# Evolutionary Algorithm

# Presented for DR. **Amr S. Ghoneim**

# Index

1. PSO introduction and explanation
2. PSO university time table
3. Code Break Down
4. Genetics introduction
5. Genetics code break down

**Particle Swarm Optimization**

**particle swarm optimization** (**PSO**) is a computational method that [optimizes](https://en.wikipedia.org/wiki/Mathematical_optimization) a problem by [iteratively](https://en.wikipedia.org/wiki/Iterative_method) trying to improve a [candidate solution](https://en.wikipedia.org/wiki/Candidate_solution) with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, here dubbed [particles](https://en.wikipedia.org/wiki/Point_particle), and moving these particles around in the [search-space](https://en.wikipedia.org/wiki/Optimization_(mathematics)#Concepts_and_notation) according to simple [mathematical formula](https://en.wikipedia.org/wiki/Formula) over the particle's [position](https://en.wikipedia.org/wiki/Position_(vector)) and [velocity](https://en.wikipedia.org/wiki/Velocity). Each particle's movement is influenced by its local best known position , but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

PSO is originally attributed to [Kennedy](https://en.wikipedia.org/wiki/James_Kennedy_(social_psychologist)), [Eberhart](https://en.wikipedia.org/wiki/Russell_C._Eberhart) and [Shi](https://en.wikipedia.org/wiki/Yuhui_Shi)[[2]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-kennedy95particle-2)[[3]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-shi98modified-3) and was first intended for [simulating](https://en.wikipedia.org/wiki/Computer_simulation) [social behaviour](https://en.wikipedia.org/wiki/Social_behaviour),[[4]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-kennedy97particle-4) as a stylized representation of the movement of organisms in a bird [flock](https://en.wikipedia.org/wiki/Flocking_(behavior)) or [fish school](https://en.wikipedia.org/wiki/Fish_school). The algorithm was simplified and it was observed to be performing optimization. The book by Kennedy and Eberhart[[5]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-kennedy01swarm-5) describes many philosophical aspects of PSO and [swarm intelligence](https://en.wikipedia.org/wiki/Swarm_intelligence). An extensive survey of PSO applications is made by [Poli](https://en.wikipedia.org/wiki/Riccardo_Poli).[[6]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-poli07analysis-6)[[7]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-poli08analysis-7) In 2017, a comprehensive review on theoretical and experimental works on PSO has been published by Bonyadi and Michalewicz.[[1]](https://en.wikipedia.org/wiki/Particle_swarm_optimization#cite_note-bonyadi16survey-1)

PSO is a [metaheuristic](https://en.wikipedia.org/wiki/Metaheuristic) as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. Also, PSO does not use the [gradient](https://en.wikipedia.org/wiki/Gradient) of the problem being optimized, which means PSO does not require that the optimization problem be [differentiable](https://en.wikipedia.org/wiki/Differentiable_function) as is required by classic optimization methods such as [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent) and [quasi-newton methods](https://en.wikipedia.org/wiki/Quasi-newton_methods). However, metaheuristics such as PSO do not guarantee an optimal solution is ever found.

**Particle swarm optimization for university**

**Time table**

The main Idea of our project is to solve time tabling problem, the timetableproblem is a type of constraint satisfaction problem. It involves assigning a set of events (for example, university lectures or school classes) to a limited amount of time slots in a way that satisfies a given set of constraints our algorithm creates a number of tables each table is created randomly and it is assigned a score and this score gets updated according to constraints violations at the end the algorithm returns the best table with the highest score

**PSO Code Break Down**

1. import pandas as pd: This line imports the pandas library and assigns it the alias pd. Pandas is a powerful data manipulation library that provides data structures and functions needed for manipulating structured data.
2. import numpy as np: This line imports the numpy library and assigns it the alias np. Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
3. import random: This line imports the random module, which contains a variety of functions used for generating random numbers.
4. from multiprocessing import Pool: This line imports the Pool class from the multiprocessing module. Pool is used to create a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

import pandas as pd  
import numpy as np  
import randomfrom multiprocessing import Pool  
from functools import partial

This code define a class named Particle. In the context of optimization algorithms, a particle usually represents a potential solution.

Here’s a breakdown of the class definition:

* class Particle: This line defines a new class named Particle.
* def \_\_init\_\_(self): This is the initializer method for the class. When a new instance of the class is created, this method is automatically called. The self parameter refers to the instance of the class.
* self.schedule = self.create\_schedule() This line calls the create\_schedule method (which is not shown in the provided code snippet) and assigns its return value to the schedule attribute of the particle. The schedule attribute represents the state or configuration of the particle, which is a potential solution to the problem at hand.
* self.score = 0 This line initializes the score attribute of the particle to 0. The score attribute is typically used to store the fitness or quality of the particle as a solution to the problem. A fitness function (not shown in the provided code snippet) would evaluate the schedule and update this score.

A screenshot of a computer program

Description automatically generated

This is a method named create\_schedule within a particle . The purpose of this method is to generate a schedule, which is a list of tuples. Each tuple represents a lecture and contains four elements: a day, a time slot, a hall, and a course & professor pair.

1. schedule = []: Initializes an empty list named schedule. This list will be populated with tuples representing lectures.
2. for \_ in range(len(course\_professors)):: Starts a loop that iterates once for each course & professor pair. The underscore \_ is a common convention in Python for a loop variable that won’t actually be used inside the loop.
3. day = random.choice(days), time\_slot = random.choice(time\_slots), hall = random.choice(halls): For each iteration of the loop, randomly selects a day, a time slot, and a hall.
4. course\_professor = course\_professors[\_]: Selects a course & professor pair. Note that this is not random selection; it selects the course & professor pair at the current index of the loop.
5. schedule.append((day, time\_slot, hall, course\_professor)): Appends a tuple containing the selected day, time slot, hall, and course & professor pair to the schedule list.
6. return schedule: After the loop has finished, the method returns the schedule list.

A screenshot of a computer

Description automatically generated

This code snippet is a method named fitness\_func within class Particle class. The purpose of this method is to evaluate the fitness of a particle (a potential solution to a scheduling problem) based on several constraints. The fitness is represented by a score, which starts at 100 and is decreased each time a constraint is violated.

1. **Initialize Variables**: The method starts by initializing several empty lists and dictionaries that will be used to track various aspects of the schedule, as well as a score variable set to 100.
2. **Iterate Over Schedule**: The method then iterates over each row in the particle’s schedule. For each row, it extracts the day, time slot, hall, and professor.
3. **Check Constraints**: The method checks several constraints for each row in the schedule:
   * It checks if two or more lectures are in the same hall at the same time.
   * It checks if a professor gives two or more lectures at the same time.
   * It stores the number of lectures per day and checks if each day has lectures more than the allowed range.
   * It stores the lectures of each professor on each day and checks if a professor gives more than 3 consecutive lectures on the same day.

If any of these constraints are violated, the score is decreased.

1. **Set Score**: Finally, the method sets the score attribute of the particle to the calculated score.

A black screen with white text

Description automatically generated

This is the main Particle Swarm Optimization (PSO) function :

* evaluate\_particle(particle): This function evaluates the fitness of a particle (a potential solution) by calling its fitness\_func method and then returns the particle.
* main(): This is the main function where the PSO algorithm is executed.
  + particles = [Particle() for \_ in range(NUM\_OF\_PARTICLES)]: It starts by creating number of particles by creating objects from particle class .
  + with Pool() as pool:: It creates a multiprocessing Pool to allow parallel computation for better performance .
  + for \_ in range(NUM\_OF\_ITERATIONS):: It runs the PSO algorithm for num of iterations. In each iteration:
    - particles = pool.map(partial(evaluate\_particle), particles): It evaluates all particles in parallel.
    - best\_particle = max(particles, key=lambda p: p.score): It finds the best particle (the one with the highest score).
  + After all iterations, it prints the score of the best particle and its schedule.
  + Finally, it creates a DataFrame from the best schedule and saves it to a CSV file.

A screen shot of a computer

Description automatically generated

**Genetics Introduction**

This document outlines the implementation and usage of a genetic algorithm designed for course scheduling. The algorithm aims to optimize the scheduling of courses across various days, time slots, and halls, while considering constraints such as avoiding scheduling conflicts and ensuring a balanced workload for professors.

**Genetics Code break down**

Implementation Overview

The genetic algorithm is implemented in Python and consists of several key components:

Fitness Function: Evaluates the quality of a scheduling solution based on defined constraints and objectives.

Mutation Operator: Introduces random changes to individual solutions to explore new possibilities.

Crossover Operator: Combines solutions from different parents to produce offspring with characteristics of both parents.

Genetic Algorithm Loop: Manages the evolution of solutions over multiple generations, including parent selection, crossover, mutation, and survivor selection.

Components

Fitness Function (fitness\_func):

Evaluates the quality of a scheduling solution based on several criteria, including:

Avoiding scheduling conflicts: Ensures that no two lectures are scheduled in the same hall at the same time and that no professor delivers multiple lectures simultaneously.

Balancing workload: Limits the number of lectures per day and consecutive lectures for each professor.

Returns a fitness score representing the quality of the solution.

Mutation Operator (mutation):

Introduces random changes to a scheduling solution to explore new possibilities.

Randomly selects a subset of genes (lecture slots) in a solution and modifies them by assigning random values for day, time slot, and hall.

Crossover Operator (uniform\_crossover):

Combines solutions from two parent individuals to produce offspring with characteristics of both parents.

Utilizes uniform crossover, where genes are selected randomly from either parent with a probability threshold.

Genetic Algorithm Loop (start):

Manages the evolution of solutions over multiple generations.

Selects parent individuals based on a specified crossover rate.

Applies uniform crossover to selected parents to generate offspring.

Applies mutation to a subset of individuals to introduce diversity.

Employs GENITOR (delete-worst) survivor selection strategy by removing the worst-performing individuals to maintain a constant population size.

Usage

To utilize the genetic algorithm for course scheduling, follow these steps:

Input Data:

Define possible values for columns such as days, time slots, halls, and course-professor combinations.

Ensure availability of a dataset containing course and professor information.

Initialize Population:

Create an initial population of scheduling solutions, represented as arrays of lecture slots.

Execute Algorithm:

Call the start function with parameters such as population size, mutation rates, crossover rates, and the number of generations.

The algorithm will iteratively evolve solutions over multiple generations.

Output:

The algorithm will output the best scheduling solution along with its fitness score.

The solution will be saved as a CSV file containing the scheduled lectures sorted by days.

**Conclusion**

The genetic algorithm for course scheduling provides an effective approach to optimize the allocation of courses across various time slots and halls while adhering to constraints and objectives. By iteratively evolving solutions through mutation, crossover, and survivor selection, the algorithm can generate high-quality schedules that meet the requirements of educational institutions.

**PSO\_with\_Genetics**

We have combined the two algorithms together with the same working techniques of them but we have edited our PSO to initialize the population of genetics so when the PSO finishes it creates a dictionary of the best solutions and then send them to the genetics this resulted in better results and better constraints following .

**Over all comparison**

We have created a table from each algorithm here are the scores of each one based on fitness function score It makes a better fitness score but it takes more time complexity as shown

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | PSO | PSO with Genetics | Genetics |
| Score based on fitness function | 98.2 | 99.2 | 99 |
| Time Complexity | O(N\*M)  Where n is number of iterations  Where m is number of particles | Complexity of PSO + complexity of genetics | Where g is number of generations p is number of parents n is size of the population and m is the number of rows in the solution |