TIMETABLE SCHEDULER

TEAM MEMBERS

01

NIKUNJ GUPTA

02

PRADYUMNA SARASWAT

03

PRASHUK JAIN

04

YASH BAJAJ

SUBMITTED TO- VINAY SIR

INDEX

PROBLEM STATEMENT

Design a timetable scheduling system that efficiently assigns time slots to a set of activities or events, ensuring that all constraints and preferences are met. The system should take into consideration factors such as room availability, resource constraints, time preferences, and any other relevant requirements.

Activities and Constraints
Resource Optimization
Time Preferences
Conflict Resolution

ALGORITHM(WITH ANALYSIS)

Timetable Scheduler Algorithm:

1. Input:
   - List of activities with their constraints and preferences.
   - Available time slots.
   - Resource availability (rooms, equipment, etc.).
2. Initialization:
   - Create an empty timetable structure to store the assigned time slots for each activity.
   - Initialize a list of available time slots with all possible time slots.
3. Activity Scheduling:
   - Iterate through each activity:
     - For each activity, consider available time slots based on constraints and preferences.
     - Apply heuristics or algorithms to prioritize time slots that satisfy constraints and preferences.
     - Assign the activity to the selected time slot.
     - Update the list of available time slots by removing the assigned time slot.
     - Conflict Resolution:
     - Detect and resolve conflicts that may arise due to overlapping schedules or resource unavailability.
     - Adjust the timetable by reassigning conflicting activities to different time slots or resolving resource conflicts.
   -
   -

Output:

- Provide the final timetable with assigned time slots for each activity.
- Display any warnings or conflicts that were resolved during the scheduling process.

Algorithm Analysis:

- Time Complexity:
- The time complexity of the algorithm depends on the number of activities and available time slots.
- If 'n' is the number of activities and 'm' is the number of time slots, the time complexity could be O(n * m) in the worst case.
- Conflict resolution and optimization steps may contribute to additional time complexity, depending on the algorithms used.
- Space Complexity:
- The space complexity is influenced by the data structures used to store the timetable, available time slots, and other information.
- If 'n' is the number of activities and 'm' is the number of time slots, the space complexity could be O(n + m).

Optimization Techniques:

- The efficiency of the scheduler can be improved by incorporating optimization techniques such as:
  - Heuristics for intelligent time slot selection.
  - Genetic algorithms, simulated annealing, or other metaheuristic approaches for global optimization.
  - Dynamic adjustments based on real-time changes or feedback.

Scalability:

- The algorithm should be designed to scale with an increasing number of activities and time slots.
- Consider parallel processing or distributed computing for large-scale scheduling problems.

This algorithm provides a basic framework for a timetable scheduler. Depending on the specific requirements and constraints of the problem, additional features and optimizations may be necessary. The efficiency of the scheduler can be further enhanced by incorporating domain-specific knowledge and leveraging advanced optimization techniques.

PROGRAM WITH OUTPUT

```
import random
import copy

class TimetableSchedulerGA:
    def _init_(self, days, time_slots, population_size):
        self.days = days
        self.time_slots = time_slots
        self.population_size = population_size
        self.population = []

        # Initialize a random population
        for _ in range(population_size):
            individual = {day: {time: None for time in time_slots} for day in days}
            self.randomly_schedule_events(individual)
            self.population.append(individual)

    def randomly_schedule_events(self, individual):
        # Randomly assign events to time slots in the individual
        events = ["Event1", "Event2", "Event3", "Event4", "Event5"]  # Placeholder event names
        for day in self.days:
            for time in self.time_slots:
                if random.random() < 0.5:  # 50% probability to schedule an event
                    event = random.choice(events)
                    individual[day][time] = event
```

```
def fitness(self, individual):
        # Placeholder fitness function (you can replace it with a more sophisticated one)
        # For simplicity, this function counts the number of unique events scheduled
        scheduled_events = set()
        for day in self.days:
            for time in self.time_slots:
                event = individual[day][time]
                if event:
                    scheduled_events.add(event)
        return len(scheduled_events)

    def crossover(self, parent1, parent2):
        # One-point crossover
        crossover_point = random.choice(self.days)
        child = copy.deepcopy(parent1)
        for day in self.days:
            if day > crossover_point:
                child[day] = copy.deepcopy(parent2[day])
        return child

    def mutate(self, individual, mutation_rate):
        # Randomly mutate events in the individual with a certain probability
        for day in self.days:
            for time in self.time_slots:
                if random.random() < mutation_rate:
                    individual[day][time] = None
```

```
def evolve(self, generations, mutation_rate):
        for generation in range(generations):
            # Evaluate fitness of each individual in the population
            fitness_scores = [self.fitness(individual) for individual in self.population]

            # Select parents for crossover based on fitness
            parents = random.choices(self.population, weights=fitness_scores, k=self.population_size)

            # Create new generation through crossover and mutation
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1, parent2 = parents[i], parents[i + 1]
                child1 = self.crossover(parent1, parent2)
                child2 = self.crossover(parent2, parent1)
                self.mutate(child1, mutation_rate)
                self.mutate(child2, mutation_rate)
                new_population.extend([child1, child2])

            self.population = new_population

            # Display the best individual in the current generation
            best_individual = max(self.population, key=self.fitness)
            print(f"Generation {generation + 1}, Best Fitness: {self.fitness(best_individual)}")
            self.display_schedule(best_individual)
```

```
def display_schedule(self, individual):
        print("\nTimetable Schedule:")
        for day in self.days:
            print(f"\n{day}:")
            for time in self.time_slots:
                event = individual[day][time]
                print(f"  {time}: {event if event else 'Free'}")
def main():
    # Define the days and time slots for the timetable
    days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
    time_slots = ["9:00 AM", "10:00 AM", "11:00 AM", "12:00 PM", "1:00 PM", "2:00 PM", "3:00 PM", "4:00 PM"]

    # Set the population size and number of generations
    population_size = 10
    generations = 50
    # Set the mutation rate
    mutation_rate = 0.1
 # Create a timetable scheduler using a genetic algorithm
    scheduler = TimetableSchedulerGA(days, time_slots, population_size)
# Evolve the population through generations
    scheduler.evolve(generations, mutation_rate)
# Display the final schedule
    best_individual = max(scheduler.population, key=scheduler.fitness)
    print("\nFinal Timetable Schedule:")
    scheduler.display_schedule(best_individual)
if _name_ == "_main_":
```

OUTPUT

main.py

Save Run

```python
from datetime import datetime, timedelta

class TimetableScheduler:
    def __init__(self):
        self.schedule = []

    def add_event(self, event_name, start_time, duration):
        """
        Add an event to the schedule.

        Parameters:
        - event_name: Name of the event.
        - start_time: Start time of the event (24-hour format, e.g., "14:30").
        - duration: Duration of the event in minutes.
        """
        start_datetime = datetime.strptime(start_time, "%H:%M")
        end_datetime = start_datetime + timedelta(minutes=duration)
        self.schedule.append({
            'event_name': event_name,
            'start_time': start_datetime,
            'end_time': end_datetime
        })

    def display_schedule(self):
        """
        Display the scheduled events.
        """
        if not self.schedule:
            print("No events scheduled.")
            return

        for event in self.schedule:
            print(f"{event['event_name']} - {event['start_time'].strftime('%H:%M')} to {event['end_time'].strftime('%H:%M'
                )}")

# Example usage
scheduler = TimetableScheduler()

# Adding events to the schedule
scheduler.add_event("Meeting 1", "09:00", 60)
```

Shell

```
Meeting 1 - 09:00 to 10:00
Lunch - 12:30 to 13:15
Coding Session - 14:00 to 16:00
>
```

SCOPE

The scope of a timetable scheduler extends across various domains and industries where efficient scheduling of activities is essential. Here are some key areas where timetable schedulers find application:

Educational Institutions
Corporate Environments
Healthcare Sector
Transportation and Logistics

CONCLUSION

In conclusion, a timetable scheduler is a valuable tool with widespread applications across various industries and domains. Its significance lies in the ability to automate and optimize the complex task of scheduling activities, efficiently managing resources, and satisfying constraints and preferences