Ant colony optimization:

```
import numpy as np
import random

def create_distance_matrix(n_cities):
    np.random.seed(0)
    matrix = np.random.randint(1, 100, size=(n_cities, n_cities))
    np.fill_diagonal(matrix, 0)
    return matrix

n_cities = 10
n_ants = 20
n_iterations = 50
alpha = 1
beta = 2
evaporation_rate = 0.5
initial_pheromone = 1

distance_matrix = create_distance_matrix(n_cities)
pheromone_matrix = np.ones((n_cities, n_cities)) * initial_pheromone

class Ant:
    def __init__(self, n_cities):
        self.n_cities = n_cities
        self.route = []
        self.distance_travelled = 0

    def select_next_city(self, current_city, visited):
        probabilities = []
        for city in range(self.n_cities):
            if city not in visited:
                pheromone = pheromone_matrix[current_city][city] ** alpha
                heuristic = (1 / distance_matrix[current_city][city]) **
beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)
        probabilities = np.array(probabilities)
        probabilities /= sum(probabilities) if sum(probabilities) > 0 else
1  # Avoid division by zero
```

```
        next_city = np.random.choice(range(self.n_cities),
p=probabilities)
        return next_city

    def find_route(self):
        current_city = random.randint(0, self.n_cities - 1)
        self.route = [current_city]
        visited = set(self.route)
        while len(visited) < self.n_cities:
            next_city = self.select_next_city(current_city, visited)
            self.route.append(next_city)
            self.distance_travelled +=
distance_matrix[current_city][next_city]
            visited.add(next_city)
            current_city = next_city
        self.distance_travelled +=
distance_matrix[self.route[-1]][self.route[0]]  # Return to start
        self.route.append(self.route[0])  # Complete the cycle

def update_pheromones(ants):
    global pheromone_matrix
    pheromone_matrix *= (1 - evaporation_rate)
    for ant in ants:
        for i in range(len(ant.route) - 1):
            city_from = ant.route[i]
            city_to = ant.route[i + 1]
            pheromone_matrix[city_from][city_to] += 1.0 /
ant.distance_travelled
            pheromone_matrix[city_to][city_from] += 1.0 /
ant.distance_travelled

def ant_colony_optimization():
    best_route = None
    best_distance = float('inf')
    for iteration in range(n_iterations):
        ants = [Ant(n_cities) for _ in range(n_ants)]
        for ant in ants:
            ant.find_route()
            if ant.distance_travelled < best_distance:
                best_distance = ant.distance_travelled
```

```
                best_route = ant.route
        update_pheromones(ants)
        print(f"Iteration {iteration + 1}: Best distance =
{best_distance}")
    return best_route, best_distance

best_route, best_distance = ant_colony_optimization()
print(f"Best route found: {best_route} with distance: {best_distance}")
import numpy as np
import random

def create_distance_matrix(n_cities):
    np.random.seed(0)
    matrix = np.random.randint(1, 100, size=(n_cities, n_cities))
    np.fill_diagonal(matrix, 0)
    return matrix

n_cities = 10
n_ants = 20
n_iterations = 50
alpha = 1
beta = 2
evaporation_rate = 0.5
initial_pheromone = 1

distance_matrix = create_distance_matrix(n_cities)
pheromone_matrix = np.ones((n_cities, n_cities)) * initial_pheromone

class Ant:
    def __init__(self, n_cities):
        self.n_cities = n_cities
        self.route = []
        self.distance_travelled = 0

    def select_next_city(self, current_city, visited):
        probabilities = []
        for city in range(self.n_cities):
            if city not in visited:
                pheromone = pheromone_matrix[current_city][city] ** alpha
```

```
                heuristic = (1 / distance_matrix[current_city][city]) **
beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)
        probabilities = np.array(probabilities)
        probabilities /= sum(probabilities) if sum(probabilities) > 0 else
1  # Avoid division by zero
        next_city = np.random.choice(range(self.n_cities),
p=probabilities)
        return next_city

    def find_route(self):
        current_city = random.randint(0, self.n_cities - 1)
        self.route = [current_city]
        visited = set(self.route)
        while len(visited) < self.n_cities:
            next_city = self.select_next_city(current_city, visited)
            self.route.append(next_city)
            self.distance_travelled +=
distance_matrix[current_city][next_city]
            visited.add(next_city)
            current_city = next_city
        self.distance_travelled +=
distance_matrix[self.route[-1]][self.route[0]]  # Return to start
        self.route.append(self.route[0])  # Complete the cycle

def update_pheromones(ants):
    global pheromone_matrix
    pheromone_matrix *= (1 - evaporation_rate)
    for ant in ants:
        for i in range(len(ant.route) - 1):
            city_from = ant.route[i]
            city_to = ant.route[i + 1]
            pheromone_matrix[city_from][city_to] += 1.0 /
ant.distance_travelled
            pheromone_matrix[city_to][city_from] += 1.0 /
ant.distance_travelled

def ant_colony_optimization():
```

```
    best_route = None
    best_distance = float('inf')
    for iteration in range(n_iterations):
        ants = [Ant(n_cities) for _ in range(n_ants)]
        for ant in ants:
            ant.find_route()
            if ant.distance_travelled < best_distance:
                best_distance = ant.distance_travelled
                best_route = ant.route
        update_pheromones(ants)
        print(f"Iteration {iteration + 1}: Best distance =
{best_distance}")
    return best_route, best_distance

best_route, best_distance = ant_colony_optimization()
print(f"Best route found: {best_route} with distance: {best_distance}")
```

Output:

```
Iteration 1: Best distance = 139
Iteration 2: Best distance = 130
Iteration 3: Best distance = 130
Iteration 4: Best distance = 130
Iteration 5: Best distance = 130
Iteration 6: Best distance = 130
Iteration 7: Best distance = 130
Iteration 8: Best distance = 130
Iteration 9: Best distance = 130
Iteration 10: Best distance = 130
Iteration 11: Best distance = 130
Iteration 12: Best distance = 130
Iteration 13: Best distance = 130
Iteration 14: Best distance = 130
Iteration 15: Best distance = 130
Iteration 16: Best distance = 130
Iteration 17: Best distance = 130
Iteration 18: Best distance = 130
Iteration 19: Best distance = 130
Iteration 20: Best distance = 130
Iteration 21: Best distance = 130
Iteration 22: Best distance = 130
Iteration 23: Best distance = 130
Iteration 24: Best distance = 130
Iteration 25: Best distance = 130
Iteration 26: Best distance = 130
Iteration 27: Best distance = 130
Iteration 28: Best distance = 130
Iteration 29: Best distance = 130
Iteration 30: Best distance = 130
Iteration 31: Best distance = 130
Iteration 32: Best distance = 130
Iteration 33: Best distance = 130
Iteration 34: Best distance = 130
Iteration 35: Best distance = 130
Iteration 36: Best distance = 130
Iteration 37: Best distance = 130
Iteration 38: Best distance = 130
Iteration 39: Best distance = 130
Iteration 40: Best distance = 130
```

```
Iteration 39: Best distance = 130
Iteration 40: Best distance = 130
Iteration 41: Best distance = 130
Iteration 42: Best distance = 130
Iteration 43: Best distance = 130
Iteration 44: Best distance = 130
Iteration 45: Best distance = 130
Iteration 46: Best distance = 118
Iteration 47: Best distance = 118
Iteration 48: Best distance = 118
Iteration 49: Best distance = 118
Iteration 50: Best distance = 118
Best route found: [7, 4, 2, 6, 1, 3, 8, 9, 0, 5, 7] with distance: 118
```

Application 2-

```
import numpy as np
import random

def create_flight_matrix(n_airports):
    np.random.seed(0)
    # Random flight durations between airports (in hours), but with random
operational costs (fuel, crew)
    duration_matrix = np.random.randint(1, 6, size=(n_airports,
n_airports))  # Flight duration in hours
    cost_matrix = np.random.randint(100, 1000, size=(n_airports,
n_airports))  # Operational cost (fuel, crew)

    # Ensure no self-loops
    np.fill_diagonal(duration_matrix, 0)
    np.fill_diagonal(cost_matrix, 0)

    return duration_matrix, cost_matrix

def create_airport_slots(n_airports):
    # Max slots per airport per hour
    return np.random.randint(3, 6, size=n_airports)  # Random available
slots per hour

n_airports = 10
n_ants = 20
n_iterations = 50
alpha = 1
```

```
beta = 2
evaporation_rate = 0.5
initial_pheromone = 1
max_aircraft_idle_time = 2  # Max idle time (hours) between consecutive
flights for an aircraft

# Create flight duration matrix and operational cost matrix
duration_matrix, cost_matrix = create_flight_matrix(n_airports)
# Create airport slot availability (slots per hour)
airport_slots = create_airport_slots(n_airports)

# Initialize pheromone matrix
pheromone_matrix = np.ones((n_airports, n_airports)) * initial_pheromone

class Flight:
    def __init__(self, n_airports):
        self.n_airports = n_airports
        self.route = []
        self.total_duration = 0
        self.total_cost = 0
        self.idle_time = 0
        self.slot_usage = [0] * n_airports  # Track slots used by the
flight at each airport

    def select_next_airport(self, current_airport, visited,
time_at_airport):
        probabilities = []
        for airport in range(self.n_airports):
            if airport not in visited:
                # Heuristic: balance between pheromone and inverse flight
duration + cost
                pheromone = pheromone_matrix[current_airport][airport] **
alpha
                heuristic = (1 /
duration_matrix[current_airport][airport]) ** beta
                cost = (1 / cost_matrix[current_airport][airport]) ** beta
                probabilities.append(pheromone * heuristic * cost)
            else:
                probabilities.append(0)
```

```
        probabilities = np.array(probabilities)
        probabilities /= sum(probabilities) if sum(probabilities) > 0 else
1  # Avoid division by zero
        next_airport = np.random.choice(range(self.n_airports),
p=probabilities)
        return next_airport

    def find_route(self):
        current_airport = random.randint(0, self.n_airports - 1)
        self.route = [current_airport]
        visited = set(self.route)
        time_at_airport = np.zeros(self.n_airports)  # Track when each
airport is next available (in hours)

        while len(visited) < self.n_airports:
            next_airport = self.select_next_airport(current_airport,
visited, time_at_airport)
            self.route.append(next_airport)
            flight_duration =
duration_matrix[current_airport][next_airport]
            flight_cost = cost_matrix[current_airport][next_airport]
            self.total_duration += flight_duration
            self.total_cost += flight_cost
            time_at_airport[next_airport] += flight_duration  # Update
next available time for airport
            visited.add(next_airport)
            current_airport = next_airport

            # Check if airport slot constraint is violated
            if time_at_airport[current_airport] >
airport_slots[current_airport]:
                self.total_cost += 1000  # Penalty for exceeding slot
capacity (example cost)

        # Return to the start airport
        self.total_duration +=
duration_matrix[self.route[-1]][self.route[0]]
        self.total_cost += cost_matrix[self.route[-1]][self.route[0]]
        self.route.append(self.route[0])  # Complete the cycle
```

```python
def update_pheromones(flights):
    global pheromone_matrix
    pheromone_matrix *= (1 - evaporation_rate)

    for flight in flights:
        for i in range(len(flight.route) - 1):
            airport_from = flight.route[i]
            airport_to = flight.route[i + 1]
            # Update pheromone based on cost (inverse of total cost, lower cost, more pheromone)
            pheromone_matrix[airport_from][airport_to] += 1.0 / (flight.total_cost + 1e-5)
            pheromone_matrix[airport_to][airport_from] += 1.0 / (flight.total_cost + 1e-5)

def airline_scheduling_optimization():
    best_route = None
    best_cost = float('inf')
    best_duration = float('inf')

    for iteration in range(n_iterations):
        flights = [Flight(n_airports) for _ in range(n_ants)]
        for flight in flights:
            flight.find_route()
            if flight.total_cost < best_cost:
                best_cost = flight.total_cost
                best_route = flight.route
            if flight.total_duration < best_duration:
                best_duration = flight.total_duration

        update_pheromones(flights)
        print(f"Iteration {iteration + 1}: Best cost = {best_cost}, Best duration = {best_duration}")

    return best_route, best_cost, best_duration

best_route, best_cost, best_duration = airline_scheduling_optimization()
print(f"Best route found: {best_route} with cost: {best_cost} and duration: {best_duration}")
```

Output:

```
Iteration 1: Best cost = 3328, Best duration = 19
Iteration 2: Best cost = 3134, Best duration = 15
Iteration 3: Best cost = 3134, Best duration = 15
Iteration 4: Best cost = 3134, Best duration = 15
Iteration 5: Best cost = 3134, Best duration = 15
Iteration 6: Best cost = 2837, Best duration = 15
Iteration 7: Best cost = 2837, Best duration = 15
Iteration 8: Best cost = 2837, Best duration = 14
Iteration 9: Best cost = 2837, Best duration = 14
Iteration 10: Best cost = 2837, Best duration = 14
Iteration 11: Best cost = 2837, Best duration = 14
Iteration 12: Best cost = 2837, Best duration = 14
Iteration 13: Best cost = 2837, Best duration = 14
Iteration 14: Best cost = 2837, Best duration = 14
Iteration 15: Best cost = 2837, Best duration = 14
Iteration 16: Best cost = 2837, Best duration = 14
Iteration 17: Best cost = 2837, Best duration = 14
Iteration 18: Best cost = 2826, Best duration = 14
Iteration 19: Best cost = 2826, Best duration = 14
Iteration 20: Best cost = 2826, Best duration = 14
Iteration 21: Best cost = 2826, Best duration = 14
Iteration 22: Best cost = 2826, Best duration = 14
Iteration 23: Best cost = 2826, Best duration = 14
Iteration 24: Best cost = 2826, Best duration = 14
Iteration 25: Best cost = 2826, Best duration = 14
Iteration 26: Best cost = 2826, Best duration = 14
Iteration 27: Best cost = 2826, Best duration = 14
Iteration 28: Best cost = 2826, Best duration = 14
Iteration 29: Best cost = 2826, Best duration = 14
Iteration 30: Best cost = 2826, Best duration = 14
Iteration 31: Best cost = 2826, Best duration = 14
Iteration 32: Best cost = 2826, Best duration = 14
Iteration 33: Best cost = 2826, Best duration = 14
Iteration 34: Best cost = 2795, Best duration = 14
Iteration 35: Best cost = 2795, Best duration = 14
```

```
Iteration 33: Best cost = 2826, Best duration = 14
Iteration 34: Best cost = 2795, Best duration = 14
Iteration 35: Best cost = 2795, Best duration = 14
Iteration 36: Best cost = 2795, Best duration = 14
Iteration 37: Best cost = 2795, Best duration = 14
Iteration 38: Best cost = 2523, Best duration = 14
Iteration 39: Best cost = 2523, Best duration = 14
Iteration 40: Best cost = 2523, Best duration = 14
Iteration 41: Best cost = 2523, Best duration = 14
Iteration 42: Best cost = 2523, Best duration = 14
Iteration 43: Best cost = 2523, Best duration = 14
Iteration 44: Best cost = 2523, Best duration = 14
Iteration 45: Best cost = 2523, Best duration = 14
Iteration 46: Best cost = 2523, Best duration = 14
Iteration 47: Best cost = 2523, Best duration = 14
Iteration 48: Best cost = 2523, Best duration = 14
Iteration 49: Best cost = 2523, Best duration = 14
Iteration 50: Best cost = 2523, Best duration = 14
Best route found: [9, 5, 2, 4, 8, 0, 1, 7, 6, 3, 9] with cost: 2523 and duration: 14
```