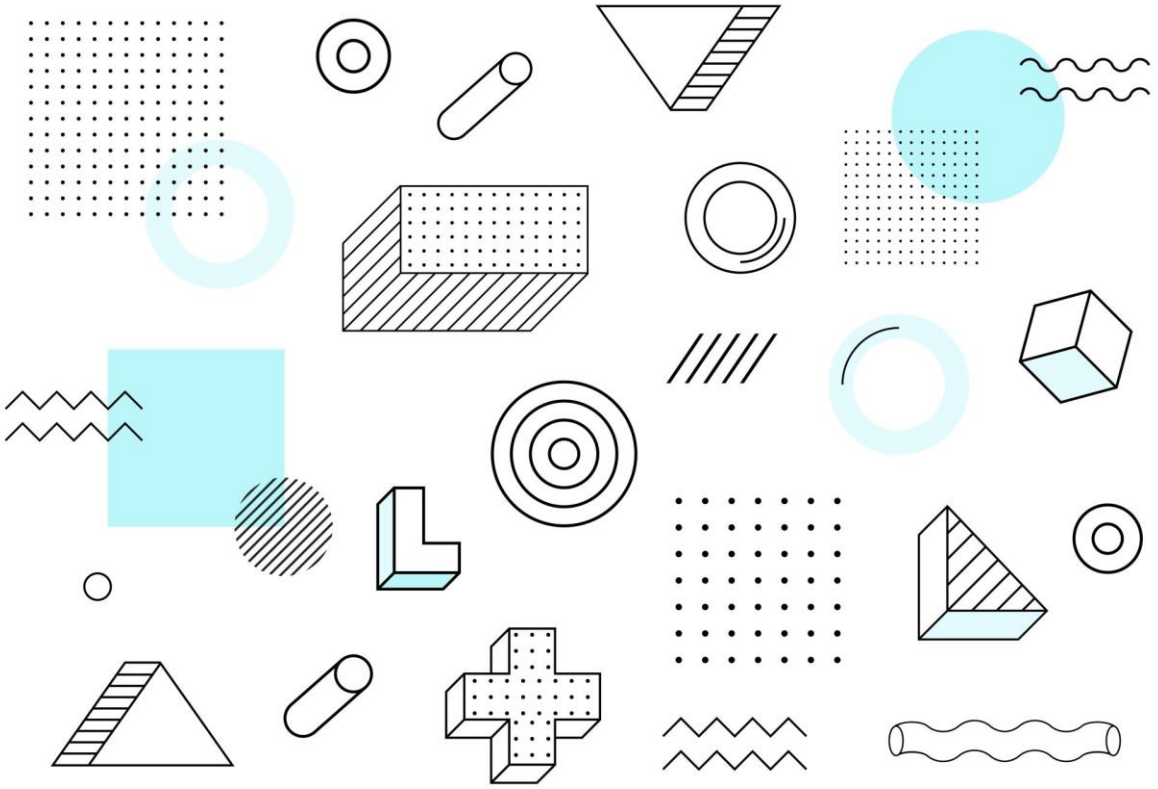
كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE
BENHA UNIVERSITY



Planning Techniques for Robotics
Lab 05 – Ant Colony Optimization (Practical)

Eng. Yousef Elbaroudy

2025/2026

GUIDELINES

- You don’t need to memorize what will be mentioned. Instead, try to understand

- Each PowerPoint Presentation will be available as PDF file on Google Drive Folder of the Course

Ant Colony Optimization (Recap)

- **Ant Colony Optimization (ACO):** a **nature-inspired optimization algorithm** based on how real ants find the shortest paths to food. It's commonly used to solve **NP-Hard combinatorial optimization problems**, especially ones involving ***paths, routes, or sequences***.

- In other words, **Ant Colony** is a way to solve optimization problems based on the way that ants indirectly communicate directions to each other.

Motivation

The primary motivation behind studying ACO lies in its remarkable **efficiency in resolving NP-hard problems** like the Traveling Salesman Problem and various routing and scheduling issues.
This efficiency makes ACO a valuable tool in optimization.

Ant Colony Optimization (Recap)

- **Ant Colony Optimization (ACO):** a **nature-inspired optimization algorithm** based on how real ants find the shortest paths to food. It's commonly used to solve **NP-Hard combinatorial optimization problems**, especially ones involving ***paths, routes, or sequences***.

- In other words, **Ant Colony** is a way to solve optimization problems based on the way that ants indirectly communicate directions to each other.

- **Ant Colony Optimization (ACO)** is a **population-based, swarm intelligent and sign-based algorithm** because it works with a set of agents (ants) that ***collectively*** search for solutions rather than improving a single solution at a time.

Baseline of ACO (Inspiration) –Recap–

- ❑ Ants exhibit remarkable ***collective intelligence*** while searching for food.
- ❑ **An ant's path represents a specific candidate solution.**
- ❑ Initially, they move **randomly to explore** the environment. **(The decision Rule)**
- ❑ Once a food source is found, ants return to the nest, **depositing pheromones along their path**. **(The learning Rule)**
- ❑ Other ants detect these pheromones and **ARE MORE LIKELY** to follow stronger trails.
- ❑ Over time, the shortest and most efficient routes emerge naturally **through repeated reinforcement**.

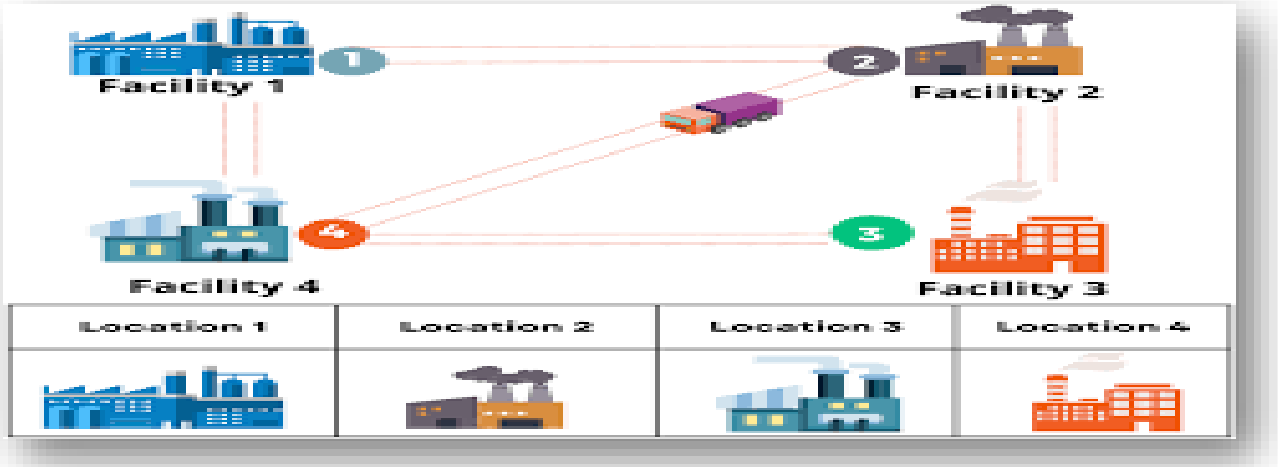
Pheromone is the basic building block of ant's behavior

Common Problem to Solve

Quadratic Assignment Problem (QAP)

a set of n facilities must be assigned to n locations in a one-to-one manner, such that the total assignment cost is minimized.

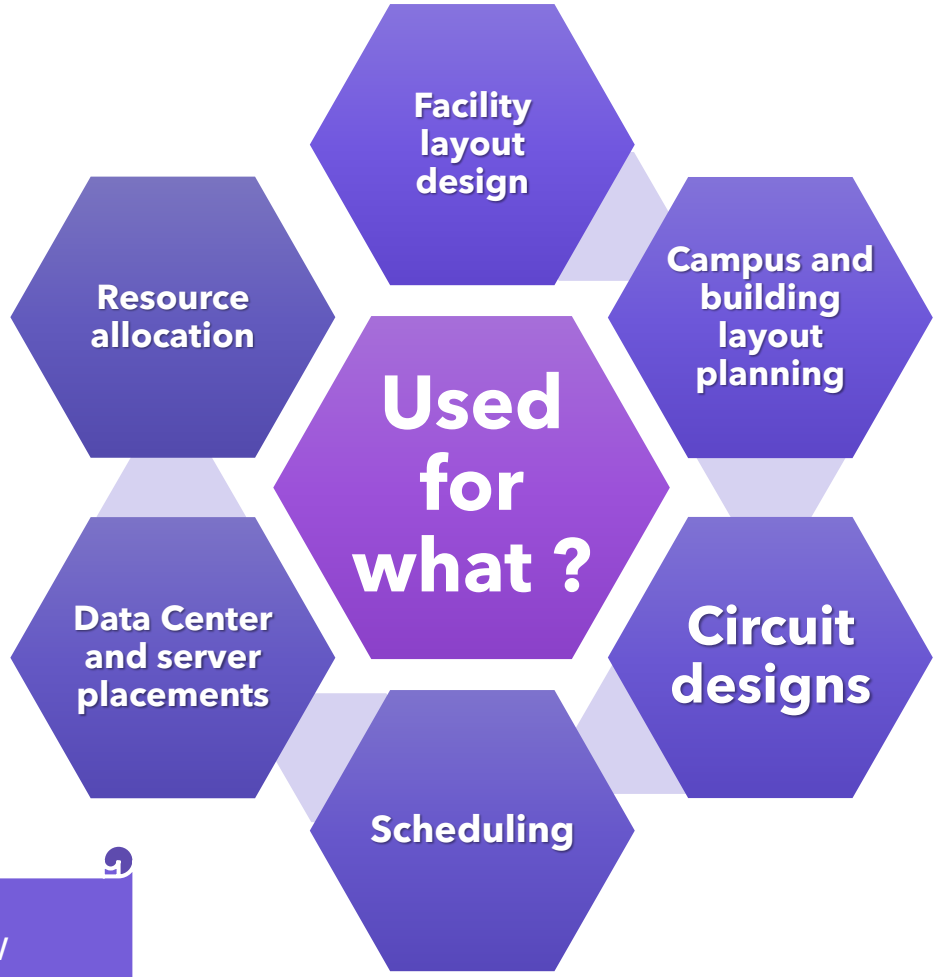
The problem is considered a **non-linear, discrete and a combinatorial optimization problem** in which a set of *n* facilities must be assigned to *n* locations on a one-to-one basis.
The objective is to **minimize the total cost**, which is *calculated as the sum of products between the flow between pairs of facilities and the distance between the locations to which those facilities are assigned*.
Because the cost depends on **pairs of assignments**, the objective function is **quadratic**.



Quadratic Assignment Problem (QAP)

Characteristics:

- One-to-one correspondence assignment between facilities and locations
- Quadratic objective function
- Discrete and Combinatorial
- Factorial growth of possible solutions (n!)
- NP-Hard Problem
- Cost structure depends on pairwise interactions

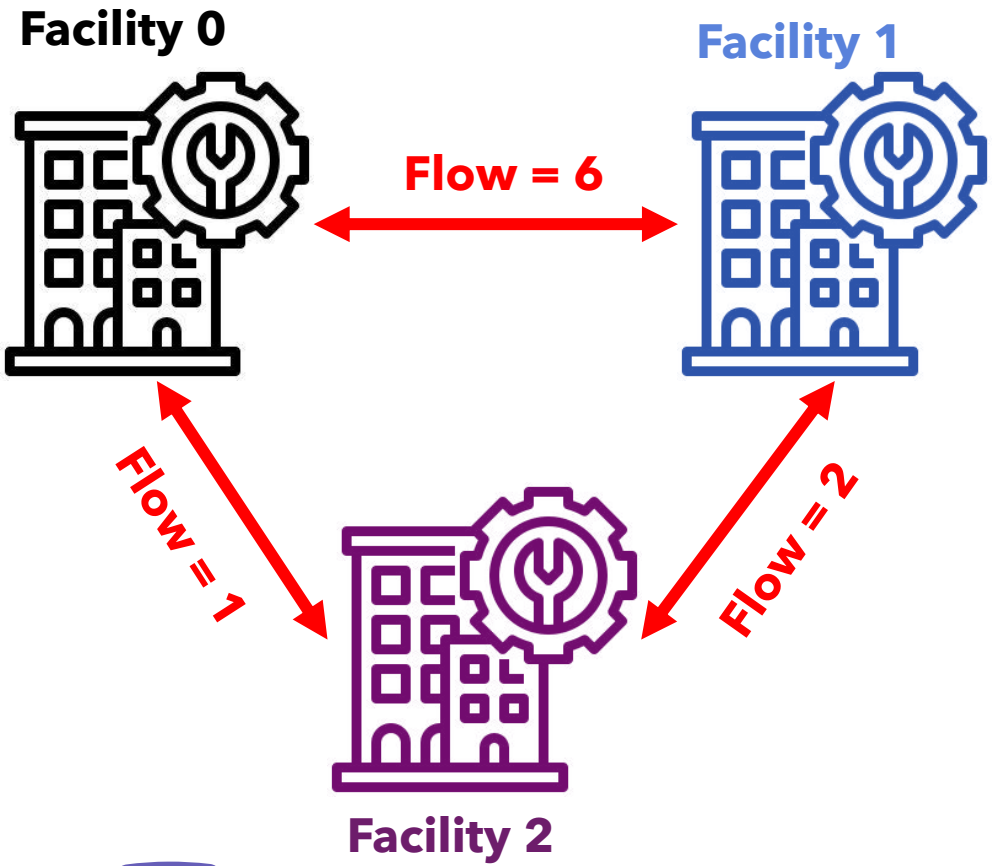


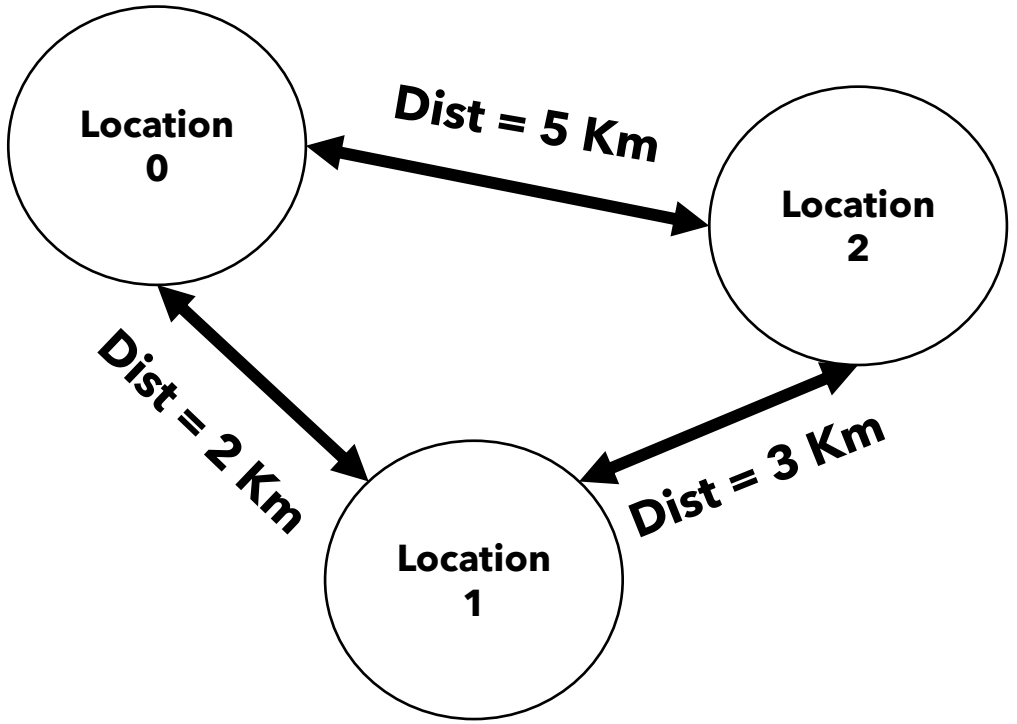
The logic core of QAP: The higher the flow between two facilities, the closer they should be

Implementation

Quadratic Assignment Problem (QAP)

Example to implement





What is the optimal assignment ?

How to represent QAP ?

Matrix Representation of Graph

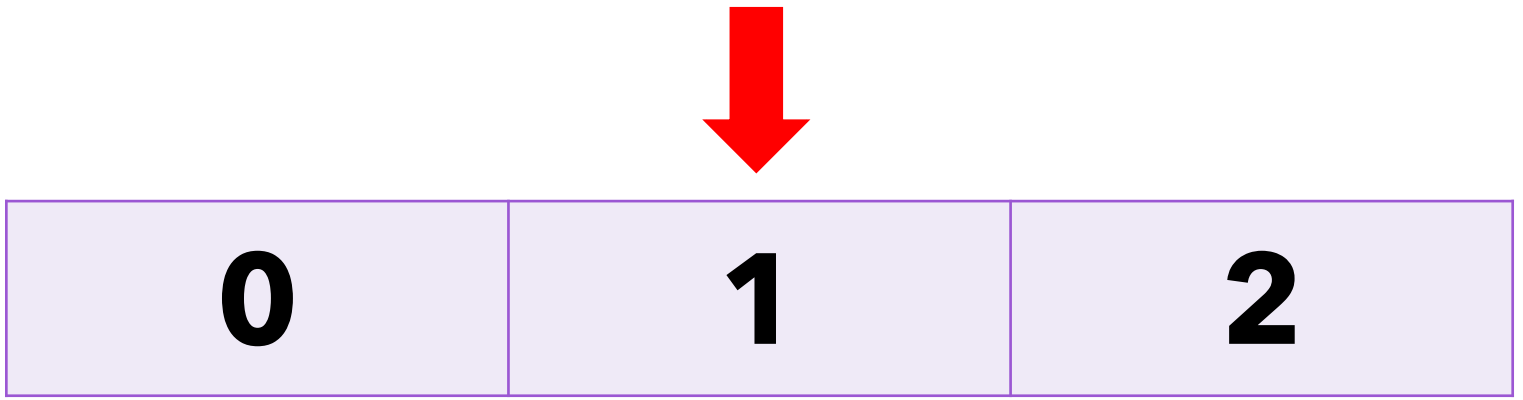
Flow Matrix	Facility 0	Facility 1	Facility 2
Facility 0	0	6	1
Facility 1	6	0	2
Facility 2	1	2	0

Distance Matrix	Location 0	Location 1	Location 2
Location 0	0	2	4
Location 1	2	0	3
Location 2	4	3	0

Quadratic class (Initialize parameters)

```python
# Represent the Quadratic Assignment Problem (QAP)
class Quadratic():
    def __init__(self, dim, flow_matrix, dist_matrix, pop_size , no_of_iterations , Q , rho , beta , alpha):
        # Initialize all parameters to be used later
        self.__flow_matrix = flow_matrix
        self.__dist_matrix = dist_matrix
        self.__facilities = [fac for fac in range(dim)] # to give a names to each facility
        self.__dim = dim # number of facilities/dimensions of the problem
        self.__initialize_parameters(pop_size , no_of_iterations, Q , rho , beta , alpha)
```

We will name facilities as numbers



Quadratic class (Initialize parameters Method)

```
#_____________________________________________________________________________________________
def __initialize_parameters(self, pop_size , no_of_iterations , Q , rho , beta , alpha):
    # To initialize the HYPERPARAMETERS of the algorithm
    self.__pop_size = pop_size
    self.__no_of_iterations = no_of_iterations
    self.__Q = Q
    self.__rho = rho
    self.__beta = beta
    self.__alpha = alpha
    self.__pheromone = np.full((self.__dim,self.__dim), fill_value=1.0) # Initialize the pheromone map
```

All hyperparameters are initialized
The pheromone map is initialized with value = 1.0
(This value can be changed)

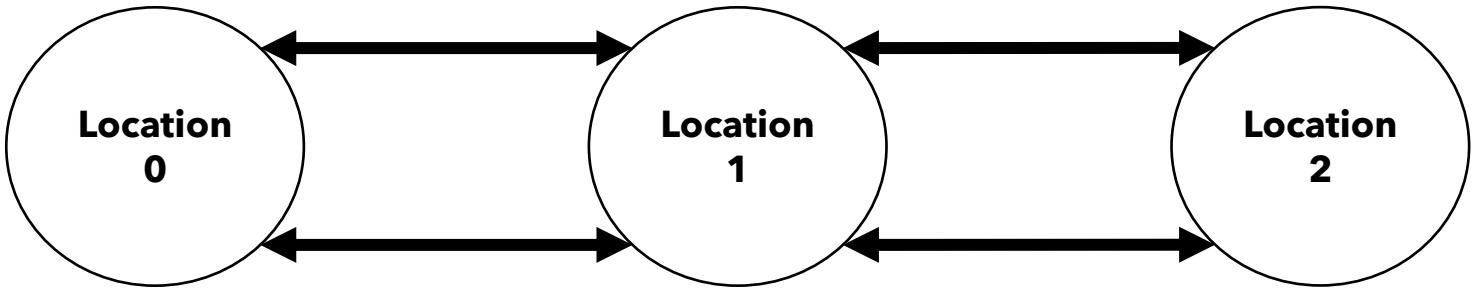
Quadratic class (Stating ants Method)

```
def __stating_ants(self):
    # State each ant in the first position (pick any initial FACILITY randomly)
    population = []
    one_ant = ['']
    for ant in range(self.__pop_size):
        ant = one_ant.copy()
        ant[0] = random.choice(self.__facilities) # pick one of the facilities randomly @ location 0
        population.append(ant)

    return population
```

Quadratic class (Stating ants Method)

```
def __stating_ants(self):
    # State each ant in the first position (pick any initial FACILITY randomly)
    population = []
    one_ant = ['']
    for ant in range(self.__pop_size):
        ant = one_ant.copy()
        ant[0] = random.choice(self.__facilities) # pick one of the facilities randomly @ location 0
        population.append(ant)

    return population
```

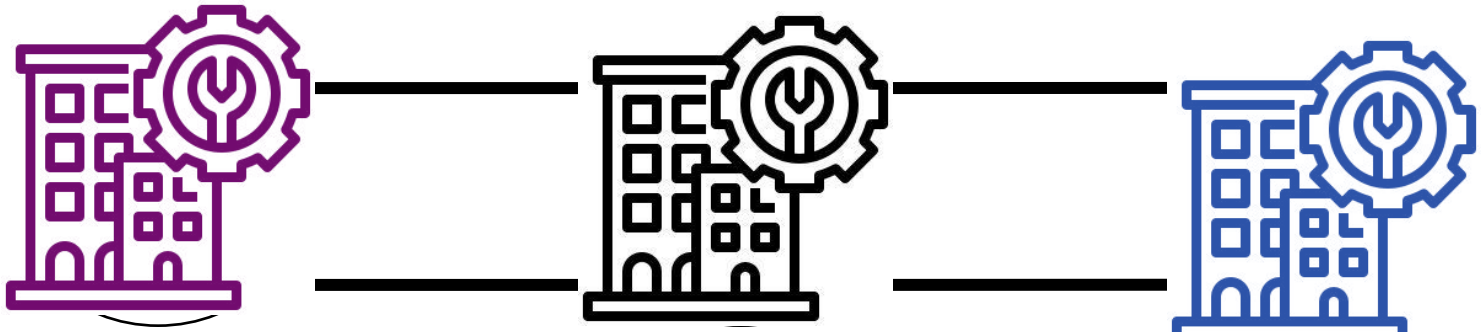
For one ant (array), each index will represent a specific location



Quadratic class (Stating ants Method)

```
def __stating_ants(self):
    # State each ant in the first position (pick any initial FACILITY randomly)
    population = []
    one_ant = ['']
    for ant in range(self.__pop_size):
        ant = one_ant.copy()
        ant[0] = random.choice(self.__facilities) # pick one of the facilities randomly @ location 0
        population.append(ant)

    return population
```

Each location can have one of the three facilities (their names are saved)



Quadratic class (Stating ants Method)

```
def __stating_ants(self):
    # State each ant in the first position (pick any initial FACILITY randomly)
    population = []
    one_ant = ['']
    for ant in range(self.__pop_size):
        ant = one_ant.copy()
        ant[0] = random.choice(self.__facilities) # pick one of the facilities randomly @ location 0
        population.append(ant)

    return population
```

The first state (facility) for each ant will be picked randomly

WHY ? In order to have a diverse combination of candidate solutions.

In TSP, it is a constraint to start from a specific position
But in QAP, the first position can be any of the facilities !

Quadratic class (Transition Method)

```python
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

The Decision Rule

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}$$

Quadratic class (Transition Method)

```python
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.rand

    return next_state
```

The desirability is computed for each possible next state

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}$$

Quadratic class (Transition Method)

```
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

Our task is to find a bijection function of facilities → locations to MINIMIZE the following:

$$\text{Cost} = \sum_{a,b \in P} w(a,b) \cdot d(f(a), f(b))$$

Quadratic class (Transition Method)

```
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

REMEMBER !

The picked facility has a connection with all facilities in the solution

Quadratic class (Transition Method)

```
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is          COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=pr          ities)

    return next_state
```

Flow of Facilities

$$\text{Cost} = \sum_{a,b \in P} w(a,b) \cdot d(f(a), f(b))$$

Distance between locations of possible facilities

Quadratic class (Transition Method)

```
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

Since the desirability is MININAL COST

$\eta = 1/(cost+1)$

Quadratic class (Transition Method)

```
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    tau = self.__pheromone[ant[-1],facility] ** self.__alpha
    cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
    probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

This is exactly for one candidate facility ONLY in the next location !

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}$$

Quadratic class (Transition Method)

```python
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

We will iterate over all possible facilities can be picked then compute the probabilities to roll the dice and pick one next state

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum\left((\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}\right)}$$

Quadratic class (Calculate fitness Method)

```
def __calculate_fitness(self, ant):
    cost = 0
    for curr_loc in range(len(ant)):
        for anoth_loc in range(len(ant)):
            # Remember that for each Facility, we will compute its cost with every other facilities (Double Summation)
            # Σi Σj Dist(i,j) x Flow(fi, fj)
            cost += self.__dist_matrix[curr_loc][anoth_loc] * self.__flow_matrix[ant[curr_loc]][ant[anoth_loc]]

    return cost
```

Since an ant represent a candidate solution (possible assignment), each index is a location, at each location there is a picked facility.

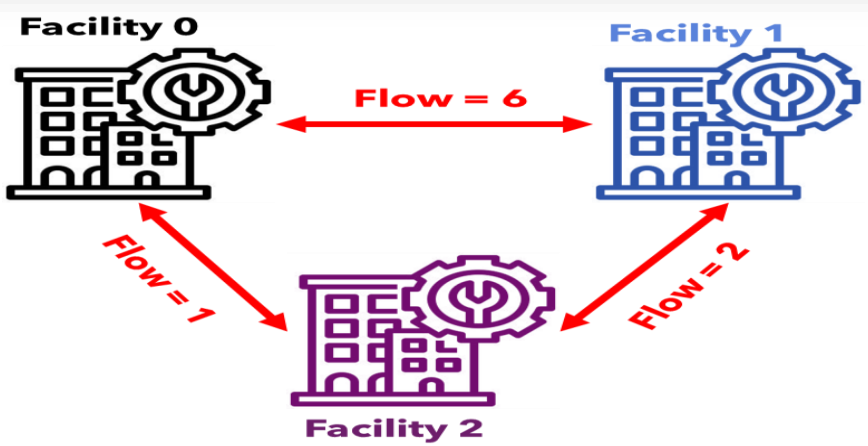
Quadratic class (Calculate fitness Method)

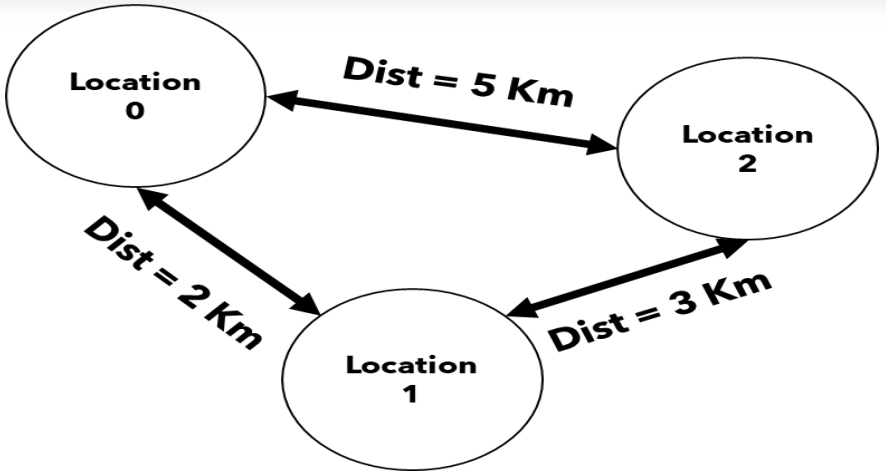
```
def __calculate_fitness(self, ant):
    cost = 0
    for curr_loc in range(len(ant)):
        for anoth_loc in range(len(ant)):
            # Remember that for each Facility, we will compute its cost with every other facilities (Double Summation)
            # Σi Σj Dist(i,j) x Flow(fi, fj)
            cost += self.__dist_matrix[curr_loc][anoth_loc] * self.__flow_matrix[ant[curr_loc]][ant[anoth_loc]]

    return cost
```

Since all facilities have connections with each other, we will iterate over each location and compute the fitness of the location and facility in it with all other facilities.

Quadratic class (Calculate fitness Method)

```python
def __calculate_fitness(self, ant):
    cost = 0
    for curr_loc in range(len(ant)):
        for anoth_loc in range(len(ant)):
            # Remember that for each Facility, we will compute its cost with every other facilities (Double Summation)
            # Σi Σj Dist(i,j) x Flow(fi, fj)
            cost += self.__dist_matrix[curr_loc][anoth_loc] * self.__flow_matrix[ant[curr_loc]][ant[anoth_loc]]

    return cost
```





Quadratic class (Update Pheromone Method)

```
def __update_pheromone(self, best_ant, fitness, count):
    self.__pheromone *= (1-self.__rho) # Evaporation
    for inx in range(len(best_ant) - 1): # [(0, 1), 2, 3] --> [0, (1, 2), 3] --> [0, 1, (2, 3)]
        self.__pheromone[best_ant[inx], best_ant[inx+1]] += (self.__Q / fitness) * count # Pheromone Reinforcement Process
```

The Learning Rule

In this implementation, we will use the BEST ANT SOLUTION ONLY to reinforce pheromone

$$\tau_{i,j}^{k} = (1-\rho)\,\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$$

Evaporation (Forgetting)

Pheromone Reinforcement process

Quadratic class (Update Pheromone Method)

```
def __update_pheromone(self, best_ant, fitness, count):
    self.__pheromone *= (1-self.__rho) # Evaporation
    for inx in range(len(best_ant) - 1): # [(0, 1), 2, 3] --> [0, (1, 2), 3] --> [0, 1, (2, 3)]
        self.__pheromone[best_ant[inx], best_ant[inx+1]] += (self.__Q / fitness) * count # Pheromone Reinforcement Process
```

Using NumPy, you don't have to iterate over the matrix, it will perform an element-wise operations

$$\tau_{i,j}^{k} = (1-\rho)\,\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$$

Evaporation (Forgetting)

Pheromone Reinforcement process

Quadratic class (Update Pheromone Method)

```python
def __update_pheromone(self, best_ant, fitness, count):
    self.__pheromone *= (1-self.__rho) # Evaporation
    for inx in range(len(best_ant) - 1): # [(0, 1), 2, 3] --> [0, (1, 2), 3] --> [0, 1, (2, 3)]
        self.__pheromone[best_ant[inx], best_ant[inx+1]] += (self.__Q / fitness) * count # Pheromone Reinforcement Process
```

Iterate over pair of facilities/locations (act as an edge) to update the corresponding one in pheromone map

$$\tau_{i,j}^{k} = (1-\rho)\,\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$$

Evaporation (Forgetting)

Pheromone Reinforcement process

Quadratic class (Update Pheromone Method)

```
def __update_pheromone(self, best_ant, fitness, count):
    self.__pheromone *= (1-self.__rho) # Evaporation
    for inx in range(len(best_ant) - 1): # [(0, 1), 2, 3] --> [0, (1, 2), 3] --> [0, 1, (2, 3)]
        self.__pheromone[best_ant[inx], best_ant[inx+1]] += (self.__Q / fitness) * count # Pheromone Reinforcement Process
```

count acts as number of ants that hold the same solution (THE BEST) so we will reinforce k count amount of pheromone on the map

$$\Delta\tau_{i,j}^{k} = \begin{cases} \dfrac{\text{Constant Amount (Q)}}{\text{Length of the Ant's Path (L)}} & \text{If ant k uses edge i,j on its path} \\ 0 & \text{,otherwise} \end{cases}$$

Ant Colony Optimization Steps

Ant Colony Optimization (ACO) Algorithm Step-by-Step

1. Initialize ACO parameters
2. Ant Solution Construction
3. Position Each ant in the stating node.
4. Each ant will select next node by applying state transition rule.
5. Repeat until ant build the best solution, then Compute the fitness value.
6. Update best solution.
7. Apply offline pheromone update.
8. Display the best result.

Quadratic class (Solution Method)

```python
def solution(self):
    # Putting all together
    # save settings during execution
    best_fitness_overall = None
    best_solution = None

    # Perform iterations of Ant Colony Optimization
    for iteration in range(self.__no_of_iterations):
        population = self.__stating_ants() # Position each ant in the beginning of iteration

        # Each ant will pick the next state until build the complete solution
        for ant in population:
            # For each ant, it will add n positions to its solution
            for pos in range(self.__dim-1):
                ant.extend(self.__transition(ant))

        # Now, compute fitness values of each ant/solution
        fitness_values = []
        for ant in population:
            fitness_values.append(self.__calculate_fitness(ant))

        # Obtain the best solution of the CURRENT ITERATION !
        best_index = fitness_values.index(min(fitness_values))
        best_ant = population[best_index]
        best_fitness = fitness_values[best_index]
        best_counts = population.count(best_ant)

        # Perform OFFLINE PHEROMONE UPDATE
        self.__update_pheromone(best_ant, best_fitness, best_counts)

        # Perform OFFLINE PHEROMONE UPDATE
        self.__update_pheromone(best_ant, best_fitness, best_counts)

        # Save the current solution if there is enhancement
        if best_fitness_overall is None or best_fitness < best_fitness_overall:
            best_fitness_overall = best_fitness
            best_solution = population[best_index]
            print(f"\ra solution found in iter: {iteration} with fitness = {best_fitness_overall}", end="")


    self.__population = population
    return best_solution, best_fitness_overall
```

Quadratic class (Solution Method)

```
def solution(self):
    # Putting all together
    # save settings during execution
    best_fitness_overall = None
    best_solution = None

    # Perform iterations of Ant Colony Optimization
    for
        population = self.__stating_ants() # Position each ant in the beginning of iteration

        # Each ant will pick the next state until build the complete solution
        for ant in population:
            # For each ant, it will add n positions to its solution
            for pos in range(self.__dim-1):
                ant.extend(self.__transition(ant))
```

Step (2): Ant Solution Construction
Step (3): Position each ant at the stating node

```
def __stating_ants(self):
    # State each ant in the first position (pick any initial FACILITY randomly)
    population = []
    one_ant = ['']
    for ant in range(self.__pop_size):
        ant = one_ant.copy()
        ant[0] = random.choice(self.__facilities) # pick one of the facilities randomly @ location 0
        population.append(ant)

    return population
```

Quadratic class (Solution Method)

```python
def solution(self):
    # Putting all together
    # save settings during execution
    best_fitness_overall = None
    best_solution = None

    # Perform iterations of Ant Colony Optimization
    for iteration in range(self.__no_of_iterations):
        population = self.__stating_ants() # Position each ant in the beginning of iteration

        # Each ant will pick the next state until build the complete solution
        for ant in population:
            # For each ant, it will add n positions to its solution
            for [illegible]:
                ant.extend(self.__transition(ant))
```

```python
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

Step (4): Each ant will select next node by applying state transition rule

Quadratic class (Solution Method)

```python
def solution(self):
    # Putting all together
    # save settings during execution
    best_fitness_overall = None
    best_solution = None

    # Perform iterations of Ant Colony Optimization
    for iteration in range(self.__no_of_iterations):
        population = self.__stating_ants() # Position each ant in the beginning of iteration

        # Each ant will pick the next state until build the complete solution
        for ant in population:
            # For each ant, it will add n positions to its solution
            for pos in range(self.__dim-1):
                ant.extend(self.__transition(ant))
```

```python
def __transition(self, ant):
    # Move each ant to the NEXT STATE (Transition Probability Model)
    probabilities = np.zeros(self.__dim)

    for facility in self.__facilities: # try all facilities to pick next
        cost = 0
        if facility in ant: # DO NOT PICK AN EXISTED FACILITY IN THE SOLUTION !
            continue
        for loc in range(len(ant)): # compute the COST of the picked facility with all facilities in the solution [Distance i,j * Flow i,j]
            cost += self.__dist_matrix[loc][len(ant)] * self.__flow_matrix[ant[loc]][facility]

        tau = self.__pheromone[ant[-1],facility] ** self.__alpha
        cost = (1/(cost+1)) ** self.__beta # The desirability is MINIMAL COST
        probabilities[facility] = tau * cost

    sum_of_probs = sum(probabilities)
    probabilities /= sum_of_probs
    next_state = np.random.choice(self.__facilities, size=1, p=probabilities)

    return next_state
```

Step (5): Repeat until ant build best solution, then compute fitness

Quadratic class (Solution Method)

```python
# Now, compute fitness values of each ant/solution
fitness_values = []
for ant in population:
    fitness_values.append(self.__calculate_fitness(ant))

# Obtain the best solution of the CURRENT ITERATION !
best_index = fitness_values.index(min(fitness_values))
best_ant = population[best_index]
best_fitness = fitness_values[best_index]
best_counts = population.count(best_ant)

# Perform OFFLINE PHEROMONE UPDATE
self.__update_pheromone(best_ant, best_fitness, best_counts)
```

```python
def __calculate_fitness(self, ant):
    cost = 0
    for curr_loc in range(len(ant)):
        for anoth_loc in range(len(ant)):
            # Remember that for each Facility, we will compute its cost with every other facilities (Double Summation)
            # Σi Σj Dist(i,j) x Flow(fi, fj)
            cost += self.__dist_matrix[curr_loc][anoth_loc] * self.__flow_matrix[ant[curr_loc]][ant[anoth_loc]]

    return cost
```

Step (5): Repeat until ant build best solution, then compute fitness

Quadratic class (Solution Method)

```python
# Obtain the best solution of the CURRENT ITERATION !
best_index = fitness_values.index(min(fitness_values))
best_ant = population[best_index]
best_fitness = fitness_values[best_index]
best_counts = population.count(best_ant)

# Perform OFFLINE PHEROMONE UPDATE
self.__update_pheromone(best_ant, best_fitness, best_counts)

# Save the current solution if there is enhancement
if best_fitness_overall is None or best_fitness < best_fitness_overall:
    best_fitness_overall = best_fitness
    best_solution = population[best_index]
    print(f"\ra solution found in iter: {iteration} with fitness = {best_fitness_overall}", end="")
```

Step (6): Update best solution
Here in (QAP), target is MINIMIZE

Quadratic class (Solution Method)

```
# Obtain the best solution of the CURRENT ITERATION !
best_index = fitness_values.index(min(fitness_values))
best_ant = population[best_index]
best_fitness = fitness_values[best_index]
best_counts = population.count(best_ant)

# Perform OFFLINE PHEROMONE UPDATE
self.__update_pheromone(best_ant, best_fitness, best_counts)

# Save the current solution if there is enhancement
```

```
def __update_pheromone(self, best_ant, fitness, count):
    self.__pheromone *= (1-self.__rho) # Evaporation
    for inx in range(len(best_ant) - 1): # [(0, 1), 2, 3] --> [0, (1, 2), 3] --> [0, 1, (2, 3)]
        self.__pheromone[best_ant[inx], best_ant[inx+1]] += (self.__Q / fitness) * count # Pheromone Reinforcement Process
```

Step (7): Apply Offline pheromone update (Based on Best ant of all)

Quadratic class (Solution Method)

```
    # Save the current solution if there is enhancement
    if best_fitness_overall is None or best_fitness < best_fitness_overall:
        best_fitness_overall = best_fitness
        best_solution = population[best_index]
        print(f"\ra solution found in iter: {iteration} with fitness = {best_fitness_overall}", end="")


self.__population = population
return best_solution, best_fitness_overall
```

Step (8): Display Result

```
flow_matrix = np.array([[0,6,1],[6,0,2],[1,2,0]])
dist_matrix = np.array([[0,2,4],[2,0,3],[4,3,0]])
quad = Quadratic(dim= 3,flow_matrix=flow_matrix, dist_matrix=dist_matrix, pop_size=20, no_of_iterations=1000, Q=3, rho=0.3, beta=1, alpha=0.5)
sol = quad.solution()
print(f"\nThe optimal assignment is ({sol[0]}) with Total cost = {sol[1]}")
```

```
a solution found in iter: 0 with fitness = 44
The optimal assignment is ([0, 1, 2]) with Total cost = 44
```

TESTING

Let's see if it is true answer ?

Since the dimension of the problem is 3
The growth of the problem is 3! = 6 combinations

Our Solution: [0,1,2]

Let's see if it is true answer ?

Since the dimension of the problem is 3
The growth of the problem is 3! = 6 combinations

Permutation	Cost
[0,1,2]	44
[0,2,1]	64
[1,0,2]	44
[1,2,0]	52
[2,0,1]	64
[2,1,0]	60

Our Solution: [0,1,2]

Let's see if it is true answer ?

Since the dimension of the problem is 3
The growth of the problem is 3! = 6 combinations

Permutation	Cost
[0,1,2]	44
[0,2,1]	64
[1,0,2]	44
[1,2,0]	52
[2,0,1]	64
[2,1,0]	60

Our Solution: [0,1,2

Optimal Solution

Quadratic Assignment Problem (QAP)

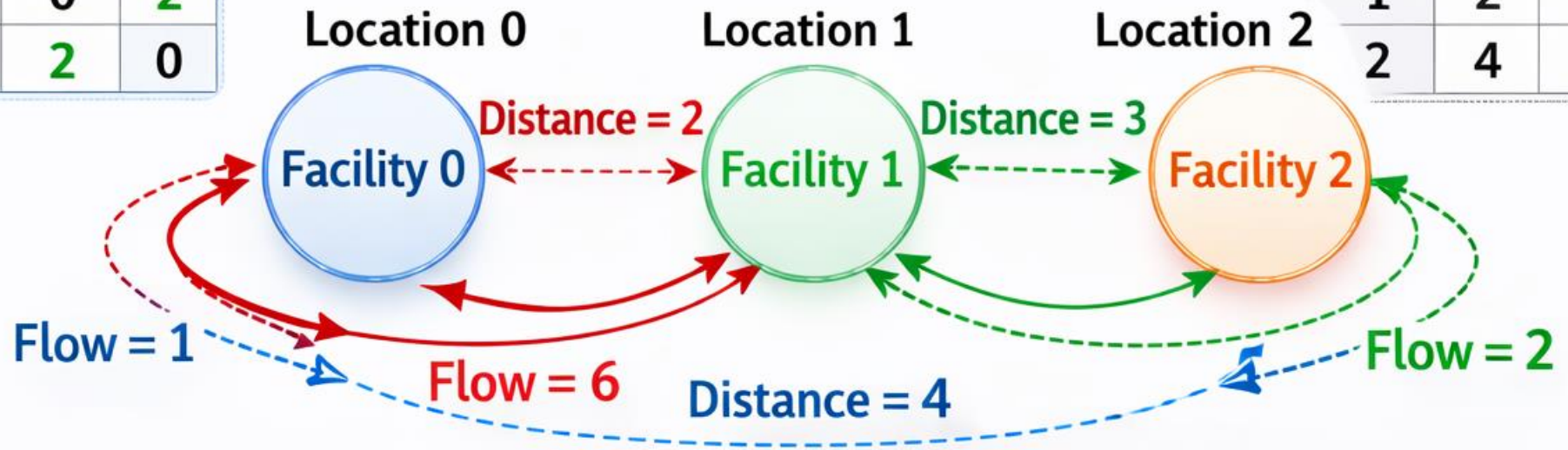
Optimal Assignment [0,1,2]

Flow Matrix F

	0	1	2
0	0	6	1
1	6	0	2
2	1	2	0

Distance Matrix D

	0	1	2
0	0	2	4
1	2	0	3
2	4	3	0



Flow × Distance Contributions

$F_0 \leftrightarrow F_1$: $6 \times 2 = 12$

$F_0 \leftrightarrow F_2$: $1 \times 4 = 4$

$F_1 \leftrightarrow F_0$: $6 \times 2 = 12$

Total Cost

$12 + 4 + 12 + 6 + 4 + 6 =$

44

Legend:

- Facility 0
- Facility 1
- Facility 2 <=> Distance 2
- ==> Distance 3 → Distance 4

Behavior of Ants and Final Pheromone Map

Convergent

```
[4]: (array([[1, 2, 0],
             [1, 2, 0],
             [1, 2, 0],
             [2, 1, 0],
             [1, 2, 0],
             [0, 1, 2],
             [0, 1, 2],
             [2, 1, 0],
             [1, 2, 0],
             [1, 2, 0],
             [2, 1, 0],
             [0, 1, 2],
             [0, 1, 2],
             [1, 2, 0],
             [1, 2, 0],
             [0, 1, 2],
             [1, 2, 0],
             [2, 1, 0],
             [2, 1, 0],
             [0, 1, 2]],
            array([[1.25325664e-155, 1.44649273e+000, 5.79026735e-154],
                   [1.36053007e-153, 1.25325664e-155, 1.44649273e+000],
                   [1.25325664e-155, 7.94035903e-154, 1.25325664e-155]]))
```

Thank You

Next Lab:
Artificial Bee Colony (Theoretical)