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.

- `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 (5).
- `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 function generates a single scheduling solution (particle), ensuring no course is scheduled more than once a day.

- **Initialization**: A new particle is represented as an empty list, and `used_slots` is a set to track which course is already scheduled on which day.
- **Course Scheduling Loop**: It iterates through all courses that need scheduling.
- **Valid Assignment Loop**:
  Inside the loop for each course, it keeps trying to find a valid assignment of lecturer, room, and timeslot that adheres to the constraints.
    - **Choosing a Lecturer**:

      This selects a lecturer who is qualified to teach the course (lecturers who can teach a course are marked with `1` in `COURSE_REQUIREMENTS[course_id]`).
    - **Choosing a Room and Timeslot**:

      A room and timeslot are chosen at random. `day_id` is calculated to determine the day for the given timeslot, assuming `TIMESLOTS` defines timeslots per day.
    - **Checking and Assigning**:

      It ensures the course isn't scheduled more than once on the same day by checking the `used_slots` set. If not scheduled, the tuple of `(course_id, lecturer_id, room_id, timeslot_id)` is added to the particle and `(course_id, day_id)` to `used_slots`.
- **Return the Particle**:

The function returns the particle, now representing a complete scheduling solution adhering to the constraints.

Each particle contains:

- The course schedule details (course, lecturer, room, and timeslot).
- The fitness value representing how good the solution is (negative number of conflicts).
- The `speed` attribute used in PSO for adjusting the solution.
- Each course is assigned to a qualified lecturer.
- No course is scheduled more than once on any given day.

3- Function to Update a Particle `update_particle`:

- `update_particle()`: Updates a particle's position in the solution space using the PSO formula that considers inertia, cognitive component (particle's best-known position), and social component (global best position).
- **particle**: The current state of a particle, which is a list of tuples. Each tuple contains the details of a scheduled course (course_id, lecturer_id, room_id, timeslot_id).
- `best`: A reference to the global best particle position found across all iterations, or the best particle from the swarm's history.
- `coeff_inertia`, `coeff_cognitive`, `coeff_social`: Coefficients controlling how much the current velocity (inertia), the particle's own best-known position (cognitive), and the global best position (social) influence the new velocity.

Inside the Function:

- **Initialization**:

  Starts with an empty list that will store the new version of the particle.

- **Loop Through Each Course in Particle**:

  Iterates over each course tuple in the current particle to update its timeslot based on various influences.

- **Compute Shift**:

- **Random component** (`random.randint(-3, 3) * coeff_inertia`): Introduces a random shift, weighted by the inertia coefficient, to keep some exploratory behavior in the particle's movement.
- **Cognitive component** (`(particle.best[idx][3] - timeslot_id) * coeff_cognitive`): Influences the particle to move towards its own historically best-known position.
- **Social component** (`(best[idx][3] - timeslot_id) * coeff_social`): Encourages the particle to move towards the global best known position.

- **Calculate New Timeslot**:

  Updates the timeslot, ensuring it remains valid by wrapping around using modulo operation with the total number of timeslots.

- **Validate New Timeslot and Append**:

  Checks if the lecturer is available at the new timeslot. If yes, the course is scheduled in the new timeslot. Otherwise, it retains its current timeslot.

- **Update Particle**:

  The original particle list is updated in place with the new_particle list.

In this function:

- `coeff_inertia` represents the inertia weight which controls how much the previous velocity affects the new velocity, or in this case, the tendency of a course to stay in its previous timeslot.
- `coeff_cognitive` and `coeff_social` weights influence the movement towards the particle's own best-known position and the global best-known position, respectively.

1. **Shift Computation**: The actual shift (or velocity) for each course's timeslot is computed as a weighted sum of three components:
   - **Random Component**: A random integer between -3 and 3 is generated and multiplied by `coeff_inertia`. This represents a random change influenced by the previous motion (simulating momentum Optimization Algorithm).

- **Cognitive Component**: The difference between the particle's best-known timeslot for the current course (`particle.best[idx][3]`) and the current timeslot is calculated and scaled by `coeff_cognitive`. This represents a pull towards the particle's own historical best position.
- **Social Component**: The difference between the global best timeslot for the current course (`best[idx][3]`) and the current timeslot is similarly calculated and scaled by `coeff_social`. This represents a pull towards the swarm's overall best position.
- The total `shift` is a weighted sum of these three components.

Considerations

- The velocity in our case does not directly translate into a traditional velocity vector since the search space (scheduling) is not continuous but rather discrete and categorical.
- The modification of the timeslot (movement in the search space) based on the calculated "velocity" (shift) ensures that each particle explores new potential solutions while also exploiting known good configurations.

Same Function with Fine Tuning to Parameters:

```
def update_particle(particle, best, gen, max_gen):
# Linearly decreasing inertia weight
w_start = float(config['PSO']['INERTIA_WEIGHT'])
w_end = 0.4  # Lower bound of inertia weight
w = w_start - (w_start - w_end) * (gen / max_gen)

phi1 = float(config['PSO']['COGNITIVE_COEFFICIENT'])
phi2 = float(config['PSO']['SOCIAL_COEFFICIENT'])

u1, u2 = np.random.uniform(0, 1), np.random.uniform(0, 1)
v_u1 = phi1 * u1 * (particle.best - particle)
v_u2 = phi2 * u2 * (best - particle)
particle.speed = w * particle.speed + v_u1 + v_u2
```

```
particle[:] = particle + particle.speed
particle[:] = np.clip(particle, 0, NUM_TIMESLOTS - 1)
```

In the function `update_particle`, there is parameter tuning applied indirectly through the adaptive or dynamic adjustment of the inertia weight `w`. Here are the details regarding the parameter tuning elements included in the function:

1. **Inertia Weight (w)**: The inertia weight `w` is dynamically adjusted during the optimization process. This is a form of parameter tuning where the inertia weight decreases linearly from `w_start` to `w_end` as the generations progress. This approach:
    - Starts with a higher inertia weight (`w_start` from the configuration), allowing the particles to explore the solution space more freely at the beginning of the optimization process.
    - Gradually decreases the inertia weight to `w_end` (in this case, hardcoded as 0.4), reducing the impact of previous velocity and allowing particles to fine-tune their positions as they converge towards the best solutions. This helps in balancing exploration (searching new areas) and exploitation (refining existing good solutions).
2. **Cognitive and Social Coefficients (phi1 and phi2)**: These coefficients are typically fixed for the duration of the algorithm's execution based on initial configuration values. They are not dynamically adjusted within the `update_particle` function itself but are critical parameters that could be tuned externally (e.g., through experimental runs or using meta-optimization techniques) to optimize performance for specific problems.
3. **Random Components (u1, u2)**: The function uses randomness (`u1`, `u2`) in computing the velocity updates. While not a tuning of a static parameter, this stochasticity is crucial for ensuring diversity in the search process and helps avoid premature convergence.

In summary, the function includes dynamic adjustment of the inertia weight, which is a direct form of parameter tuning within the context of the PSO algorithm. This function is set up to adjust the inertia weight based on the progression through generations, which is a commonly used strategy in PSO to enhance convergence behavior.

4- Function to Evaluate the fitness `evaluate`:

- `evaluate()`: Computes the fitness of a particle by counting various types of scheduling conflicts.

Penalties Dictionary

- This dictionary defines the severity of different types of scheduling conflicts. A higher number indicates a more severe penalty. The types of conflicts are:
    - `lecturer_conflict`: Occurs when the same lecturer is scheduled to teach different courses at the same timeslot.
    - `room_conflict`: Happens when more than one course is scheduled in the same room at the same timeslot.
    - `availability_conflict`: Arises if a lecturer is scheduled to teach at a timeslot when they are not available.
    - `level_conflict`: (Though this isn't directly referenced in the penalties applied, it's presumed to be considered in the course leveling process).
    - `gaping_conflict`: Occurs when there are gaps (more than one timeslot) between courses scheduled on the same day for the same course level, indicating inefficient use of time.

Conflict Tracking Structures

- `lecturer_time_conflict`: A dictionary to track which lecturers are teaching at which timeslots.
- `room_time_conflict`: A dictionary to track which rooms are used at which timeslots.
- `level_day_timeslots`: A nested dictionary to track the timeslots used per course level on each day.

Conflict Checks in the Particle

- The function iterates through each course tuple in the particle, performing several checks:

    - **Lecturer Conflict**: Checks if the same lecturer is assigned to more than one course at the same timeslot.
    - **Room Conflict**: Checks if the same room is used by more than one course at the same timeslot.
    - **Availability Conflict**: Verifies if the lecturer is available to teach at the assigned timeslot.
    - **Level and Gaping Conflict**: For each course, the function determines its level and the day based on the timeslot, then checks if the scheduling of courses within the same level and day has unnecessary gaps.

Post-Processing for Gaping Conflicts

- After processing all courses in the particle, the function further checks for `gaping_conflict` by analyzing the timeslots for each course level on each day.
- If consecutive timeslots within a day for a course level have more than one timeslot gap between them, a `gaping_conflict` penalty is added.

Return Value

- The function returns a tuple containing the total number of conflict penalties. The comma following `conflicts` indicates that the return value is a tuple, which is a common requirement for fitness functions in genetic algorithms and related optimization techniques.

Configurations:

Toolbox Setup

- This initializes a new `Toolbox` instance, which is a way for DEAP to store various functions including generation, mutation, mating, and selection functions in genetic algorithms or in this case, particle initialization and updating functions for PSO.

Particle Initialization

- Registers the function `create_particle` under the alias "particle" in the toolbox. This function is responsible for creating a single particle, which represents a potential solution in the search space (in your case, a particular course schedule).

Population Initialization

- Registers a method to generate an entire population of particles. `tools.initRepeat` is a helper from DEAP which facilitates the repeated application of a given function (`toolbox.particle` in this case) to generate multiple particles.
- It uses `list` as the container to store the population, and `toolbox.particle` as the method for generating each individual in that list.

Update Mechanism

- Registers the `update_particle` function under the alias "update". This function updates a particle's position in the search space based on its velocity, its own best-known position, and the best-known position among all particles. The update includes adjustments according to coefficients of inertia, cognitive, and social components.

Fitness Evaluation

- Registers the `evaluate` function under the alias "evaluate". This function computes the fitness of a particle by assessing the number of conflicts in a scheduling solution, with different types of conflicts contributing different penalty values to the fitness.

5-Main Function `main` :

Set Up and Initialize the PSO

1. **Set Random Seed**:

   This initializes the random number generator to ensure reproducibility of results. The seed `42` is a common choice used in examples.

2. **Initialize Population**:

   This creates an initial population of particles using the `toolbox` object, which is presumably configured elsewhere. Each particle represents a potential scheduling solution.

3. **Initialize Best Particle Tracker**:

   This initializes a variable to keep track of the best particle found across all iterations.

Run the PSO Algorithm

1. **PSO Loop**:

   - The outer loop runs the PSO for a specified number of generations (`MAX_GENERATIONS`).
   - Each particle is evaluated for its fitness, which measures how good the scheduling solution is (presumably, a lower number of scheduling conflicts is better).
   - The `part.best` stores the best position this particle has ever found.
   - The `best` variable tracks the best solution found across all particles.

2. **Print Best Results**:

   Prints the best particle found and its fitness score.

Organize Schedule by Level and Visualize

1. **Prepare Data Structures for Visualization**:

   Prepares a nested dictionary to organize schedules by course levels, days, and timeslots.

2. **Populate Level Schedules**:

   Populates `level_schedules` based on the best particle, placing each course in the appropriate slot according to its level.

3. **Display Timetables for Each Level**:
   Uses matplotlib to generate tables displaying the timetables for each course level.

Aggregate and Visualize All Courses Together

1. **Prepare and Populate General Schedule**:
   Similar to the earlier visualization but aggregates all courses into a single schedule table regardless of the course level.
2. **Visualize the Detailed Course Schedule**:
   Uses matplotlib to create a more detailed table showing all courses, lecturers, and rooms in their assigned timeslots and days.

User Interaction:

Allows users to:

- Select their student level.
- Choose specific courses to be displayed.
- View the corresponding timetable adjusted to their selections.

`display_courses_by_level(level)`:

designed to display a list of course IDs associated with a particular student level from a predefined list called `POPULAR_COURSE_COMBINATIONS`.

Parameters

- **level** (int): Represents the student level for which courses need to be displayed. It is expected that this value corresponds to an index (plus one, because it's human-readable and not zero-based) into the `POPULAR_COURSE_COMBINATIONS` list.

Functionality

1. **Validate Input**:
   - This block checks if the input `level` is within the valid range (1 to the length of `POPULAR_COURSE_COMBINATIONS`).

- If the input `level` is out of bounds, it prints an error message indicating the valid range and returns an empty list. This is a preventive measure to avoid accessing out-of-range indices in the list, which would cause an error.

2. **Retrieve Courses**:
    - This line retrieves the list of courses corresponding to the given `level`. The `1` adjustment is necessary because list indices in Python start at 0, whereas the `level` parameter is expected to start from 1 (making it more intuitive for users).
3. **Display Course Information**:
    - This part first prints a header indicating which level's courses are being displayed.
    - It then iterates through the list of `courses` retrieved earlier and prints each `course_id`.

Return Value

- **returns** `courses`: After displaying the course IDs, the function returns the list of courses. This allows the function not only to serve as a tool for user information but also to be integrated into other parts of an application where the list of courses might be needed programmatically.

`display_timetable_for_selected_courses(best, selected_courses)` :

filters a given timetable (provided as the parameter `best`, which is likely the best particle or solution from a Particle Swarm Optimization process) to only include certain `selected_courses`, and then visualizes this filtered timetable using `matplotlib`

Parameters

- **best**: A list of tuples, where each tuple contains information about a course's scheduling (`course_id`, `lecturer_id`, `room_id`, `timeslot_id`). This represents the optimal or best scheduling solution obtained from an optimization algorithm.

- **selected_courses**: A list of course IDs that the user wants to visualize in the timetable.

Functionality

1. **Initialize Filtered Schedule**:
    - This line initializes a dictionary of dictionaries. The outer dictionary has days as keys (`DAYS`), and the inner dictionary uses timeslots as keys (`TIMESLOTS`), initializing an empty list for each timeslot of each day. This will be used to store course information for selected courses only.
2. **Filter Schedule for Selected Courses**:
    - This loop iterates over each course in the `best` schedule. If a course's ID is in the list of `selected_courses`, it calculates which day and which timeslot the course is assigned to, based on `timeslot_id`.
    - The course details are formatted into a string (`entry`) and added to the corresponding timeslot of the corresponding day in the `filtered_schedule`.
3. **Display the Timetable Using Matplotlib**:
    - A figure and axes are created using `matplotlib.pyplot.subplots`.
    - `table_data` is prepared by extracting schedule details for each day and timeslot, formatted into a 2D list structure that `matplotlib` can use to construct a table.
    - The table is created and customized with labels for rows (`DAYS`) and columns (`TIMESLOTS`), and some styling choices like cell color and font sizes are set.
    - The table is displayed with `plt.show()`, which brings up a visual representation of the timetable for the selected courses.