [[] for _ in range(population_size)]
    pbest = population.copy()
    gbest = min(population, key=lambda tour: calculate_total_cost(tour, distance_matrix,
points)[0])

    def swap_elements(tour, i, j):
        tour[i], tour[j] = tour[j], tour[i]
        return tour

    def apply_velocity(tour, velocity):
        new_tour = tour.copy()
```

```
    for i, j in velocity:
        new_tour = swap_elements(new_tour, i, j)
    return new_tour

def generate_velocity(tour1, tour2):
    velocity = []
    temp_tour = tour1.copy()
    for i in range(num_points):
        if temp_tour[i] != tour2[i]:
            swap_idx = temp_tour.index(tour2[i])
            velocity.append((i, swap_idx))
            temp_tour = swap_elements(temp_tour, i, swap_idx)
    return velocity

def update_velocity(velocity, pbest_velocity, gbest_velocity):
    new_velocity = []
    for v in velocity:
        if random.random() < w:
            new_velocity.append(v)
    for pv in pbest_velocity:
        if random.random() < c1:
            new_velocity.append(pv)
    for gv in gbest_velocity:
        if random.random() < c2:
            new_velocity.append(gv)
    return new_velocity

def apply_constraints(tour):
    if start_point is not None:
        tour.remove(start_point)
        tour.insert(0, start_point)
    if end_point is not None:
        tour.remove(end_point)
        tour.append(end_point)
    return tour

for _ in range(iterations):
    for i in range(population_size):
        velocity = velocities[i]
        new_velocity_pbest = generate_velocity(population[i], pbest[i])
```

```
            new_velocity_gbest = generate_velocity(population[i], gbest)
            velocities[i] = update_velocity(velocity, new_velocity_pbest, new_velocity_gbest)

            population[i] = apply_constraints(apply_velocity(population[i], velocities[i]))
            cost, _, _ = calculate_total_cost(population[i], distance_matrix, points)
            pbest_cost, _, _ = calculate_total_cost(pbest[i], distance_matrix, points)

            if cost < pbest_cost:
                pbest[i] = population[i]

        gbest = min(pbest, key=lambda tour: calculate_total_cost(tour, distance_matrix, points)[0])

    best_cost, best_distance, best_rotation = calculate_total_cost(gbest, distance_matrix, points)

    return gbest, best_cost, best_distance, best_rotation

# Create a distance matrix for all subclusters
all_points = df_clusters[['x', 'y']].values
distance_matrix = np.linalg.norm(all_points[:, None] - all_points[None, :], axis=-1)

# Define combined clusters with start and end points
combined_clusters = {
    '3_5': {'indices': df_clusters[(df_clusters['MainCluster'] == 3) | (df_clusters['MainCluster'] ==
5)]['SubCluster'].values - 1, 'start': 19 - 1, 'end': 20 - 1},
    '4_7': {'indices': df_clusters[(df_clusters['MainCluster'] == 4) | (df_clusters['MainCluster'] ==
7)]['SubCluster'].values - 1, 'start': 36 - 1, 'end': 37 - 1},
    '1_6': {'indices': df_clusters[(df_clusters['MainCluster'] == 1) | (df_clusters['MainCluster'] ==
6)]['SubCluster'].values - 1, 'start': 31 - 1, 'end': 32 - 1},
}

# Define individual clusters with start and end points
individual_clusters = {
    0: {'indices': df_clusters[df_clusters['MainCluster'] == 0]['SubCluster'].values - 1, 'start': 2 - 1,
'end': 5 - 1},
    2: {'indices': df_clusters[df_clusters['MainCluster'] == 2]['SubCluster'].values - 1, 'start': 15 -
1, 'end': 13 - 1},
}

# Optimize combined clusters
optimal_routes = {}
```

```
total_cost = 0
total_runtime = 0

for key, cluster_data in combined_clusters.items():
    subcluster_indices = cluster_data['indices']
    subcluster_distance_matrix = distance_matrix[np.ix_(subcluster_indices, subcluster_indices)]

    start_time = time.time()
    best_tour, best_cost, best_distance, best_rotation = pso_tsp(subcluster_distance_matrix,
all_points[subcluster_indices], start_point=subcluster_indices.tolist().index(cluster_data['start']),
end_point=subcluster_indices.tolist().index(cluster_data['end']))
    end_time = time.time()

    optimal_route = subcluster_indices[best_tour]
    optimal_routes[key] = optimal_route

    total_cost += best_cost
    total_runtime += end_time - start_time

    print(f"Optimal route for Combined Clusters {key}: {optimal_route + 1}")
    print(f"Cost: {best_cost} meters, Distance: {best_distance} meters, Rotation: {best_rotation}
degrees")
    print(f"Runtime: {end_time - start_time} seconds\n")

# Optimize individual clusters
for main_cluster, cluster_data in individual_clusters.items():
    subcluster_indices = cluster_data['indices']
    subcluster_distance_matrix = distance_matrix[np.ix_(subcluster_indices, subcluster_indices)]

    start_time = time.time()
    best_tour, best_cost, best_distance, best_rotation = pso_tsp(subcluster_distance_matrix,
all_points[subcluster_indices], start_point=subcluster_indices.tolist().index(cluster_data['start']),
end_point=subcluster_indices.tolist().index(cluster_data['end']))
    end_time = time.time()

    optimal_route = subcluster_indices[best_tour]
    optimal_routes[main_cluster] = optimal_route

    total_cost += best_cost
    total_runtime += end_time - start_time
```

```
    print(f"Optimal route for Main Cluster {main_cluster}: {optimal_route + 1}")
    print(f"Cost: {best_cost} meters, Distance: {best_distance} meters, Rotation: {best_rotation}
degrees")
    print(f"Runtime: {end_time - start_time} seconds\n")

# Order of main clusters: combined clusters 3_5 > combined clusters 4_7 > clusters 0 > clusters
2 > combined clusters 1_6
final_order = ['3_5', '4_7', 0, 2, '1_6']
final_route = []

for key in final_order:
    final_route.extend(optimal_routes[key])
import matplotlib.colors as mcolors

points = subclusters_centroid[:, 1:]
vor = Voronoi(points)



# Create a plot
plt.figure(figsize=(14, 10))

# Plot a solid color field inside the boundary
field_polygon = plt.Polygon(boundary.exterior.coords, closed=True, color='green', alpha=0.5)
plt.gca().add_patch(field_polygon)

# Plot points
plt.scatter(subclusters_centroid[:, 1], subclusters_centroid[:, 2], c='green', s=30, marker='o',
label='Subcluster Centroids', alpha=0.5)

# Annotate subcluster centroids with numbers
for i, (x, y) in enumerate(subclusters_centroid[:, 1:]):
    plt.text(x, y, str(subcluster_number[i]), fontsize=9, ha='left', va='center_baseline',
color='black')

# Define custom colors for agriculture theme
colors = ['green', 'brown', 'blue', 'orange', 'yellow']  # Add more colors as needed

# Plot Voronoi regions with agriculture-themed colors
```

```
for region in range(len(vor.point_region)):
    region_idx = vor.point_region[region]
    if -1 in vor.regions[region_idx]:  # skip the infinite region
        continue
    polygon = [vor.vertices[i] for i in vor.regions[region_idx]]
    shape = Polygon(polygon)
    if boundary.intersects(shape):
        intersection = boundary.intersection(shape)
        color = colors[region % len(colors)]
        if intersection.geom_type == 'Polygon':
            plt.fill(*zip(*intersection.exterior.coords), color=color, alpha=0.5)
        elif intersection.geom_type == 'MultiPolygon':
            for poly in intersection:
                plt.fill(*zip(*poly.exterior.coords), color=color, alpha=0.5)

# Plot boundary
x, y = boundary.exterior.xy
plt.plot(x, y, color='black', linewidth=3, label='Boundary')

# Example final route plot (replace with your actual route data)
all_points = df_clusters[['x', 'y']].values # Example data
for i in range(len(final_route) - 1):
    plt.plot([all_points[final_route[i]][0], all_points[final_route[i+1]][0]],
             [all_points[final_route[i]][1], all_points[final_route[i+1]][1]],
             'bo-')

# Remove the line connecting the last point back to the start
plt.scatter(all_points[:, 0], all_points[:, 1], c='red', s=30)

plt.title("Optimal Route for All Subclusters")
plt.xlabel("X coordinate")
plt.ylabel("Y coordinate")
plt.legend()
plt.show()

# Convert final route indices to SubCluster numbers
final_route_subcluster = df_clusters.iloc[final_route]['SubCluster'].values

# Print final route
print("Final Optimal Route: ", final_route_subcluster)
```

```
print("Final Distance: ", calculate_total_cost(final_route, distance_matrix, all_points)[1])
print(f"Total Cost: {total_cost} meters")
print(f"Total Runtime: {total_runtime} seconds")
```

source code Ant Colony Optimization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from shapely.geometry import Polygon, Point
from scipy.spatial import Voronoi, voronoi_plot_2d
import random
import time

# Load the clusters data
df_clusters =
pd.read_csv('https://raw.githubusercontent.com/arvin0711/Kuliah-Soft-Computing/main/EAS_KL_clusterstering.csv')

# Extract main and subcluster centroids
main_cluster_centroids = df_clusters[['MainCluster', 'x',
'y']].groupby('MainCluster').mean().reset_index()
subclusters_centroid = df_clusters[['SubCluster', 'x', 'y']].values
subcluster_number = df_clusters['SubCluster'].values

# Tentukan koordinat dari sudut-sudut batas non-square boundary
boundary_coords = [
  (1.62866, -13.4136),
  (108.826, -119.399),
  (209.65, -179.239),
  (-123.173, -709.999),
  (-225.018, -507.889),
```

```
  (-30.0513, -393.164),
  (-111.937, -220.279)
]

# Buat objek Polygon yang mewakili batas non-square boundary
boundary = Polygon(boundary_coords)

# Generate a background that looks like a field with multiple colors
x_min, y_min, x_max, y_max = boundary.bounds
x_field, y_field = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max,
100))
field_colors = np.random.rand(100, 100)

# Create a mask for the field inside the boundary
field_mask = np.ones_like(field_colors)  # Start with all white (1 in terrain colormap)
for i in range(field_colors.shape[0]):
  for j in range(field_colors.shape[1]):
    point = Point(x_field[i, j], y_field[i, j])
    if boundary.contains(point):
      field_mask[i, j] = field_colors[i, j]

# Function to calculate the total cost of a given tour
def calculate_total_cost(tour, distance_matrix, points):
  total_distance = 0
  total_rotation = 0

  # Calculate total distance
  for i in range(len(tour) - 1):
    total_distance += distance_matrix[tour[i], tour[i+1]]

  # Calculate total rotation
```

```
    for i in range(len(tour) - 2):
        forward_vector = points[tour[i+1]] - points[tour[i]]
        backward_vector = points[tour[i+2]] - points[tour[i+1]]
        dot_product = np.dot(forward_vector, backward_vector)
        length_forward = np.linalg.norm(forward_vector)
        length_backward = np.linalg.norm(backward_vector)
        cosine_angle = dot_product / (length_forward * length_backward)
        angle_rad = np.arccos(np.clip(cosine_angle, -1.0, 1.0))
        angle_deg = np.degrees(angle_rad)

        if angle_deg < 15:
            angle_deg = 0

        total_rotation += angle_deg

    total_cost = (total_distance * 8.8/4 * 0.6) + (total_rotation / 50 * 0.4)
    return total_cost, total_distance, total_rotation

# Ant Colony Optimization to solve TSP for a given cluster
def ant_colony_optimization_tsp(distance_matrix, points, num_ants=100, num_iterations=200,
alpha=1.0, beta=5.0, evaporation_rate=0.5, q=100, start_point=None, end_point=None):
    num_points = len(distance_matrix)
    pheromone_matrix = np.ones((num_points, num_points)) / num_points
    best_tour = None
    best_cost = float('inf')

    for _ in range(num_iterations):
        all_tours = []
        all_costs = []

        for _ in range(num_ants):
```

```
        tour = [start_point] if start_point is not None else [random.choice(range(num_points))]
        while len(tour) < num_points:
            current_point = tour[-1]
            probabilities = []
            for next_point in range(num_points):
                if next_point not in tour:
                    pheromone = pheromone_matrix[current_point, next_point] ** alpha
                    visibility = (1 / distance_matrix[current_point, next_point]) ** beta
                    probabilities.append(pheromone * visibility)
                else:
                    probabilities.append(0)
            probabilities = np.array(probabilities)
            probabilities /= probabilities.sum()
            next_point = np.random.choice(range(num_points), p=probabilities)
            tour.append(next_point)

        if end_point is not None:
            if end_point in tour:
                tour.remove(end_point)
            tour.append(end_point)

        all_tours.append(tour)
        cost, _, _ = calculate_total_cost(tour, distance_matrix, points)
        all_costs.append(cost)

        if cost < best_cost:
            best_cost = cost
            best_tour = tour

    pheromone_matrix *= (1 - evaporation_rate)
    for i in range(num_ants):
```

```
        for j in range(num_points - 1):
            pheromone_matrix[all_tours[i][j], all_tours[i][j + 1]] += q / all_costs[i]

  best_cost, best_distance, best_rotation = calculate_total_cost(best_tour, distance_matrix,
points)
  return best_tour, best_cost, best_distance, best_rotation

# Create a distance matrix for all subclusters
all_points = df_clusters[['x', 'y']].values
distance_matrix = np.linalg.norm(all_points[:, None] - all_points[None, :], axis=-1)

# Define combined clusters with start and end points
combined_clusters = {
  '3_5': {'indices': df_clusters[(df_clusters['MainCluster'] == 3) | (df_clusters['MainCluster'] ==
5)]['SubCluster'].values - 1, 'start': 27 - 1, 'end': 28 - 1},
  '4_7': {'indices': df_clusters[(df_clusters['MainCluster'] == 4) | (df_clusters['MainCluster'] ==
7)]['SubCluster'].values - 1, 'start': 24 - 1, 'end': 22 - 1},
  '1_6': {'indices': df_clusters[(df_clusters['MainCluster'] == 1) | (df_clusters['MainCluster'] ==
6)]['SubCluster'].values - 1, 'start': 7 - 1, 'end': 8 - 1},
}

# Define individual clusters with start and end points
individual_clusters = {
  0: {'indices': df_clusters[df_clusters['MainCluster'] == 0]['SubCluster'].values - 1, 'start': 4 - 1,
'end': 1 - 1},
  2: {'indices': df_clusters[df_clusters['MainCluster'] == 2]['SubCluster'].values - 1, 'start': 11 - 1,
'end': 12 - 1},
}

# Optimize combined clusters
optimal_routes = {}
```

```
total_cost = 0
total_runtime = 0

for key, cluster_data in combined_clusters.items():
    subcluster_indices = cluster_data['indices']
    subcluster_distance_matrix = distance_matrix[np.ix_(subcluster_indices, subcluster_indices)]

    start_time = time.time()
    best_tour, best_cost, best_distance, best_rotation = ant_colony_optimization_tsp(
        subcluster_distance_matrix,
        all_points[subcluster_indices],
        start_point=subcluster_indices.tolist().index(cluster_data['start']),
        end_point=subcluster_indices.tolist().index(cluster_data['end'])
    )
    end_time = time.time()

    optimal_route = subcluster_indices[best_tour]
    optimal_routes[key] = optimal_route

    total_cost += best_cost
    total_runtime += end_time - start_time

    print(f"Optimal route for Combined Clusters {key}: {optimal_route + 1}")
    print(f"Cost: {best_cost} meters, Distance: {best_distance} meters, Rotation: {best_rotation}
degrees")
    print(f"Runtime: {end_time - start_time} seconds\n")

# Optimize individual clusters
for main_cluster, cluster_data in individual_clusters.items():
    subcluster_indices = cluster_data['indices']
    subcluster_distance_matrix = distance_matrix[np.ix_(subcluster_indices, subcluster_indices)]
```

```
    start_time = time.time()
    best_tour, best_cost, best_distance, best_rotation = ant_colony_optimization_tsp(
        subcluster_distance_matrix,
        all_points[subcluster_indices],
        start_point=subcluster_indices.tolist().index(cluster_data['start']),
        end_point=subcluster_indices.tolist().index(cluster_data['end'])
    )
    end_time = time.time()

    optimal_route = subcluster_indices[best_tour]
    optimal_routes[main_cluster] = optimal_route

    total_cost += best_cost
    total_runtime += end_time - start_time

    print(f"Optimal route for Main Cluster {main_cluster}: {optimal_route + 1}")
    print(f"Cost: {best_cost} meters, Distance: {best_distance} meters, Rotation: {best_rotation}
degrees")
    print(f"Runtime: {end_time - start_time} seconds\n")

# Order of main clusters: combined clusters 3_5 > combined clusters 4_7 > clusters 0 > clusters
2 > combined clusters 1_6
final_order = ['3_5', '4_7', 0, 2, '1_6']
final_route = []

for key in final_order:
    final_route.extend(optimal_routes[key])

# Visualize the final optimal route
plt.figure(figsize=(14, 10))  # Adjusted size to match the initial example
```

```
# Plot the field background
plt.imshow(field_mask, extent=[x_min, x_max, y_min, y_max], origin='lower', cmap='terrain',
alpha=0.5, aspect='auto')

# Overlay a white background
plt.gca().patch.set_facecolor('white')

# Plot points
plt.scatter(subclusters_centroid[:, 1], subclusters_centroid[:, 2], c='green', s=30, marker='o',
label='Subcluster Centroids', alpha=0.5)

# Annotate subcluster centroids with numbers
for i, (x, y) in enumerate(subclusters_centroid[:, 1:]):
  plt.text(x, y, str(subcluster_number[i]), fontsize=9, ha='left', va='center_baseline',
color='black')

# Plot Voronoi diagram for subclusters
vor = Voronoi(subclusters_centroid[:, 1:])
voronoi_plot_2d(vor, ax=plt.gca(), show_vertices=False, line_colors='black', line_width=1.5,
line_alpha=0.6, point_size=2)

# Plot boundary
x, y = boundary.exterior.xy
plt.plot(x, y, color='black', linewidth=3, label='Boundary')

# Plot the optimal route
for i in range(len(final_route) - 1):
  plt.plot([all_points[final_route[i]][0], all_points[final_route[i+1]][0]],
[all_points[final_route[i]][1], all_points[final_route[i+1]][1]], 'bo-')

# Remove the line connecting the last point back to the start
```

```
plt.scatter(all_points[:, 0], all_points[:, 1], c='red', s=30)
plt.title("Optimal Route for All Subclusters")
plt.xlabel("X coordinate")
plt.ylabel("Y coordinate")
plt.legend()
plt.show()

# Convert final route indices to SubCluster numbers
final_route_subcluster = df_clusters.iloc[final_route]['SubCluster'].values

# Print final route
print("Final Optimal Route: ", final_route_subcluster)
print("Final Distance: ", calculate_total_cost(final_route, distance_matrix, all_points)[1])
print(f"Total Cost: {total_cost} meters")
print(f"Total Runtime: {total_runtime} seconds")
```