Job Scheduling in Cloud Environments using Computational Intelligence

Introduction

Cloud computing is widely used today to store data, run applications, and provide services to users around the world. One of the most important tasks in cloud computing is **job scheduling**. This means deciding which task should run on which server, and when it should run.

If scheduling is not done well, some servers may become overloaded while others stay idle. This can lead to longer waiting times, slower performance, and wasted resources. So, finding a smart way to schedule jobs is very important.

In this project, we use **Computational Intelligence (CI)** techniques to improve job scheduling. One of the most powerful CI methods is **Ant Colony Optimization (ACO)**. ACO is inspired by the behavior of real ants when they search for food. Ants use a chemical called pheromone to mark good paths, and over time, more ants follow the best path.

We apply ACO and several versions of it to schedule tasks in a cloud environment. These versions include:

- **AS** – Ant System
- **ACS** – Ant Colony System
- **EAS** – Elitist Ant System
- **MMAS** – Max-Min Ant System
- **ASrank** – Rank-Based Ant System
- **ACO-GA** – A hybrid of ACO and Genetic Algorithm

Each of these algorithms helps find better ways to assign tasks to cloud servers. We simulate a cloud environment and compare the performance of these algorithms based on criteria like waiting time and load balancing.

The goal of this project is to **improve cloud job scheduling** using intelligent algorithms that mimic natural behavior, leading to faster and more efficient cloud systems.

Algorithm 1: Ant System (AS) Scheduler

The **Ant System (AS)** is a classical optimization algorithm inspired by the foraging behavior of ants. In our context, the AS algorithm is used to schedule tasks efficiently across multiple cloud computing nodes.

Each **ant** builds a solution by assigning tasks to computing nodes. The decision is based on two main factors:

- **Pheromone information** (how good a past solution was),
- **Heuristic information** (how suitable the node is for the task now).

Mathematical Model

For each task Ti and node Nj, the probability that an ant assigns the task to that node is calculated using:

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\sum_{k \in \text{valid nodes}} (\tau_{ik})^{\alpha} \cdot (\eta_{ik})^{\beta}}$$

Where:

- τ_{ij} is the **pheromone level** between task T_i and node N_j,
- η_{ij} is the **heuristic value**, computed as:

$$\eta_{ij} = \frac{1}{\text{ExecutionTime}_{ij} \cdot (1 + \text{Load}_j)}$$

- α controls the influence of pheromones,
- β controls the influence of heuristic values.

Fitness Function

The **fitness** of a solution is calculated based on three factors:

$$\text{Fitness} = \frac{1}{\text{Makespan} + \varepsilon} \cdot \frac{1}{\sigma + \varepsilon} \cdot \frac{1}{\text{AverageWait} + \varepsilon}$$

Where:

- **Makespan**: Total time of the most loaded node,
- σ: Standard deviation of node utilizations (used for load balancing),
- **AverageWait**: Average wait time for task execution,
- ε: A small value to prevent division by zero.

Pheromone Update Rule

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \sum_{\text{ants}} \Delta\tau_{ij}^{\text{ant}}$$

Where:

- ρ is the evaporation rate,
- $\Delta\tau_{ij}^{\text{ant}} = q \cdot \text{Fitness}$ if the ant used edge (i, j),
- q is a constant.

Main Steps:

1. **Initialization**:
   - Set the number of ants, iterations, and parameters (pheromone importance alpha, heuristic importance beta, evaporation rate rho, and pheromone update constant q).
   - Initialize pheromone values for each task-node pair.
2. **Task Assignment**:
   - For each ant, tasks are assigned to nodes using a probability rule that combines pheromone strength and a heuristic function.
   - The heuristic is based on the **execution time** and **current load** of each node.
3. **Fitness Evaluation**:
   - After assigning all tasks, the algorithm calculates:
     - **Makespan** (the time taken to complete all tasks),
     - **Average waiting time**, and
     - **Load balance** (based on CPU utilization).

    - A fitness score is computed using these values.

4. **Pheromone Update**:
    - After all ants complete their assignments, the pheromone matrix is updated.
    - Better solutions (with higher fitness) contribute more to the pheromone increase.
    - Pheromones also evaporate slightly in each iteration.
5. **Repeat**:
    - The process is repeated for a fixed number of iterations.
    - The best solution found so far is kept and returned at the end.

Code Summary:

- assign_tasks() assigns tasks based on probabilistic decision rules.
- calculate_makespan() computes the total time needed by the most loaded node.
- calculate_fitness() combines makespan, load balance, and wait time into one score.
- update_pheromone() adjusts the pheromone levels based on ant performance.
- run() manages the entire optimization loop and keeps track of the best schedule.

Algorithm 2: Ant Colony Optimization (ACO) for Task Scheduling

Ant Colony Optimization (ACO) is a smart technique inspired by how ants find the shortest paths. In task scheduling, we use ACO to assign jobs (tasks) to computers (nodes) efficiently. Each ant simulates a possible assignment, and through learning (pheromone trails), the best ways are reinforced over time.

Mathematical Model

Goal: Encourage assignment to fast and lightly loaded nodes.

$$\eta_{ij} = \frac{1}{\left(\frac{D_i}{CPU_j+\epsilon}\right) \cdot \left(\frac{1}{U_j+\epsilon}\right)}$$

- D_i: Task i's duration
- CPU_j: Available CPU on node j
- U_j: Utilization of node j
- ϵ: Small value to avoid divide by zero

```python
def calculate_heuristic(self, task, node):
    exec_time = task.duration / (node.available_cpu + 0.001)
    load_balance = 1 / (node.calculate_utilization() + 0.001)
    return 1 / (exec_time * load_balance)
```

Use both pheromone strength and heuristic info:

$$P_{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{k \in N_i} [\tau_{ik}]^\alpha \cdot [\eta_{ik}]^\beta}$$

- τ_{ij}: Pheromone on assigning task i to node j
- η_{ij}: Heuristic info
- α, β: Control importance of pheromone vs heuristic

```python
def calculate_probability(self, task_idx, node_idx):
    pheromone = self.pheromone[task_idx][node_idx] ** self.alpha
    heuristic = self.calculate_heuristic(self.tasks[task_idx], self.nodes[node_idx]) ** self.beta
    return pheromone * heuristic
```

<u>Pheromone Update (Exploitation and Exploration):</u>

Evaporation:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij}$$

Reinforcement:

$$\tau_{ij} = \tau_{ij} + \rho \cdot \Delta\tau_{ij}$$

- ρ: Evaporation rate
- $\Delta\tau_{ij}$: Reinforcement term (e.g., constant or fitness-based)

```python
def update_pheromone(self, solutions):
    self.pheromone *= (1 - self.rho)
    for solution in solutions:
        fitness = solution['fitness']
        for task_idx, node_idx in solution['assignments']:
            self.pheromone[task_idx][node_idx] += self.q * fitness
```

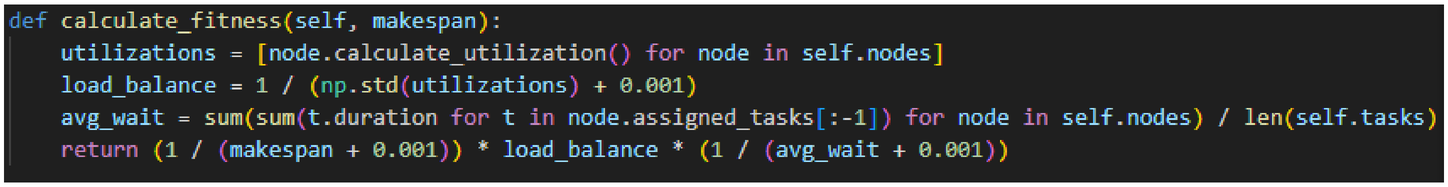
```python
self.pheromone[self.tasks.index(task)][selected_idx] *= (1 - self.rho)
self.pheromone[self.tasks.index(task)][selected_idx] += (self.rho * 0.1)
```

<u>Fitness Function</u>

Goal: Maximize task balance and minimize time.

$$Fitness = \left(\frac{1}{Makespan + \epsilon}\right) \cdot \left(\frac{1}{\sigma(U) + \epsilon}\right) \cdot \left(\frac{1}{AvgWait + \epsilon}\right)$$

- $\sigma(U)$: Std. dev of utilization
- $AvgWait$: Average wait time for tasks



```
def calculate_fitness(self, makespan):
    utilizations = [node.calculate_utilization() for node in self.nodes]
    load_balance = 1 / (np.std(utilizations) + 0.001)
    avg_wait = sum(sum(t.duration for t in node.assigned_tasks[:-1]) for node in self.nodes) / len(self.tasks)
    return (1 / (makespan + 0.001)) * load_balance * (1 / (avg_wait + 0.001))
```

Algorithm 3: Elitist Ant System (EAS) Task Scheduling

Task scheduling in cloud computing environments is a combinatorial optimization problem aimed at minimizing execution time (makespan), improving load balance, and maximizing resource utilization. Ant Colony Optimization (ACO), a metaheuristic inspired by the foraging behavior of ants, has proven effective in solving such problems. The Elitist Ant System (EAS), an enhancement over classical ACO, introduces an elitist component that reinforces the best solution found so far more aggressively.

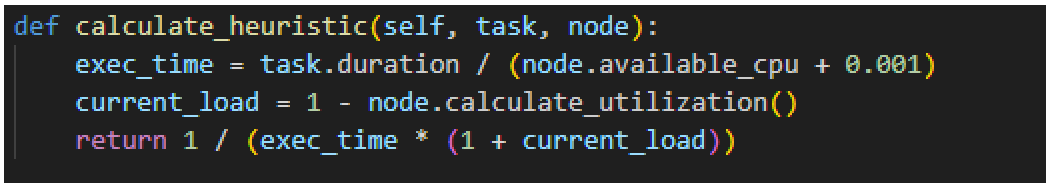
<u>Mathematical Model</u>:

Heuristic Information

For each task T_i and node N_j, a heuristic value η_{ij} is calculated as:

$$\eta_{ij} = \frac{1}{\left(\frac{t_i}{C_j+\epsilon}\right) \cdot (1 + (1 - U_j))}$$

- t_i: duration of task i
- C_j: available CPU on node j
- U_j: current CPU utilization of node j
- ϵ: small value to prevent division by zero



```
def calculate_heuristic(self, task, node):
    exec_time = task.duration / (node.available_cpu + 0.001)
    current_load = 1 - node.calculate_utilization()
    return 1 / (exec_time * (1 + current_load))
```

Transition Probability

The probability that an ant assigns task T_i to node N_j is given by:

$$P_{ij} = \frac{(\tau_{ij})^\alpha \cdot (\eta_{ij})^\beta}{\sum_{k \in V_i} (\tau_{ik})^\alpha \cdot (\eta_{ik})^\beta}$$

- τ_{ij}: pheromone level on the edge between T_i and N_j
- α, β: parameters controlling importance of pheromone vs. heuristic
- V_i: set of valid nodes for task T_i

```
def calculate_probability(self, task_idx, node_idx):
    pheromone = self.pheromone[task_idx][node_idx] ** self.alpha
    heuristic = self.calculate_heuristic(self.tasks[task_idx], self.nodes[node_idx]) ** self.beta
    return pheromone * heuristic
```

Pheromone Update

After each iteration, pheromone levels are updated as:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k + e \cdot \Delta\tau_{ij}^{best}$$

- ρ: evaporation rate
- $\Delta\tau_{ij}^k = \frac{Q}{makespan_k}$: contribution from ant k
- e: elitist factor
- $best$: best solution found so far

```
def update_pheromone(self, solutions, best_solution):
    self.pheromone *= (1 - self.rho)

    for solution in solutions:
        for task_idx, node_idx in solution['assignments']:
            self.pheromone[task_idx][node_idx] += self.q / solution['makespan']

    for task_idx, node_idx in best_solution['assignments']:
        self.pheromone[task_idx][node_idx] += self.e * self.q / best_solution['makespan']
```

Algorithm 4: Max-Min Ant System (MMAS) for Task Scheduling

The **MMAS** is a combinatorial optimization algorithm that applies principles of Ant Colony Optimization (ACO) but with modifications. It focuses on finding the optimal task-node assignment in a scheduling problem.

Mathematical Concepts

1. **Heuristic Calculation**:
   The **heuristic** value for a task-node pair is determined by considering both execution time and load balancing. For task tit_iti and node njn_jnj, the heuristic is given by:

$$\text{Heuristic}(t_i, n_j) = \frac{1}{\text{Execution Time} \times (1 + \text{Load Balance})}$$

   Where:

- **Execution Time** is the ratio of task duration over available CPU resources of the node:

$$\text{Execution Time} = \frac{\text{duration}(t_i)}{\text{available CPU}(n_j) + \epsilon}$$

  (where ϵ is a small constant to avoid division by zero).

- **Load Balance** is the inverse of the node's utilization:

$$\text{Load Balance} = \frac{1}{\text{Utilization}(n_j) + \epsilon}$$

  where **Utilization** refers to the node's current resource usage.

```
# Calculate heuristic value for task-node pair
def calculate_heuristic(self, task, node):

    # Combine execution time and load balance factors
    exec_time = task.duration / (node.available_cpu + 0.001)
    load_balance = 1 / (node.calculate_utilization() + 0.001)
    return 1 / (exec_time * (1 + load_balance))
```

2. **Task Assignment Probability**:
   In MMAS, ants decide which node to assign each task based on a **probability** distribution that considers both pheromone levels and heuristic values. The probability P(ti,nj) of assigning task tit_iti to node nj is given by:

$$P(t_i, n_j) = \frac{[\text{pheromone}(t_i, n_j)]^\alpha \times [\text{Heuristic}(t_i, n_j)]^\beta}{\sum_{k \in \text{valid nodes}} [\text{pheromone}(t_i, n_k)]^\alpha \times [\text{Heuristic}(t_i, n_k)]^\beta}$$

- α and β control the relative importance of pheromone and heuristic information.
- The sum in the denominator ensures the probabilities for all valid node assignments sum to 1.

3. **Pheromone Update (Global):**

After each iteration, the pheromone values are updated using the **global pheromone update rule**. The pheromone on the task-node pair (t_i, n_j) is updated as follows:

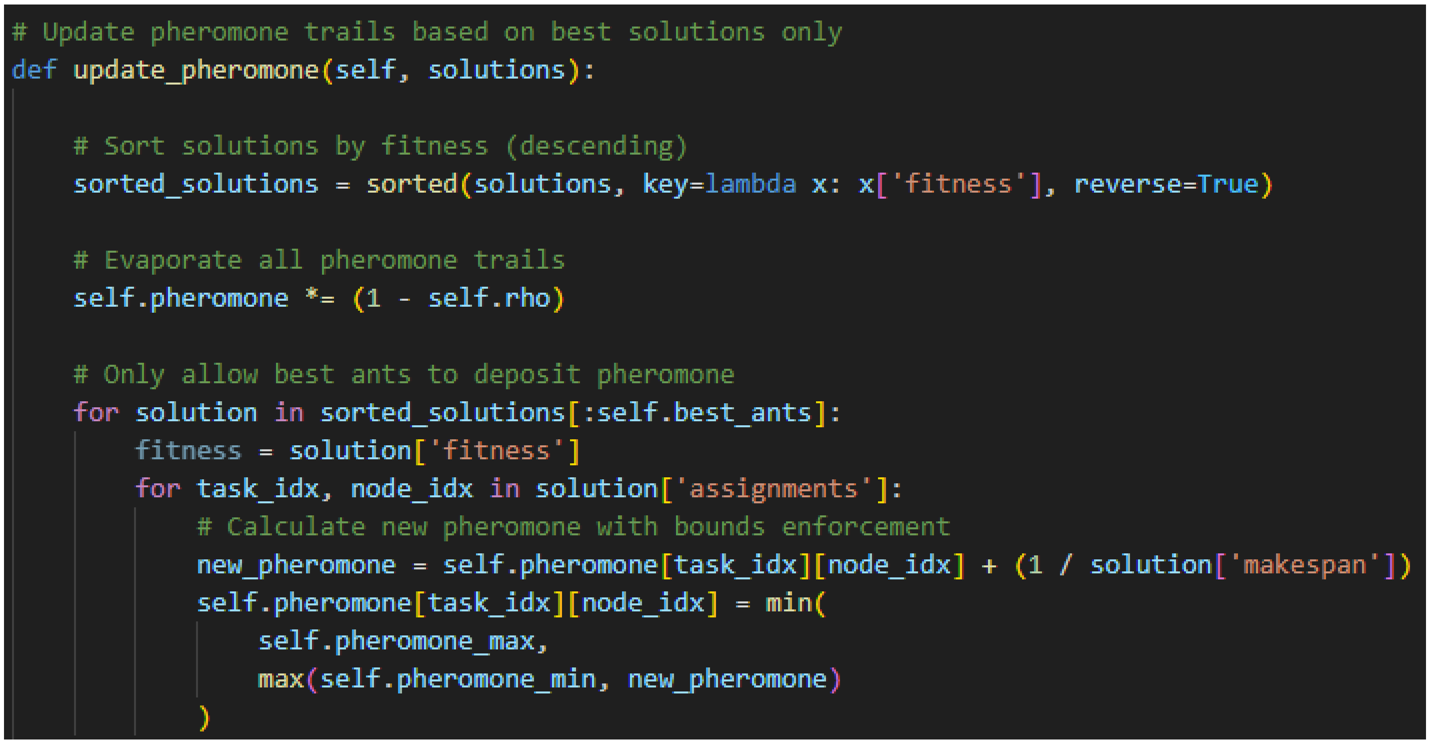
$$\text{pheromone}(t_i, n_j) = (1 - \rho) \times \text{pheromone}(t_i, n_j) + \rho \times \Delta\text{pheromone}(t_i, n_j)$$

Where:

- ρ is the evaporation rate.
- $\Delta\text{pheromone}(t_i, n_j)$ is the pheromone deposit from the best ant solution:

$$\Delta\text{pheromone}(t_i, n_j) = \frac{1}{\text{Makespan}}$$

(lower makespan implies a better solution, so the pheromone deposit is higher for better solutions).



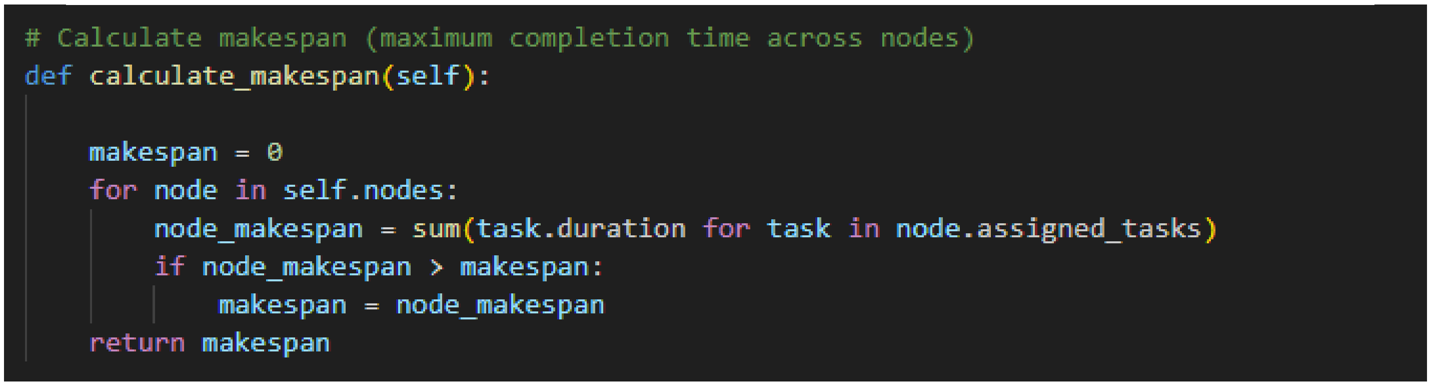
```python
# Update pheromone trails based on best solutions only
def update_pheromone(self, solutions):

    # Sort solutions by fitness (descending)
    sorted_solutions = sorted(solutions, key=lambda x: x['fitness'], reverse=True)

    # Evaporate all pheromone trails
    self.pheromone *= (1 - self.rho)

    # Only allow best ants to deposit pheromone
    for solution in sorted_solutions[:self.best_ants]:
        fitness = solution['fitness']
        for task_idx, node_idx in solution['assignments']:
            # Calculate new pheromone with bounds enforcement
            new_pheromone = self.pheromone[task_idx][node_idx] + (1 / solution['makespan'])
            self.pheromone[task_idx][node_idx] = min(
                self.pheromone_max,
                max(self.pheromone_min, new_pheromone)
            )
```

4. **Makespan Calculation**:

The **makespan** is the maximum completion time of the tasks across all nodes. It is given by:

$$\text{Makespan} = \max_{j \in \text{nodes}} \left(\sum_{t_i \in n_j} \text{duration}(t_i) \right)$$

where the sum represents the total processing time of tasks assigned to node n_j.



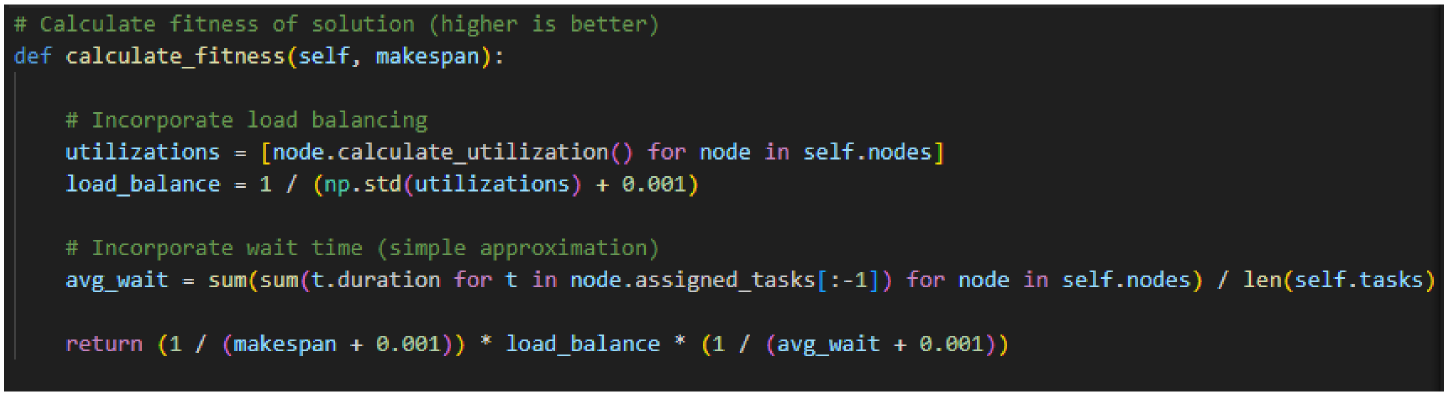
```python
# Calculate makespan (maximum completion time across nodes)
def calculate_makespan(self):

    makespan = 0
    for node in self.nodes:
        node_makespan = sum(task.duration for task in node.assigned_tasks)
        if node_makespan > makespan:
            makespan = node_makespan
    return makespan
```

5. **Fitness Calculation**:

The **fitness** of a solution is computed considering:

- **Makespan** (minimize),
- **Load balancing** (maximize),
- **Average wait time** (minimize).

$$\text{fitness} = \frac{1}{\text{Makespan} + \epsilon} \times \text{Load Balance} \times \frac{1}{\text{Avg Wait Time} + \epsilon}$$



```python
# Calculate fitness of solution (higher is better)
def calculate_fitness(self, makespan):

    # Incorporate load balancing
    utilizations = [node.calculate_utilization() for node in self.nodes]
    load_balance = 1 / (np.std(utilizations) + 0.001)

    # Incorporate wait time (simple approximation)
    avg_wait = sum(sum(t.duration for t in node.assigned_tasks[:-1]) for node in self.nodes) / len(self.tasks)

    return (1 / (makespan + 0.001)) * load_balance * (1 / (avg_wait + 0.001))
```

6. **Pheromone Bounds**:

In MMAS, pheromone values are constrained within the range $[\text{pheromone\_min}, \text{pheromone\_max}]$. This prevents runaway pheromone growth and ensures the algorithm remains balanced:

$$\text{pheromone}(t_i, n_j) = \max(\text{pheromone\_min}, \min(\text{pheromone\_max}, \text{pheromone}(t_i, n_j)))$$

Key Steps in MMAS Algorithm:

1. **Initialize pheromone matrix**: Set initial pheromone values to a high value (e.g., pheromone_max\text{pheromone_max}pheromone_max).
2. **Ant Exploration**: Each ant constructs a solution by assigning tasks to nodes based on the pheromone matrix and the calculated heuristics.
3. **Evaluate Solutions**: After all ants have assigned tasks, evaluate the solutions by calculating the **makespan**, **fitness**, and updating the pheromone matrix for the best solutions.
4. **Global pheromone update**: Update pheromone values based on the best ants' solutions, reinforcing good task-node assignments.
5. **Iteration**: Repeat the process for a specified number of iterations or until convergence.

Algorithm 5: ASrank-Based Task Scheduling

This algorithm schedules a set of tasks onto a set of nodes (resources) using a modified Ant Colony Optimization (ACO) called **ASrank**, which ranks ant solutions and gives more weight to better ones during pheromone updates.

Mathematical Concepts:

1. **Probability of Assigning Task t_i to Node n_j**:

$$P_{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{k\in \text{valid nodes}} [\tau_{ik}]^\alpha \cdot [\eta_{ik}]^\beta}$$

Where:

- τ_{ij} is the **pheromone** level for assigning task `i` to node `j`.
- η_{ij} is the **heuristic desirability** of assigning task `i` to node `j`.
- α and β control the influence of pheromone and heuristic information, respectively.

2. **Heuristic Desirability** η_{ij}:

$$\eta_{ij} = \frac{1}{\text{execution\_time}_{ij} \cdot (1 + \text{load\_imbalance}_{j})}$$

- $\text{execution\_time}_{ij} = \frac{t_i.\text{duration}}{n_j.\text{available\_cpu} + \varepsilon}$
- $\text{load\_imbalance}_{j} = \frac{1}{n_j.\text{utilization} + \varepsilon}$

Where ε is a small constant (e.g., 0.001) to avoid division by zero.

```
# Calculate heuristic desirability of assigning task to node
def calculate_heuristic(self, task, node):

    # Consider both execution time and load balancing
    exec_time = task.duration / (node.available_cpu + 0.001)
    load_balance = 1 / (node.calculate_utilization() + 0.001)
    return 1 / (exec_time * (1 + load_balance))
```

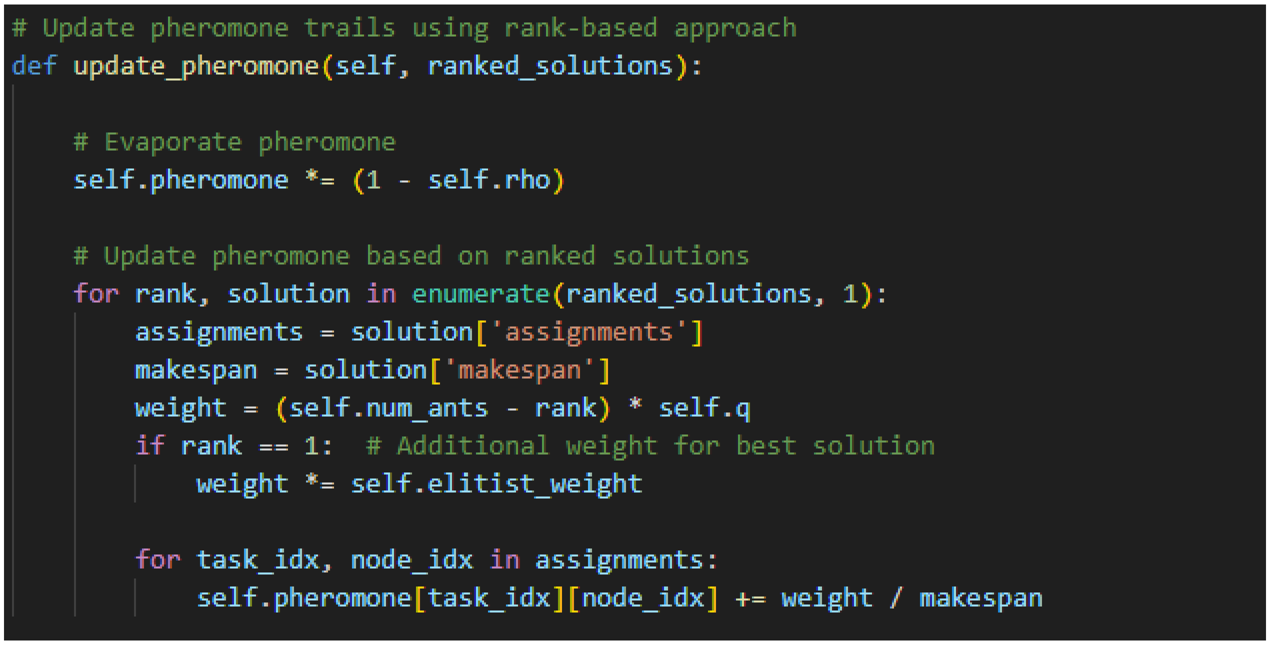
3. **Pheromone Update (ASrank)**:

Each iteration, pheromones are updated as:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \sum_{r=1}^{k} w_r \cdot \frac{Q}{\text{makespan}_r}$$

Where:

- ρ is the evaporation rate.
- Q is a constant pheromone deposit amount.
- w_r is the **rank-based weight**: $w_r = k - r$
- makespan_r is the makespan of the solution of the r^{th} ranked ant.
- k is the elitist weight or number of top solutions to consider.

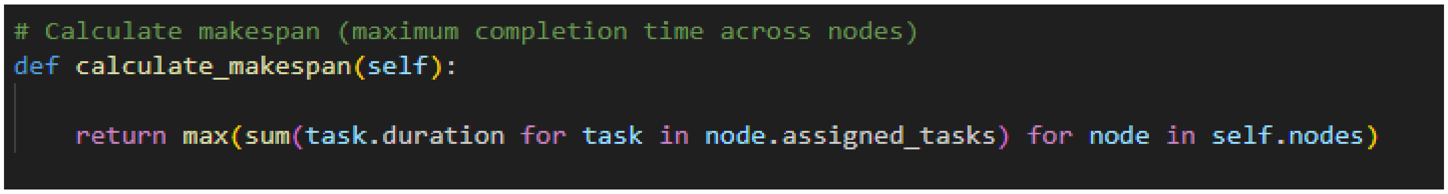


```python
# Update pheromone trails using rank-based approach
def update_pheromone(self, ranked_solutions):

    # Evaporate pheromone
    self.pheromone *= (1 - self.rho)

    # Update pheromone based on ranked solutions
    for rank, solution in enumerate(ranked_solutions, 1):
        assignments = solution['assignments']
        makespan = solution['makespan']
        weight = (self.num_ants - rank) * self.q
        if rank == 1:  # Additional weight for best solution
            weight *= self.elitist_weight

        for task_idx, node_idx in assignments:
            self.pheromone[task_idx][node_idx] += weight / makespan
```

4. Makespan:

$$\text{makespan} = \max_{j \in \text{nodes}} \sum_{t \in n_j.\text{assigned\_tasks}} t.\text{duration}$$

This is the total time until all tasks are completed—i.e., the **maximum node load.**



```python
# Calculate makespan (maximum completion time across nodes)
def calculate_makespan(self):

    return max(sum(task.duration for task in node.assigned_tasks) for node in self.nodes)
```

Algorithm 6: Hybrid ACO-GA Scheduler

The **ACO-GA Scheduler** is a hybrid optimization algorithm combining **Ant Colony Optimization (ACO)** and **Genetic Algorithm (GA)** for efficient **task scheduling** in a distributed computing environment. It assigns tasks to nodes (resources) by simulating the intelligent behavior of ants (ACO) and the evolutionary strategies of biological organisms (GA).

This hybrid algorithm uses:

- **ACO (Ant Colony Optimization)** to explore promising solutions using pheromone trails and heuristics.
- **GA (Genetic Algorithm)** to further exploit and refine solutions found by ACO, promoting diversity and avoiding local optima.

Mathematical Foundations

ACO Components

Each ant builds a solution by selecting nodes for each task based on:

- **Pheromone intensity** τij
- **Heuristic desirability** ηij

Heuristic Function

The heuristic ηij for assigning task iii to node j is:

$$\eta_{ij} = \frac{1}{T_{ij} \cdot L_j}$$

Where:

- $T_{ij} = \frac{\text{task.duration}}{\text{node.available\_cpu} + \epsilon}$ is the estimated execution time.
- $L_j = \frac{1}{\text{node.utilization} + \epsilon}$ represents the inverse load (to encourage load balancing).
- $\epsilon = 0.001$ avoids division by zero.

Probability of Assignment

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\sum_{k \in \text{valid}} (\tau_{ik})^{\alpha} \cdot (\eta_{ik})^{\beta}}$$

Where:

- α controls pheromone influence.
- β controls heuristic influence.

Pheromone Update

Pheromone evaporation and reinforcement:

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \rho \cdot \Delta\tau_{ij}$$

Where:

- ρ is the evaporation rate.
- $\Delta\tau_{ij}$ is the reward (e.g., based on fitness or a constant for local updates).

GA Components

After the ACO step, Genetic Algorithm improves global exploration by:

- **Selection**: Choosing parents based on fitness.
- **Crossover**: Combining two parents to create offspring.
- **Mutation**: Introducing random alterations for diversity.

Chromosome Encoding

Each solution (chromosome) is a task-node mapping:

$$\text{Chromosome} = [n_1, n_2, ..., n_N] \quad \text{where } n_i \text{ is the node for task } i$$

Crossover

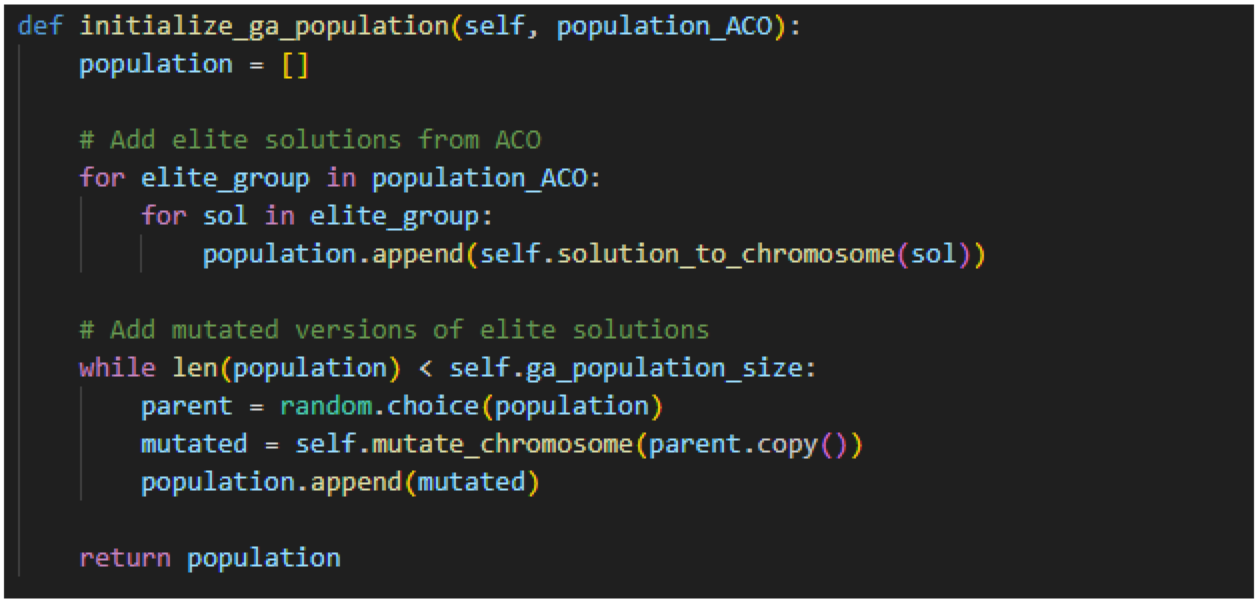
One-point or uniform crossover:

$$\text{child}[i] = \begin{cases} \text{parent1}[i] & \text{if } r < \text{crossover rate} \\ \text{parent2}[i] & \text{otherwise} \end{cases}$$

Mutation

Randomly reassign a task to a different node with probability μ:

$$\text{If } r < \mu \Rightarrow \text{Mutate task } i$$



```
def initialize_ga_population(self, population_ACO):
    population = []

    # Add elite solutions from ACO
    for elite_group in population_ACO:
        for sol in elite_group:
            population.append(self.solution_to_chromosome(sol))

    # Add mutated versions of elite solutions
    while len(population) < self.ga_population_size:
        parent = random.choice(population)
        mutated = self.mutate_chromosome(parent.copy())
        population.append(mutated)

    return population
```

References:

1. Dorigo, M., & Gambardella, L. M. (1997). *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem*. IEEE Transactions on Evolutionary Computation, 1(1), 53–66.

2. Stützle, T., & Hoos, H. H. (2000). *MAX–MIN Ant System*. Future Generation Computer Systems, 16(8), 889–914.

3. Rajkumar Buyya et al. (2013). *Cloud Computing Principles and Paradigms*. Wiley – Chapter on resource scheduling.

4. Dorigo, M., & Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1), 53-66.
   Stützle, T., & Hoos, H. H. (2000). MAX–MIN ant system. *Future Generation Computer Systems*, 16(8), 889-914.

5. Dorigo, M., & Gambardella, L. M. (1997). *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem*. IEEE Transactions on Evolutionary Computation.
6. Blum, C. (2005). *Ant colony optimization: Introduction and recent trends*. Physics of Life Reviews.
7. Ant Colony Optimization – Scholarpedia
8. Ant Colony Optimization Algorithm Tutorial – GeeksforGeeks