



**GENERAL SIR JHON KOTELAWALA DEFENCE
UNIVERSITY**

DEPARTMENT OF COMPUTER SCIENCE

NATURE INSPIRED COMPUTING

CS-3192

REFLECTIVE JOURNAL

DNA DE SILVA

D/BCS/22/0019

CONTENT

1)Genetic Algorithm

2)Ant Colony Optimization

3)Simulated Annealing

4)Particle Swarm Optimization

01) GENETIC ALGORITHM

A Genetic Algorithm (GA) is a search and optimization technique inspired by the process of natural selection and evolution in biology. It is a part of evolutionary algorithms and works on the principle of "survival of the fittest." GAs are used to solve complex optimization problems where traditional methods may not be efficient or feasible.

Key Features of GA:

Population: A set of potential solutions (individuals) for the problem.

Chromosome: Representation of an individual solution.

Fitness Function: Evaluates how good each solution is.

Genetic Operators:

Selection: Chooses individuals for reproduction based on fitness.

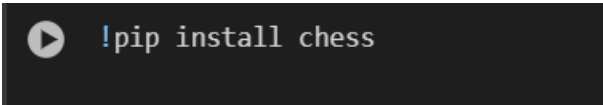
Crossover: Combines two parent solutions to create offspring.

Mutation: Introduces randomness by altering parts of a solution.

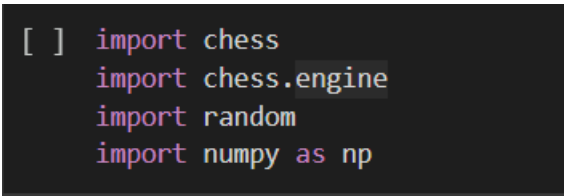
Genetic Algorithms are versatile and can be applied to a wide range of optimization problems.

Real World Example: [Optimizing Chess Strategies Using Genetic Algorithms: A Game AI Approach.](#)

This is an excellent application of Genetic Algorithms in Game AI, focusing on optimizing chess strategies. It mimics the evolutionary process to improve decision-making over time. By understanding and customizing this code, you can explore GAs for other optimization problems in various domains.



```
!pip install chess
```



```
[ ] import chess
import chess.engine
import random
import numpy as np
```

- imports the chess library, which provides tools for working with chess games, including managing the game state, generating legal moves, and evaluating the outcome of games.
- imports the engine module from the chess library, which allows integration with chess engines (like Stockfish) for analyzing positions or generating moves.
- Provides access to Python's random number generation tools, which are essential for introducing stochastic behavior in simulations.
- Imports NumPy, a library for numerical computations and working with arrays. It simplifies mathematical operations and manipulations of arrays.

```
[ ] # Step 1: Initialize Population
def initialize_population(size, chromosome_length):
    return [np.random.uniform(-1, 1, chromosome_length) for _ in range(size)]
```

This function initializes the population of strategies. Each strategy is a vector (chromosome) of `chromosome_length` elements, where each element is a random float between -1 and 1. This vector represents the strategy's weights for different chess features (e.g., material, mobility).

```
[ ] # Step 2: Define Fitness Function
def evaluate_fitness(strategy, num_games=10):
    fitness = 0
    for _ in range(num_games):
        board = chess.Board()
        while not board.is_game_over():
            # AI plays as white
            move = choose_move(board, strategy)
            if move:
                board.push(move)
            else:
                break
            if board.is_game_over():
                break
            # Random move for the opponent
            legal_moves = list(board.legal_moves)
            if legal_moves:
                board.push(random.choice(legal_moves))
        # Evaluate fitness (e.g., win = +1, loss = -1, draw = 0)
        if board.result() == "1-0": # AI wins
            fitness += 1
        elif board.result() == "0-1": # AI loses
            fitness -= 1
    return fitness
```

This function simulates games between the AI (using a given strategy) and a random opponent. It plays `num_games` games, updating the fitness score depending on whether the AI wins, loses, or draws the game. A win adds +1 to the fitness, a loss subtracts 1, and a draw doesn't change fitness.

- `strategy`: A vector of weights, `num_games`: The number of chess games to play in the evaluation process (default is 10).
- `fitness = 0` - The fitness score starts at 0. This will accumulate points based on the outcomes of the simulated games.
- A new chessboard (`board = chess.Board()`) is initialized for each game.
- The game continues until one side wins, the game ends in a draw, or a stalemate occurs (`board.is_game_over()`).
- The AI decides its moves using the `choose_move()` function, which uses the strategy to evaluate the board and pick the best move.
- Game Result:
 - "1-0": The AI (playing as white) wins the game, so the fitness score increases by 1.
 - "0-1": The AI loses the game, so the fitness score decreases by 1.
 - "1/2-1/2" (draw): No change in fitness.

```
[ ] # Step 3: Choose Move Based on Strategy
def choose_move(board, strategy):
    legal_moves = list(board.legal_moves)
    if not legal_moves:
        return None
    move_scores = []
    for move in legal_moves:
        board.push(move)
        score = evaluate_board(board, strategy)
        move_scores.append(score)
        board.pop()
    return legal_moves[np.argmax(move_scores)]
```

This function picks the best move from all legal moves based on the strategy. Each legal move is evaluated by calling `evaluate_board`, and the move with the highest score is chosen.

```
[ ] # Step 4: Evaluate Board Based on Strategy
def evaluate_board(board, strategy):
    material_score = sum(strategy[0] * (len(board.pieces(piece, chess.WHITE)) - len(board.pieces(piece, chess.BLACK)))
                        for piece in range(1, 7))
    mobility_score = strategy[1] * len(list(board.legal_moves))
    center_control_score = strategy[2] * sum(board.piece_at(chess.square(col, row)) is not None
                                           for col in range(3, 5) for row in range(3, 5))
    return material_score + mobility_score + center_control_score
```

This function calculates a "score" for the current board state, based on a given strategy (chromosome).

- `material_score`: Evaluates the difference in material between white and black.
- `mobility_score`: Measures the number of legal moves the AI can make (greater mobility is usually better).
- `center_control_score`: Evaluates whether the AI controls the center of the board.

These scores are weighted by values in the strategy chromosome.

```
[ ] # Step 5: Selection
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    probabilities = [f / total_fitness for f in fitnesses]
    parents = random.choices(population, probabilities, k=2)
    return parents
```

This function selects two parents from the population, with higher fitness individuals having a greater chance of being selected. It uses a probability distribution based on the fitness values.

This total is used to calculate the relative probability of each strategy being chosen. Strategies with higher fitness get a larger probability, meaning they are more likely to be selected as parents.

Ex: Population: ["A", "B", "C", "D"] Fitnesses: [10, 30, 50, 10]

Steps- 1) Total fitness: $10 + 30 + 50 + 10 = 100$.

2) Probabilities: "A": $10 / 100 = 0.1$ (10% chance), "B": $30 / 100 = 0.3$ (30% chance)

"C": $50 / 100 = 0.5$ (50% chance), "D": $10 / 100 = 0.1$ (10% chance)

3) Two parents are randomly selected based on these probabilities. For example, the result could be ["C", "B"] because "C" has the highest chance.

```
[ ] # Step 6: Crossover
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2
```

This function simulates the crossover operation, where two parent chromosomes are combined to create two children. A random crossover point is selected, and the chromosomes are swapped at that point.

```
[ ] # Step 7: Mutation
def mutate(chromosome, mutation_rate=0.1):
    for i in range(len(chromosome)):
        if random.random() < mutation_rate:
            chromosome[i] += np.random.normal()
    return chromosome
```

This function applies a mutation to a chromosome. For each gene in the chromosome, there's a probability (given by `mutation_rate`) that it will be mutated by adding a small random number drawn from a normal distribution.

```
[ ] # Step 8: Genetic Algorithm
def genetic_algorithm(pop_size, chromosome_length, num_generations):
    population = initialize_population(pop_size, chromosome_length)
    for generation in range(num_generations):
        fitnesses = [evaluate_fitness(individual) for individual in population]
        new_population = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population
        best_fitness = max(fitnesses)
        print(f"Generation {generation + 1}, Best Fitness: {best_fitness}")
    return population[np.argmax(fitnesses)]
```

This is the main genetic algorithm loop. It runs for (`num_generations`) generations, evolving the population of strategies.

- The population is evaluated, and fitness scores are calculated.
- Parents are selected based on fitness, and crossover and mutation are applied to generate new offspring.
- The best fitness value from each generation is printed.

In step 9, This line runs the genetic algorithm, initializing the population with 10 strategies (chromosomes) of length 3, and runs it for 20 generations. The best strategy found is printed at the end.

```
[ ] # Step 9: Run the Genetic Algorithm
best_strategy = genetic_algorithm(pop_size=10, chromosome_length=3, num_generations=20)
print("Best Strategy:", best_strategy)
```

```
⇨ Generation 1, Best Fitness: 4
Generation 2, Best Fitness: 4
Generation 3, Best Fitness: 5
Generation 4, Best Fitness: 5
Generation 5, Best Fitness: 4
Generation 6, Best Fitness: 4
Generation 7, Best Fitness: 6
Generation 8, Best Fitness: 4
Generation 9, Best Fitness: 7
Generation 10, Best Fitness: 7
Generation 11, Best Fitness: 6
Generation 12, Best Fitness: 6
Generation 13, Best Fitness: 7
Generation 14, Best Fitness: 6
Generation 15, Best Fitness: 6
Generation 16, Best Fitness: 7
Generation 17, Best Fitness: 7
Generation 18, Best Fitness: 6
Generation 19, Best Fitness: 6
Generation 20, Best Fitness: 6
Best Strategy: [ 0.67313241 -1.50447102  0.02754276]
```

02) ANT COLONY OPTIMIZATION

Ant Colony Optimization (ACO) is a nature-inspired optimization algorithm that mimics the behavior of real-world ants searching for food. Ants in a colony communicate indirectly by leaving pheromone trails on the ground. Over time, these pheromone trails help guide other ants toward optimal paths. The stronger the pheromone concentration on a path, the more likely it is to be chosen by other ants.

In computational terms, ACO uses this behavior to solve complex optimization problems. It is particularly effective for problems where the solution can be represented as a path through a graph, such as:

- Traveling Salesman Problem (TSP),
- Network routing,
- Vehicle routing,
- Scheduling problems.

Real world example: [Network Routing Optimization Using ACO](#)

Scenario: In a computer network, data packets need to travel between nodes (such as routers or servers) efficiently. The goal is to find the shortest and most reliable routes to minimize delays and ensure consistent communication.

Imagine a computer network as a graph:

- Nodes represent routers, servers, or switches.
- Edges represent the connections between them, with a "cost" assigned to each edge (e.g., time delay, bandwidth, or reliability).

Data packets (represented by ants in this case) must travel from a source node to a destination node. The challenge is to determine the best paths in terms of cost and reliability.

- **Ants as Packets:**Ants act as data packets that explore the network, traveling from the source to the destination.
- **Cost as Pheromone Trail:**Ants leave behind virtual pheromones on the paths they take. Paths with lower cost (e.g., shorter time or higher reliability) accumulate more pheromones.
- **Dynamic Updates:**Over time, as more ants traverse the network, better paths become reinforced, guiding future ants toward those paths.
- **Evaporation and Exploration:**Pheromones evaporate over time, encouraging ants to explore other paths instead of being stuck on suboptimal ones.

Real-World Use of ACO in Network Routing:

- **Telecommunications:** ACO is used in dynamic routing protocols to improve the efficiency of IP networks. For example, algorithms like AntNet utilize ACO principles for routing packets across networks.
- **Wireless Sensor Networks:** ACO helps optimize energy consumption and communication efficiency in sensor networks.
- **Data Center Networks:** ACO can assist in finding optimal paths for data transfers in large-scale data centers.

Python Idea: Represent the Network as a Graph

To implement ACO for network routing in Python:

1. Graph Representation:

- Use a graph where nodes represent routers or devices and edges represent network links.
- Assign a cost to each edge based on factors like latency or reliability.

2. ACO Algorithm:

- Simulate ants exploring paths in the graph.
- Update pheromones on paths based on the quality (cost) of routes taken by ants.
- Use probabilistic selection to guide ants based on pheromone concentration.

3. Result:

- The algorithm finds the least-cost path from the source node to the destination node.

Let's see how I can implement the code.

Step 1: Understand the Problem.

The goal is to find the shortest and most reliable path between two nodes in a network.

- **Network Representation:** A graph where:
 - **Nodes** represent devices or computers.
 - **Edges** represent the connection between nodes, with a "cost" (e.g., time or reliability).
- **Ants' Role:** Ants explore the graph to find paths from the source to the destination, leaving pheromones on good paths to guide future ants.

Step 2: Algorithm Outline

1. Initialization

- Create a graph with nodes and edges.
- Initialize pheromone levels on all edges.

2. Simulating Ants

- Each ant starts at the source node and tries to reach the destination node.
- Decisions are made based on:
 - The amount of pheromone on edges.
 - The "cost" of the edges (lower cost = more attractive).

3. Pheromone Update

- Ants deposit pheromone on paths they use, with better paths receiving more pheromones.
- Pheromones evaporate over time to prevent convergence to suboptimal solutions.

4. Iteration

- Repeat the process for multiple iterations to refine the paths.

```
[ ] import numpy as np
    import random
    import networkx as nx
    import matplotlib.pyplot as plt
```

Numpy- A library for numerical computing in Python.

Random-generating random numbers and making random selections.

Networkx-used to create, manipulate, and analyze complex networks (graphs).

Matplotlib-Allows customizations for graph layouts, node sizes, edge labels, colors, and more.

```
# Define the graph as a dictionary
# Each edge has a "cost" (e.g., time or reliability)
graph = {
    0: {1: 2, 2: 2},
    1: {2: 1, 3: 4},
    2: {3: 1},
    3: {}
}
```

Purpose: Represents the network as a directed graph where:

- Keys (0, 1, etc.) are nodes.
- Values (1: 2, 2: 2) are neighboring nodes with associated costs.

Example: Node 0 is connected to 1 and 2, each with a cost of 2.

```
[ ] # Parameters for ACO
    num_ants = 5
    num_iterations = 100
    alpha = 1 # Importance of pheromone
    beta = 2 # Importance of edge cost
    rho = 0.5 # Pheromone evaporation rate
    pheromone_initial = 1.0
```

- num_ants: Number of ants exploring the graph in each iteration.
- num_iterations: Total number of iterations to refine the solution.
- alpha: Controls the influence of pheromone levels on path selection.
- beta: Controls the influence of edge cost (heuristic) on path selection.
- rho: Pheromone evaporation rate; prevents stagnation.
- pheromone_initial: Initial pheromone level on all edges.

```
[ ] # Initialize pheromones
    pheromones = {(node, neighbor): pheromone_initial for node in graph for neighbor in graph[node]}
```

Creates a dictionary where each edge (node, neighbor) starts with a default pheromone level (1.0).

```
[ ] # Helper function to calculate probabilities for an ant's next move
def calculate_probabilities(current_node, unvisited, pheromones, graph, alpha, beta):
    probabilities = []
    for next_node in unvisited:
        edge = (current_node, next_node)
        pheromone = pheromones[edge] ** alpha
        cost = graph[current_node][next_node]
        heuristic = (1 / cost) ** beta
        probabilities.append(pheromone * heuristic)
    total = sum(probabilities)
    return [p / total for p in probabilities]
```

Computes probabilities for selecting the next node based on:

- Pheromone Levels (alpha): Tracks ant activity.
- Edge Cost Heuristic (beta): Rewards shorter paths.

Output: A list of normalized probabilities for choosing each unvisited neighbor.

```
[ ] # Function to simulate one ant's journey
def simulate_ant(start, end, graph, pheromones, alpha, beta):
    current_node = start
    path = [current_node]
    unvisited = set(graph.keys())
    unvisited.remove(current_node)
    total_cost = 0

    while current_node != end:
        unvisited_neighbors = [n for n in graph[current_node] if n in unvisited]
        if not unvisited_neighbors:
            return None, float('inf') # No path found

        probabilities = calculate_probabilities(current_node, unvisited_neighbors, pheromones, graph, alpha, beta)
        next_node = random.choices(unvisited_neighbors, probabilities)[0]

        path.append(next_node)
        total_cost += graph[current_node][next_node]
        current_node = next_node
        unvisited.remove(current_node)

    return path, total_cost
```

Simulates one ant starting at start and trying to reach end:

- Chooses the next node probabilistically based on pheromone and cost.
- Tracks the path taken and its total cost.

Edge Case: If no unvisited neighbors exist, the path fails.

```
[ ] # Update pheromones based on ants' paths
def update_pheromones(paths, pheromones, rho):
    # Evaporate pheromones
    for edge in pheromones:
        pheromones[edge] *= (1 - rho)

    # Add pheromones based on paths
    for path, cost in paths:
        for i in range(len(path) - 1):
            edge = (path[i], path[i + 1])
            pheromones[edge] += 1 / cost
```

- This function, update_pheromones, is responsible for updating the pheromone levels on the edges of the graph in the Ant Colony Optimization (ACO) algorithm.
- For every edge in the graph, reduce the pheromone level by a factor of $(1 - \rho)$. If $\text{pheromones}[(0, 1)] = 1.0$ and $\rho = 0.5$, the new pheromone level will be: $1.0 \times (1 - 0.5) = 0.5$.
- Calculate the contribution to pheromones as $1 / \text{cost}$. Shorter (lower-cost) paths contribute more pheromones. For every edge $(\text{path}[i], \text{path}[i + 1])$ in the path, add pheromones.

If the path is $[0, 1, 3]$ and the cost is 5: Each edge in the path (e.g., $(0, 1)$ and $(1, 3)$) will have $1 / 5 = 0.2$ pheromones added. If the initial pheromone level of $(0, 1)$ was 0.5, the new level will be:

$0.5 + 0.2 = 0.7$

```
[ ] # ACO main function
def ant_colony_optimization(start, end, graph, num_ants, num_iterations, alpha, beta, rho):
    best_path = None
    best_cost = float('inf')

    for _ in range(num_iterations):
        paths = []

        for _ in range(num_ants):
            path, cost = simulate_ant(start, end, graph, pheromones, alpha, beta)
            if path:
                paths.append((path, cost))
                if cost < best_cost:
                    best_path, best_cost = path, cost

        update_pheromones(paths, pheromones, rho)

    return best_path, best_cost
```

- Repeats the simulation for multiple iterations.
- Tracks the best path and cost found across all iterations.
- Updates pheromones after each iteration to refine solutions.

```
[ ] # Run the ACO algorithm
start_node = 0
end_node = 3
best_path, best_cost = ant_colony_optimization(start_node, end_node, graph, num_ants, num_iterations, alpha, beta, rho)

print("Best Path:", best_path)
print("Best Cost:", best_cost)
```

Best Path: [0, 2, 3]
Best Cost: 3

Goal is to find the shortest/least-cost path from start_node to end_node in the graph. The ACO algorithm iteratively improves solutions using ant-inspired behavior.

Result:

The best path is the sequence of nodes with the least cost.

The best cost is the total "weight" of this path, indicating efficiency.

```
[ ] # Visualization Function
def visualize_graph(graph, best_path=None):
    G = nx.DiGraph()

    # Add edges and their weights
    for node in graph:
        for neighbor, weight in graph[node].items():
            G.add_edge(node, neighbor, weight=weight)

    pos = nx.spring_layout(G) # Layout for positioning nodes
    plt.figure(figsize=(8, 6))

    # Draw nodes and edges
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=500, font_size=10)
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

    # Highlight the best path
    if best_path:
        path_edges = [(best_path[i], best_path[i + 1]) for i in range(len(best_path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='red', width=2.5)

    plt.title("Network Graph and Best Path")
    plt.show()

# Visualize the graph and best path
visualize_graph(graph, best_path)
```

This function, `visualize_graph`, is designed to create a visual representation of the graph and optionally highlight the best path identified by the Ant Colony Optimization (ACO) algorithm.

G: Represents the graph as a directed graph using NetworkX.

Loops through each node in the graph dictionary. For each neighbor of the current node, an edge is added to the `G` graph along with its weight (cost).

`nx.spring_layout(G)`: Computes positions for the nodes in a visually appealing way, with edges spaced out and nodes positioned to minimize overlap.

`plt.figure(figsize=(8, 6))`: Sets the size of the visualization window

Nodes:

- `nx.draw` draws the graph with nodes and edges.
- Nodes are displayed as light blue circles.

Attributes:

- `with_labels=True`: Displays node labels (e.g., 0, 1, 2).
- `node_size=500`: Sets the size of the nodes.
- `font_size=10`: Adjusts the size of the labels.

Edges:

Edge weights: Displayed as labels on the edges.

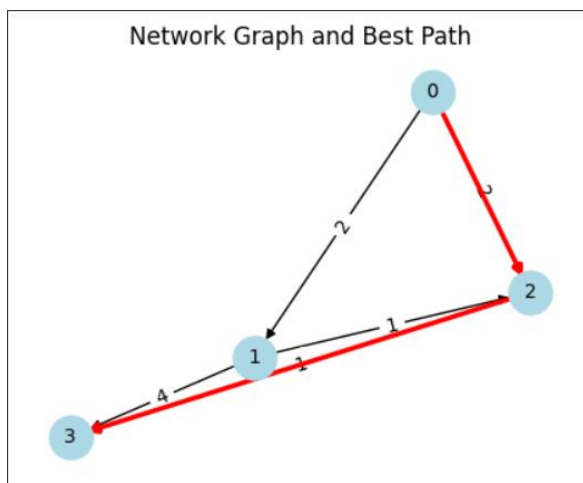
- `nx.get_edge_attributes(G, 'weight')`: Retrieves the weight of each edge.
- `nx.draw_networkx_edge_labels`: Displays these weights at their respective edges.

If `best_path` is provided (e.g., `[0, 2, 3]`):

- Extracts the edges that make up the path (`[(0, 2), (2, 3)]`).
- Highlights these edges in **red** and makes them **thicker** for emphasis.

`plt.title`: Adds a title to the graph.

`plt.show`: Displays the graph visualization.



(03) SIMULATED ANNEALING

Simulated Annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It is used to find an approximate solution to optimization problems, especially those with large and complex search spaces.

Real World Example: [Timetable Scheduling with SA.](#)

Scenario: You are tasked with creating an optimal timetable for a school or university. The timetable must ensure that:

- No two classes overlap in the same room or teacher.
- Teachers are not assigned to multiple classes at the same time.
- Classes and teachers are assigned to preferred time slots whenever possible.

Timetable scheduling involves satisfying hard constraints (e.g., no overlapping) and soft constraints (e.g., teacher preferences). Simulated Annealing (SA) can help by starting with an initial random timetable, iteratively tweaking it, and improving it based on a cost function. Occasional acceptance of worse solutions ensures the algorithm can escape local optima.

School timetabling for classes and exams, University lectures and lab scheduling, Workforce shift planning in hospitals or offices are some real-world cases that we can use this. Let's see how this code works.

```
[25] import random
import math
import prettytable
```

- random introduces the necessary randomness to explore solutions in the Simulated Annealing algorithm.
- math ensures proper probabilistic calculations during the annealing process.
- prettytable formats the output to display the final timetable in a human-readable way.

```
[39] # Example data
classes = ["Math", "English", "Science", "History", "Art"]
teachers = ["T1", "T2", "T3", "T4", "T5"]
rooms = ["R1", "R2", "R3"]
time_slots = ["9-10", "10-11", "11-12", "1-2", "2-3"]

# Hard constraints: No overlaps of teachers/rooms at the same time.
# Soft constraints: Assign preferred time slots for classes/teachers.
preferences = {
    "Math": {"time": "9-10", "teacher": "T1"},
    "English": {"time": "10-11", "teacher": "T2"},
    "Science": {"time": "11-12", "teacher": "T3"},
    "History": {"time": "1-2", "teacher": "T4"},
    "Art": {"time": "2-3", "teacher": "T5"}
}
```

Lists of available classes, teachers, rooms, and time slots to be scheduled. Specifies ideal time slots and teachers for each class. These are soft constraints, meaning it's good to follow them, but not strictly required.

```
[40] # Initialize a random timetable
def generate_initial_timetable():
    timetable = {}
    for slot in time_slots:
        timetable[slot] = []
        for _ in range(len(rooms)): # One class per room
            timetable[slot].append({
                "class": random.choice(classes),
                "teacher": random.choice(teachers),
                "room": random.choice(rooms)
            })
    return timetable
```

This function creates a random timetable as a starting point for the optimization. For each time slot:

- It assigns random classes, teachers, and rooms to each available room.
- The result is a random but complete schedule.

```

# Calculate cost function
def calculate_cost(timetable):
    cost = 0
    # Check hard constraints
    for slot, assignments in timetable.items():
        teachers_assigned = set()
        rooms_assigned = set()
        for entry in assignments:
            # Penalize overlapping teachers or rooms
            if entry["teacher"] in teachers_assigned:
                cost += 100 # Overlap penalty for teacher
            if entry["room"] in rooms_assigned:
                cost += 100 # Overlap penalty for room
            teachers_assigned.add(entry["teacher"])
            rooms_assigned.add(entry["room"])

    # Check soft constraints
    for slot, assignments in timetable.items():
        for entry in assignments:
            class_name = entry["class"]
            if class_name in preferences:
                # Penalize if not in preferred time slot
                if slot != preferences[class_name]["time"]:
                    cost += 10
                # Penalize if not assigned preferred teacher
                if entry["teacher"] != preferences[class_name]["teacher"]:
                    cost += 10

    return cost

```

Hard Constraints: Checks for overlapping teachers or rooms and heavily penalizes them (cost +100). Soft Constraints: Adds a smaller penalty (+10) if a class isn't in its preferred time slot or doesn't have its preferred teacher. The total cost is returned after evaluating all constraints.

```

# Generate neighbor solution
def generate_neighbor(timetable):
    # Randomly swap a class, teacher, or room in one slot
    new_timetable = {slot: assignments[:] for slot, assignments in timetable.items()}
    slot = random.choice(time_slots)
    if new_timetable[slot]:
        entry = random.choice(new_timetable[slot])
        entry["class"] = random.choice(classes)
        entry["teacher"] = random.choice(teachers)
        entry["room"] = random.choice(rooms)
    return new_timetable

```

The generate_neighbor function generates a **neighboring solution** for the current timetable by making a small random change to it.

In Loop Check if the time slot has assignments:

- If the time slot is empty, nothing is done (no neighbors are generated).
- If it has assignments, proceed to make a random change.

Randomly select an assignment in the time slot: An entry is a dictionary representing a single scheduled class (e.g., {"class": "Math", "teacher": "T1", "room": "R1"}).

```
# Simulated Annealing algorithm
def simulated_annealing(initial_temperature, cooling_rate, min_temperature, max_iterations):
    current_timetable = generate_initial_timetable()
    current_cost = calculate_cost(current_timetable)
    best_timetable = current_timetable
    best_cost = current_cost

    temperature = initial_temperature

    while temperature > min_temperature:
        for _ in range(max_iterations):
            neighbor = generate_neighbor(current_timetable)
            neighbor_cost = calculate_cost(neighbor)
            delta_cost = neighbor_cost - current_cost

            # Accept or reject the neighbor solution
            if delta_cost < 0 or random.random() < math.exp(-delta_cost / temperature):
                current_timetable = neighbor
                current_cost = neighbor_cost

            if current_cost < best_cost:
                best_timetable = current_timetable
                best_cost = current_cost

        # Cool down the temperature
        temperature *= cooling_rate

    return best_timetable, best_cost
```

The simulated_annealing function implements the **Simulated Annealing (SA)** algorithm to find an optimized timetable. This algorithm mimics the process of annealing in metallurgy, where a material is heated and slowly cooled to find a low-energy state.

- initial_temperature: The starting temperature for the algorithm. Higher temperatures allow for greater exploration of solutions.
- cooling_rate: A value less than 1 (e.g., 0.95) that determines how quickly the temperature decreases during the process.
- min_temperature: The stopping condition when the temperature becomes too low.
- max_iterations: The number of attempts to generate new solutions at each temperature level.

Always accept better solutions: If $\text{delta_cost} < 0$, the neighbor is better and is always accepted.

Occasionally accept worse solutions: If $\text{delta_cost} > 0$, the neighbor is worse but may still be accepted with a probability

```
[30] # Run the Simulated Annealing
      best_timetable, best_cost = simulated_annealing(
          initial_temperature=1000,
          cooling_rate=0.95,
          min_temperature=1,
          max_iterations=100
      )
```

- initial_temperature=1000: Starts the process with high randomness.
- cooling_rate=0.95: Reduces temperature by 5% each iteration.
- min_temperature=1: Stops when temperature gets too low.
- max_iterations=100: Limits iterations per temperature step.

```
[31] # Display the result
      print("Best Timetable (with minimum cost):")
      for slot, assignments in best_timetable.items():
          print(f"{slot}: {assignments}")
      print(f"Cost: {best_cost}")
```

Displays the time slots, rooms, and teacher/class assignments, along with the final cost.

```
Best Timetable (with minimum cost):
9-10: [{'class': 'Math', 'teacher': 'T5', 'room': 'R2'}, {'class': 'English', 'teacher': 'T2', 'room': 'R2'}, {'class': 'Science', 'teacher': 'T2', 'room': 'R1'}]
10-11: [{'class': 'Math', 'teacher': 'T4', 'room': 'R3'}, {'class': 'Math', 'teacher': 'T1', 'room': 'R2'}, {'class': 'History', 'teacher': 'T3', 'room': 'R3'}]
11-12: [{'class': 'Math', 'teacher': 'T1', 'room': 'R1'}, {'class': 'Art', 'teacher': 'T2', 'room': 'R3'}, {'class': 'History', 'teacher': 'T5', 'room': 'R1'}]
1-2: [{'class': 'Math', 'teacher': 'T4', 'room': 'R1'}, {'class': 'History', 'teacher': 'T1', 'room': 'R1'}, {'class': 'Math', 'teacher': 'T4', 'room': 'R1'}]
2-3: [{'class': 'Science', 'teacher': 'T4', 'room': 'R3'}, {'class': 'Art', 'teacher': 'T3', 'room': 'R3'}, {'class': 'English', 'teacher': 'T5', 'room': 'R2'}]
Cost: 320
```

```
[32] from prettytable import PrettyTable


      def display_timetable(timetable):
          table = PrettyTable()
          table.field_names = ["Time Slot", "Room", "Class", "Teacher"]

          for slot, assignments in timetable.items():
              for entry in assignments:
                  table.add_row([slot, entry["room"], entry["class"], entry["teacher"]])

          print(table)

      # Display the best timetable
      display_timetable(best_timetable)
```

The display_timetable function is used to display the timetable in a nice, readable table format using the PrettyTable module. This imports the PrettyTable library, which is used to create and display tables in a clean and easy-to-read format.



Time Slot	Room	Class	Teacher
9-10	R2	Math	T5
9-10	R2	English	T2
9-10	R1	Science	T2
10-11	R3	Math	T4
10-11	R2	Math	T1
10-11	R3	History	T3
11-12	R1	Math	T1
11-12	R3	Art	T2
11-12	R1	History	T5
1-2	R1	Math	T4
1-2	R1	History	T1
1-2	R1	Math	T4
2-3	R3	Science	T4
2-3	R3	Art	T3
2-3	R2	English	T5

(04) PARTICLE SWARM OPTIMIZATION

Particle Swarm Optimization (PSO) is an optimization technique inspired by the social behavior of swarms, such as birds flocking or fish schooling. It is widely used to solve optimization problems by finding the best solution in a multi-dimensional search space. In PSO, potential solutions are represented as "particles" that move through the search space, adjusting their positions based on their personal experience (personal best) and the experiences of the entire swarm (global best). A fitness function evaluates the quality of each particle's position, guiding the swarm toward the optimal solution.

The movement of particles is governed by velocity updates, influenced by inertia (maintaining the current direction), personal learning, and social learning components. PSO is known for its simplicity, requiring only a few parameters, such as inertia weight and learning coefficients, which balance exploration and exploitation. It does not rely on gradient information, making it suitable for non-linear, multi-modal problems. Common applications of PSO include engineering design, drone deployment, routing problems, feature selection in machine learning, and economic

modeling. Despite its advantages, such as versatility and global optimization capability, PSO can struggle with high-dimensional problems and premature convergence if swarm diversity is lost. Proper parameter tuning is crucial for achieving optimal results.

Real World Example: Particle Swarm Optimization (PSO) for Drone Swarm Deployment.

Problem- Imagine there's a disaster, like an earthquake, and we need drones to cover the affected area for rescue operations. Each drone is equipped with sensors, and our goal is to make sure that the drones spread out over the area in a way that maximizes coverage (the area each drone can observe) while minimizing overlap (drones covering the same spot multiple times).

We can think of this as a search problem, where each drone (called a "particle" in PSO) moves through a virtual space (the area), and over time, each drone adjusts its position to cover more ground and avoid redundancy.

In PSO:

- Each drone represents a particle in a swarm.
- Every particle (drone) has a position in the grid (the area).
- Each particle has a velocity, which determines how it moves to new positions.
- Each particle also knows the best position it has reached (called personal best), and it also learns from the best position found by any particle in the swarm (called global best).

```
[ ] import numpy as np
import random
```

```
# Define the grid size (disaster area)
grid_size = (10, 10) # 10x10 grid

# Define the number of drones (particles)
num_drones = 5

# Define PSO parameters
w = 0.5 # inertia weight
c1 = 1.0 # personal learning coefficient
c2 = 1.5 # global learning coefficient
```

- Grid size give the disaster area.it means there are 10 rows and 10 columns.

- the variable `num_drones` defines the number of drones (particles) in the swarm. In this case, we have 5 drones.
- The **inertia weight** determines how much of the previous velocity (movement) influences the current velocity of the drone. The **personal learning coefficient** controls how much the drone is influenced by its own previous best position (personal best). The **global learning coefficient** determines how much the drone is influenced by the best position found by the entire swarm (global best).

```
[9] # Initialize positions and velocities for the drones
    positions = np.random.randint(0, grid_size[0], (num_drones, 2)).astype(float) # Convert to float
    velocities = np.random.rand(num_drones, 2) * 2 - 1 # Random initial velocities
```

np.random.randint(low, high, size) generates random integers between low (inclusive) and high (exclusive). **low = 0** means the starting positions of the drones will be between 0 and the grid size. **high = grid_size[0]** (which is 10 in this case) ensures the drones are placed somewhere within the 10x10 grid. **(num_drones, 2)** specifies that drones, where each drone will have 2 coordinates (representing x and y positions on the grid). **.astype(float)**: This converts the generated positions from integers to floating-point numbers (float64).

np.random.rand(m, n) generates a matrix of random floating-point numbers between 0 and 1 with dimensions m x n. *** 2 - 1** (This operation scales the random values between 0 and 1 to a range of -1 to 1).

```
[ ] # Function to evaluate coverage (for simplicity, we assume coverage is the number of unique positions)
    def evaluate_coverage(positions):
        return len(set(tuple(p) for p in positions)) # Unique positions in the grid
```

The function `evaluate_coverage` is designed to measure how well a group of positions covers a given area. **positions**: A list or array of positions, where each position is typically represented as a coordinate (e.g., [x, y]) in a 2D grid. **set(tuple(p) for p in positions)**: A set automatically removes duplicates, so if there are multiple drones (or objects) occupying the same position, they will only be counted once.


```

# PSO Algorithm
def pso():
    # Initialize best positions and coverage
    personal_best_positions = np.copy(positions)
    personal_best_coverage = np.array([evaluate_coverage(positions) for _ in range(num_drones)])
    global_best_position = positions[np.argmax(personal_best_coverage)]

    for iteration in range(100): # Run for 100 iterations
        # Evaluate current coverage
        current_coverage = np.array([evaluate_coverage(positions) for _ in range(num_drones)])

        # Update personal best positions
        for i in range(num_drones):
            if current_coverage[i] > personal_best_coverage[i]:
                personal_best_coverage[i] = current_coverage[i]
                personal_best_positions[i] = positions[i]

        # Update global best position
        global_best_position = positions[np.argmax(personal_best_coverage)]

        # Update velocities and positions
        for i in range(num_drones):
            r1, r2 = np.random.rand(2)
            velocities[i] = w * velocities[i] + c1 * r1 * (personal_best_positions[i] - positions[i]) + c2 * r2 * (global_best_position - positions[i])
            positions[i] += velocities[i]

        # Ensure positions stay within the grid boundaries
        positions[i] = np.clip(positions[i], 0, grid_size[0] - 1)

    # Print the best coverage in each iteration
    print(f"Iteration {iteration + 1}, Best Coverage: {personal_best_coverage.max()}")

    return global_best_position

```

- **personal_best_positions:** Each drone remembers its best position so far (based on coverage).
- **personal_best_coverage:** Keeps track of the best coverage score for each drone.
- **global_best_position:** Finds the position of the drone with the best coverage across the swarm.
- The optimization process runs for 100 iterations. Each iteration tries to improve the drones' positions to increase coverage.
- Calculate the coverage score for each drone's current position. Coverage is the number of unique grid positions occupied.
- If the current position provides better coverage than its previous best, update:
 - Its best coverage score.
 - Its best position.
- Find the position of the drone with the highest coverage score across the swarm. Update the global best position based on this.

Update Velocity:

- Each drone adjusts its velocity based on:
- Inertia (w): Keeps the drone moving in its current direction.
- Personal Learning (c1): Moves toward its own best position.
- Social Learning (c2): Moves toward the global best position.

Update Position:

- Add the new velocity to the current position.
- Ensure the position stays within the grid boundaries using np.clip.
- Convert the positions to integers to match the grid.

After each iteration, display the best coverage score achieved by the swarm so far. After 100 iterations, return the **best position** found by the swarm.

```
[10] # Run PSO to find optimal drone positions
      optimal_positions = pso()

      print(f"Optimal positions for drones: {optimal_positions}")
```

This code runs the Particle Swarm Optimization (PSO) algorithm and prints the result.

Here is the output of this code.

```
[10] Iteration 1, Best Coverage: 4
      ↗ Iteration 2, Best Coverage: 5
      ↘ Iteration 3, Best Coverage: 5
        Iteration 4, Best Coverage: 5
        Iteration 5, Best Coverage: 5
        Iteration 6, Best Coverage: 5
        Iteration 7, Best Coverage: 5
        Iteration 8, Best Coverage: 5
        Iteration 9, Best Coverage: 5
        Iteration 10, Best Coverage: 5
        Iteration 11, Best Coverage: 5
        Iteration 12, Best Coverage: 5
        Iteration 13, Best Coverage: 5
        Iteration 14, Best Coverage: 5
        Iteration 15, Best Coverage: 5
        Iteration 16, Best Coverage: 5
        Iteration 17, Best Coverage: 5
        Iteration 18, Best Coverage: 5
        Iteration 19, Best Coverage: 5
        Iteration 20, Best Coverage: 5
        Iteration 21, Best Coverage: 5
        Iteration 22, Best Coverage: 5
        Iteration 23, Best Coverage: 5
        Iteration 24, Best Coverage: 5
        Iteration 25, Best Coverage: 5
        Iteration 26, Best Coverage: 5
        Iteration 27, Best Coverage: 5
        Iteration 28, Best Coverage: 5
        Iteration 29, Best Coverage: 5
        Iteration 30, Best Coverage: 5
        Iteration 31, Best Coverage: 5
        Iteration 32, Best Coverage: 5
        Iteration 33, Best Coverage: 5
        Iteration 34, Best Coverage: 5
        Iteration 35, Best Coverage: 5
        Iteration 36, Best Coverage: 5
        Iteration 37, Best Coverage: 5
        Iteration 38, Best Coverage: 5
        Iteration 39, Best Coverage: 5
        Iteration 40, Best Coverage: 5
        Iteration 41, Best Coverage: 5
        Iteration 42, Best Coverage: 5
        Iteration 43, Best Coverage: 5
        Iteration 44, Best Coverage: 5
        Iteration 45, Best Coverage: 5
        Iteration 46, Best Coverage: 5
        Iteration 47, Best Coverage: 5
        Iteration 48, Best Coverage: 5
        Iteration 49, Best Coverage: 5
        Iteration 50, Best Coverage: 5
        Iteration 51, Best Coverage: 5
        Iteration 52, Best Coverage: 5
        Iteration 53, Best Coverage: 5
        Iteration 54, Best Coverage: 5
        Iteration 55, Best Coverage: 5
        Iteration 56, Best Coverage: 5
        Iteration 57, Best Coverage: 5
        Iteration 58, Best Coverage: 5
        Iteration 59, Best Coverage: 5
        Iteration 60, Best Coverage: 5
        Iteration 61, Best Coverage: 5
        Iteration 62, Best Coverage: 5
        Iteration 63, Best Coverage: 5
        Iteration 64, Best Coverage: 5
        Iteration 65, Best Coverage: 5
        Iteration 66, Best Coverage: 5
        Iteration 67, Best Coverage: 5
```

```
Iteration 68, Best Coverage: 5
Iteration 69, Best Coverage: 5
Iteration 70, Best Coverage: 5
Iteration 71, Best Coverage: 5
Iteration 72, Best Coverage: 5
Iteration 73, Best Coverage: 5
Iteration 74, Best Coverage: 5
Iteration 75, Best Coverage: 5
Iteration 76, Best Coverage: 5
Iteration 77, Best Coverage: 5
Iteration 78, Best Coverage: 5
Iteration 79, Best Coverage: 5
Iteration 80, Best Coverage: 5
Iteration 81, Best Coverage: 5
Iteration 82, Best Coverage: 5
Iteration 83, Best Coverage: 5
Iteration 84, Best Coverage: 5
Iteration 85, Best Coverage: 5
Iteration 86, Best Coverage: 5
Iteration 87, Best Coverage: 5
Iteration 88, Best Coverage: 5
Iteration 89, Best Coverage: 5
Iteration 90, Best Coverage: 5
Iteration 91, Best Coverage: 5
Iteration 92, Best Coverage: 5
Iteration 93, Best Coverage: 5
Iteration 94, Best Coverage: 5
Iteration 95, Best Coverage: 5
Iteration 96, Best Coverage: 5
Iteration 97, Best Coverage: 5
Iteration 98, Best Coverage: 5
Iteration 99, Best Coverage: 5
Iteration 100, Best Coverage: 5
Optimal positions for drones: [5.25417908 3.84274016]
```