Particle Swarm Optimization (PSO) for University Timetable Scheduling

Introduction and Overview:

Project idea and overview:

The aim of this project is to implement a Particle Swarm Optimization (PSO) algorithm to optimize the scheduling of university courses, lecturers, and rooms across a weekly timetable. It aims to minimize conflicts such as overlapping course times for a single lecturer, inadequate room capacity, and courses scheduled outside a lecturer's available timeslots.

Applied Algorithm:

Particle Swarm Optimization (PSO):

Particle Swarm Optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its local best-known position and is also guided toward the best-known positions in the search-space, which are updated as better positions are found by other particles.

Key Components of PSO:

1. **Particles**:

- Each particle represents a potential solution to the problem. The entire group of particles is known as a swarm.
- Particles have positions which correspond to potential solutions in the search space.

2. **Position**:
   - The position of a particle in the search space represents a candidate solution.
   - Each position has an associated value determined by the fitness function, which indicates the quality of the solution.
3. **Velocity**:
   - Velocity represents the rate of change of the particle’s position. It dictates both the speed and direction in which a particle should move through the search space.
   - The velocity update rule helps particles to converge towards promising areas in the search space.
4. **Fitness Function**:
   - This function evaluates how good a solution is (usually by calculating some form of error or cost). The objective is either to maximize or minimize this function.
   - Each particle's fitness is computed using this function, based on its current position.
5. **Personal Best (pbest)**:
   - Each particle remembers the best position it has ever visited, which is associated with the highest fitness (for maximization problems) or the lowest fitness (for minimization problems) it has achieved so far.
   - This personal best guides the particle’s movement, giving it a "cognitive" component.
6. **Global Best (gbest)**:
   - Out of all the personal bests across the swarm, the position corresponding to the best fitness is kept as the global best.

- The global best provides a "social" component, as it influences the movement of all particles in the swarm.

7. **Velocity Update Rule**:
   - The velocity of each particle is updated based on its previous velocity, the distance from its personal best position, and the distance from the global best position.
   - This update is a blend of inertia (maintaining direction), cognitive influence (attraction to personal best), and social influence (attraction to global best).
8. **Position Update Rule**:
   - After updating its velocity, each particle's position is updated by adding the new velocity to its current position.
   - This update moves the particle through the search space towards potentially better solutions.
9. **Parameters**:
   - **Inertia weight (w)**: Controls the impact of the previous velocity on the current one, providing momentum to avoid local minima.
   - **Cognitive coefficient (c1)**: Determines the influence of the personal best on the velocity update.
   - **Social coefficient (c2)**: Determines the influence of the global best on the velocity update.
   - These parameters balance exploration (searching through a broad area) and exploitation (thoroughly searching a promising area).
10. **Stopping Criteria**:
    - The algorithm stops based on certain criteria, such as reaching a maximum number of iterations, achieving a satisfactory fitness level, or the particles' positions converging.

Dependencies:

The code relies on several Python packages:

- `deap`: Provides tools for evolutionary algorithms and PSO.
  Distributed Evolutionary Algorithms in Python. DEAP is a versatile evolutionary computation framework that enables users to perform genetic algorithms, genetic programming, and other evolutionary algorithms in a simple and Pythonic way.

1. `base`

The `base` module in DEAP includes foundational classes that are used to construct the other components of evolutionary algorithms.

- `Fitness`: A class used to define the fitness of an individual. It can handle single or multiple fitness values and also manage whether each objective should be maximized or minimized.
- `Toolbox`: A class that acts like a container for tools. It allows you to register any type of function under a name and call it later as if it were a method of the toolbox.

2. `creator`

The `creator` module is a meta-factory allowing to create classes on-the-fly. This capability is utilized primarily to create new types of individuals or populations suited to specific problems.

- `create`: The function used to create new types with inheritance from specified base classes. Common uses include creating classes for individuals where fitness is an attribute of those classes (e.g., `creator.create("FitnessMax", base.Fitness, weights=(1.0,))` to create a fitness class that maximizes its attribute).

3. `tools`

The `tools` module is perhaps the most functional part of DEAP, providing numerous ready-to-use genetic operators and tools needed for evolutionary algorithms.

- **Selection**:
  - `selTournament`: Tournament selection, selects the best individual among `tournsize` randomly chosen individuals.
  - `selRoulette`: Roulette wheel selection, selects individuals based on their fitness values.
  - `selBest`: Selects the best individuals in a population.
- **Mutation**:
  - `mutGaussian`: Applies a Gaussian mutation that perturbs the attributes of the input individuals.
  - `mutFlipBit`: Flips bits in a binary string or bitlist.
- **Crossover**:
  - `cxOnePoint`: One point crossover, splits two individuals at the same random point and swaps segments.
  - `cxTwoPoint`: Similar to one-point but with two points, creating a segment that is swapped.
- **Variation**:
  - `varAnd`: A common variation that applies crossover and mutation.

- The `configparser` module in Python is used for working with configuration files. It provides a way to read, write, and modify configuration files, which are commonly stored in a format similar to the Windows INI style. These files are useful for managing application settings, user preferences, and operational parameters outside of the codebase, making applications easier to configure and maintain.

- `numpy`: Adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions..

- `pandas`: Required for handling data in a structured form, although not explicitly used in the current implementation.
- `matplotlib`: For creating visualizations of the timetables.

Configuration:

Constants

- `NUM_COURSES`: Total number of courses (20). We assumed 20 courses every level takes 5 courses.
- `NUM_LECTURERS`: Number of lecturers (10).
- `NUM_TIMESLOTS`: Number of timeslots available (20). 4 timeslots per day over a 5-day week.
- `NUM_ROOMS`: Number of rooms available (6).
- `POPULATION_SIZE`: Size of the population in the PSO (2000).
- `MAX_GENERATIONS`: Maximum number of iterations for the PSO (700). This will be our STOPING CRITERIA.
- `COURSE_HOURS`: Random duration between 1 to 3 hours for each course.
- `POPULAR_COURSE_COMBINATIONS`: Typical sets of courses taken together by students at various levels.
  Courses 0, 1, 2, 16, 17and 3 are taken by Level 1.
  Courses 4, 5, 6, 18, 19and 7 are taken by Level 2.
  And so on....

Data Structures

- `LECTURER_AVAILABILITY`: A matrix indicating the availability of each lecturer for each timeslot.
  - Rows represent different lecturers.
  - Columns represent different timeslots throughout a day or week.

The matrix is filled with 0s and 1s:

- `0` indicates the lecturer is unavailable during that timeslot.
- `1` indicates the lecturer is available during that timeslot.
- The dimensions of the matrix are `(NUM_LECTURERS, NUM_TIMESLOTS)`, where `NUM_LECTURERS` is the total number of lecturers and `NUM_TIMESLOTS` is the number of available timeslots.
- `np.random.choice([0, 1], ..., p=[0.2, 0.8])` is used to randomly select 0 or 1 for each entry in the matrix. The probability `p=[0.2, 0.8]` means there's a 20% chance of picking 0 (unavailable) and an 80% chance of picking 1 (available).

- `COURSE_REQUIREMENTS`: A matrix specifying which lecturer is qualified to teach which course.
  - Rows represent different courses.
  - Columns represent lecturers.

  The matrix is filled with 0s and 1s:
  - `0` means the lecturer at that column cannot teach the course at that row.
  - `1` means the lecturer at that column can teach the course at that row.
  - The shape `(NUM_COURSES, NUM_LECTURERS)` defines a matrix with `NUM_COURSES` rows and `NUM_LECTURERS` columns, where `NUM_COURSES` is the total number of courses offered.
  - Each element is randomly chosen as either 0 or 1 with an equal probability (default 50% for each since no `p` parameter is specified here).

- `ROOM_CAPACITY`: Array containing the capacity of each room.
  - Each element represents the capacity of a room (i.e., how many people it can accommodate).
  - `np.random.randint(10, 100, size=NUM_ROOMS)` generates random integers between 10 and 99 for each room.

    - `NUM_ROOMS` is the total number of rooms available. This parameter defines the length of the `ROOM_CAPACITY` array.
- `course_to_level` is designed to map a given course identified by its `course_id` to its corresponding educational level based on predefined `POPULAR_COURSE_COMBINATIONS`.

Functions:

1- Class Creation with DEAP:

- The code uses the DEAP library to create custom classes for the particle and its fitness.
- Defines a new class called `FitnessMin` derived from `base.Fitness`. The `weights=(-1.0,)` specifies that the fitness should be minimized, as in PSO, lower fitness values are better. This fitness class will be used to evaluate the quality of the scheduling solutions.
- **Particle**: A subclass of `list` designed to represent a particle in PSO. It includes:
    - `fitness`: An instance of `FitnessMax`, used to evaluate how good the particle is.
    - `best`: Initially set to `None`, it is meant to store the best position the particle has discovered (i.e., the best solution it has found).
    - `speed`: A list representing the velocity of the particle in the search space.

2- Function to Create a Particle `create_particle()`:

This code defines a Particle class and provides two methods for creating instances of this class: `create_particle_random()` and `create_particle_level_based()`.

The Particle class is created using Python's `creator` module, which is commonly used in genetic programming libraries like DEAP (Distributed Evolutionary

Algorithms in Python). It's typically used for creating classes with specific attributes.

Here's a breakdown of the code:

1. **Particle Class Definition**:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.
0,))
creator.create("Particle", list, fitness=creator.FitnessMi
n, best=None, speed=list)
```

   - It creates two classes: `FitnessMin` and `Particle`.
   - `FitnessMin` is created inheriting from `base.Fitness` with the intention of minimizing fitness values.
   - `Particle` is created as a subclass of the Python built-in `list` class. It has additional attributes `fitness`, `best`, and `speed`.

2. **Particle Creation Methods**:
   - `create_particle(method='random')` is a general method for creating a Particle instance. It allows for two initialization methods: `'random'` and `'level_based'`.
   - `create_particle_random()` creates a particle with random assignments for each course. It ensures that each course is assigned to a unique time slot, lecturer, room, and day.
   - `create_particle_level_based()` creates a particle with assignments based on course levels. It ensures that courses of the same level are not scheduled at the same time slot.

3. **Initialization**:
   - In both creation methods, a Particle instance is constructed using the attributes defined earlier (`creator.Particle`). The attributes are populated based on the specific creation method logic.

Overall, this code sets up a framework for generating particles with specific attributes and initialization methods, likely for use in some optimization or simulation algorithm.

3- Function to Update a Particle `update_particle`:

This function, `update_particle()`, is responsible for updating a given particle's attributes based on the best solution found so far, as well as some coefficients that determine the influence of various factors on the update process. Here's what each part of the function does:

1. **Function Signature**:

```
def update_particle(particle, best, coeff_inertia=0.5, coe
ff_cognitive=0.5, coeff_social=0.5):
```

   - It takes in the following parameters:
     - `particle`: The particle to be updated.
     - `best`: The best solution found so far.
     - `coeff_inertia`, `coeff_cognitive`, `coeff_social`: Coefficients determining the influence of inertia, cognitive component, and social component, respectively. They default to 0.5 if not provided.

2. **Updating the Particle**:

```
new_particle = []
```

   - Initializes an empty list to store the updated particle.

3. **Iterating Through Particle Attributes**:

```
for idx, (course_id, lecturer_id, room_id, timeslot_id) in
enumerate(particle):
```

   - Iterates through each attribute (course_id, lecturer_id, room_id, timeslot_id) of the particle using enumeration to keep track of the index.

4. **Determining Shift**:

```
shift = random.randint(-3, 3) * coeff_inertia + \\
        (particle.best[idx][3] - timeslot_id) * coeff_cogn
```

```
itive + \\
        (best[idx][3] - timeslot_id) * coeff_social
```

- Calculates a shift for the timeslot based on:
  - Random inertia component.
  - Cognitive component: Difference between the current timeslot and the best timeslot found in the current particle.
  - Social component: Difference between the current timeslot and the best timeslot found in the entire population.

5. **Applying Shift**:

```
new_timeslot = (timeslot_id + int(shift)) % NUM_TIMESLOTS
```

- Calculates the new timeslot by adding the calculated shift to the current timeslot and ensuring it wraps around within the bounds of the available timeslots.

6. **Checking Validity**:

```
if LECTURER_AVAILABILITY[lecturer_id][new_timeslot] == 1:
    new_particle.append((course_id, lecturer_id, room_id,
new_timeslot))
else:
    new_particle.append((course_id, lecturer_id, room_id,
timeslot_id))
```

- Checks if the new timeslot is valid for the lecturer. If it is, the updated attribute is added to the new particle. Otherwise, the original attribute is added.

7. **Updating the Particle**:

```
particle[:] = new_particle
```

- Updates the original particle with the new attributes.

In summary, this function calculates a new particle by adjusting the timeslots of its attributes based on random inertia, cognitive (personal best), and social (global best) components, while ensuring the validity of the new timeslots.

3.1 - Function to update particle with over selection

This function, `update_particle_with_overselection()`, updates a given particle's attributes based on the best solutions found in the population, incorporating over-selection of elite individuals. Let's break it down:

1. **Function Signature**:

```
def update_particle_with_overselection(particle, populatio
n, coeff_inertia=0.5, coeff_cognitive=0.5, coeff_social=0.
5, elite_frac=0.2):
```

   - It takes in the following parameters:
     - `particle`: The particle to be updated.
     - `population`: The population containing solutions.
     - `coeff_inertia`, `coeff_cognitive`, `coeff_social`: Coefficients determining the influence of inertia, cognitive component, and social component, respectively. They default to 0.5 if not provided.
     - `elite_frac`: The fraction of elite individuals in the population. It defaults to 0.2 if not provided.

2. **Sorting Population**:

```
sorted_population = sorted(population, key=lambda p: p.fit
ness.values[0])
```

   - Sorts the population based on the fitness values of each individual.

3. **Selecting Elites**:

```
elites = sorted_population[:int(len(sorted_population) * e
lite_frac)]
```

- Selects a portion of the population as elites based on the specified fraction (`elite_frac`).

4. **Selecting Non-Elites**:

```
non_elites = sorted_population[int(len(sorted_population)
* elite_frac):]
```

- Selects the remaining individuals in the population as non-elites.

5. **Updating the Particle**:

```
new_particle = []
for idx, (course_id, lecturer_id, room_id, timeslot_id) in
enumerate(particle):
```

- Initializes an empty list to store the updated particle.
- Iterates through each attribute (course_id, lecturer_id, room_id, timeslot_id) of the particle using enumeration to keep track of the index.

6. **Determining the Best Solution for Particle**:

```
best = elites[0] if particle in elites else random.choice
(non_elites)
```

- Selects the best solution for the particle:
  - If the particle itself is among the elite individuals, the best solution is the first elite individual.
  - Otherwise, a random non-elite individual is chosen as the best solution.

7. **Calculating Shift**:

```
shift = random.randint(-3, 3) * coeff_inertia + \\
        (particle.best[idx][3] - timeslot_id) * coeff_cogn
itive + \\
        (best[idx][3] - timeslot_id) * coeff_social
```

- Calculates a shift for the timeslot similar to the previous function.

8. **Applying Shift** and Validity Check**:
   - Same as in the previous function.
9. **Updating the Particle**:

```
particle[:] = new_particle
```

   - Updates the original particle with the new attributes.

This function essentially enhances the particle update process by incorporating over-selection, where a fraction of the population consisting of the best individuals (elites) has a greater influence on the update compared to randomly selected individuals.

4- Function to Evaluate the fitness `evaluate`:

This function, `evaluate(particle)`, calculates the fitness of a given particle based on various constraints and conflicts encountered in scheduling. Let's break down the steps:

1. **Defining Penalty Weights**:
   - The function initializes a dictionary called `penalties`, which assigns penalty weights to different types of conflicts. These penalties represent the severity of each type of conflict encountered during scheduling.
2. **Initializing Conflict Trackers**:
   - Several dictionaries are initialized to keep track of conflicts at different levels:
     - `lecturer_time_conflict`: Tracks conflicts between lecturers and timeslots.

- `room_time_conflict`: Tracks conflicts between rooms and timeslots.
- `level_timeslot_conflict`: Tracks conflicts between course levels and timeslots.
- `level_day_timeslots`: Tracks course timeslots by level and day.

3. **Iterating Through Particle Attributes**:
   - The function iterates through each attribute of the particle (course_id, lecturer_id, room_id, timeslot_id) to identify conflicts and constraints.
4. **Checking Lecturer Conflicts**:
   - It checks if a lecturer is already assigned to another course at the same timeslot. If so, it incurs a severe conflict penalty (`lecturer_conflict`).
5. **Checking Room Capacity Conflicts**:
   - It checks if a room is already occupied at the same timeslot. If so, it incurs a moderate conflict penalty (`room_conflict`).
6. **Checking Lecturer Availability**:
   - It checks if the assigned lecturer is available at the assigned timeslot. If not, it incurs a basic conflict penalty (`availability_conflict`).
7. **Checking Level Scheduling and Gaping Conflicts**:
   - It tracks the timeslots of courses for each level and checks for conflicts within the same level. Additionally, it checks for gaps in the schedule of each level, where a course of the same level should ideally follow the previous one without any gap. Gaping conflicts are the least severe and incur a lower penalty (`gaping_conflict`).
8. **Calculating Total Conflicts**:
   - The total number of conflicts is accumulated based on the penalties incurred for each type of conflict encountered during scheduling.
9. **Returning Fitness**:
   - The function returns a tuple containing the total number of conflicts as the fitness value of the particle. Since this is a minimization problem, the lower the fitness value, the better the solution.

In summary, this function provides an advanced fitness evaluation mechanism for a scheduling problem by considering various types of conflicts and constraints, assigning penalties accordingly, and computing the overall fitness of a particle based on these penalties.

5-Main Function `run_pso()`:

This function, `run_pso(initialization_method, update_method='normal')`, is an enhanced version of the PSO algorithm which allows for customization of both initialization and update methods. Let's break down its structure and functionality:

1. **Function Definition**:

```
def run_pso(initialization_method, update_method='norma
l'):
```

   - Defines a function named `run_pso` that takes two parameters: `initialization_method` (required) and `update_method` (optional, defaulting to 'normal').

2. **Seeding Random Number Generators**:

```
random.seed(42)
np.random.seed(42)
```

   - Seeds the random number generators to ensure reproducibility.

3. **Initialization of Population and Best Particle**:

```
population = [create_particle(initialization_method) for _
in range(POPULATION_SIZE)]
best = creator.Particle(population[0])  # Initialize best
with the first particle
best.fitness.values = toolbox.evaluate(best)
```

- Initializes the population by creating particles using the specified initialization method.
- Initializes the best particle with the first particle in the population.
- Evaluates the fitness of the best particle.

4. **Main PSO Loop**:

```
for gen in range(1, MAX_GENERATIONS + 1):
```

- Iterates through each generation starting from 1 (generation 0 is already evaluated).

5. **Particle Evaluation and Update**:

```
for part in population:
    toolbox.update(part, best)
    part.fitness.values = toolbox.evaluate(part)
    if not part.best or part.best.fitness < part.fitness:
        part.best = creator.Particle(part)
        part.best.fitness.values = part.fitness.values
    if best.fitness < part.fitness:
        best = creator.Particle(part)
        best.fitness.values = part.fitness.values
```

- Evaluates the fitness of each particle and updates its position using the specified update method.
- Updates the personal best of each particle (`part.best`) and the global best (`best`) if necessary.

6. **Printing Generation Information**:

```
print("Generation:", gen)
print("Best Particle:", best)
print("Best Fitness:", best.fitness.values[0])
```

- Prints information about the current generation, best particle, and its fitness.

7. **Update Method Selection**:

```
if update_method == 'normal':
    for part in population:
        toolbox.update(part, best)
elif update_method == 'overselection':
    for part in population:
        update_particle_with_overselection(part, populatio
n)
```

- Determines the update method based on the `update_method` parameter.
- If `update_method` is set to 'normal', it updates particles using the normal update method.
- If `update_method` is set to 'overselection', it updates particles using an alternative update method (possibly incorporating over-selection).

8. **Finalization and Result Reporting**:

```
if best is not None:
    print("Final Best Particle =", best)
    print("Final Best Fitness:", best.fitness.values[0])
    organize_and_display_schedule(best)
else:
    print("PSO failed to find a best solution.")
```

- Prints the final best particle and its fitness if a solution is found.
- Calls a function to organize and display the schedule based on the best particle.
- If no solution is found, it prints a failure message.

9. **Return Statement**:

```
return best, fitness_history
```

- Returns the best solution found and the fitness history over generations.

This function provides flexibility by allowing users to specify both the initialization method and the update method for the PSO algorithm, making it adaptable to different problem domains and optimization scenarios.