Optimization of University Timetabling Using Constraint Satisfaction Problem

Farrel Athalla Putra - 13523118[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]farrelxag@gmail.com, 13523118@std.stei.itb.ac.id

***Abstract*—University timetabling is a complex problem involving multiple constraints such as lecturer availability, student preferences, and classroom capacities. This paper presents an approach to solving the university scheduling problem using Constraint Satisfaction Problem (CSP) techniques. Each class is modeled as a variable, with domains representing available time slots and rooms, while constraints ensure no conflicts between lecturer schedules, overlapping student enrollments, and room capacities. The CSP is solved using backtracking with constraint propagation to ensure efficient and feasible solutions. Results demonstrate the approach's capability to generate conflict-free timetables while optimizing resource utilization. This method showcases the potential of CSP for scalable and flexible scheduling in academic institutions.**

***Keywords*—class scheduling, Constraint Satisfaction Problem, optimization, timetabling.**

I. Introduction

Efficient scheduling is a cornerstone of effective academic management, particularly in universities where multiple constraints must be balanced. The scheduling process must accommodate lecturers, students, and classrooms without creating conflicts, such as overlapping schedules or exceeding room capacities. A well-designed timetable ensures that academic activities run smoothly, maximizing the utilization of resources while minimizing disruptions. Conversely, a poorly planned schedule can lead to inefficiencies, reduced productivity, and dissatisfaction among stakeholders.

To tackle the complexity of university timetabling, computational methods have emerged as essential tools. One prominent approach is the use of Constraint Satisfaction Problems (CSP), a framework that defines the problem in terms of variables, domains, and constraints. In the context of scheduling, variables represent classes, domains correspond to available time slots and rooms, and constraints ensure that conflicts are avoided. This structured methodology facilitates the generation of feasible and conflict-free timetables.

At the heart of CSP lies graph theory, a fundamental branch of discrete mathematics that serves as the backbone of the scheduling process. Graphs represent the relationships between variables, where nodes signify classes and edges signify conflicts, such as shared lecturers or students. By leveraging graph-theoretic principles, CSP can detect and resolve conflicts efficiently. For instance, graph coloring techniques assign time slots (colors) to nodes while ensuring that no adjacent nodes share the same color. This approach guarantees that conflicting classes are scheduled at different times, making it a powerful solution for complex scheduling challenges.

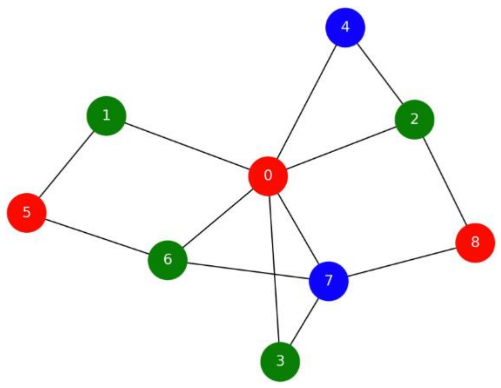


Fig 1.1 Constraint Satisfaction Problem Graph
(Taken from Abhijeet Nayak's Medium)

II. Theoretical Basis

A. Graph

Graphs are generally used to represent discrete objects and the relationships between those objects. A graph G is defined as $G = (V, E)$, where V is a non-empty set of vertices (or nodes), defined as $V = \{v_1, v_2, \dots, v_n\}$ and E is a set of edges that connects pairs of vertices, defined as $E = \{e_1, e_2, \dots, e_n\}$. The set V must not be empty, meaning a graph cannot exist without vertices, but the set E can be empty, meaning a graph is allowed to have no edges.

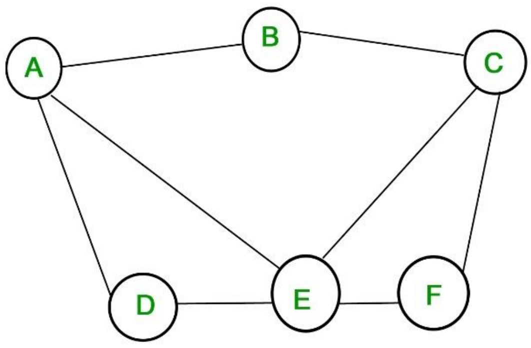


Fig 2.1 Example of Graph
(Taken from GeeksforGeeks)

Based on the presence or absence of loops or multiple edges in a graph, graphs are classified into two types:

a. Simple Graph
   A simple graph is a graph that does not contain multiple edges (parallel edges) or loops (edges that connect a vertex to itself).
b. Non-Simple Graph
   A non-simple graph contains either:
   1. Multi-Graph
      A graph with multiple edges (parallel edges) between the same pair of vertices.
   2. Pseudo-Graph
      A graph containing at least one loop, where an edge starts and ends at the same vertex.

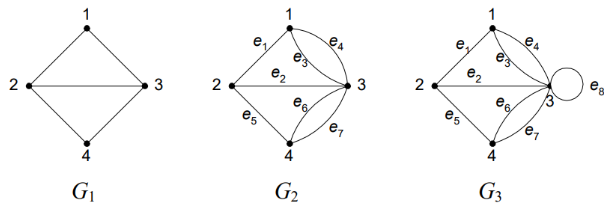


Fig 2.2. a) Simple Graph b) Multi-Graph c) Pseudo-Graph (Taken from Rinaldi Munir Webpage)

In G_2, the edges $e_3 = (1,3)$ and $e_4 = (1,3)$ are called multiple edges (or parallel edges) because both edges connect the same pair of vertices, namely vertex 1 and vertex 3. In G_3, the edge $e_8 = (3,3)$ is called a loop because it starts and ends at the same vertex, namely vertex 3.

Graph can also be classified based on whether the edges have a direction:

a. Undirected Graph
   A graph where the edges have no direction, meaning the relationship between connected vertices is bidirectional.

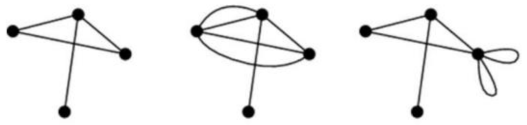


Fig 2.3 Undirected Graph
(Taken from Rinaldi Munir Webpage)

b. Directed Graph (Digraph)
   A graph where each edge has a direction, indicating a one-way relationship between the vertices it connects.

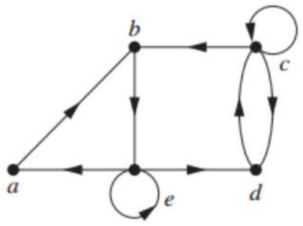


Fig 2.4 Directed Graph
(Taken from Rinaldi Munir Webpage)

Graphs have various terminologies that describe their structural properties:

a. Adjacency

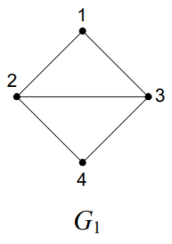


Fig 2.5 Graph with Adjacent Vertices
(Taken from Rinaldi Munir Webpage)

Adjacency refers to two vertices being directly connected by an edge. For instance, in graph G_1, vertex 1 is adjacent to vertices 2 and 3, but not to vertex 4.

b. Incidency

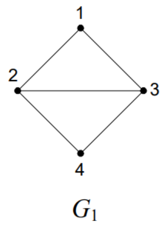


Fig 2.6 Graph with Incidents
(Taken from Rinaldi Munir Webpage)

Incidency describes the relationship between an edge and the vertices it connects. For example, in G_1, edge $(2,3)$ is incident to vertices 2 and 3, while edge $(1,2)$ is not incident to vertex 4.

c. Isolated Vertex

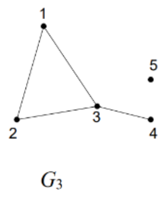


Fig 2.7 Graph with Isolated Vertex
(Taken from Rinaldi Munir Webpage)

A vertex with no edges is called isolated vertex, as seen in graph G_3, where vertex 5 is isolated.

d. Null Graph (Empty Graph)

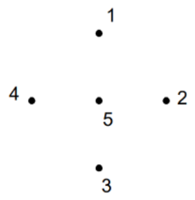


Fig 2.8 Null Graph
(Taken from Rinaldi Munir Webpage)

A graph with no edges at all is called a null graph or empty graph.

e. Degree

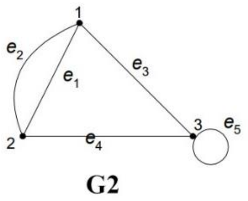


Fig 2.9 Degree in Graph
(Taken from Rinaldi Munir Webpage)

The degree of a vertex is defined as the number of edges incident to it, denoted by $d(v)$. For example, in graph G_2, vertex 1 has a degree of $d(1) = 3$, as it is connected to multiple edges, and vertex 3 has a degree of $d(3) = 4$, due to the presence of both multiple edges and a loop.

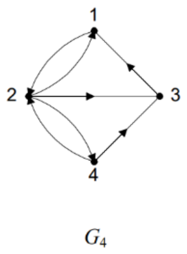


Fig 2.10 Degree in Directed Graph
(Taken from Rinaldi Munir Webpage)

In the directed graph, the degree of a vertex is further divided into in-degree and out-degree. In graph G_4, $d_{in}(1) = 2$; $d_{out}(1) = 2$, $d_{in}(2) = 2$; $d_{out}(2) = 3$, $d_{in}(3) = 2$; $d_{out}(3) = 1$, $d_{in}(4) = 1$; $d_{out}(4) = 2$.

f. Path
A path of length n from the initial vertex v_0 to the destination vertex v_n in a graph G is a sequence of alternating vertices and edges in the form $v_0, e_1, v_1, e_2, v_2 \dots, v_{n-1}, e_n, v_n$, such that $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$,…, $e_n = (v_{n-1}, v_n)$ are edges of the graph G.

Special types of graphs play an important role in graph theory, with each type exhibiting unique characteristics:

a. Complete Graph
A complete graph is a simple graph in which every vertex is connected to every other vertex. A complete graph with n vertices is denoted by K_n.

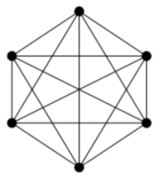


Fig 2.11 Example of a Complete Graph
(Taken from Rinaldi Munir Webpage)

b. Cycle Graph
A cycle graph is a simple graph in which every vertex has a degree of two. A cycle graph with n vertices is denoted by C_n.

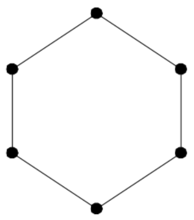


Fig 2.12 Example of a Cycle Graph
(Taken from Rinaldi Munir Webpage)

c. Bipartite Graph
A bipartite graph is a graph G in which its set of vertices can be divided into two disjoint subsets V_1 and V_2, such that every edge in G connects a vertex in V_1 to a vertex in V_2. This type of graph is denoted as $G(V_1, V_2)$.

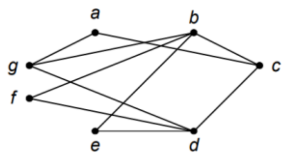


Fig 2.13 Example of a Bipartite Graph
(Taken from Rinaldi Munir Webpage)

B. Graph Coloring

Graph coloring is the assignment of labels, commonly referred to as "colors," to elements of a graph, subject to certain constraints. The most typical form is vertex coloring, where colors are assigned to vertices such that no two adjacent vertices have the same color. Formally, for a graph $G = (V, E)$, a coloring is a function $f: V \to C$, where C is the set of colors, such that for every edge $(u, v) \in E, f(u) \neq f(v)$.

The chromatic number of a graph, denoted by $\chi(G)$, is the is the smallest number of colors needed to color the graph while satisfying the coloring constraints. For example:

a. A complete graph K_n has $\chi(K_n) = n$, as every vertex is adjacent to all other vertices.
b. A cycle graph C_n has $\chi(C_n) = 2$ if n is even and $\chi(C_n) = 3$ if n is odd.
c. A bipartite graph $K_{m,n}$ has $\chi(G) = 2$, where one color is assigned to the vertices in set V_1 and another color is assigned to the vertices in set V_2.

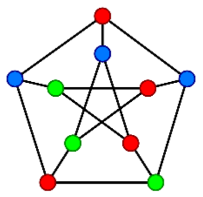


Fig 2.14 Example of Graph Coloring
(Taken from Wikipedia)

III. Approach and Methodology

The scheduling system developed in this paper utilizes Constraint Satisfaction Problems (CSP) to create conflict-free university timetables. The primary objective is to assign time slots and classrooms to courses while satisfying constraints related to lecturers, students, classroom capacities, and course durations. CSP models this problem by defining courses as variables X_c, with their domains $D(X_c)$ consisting of all feasible combinations of time slots and classrooms:

$$D(X_c) = \{(t_1, R_1), (t_2, R_2), \ldots, (t_m, R_k)\}.$$

Constraints are represented mathematically to ensure valid assignments. For example, the classroom capacity constraint ensures that the assigned classroom can accommodate the number of students enrolled in the course:

$$Capacity(R) \geq Students(C).$$

Non-overlapping schedules are enforced to ensure no lecturer or student is assigned to multiple courses at the same time:

$$\begin{gathered} t_i \neq t_j \\ \forall i, j \in Courses \quad if\ Lecturer(i) = Lecturer(j) \\ or\ Students(i) \cap Students(j) \neq \emptyset. \end{gathered}$$

Additionally, classrooms must be available for only one course during a time slot:

$$Room(t, C) \neq Room(t, C') \qquad \forall C, C' \quad if\ t_c = t_{c'}$$

and the course duration must be satisfied:

$$t_{end}(C) - t_{start}(C) = Duration(C).$$

The CSP solution begins with the construction of a constraint graph, where nodes represent courses and edges represent conflicts such as shared lecturers or overlapping students. If graph coloring is applied, the chromatic number $\chi(G)$ determines the minimum number of time slots required $\chi(G) \leq n$, where n is the number of available time slots. Domain reduction techniques refine possible assignments, such as removing time slots where classroom capacities are insufficient. Constraint propagation, such as Arc Consistency (AC-3), ensures that for each value x in a domain $D(X)$, there exists a value y in a connected variable's domain $D(Y)$ that satisfies their constraint:

$$\forall x \in D(X), \exists y \in D(Y) \text{ such that } (x, y) \text{ satisfies } C(X, Y).$$

A backtracking search algorithm is then used to assign time slots and classrooms, enhanced by heuristics such as Minimum Remaining Values (MRV), which prioritizes variables with the fewest valid options, and the Degree Heuristic, which prioritizes variables with the most constraints. Optimization is incorporated to minimize or maximize key objectives, such as minimizing the number of time slots used:

$$min \sum_{t \in Tiime\ Slots} \delta(t) = \begin{cases} 1 & if\ t\ is\ used, \\ 0 & otherwise. \end{cases}$$

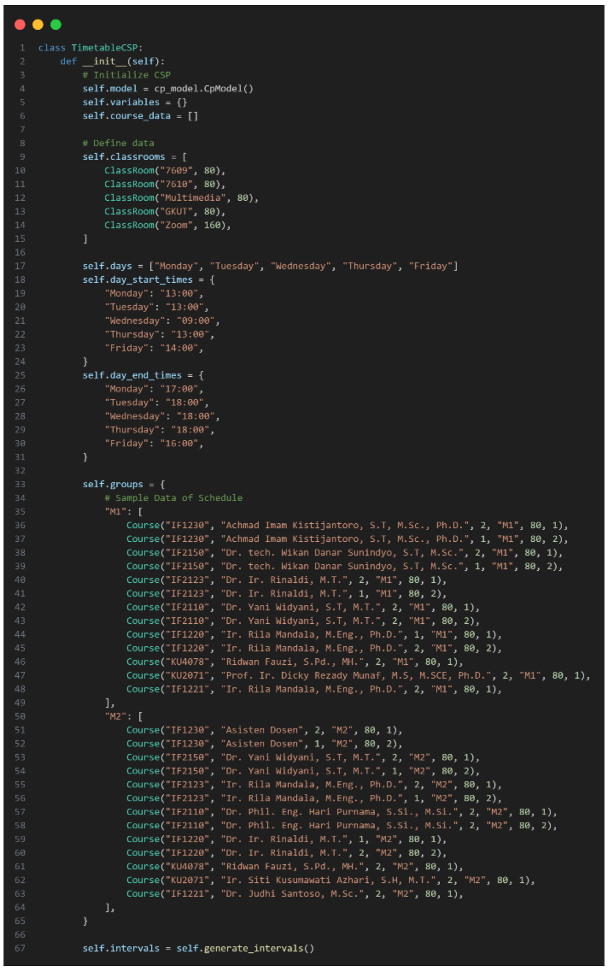
or maximizing classroom utilization:

$$max \sum_{C \in Courses} \frac{Students(C)}{Capacity(Room(C))}.$$

Once all variables are assigned, the system verifies the schedule to ensure no constraints are violated.

IV. Constraint Satisfaction Problem

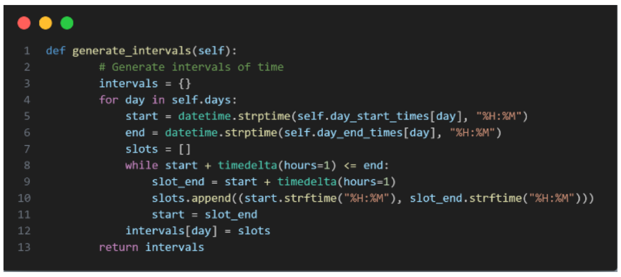
To test the program, we need to create a sample dataset of schedules and classrooms. In this case, I am using a schedule and classroom data from Informatics Engineering students in their 3rd semester at ITB for 2024, divided into two groups. Group M1 consists of students with odd NIMs, while group M2 includes those with even NIMs. The program will be tested using this dataset to evaluate its performance and explore potential alternatives to the current schedule. The sample schedule and group data can be modified if it follows the structure of the 'ClassRoom' and 'Course' class objects, as well as the specified time interval format.



```python
class TimetableCSP:
    def __init__(self):
        # Initialize CSP
        self.model = cp_model.CpModel()
        self.variables = {}
        self.course_data = []

        # Define data
        self.classrooms = [
            ClassRoom("7609", 80),
            ClassRoom("7610", 80),
            ClassRoom("Multimedia", 80),
            ClassRoom("GKUT", 80),
            ClassRoom("Zoom", 160),
        ]

        self.days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
        self.day_start_times = {
            "Monday": "13:00",
            "Tuesday": "13:00",
            "Wednesday": "09:00",
            "Thursday": "13:00",
            "Friday": "14:00",
        }
        self.day_end_times = {
            "Monday": "17:00",
            "Tuesday": "18:00",
            "Wednesday": "18:00",
            "Thursday": "18:00",
            "Friday": "16:00",
        }

        self.groups = {
            # Sample Data of Schedule
            "M1": [
                Course("IF1230", "Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D.", 2, "M1", 80, 1),
                Course("IF1230", "Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D.", 1, "M1", 80, 2),
                Course("IF2150", "Dr. tech. Wikan Danar Sunindyo, S.T, M.Sc.", 2, "M1", 80, 1),
                Course("IF2150", "Dr. tech. Wikan Danar Sunindyo, S.T, M.Sc.", 1, "M1", 80, 2),
                Course("IF2123", "Dr. Ir. Rinaldi, M.T.", 2, "M1", 80, 1),
                Course("IF2123", "Dr. Ir. Rinaldi, M.T.", 1, "M1", 80, 2),
                Course("IF2110", "Dr. Yani Widyani, S.T, M.T.", 2, "M1", 80, 1),
                Course("IF2110", "Dr. Yani Widyani, S.T, M.T.", 2, "M1", 80, 2),
                Course("IF1220", "Ir. Rila Mandala, M.Eng., Ph.D.", 1, "M1", 80, 1),
                Course("IF1220", "Ir. Rila Mandala, M.Eng., Ph.D.", 2, "M1", 80, 2),
                Course("KU4078", "Ridwan Fauzi, S.Pd., MH.", 2, "M1", 80, 1),
                Course("KU2071", "Prof. Ir. Dicky Rezady Munaf, M.S, M.SCE, Ph.D.", 2, "M1", 80, 1),
                Course("IF1221", "Ir. Rila Mandala, M.Eng., Ph.D.", 2, "M1", 80, 1),
            ],
            "M2": [
                Course("IF1230", "Asisten Dosen", 2, "M2", 80, 1),
                Course("IF1230", "Asisten Dosen", 1, "M2", 80, 2),
                Course("IF2150", "Dr. Yani Widyani, S.T, M.T.", 2, "M2", 80, 1),
                Course("IF2150", "Dr. Yani Widyani, S.T, M.T.", 1, "M2", 80, 2),
                Course("IF2123", "Ir. Rila Mandala, M.Eng., Ph.D.", 2, "M2", 80, 1),
                Course("IF2123", "Ir. Rila Mandala, M.Eng., Ph.D.", 1, "M2", 80, 2),
                Course("IF2110", "Dr. Phil. Eng. Hari Purnama, S.Si., M.Si.", 2, "M2", 80, 1),
                Course("IF2110", "Dr. Phil. Eng. Hari Purnama, S.Si., M.Si.", 2, "M2", 80, 2),
                Course("IF1220", "Dr. Ir. Rinaldi, M.T.", 1, "M2", 80, 1),
                Course("IF1220", "Dr. Ir. Rinaldi, M.T.", 2, "M2", 80, 2),
                Course("KU4078", "Ridwan Fauzi, S.Pd., MH.", 2, "M2", 80, 1),
                Course("KU2071", "Ir. Siti Kusumawati Azhari, S.H, M.T.", 2, "M2", 80, 1),
                Course("IF1221", "Dr. Judhi Santoso, M.Sc.", 2, "M2", 80, 1),
            ],
        }

        self.intervals = self.generate_intervals()
```

Fig 4.1 Sample Data of Schedule

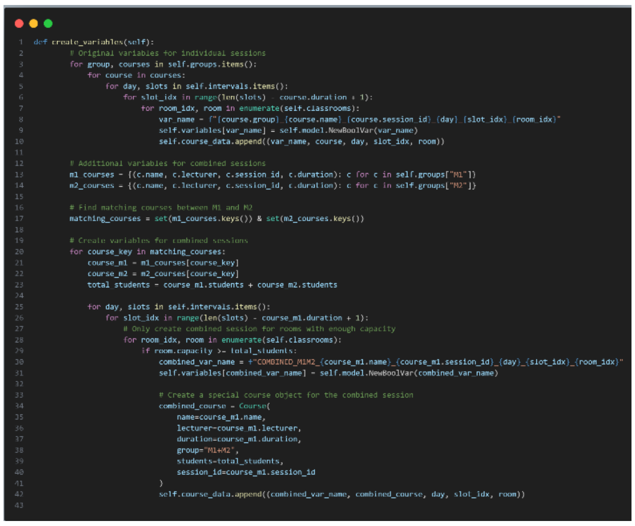
To align the required course durations with the available time slots, the format is converted from single time slots to multiple time intervals, ensuring that the total duration matches the requirements of all courses.



```python
def generate_intervals(self):
        # Generate intervals of time
        intervals = {}
        for day in self.days:
            start = datetime.strptime(self.day_start_times[day], "%H:%M")
            end = datetime.strptime(self.day_end_times[day], "%H:%M")
            slots = []
            while start + timedelta(hours=1) <= end:
                slot_end = start + timedelta(hours=1)
                slots.append((start.strftime("%H:%M"), slot_end.strftime("%H:%M")))
                start = slot_end
            intervals[day] = slots
        return intervals
```

Fig 4.2 Create Intervals of Time

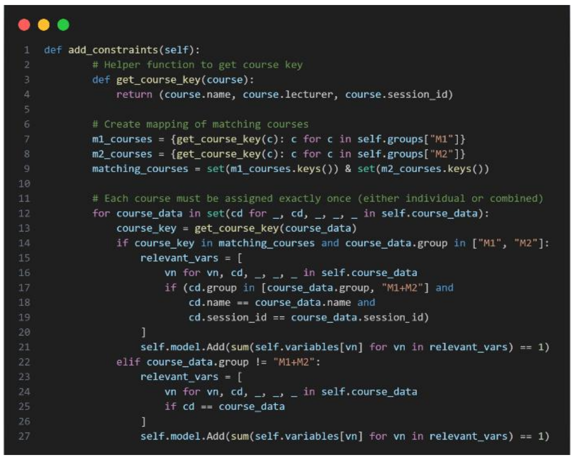
Next, create variables for each course and for combined courses. Combined courses refer to instances where the same course with the same lecturer can be conducted simultaneously in a classroom that accommodates the total capacity of the students. This step is essential for evaluating multiple possible combinations effectively.



```python
def create_variables(self):
        # Original variables for individual sessions
        for group, courses in self.groups.items():
            for course in courses:
                for day, slots in self.intervals.items():
                    for slot_idx in range(len(slots) - course.duration + 1):
                        for room_idx, room in enumerate(self.classrooms):
                            var_name = f"{course.group}_{course.name}_{course.session_id}_{day}_{slot_idx}_{room_idx}"
                            self.variables[var_name] = self.model.NewBoolVar(var_name)
                            self.course_data.append((var_name, course, day, slot_idx, room))

        # Additional variables for combined sessions
        m1_courses = {(c.name, c.lecturer, c.session_id, c.duration): c for c in self.groups["M1"]}
        m2_courses = {(c.name, c.lecturer, c.session_id, c.duration): c for c in self.groups["M2"]}

        # Find matching courses between M1 and M2
        matching_courses = set(m1_courses.keys()) & set(m2_courses.keys())

        # Create variables for combined sessions
        for course_key in matching_courses:
            course_m1 = m1_courses[course_key]
            course_m2 = m2_courses[course_key]
            total_students = course_m1.students + course_m2.students

            for day, slots in self.intervals.items():
                for slot_idx in range(len(slots) - course_m1.duration + 1):
                    # Only create combined session for rooms with enough capacity
                    for room_idx, room in enumerate(self.classrooms):
                        if room.capacity >= total_students:
                            combined_var_name = f"COMBINED_M1M2_{course_m1.name}_{course_m1.session_id}_{day}_{slot_idx}_{room_idx}"
                            self.variables[combined_var_name] = self.model.NewBoolVar(combined_var_name)

                            # Create a special course object for the combined session
                            combined_course = Course(
                                name=course_m1.name,
                                lecturer=course_m1.lecturer,
                                duration=course_m1.duration,
                                group="M1+M2",
                                students=total_students,
                                session_id=course_m1.session_id
                            )
                            self.course_data.append((combined_var_name, combined_course, day, slot_idx, room))
```

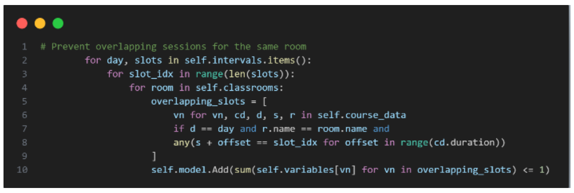
Fig 4.3 Create Variables

Creating a university schedule involves considering numerous constraints that vary depending on the institution. In this case, we implement universal constraints as practiced at ITB. Each course must be assigned a schedule to ensure students can attend the courses they have selected. No room can host two different courses simultaneously, as this would disrupt the focus of the students. Similarly, each student group cannot be scheduled for two or more different courses at the same time to prevent overlapping, which would make it impossible for students to attend both. Lecturers must also avoid overlapping teaching schedules to ensure they can conduct all assigned classes effectively. Additionally, courses have specific credit requirements that determine their duration. For instance, a 4-credit course requires 4 hours of class time, often divided into two 2-hour sessions, which cannot occur on the same day. Lastly, classrooms must meet the capacity requirements of the students to ensure an optimal teaching and learning experience.



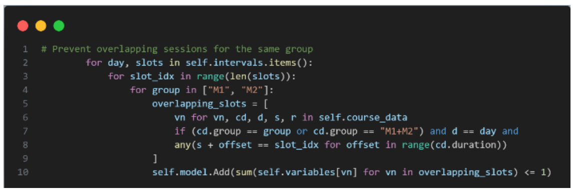
```python
def add_constraints(self):
        # Helper function to get course key
        def get_course_key(course):
            return (course.name, course.lecturer, course.session_id)

        # Create mapping of matching courses
        m1_courses = {get_course_key(c): c for c in self.groups["M1"]}
        m2_courses = {get_course_key(c): c for c in self.groups["M2"]}
        matching_courses = set(m1_courses.keys()) & set(m2_courses.keys())

        # Each course must be assigned exactly once (either individual or combined)
        for course_data in set(cd for _, cd, _, _, _ in self.course_data):
            course_key = get_course_key(course_data)
            if course_key in matching_courses and course_data.group in ["M1", "M2"]:
                relevant_vars = [
                    vn for vn, cd, _, _, _ in self.course_data
                    if (cd.group in [course_data.group, "M1+M2"] and
                        cd.name == course_data.name and
                        cd.session_id == course_data.session_id)
                ]
                self.model.Add(sum(self.variables[vn] for vn in relevant_vars) == 1)
            elif course_data.group != "M1+M2":
                relevant_vars = [
                    vn for vn, cd, _, _, _ in self.course_data
                    if cd == course_data
                ]
                self.model.Add(sum(self.variables[vn] for vn in relevant_vars) == 1)
```

Fig 4.4 Implementation of Overlapping Course Constraint



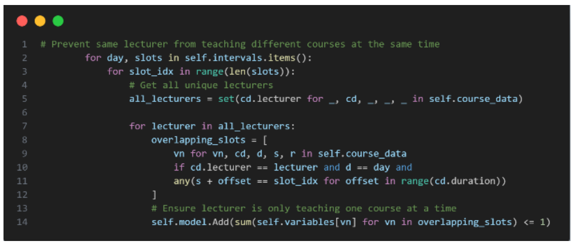
```python
# Prevent overlapping sessions for the same room
        for day, slots in self.intervals.items():
            for slot_idx in range(len(slots)):
                for room in self.classrooms:
                    overlapping_slots = [
                        vn for vn, cd, d, s, r in self.course_data
                        if d == day and r.name == room.name and
                        any(s + offset == slot_idx for offset in range(cd.duration))
                    ]
                    self.model.Add(sum(self.variables[vn] for vn in overlapping_slots) <= 1)
```

Fig 4.5 Implementation of Overlapping Room Constraint



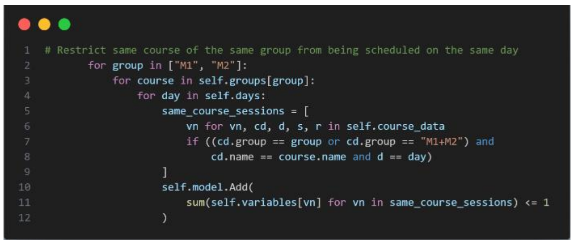
```python
# Prevent overlapping sessions for the same group
        for day, slots in self.intervals.items():
            for slot_idx in range(len(slots)):
                for group in ["M1", "M2"]:
                    overlapping_slots = [
                        vn for vn, cd, d, s, r in self.course_data
                        if (cd.group == group or cd.group == "M1+M2") and d == day and
                        any(s + offset == slot_idx for offset in range(cd.duration))
                    ]
                    self.model.Add(sum(self.variables[vn] for vn in overlapping_slots) <= 1)
```

Fig 4.6 Implementation of Overlapping Group Constraint



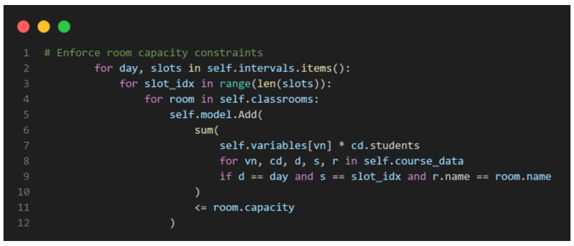
```python
# Prevent same lecturer from teaching different courses at the same time
        for day, slots in self.intervals.items():
            for slot_idx in range(len(slots)):
                # Get all unique lecturers
                all_lecturers = set(cd.lecturer for _, cd, _, _, _ in self.course_data)

                for lecturer in all_lecturers:
                    overlapping_slots = [
                        vn for vn, cd, d, s, r in self.course_data
                        if cd.lecturer == lecturer and d == day and
                        any(s + offset == slot_idx for offset in range(cd.duration))
                    ]
                    # Ensure lecturer is only teaching one course at a time
                    self.model.Add(sum(self.variables[vn] for vn in overlapping_slots) <= 1)
```

Fig 4.7 Implementation of Overlapping Lecturer Constraint



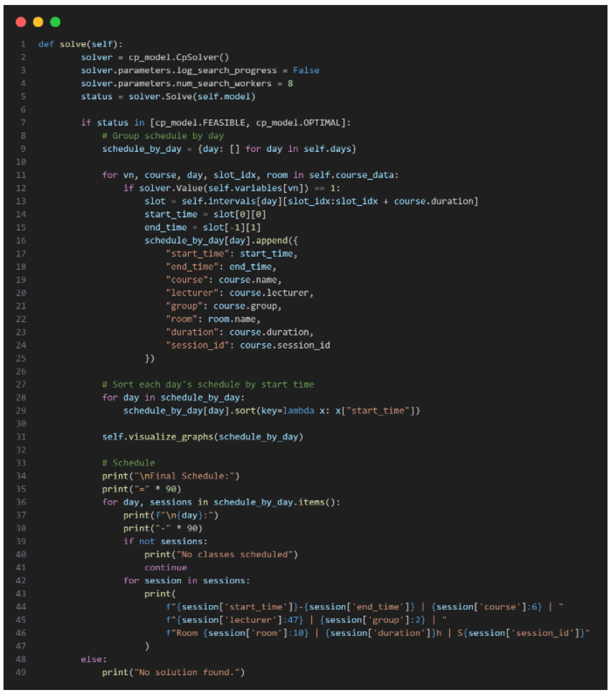
```python
# Restrict same course of the same group from being scheduled on the same day
        for group in ["M1", "M2"]:
            for course in self.groups[group]:
                for day in self.days:
                    same_course_sessions = [
                        vn for vn, cd, d, s, r in self.course_data
                        if ((cd.group == group or cd.group == "M1+M2") and
                            cd.name == course.name and d == day)
                    ]
                    self.model.Add(
                        sum(self.variables[vn] for vn in same_course_sessions) <= 1
                    )
```

Fig 4.8 Implementation of Course Constraint



```python
# Enforce room capacity constraints
        for day, slots in self.intervals.items():
            for slot_idx in range(len(slots)):
                for room in self.classrooms:
                    self.model.Add(
                        sum(
                            self.variables[vn] * cd.students
                            for vn, cd, d, s, r in self.course_data
                            if d == day and s == slot_idx and r.name == room.name
                        )
                        <= room.capacity
                    )
```

Fig 4.9 Implementation of Room Capacity Constraint



```python
def solve(self):
        solver = cp_model.CpSolver()
        solver.parameters.log_search_progress = False
        solver.parameters.num_search_workers = 8
        status = solver.Solve(self.model)

        if status in [cp_model.FEASIBLE, cp_model.OPTIMAL]:
            # Group schedule by day
            schedule_by_day = {day: [] for day in self.days}

            for vn, course, day, slot_idx, room in self.course_data:
                if solver.Value(self.variables[vn]) == 1:
                    slot = self.intervals[day][slot_idx:slot_idx + course.duration]
                    start_time = slot[0][0]
                    end_time = slot[-1][1]
                    schedule_by_day[day].append({
                        "start_time": start_time,
                        "end_time": end_time,
                        "course": course.name,
                        "lecturer": course.lecturer,
                        "group": course.group,
                        "room": room.name,
                        "duration": course.duration,
                        "session_id": course.session_id
                    })

            # Sort each day's schedule by start time
            for day in schedule_by_day:
                schedule_by_day[day].sort(key=lambda x: x["start_time"])

            self.visualize_graphs(schedule_by_day)

            # Schedule
            print("\nFinal Schedule:")
            print("=" * 90)
            for day, sessions in schedule_by_day.items():
                print(f"\n{day}:")
                print("-" * 90)
                if not sessions:
                    print("No classes scheduled")
                    continue
                for session in sessions:
                    print(
                        f"{session['start_time']}-{session['end_time']} | {session['course']:6} | "
                        f"{session['lecturer']:47} | {session['group']:2} | "
                        f"Room {session['room']:10} | {session['duration']}h | S{session['session_id']}"
                    )
        else:
            print("No solution found.")
```

Fig 4.10 Implementation of CSP Model

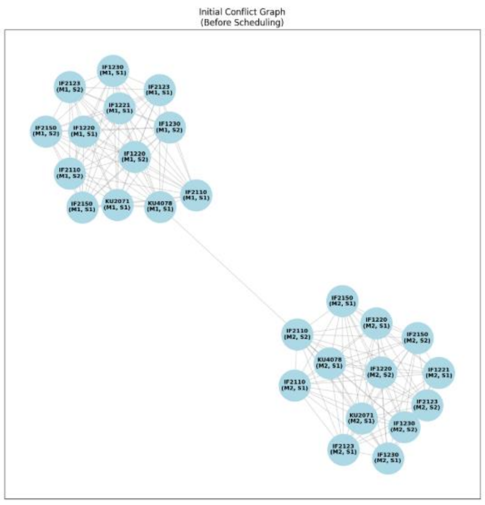


Fig 4.11 Schedule Graph before CSP

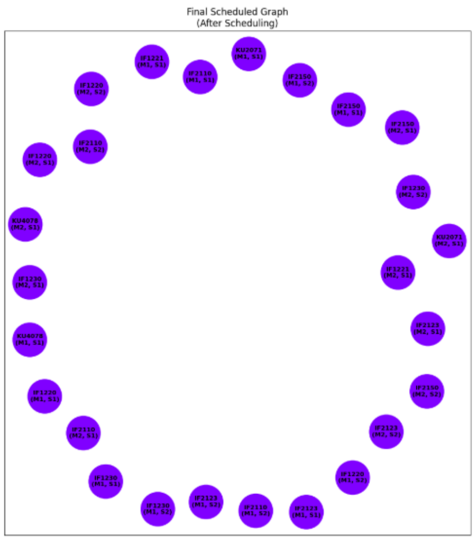
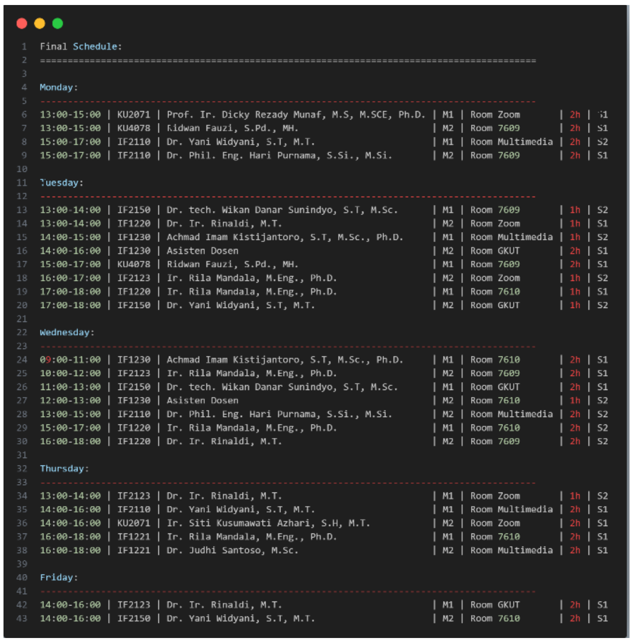


Fig 4.12 Colored Schedule Graph After CSP

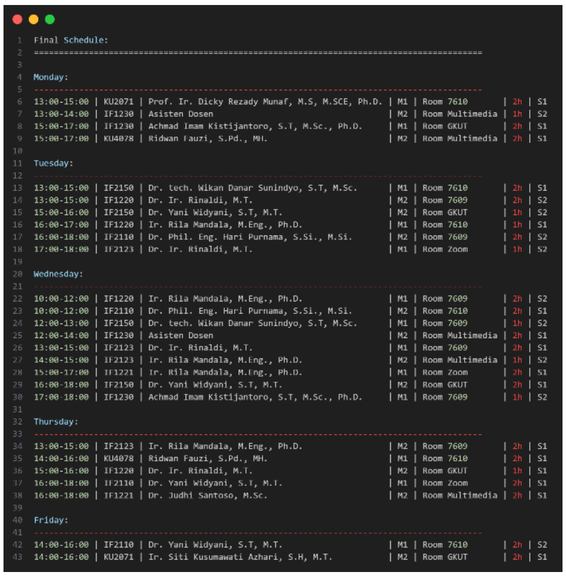


```
Final Schedule:
==========================================================================================

Monday:
------------------------------------------------------------------------------------------
13:00-15:00 | KU2071 | Prof. Ir. Dicky Rezady Munaf, M.S, M.SCE, Ph.D. | M1 | Room Zoom       | 2h | S1
13:00-15:00 | KU4078 | Ridwan Fauzi, S.Pd., MH.                        | M2 | Room 7609       | 2h | S1
15:00-17:00 | IF2110 | Dr. Yani Widyani, S.T, M.T.                     | M1 | Room Multimedia | 2h | S2
15:00-17:00 | IF2110 | Dr. Phil. Eng. Hari Purnama, S.Si., M.Si.       | M2 | Room 7609       | 2h | S1

Tuesday:
------------------------------------------------------------------------------------------
13:00-14:00 | IF2150 | Dr. tech. Wikan Danar Sunindyo, S.T, M.Sc.      | M1 | Room 7609       | 1h | S2
13:00-14:00 | IF1220 | Dr. Ir. Rinaldi, M.T.                           | M2 | Room Zoom       | 1h | S1
14:00-15:00 | IF1230 | Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D.     | M1 | Room Multimedia | 1h | S2
14:00-16:00 | IF1230 | Asisten Dosen                                   | M2 | Room GKUT       | 2h | S1
15:00-17:00 | KU4078 | Ridwan Fauzi, S.Pd., MH.                        | M1 | Room 7609       | 2h | S1
16:00-17:00 | IF2123 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M2 | Room Zoom       | 1h | S2
17:00-18:00 | IF1220 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M1 | Room 7610       | 1h | S1
17:00-18:00 | IF2150 | Dr. Yani Widyani, S.T, M.T.                     | M2 | Room GKUT       | 1h | S2

Wednesday:
------------------------------------------------------------------------------------------
09:00-11:00 | IF1230 | Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D.     | M1 | Room 7610       | 2h | S1
10:00-12:00 | IF2123 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M2 | Room 7609       | 2h | S1
11:00-13:00 | IF2150 | Dr. tech. Wikan Danar Sunindyo, S.T, M.Sc.      | M1 | Room GKUT       | 2h | S1
12:00-13:00 | IF1230 | Asisten Dosen                                   | M2 | Room 7610       | 1h | S2
13:00-15:00 | IF2110 | Dr. Phil. Eng. Hari Purnama, S.Si., M.Si.       | M2 | Room Multimedia | 2h | S2
15:00-17:00 | IF1220 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M1 | Room 7610       | 2h | S2
16:00-18:00 | IF1220 | Dr. Ir. Rinaldi, M.T.                           | M2 | Room 7609       | 2h | S2

Thursday:
------------------------------------------------------------------------------------------
13:00-14:00 | IF2123 | Dr. Ir. Rinaldi, M.T.                           | M1 | Room Zoom       | 1h | S2
14:00-16:00 | IF2110 | Dr. Yani Widyani, S.T, M.T.                     | M1 | Room Multimedia | 2h | S1
14:00-16:00 | KU2071 | Ir. Siti Kusumawati Azhari, S.H, M.T.           | M2 | Room Zoom       | 2h | S1
16:00-18:00 | IF1221 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M1 | Room 7610       | 2h | S1
16:00-18:00 | IF1221 | Dr. Judhi Santoso, M.Sc.                        | M2 | Room Multimedia | 2h | S1

Friday:
------------------------------------------------------------------------------------------
14:00-16:00 | IF2123 | Dr. Ir. Rinaldi, M.T.                           | M1 | Room GKUT       | 2h | S1
14:00-16:00 | IF2150 | Dr. Yani Widyani, S.T, M.T.                     | M2 | Room 7610       | 2h | S1
```

Fig 4.13 Result of Schedule

From Fig. 4.11, it is evident that prior to applying CSP, the schedules for the same group overlapped, as indicated by the adjacency of the graph. In Fig 4.12, after implementing CSP, the graph transforms into a Null Graph with no edges, and the colored graph uses only a single color. This indicates that there are no overlapping schedules, ensuring that all courses can be conducted without conflict.

Figure 4.13 illustrates the final schedule, where no overlapping courses are present, and all constraints are successfully satisfied. The results also showcase various scheduling possibilities for the 3rd-semester Informatics Engineering students at ITB for 2024, providing alternative options for future consideration. Since the program operates using a randomized approach for each execution, it generates a unique schedule every time, offering multiple scheduling alternatives.



```
Final Schedule:
==========================================================================================

Monday:
------------------------------------------------------------------------------------------
13:00-15:00 | KU2071 | Prof. Ir. Dicky Rezady Munaf, M.S, M.SCE, Ph.D. | M1 | Room 7610       | 2h | S1
13:00-14:00 | IF1230 | Asisten Dosen                                   | M2 | Room Multimedia | 1h | S2
15:00-17:00 | IF1230 | Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D.     | M1 | Room GKUT       | 2h | S1
15:00-17:00 | KU4078 | Ridwan Fauzi, S.Pd., MH.                        | M2 | Room Multimedia | 2h | S1

Tuesday:
------------------------------------------------------------------------------------------
13:00-15:00 | IF2150 | Dr. tech. Wikan Danar Sunindyo, S.T, M.Sc.      | M1 | Room 7610       | 2h | S1
13:00-15:00 | IF1220 | Dr. Ir. Rinaldi, M.T.                           | M2 | Room 7609       | 2h | S2
15:00-16:00 | IF2150 | Dr. Yani Widyani, S.T, M.T.                     | M2 | Room GKUT       | 1h | S2
16:00-17:00 | IF1220 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M1 | Room 7610       | 1h | S1
16:00-18:00 | IF2110 | Dr. Phil. Eng. Hari Purnama, S.Si., M.Si.       | M2 | Room 7609       | 2h | S2
17:00-18:00 | IF2123 | Dr. Ir. Rinaldi, M.T.                           | M1 | Room Zoom       | 1h | S2

Wednesday:
------------------------------------------------------------------------------------------
10:00-12:00 | IF1220 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M1 | Room 7609       | 2h | S2
10:00-12:00 | IF2110 | Dr. Phil. Eng. Hari Purnama, S.Si., M.Si.       | M2 | Room 7610       | 2h | S1
12:00-13:00 | IF2150 | Dr. tech. Wikan Danar Sunindyo, S.T, M.Sc.      | M1 | Room 7609       | 1h | S2
12:00-14:00 | IF1230 | Asisten Dosen                                   | M2 | Room Multimedia | 2h | S1
13:00-15:00 | IF2123 | Dr. Ir. Rinaldi, M.T.                           | M1 | Room 7609       | 2h | S1
14:00-15:00 | IF2123 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M2 | Room Multimedia | 1h | S2
15:00-17:00 | IF1221 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M1 | Room Zoom       | 2h | S1
16:00-18:00 | IF2150 | Dr. Yani Widyani, S.T, M.T.                     | M2 | Room GKUT       | 2h | S1
17:00-18:00 | IF1230 | Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D.     | M1 | Room 7609       | 1h | S2

Thursday:
------------------------------------------------------------------------------------------
13:00-15:00 | IF2123 | Ir. Rila Mandala, M.Eng., Ph.D.                 | M2 | Room 7609       | 2h | S1
14:00-16:00 | KU4078 | Ridwan Fauzi, S.Pd., MH.                        | M1 | Room 7610       | 2h | S1
15:00-16:00 | IF1220 | Dr. Ir. Rinaldi, M.T.                           | M2 | Room GKUT       | 1h | S1
16:00-18:00 | IF2110 | Dr. Yani Widyani, S.T, M.T.                     | M1 | Room Zoom       | 2h | S1
16:00-18:00 | IF1221 | Dr. Judhi Santoso, M.Sc.                        | M2 | Room Multimedia | 2h | S1

Friday:
------------------------------------------------------------------------------------------
14:00-16:00 | IF2110 | Dr. Yani Widyani, S.T, M.T.                     | M1 | Room 7610       | 2h | S2
14:00-16:00 | KU2071 | Ir. Siti Kusumawati Azhari, S.H, M.T.           | M2 | Room GKUT       | 2h | S1
```

Fig 4.14 Alternative Result of Schedule

V. Conclusion

The optimization of university timetabling using Constraint Satisfaction Problems (CSP) presents a structured and effective approach to addressing the complex scheduling needs of higher education institutions. By leveraging CSP, the process ensures that multiple constraints—such as room capacity, lecturer availability, course durations, and student group compatibility—are satisfied.

While advanced optimization techniques such as machine learning or metaheuristic algorithms could enhance scalability and adaptability, CSP remains a robust and reliable choice for solving structured scheduling problems. This approach has significant potential applications not only for academic institutions but also for other domains requiring efficient resource and time management, such as healthcare, event planning, and workforce scheduling.

VI. Appendix

The source code used to implement the optimization of university timetabling using CSP:

https://github.com/farrelathalla/Optimization-of-University-Timetabling-Using-CSP.git

VII. Acknowledgment

The author wishes to express gratitude, first and foremost, to Allah SWT for the guidance provided throughout the learning process and the writing of this paper. Appreciation is also extended to the lecturers of ITB Discrete Mathematics IF1220, Mr. Rinaldi Munir and Mr. Rila Mandala, for imparting their knowledge and guiding the students during the course. Additionally, the author is deeply thankful to family and friends for their unwavering support throughout the semester.

References

[1] B. Naderi, "Modeling and Scheduling University Course Timetabling Problems," *International Journal of Research in Industrial,* vol. 5, no. 1-4, pp. 1-15, 2016.

[2] T. E. Sakka, "University Course Timetable using Constraint Satisfaction and Optimization," *International Journal of Computing Academik Research (IJCAR),* vol. 4, no. 3, pp. 83-95, 2015.

[3] E. Ozcan, E. K. Burke and B. McCollum, "Proceedings of the 10th International Conference on the Practice and Theory of Automated Timetabling," in *patatconference.org*, York, United Kingdom, 2014.

[4] Munir, Rinaldi."http://informatika.stei.itb.ac.id/~rinaldi.munir/"

[5] https://medium.com/@abhijeetknayak/constraint-satisfaction-problems-map-coloring-38c60882be36

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 Desember 2024

[illegible]

Farrel Athalla Putra - 13523118