

# **OptiTime: Intelligent Campus Timetable Optimization**

## **Project Report**

**Course:** Optimization Techniques

**Team Name:** Time Lords

### **Team Members:**

Shivansh Shah (BT2024243)

Ravi Abhinav (BT2024239)

R.V. Pranav (BT2024221)

November 23, 2025

# 1 Problem Statement and Motivation

## 1.1 The Problem

University timetabling is a classic **NP-Hard** combinatorial optimization problem. It involves assigning a set of courses ( $C$ ) to specific time slots ( $T$ ) and rooms ( $R$ ) while satisfying a complex web of conflicting constraints. As student enrollment grows and course options diversify (including electives, labs, and tutorials), the search space for a valid schedule expands exponentially, making manual scheduling infeasible.

## 1.2 Motivation

Manual scheduling approaches are inefficient and prone to human error, often resulting in:

- **Hard Constraint Violations:** Inadvertently double-booking rooms or professors.
- **Student Clashes:** Students unable to take their required combinations of courses due to overlapping slots.
- **Sub-optimal Quality:** While a schedule might be "valid" (feasible), it is often "bad" in practice. Examples include classes running late into the evening causing student fatigue, or professors teaching multiple hours back-to-back without breaks.

**OptiTime** aims to automate this process using **Mixed-Integer Linear Programming (MILP)**. Our goal is not just to find a *feasible* timetable, but an *optimal* one that balances resource usage with human comfort preferences.

# 2 Mathematical Formulation

We modeled the problem using a **Binary Integer Programming** approach.

## 2.1 Decision Variables

We define a binary decision variable  $x_{c,t,r}$  as follows:

$$x_{c,t,r} = \begin{cases} 1, & \text{if course } c \text{ is scheduled at time } t \text{ in room } r \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

## 2.2 Hard Constraints (Must be satisfied)

**1. Course Slot Requirement:** Every course  $c$  must be scheduled exactly  $h_c$  times per week.

$$\sum_{t \in T} \sum_{r \in R} x_{c,t,r} = h_c, \quad \forall c \in C \quad (2)$$

**2. Room Capacity & Conflict:** A room cannot host more than one class at a time, and the room capacity must meet the course enrollment.

$$\sum_{c \in C} x_{c,t,r} \leq 1, \quad \forall t \in T, \forall r \in R \quad (3)$$

Note: We pre-filter variables such that  $x_{c,t,r}$  is defined only if  $\text{Capacity}(r) \geq \text{Enrollment}(c)$ .

**3. Instructor Availability:** An instructor cannot teach two courses simultaneously.

$$\sum_{r \in R} \sum_{c \in \text{Instructor's Courses}} x_{c,t,r} \leq 1, \quad \forall t \in T \quad (4)$$

**4. Student Conflict (The "Clash" Constraint):** If a student takes both course  $c_i$  and  $c_j$ , they cannot be scheduled at the same time.

$$\sum_{r \in R} x_{c_i,t,r} + \sum_{r \in R} x_{c_j,t,r} \leq 1, \quad \forall t \in T \quad (5)$$

## 2.3 Objective Function (Soft Constraints)

Initially, our model focused purely on feasibility. However, to improve schedule quality, we introduced a penalty-based minimization objective:

$$\text{Minimize } Z = W_1 \sum (\text{Time Penalty}) + W_2 \sum (\text{Prof Overload}) \quad (6)$$

- **Student Fatigue ( $W_1$ ):** We assign weights to time slots (e.g., late afternoon slots have higher weights). The solver minimizes the total weight, effectively pushing classes to earlier slots to reduce fatigue.
- **Professor Workload ( $W_2$ ):** We introduce auxiliary variables to penalize instances where a professor teaches more than 2 slots per day.

## 3 Methodology and Solver Details

### 3.1 Tech Stack

- **Language:** Python 3.10
- **Modeling Library:** PuLP (Python LP modeler)
- **Solver:** CBC (Coin-OR Branch and Cut) - An open-source mixed integer programming solver.
- **Data Processing:** Pandas for CSV manipulation and conflict matrix generation.
- **Visualization:** Streamlit & Plotly for interactive dashboards.

### 3.2 Methodology Pipeline

1. **Data Preprocessing:** We ingest `student_data_large.csv` and `courses.csv`. A critical optimization step involves the generation of a **Conflict Matrix**. Instead of checking every student against every constraint, we iterate through student enrollments to identify specific pairs of courses that share students. Only these pairs are added as constraints, significantly reducing solver overhead and model building time.
2. **Model Construction:** We instantiate a `pulp.LpMinimize` problem. Variables are created sparsely (only for valid room capacities) to save memory usage during the optimization process.
3. **Solving:** The model is passed to the **CBC Solver**. We utilize a **Branch and Bound** algorithm to search the solution tree for the optimal set of binary variables.

4. **Validation & Visualization:** The output is audited by `validate.py` to ensure mathematical correctness. A Streamlit dashboard allows students to enter their ID and view their personalized, clash-free schedule.

## 4 Results, Analysis, and Discussion

### 4.1 Results

The model successfully generated a valid timetable for the provided dataset (`student_data_large.csv`).

- **Feasibility:** The audit script (`validate.py`) confirmed **zero** hard constraint violations. All 17 courses received their required slots, and no student had overlapping classes.
- **Optimization:** The penalty model successfully shifted approximately 80% of classes to pre-lunch slots compared to the feasibility-only model, directly addressing the "Student Fatigue" metric.

### 4.2 Challenges and Fixes

During development, we encountered two significant hurdles:

#### 1. The "Quality" Challenge:

- *Issue:* Initially, we lacked an objective function. The solver returned the first valid schedule it found, which was mathematically correct but practically unusable (e.g., classes at 5 PM on Fridays or professors teaching 4 hours straight).
- *Fix:* We implemented `penalty_model.py`. By introducing weighted penalties for late slots and high daily workloads, we forced the solver to search for a "human-friendly" schedule, not just a valid one.

#### 2. The "Data" Challenge:

- *Issue:* Finding a realistic, large-scale dataset that linked specific students to courses was difficult. Random generation resulted in unrealistic graphs where every course clashed with every other course.
- *Fix:* We created a synthetic dataset (`student_data_large.csv`) modeled after our own college structure. We simulated "Batches" (e.g., BTech CSE 2025) where students take common cores and specific electives, creating a realistic density of conflicts.

### 4.3 Future Improvements

To make OptiTime a production-ready system, we propose the following enhancements:

- **Professor-Specific Schedules:** Currently, the visualization focuses on students. We plan to add a dashboard for faculty to view their teaching loads and room locations.
- **Elective Recommender:** Using the generated timetable, we can build a feature that recommends electives to students based on their free slots, ensuring they pick courses that don't clash with their core requirements and align with personal preferences.
- **Removing the Time Limit:** Currently, we impose a 100-second hard limit on the optimizer to ensure rapid feedback during demos. For a final deployment, we would remove this limit or migrate to a cloud-based commercial solver (like Gurobi) to allow for deeper optimization of significantly larger datasets.

## 5 Conclusion

OptiTime demonstrates that MILP is a powerful tool for solving university scheduling problems. By rigorously defining constraints and iteratively improving our objective function, we created a system that saves administrative time and improves the daily lives of students and faculty.