***Adaptive University Timetabling***

***Introduction***

Timetabling is a fundamental problem in educational institutions, where the goal is to assign a set of courses to specific timeslots, rooms, and instructors in such a way that various constraints are satisfied. It falls under the category of Constraint Satisfaction Problems (CSPs), which require finding solutions that meet a set of predefined rules or restrictions.In the context of university timetabling, constraints are typically divided into two categories: hard constraints, which must be strictly satisfied (e.g., no room conflicts, instructor availability, and room capacity), and soft constraints, which are desirable but not mandatory (e.g., minimizing idle time for instructors or students). A feasible timetable must satisfy all hard constraints, while the optimization objective is often to minimize violations of soft constraints or improve overall efficiency.Solving this problem is challenging due to its combinatorial nature; the number of possible schedules grows exponentially with the number of courses, rooms, and timeslots. As a result, metaheuristic algorithms such as Genetic Algorithms, Simulated Annealing, and Swarm Intelligence methods are commonly used to explore the search space efficiently and find near-optimal solutions.This project formulates university timetabling as a constraint satisfaction optimization problem, evaluates solutions using a penalty-based fitness function, and applies intelligent search techniques to generate effective and conflict-free schedules.

**Related Work**

Several studies have tackled UTP using evolutionary algorithms:

Here is the **Related Work** section reformulated as one-line summaries for each paper:

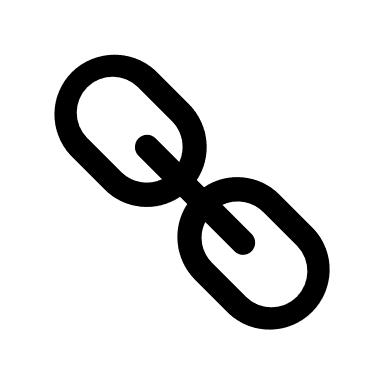
**Related Work**

1. **Premasiri (Rajarata University)** proposed a Genetic Algorithm-based system for course timetabling that successfully satisfied all hard constraints and optimized soft constraints such as lecturer preferences and student workloads.
2. **Burke et al. (University of Nottingham)** developed a GA-based exam timetabling system that used a graph coloring heuristic for initialization and outperformed traditional heuristics in high-conflict scenarios.
3. **Herath (University of Mississippi)** implemented a GA method for university course timetabling, achieving feasible schedules by balancing room capacities and minimizing consecutive student sessions.
4. **Akı (Trakya University)** applied a GA approach for course timetabling with heuristic-initialized populations, ensuring satisfaction of hard constraints and improving room utilization.
5. **HPSOT (Hybrid PSO Tool)** utilized a hybrid Particle Swarm Optimization approach with local search operators to generate cost-effective timetables, demonstrating scalability and superior convergence on Thai university dataset

Project overview

## Dataset :

The dataset used is from “kaggle”

https://www.kaggle.com/datasets/danielefm/students-timetables-university-of-brasilia

a widely accepted benchmark for university course scheduling. It includes a list of courses, rooms, time slots, student groups, and constraints.

## Problem Representation

**The university timetable consists of:**

* Courses (Math, Physics, etc.)
* Timeslots (available hours/days)
* Rooms (lecture halls, labs)
* Instructors (assigned teachers)
* Student Groups (registered students for each course)

**Each possible timetable is encoded as a particle in PSO or chromosome in GA, containing course assignments.**

**Example Encoding:**

**[ [Course1, Timeslot3, Room2, InstructorA, {StudentSet1}], [Course2, Timeslot1, Room4, InstructorB, {StudentSet2}], ... ]**

**Fitness Function**

**The fitness function ensures optimized scheduling:**

* **Hard Constraints (must be followed)**
  + No student has overlapping classes.
  + Number of students is compatible with room capacities.
  + All courses must have an assignment in the timetable.
  + Professors are not assigned to different courses at the same timeslot.
  + Classrooms cannot be assigned to more than one class at the same time.
  + Fridays and end of day slots are not permitted (Week starts on Sunday and ends on Thursday).
* **Soft Constraints (preferences)**
  + Minimize scheduling gaps for students.
  + No more than two lectures per day for each student.
  + Ensure balanced workload distribution.
  + No more than two lectures per day for each professor.
  + Minimize scheduling gaps for professors.

Optimization technique :

**🔧 Fitness Function**

**Pseudocode: fitness\_function**

INPUT:

- timetable: list of (course\_id, instructor\_id, room\_id, timeslot)

- courses: dict of all course info

- students: dict mapping student\_id to list of course\_ids

- rooms: dict mapping room\_id to (name, capacity)

INITIALIZE:

- penalty = 0

- room\_schedule = {} # tracks timeslots used per room

- instr\_schedule = {} # tracks timeslots used per instructor

- allowed\_timeslots = {1, 2, ..., 25}

FOR each (course\_id, instructor\_id, room\_id, timeslot) in timetable:

1. IF timeslot not in allowed\_timeslots:

penalty += 100

2. IF room\_id has conflict (i.e., timeslot already used):

penalty += 100

Add timeslot to room's schedule

3. Get room capacity

Count enrolled students for the course

IF enrolled > capacity:

penalty += (enrolled - capacity) \* 10

4. IF instructor\_id has conflict (i.e., timeslot already used):

penalty += 100

Add timeslot to instructor's schedule

AFTER loop:

5. FOR each course in course list:

IF course not in assigned courses from timetable:

penalty += 100

RETURN total penalty

**🔍 Explanation of Constraints**

**1. ✅ Valid Timeslot Range**

* Timetable slots must be between 1 and 25.
* Violations cost **100 penalty points**.

**2. Room Conflicts**

* A room **cannot host two courses** at the same time.
* If two courses share the same room and timeslot, **+100 penalty**.

**3. Room Capacity Violations**

* Each room has a capacity.
* If **enrolled students > room capacity**, the penalty is:
* (enrolled - capacity) × 10

**4.instructor Conflicts**

* An instructor **can’t teach multiple courses at once**.
* If one is double-booked in a timeslot, **+100 penalty**.

**5. All Courses Must Be Assigned**

* If any course is missing from the timetable, **+100 penalty** per missing course.

**🧠 Goal**

The **fitness score = penalty**.  
 **Lower is better.**  
An ideal solution has a penalty of **0**, which means all hard constraints are satisfied.

**Artificial Bee Colony (ABC) optimization code**

**✅ Overview**

* **Goal**: Minimize the cost (fitness) of a timetable using swarm intelligence.
* **Fitness**: Calculated using a custom fitness\_function that evaluates constraint satisfaction and quality of the timetable.
* **Main Entities**:
  + Bee: Represents a solution (food source) with a timetable and its fitness.
  + colony\_size: Number of bees.
  + limit: Max number of failed trials before becoming a scout.
  + max\_iter: Number of optimization iterations.

**Explanation of the Code**

**Classes & Functions:**

**1. class Bee:**

Stores the timetable, fitness score, and a counter for unsuccessful trial updates.

**2. generate\_random\_timetable(...):**

Creates a random initial timetable by assigning each course:

* An instructor (from instructors)
* A random room
* A random time slot

**3. neighborhood\_solution(...):**

Creates a neighboring solution by randomly modifying one entry (room and time) of a timetable.

**abc\_optimize(...) Function Steps:**

1. **Initialization (Employed Bees)**:
   * Generate colony\_size random timetables.
   * Calculate fitness for each using fitness\_function.
   * Track the best bee (solution) found so far.
2. **Main Optimization Loop (max\_iter iterations)**:

**🔸 Employed Bees Phase:**

* + For each bee:
    - Generate a neighbor solution.
    - If it's better → accept it and reset trial count.
    - Else → increment the trial count.

**🔸 Onlooker Bees Phase:**

* + Calculate selection probability for each bee (based on fitness).
  + Each onlooker bee selects a bee to follow and attempts to improve its solution using neighborhood search.

**🔸 Scout Bees Phase:**

* + Bees with trial count > limit are reinitialized with a new random solution (exploration).

**🔸 Tracking Best Solution:**

* + The best solution is updated if a better one is found.

**🔸 Progress Logging:**

* + Logs best fitness per iteration and optionally runs a callback.

1. **Return**:
   * Best timetable found.
   * History of best fitness per iteration.

**🧾 Pseudocode**

Define Bee(timetable, fitness):

timetable ← input timetable

fitness ← input fitness

trials ← 0

Function generate\_random\_timetable(courses, instructors, rooms, timeslots):

timetable ← empty list

For each course in courses:

instructor ← instructors[course]

room ← randomly select from rooms

timeslot ← randomly select from timeslots

Add (course, instructor, room, timeslot) to timetable

Return timetable

Function neighborhood\_solution(timetable, rooms, timeslots):

Copy timetable to new\_tt

Select random index in new\_tt

Modify room and timeslot at that index randomly

Return new\_tt

Function abc\_optimize(courses, instructors, students, rooms, colony\_size, limit, max\_iter, iteration\_callback):

Initialize timeslots 1 to 25

Create empty bee list

For each bee in colony\_size:

timetable ← generate\_random\_timetable

fitness ← evaluate using fitness\_function

Add Bee(timetable, fitness) to bees

best\_bee ← bee with minimum fitness

history ← empty list

For iteration in 0 to max\_iter:

For each bee in bees: # Employed Bee Phase

new\_tt ← neighborhood\_solution

new\_fit ← fitness\_function(new\_tt)

If new\_fit < bee.fitness:

Update bee's timetable and fitness

Reset bee.trials

Else:

Increment bee.trials

Compute probabilities from bees' fitnesses # Onlooker Phase

For each onlooker:

selected\_bee ← probabilistic selection

new\_tt ← neighborhood\_solution(selected\_bee)

new\_fit ← fitness\_function(new\_tt)

If new\_fit < selected\_bee.fitness:

Update selected\_bee timetable, fitness

Reset trials

For each bee in bees: # Scout Bee Phase

If bee.trials > limit:

Replace bee with new random solution

Update best\_bee if better solution found

Add best\_bee.fitness to history

Print iteration status

If iteration\_callback exists, call it

Return best\_bee.timetable and history

**Genetic Algorithm**

**Core Concepts Used**

* **Chromosome**: A 3D array representing two assignment options for each course.
* **Selection**: Tournament or roulette-wheel.
* **Crossover**: One-point or uniform.
* **Mutation**: Random reset or swap.
* **Fitness Function**: Measures timetable quality (lower is better).

**Pseudocode**

FUNCTION initialize\_individual(num\_courses, num\_timeslots, num\_instructors)

FOR each course

Assign two random [timeslot, instructor] pairs

RETURN individual

CLASS GeneticAlgorithm:

INIT(config, selection\_type, crossover\_type, mutation\_type, seed)

Set config and types

Seed RNG (optional)

Initialize population with random individuals

FUNCTION calculate\_fitness(individual, data)

FOR each course:

Pick first [timeslot, instructor] option

Map course to selected slot

RETURN fitness\_function(timetable)

FUNCTION optimize(data)

best\_fitness ← ∞

best\_solution ← None

FOR generation in max\_generations:

FOR each individual:

individual.fitness ← calculate\_fitness(individual)

Update best\_fitness and best\_solution

IF selection\_type is 'tournament':

selected ← tournament\_selection(population)

ELSE:

selected ← roulette\_selection(population)

new\_population ← []

WHILE new\_population size < population\_size:

Select two parents randomly from selected

Apply crossover to produce two children

Apply mutation to both children

Add children to new\_population

population ← new\_population

Convert best\_solution to JSON format

Save JSON to file

RETURN best\_solution, best\_fitness

**Explanation of Key Parts**

**🔹 Individual Class**

* Each individual has a chromosome of shape (num\_courses, 2, 2).
* For each course: two candidate options: [timeslot, instructor].

**🔹 Initialization**

* Population initialized randomly.
* RNG can be seeded for reproducibility.

**🔹 calculate\_fitness**

* Converts an individual's chromosome to a fixed timetable by selecting the first option for each course.
* Calls a predefined fitness\_function with converted structure.

**🔹 Optimization Loop**

* **Fitness Evaluation**: All individuals get a fitness score.
* **Best Tracking**: Keeps the best chromosome found.
* **Selection**: Picks fitter individuals for reproduction.
* **Crossover**:
  + *One-point*: Cut chromosome at one point and swap parts.
  + *Uniform*: Mix genes randomly from both parents.
* **Mutation**:
  + *Random Reset*: Replace random genes with new values.
  + *Swap*: Swap positions of genes.
* **Repopulation**: Rebuild population with new offspring.
* **Result Saving**: Writes the best timetable to final\_timetable.json.

**Output**

* A .json file: final\_timetable.json contains the best schedule found.
* Format: {course\_index: [timeslot, instructor\_id], ...}

**updated Genetic Algorithm with Memetic Local Search**

**✅ Key Enhancements Over Previous GA Version**

* Chromosomes now store: (course\_id, instructor\_id, room\_id, timeslot)
* **Local Search (Memetic Algorithm)**: Top individuals are locally improved via hill climbing.
* **Flexible GA Operators**: Supports multiple selection, crossover, and mutation strategies.
* **Dynamic Mutation Rate**: Increases when population stagnates.
* **Elitism**: Best 2 individuals are preserved across generations.

**🔁 Pseudocode Summary**

Initialize population of Individuals (each has a timetable chromosome)

FOR each generation:

Evaluate fitness of each individual

Sort population by fitness

Update best solution found

Adjust mutation rate based on stagnation

Apply local search to top individuals (memetic improvement)

Select mating pool (Tournament / Roulette / Rank)

Copy elite individuals (elitism)

FOR rest of population:

Select 2 parents

Crossover with some probability

Mutate with some probability

Add offspring to new population

Replace population

RETURN best timetable and fitness history

**🧠 Cultural Genetic Algorithm – Pseudocode**

**Pseudocode:**

INPUT:

- Courses, Students, Rooms

- Selection, Crossover, Mutation types

- Population size, Generations, Mutation/Crossover rates

- Memetic rate, Acceptance ratio, Influence rate

INITIALIZE:

- Randomly create initial population of individuals (chromosomes)

- Each chromosome assigns each course a (room, timeslot)

- Set best\_global = None, best\_fit\_global = ∞

FOR each generation:

1. EVALUATE FITNESS

FOR each individual in population:

- Calculate fitness using the fitness\_function

2. UPDATE GLOBAL BEST

- Sort population by fitness (lower is better)

- If best individual improves global best:

- Update best\_global and best\_fit\_global

- Reset stagnation counter

- Else:

- Increment stagnation counter

3. BUILD BELIEF SPACE

- Select top X% individuals (based on acceptance\_ratio)

- For each gene (course assignment):

- Compute most frequent room (mode)

- Compute average or median timeslot

- Store (mode\_room, avg\_time) as belief

4. ADAPT MUTATION RATE

- If stagnation > threshold:

- Increase mutation rate

- Else:

- Reset to base mutation rate

5. LOCAL SEARCH (Memetic)

- Apply hill-climbing local search on top memetic\_rate% individuals

6. SELECTION

- Select parent pool using selected method (e.g., tournament)

7. ELITISM

- Copy best 2 individuals to next generation

8. REPRODUCTION LOOP:

WHILE new population is incomplete:

- Select 2 parents randomly from pool

- Perform crossover (based on type & crossover\_rate)

- Apply cultural influence using belief space

- Apply mutation (based on type & mutation\_rate)

- Create new individuals and add to population

9. Replace population with new generation

10. Print progress (generation number, best fitness, mutation rate)

RETURN best\_global, fitness\_history

**🔎 Explanation of Each Component**

**1. Individual Representation (Chromosome)**

Each chromosome is a list of tuples:

(course\_id, instructor\_id, room\_id, timeslot)

Each gene represents one course's assignment in terms of:

* Where and when it is scheduled
* Who is teaching it

**2. Fitness Function**

Evaluates how "good" a timetable is by penalizing:

* Student conflicts (overlapping classes)
* Instructor conflicts
* Room capacity violations
* Room/time collisions

Lower fitness = better timetable.

**3. Belief Space (Cultural Knowledge)**

Derived from the **top-performing individuals** (elite group):

* Captures patterns that seem to work well
* Example: “Course X is often assigned to Room 101 at Slot 3”
* Used to **guide offspring** during reproduction

**4. Cultural Influence**

During reproduction, offspring genes are **nudged toward** values from the belief space:

* E.g., use the elite group's preferred room or average time
* Helps accelerate convergence by focusing the search

**5. Local Search (Memetic Component)**

Applies **hill-climbing** to the top individuals:

* Slightly alters gene values (e.g., change room or time)
* Keeps the change only if fitness improves
* This refines solutions beyond random evolution

**6. Mutation and Crossover Types**

You support various strategies:

* **Mutation**: Random Reset, Swap, Scramble
* **Crossover**: One-point, Two-point, Uniform  
  These increase diversity and recombine knowledge.

**7. Elitism**

Top 2 individuals are **copied directly** to the next generation:

* Prevents loss of the best solutions
* Maintains steady progress

**8. Stagnation Detection**

If fitness hasn’t improved for 10+ generations:

* Mutation rate is increased
* Encourages exploration to escape local optima

**🧠 Genetic Algorithm**

**Pseudocode: Genetic Algorithm**

INPUT:

- Courses, Students, Rooms

- Selection, Crossover, Mutation types

- Population size, Max generations

- Mutation rate, Crossover rate

INITIALIZE:

- Create initial population of Individuals (each a chromosome of course assignments)

- Set best\_global\_solution = None, best\_global\_fitness = ∞

FOR each generation:

1. EVALUATE FITNESS

FOR each individual in population:

2. SORT population by fitness (ascending)

3. UPDATE GLOBAL BEST

- If best fitness < best\_global\_fitness:

- Update best\_global\_solution

- Reset stagnant\_generations

- Else:

- Increment stagnant\_generations

4. ADJUST MUTATION RATE

- If stagnant\_generations > 10:

- Increase mutation rate

- Else if improved:

- Reset to base mutation rate

- Else:

- Decay mutation rate toward base

5. SELECTION

- Select mating pool based on selection\_type

6. ELITISM

- Copy best 2 individuals to next generation

7. REPRODUCTION LOOP

WHILE new population not full:

- Randomly select 2 parents from pool

- Apply crossover with crossover\_rate

- Apply mutation with mutation\_rate

- Add resulting offspring to new population

8. Replace old population with new generation

9. Print generation info: best fitness, mutation rate

RETURN best\_global\_solution, fitness\_history

**Explanation of Key Components**

**1. Chromosome Representation**

Each chromosome is a list of genes:

(course\_id, instructor\_id, room\_id, timeslot)

Each gene represents a single course assignment in the timetable.

**2. Fitness Function**

Defined in fitness\_function():

* Penalizes conflicts:
  + Student overlaps
  + Instructor double-booking
  + Room over-capacity
  + Room/time clashes
* Goal: **minimize total penalty**, so **lower fitness is better**

**3. Elitism**

Top 2 individuals (with best fitness) are **carried over unchanged** to next generation:

* Ensures best solutions aren't lost
* Encourages steady progress

**4. Selection Strategies**

You can choose from:

* **Tournament selection**
* **Roulette wheel selection**
* **Rank selection**

Each selects individuals **probabilistically** based on fitness.

**5. Crossover Strategies**

Mix genes from two parents:

* **One Point**: Split at one point, swap tails
* **Two Point**: Swap a segment
* **Uniform**: Randomly choose genes from each parent

Used with a given crossover\_rate.

**6. Mutation Strategies**

Randomly modify offspring:

* **Random Reset**: Change room/time randomly
* **Swap**: Swap two genes
* **Scramble**: Shuffle a subset of genes

Used with a mutation\_rate that adjusts dynamically.

**7. Dynamic Mutation Rate**

To avoid getting stuck in local minima:

* **If fitness doesn’t improve for >10 generations**:
  + Boost mutation rate (exploration)
* **If improving**:
  + Reset or reduce mutation rate (exploitation)

**Simulated Annealing for Timetabling – Pseudocode**

**Pseudocode: Simulated Annealing**

INPUT:

- Courses, Instructors, Students, Rooms

- Initial temperature, Cooling rate, Max iterations

INITIALIZE:

- Generate a random initial timetable

- Evaluate its fitness

- Set current\_solution = initial\_timetable

- Set best\_solution = current\_solution

- Set temperature = initial\_temp

FOR each iteration up to max\_iter:

1. Generate a neighbor solution (small modification)

2. Evaluate its fitness

3. IF neighbor is better:

- Accept it as current solution

ELSE:

- Accept it probabilistically using:

acceptance\_prob = exp(-(Δfitness) / temperature)

4. IF current solution is better than best:

- Update best\_solution

5. Record best fitness

6. Decrease temperature: temperature \*= cooling\_rate

RETURN best\_solution, fitness\_progress

**Explanation of Components**

**1. Timetable Representation**

Each **timetable** (solution) is a list of tuples:

(course\_id, instructor\_id, room\_id, timeslot)

Each tuple assigns a course to a room and time with its instructor.

**2. Initial Solution**

The algorithm begins with a **randomly generated valid timetable**, assigning:

* a random room
* a random timeslot  
  for each course.

**3. Fitness Function**

Imported from fitness\_function(), it evaluates how good a timetable is:

* Fewer conflicts = lower fitness
* Lower fitness is better

Conflicts can include:

* Room/time overlaps
* Student schedule clashes
* Instructor double-bookings
* Exceeding room capacity

**4. Neighborhood Solution**

A small **perturbation** of the current solution:

* Randomly selects a course assignment
* Changes its room and timeslot

This helps explore the search space incrementally.

**5. Simulated Annealing Logic**

Inspired by metallurgy (cooling molten metal):

* Start with **high temperature** (high exploration)
* Gradually **cool down** (focus on exploitation)

Even **worse solutions** can be accepted early on using:

acceptance\_prob = exp(-(Δfitness) / temperature)

This allows **escaping local minima**.

**🧪 6. Temperature Cooling**

Each iteration:

temperature \*= cooling\_rate

As temperature drops:

* Probability of accepting worse solutions **decreases**
* Algorithm becomes more **greedy**

**Hybrid PCO & SA**

Initialize HybridGPSOSA

↓

Initialize PSO population (Particles)

↓

Run PSO loop (global search):

• Evaluate fitness

• Update velocities & positions

• Track global best

↓

Pass global best to SA

↓

Run SA loop (local refinement):

• Random mutation

• Accept/reject based on fitness and temperature

↓

Return final optimized Individual (chromosome + fitness)

**🧠 Flow of the HybridGPSOSA Code**

**Phase 1 – Global Search via Particle Swarm Optimization (PSO)**

**2.1. PSO Population Initialization**

* A population of Particle objects is created.
* Each Particle originates from an Individual, whose chromosome is a list of:
* (course\_id, instructor\_id, room\_id, timeslot)
* Each particle is assigned a random **velocity**.

**2.2. PSO Main Loop**

* For a fixed number of iterations (or until stagnation):
  + **Evaluate fitness** of each particle using a fitness function.
  + Update **global best solution** if a better one is found.
  + If no improvement over several iterations, break (early stopping).
  + **Update velocity** of each particle using the PSO formula:
    - Inertia, cognitive (personal), and social (global) components.
  + **Update position**: apply velocity to room and timeslot values.

**3. Phase 2 – Local Search via Simulated Annealing (SA)**

**3.1. Initialization**

* Starts from the best solution found by PSO.
* Set current and best solutions to the PSO output.
* Set initial temperature T.

**3.2. SA Main Loop**

* For a fixed number of iterations or until temperature is too low:
  + **Randomly modify one gene** (room or timeslot of a course).
  + Evaluate the fitness of the new (neighbor) solution.
  + **Accept the neighbor** if:
    - It improves fitness (lower is better), or
    - It worsens fitness but satisfies the probabilistic acceptance condition.
  + **Update temperature** using exponential decay.

**4. Final Output**

* The final optimized chromosome from SA is wrapped in an Individual object.
* The final individual holds the best timetable found and its associated fitness.