6IMODSIM PROJECT
Final Project Progress Work Week 1

Marc Lloyd G. Corporal
Kiana Beatriz C. De Jesus
Kirsten Dwayne A. Dizon

UNIVERSITY CLASS SCHEDULER

Model Introduction and Background

Every academic year during the enrollment period, class scheduling is no stranger to the universities. Class scheduling is the process of arranging and distributing classes along with their respective subjects and instructors throughout the rooms and time slots. It is done with the help of organizers, but when it is done manually, it can take too much of their time and labor. In addition, problems arise from it. Human mistakes can result in scheduling conflict; either a room and a time slot might be occupied already by another class, or an instructor might be overbooked for two classes at the same time (TutorShell Bogs, 2022). Therefore, to avoid these concerns, a digitalized class scheduling would be the key answer.

To elaborate, class scheduling in higher education is a detailed timetable that lists the academic courses a student must attend during a particular term or semester (uniRank, 2024). It assists students in handling their time and making sure they participate in the necessary sessions for their registered courses throughout the semester or term. Therefore, the university is responsible for having an intuitive and efficient class schedule maker while improving the overall academic experience for students, faculty, and administrative staff.

Class scheduling software should offer a clear overview of rooms, time slots, availability, and conflicts, helping to identify issues that need attention, such as instructors being double-booked or classes scheduled that overlap with each other (Accruent EMS, 2023). Creating class schedules is a complex and challenging task for educational institutions, requiring the coordination of multiple factors such as class timings, room assignments, instructor availability, and student enrollments. Thus, the significance of this model is implementing the needed sufficiency of a class scheduler for a university and it benefits people in several ways.

First, it benefits the organizers by automating the detection and resolution of overlapping courses, the program minimizes time and effort spent on manual scheduling and reduces the likelihood of errors.

Second, it benefits the students by helping them have equal places for learning upon enrollment and having them meet their varied course preferences.

Third, it benefits the instructors by assigning them to their sole respective class during a period.

Finally, it benefits the university by boosting its capacity as it makes the best use of resources.

The range of this model focuses on the development of a university class scheduler tailored specifically for Holy Angel University, particularly within the School of Computing Department, to streamline the unique scheduling challenges faced by the department every enrollment period.

The University Class Scheduler model is composed of beneficial functionalities. The program fully automates the scheduling process, reducing the potential for errors that occur with manual scheduling. Employing an algorithm-driven system eliminates typical mistakes such as overbooking, assigning incorrect rooms and time slots, or scheduling outside of available hours. This digital approach ultimately enhances the precision and dependability of the scheduling system. A key feature of the model is its capability to create the most efficient schedule by considering predefined constraints. It employs algorithms that assess these variables. Another feature is a room allocation system that automatically designates suitable classrooms for each scheduled class.

As mentioned, a major challenge in this process is avoiding scheduling inconsistencies, where students or instructors may be assigned to clashing time slots in a room. Thus, it is essential to design it with specific constraints to prevent conflicts and ensure efficient management of resources. Here are the following constraints:

1) An instructor cannot be in two (2) rooms simultaneously during the same time slot, ensuring their availability for each class they teach.
2) Each room can accommodate only one (1) class block per time slot (that further implies a block can only take one (1) subject per time slot), which prevents overlapping sessions and guarantees that all students have access to the necessary instructional space.
3) Each subject has a designated duration or own length in hours that dictates how long it occupies a room, which is crucial for effective scheduling; of course, that subject can only occupy any time slot if it doesn't conflict with another subject,
4) A subject must define what type of room it will use (if it is for lecture, for laboratory, or for both).

5) If a subject has both lecture and laboratory components, it should use a laboratory room in the first meeting in a week, and lecture room in the second week or vice versa (but the case of both the laboratory and lecture component will be on the same day should never happen).

6) A subject should be able to occupy decimal hours (e.g., one (1) hour and thirty (30) minutes, and more).

7) A subject can have multiple available instructors, and it randomly selects from the list of instructors.

8) The available scheduling window is restricted to between seven (7) AM and nine (9) PM, further structuring the timetable.

9) There are only available days in a week, which starts with Monday (M) and ends with Friday (F), and in a single week, a subject may only be plotted twice but not in the same day.

10) After each subject, there should be an interval of between five (5) to ten (10) minutes.

11) After every five (5) hours, there should be a break lasting a range of thirty (30) minutes to one (1) hour.

All in all, the overall purpose of the class scheduling model is to yield an intuitive and efficient class schedule while improving the general academic experience for students, faculty, and administrative staff.

Objectives of the Model

The overall objective of the University Class Scheduler Model is to have an optimized scheduling process in a university while serving as a detailed representation of what it does as a model. It results in producing the best schedule that benefits the students, instructors, and facilities, as it is specifically made for the blocks within college departments. Listed below are the primary objectives of the model:

1) To save time and labor in scheduling process through digitalization;
2) To provide an accurate timetable that is free from human errors; and
3) To practice resource optimization by only assigning a class into an available room and time slot.

Language and Algorithm

There are some algorithms that can be used to solve scheduling problems. These algorithms include graph coloring (Ganguli & Roy, 2017), constraint programming (Falaschi et al., 1997), and mixed integer linear programming (MILP) (Jankauskas et al., 2019). However, among these methods, the researchers found that the most suitable algorithm to use for the problem they are trying to solve is the genetic algorithm.

Genetic algorithms are heavily inspired by Darwinian evolution. It mimics the process of natural selection where the fittest organism, or in this case the solution, almost always comes out on top and survives (Yi & Zou, 2018). According to Wall (1996), genetic algorithms offer a robust and flexible solution to scheduling problems by operating on a population of solutions. If a better solution is required, then simply letting the process run longer can yield a more desirable outcome. As a result, this type of algorithm often outperforms other solutions (Jankauskas et al., 2019; Wall 1996), and is used predominantly in the same problem space (Hosny & Fatima, 2011). Having said that, it is not without its weaknesses. According to Badoni et al. (2014), genetic algorithms are prone to be stuck in a local optimum solution which hinders them to find a better solution. This can be improved by tweaking the underlying calculations in the algorithm (GeeksforGeeks, 2017).

On the other hand, the chosen programming language to implement this algorithm is Python. Python is one of the most popular programming languages out there. It ranked 3rd on the most popular programming languages category of the 2023 Stack Overflow Developer Survey, just below the core web technologies, namely: JavaScript and HTML/CSS (Stack Overflow, 2023). It is a high-level programming language that has a simple syntax and is extremely easy to use. As such, using this language to solve a hard problem would balance things out as opposed to using other languages like Java that are notorious for its boilerplate which may further increase the difficulty of the task.

Sample Model Code

```
# '''
# Class Scheduler
# -- The goal is to use a genetic algorithm to create a class schedule that has no conflicts

# CONSTRAINTS
# These are the constraints to prevent conflicts from schedules
#  - A professor cannot be in two rooms at once in a given time slot
#  - A room can only accomodate one block per time slot.
#  - A block can only take one subject per time slot
#  - Each subject has their own length in hours which is the time they would occupy a
room
#  - The available time is only 7am to 9pm.
#  - A subject can occupy any time slot as long as it doesn't conflict with another subject,
#    it fulfills the subject's length in hours, and is within 7am - 9pm

# Additional Constraints
#  - The available days in a week is Monday to Friday
#  - A subject may only be plotted twice in a single week but not in the same day
#  - After each subject, there should be an interval of between 5-10 minutes
#  - After every 5 hours, there should be a break lasting a range of 30 minutes to 1 hour.
#  - A room must be defined by its type (e.g, Lab, Lecture)
#  - A subject must define what type of room it will use (e.g, Lab, Lecture or both)
#  - If a subject has both Lecture and Lab components, it should use a Lab room in the
first
#    meeting in a week, and Lecture room in the second week or vice versa.
#  - A subject should be able to occupy decimal hours (e.g., 1 hour and 30 minutes, etc)
#  - A subject can have multiple available instructors and it randomly selects from the
list of instructors.
# '''
import math
import random
import time
from datetime import datetime, timedelta
from enum import Enum, IntEnum
from typing import List

from PIL import Image, ImageDraw, ImageFont
from tabulate import tabulate

#  TODO:
# [ ] - implement room types
# [ ] - implement additional time constraints
# [ ] - print the table wherein each class would have their own printed schedule
```

```
def generate_visual_schedule(schedule, block, filename="class_schedule.png"):
    # Set up the image
    width, height = 1200, 900
    image = Image.new("RGB", (width, height), color="white")
    draw = ImageDraw.Draw(image)

    # Try to load Arial font, fall back to default if not available
    try:
        font = ImageFont.truetype("arial.ttf", 12)
        title_font = ImageFont.truetype("arial.ttf", 24)
    except IOError:
        font = ImageFont.load_default()
        title_font = ImageFont.load_default()

    # Define colors
    colors = ["#FFA07A", "#98FB98", "#87CEFA", "#DDA0DD", "#F0E68C"]

    # Draw title
    title = f"Class Schedule for {block.get_block_dept().get_dept_prefix()}-
{block.get_block_number()}"
    draw.text((20, 20), title, font=title_font, fill="black")

    # Define grid
    days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
    times = [f"{h:02d}:00" for h in range(7, 22)]  # 7 AM to 9 PM
    cell_width = (width - 100) // len(days)
    cell_height = (height - 100) // (
        len(times) * 2
    )  # Divide each hour into two 30-minute slots

    # Draw grid
    for i, day in enumerate(days):
        draw.text((100 + i * cell_width + 5, 60), day, font=font, fill="black")
        for j, t in enumerate(times):
            draw.text((20, 100 + j * cell_height * 2), t, font=font, fill="black")
            draw.line(
                [(100, 80 + j * cell_height * 2), (width, 80 + j * cell_height * 2)],
                fill="black",
            )

    for i in range(len(days) + 1):
        draw.line(
            [(100 + i * cell_width, 80), (100 + i * cell_width, height)], fill="black"
        )
    # Plot classes
    for day in schedule:
        day_index = days.index(day.name.capitalize())
        for start_time, end_time, subject, room in schedule[day]:
            start_minutes = start_time.hour * 60 + start_time.minute - 7 * 60
            end_minutes = end_time.hour * 60 + end_time.minute - 7 * 60
            if end_minutes <= start_minutes:  # Handle classes ending after midnight
                end_minutes = 14 * 60  # Set to 9 PM

            start_y = 80 + (start_minutes * cell_height) // 30
            end_y = 80 + (end_minutes * cell_height) // 30

            color = random.choice(colors)
            draw.rectangle(
                [
                    100 + day_index * cell_width,
                    start_y,
                    100 + (day_index + 1) * cell_width,
                    end_y,
                ],
                fill=color,
                outline="black",
            )

            text = 
f"{subject.get_subject_name()}\n{room.get_room_num()}\n{start_time.strftime('%I:%
M %p')}-{end_time.strftime('%I:%M %p')}"
            draw.text(
                (105 + day_index * cell_width, start_y + 5),
                text,
                font=font,
                fill="black",
            )

    # Save the image
    image.save(filename)
    print(f"Schedule image saved as {filename}")

class RoomType(Enum):
    LECTURE = 1
    LAB = 2

class Day(IntEnum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5

class Instructor:
    def __init__(self, name: str) -> None:
        self._name = name

    def get_name(self) -> str:
        return self._name

    def __str__(self) -> str:
        return self._name

class Room:
    def __init__(self, room_num: str, max_capacity: int, room_type: RoomType) -> None:
        self._room_num = room_num
        self._max_capacity = max_capacity
        self._room_type = room_type

    def get_room_num(self) -> str:
        return self._room_num

    def get_max_capacity(self) -> int:
        return self._max_capacity

    def get_room_type(self) -> RoomType:
        return self._room_type

# TODO: This should support minutes as well instead of full hours
class Subject:
    def __init__(
        self, name: str, instructors: list[Instructor], duration: timedelta
    ) -> None:
        self._name = name
        self._instructors = instructors
        self.duration = duration

    def get_subject_name(self) -> str:
        return self._name

    def get_subject_instructors(self) -> list[Instructor]:
        return self._instructors
    def get_subject_duration(self) -> timedelta:
        return self.duration

    def __str__(self) -> str:
        return self._name

class Department:
    def __init__(self, prefix: str, subjects: list[Subject]) -> None:
        self._prefix = prefix  # CS, WD, NA, EMC, CYB
        self._subjects = subjects

    def get_dept_prefix(self) -> str:
        return self._prefix

    def get_dept_subjects(self) -> list[Subject]:
        return self._subjects

class Block:
    def __init__(
        self, number: str, dept: Department, subjects: list[Subject], num_students: int
    ) -> None:
        self._number = number  # 303, 102, 202
        self._dept = dept  # dept.prefix i.e., CS,
        self._subjects = subjects
        self._num_students = num_students
        self._room = None
        self._instructor = None

    def get_block_number(self) -> str:
        return self._number

    def get_block_dept(self) -> Department:
        return self._dept

    def get_block_subjects(self) -> list[Subject]:
        return self._subjects

    def get_room(self):
        return self._room

    def set_room(self, room):
        self._room = room

    def set_instructor(self, instructor):
        self._instructor = instructor
    def __str__(self) -> str:
        return f"${self._dept.get_dept_prefix()}-${self._number}, ${self._subjects}, 
${self._room.get_room_num()}, ${self._instructor.get_name()}"  # type: ignore

class TimeSlot:
    def __init__(self, day: Day, start_time: datetime, end_time: datetime):
        self._day = day
        self.start_time = start_time
        self.end_time = end_time

    def get_day(self) -> Day:
        return self._day

    def __str__(self):
        return f"{self.start_time.strftime('%I:%M %p')} - {self.end_time.strftime('%I:%M 
%p')}"

class Schedule:
    def __init__(self, blocks: List[Block], rooms: List[Room]):
        self.blocks = blocks
        self.rooms = rooms
        self.assignments = []  # List of (Block, Subject, Room, TimeSlot) tuples
        self.generate_random_schedule()

    def generate_random_schedule(self):
        self.assignments = []
        for block in self.blocks:
            for subject in block.get_block_subjects():
                # Schedule the subject twice
                scheduled_days = set()
                for _ in range(2):
                    room = random.choice(self.rooms)

                    # Generate a random start time between 7:00 AM and 9:00 PM, aligned to 5-
minute intervals
                    start_time = datetime.combine(
                        datetime.today(), datetime.min.time()
                    ) + timedelta(
                        hours=7,
                        minutes=random.randint(0, 14 * 12)
                        * 5,  # 14 hours * 12 5-minute intervals
                    )

                    end_time = start_time + subject.get_subject_duration()
                    # If end time is after 9:00 PM, adjust start time
                    if end_time.hour >= 21:
                        start_time = (
                            datetime.combine(datetime.today(), datetime.min.time())
                            + timedelta(hours=21)
                            - subject.get_subject_duration()
                        )
                        end_time = datetime.combine(
                            datetime.today(), datetime.min.time()
                        ) + timedelta(hours=21)

                    # Choose a day that hasn't been scheduled yet
                    available_days = set(Day) - scheduled_days
                    if available_days:
                        day = random.choice(list(available_days))
                        scheduled_days.add(day)
                    else:
                        day = random.choice(list(Day))  # Fallback, should rarely happen

                    time_slot = TimeSlot(day, start_time, end_time)
                    self.assignments.append((block, subject, room, time_slot))

    def calculate_fitness(self) -> float:
        conflicts = {
            "time": 0,
            "room": 0,
            "instructor": 0,
            "block": 0,
            "interval": 0,
            "subject_occurrence": 0,
        }
        total_assignments = len(self.assignments)

        def time_overlap(time1: TimeSlot, time2: TimeSlot) -> bool:
            return (
                time1.start_time < time2.end_time and time2.start_time < time1.end_time
            )

        def time_diff_minutes(time1: datetime, time2: datetime) -> int:
            return abs(int((time1 - time2).total_seconds() / 60))

        subject_occurrences = {}

        for i, (block1, subject1, room1, time1) in enumerate(self.assignments):
```

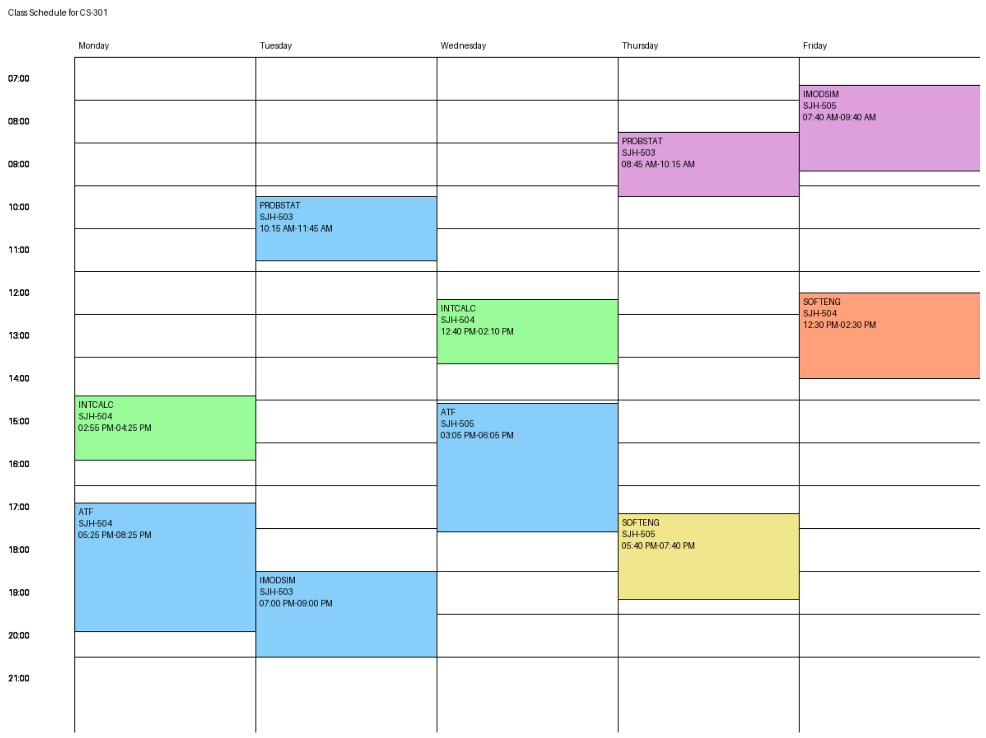
```
day1 = time1.get_day()

# Check if subject is within 7am to 9pm
if time1.start_time.hour < 7 or time1.end_time.hour > 21:
    conflicts["time"] += 1

# Track subject occurrences
subject_key = (block1, subject1)
if subject_key not in subject_occurrences:
    subject_occurrences[subject_key] = []
subject_occurrences[subject_key].append(day1)

# Check for conflicts with other assignments
for j, (block2, subject2, room2, time2) in enumerate(
    self.assignments[i + 1 :], i + 1
):
    day2 = time2.get_day()

    if day1 == day2 and time_overlap(time1, time2):
        # Room conflict
        if room1 == room2:
            conflicts["room"] += 1

        # Instructor conflict
        if any(
            instr in subject2.get_subject_instructors()
            for instr in subject1.get_subject_instructors()
        ):
            conflicts["instructor"] += 1

        # Block conflict
        if block1 == block2:
            conflicts["block"] += 1

# Check for proper intervals between subjects
block_assignments = [
    a for a in self.assignments if a[0] == block1 and a[3].get_day() == day1
]
sorted_assignments = sorted(
    block_assignments, key=lambda a: a[3].start_time
)

for k in range(len(sorted_assignments) - 1):
    _, _, _, current_slot = sorted_assignments[k]
    _, _, _, next_slot = sorted_assignments[k + 1]
```

```
        interval = time_diff_minutes(
            next_slot.start_time, current_slot.end_time
        )

        # Penalize if interval is less than 5 minutes or more than 10 minutes
        if interval < 5 or interval > 10:
            conflicts["interval"] += 1

# Check for breaks after every 5 hours
for block in self.blocks:
    for day in Day:
        day_assignments = sorted(
            [
                a
                for a in self.assignments
                if a[0] == block and a[3].get_day() == day
            ],
            key=lambda a: a[3].start_time,
        )

        if day_assignments:
            cumulative_time = timedelta()
            last_break_end = day_assignments[0][3].start_time

            for _, _, _, time_slot in day_assignments:
                cumulative_time += time_slot.end_time - time_slot.start_time

                if cumulative_time >= timedelta(hours=5):
                    break_duration = time_diff_minutes(
                        time_slot.end_time, last_break_end
                    )
                    if break_duration < 30 or break_duration > 60:
                        conflicts["time"] += 1
                    cumulative_time = timedelta()
                    last_break_end = time_slot.end_time

# Check subject occurrences
for occurrences in subject_occurrences.values():
    if len(occurrences) != 2:
        conflicts["subject_occurrence"] += 1
    elif len(set(occurrences)) != 2:
        conflicts["subject_occurrence"] += 1

# Calculate total conflicts with weights
```

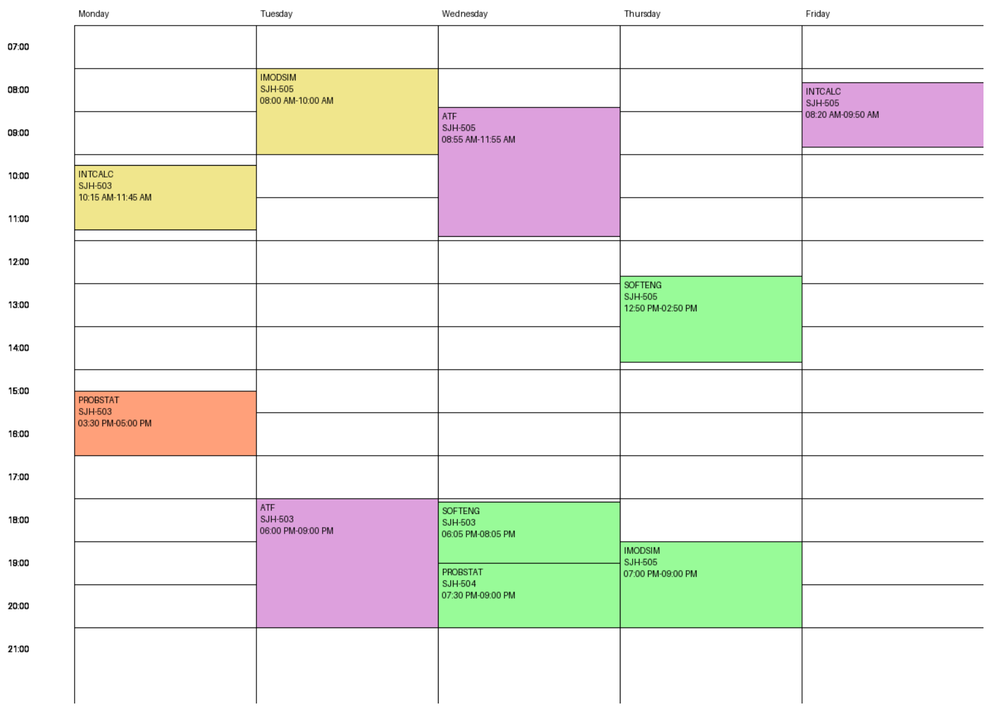
```
    total_conflicts = (
        conflicts["time"] * 20
        + conflicts["room"] * 3
        + conflicts["instructor"] * 3
        + conflicts["block"] * 3
        + conflicts["interval"] * 1
        + conflicts["subject_occurrence"] * 2
    )

    # Normalize conflicts (0 to 1, where 0 is best)
    max_possible_conflicts = (
        total_assignments * 32
    )  # Assuming worst case: all constraints violated for all assignments
    normalized_conflicts = total_conflicts / max_possible_conflicts

    # Calculate fitness (0 to 1, where 1 is best)
    fitness = 1 - normalized_conflicts

    return fitness

def print_block_schedule(self, block: Block):
    schedule = {day: [] for day in Day}

    for b, subject, room, time_slot in self.assignments:
        if b == block:
            day = time_slot.get_day()
            start_time = time_slot.start_time
            end_time = time_slot.end_time
            schedule[day].append((start_time, end_time, subject, room))

    # Sort schedules for each day
    for day in Day:
        schedule[day].sort(key=lambda x: x[0])

    # Generate all possible time slots
    all_time_slots = []
    current_time = datetime.combine(
        datetime.today(), datetime.min.time()
    ) + timedelta(hours=7)
    end_of_day = datetime.combine(
        datetime.today(), datetime.min.time()
    ) + timedelta(hours=21)

    while current_time < end_of_day:
        all_time_slots.append(current_time)
```

```
            current_time += timedelta(minutes=5)

        table_data = []
        for time_slot in all_time_slots:
            row = [time_slot.strftime("%I:%M %p")]
            for day in Day:
                cell_content = ""
                for start_time, end_time, subject, room in schedule[day]:
                    if start_time <= time_slot < end_time:
                        cell_content = (
                            f"{subject.get_subject_name()}\n{room.get_room_num()}"
                        )
                        break
                row.append(cell_content)
            table_data.append(row)

        # Remove consecutive duplicate rows
        compressed_table_data = []
        for row in table_data:
            if not compressed_table_data or row != compressed_table_data[-1]:
                compressed_table_data.append(row)

        # Print the table
        headers = ["Time"] + [day.name.capitalize() for day in Day]
        print(
            f"\nSchedule for {block.get_block_dept().get_dept_prefix()}-
{block.get_block_number()}:"
        )
        print(tabulate(compressed_table_data, headers=headers, tablefmt="grid"))

        # Print intervals between classes
        for day in Day:
            if schedule[day]:
                print(f"\nIntervals for {day.name}:")
                for i in range(len(schedule[day]) - 1):
                    current_end = schedule[day][i][1]
                    next_start = schedule[day][i + 1][0]
                    interval = (next_start - current_end).total_seconds() / 60
                    print(
                        f"  {current_end.strftime('%I:%M %p')} - {next_start.strftime('%I:%M
%p')}: {interval} minutes"
                    )

    def generate_visual_schedule(self, block, filename="class_schedule.png"):
        schedule = {day: [] for day in Day}
        for b, subject, room, time_slot in self.assignments:
            if b == block:
                day = time_slot.get_day()
                start_time = time_slot.start_time
                end_time = time_slot.end_time
                schedule[day].append((start_time, end_time, subject, room))

        # Sort schedules for each day
        for day in Day:
            schedule[day].sort(key=lambda x: x[0])

        generate_visual_schedule(schedule, block, filename)

class GeneticAlgorithm:
    def __init__(
        self,
        population_size: int,
        mutation_rate: float,
        blocks: List[Block],
        rooms: List[Room],
        fitness_limit=1.00,
        elitism_rate=0.1,
    ):
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.blocks = blocks
        self.rooms = rooms
        self.population = [Schedule(blocks, rooms) for _ in range(population_size)]
        self.fitness_limit = fitness_limit
        self.elitism_rate = elitism_rate

    def _select_parent(self) -> Schedule:
        # Arbitrary precision level
        precision_level = 0.5

        # Yamane's Formula
        sample_size = self.population_size / (
            1 + (self.population_size * (precision_level**2))
        )

        # select a sample from the given population and calculated sample size
        tournament = random.sample(self.population, math.floor(sample_size))

        # select the fittest (highest/max) schedule to be the parent based on the
        # tournament list and their fitness score
        return max(tournament, key=lambda schedule: schedule.calculate_fitness())

    def _crossover(self, parent1: Schedule, parent2: Schedule) -> Schedule:
        child = Schedule(self.blocks, self.rooms)

        midpoint = len(parent1.assignments) // 2

        child.assignments = (
            parent1.assignments[:midpoint] + parent2.assignments[midpoint:]
        )

        return child

    def _mutate(self, schedule: Schedule):
        for i in range(len(schedule.assignments)):
            if random.random() < self.mutation_rate:
                block, subject, _, _ = schedule.assignments[i]
                room = random.choice(self.rooms)

                # Generate a random start time between 7:00 AM and 9:00 PM, aligned to 5-
minute intervals
                start_time = datetime.combine(
                    datetime.today(), datetime.min.time()
                ) + timedelta(
                    hours=7,
                    minutes=random.randint(0, 14 * 12)
                    * 5,  # 14 hours * 12 5-minute intervals
                )

                end_time = start_time + subject.get_subject_duration()

                # If end time is after 9:00 PM, adjust start time
                if end_time.hour >= 21:
                    start_time = (
                        datetime.combine(datetime.today(), datetime.min.time())
                        + timedelta(hours=21)
                        - subject.get_subject_duration()
                    )
                    end_time = datetime.combine(
                        datetime.today(), datetime.min.time()
                    ) + timedelta(hours=21)

                # Ensure the new day is different from the other occurrence of this subject
                current_days = [
                    assign[3].get_day()
                    for assign in schedule.assignments
                    if assign[0] == block and assign[1] == subject
                ]
                available_days = list(set(Day) - set(current_days))
                if available_days:
                    day = random.choice(available_days)
                else:
                    day = random.choice(list(Day))

                time_slot = TimeSlot(day, start_time, end_time)
                schedule.assignments[i] = (block, subject, room, time_slot)

    def evolve(self, generations: int):
        evolution_history = []
        start_time = time.time()
        for gen in range(generations):
            self.population.sort(key=lambda x: x.calculate_fitness(), reverse=True)

            # Elitism
            elite_size = int(self.population_size * self.elitism_rate)
            new_population = self.population[:elite_size]

            while len(new_population) < self.population_size:
                parent1 = self._select_parent()
                parent2 = self._select_parent()
                child = self._crossover(parent1, parent2)
                self._mutate(child)
                new_population.append(child)

            self.population = new_population

            best_schedule = self.get_best_schedule()
            evolution_history.append((gen, best_schedule))

            if math.isclose(
                best_schedule.calculate_fitness(), self.fitness_limit, abs_tol=0.001
            ):
                break

        elapsed_time = time.time() - start_time
        best_generation = evolution_history[-1][0]
        return evolution_history, elapsed_time, best_generation

    def get_best_schedule(self) -> Schedule:
            return max(self.population, key=lambda schedule: schedule.calculate_fitness())

    if __name__ == "__main__":
        # Create sample data (instructors, subjects, departments, blocks, rooms)
        sir_uly = Instructor("Sir Ulysses Monsale")
        maam_lou = Instructor("Ma'am Louella Salenga")
        sir_glenn = Instructor("Sir Glenn Mañalac")
        sir_lloyd = Instructor("Sir Lloyd Estrada")
        maam_raquel = Instructor("Ma'am Raquel Rivera")

        IMODSIM = Subject("IMODSIM", [maam_lou], timedelta(hours=2))
        PROBSTAT = Subject("PROBSTAT", [sir_lloyd], timedelta(hours=1, minutes=30))
        INTCALC = Subject("INTCALC", [sir_glenn], timedelta(hours=1, minutes=30))
        ATF = Subject("ATF", [sir_uly], timedelta(hours=3))
        SOFTENG = Subject("SOFTENG", [maam_raquel], timedelta(hours=2))

        ROOMS = [
            ["SJH-503", 45, RoomType.LECTURE],
            ["SJH-504", 45, RoomType.LAB],
            ["SJH-505", 45, RoomType.LECTURE],
        ]

        dept = Department("CS", [INTCALC, PROBSTAT, IMODSIM, ATF, SOFTENG])
        dept_subjects = dept.get_dept_subjects()

        block1 = Block("301", dept, dept_subjects, 45)
        block2 = Block("302", dept, dept_subjects, 45)
        block3 = Block("303", dept, dept_subjects, 45)

        rooms = [
            Room(room_num, capacity, room_type) for room_num, capacity, room_type in
    ROOMS
        ]

        ga = GeneticAlgorithm(
            population_size=500,
            mutation_rate=0.3,
            blocks=[block1, block2, block3],
            rooms=rooms,
            elitism_rate=0.1,
        )

        print("Initial population created.")
        print(f"Number of schedules in population: {len(ga.population)}")
        print(
        f"Number of assignments in first schedule: {len(ga.population[0].assignments)}"
    )

    evolution_history, elapsed_time, best_generation = ga.evolve(generations=1000)

    print("\nEvolution completed.")
    best_schedule = evolution_history[-1][1]  # Get the best schedule
    for idx, block in enumerate([block1, block2, block3]):
        best_schedule.print_block_schedule(block)
        best_schedule.generate_visual_schedule(block, f"block{idx+1}_schedule.png")

    print(f"\nFitness: {best_schedule.calculate_fitness():.4f}")
    print(f"Best solution found in generation: {best_generation}")
    print(f"Time taken to find the best solution: {elapsed_time:.2f} seconds")
```

Sample Output

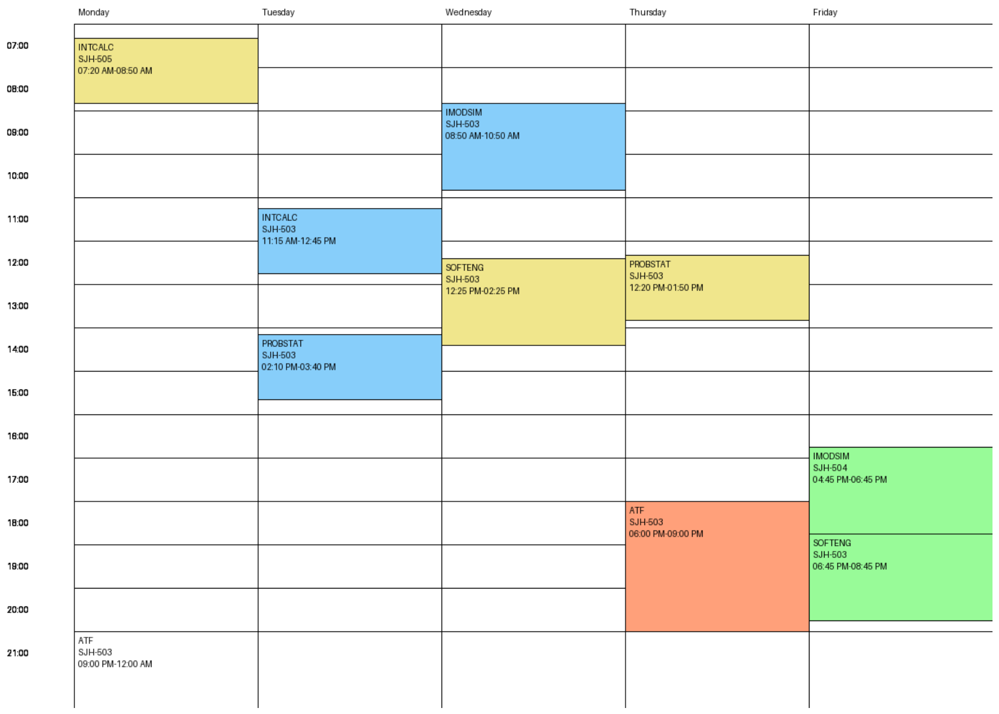


Class Schedule for CS-302



	Monday	Tuesday	Wednesday	Thursday	Friday
07:00					
08:00		IMODSIM SJH-505 08:00 AM-10:00 AM	ATF SJH-505 08:55 AM-11:55 AM		INTCALC SJH-505 08:20 AM-09:50 AM
09:00					
10:00	INTCALC SJH-503 10:15 AM-11:45 AM				
11:00					
12:00				SOFTENG SJH-505 12:50 PM-02:50 PM	
13:00					
14:00					
15:00	PROBSTAT SJH-503 03:30 PM-05:00 PM				
16:00					
17:00					
18:00		ATF SJH-503 06:00 PM-09:00 PM	SOFTENG SJH-503 06:05 PM-08:05 PM		
19:00			PROBSTAT SJH-504 07:30 PM-09:00 PM	IMODSIM SJH-505 07:00 PM-09:00 PM	
20:00					
21:00					

Class Schedule for CS-303



	Monday	Tuesday	Wednesday	Thursday	Friday
07:00	INTCALC SJH-505 07:20 AM-08:50 AM				
08:00			IMODSIM SJH-503 08:50 AM-10:50 AM		
09:00					
10:00					
11:00		INTCALC SJH-503 11:15 AM-12:45 PM			
12:00			SOFTENG SJH-503 12:25 PM-02:25 PM	PROBSTAT SJH-503 12:20 PM-01:50 PM	
13:00					
14:00		PROBSTAT SJH-503 02:10 PM-03:40 PM			
15:00					
16:00					IMODSIM SJH-504 04:45 PM-06:45 PM
17:00					
18:00				ATF SJH-503 06:00 PM-09:00 PM	SOFTENG SJH-503 06:45 PM-08:45 PM
19:00					
20:00					
21:00	ATF SJH-503 09:00 PM-12:00 AM				

References

Abdelhalim, E. A., & El Khayat, G. A. (2016). A utilization-based genetic algorithm for solving the university timetabling problem (UGA). *Alexandria Engineering Journal*, *55*(2), 1395–1409. https://doi.org/10.1016/j.aej.2016.02.017

Accruent EMS. (2023). *Choosing the Best Class Scheduling Software: Benefits, Features & Why Your Campus Needs It Today*. Emssoftware.com. https://emssoftware.com/resources/blog-posts/choosing-best-class-scheduling-software-benefits-features-why-your-campus#:~:text=What%20are%20the%20Benefits%20of%20Using%20Class%20Scheduling.

Badoni, R. P., Gupta, D. K., & Mishra, P. (2014). A new hybrid algorithm for university course timetabling problem using events based on groupings of students. *Computers & Industrial Engineering*, *78*, 12–25. https://doi.org/10.1016/j.cie.2014.09.020

Falaschi, M., Gabbrielli, M., Marriott, K., & Palamidessi, C. (1997). Constraint logic programming with dynamic scheduling: A semantics based on closure operators. *Information and Computation*, *137*(1), 41–67. https://doi.org/10.1006/inco.1997.2638

Ganguli, R., & Roy, S. (2017). A study on course timetable scheduling using graph coloring approach. *International Journal of Computational and Applied*

Mathematics, *12*(2), 469–485. https://www.ripublication.com/ijcam17/ijcamv12n2_26.pdf

GeeksforGeeks. (2017, June 29). *Genetic algorithms*. GeeksforGeeks. https://www.geeksforgeeks.org/genetic-algorithms/

Herath, A. (2017). *eGrove eGrove Electronic Theses and Dissertations Graduate School 2017 Genetic Algorithm For University Course Timetabling Problem Genetic Algorithm For University Course Timetabling Problem*. https://egrove.olemiss.edu/cgi/viewcontent.cgi?article=1442&context=etd

Hosny, M., & Fatima, S. (2011). A survey of genetic algorithms for the university timetabling problem. *International Proceedings of Computer Science and Information Technology*, *13*, 34–39. https://www.academia.edu/download/69269150/A_Survey_of_Genetic_Algorithms_for_the_U20210909-23708-15pjjwj.pdf

Jankauskas, K., Papageorgiou, L. G., & Farid, S. S. (2019). Fast genetic algorithm approaches to solving discrete-time mixed integer linear programming problems of capacity planning and scheduling of biopharmaceutical manufacture. *Computers & Chemical Engineering*, *121*, 212–223. https://doi.org/10.1016/j.compchemeng.2018.09.019

Stack Overflow. (2023). *Stack overflow developer survey 2023*. Stack Overflow. https://survey.stackoverflow.co/2023/#technology-most-popular-technologies

TutorShell Blogs. (2022, November 21). *5 Reasons Why Teachers Need Class Scheduling Software - TutorShell Blogs*. TutorShell Blogs. https://blog.tutorshell.com/why-teacher-need-class-scheduling-software/

uniRank. (2024). *Class Schedule*. 4icu.org. https://www.4icu.org/glossary/class-schedule/

Wall, M. (1996). *A genetic algorithm for resource-constrained scheduling*. http://lancet.mit.edu/mwall/phd/thesis/thesis.pdf

Yi, S., & Zou, S. (2018). Genetic algorithm theory and its application. *3rd International Conference on Automation, Mechanical Control and Computational Engineering, 166*. https://www.researchgate.net/publication/325564108_Genetic_Algorithm_Theory_and_Its_Application/fulltext/5b163063a6fdcc31bbf53bc5/Genetic-Algorithm-Theory-and-Its-Application.pdf