OptiTime: Intelligent Campus Timetable Optimization

Optimization Techniques Course Project

Team: Time Lords

Shivansh Shah (BT2024243)
Ravi Abhinav (BT2024239)
R.V. Pranav (BT2024221)

GitHub Repository: https://github.com/GOW444/OptiTime

November 23, 2025

Contents

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:

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

2.2 Hard Constraints (Must be satisfied)

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

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

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

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

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

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

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

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.

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

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:

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

- **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.

3.3 Code Repository

The full implementation for OptiTime (data generators, models, validator, and frontend) is available on GitHub:
Repository: https://github.com/GOW444/OptiTime

3.4 How to run (Quick start)

Below is a concise workflow to run the project locally. For full details see the repository README.

Requirements

- Python 3.10 (recommended)
- A virtual environment is recommended to isolate dependencies.
- Install Python packages listed in `requirements.txt`.

Step 1 — Clone and prepare environment

```
git clone https://github.com/GOW444/OptiTime.git
cd OptiTime

python3 -m venv venv
# Linux / macOS
source venv/bin/activate
# Windows PowerShell
# .\venv\Scripts\Activate.ps1

pip install -r requirements.txt
```

Step 2 — Generate data

```
# produces files in the pwd
python generate_data.py
# (or for larger datasets)
python gen_data_comp.py
```

Step 3 — Run models

```
# Feasibility-only model (checks hard constraints)
python new_model.py
```

```
# Output: new_timetable_output.json

# Penalty/optimization model (minimizes student fatigue, prof overload)
python penalty_model.py
# Output: timetable_output.json
```

Step 4 — Validate the generated timetable

```
python validate.py
# Validator checks slot counts, student clashes, professor workloads, etc.
```

Step 5 — Visualize (Streamlit frontend)

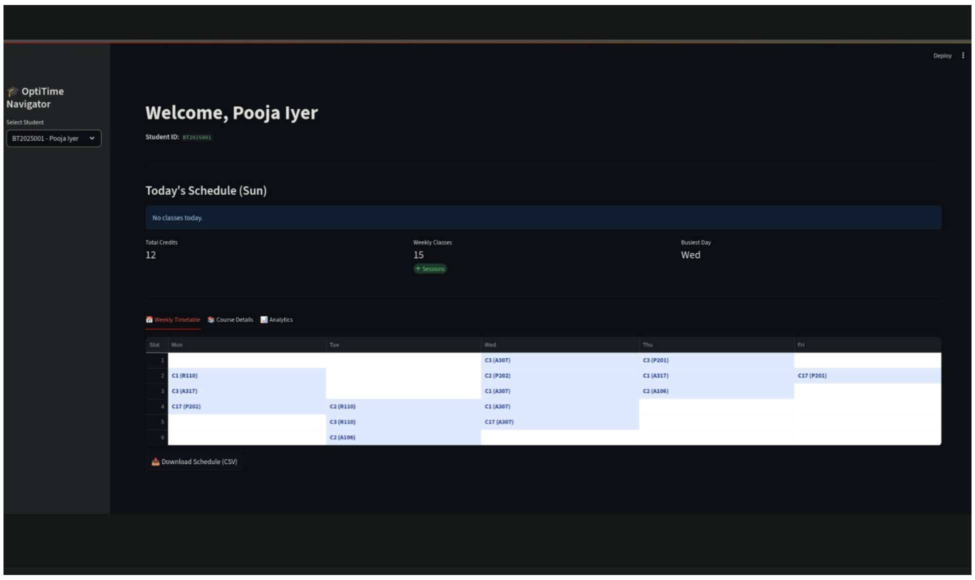
```
# from project root
streamlit run frontend/visualize.py
# or if file is at root
# streamlit run visualize.py
```

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.



Welcome, Pooja Iyer

Student ID: BT2025001

Today's Schedule (Sun)

No classes today.

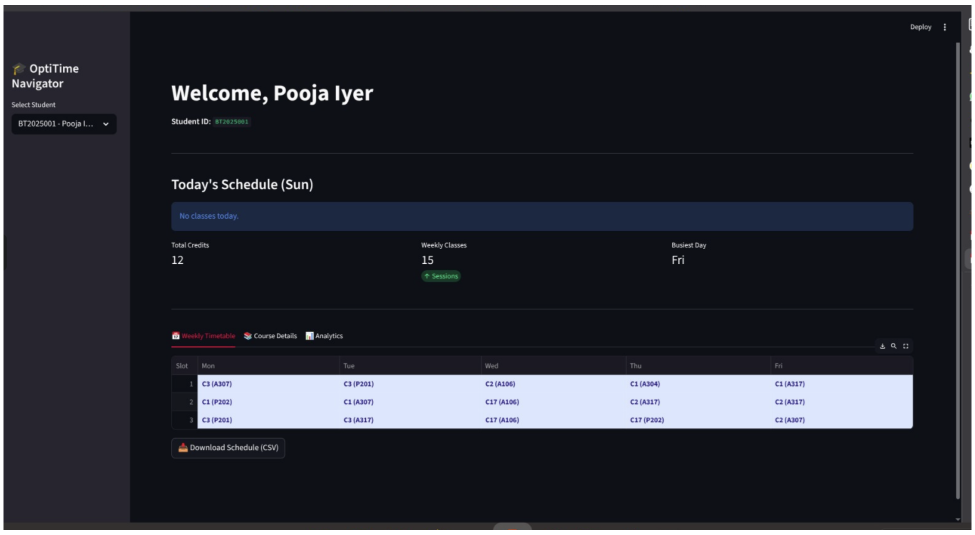
Total Credits 12

Weekly Classes 15 ↑ Sessions

Busiest Day Wed

Slot	Mon	Tue	Wed	Thu	Fri
1			C3 (A307)	C3 (P201)	
2	C1 (R110)		C2 (P202)	C1 (A317)	C17 (P201)
3	C3 (A317)		C1 (A307)	C2 (A106)	
4	C17 (P202)	C2 (R110)	C1 (A307)		
5		C3 (R110)	C17 (A307)		
6		C2 (A106)			

Figure 1: Timetable without using objective function (only feasibility is checked)



Welcome, Pooja Iyer

Student ID: BT2025001

Today's Schedule (Sun)

No classes today.

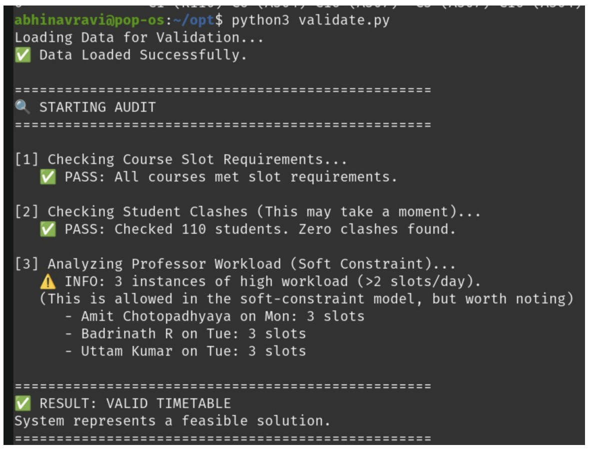
Total Credits 12

Weekly Classes 15 ↑ Sessions

Busiest Day Fri

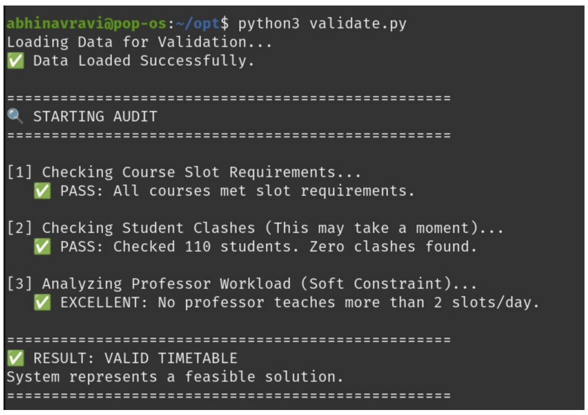
Slot	Mon	Tue	Wed	Thu	Fri
1	C3 (A307)	C3 (P201)	C2 (A106)	C1 (A304)	C1 (A317)
2	C1 (P202)	C1 (A307)	C17 (A106)	C2 (A317)	C2 (A317)
3	C3 (P201)	C3 (A317)	C17 (A106)	C17 (P202)	C2 (A307)

Figure 2: Timetable after adding objective function (Penalty Terms - Student Fatigue, Prof Overload)



```
abhinavravi@pop-os:~/opt$ python3 validate.py
Loading Data for Validation...
✅ Data Loaded Successfully.

==================================================
🔍 STARTING AUDIT
==================================================

[1] Checking Course Slot Requirements...
   ✅ PASS: All courses met slot requirements.

[2] Checking Student Clashes (This may take a moment)...
   ✅ PASS: Checked 110 students. Zero clashes found.

[3] Analyzing Professor Workload (Soft Constraint)...
   ⚠️ INFO: 3 instances of high workload (>2 slots/day).
   (This is allowed in the soft-constraint model, but worth noting)
     - Amit Chotopadhyaya on Mon: 3 slots
     - Badrinath R on Tue: 3 slots
     - Uttam Kumar on Tue: 3 slots

==================================================
✅ RESULT: VALID TIMETABLE
System represents a feasible solution.
==================================================
```

Figure 3: *
Before Objective Function



```
abhinavravi@pop-os:~/opt$ python3 validate.py
Loading Data for Validation...
✅ Data Loaded Successfully.

==================================================
🔍 STARTING AUDIT
==================================================

[1] Checking Course Slot Requirements...
   ✅ PASS: All courses met slot requirements.

[2] Checking Student Clashes (This may take a moment)...
   ✅ PASS: Checked 110 students. Zero clashes found.

[3] Analyzing Professor Workload (Soft Constraint)...
   ✅ EXCELLENT: No professor teaches more than 2 slots/day.

==================================================
✅ RESULT: VALID TIMETABLE
System represents a feasible solution.
==================================================
```

Figure 4: *
After Objective Function

Figure 5: Comparison of timetable before and after adding penalty-based objective

4.2 Challenges and Fixes

During the development lifecycle, we encountered and resolved two primary challenges:

1. The Objective Function Challenge:

- **Issue:** Initially, our model lacked an objective function. The solver simply returned the first valid schedule it found. While mathematically correct, the results were practically inefficient, often placing classes late on Fridays or scheduling professors for 4-hour continuous blocks.
- **Fix:** We implemented `penalty_model.py`. By introducing weighted penalties for late time slots and daily workload limits, we transformed the problem from simple satisfaction to optimization, significantly reducing student fatigue metrics.

2. **The Dataset Generation Challenge:**
    - **Issue:** Creating a realistic large-scale dataset was difficult. Purely random generation resulted in unrealistic conflict graphs where almost every course clashed with every other course, making the problem infeasible.
    - **Fix:** We used our own college curriculum structure as motivation. We simulated "Batches" (e.g., BTech CSE 2025) where groups of students take common core courses and specific electives. This created a realistic density of conflicts that mirrored real-world scenarios.

5 Future Improvements

To transition OptiTime from a prototype to a production-ready system, we propose the following enhancements:

1. **Professor-Specific Dashboards:** Currently, the visualization focuses on student schedules. We plan to implement a view specifically for faculty to track their weekly teaching load and room assignments.
2. **Smart Elective Recommendation:** Using the generated timetable as a base, we can build a recommendation engine that suggests electives to students based on slot availability (ensuring zero clashes) and personal preference alignment.
3. **Removing Solver Hard Limits:** Currently, we enforce a 100-second hard time limit on the optimizer to ensure rapid feedback during demonstrations. For a production environment, we would remove this limit or migrate to a commercial cloud-based solver (like Gurobi) to allow for deeper exploration of the solution space for larger datasets.

6 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 minimizes administrative burden while maximizing the quality of life for students and faculty.