INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Geração de Horários para Instituições de Ensino Superior

SAMUEL PEDRO DA ROCHA PEREIRA
Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Schedule Generation for Higher Education Institutions

Samuel Pereira

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Advisor: Nuno Bettencourt, PhD

Porto, September 25, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 25, 2025

Abstract

University Course Scheduling Problem (UCSP) is classified as an NP-Hard problem, which is a combinatorial optimization problem that requires effective methods to manage the high level of complexity in constraints, optimization of time and resources, and assure its practical applicability to higher educational institutions. In an attempt to solve the UCSP, this document analyzes a wide array of methodologies, including Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Local Search (LS), among others. Strong emphasis is made on the aptitude and limitations of each approach, further to adaptability to specific scheduling scenarios. This work addresses the key research questions so that relevant insights, valuable for enhanced understanding, could be extracted to drive further improvement in automated scheduling systems. Based on the findings, a solution was designed and developed with a modular architecture, implementing GA and PSO, along with a dedicated correction module, which uses LS to explore nearby solutions and reduce constraint violations. Generated schedules were evaluated through fitness functions, identifying conflicts in resource assignments and penalizing them accordingly, which allowed comparisons with other schedules, including manually created ones, based on implemented criteria. This solution was then experimented with a real-world scenario, using data from the Bachelor's Degree in Informatics Engineering at ISEP. The results compare various configurations and strategies, for each algorithm, demonstrating the effectiveness of the proposed solution in generating feasible and optimized schedules, highlighting its potential for practical deployment and future research in automated university course scheduling.

Keywords: University Course Scheduling Problem (UCSP), Combinatorial Optimization, NP-Hard Problems, Genetic Algorithms (GA), Particle Swarm Optimization (PSO), Local Search (LS), Automated Scheduling Systems

Resumo

O Problema de Escalonamento de Cursos Universitários (UCSP) é classificado como um problema NP-Hard, sendo um problema de otimização combinatória que requer métodos eficazes para gerir o elevado nível de complexidade das restrições, otimizar o tempo e os recursos, e garantir a sua aplicabilidade prática em instituições de ensino superior. Na tentativa de resolver o UCSP, este documento analisa uma vasta gama de metodologias, incluindo Algoritmos Genéticos (GA), Otimização por Enxame de Partículas (PSO) e Pesquisa Local (LS), entre outras. É dada especial ênfase à aptidão e limitações de cada abordagem, bem como à adaptabilidade a cenários específicos da criação de horários. Este trabalho aborda as principais questões de investigação, de modo a extrair informações relevantes e valiosas para uma melhor compreensão, promovendo melhorias nos sistemas automatizados de agendamento. Com base nos resultados, foi concebida e desenvolvida uma solução com arquitetura modular, implementando GA e PSO, juntamente com um módulo de correção dedicado, que utiliza LS para explorar soluções próximas e reduzir violações de restrições. Os horários gerados foram avaliados através de funções de fitness, identificando conflitos na atribuição de recursos e penalizando-os adequadamente, o que permitiu comparações com outros horários, incluindo os criados manualmente, com base nos critérios implementados. Esta solução foi então experimentada num cenário real, utilizando dados da Licenciatura em Engenharia Informática do ISEP. Os resultados comparam várias configurações e estratégias para cada algoritmo, demonstrando a eficácia da solução proposta na geração de horários viáveis e otimizados, destacando o seu potencial para implementação prática e investigação futura em sistemas automatizados de agendamento universitário.

Contents

List of Figures

List of Source Code

List of Acronyms

ACO	Ant Colony Programming.
AHH	Arithemetic Heuristics.
DDD	Domain-Driven Design.
FL	Fuzzy Logic.
GA	Genetic Algorithm.
HHH	Hierarchical Heuristics.
HM	Harmony Memory.
HMS	Harmony Memory Size.
HS	Harmony Search.
ILP	Integer Linear Programming.
ISEP	Instituto Superior de Engenharia do Porto.
LEI	Licenciatura em Engenharia Informática (Bachelor's Degree in Informatics Engineering).
LS	Local Search.
OT	Orientation Tutorial Classes.
P.Porto	Polytechnic Institute of Porto.
PL	Practical Laboratorial.
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses.
PSO	Particle Swarm Optimization.
SA	Simulated Annealing.
T	Theoretical Classes.
TP	Theoretical-Practical Classes.
UCSP	University Course Scheduling Problem.

Chapter 1

Introduction

An efficient and well planned schedule is one of the key points for the success of any educational institution, directly impacting students' academic progression, faculty workload management, and institutional resource allocation. While traditional scheduling methods rely heavily on manual processes, the increasing complexity of modern academic institutions and the emerging new technologies have highlighted the limitations of these approaches. The challenges inherent in scheduling involve balancing diverse stakeholder needs, adhering to institutional policies, and responding to dynamic changes in real time. This dissertation addresses these challenges by exploring advanced technological solutions to develop a comprehensive, scalable, and adaptable scheduling framework.

1.1 Context and Motivation

The Polytechnic Institute of Porto (P.Porto) is one of the academic institutions that aggregates a wide range of higher education facilities, including Instituto Superior de Engenharia do Porto (ISEP). Among the reasons which made me apply for a Master in Information and Knowledge Systems in Informatics Engineering, in this institution was the strong tradition of innovation in engineering education along with practical relevance that has made this establishment popular.

During the period spent at ISEP, inefficient situations concerning the schedules were faced with colleagues and by my observation: overlapping classes, excessively long breaks between classes, and delays in publishing schedules. The mentioned inconveniences disturbed not only the students and the academic staff but also showed some underlying problems in the scheduling framework.

The scheduling team also faced challenges, such as excessive time and resources spent, errors in schedules that current Excel-based tools could not resolve, that were further accentuated by global trends in higher education, including increasing student enrollments, expanded program offerings, and heightened demands for optimal resource utilization. Traditional manual scheduling methods often fell short in addressing these evolving needs. Consequently, the adoption of advanced computational methods, such as artificial intelligence and data-driven decision-making tools, has become a necessity, emerging the idea of this dissertation proposal.

The proposed research aims to systematically address University Course Scheduling Problem (UCSP) by exploring advanced, technology-driven solutions. Universities now have the opportunity to not only mitigate scheduling inefficiencies but also significantly enhance stakeholder satisfaction and institutional performance.

The research work focuses on the investigation and development of scalable and adaptive solutions that can bridge the gap between traditional methodologies and technological advancements for modern university scheduling.

1.2 Problem Description

The UCSP (Muklason et al. 2023) is a critical operational problem in higher education institutions. UCSP requires the consideration of numerous variables, including course requirements, faculty availability, room capacities, and institutional policies. Moreover, it demands solutions that are not only technically robust but also adaptable, equitable, and scalable to meet the evolving needs of diverse academic environments.

The process of creating university schedules is inherently complex, as it involves aligning the needs and preferences of various stakeholders with the constraints of institutional resources. These challenges are exacerbated by the growing size and diversity of academic programs, increased student enrollments, and heightened expectations for efficient operations. Inefficient scheduling systems can lead to several significant issues:

- **Operational Inefficiency:** Poorly managed schedules result in wasted time and resources, requiring excessive manual intervention to rectify errors or accommodate last-minute changes. This inefficiency often translates into additional administrative costs and delays in schedule releases.
- **Stakeholder Dissatisfaction:** Overlapping classes, unreasonably long breaks between sessions, and inflexible scheduling are frequent sources of frustration for students and faculty. These issues can negatively impact academic progression, work-life balance, and overall satisfaction with the institution.
- **Underutilization of Resources:** Ineffective scheduling often leads to underutilization of classrooms, laboratories, and other facilities. This not only increases operational costs but also limits the institution's ability to accommodate growing academic demands.

The traditional reliance on manual scheduling methods and basic tools, such as Excel spreadsheets, is insufficient to address these multifaceted challenges. These approaches struggle to adapt to the dynamic and increasingly complex needs of modern universities, resulting in errors, inefficiencies, and delays.

1.2.1 Stakeholders

Stakeholders in university scheduling each have unique roles, expectations, constraints, needs, and priorities:

- **Scheduling Team:** At the core of the process, the scheduling team is responsible for creating, verifying and maintaining schedules. They often work under tight deadlines and must balance institutional policies, and faculty and student needs and preferences. This group faces significant challenges, including managing complex variables, handling frequent changes, and addressing errors within existing tools. Their insights and feedback are invaluable for identifying pain points and evaluating potential solutions.
- **University Administration:** Responsible for defining policies and ensuring schedules align with institutional goals, such as optimizing classroom utilization and meeting

accreditation standards. They often focus on balancing operational efficiency with stakeholder satisfaction

- **Faculty Members:** Faculty members require schedules that accommodate their teaching responsibilities while leaving room for research, administrative duties, and personal commitments. They may have specific preferences for teaching times or course loads, which must be considered to ensure satisfaction and productivity.
- **Students:** Central to the scheduling process, students seek conflict-free schedules that also accommodate the maintenance of a balance between academics, extracurricular activities, and part-time employment.
- **Technical Staff:** Tasked with maintaining and upgrading the systems in use, technical staff ensure the reliability, accuracy, and adaptability of these tools to evolving institutional requirements.

By addressing the needs and priorities of these stakeholders, the research aims to develop a scheduling framework that enhances operational efficiency, stakeholder satisfaction, and overall institutional performance.

1.2.2 Benefits

An effective and well-designed university scheduling system can have numerous benefits at many levels: operational, academic, and institutional. By addressing inefficiencies and leveraging advanced computational techniques, the proposed framework aims to deliver improvements that positively impact all stakeholders.

- **Operational Efficiency:** Reducing the time and resources needed for scheduling, including the reduction in manual interventions. This leads to fewer errors, faster schedule releases, and a significant reduction in administrative overhead. The scheduling team, for instance, can focus on strategic tasks rather than repeatedly resolving conflicts or adjusting schedules to fit institutional constraints.
- **Enhanced Student Experience:** A conflict-free, well-structured schedule enables students to have greater success. Avoiding issues like overlapping classes or extended gaps between sessions also enhances students' overall university experience, making it easier to balance academics with extracurricular activities, internships, or part-time jobs.
- **Faculty Workload Management:** Respecting faculty preferences and workload capacities, promoting satisfaction and productivity are a few of the benefits. Ensuring equitable distribution of teaching hours and accommodating personal constraints contribute to a supportive academic environment. Faculty members can devote more energy to teaching and research, free from the frustrations of inconsistent or unfair scheduling.
- **Optimal Resource Utilization:** Classrooms, laboratories, and other facilities are finite resources that require careful planning to ensure maximum efficiency. An optimized schedule prevents underutilization or overcrowding, balancing demand with availability. This not only reduces operational costs but also supports sustainability efforts by minimizing the need for additional infrastructure. Unassigned or underutilized resources can be repurposed for revenue generation, such as leasing to external organizations for

events or training sessions, or allocated to support curriculum expansion without significant new infrastructure investments. By maximizing existing capacity, institutions can introduce new courses and better accommodate growing academic demands.

- **Institutional Performance and Reputation:** Timely and effective scheduling demonstrates institutional competence, trust and satisfaction among students, faculty, and administrative staff. Improved scheduling processes can enhance an institution's reputation, attracting more students and faculty.
- **Adaptability to Change:** In dynamic academic environments, scheduling needs can shift rapidly due to new course offerings, policy changes, or external factors such as public health emergencies. An adaptable system ensures resilience, allowing institutions to respond to changes quickly without compromising quality or equity.

1.3 Methodology

The iterative and participatory nature of action research makes it an ideal methodology for developing and refining scheduling solutions. The process involves close collaboration with stakeholders, including the scheduling team, faculty, and technical staff, to ensure the solution aligns with needs and goals of the project.

1.3.1 Phases of Action Research

1. **Problem Analysis** (Chapters 1-3):
   - Gather requirements for scheduling.
   - Analyze existing scheduling systems and their limitations.
   - Analyze the integration with existing systems.
2. **Initial Solution Design** (Chapter 4):
   - Develop prototype scheduling models starting with single class scheduling.
3. **Incremental Complexity Scaling** (Chapter 5):
   - Gradually increase the complexity in scheduling, generating schedules for courses, combining schedules from multiple classes, and generating schedules for departments, combining schedules from multiple courses.
   - Test the system's ability to handle dynamic changes, such as last-minute faculty unavailability or fluctuating student enrollments.
4. **Evaluation and Feedback** (Chapter 6):
   - Implement the scheduling solution in controlled environments.
   - Collect feedback from stakeholders to identify strengths and areas for improvement.
   - Refine the model based on iterative testing and stakeholder input.
5. **Validation** (Chapter 6):

- Compare the performance of the automated scheduling solution against manually created schedules using predefined metrics, such as efficiency, conflict resolution, and stakeholder satisfaction.

1.4 Goals

The dissertation aims to achieve the following objectives:

- **Primary Goal:** Improve the schedule generation process, either by reducing the time and resources needed, improving the quality of the generated schedules, or both outcomes.
- **Secondary Goals:**
  - Explore and apply cutting-edge algorithms and AI methods to automate and optimize the scheduling process.
  - Design a proposed solution capable of accommodating dynamic institutional changes, such as varying class sizes, faculty availability, and infrastructural constraints.
  - Establish and evaluate metrics to assess the performance of the proposed scheduling solution in comparison to traditional methods. Metrics include time savings, resource utilization, conflict resolution, and stakeholder satisfaction.

1.5 Contribution

The project's source code is available on GitHub[1], enabling researchers and academic institutions to review, reuse, and extend the developed scheduling framework.

1.6 Limitations

While this dissertation presents a robust approach to addressing the problem in analysis, it is important to recognize its limitations:

- Small research team limits the capacity for concurrent exploration of diverse methodologies and exhaustive testing. The reliance on a small number of perspective might also reduce the diversity of insights compared to large collaborative research teams.
- The study's data collection and validation are confined to the specific context of P.Porto and its scheduling practices and policies.
- Developing, testing, and refining scheduling models are also limited, due to the time constraints of the nature for this research.

By identifying these limitations, the dissertation provides a transparent foundation for further research, highlighting areas that future studies could address to build upon this work.

[1] `https://github.com/1201274/SchoolScheduler`

1.7 Ethical Considerations

This work follows the guidelines of the ethical order from the Order of Engineers (Engenheiros 2016) and the Code of Good Practices and Conduct of P. Porto (Porto 2020). An integral part of this commitment includes the Declaration of Integrity, guaranteeing at all times respect for academic honesty, originality, and good practices in developing the research.

One of the key ethical considerations in this dissertation is the **fairness** (Tung Ngo et al. 2021) of the automated scheduling process in the equal treatment of students and faculty. The algorithm should be oriented toward the principle of fairness in distributing course loads, teaching assignments, and schedules to all stakeholders.

- **Fairness for Students:** The scheduling algorithm needs to be very sensitive to the differing needs and preferences of the students regarding their availability, course choices, and extracurricular commitments. It is vital to ensure that the schedules produced will distribute the academic workloads fairly among students, considering variables such as class timings, preferred subjects, and any special requirements.
- **Equity for the Faculty:** The schedule generation would also treat the faculty equitably. This system shall not provide excessive loads by giving the faculties too many hours of teaching along with conflicts in each professional's preferences over time. Course assignments must come out balanced in nature to keep out any forms of bias that tend to take an upper hand as regards a set of particular faculty based on some factors, say rank, tenureship among other personal details.

Other ethical considerations in this dissertation are the following:

- **Academic Integrity:** In consideration of the IPP Code of Conduct (Porto 2020), this dissertation is committed to academic integrity in properly citing every relevant source, thereby giving credit to the authors of those works and, hence, not committing plagiarism.
- **Transparency and Accountability:** Scheduling algorithm development is transparent, and the decision-making process is clearly accessible to all stakeholders. Issues or concerns that may be considered unfair in relation to the system can be openly conveyed and fixed.
- **Social Responsibility:** This research recognizes the **social impact** of automated scheduling, especially in terms of providing equal opportunity for all students and equal distribution of workload among faculty. The system is designed to address economic, cultural, and logistical challenges so that the process of scheduling benefits all parties involved.
- **Data Privacy:** The research will ensure that any data collected from stakeholders, such as students and faculty, is handled adhering to data protection regulations and institutional policies. Personal information will be anonymized where possible, and data will only be used for the purposes of this research.

This dissertation will, therefore, focus on fairness, transparency, and accountability in an effort to contribute toward a more equitable and efficient academic scheduling system, while observing the highest ethical standards in software development and research.

1.7.1 Planning

Planning is integral to the successful execution of this research, providing a clear roadmap that ensures alignment with objectives and stakeholder expectations. The project was divided into distinct phases, each designed to systematically address the research and implementation goals:

- **Planning**: Outlying the development process, to provide a comprehensive view of how the research will progress from inception to completion, ensuring clarity in execution.
- **State-of-the-Art**: Conducting a comprehensive literature review to assess current practices, identify gaps, and inform subsequent phases.
- **Intermediate Presentation**: Preparing and presenting an intermediate report to showcase progress and gather feedback.
- **Presentation Review**: Applying the feedback to the work done.
- **Schedules for Classes**: Designing, implementing, and testing scheduling models tailored for individual class schedules.
- **Schedules for Courses**: Extending the models to coordinate multi-course scheduling, ensuring cross-compatibility and scalability.
- **Schedules for Departments**: Developing solutions for departmental-level scheduling, addressing interdepartmental dependencies and resource allocation.
- **Conclusion**: Summarizing findings and presenting results.

1.8 Document Structure

The dissertation's structure unfolds logically after the introduction. The **State-of-the-Art** chapter examines the evolution of university scheduling practices, emphasizing advancements in computational methods, artificial intelligence, and optimization techniques. It identifies gaps in existing research and highlights the need for innovative approaches to address the challenges inherent in higher education scheduling.

The **System Analysis and Design** chapter delves into the specific requirements and constraints of the scheduling problem, detailing the methodologies and algorithms employed in the proposed solution. It outlines the system architecture, data models, and the integration of stakeholder feedback into the design process.

Based on the work developed in this chapter, the subsequent chapters present the **Implementation** and a **Case Study** chapters, which provide a comprehensive overview of the practical application of the proposed scheduling framework. The Implementation chapter details the technical aspects of the system, while the Case Study chapter presents a real-world application of the scheduling solution, experimenting with different scenarios and configurations.

Finally, the **Conclusion** chapter synthesizes the research findings, discusses the implications of the work, and outlines potential avenues for future research and development in university scheduling.

Chapter 2

State-of-the-Art

This chapter presents a systematic review of the literature related to the UCSP. The purpose of this review is to identify and analyze existing approaches in educational scheduling. The chapter outlines the methodology used and its findings.

2.1 Methodology

This systematic review follows the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) framework to critically analyze existing studies related to scheduling algorithms, variables, and techniques of evaluation. PRISMA ensures that the review of existing literature is structured and systematic. This will be done through the following process.

2.1.1 Identification

Conducting comprehensive database searches using keywords.

- **Query used:** "school schedule generation" OR "educational scheduling" OR "course timetabling" AND ("AI" OR "artificial intelligence" OR "heuristic" OR "genetic algorithms" OR "machine learning")
- **Sources:**
  - ACM Digital Library: 61
  - Google Scholar: 17
  - IEEE Xplore: 34
  - Web of Science: 50
  - Chat GPT: 8
  - Other sources: 14
  - Duplicates: 20
  - **Total**: 164

2.1.2 Screening

To ensure a focused and meaningful analysis of higher education scheduling, a structured filtering process was applied to the literature review:

- **Inclusion Criteria:**
  - Studies directly related to schedule generation in educational institutions.
  - Articles comparing the application and performance of different algorithms or AI techniques in scheduling.
- **Exclusion Criteria:**
  - Articles older than ten years, to maintain relevance to current technologies and methodologies.
  - Studies written in languages other than English or Portuguese, ensuring accessibility and comprehension.
  - Articles without a publicly available PDF file, limiting the ability to perform in-depth analysis.

After applying these filters, the initial pool of articles was refined to 58 studies, for full-text reading.

2.1.3 Eligibility

Reviewing the shortlisted studies in detail to extract insights on variables, methods, and evaluation metrics. Only 30 studies were included in this review.

2.1.4 Synthesis

Categorizing and analyzing findings to identify patterns, gaps, and opportunities in the current research landscape.

2.1.5 Research Questions

This research addresses the following key questions:

1. What algorithms and AI techniques are applied in schedule generation for educational institutions?
2. What methods can be employed to evaluate and compare the quality of generated schedules?
3. Which variables and attributes most significantly influence the scheduling process and its outcomes?

These questions guide the exploration of the literature and inform the design and testing of the proposed scheduling framework.

2.2 Literature Review

University Course Scheduling Problem is classified as a Non-deterministic Polynomial-time Hard (NP-Hard) problem, since it is a combinatorial optimization problem, which is solved by searching for an optimal solution for the time and resources given(Muklason et al. 2023). By addressing these research questions, relevant insights can be retrieved that refine previous

approaches and optimize schedule generation. This comprehensive understanding helps improve algorithmic designs, adapt strategies to diverse institutional requirements, and develop more efficient and scalable scheduling systems.

2.2.1 Algorithms and AI Techniques for Schedule Generation

There are numerous types of algorithms which are applied to scheduling research; some of these are metaheuristics, hyperheuristics, mathematical optimization, matheuristics, and hybrid approaches(Tan et al. 2021).

Genetic Algorithm (GA)

Genetic Algorithm (GA), inspired by natural selection, are evolutionary algorithms (Muklason et al. 2023). These population-based heuristic methods are widely used in scheduling problems (Tung Ngo et al. 2021). Solutions to scheduling problems are represented as chromosomes (Saptarini, Ciptayani, and Pumama 2020), and the initial population is generated randomly (Romaguera et al. 2024).

It follows a four-phase process for each generation of the algorithm:

1. **Parent Selection:** Selection of two parent chromosomes based on their fitness (Jiang and Liu 2019).
2. **Crossover:** New offspring are generated by the recombination of two parents, using techniques such as single-point, two-point, or N-point crossover (Saptarini, Ciptayani, and Pumama 2020).
3. **Mutation:** Modify the genetic value of some fittest individuals in the population(Jiang and Liu 2019). The probability is quite small in value(Saptarini, Ciptayani, and Pumama 2020).
4. **Selection:** Replacement of the weaker individuals of a population with fitter ones(Romaguera et al. 2024) with the purpose of enhancing the gene pool, increasing the chances of finding a better solution(Devi et al. 2024).

Even though the approaches driven by GA achieve good solutions, computation times are considerably large — around two hours for handling 3,000 students (Tung Ngo et al. 2021). In general, if the number of resources and the complexity of their constraints increase, this algorithm might lead to more execution time (Romaguera et al. 2024).

Other recent proposals try to enhance the performance of GAs by combining them with other techniques. For example, the following methods are proposed:

- Local Search (Devi et al. 2024)
- Quantum Technology (Shuguang and Lin 2019)
- Harmony Search (Rodprasert, Taetragool, and Akkarajitsakul 2023)
- Chaos (Luo and Niu 2024)
- Simulated Annealing (Yazdani, Naderi, and Zeinali 2017)
- Fuzzy Logic (Teoh, Wibowo, and Ngadiman 2015)
- Ant Colony Optimization (Al-Mahmud and Akhand 2014).

Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) draws inspiration from the social behavior of swarms and implements a simulation of organisms like birds, fish, bats, or fireflies moving (Hossain et al. 2019). The algorithm works on the basis of exploration and exploitation of the search space. (Teoh, Wibowo, and Ngadiman 2015).

PSO works with a swarm of particles which are randomly assigned to the search space; each particle represents a solution. In each iteration, it performs the following:

1. **Velocity Computation:** Compute the velocities for individual particles.
2. **Particles Movement:** Particles move according to velocities.
3. **Fitness Calculation:** Evaluate the fitness of every particle and update global and personal best positions if necessary (Hossain et al. 2019).

Although designed for continuous optimization, PSO has been adapted for combinatorial tasks such as university course scheduling. However, its performance depends on effective modifications since timetabling problems are inherently discrete (Hossain et al. 2019).

PSO has been proved, in many cases, to exhibit better fitness than other GAs (Ramdania et al. 2019) (Teoh, Wibowo, and Ngadiman 2015).

In order to improve the results, the PSO has been combined with methods such as:

- Local Search (Rashmi and Abhishek 2021)
- Selective Search (Hossain et al. 2019).

Fuzzy Logic

Fuzzy Logic (FL) is a type of probabilistic reasoning that gives a wide spectrum of decision-making options. It extends the evaluation by incorporating linguistic variables into it, which can assess the constraints with varied truth values rather than simple "true" or "false." This ability makes the decision-making process more flexible and accurate(Teoh, Wibowo, and Ngadiman 2015).

FL has succeeded in optimizing complicated scheduling tasks for results that much better reflect a real-world scenario. The use of linguistic variables can result in more stable solutions in less time(Rashmi and Abhishek 2021). FL does help with the optimization of soft constraint configuration by means of fuzzy values, thereby gaining more efficiency in scheduling. (Babaei and Hadidi 2017)

Integer Linear Programming

Integer Linear Programming (ILP) is the optimization model that uses an integer variable with a linear constraint for solving the problems in linear format.

Because it is a deterministic method, it calculates virtually the same solutions under multiple executions, producing a zero standard deviation in cost.

ILP proved less viable in some scheduling models because ILP generated highly valued costs along with a larger number of unscheduled classes. On the other hand, metaheuristic approaches, including GA and Harmony Search (HS), consistently outperform ILP in both efficiency and cost. (Rodprasert, Taetragool, and Akkarajitsakul 2023)

Ant Colony Optimization (ACO)

Ant Colony Programming (ACO) takes inspiration from the deposition of pheromone from ants to guide identify the shortest paths to food. ACO is a stochastic and multi-directional search algorithm, but does not guarantee an optimal solution (Teoh, Wibowo, and Ngadiman 2015). Enhancements include combinations with:

- Local Search (Sakal, Fieldsend, and Keedwell 2021)
- Genetic Algorithms (Al-Mahmud and Akhand 2014).

Graph Coloring Algorithm

Graph Coloring represents courses as vertices and conflicts as edges. It assigns timeslots to vertices while ensuring that no adjacent nodes get the same timeslot. Although very powerful in constructing initial solutions, it performs poorly when addressing soft constraints (Rashmi and Abhishek 2021).

Graph Coloring is a methodology that models a problem using vertices or nodes, which are interconnected by edges. In the course timetabling context, vertices represent courses, whereas the color of the vertex is mapped to a specific timeslot. Edges connect vertices of those courses that are in conflict and hence cannot be scheduled in the same timeslot(Muklason et al. 2023).

This solves the timetabling problem, whereby events or courses are arranged in timeslots through heuristics. Each course is scheduled into some timeslot so that conflicts may be avoided without any violation of the hard constraint. It consists of two phases:

1. **Construction phase:** where potential solutions are generated
2. **Improvement phase:** which refines the solution to return the optimum schedule.

But this technique is very time-consuming to solve the problem and is not effective for soft constraints settlement.(Rashmi and Abhishek 2021)

Harmony Search

Harmony Search (HS) Algorithm imitates the process of harmonizing by a band that tries to compose better harmonies. A solution is called a harmony, and the set of solutions is named Harmony Memory (HM). The Harmony Memory Size (HMS) determines the size of the population. The proccess consists of a few steps:

1. **Harmony Memory construction:** generated randomly
2. **Harmony improvisation:** generating a new harmony considering memory, adjusting the pitch, and selecting values randomly
3. **Harmony Memory updating:** by comparing the new harmony with the worst harmony in HM. If it offers a better solution, it replaces the worst harmony and updates the HM.

This process continues until the number of iterations is reached as specified. There are three major rules that principally govern HS, namely memory consideration rate (HMCR), pitch adjustment rate (PAR), and pitch bandwidth (bw).(Hossain et al. 2019; Rodprasert, Taetragool, and Akkarajitsakul 2023).

Local Search (LS)

Local Search (LS) algorithms start with an initial solution and then perform an iterative process in the neighborhood of solutions to improve the objective function (Feutrier, Kessaci, and Veerapen 2021). However, these methods have the disadvantage of getting stuck in local optima or on plateaus (Jamal 2017).

Based on this algorithm, variants such as Tabu Search and Simulated Annealing started to appear.

Tabu Search

Tabu Search uses a temporary memory, the tabu list, to store the recently visited solutions in order not to evaluate them again and get stuck in local optima (Teoh, Wibowo, and Ngadiman 2015). This mechanism allows the algorithm to explore non-improving moves and escape cycles(Rashmi and Abhishek 2021).

Although effective in escaping cycles, resource evaluations and problem formulations are computationally costly (Rashmi and Abhishek 2021). Tabu Search was found combined with:

- Simulated Annealing (Shao, Lee, and Kim 2024)
- Conic Scalarization (Can, Ustun, and Saglam 2023).

Simulated Annealing (SA)

Simulated Annealing (SA) is inspired by the process in metallurgy: heated metals are given time to cool down to build a stable crystal lattice structure with an energy state. The algorithm initially generates a random solution, then generates an adjacent solution. Both solutions are evaluated through objective functions. It accepts the neighboring solution if the cost is less than the cost of the original solution so that the energy of the system goes down. The worsening of the new solution can be accepted according to a gradually decreasing probability with time.(Teoh, Wibowo, and Ngadiman 2015)

The SA has acceptance criteria that enables the algorithm to escape from local optima, which determines whether or not the new solution generated is accepted. There is a possibility of accepting worse solutions under specific conditions(Teoh, Wibowo, and Ngadiman 2015). Sometimes, SA is partnered with Genetic Algorithm(Yazdani, Naderi, and Zeinali 2017)

Hyper-Heuristics

Hyper-heuristics are high-level search strategies that operate over the heuristic space, selecting and applying low-level heuristics (move operators) based on information gathered from the search process(Muklason et al. 2023). There are two types of hyper-heuristics: those that generate Arithemetic Heuristics (AHH) and those that create Hierarchical Heuristics (HHH).(Pillay and Özcan 2019)

They function especially well in complex problems such as course timetabling since they offer the ideal platform for the selection of the most appropriate heuristic that suits the solution needed. The search strategy will provide a high-level heuristic choice for refining the solution over time. Evidence from many research studies indicates that in timetabling of courses, hyper-heuristics have produced superior outcomes against competing algorithms. These

algorithms are straightforward to implement and address complicated issues with relatively very fair processing times(Muklason et al. 2023).

Other Algorithms

- Firefly Algorithm(Thepphakorn and Pongcharoen 2023)
- Hill Climbing(Feutrier, Kessaci, and Veerapen 2021; Jamal 2017)
- Clustering(Babaei and Hadidi 2017; Erdeniz and Felfernig 2018)
- Piece of Pie Search(Pothitos and Stamatopoulos 2016)
- Great Deluge(Feutrier, Kessaci, and Veerapen 2021)
- B-Method(Schneider, Leuschel, and Witt 2018)
- Binary Cuckoo Search(Zheng et al. 2022)

2.2.2 Methods to Evaluate and Compare Generated Schedules

Schedules are evaluated based on their ability to meet the constraints and preferences set by the educational institution. The evaluation process is crucial for determining the quality of a schedule and involves both quantitative and qualitative measures. There are two kinds of constraints: **Hard Constraints** and **Soft Constraints**.

- **Hard Constraints:** These are requirements that need to be strictly followed by the schedule for it to be valid. A typical example is that a teacher cannot give two lectures at the same time. If a schedule violates any of the hard constraints, then the solution is invalid(Tan et al. 2021).
- **Soft Constraints:** These are preferences and aspects that improve the quality of schedules. The non-compliance with these constraints does not invalidate the solution, only the quality is penalized.(Tan et al. 2021). For instance, assigning classes in days of the week on which a teacher prefers to teach is an example of a soft constraint.(Hossain et al. 2019).

The evaluation of schedules is typically done through a fitness function, which quantifies how well a schedule meets the constraints and preferences. The fitness function applies weights to each constraint, allowing for a more nuanced evaluation of the schedule's quality.

2.2.3 Variables and Attributes in the Scheduling Process

Some of the most important variables and attributes that consider automated educational schedule generators, where resources and constraints and their links play an important role, are listed below. The following elements have been identified and categorized as (Devi et al. 2024; Teoh, Wibowo, and Ngadiman 2015; Tung Ngo et al. 2021; Yang, Wu, and Teng 2016):

- **List of Teachers:** This is the list containing the infromation of all teachers to be scheduled. Each Teacher has:
  - **Acronym:** The identifying acronym of the teacher.
  - **Maximum Daily Workload:** This is the maximum number of hours or classes a teacher can teach per day

  - **List of Constraints:** Constraints that apply to the teacher, such as preferred teaching days or times.

- **List of Departments:** It defines the organizational structure of the institution by grouping related academic programs and faculties. Each Department has:
  - **Course List:** Courses offered by the departments, detailing the subjects taught. Each Course has:
    - **List of Subject:** It represents the modular components of each course. Each Subject has:
      - **List of Class Types:** Details specific sessions within a curricular unit. Each Class Type has a **Duration** and a **Type:** Indicates the format, such as a Theoretical Classes (T), Theoretical-Practical Classes (TP), Orientation Tutorial Classes (OT), or Practical Laboratorial (PL).
    - **Class List:** A breakdown of student groups enrolled in each course. Each class has:
      - **Number of Students:** Shows the size of the class, important for room allocation and capacity planning.
- **List of Timeslots:** Times at which classes may be scheduled. Each Timeslot has:
  - **Day of the Week:** Indicates the day of the week
  - **Start and End Times:** Entails from and to what time the slot representing.
  - **Period of the Day:** Morning, afternoon, evening or nighttime classes.
- **Rooms List:** List of all classrooms and facilities available to be assigned. Each Room has:
  - **Maximum Capacity:** This defines the number of students the room can accommodate.
  - **Type of Room:** Indicates the format of the room, if it is an Auditorium, Laboratory, or Online.
- **List of Constraints:** Constraints are the rules and preferences for generating schedules. Each Constraint is either:
  - **Hard Constraints:** Those which must be strictly satisfied. Each hard constraint has:
    - **Constraint:** The particular rule or condition.
  - **Soft Constraints:** Preferences, ideally to be met but not mandatory. Each soft constraint has:
    - **Constraint:** The particular rule or condition.
    - **Weight (Optional):** This is the weight or importance that the constraint carries, often used in optimization algorithms in rating solutions.
    - **Fuzzy Degree of Completion (Optional):** This attribute quantifies the extent to which a soft constraint is satisfied, expressed as a value between 0 (not satisfied) and 1 (fully satisfied)

- **A Solution or List of Solutions:** The result of the scheduling process. Each Solution has:
  - **Schedule:** Defines the resource allocation. Each Schedule has:
    - **List of Assignment:** This is the list of assignments of resources to timeslots. Each Assignment has:
      - **Teacher:** The instructor of the lesson.
      - **Room:** The location of the class.
      - **Class:** The students attending.
      - **Subject:** The subject being taught.
      - **Class Type:** The specific session.
  - **Fitness Value:** It tells about the quality of the schedule, especially in satisfying the constraints.
  - **List of Violated Constraints:** This gives a breaksakdown of all unfulfilled preferences and is useful in post-processing or further optimization.

Other variables specific to the algorithms will appear during their application, required due to the unique structure and requirements of scheduling problems. These variables can be associated with properties, such as population diversity in Genetic Algorithms, or number of iterations in Particle Swarm Optimization.

2.3 Summary

The University Course Scheduling Problem is a very challenging problem because of its intrinsic complexity and the numerous constraints involved.

This review has discussed various algorithms and techniques proposed for the problem, underlining their potential and limitations. Among these, methods such as Genetic Algorithms, Particle Swarm Optimization, and Hyper-Heuristics have shown promising results for different aspects of UCSP. However, most of them have normally proved to be effective only in certain problem contexts and for particular algorithm designs. An effective solution to UCSP involves a trade-off between computational efficiency and solution quality, considering real-world constraints like lecturer preferences and institutional policies. The review also brings into focus the importance of hybrid approaches and adaptive methods that can combine the strengths of multiple techniques. The research work will give a concrete basis for further exploration with the answers to the research questions and the synthesizing of existing literature. Future research should be directed toward integrating emerging technologies, such as machine learning and quantum computing, into developing more robust and adaptive solutions. Ultimately, the enhancement of UCSP methodologies holds promise for streamlining administrative processes, enhancing resource utilization, and improving the overall educational experience. The following chapters present a practical application, focusing on two algorithms: Genetic Algorithm and Particle Swarm Optimization.

Chapter 3

Analysis

This chapter provides a detailed analysis of the system. It outlines the project scope, domain model, and requirements that will guide the design of the scheduling system.

3.1 Project Scope

A common challenge in software development is the lack of a clear understanding of the problem domain, which can lead to misaligned requirements and ineffective solutions. To address this, it is essential to establish a well-defined project scope through the application of common software engineering practices, particularly Domain-Driven Design.

Domain-Driven Design (DDD) is a methodology that emphasizes the importance of understanding the problem domain and creating a model that accurately reflects it. In this project, DDD will be employed to ensure that the scheduling system is designed with a clear focus on the domain of university course scheduling. This is represented in the form of a domain model.

3.2 Domain Model

The domain model will capture the essential concepts and relationships within the university course scheduling domain, providing a foundation for the system's architecture and functionality. This model will be developed through collaboration with stakeholders to ensure that it accurately reflects their needs and expectations.

Figure 3.1 illustrates a portion of the domain model for the scheduling system, highlighting the key entities and their relationships.

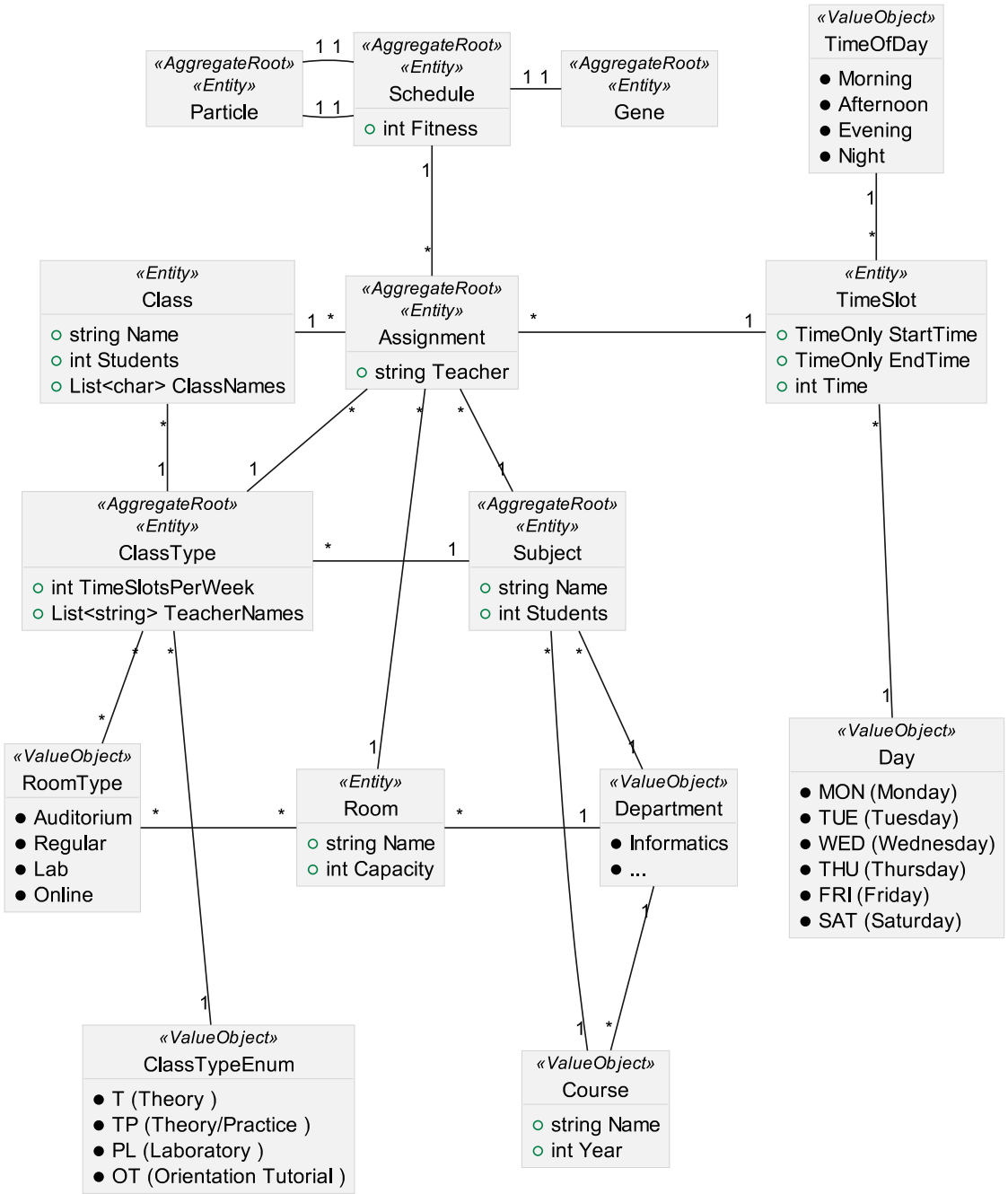

Figure 3.1: Domain Model for the Scheduling System

The main entities in the domain model are:

- **Assignment**: Represents the assignment of a class to a specific timeslot, room, and teacher.
- **Schedule**: Represents a collection of assignments that make up a complete schedule.

To support the scheduling process, the following entities and value objects are also included in the domain model:

- **TimeSlot**: Represents a specific time period during which a class can be scheduled, including the day of the week and time of day (e.g., morning, afternoon ...).
- **Room**: Represents a physical room where classes can be held, including its type and capacity.
- **Department**: Represents the academic department, defining the scope of courses and rooms within that department.
- **Course**: Represents a course offered by the university, which can consist of multiple subjects and classes.
- **Subject**: Represents a course subject that can be scheduled.
- **ClassType**: Represents the type of class (e.g., lecture, lab) associated with a subject.
- **Class**: Represents a specific instance of a class type for a specific class of students.

The system will also utilize specialised entities to operate the scheduling algorithms, such as:

- **Gene**: Represents a candidate solution in Genetic Algorithms.
- **Particle**: Represents a candidate solution in Particle Swarm Optimization, containing the current and personal best schedule.

3.3 Requirements

The requirements f or the scheduling system are derived from the domain model and the needs of the stakeholders. They are categorized into several key areas to ensure a comprehensive understanding of the system's functionality and constraints. A industry standard for requirements specification is FURPS, which divides requirements into the following categories.

3.3.1 Functionality

The functionality defines the core features the system must provide in order to address the University Course Scheduling Problem. It encompasses the essential capabilities that ensure schedules are generated, evaluated, and customized according to institutional needs.

- Automated schedule generation using heuristic and AI-based techniques.
- Resource allocation covering classrooms and teachers, ensuring optimal utilization.
- Conflict detection for overlapping courses, instructor assignments, and resource usage.
- Evaluation mechanisms to compare schedule quality.
- Support for manual adjustments to generated schedules.
- Automated conflict correction and schedule improvement capabilities.
- Visual interface to display generated schedules, from different points of view.
- Export metadata to generate graphs and statistics.

3.3.2 Usability

Usability focuses on how effectively users can interact with the system. It emphasizes the need for an intuitive interface, and ease of use.

- Intuitive, user-friendly user interfaces, supported by guided workflows and contextual help.

3.3.3 Reliability

Reliability ensures that the system consistently produces valid schedules while remaining robust against errors, interruptions, or unexpected data inconsistencies.

- High accuracy in satisfying institutional rules and constraints.
- Fault-tolerance to handle incomplete or inconsistent input data.
- Logging and monitoring of system activities for accountability.

3.3.4 Performance

Performance requirements specify the system's efficiency, scalability, and responsiveness. These criteria ensure that even large universities can generate schedules within reasonable time frames while making optimal use of resources.

- Scalability to support multiple departments and rooms without degradation.
- Optimized algorithms for an efficient use of memory and processing resources during scheduling runs.

3.3.5 Supportability

Supportability reflects the system's ability to evolve, adapt, and be maintained over time. It emphasizes modularity, extensibility, and configurability, ensuring that the system remains sustainable.

- Modular architecture to simplify maintenance and upgrades.
- Support for multiple scheduling algorithms, including Genetic Algorithms and Particle Swarm Optimization.
- Configurable scheduling engine to adapt to current necessities.

3.4 Summary

This chapter has outlined the project scope, domain model, and requirements for the scheduling system. By applying Domain-Driven Design principles, a clear understanding of the problem domain has been established, providing a solid foundation for the system's design and implementation. The next chapter will build upon this foundation to develop a detailed design for the scheduling system.

Chapter 4

Design

The design phase translates the requirements into a structured architecture that guides the development of the scheduling system. This chapter outlines the system architecture, design patterns, and principles that will be employed to ensure a robust and maintainable solution.

4.1 Design Patterns

Design patterns provide reusable solutions to common problems in software design, and enhance the system's modularity, maintainability, and extensibility. Key design patterns include:

- **General Responsibility Assignment Software Patterns (GRASP)**: Provides guidelines for assigning responsibilities to classes and objects, ensuring a clear and coherent design.
  - **Information Expert**: Assigns responsibilities to the class that has the necessary information to fulfill them, promoting encapsulation and cohesion.
  - **Creator**: Assigns the responsibility of creating an object to a class has the information needed to create it, ensuring proper object creation and management.
  - **Controller**: Assigns the responsibility of handling system events to a controller class, managing the flow of control and interactions between components.
  - **Low Coupling**: Promotes low dependency between classes, enhancing maintainability and flexibility.
  - **High Cohesion**: Encourages classes to have focused responsibilities, improving clarity and reducing complexity.
  - **Polymorphism**: Utilizes polymorphic behavior to handle variations in algorithms or operations, enhancing extensibility and adaptability.
  - **Pure Fabrication**: Creates artificial classes to achieve low coupling and high cohesion, regardless of real-world entities representation, enhancing design quality.
- **Model-View-Controller (MVC)**: Separates the user interface, business logic, and data model to promote modularity and ease of maintenance.
- **Strategy Pattern**: Allows for the selection of different scheduling algorithms at runtime, promoting extensibility and adaptability.

- **Factory Pattern**: Facilitates the creation of complex objects, such as schedules, without exposing the instantiation logic.
- **Singleton Pattern**: Ensures a single instance of a class, providing a global point of access and managing shared resources effectively.

4.2 Design Principles

Design principles guide the overall architecture and implementation of the system, ensuring that it is robust, maintainable, and adaptable to future changes. Key design principles include:

- **Separation of Concerns**: Each component of the system should have a distinct responsibility, promoting modularity and ease of maintenance.
- **SOLID Principles**: A set of five design principles that promote maintainable and scalable software architecture.
  - **Single Responsibility Principle**: Each class should have on and only one responsibility, ensuring that it is focused and cohesive.
  - **Open/Closed Principle**: The system should be open for extension but closed for modification, allowing new features to be added without altering existing code.
  - **Liskov Substitution Principle**: Class attributes should be replaceable with instances of their subtypes without affecting the validity of the program.
  - **Interface Segregation Principle**: Clients should not be forced to depend on interfaces not in use, ensuring that interfaces are tailored to specific needs.
  - **Dependency Inversion Principle**: High-level modules should not depend on low-level modules; both should depend on abstractions, promoting flexibility and testability.

4.3 System Architecture

The system architecture is designed to support the requirements outlined in the previous section, ensuring that the scheduling system is modular, scalable, and maintainable, by the application of the previously mentioned design patterns and principles. The architecture follows a layered approach, separating concerns and allowing for independent development of each layer. The architecture is represented using the C4 model (Vázquez-Ingelmo, García-Holgado, and García-Peñalvo 2020), which provides a clear and structured way to visualize the system at different levels of abstraction. The C4 model consists of four levels: Context, Container, Component, and Code. For the purpose of this project, we will focus on the Container and Component levels to provide a detailed view of the system's architecture.

4.3.1 Container Diagram

The Container Diagram provides a high-level overview of the system's architecture, illustrating the main containers (applications and services) and their interactions. Each container represents a distinct part of the system, with its own responsibilities and technologies. The following figure, Figure 4.1, illustrates the Container Diagram for the scheduling system.

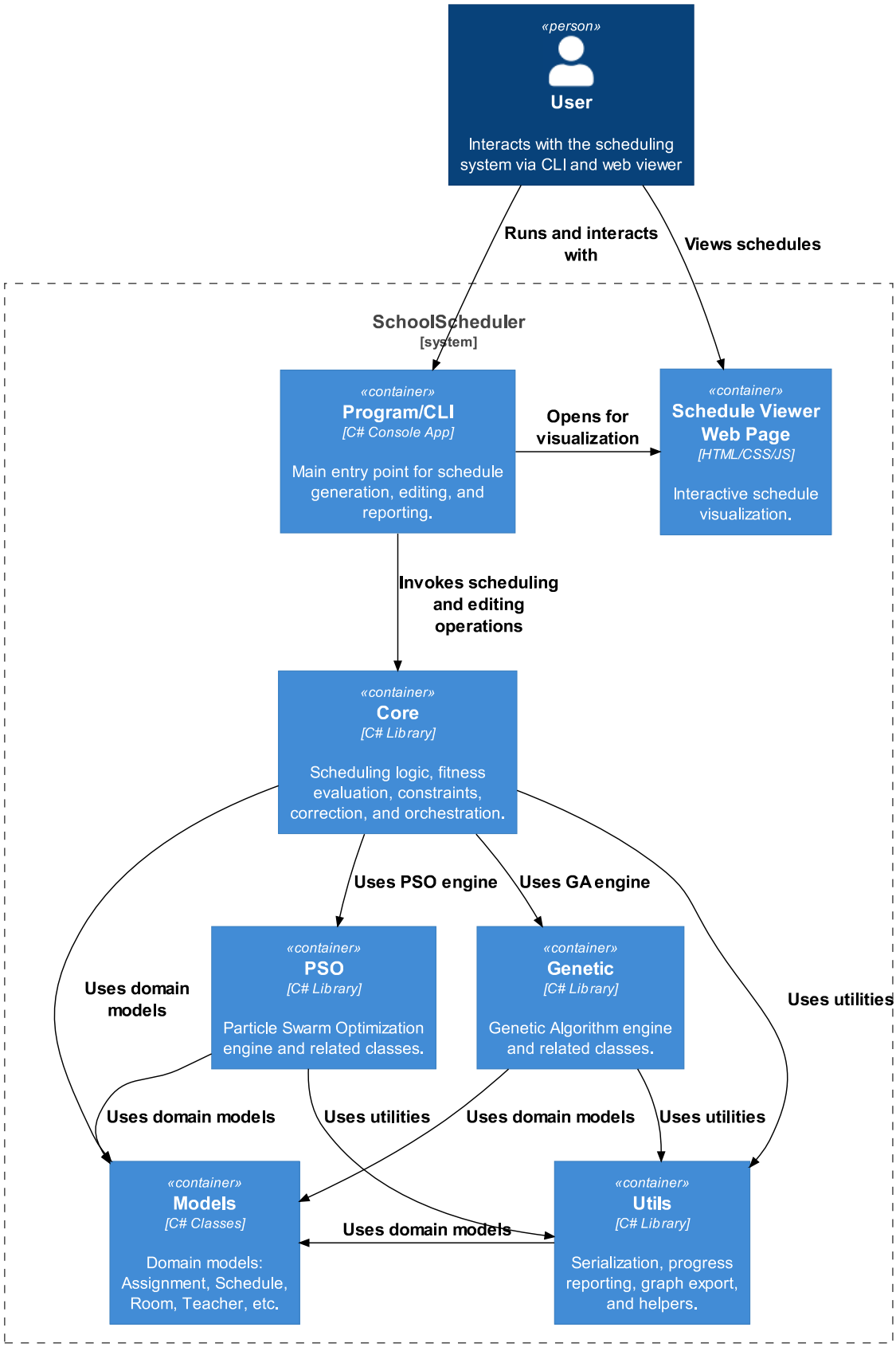

Figure 4.1: C4 Container Diagram

The architecture consists of the following layers:

- **Presentation Layer**: This layer provides the user interface for interacting with the scheduling system.
  - **Program/CLI**: A command-line interface that allows users to interact with the scheduling system through text-based commands.
  - **Schedule Viewer Web Page**: A web-based interface that allows users to view generated schedules, from different points of view, such as by room, teacher, or class. Each view includes details such as assigned rooms, times, teachers, and some of the conflicts.
- **Application Layer**: This layer contains the core business logic of the scheduling system. It includes components for managing schedules, constraints and conflicts, and resources, as well as coordinating the scheduling algorithms.
  - **Core**: The main component responsible for orchestrating the scheduling process, including invoking the appropriate scheduling algorithms and managing the overall workflow.
  - **Utils**: Provides utility functions and classes used throughout the application.
- **Scheduling Algorithms Layer**: This layer encapsulates the various scheduling algorithms used by the system, such as Genetic Algorithms and Particle Swarm Optimization. Each algorithm is implemented as a separate component, allowing for easy addition or modification of algorithms in the future.
  - **Genetic**: Implements the Genetic Algorithm for schedule generation.
  - **PSO**: Implements the Particle Swarm Optimization algorithm.
- **Model Layer**: This layer defines the domain model.
  - **Models**: Contains the domain entities, including entities such as Schedule, Assignment, Room, Teacher, and TimeSlot, and their relationships.

4.3.2 Component Diagrams

The Component Diagrams provide a more detailed view of the individual components within each container, illustrating their responsibilities and interactions. Each component is responsiblefor a specific piece of functionality, promoting modularity and ease of maintenance. Due to the complexity and number of components in the system, the Component Diagrams are divided into partial representations, focusing on the components affecting the PSO and GA algorithms, as well as other core components of the system, represented in Figures 4.2, 4.3, and 4.4 respectively.

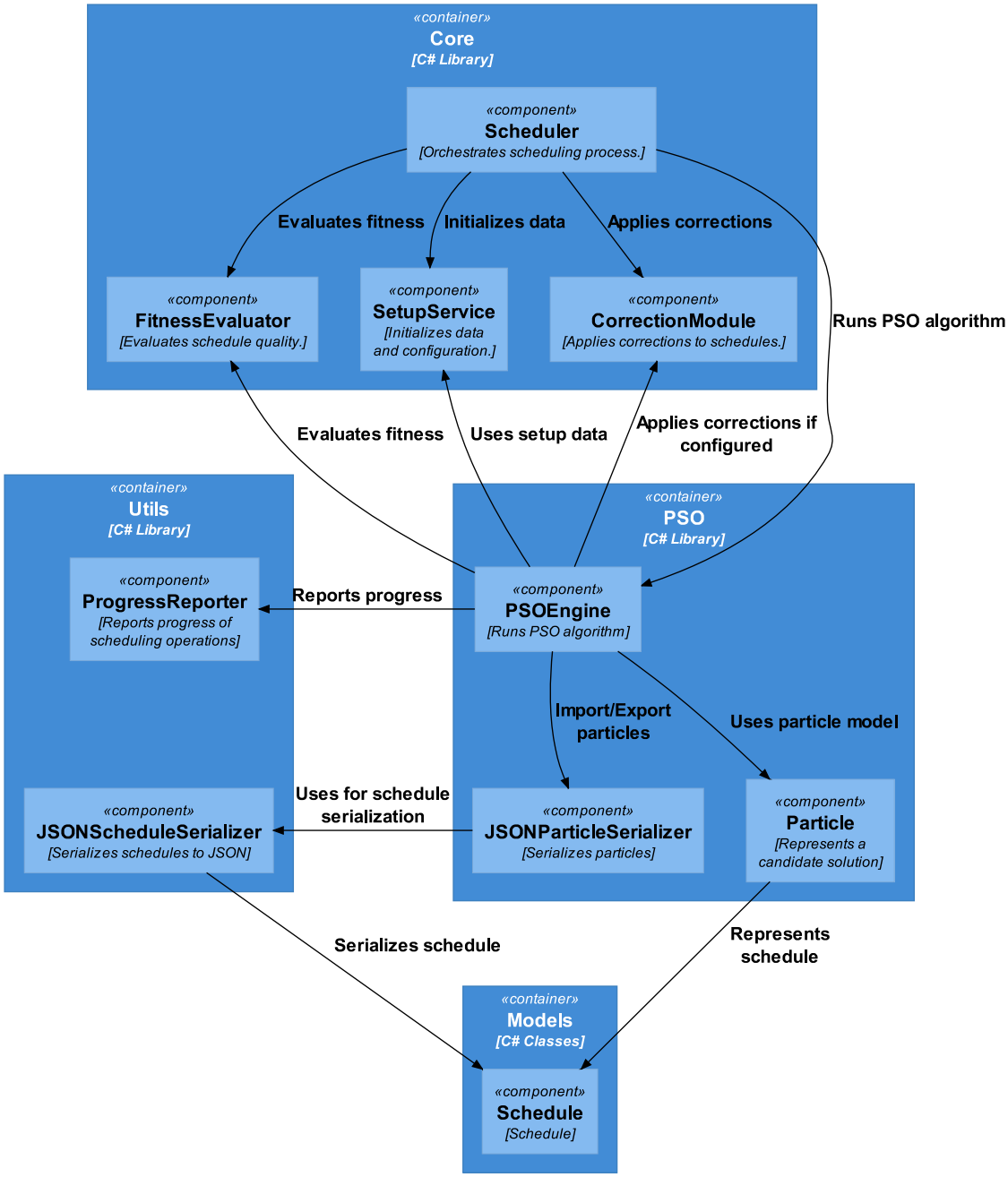

Figure 4.2: C4 Component Diagram: Scheduling Algorithms - Particle Swarm Optimization

The main components for the PSO algorithm include:

- **Core Components**:
  - **Scheduler**: The main component responsible for orchestrating the scheduling process, including invoking the appropriate scheduling algorithms and managing the overall workflow.
  - **SetupService**: Initializes the necessary data structures and resources for the scheduling process.

  - **FitnessEvaluator**: Evaluates the fitness of generated schedules based on predefined criteria.
  - **CorrectionModule**: Applies corrections to generated schedules to resolve conflicts and improve quality.
- **PSO Components**:
  - **PSO Engine**: Implements the Particle Swarm Optimization algorithm for schedule generation.
  - **Particle**: Represents a candidate solution in the PSO algorithm, containing the current and personal best schedule.
  - **JSON Particle Serializer**: Serializes particles to JSON format for storage.
- **Utils Components**:
  - **JSON Schedule Serializer**: Serializes schedules to JSON format for storage.
  - **Progress Reporter**: Reports the progress of the scheduling process, estimates remaining time of execution, and transmits other information, allowing users to monitor its status.
- **Models Components**:
  - **Schedule**: Represents a collection of assignments that make up a complete schedule. Each assignment includes details such as the assigned room, timeslot, teacher, subject, and class.

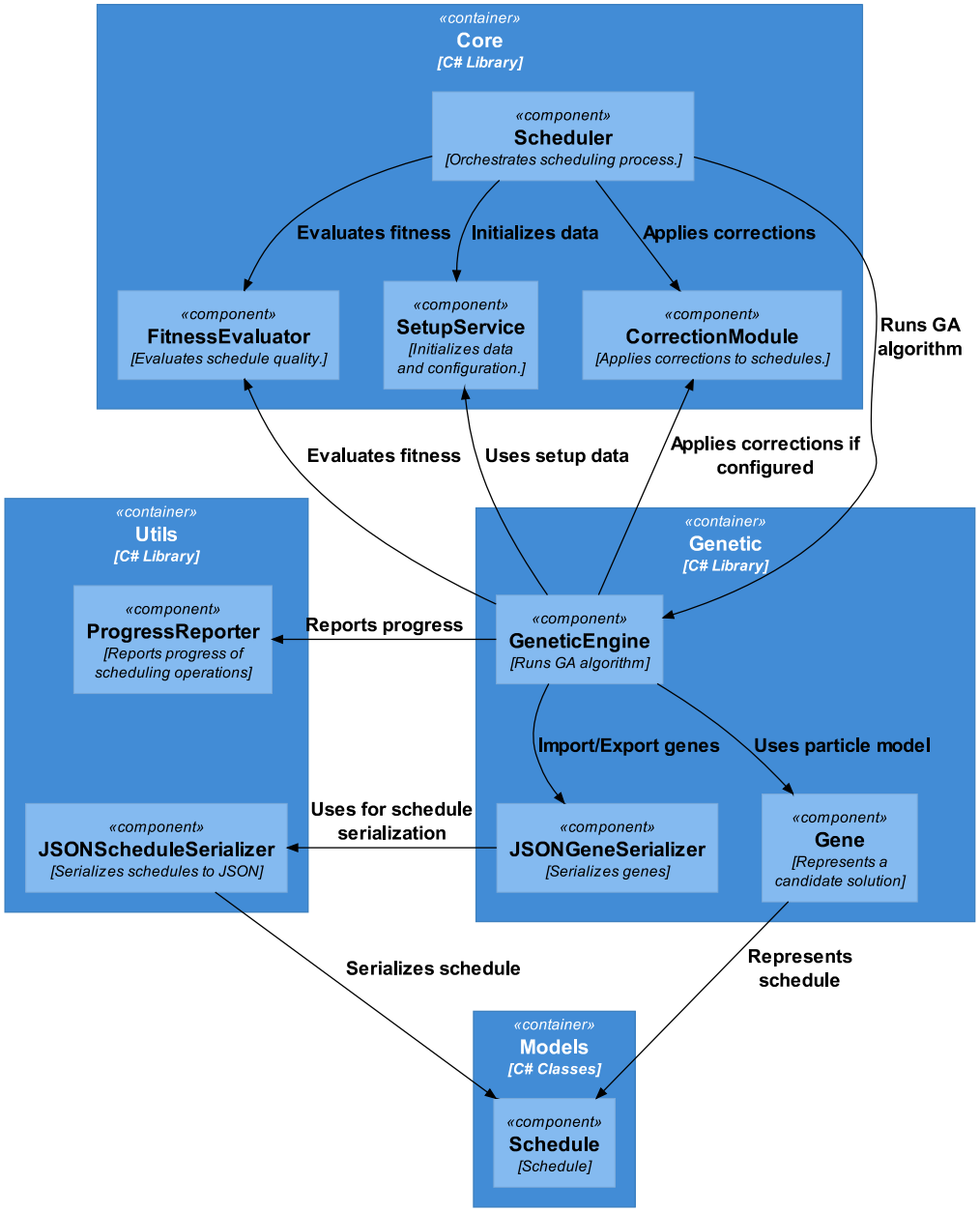

Figure 4.3: C4 Component Diagram: Scheduling Algorithms - Genetic Algorithm

The GA algorithm uses the same core, utils, and models components as the PSO algorithm. By reusing these components, the system maintains consistency, reduces redundancy, and can act as a blueprint for future algorithms to be implemented. The main components specific to the GA algorithm include:

- **GA Components**:
  - **GA Engine**: Implements the Genetic Algorithm for schedule generation.
  - **Gene**: Represents a candidate solution in the Genetic Algorithm.
  - **JSON Gene Serializer**: Serializes genes to JSON format for storage or transmission.

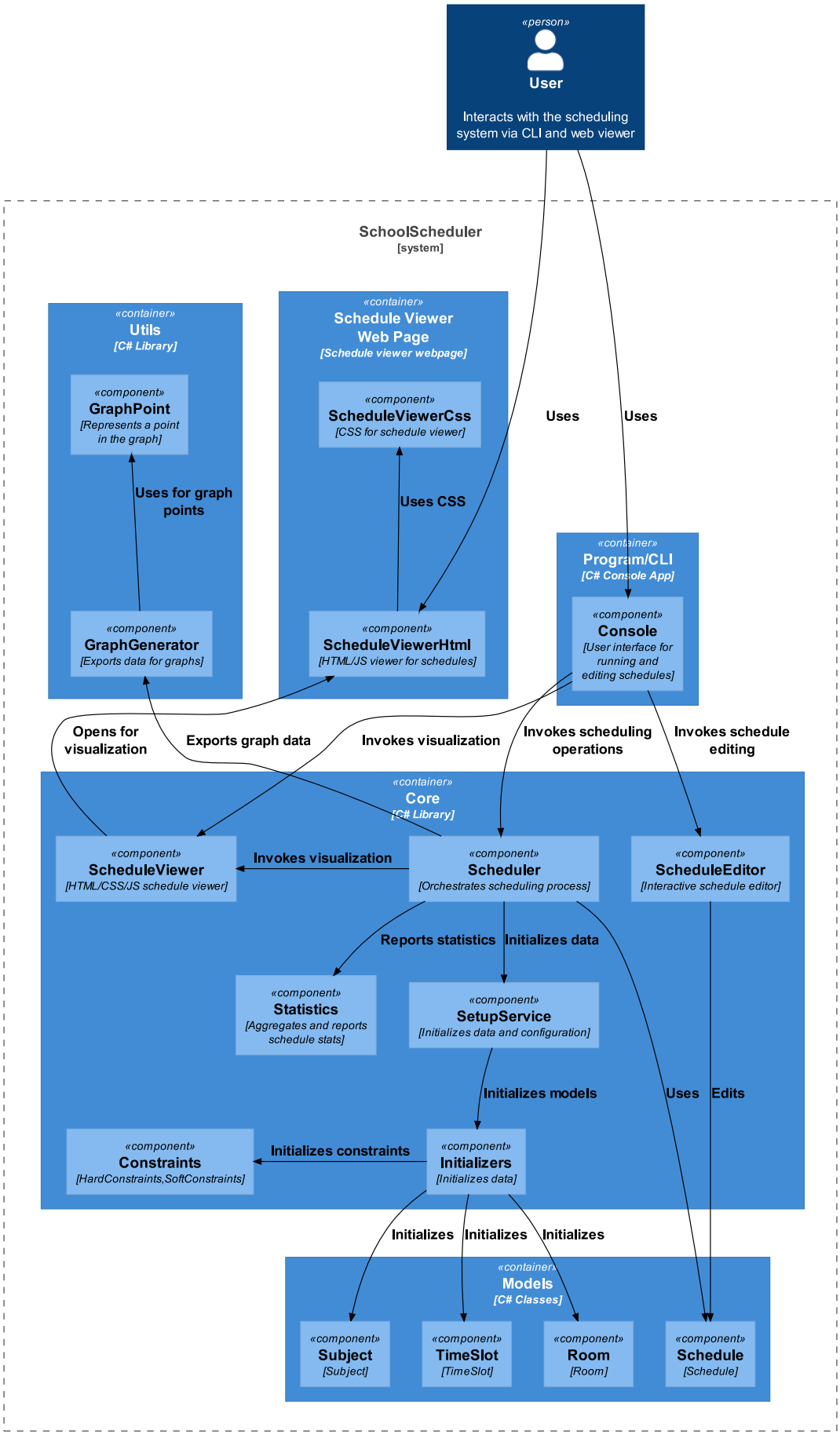

Figure 4.4: C4 Component Diagram: Other Core Components

Other components of the system include:

- **Program/CLI Components**:
  - **Console**: A command-line interface that allows users to interact with the scheduling system through text-based commands.
- **Core Components**:
  - **Schedule Editor**: Provides functionality for manually adjusting generated schedules.
  - **Schedule Viewer**: Provides functionality for visualizing generated schedules.
  - **Statistics**: Reports statistics on the generated schedules.
  - **Constraints**: Defines the rules and constraints that must be satisfied during the scheduling process.
  - **Initializers**: Initializes the domain model entities, such as rooms, subjects, constraints, and timeslots.
- **Schedule Viewer Web Page Components**:
  - **Schedule Viewer HTML and CSS**: A web-based interface to view the generated schedules.
- **Utils Components**:
  - **Graph Generator**: Exports metadata to generate graphs and statistics.
  - **Graph Point**: Represents a point in the graph.

4.4 Summary

This chapter has provided a comprehensive analysis and design of the scheduling system. By establishing a clear project scope and defining detailed requirements, the system architecture was designed to be modular, scalable, and maintainable. The use of design patterns and principles ensures that the system can evolve over time to meet changing requirements and incorporate new scheduling algorithms as needed. The detailed C4 diagrams provide a clear understanding of the system's structure and interactions, guiding the development process effectively, and setting the stage for the implementation phase, which will be covered in the next chapter.

Chapter 5

Implementation

This chapter presents a comprehensive overview of the implementation of the UCSP solution. The following subsections describe the technologies and programming languages employed, the organization of the codebase, and the development environment setup. Furthermore, the chapter details the main components of the system, including the models, core modules, utility functions, and the algorithms utilized for scheduling. Each component is discussed to provide insight into its role and functionality within the overall solution.

5.1 Technologies Used

The implementation of the UCSP solution was carried out using C# within the .NET framework. The choice of .NET and C# was driven by their robust features, performance compared to the counterparts (e.g., C++, Python), extensive libraries, and familiarity, due to prior experience with these technologies, which facilitated efficient development and implementation of complex algorithms. A simple web page for visualizing the schedules was developed using HTML, CSS, and JavaScript, importing the schedules saved in JSON files generated by the main application.

5.2 Implementation Details

This section provides an in-depth look at the implementation details of the UCSP solution, covering the structure of the codebase, the main components, and their functionalities.

5.2.1 Models

Each model was implemented following the design from the previous chapter, designed to encapsulate the properties and behaviors relevant to its role in the system. Figure 5.1 illustrates the class diagram for the models used in the system, highlighting their relationships and key attributes.

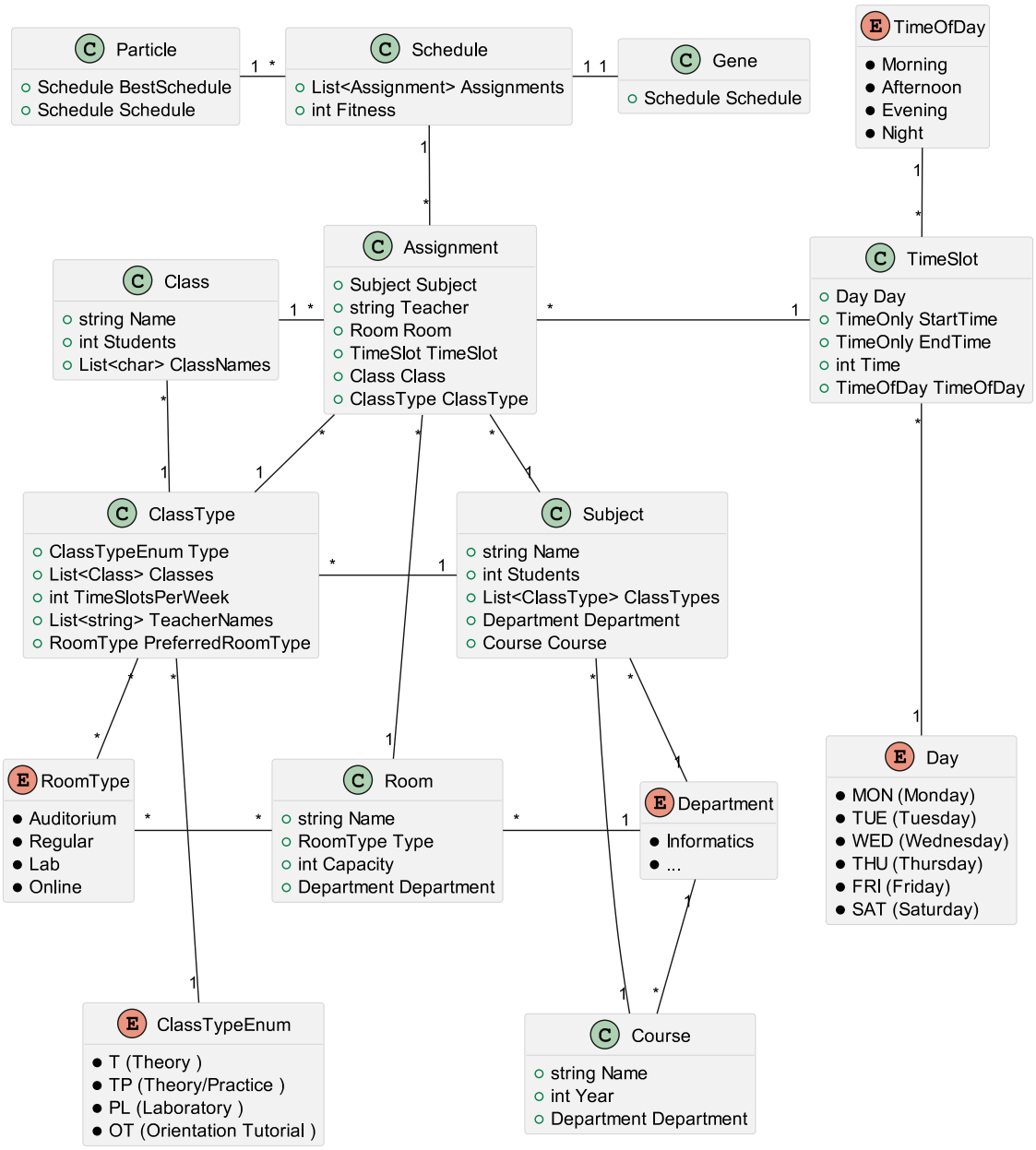

Figure 5.1: Class Diagram

Schedule

Schedule class, Listing 5.1, represents a complete schedule, containing a list of assignments and fitness. It provides a constructor to initialize the list of assignments (lines 16-47), creating assignments based on the provided subjects and randomly assigning the timeslot, room, and teacher from class type list of teachers. It creates assignments equal to the number of timeslots per week defined in the class type, creating for each class of each class type of each subject. The class also includes methods for cloning the schedule (lines 49-52), preventing unintended modifications when updating the original and clone schedules. Finally, there is a method to update the fitness of the schedule using a provided fitness evaluator (lines 55-58).

```
using SchoolScheduler.Core;

namespace SchoolScheduler.Models
{
    public class Schedule
    {
        public List<Assignment> Assignments { get; set; }
        public int Fitness { get; set; }

        public Schedule(List<Assignment> assignments, int fitness)
        {
            Assignments = [.. assignments];
            Fitness = fitness;
        }

        public Schedule(List<Subject> subjects, List<Room> rooms, List<TimeSlot> slots, Random rand)
        {
            Assignments = new List<Assignment>();

            foreach (var subject in subjects)
            {
                foreach (var classType in subject.ClassTypes)
                {
                    foreach (var className in classType.Classes)
                    {
                        for (int h = 0; h < classType.TimeSlotsPerWeek; h++)
                        {
                            var teacher = classType.TeacherNames[rand.Next(classType.TeacherNames.Count)];
                            var room = rooms[rand.Next(rooms.Count)];
                            var slot = slots[rand.Next(slots.Count)];

                            Assignments.Add(new Assignment
                            {
                                Subject = subject,
                                Teacher = teacher,
                                Room = room,
                                TimeSlot = slot,
                                Class = className,
                                ClassType = classType
                            });
                        }
                    }
                }
            }

            Fitness = int.MaxValue;
        }

        public Schedule Clone()
        {
            return new Schedule(Assignments.Select(a => a.Clone()).ToList(), Fitness);
        }


        public void Evaluate(FitnessEvaluator evaluator)
        {
            Fitness = evaluator.EvaluateWithWeights(Assignments);
        }
    }
}
```

Listing 5.1: Schedule.cs

Assignment

Assignment class represents a single assignment in the schedule, linking a subject, class type, class, teacher, room, and timeslot, as it is shown in Listing 5.2. Similar to the Schedule class, it includes a clone method to create a copy of the assignment (lines 12-30).

```
1  namespace SchoolScheduler.Models
2  {
3      public class Assignment
4      {
5          public Subject Subject { get; set; }
6          public string Teacher { get; set; }
7          public Room Room { get; set; }
8          public TimeSlot TimeSlot { get; set; }
9          public Class Class { get; set; }
10         public ClassType ClassType { get; set; }
11
12         public Assignment Clone()
13         {
14             return new Assignment
15             {
16                 Subject = Subject,
17                 Teacher = Teacher,
18                 Room = new Room(Room.Name, Room.Type, Room.Capacity, Room.Department),
19                 TimeSlot = new TimeSlot(TimeSlot.Day, TimeSlot.StartTime, TimeSlot.EndTime),
20                 Class = new Class { Name = Class.Name, Students = Class.Students, ClassNames = new List
       <char>(Class.ClassNames) },
21                 ClassType = new ClassType
22                 {
23                     Type = ClassType.Type,
24                     Classes = ClassType.Classes.Select(c => new Class { Name = c.Name, Students = c.
       Students, ClassNames = new List<char>(c.ClassNames) }).ToList(),
25                     TimeSlotsPerWeek = ClassType.TimeSlotsPerWeek,
26                     TeacherNames = new List<string>(ClassType.TeacherNames),
27                     PreferredRoomType = ClassType.PreferredRoomType
28                 }
29             };
30         }
31     }
32 }
```

Listing 5.2: Assignment.cs

Subject

Listing 5.3 describes Subject class, which contains properties such as the subject name, associated class types, department, and course. It also includes the number of students enrolled in the subject. Equals (lines 11-14) and GetHashCode (lines 16-19) methods are overridden to ensure that subjects are compared based on their names, department, course, and class types, instead of their memory references.

```
1  namespace SchoolScheduler.Models
2  {
3      public class Subject
4      {
5          public string Name { get; set; }
6          public int Students { get; set; }
7          public List<ClassType> ClassTypes { get; set; }
8          public Department Department { get; set; }
9          public Course Course { get; set; }
10
11         public override bool Equals(object? obj)
```

```
12        {
13            // Comparison based on Name, Department, Course, and ClassTypes
14        }
15
16        public override int GetHashCode()
17        {
18            // Hashing function based on Name, Department, Course, and ClassTypes
19        }
20    }
21 }
```

Listing 5.3: Subject.cs

TimeSlot

TimeSlot class, Listing 5.4, represents a specific timeslot in the schedule, defined by the day of the week, start and end times. It also includes attributes to indicate time of the day (morning, afternoon, evening, and night) and timeslot duration in minutes, calculated from the start time (lines 18-21). These attributes are useful for implementing certain constraints and correction functions. Similar to Subject, the Equals (lines 24-27) and GetHashCode (lines 29-32) methods are overridden to ensure that timeslots are compared based on their day, start and end time, rather than their memory references. TimeSlot also implements IComparable interface (lines 34-37) to allow sorting of timeslots based on day and start time.

```
1  namespace SchoolScheduler.Models
2  {
3      public class TimeSlot : IComparable<TimeSlot>
4      {
5          public Day Day { get; set; }
6          public TimeOnly StartTime { get; set; }
7          public TimeOnly EndTime { get; set; }
8          public int Time { get; set; } // in minutes
9          public TimeOfDay TimeOfDay{ get; set; }
10
11         public TimeSlot(Day day, TimeOnly startTime, TimeOnly endTime)
12         {
13             Day = day;
14             StartTime = startTime;
15             EndTime = endTime;
16             Time = (int)(endTime - startTime).TotalMinutes;
17
18             if (startTime.Hour < 13) TimeOfDay = TimeOfDay.MORNING;
19             else if (startTime.Hour < 18) TimeOfDay = TimeOfDay.AFTERNOON;
20             else if (startTime.Hour < 21) TimeOfDay = TimeOfDay.EVENING;
21             else TimeOfDay = TimeOfDay.NIGHT;
22         }
23
24         public override bool Equals(object? obj)
25         {
26             // Comparison based on Day, StartTime, and EndTime
27         }
28
29         public override int GetHashCode()
30         {
31             // Hashing function based on Day, StartTime, and EndTime
32         }
33
34         public int CompareTo(TimeSlot? other)
35         {
36             // Comparison based on Day and StartTime
```

```
37        }
38    }
39
40    public enum Day
41    {
42        SUN = 0, MON = 1, TUE = 2, WED = 3, THU = 4, FRI = 5, SAT = 6
43    }
44
45    public enum TimeOfDay
46    {
47        MORNING = 0, AFTERNOON = 1, EVENING = 2, NIGHT = 3
48    }
49 }
```

Listing 5.4: TimeSlot.cs

Room

Room class represents a physical room where classes can be held. Listing 5.5 is an excerpt of the class implementation, that includes properties such as room name, capacity, department, and type (e.g., auditorium, laboratory). The room type (lines 21-25) is a flag enum, allowing for multiple types to be assigned as preferred room types for a class type. It also overrides Equals (lines 10-13) and GetHashCode (lines 15-18) methods to ensure that rooms are compared based on their names and department.

```
1  namespace SchoolScheduler.Models
2  {
3      public class Room
4      {
5          public string Name { get; set; }
6          public RoomType Type { get; set; }
7          public int Capacity { get; set; }
8          public Department Department { get; set; }
9
10         public override bool Equals(object? obj)
11         {
12             // Comparison based on Name and Department
13         }
14
15         public override int GetHashCode()
16         {
17             // Hashing function based on Name and Department
18         }
19     }
20
21     [Flags]
22     public enum RoomType
23     {
24         AUDITORIUM = 1 << 0, REGULAR = 1 << 1, LAB = 1 << 2, ONLINE = 1 << 3
25     }
26 }
```

Listing 5.5: Room.cs

Other Models

Other models implemented follow the same pattern as the ones described above, using properties relevant to their roles, represented in Figure 5.1, and overriding Equals and GetHashCode methods for proper comparison.

5.2.2 Core

The core module serves as the backbone of the UCSP solution, orchestrating the interaction between various components and managing the overall scheduling process. It encompasses several key classes and interfaces that facilitate the setup, evaluation, and correction of schedules. The core module is designed to be modular and extensible, allowing for easy integration of new features and algorithms in the future.

SetupService

The SetupService class, Listing 5.6, is responsible for initializing the scheduling environment. It currently loads static data (lines 14-20) for subjects, class types and classs, rooms, timeslots, and constraints. It can later be extended to load data from external sources, such as databases or files. The class acts as a single point of data source, ensuring that all necessary information is available for the scheduling process.

```
1  using SchoolScheduler.Core.Constraints;
2  using SchoolScheduler.Core.Initializers;
3  using SchoolScheduler.Models;
4
5  namespace SchoolScheduler.Core
6  {
7      public class SetupService
8      {
9          public List<Subject> Subjects { get; private set; }
10         public List<Room> Rooms { get; private set; }
11         public List<TimeSlot> TimeSlots { get; private set; }
12         public List<Constraint> Constraints { get; private set; }
13
14         public void Initialize()
15         {
16             Subjects = new SubjectInitializer().Initialize();
17             Rooms = new RoomInitializer().Initialize();
18             TimeSlots = new TimeSlotInitializer().Initialize();
19             Constraints = new ConstraintInitializer().Initialize();
20         }
21     }
22 }
```

Listing 5.6: SetupService.cs

Constraints

Listing 5.7 shows the implementation of the Constraint class (lines 5-9), that serves as a base class for all constraints, providing a common interface and shared functionality. It defines a function that can be later executed to retrieve the violated constraints by a schedule. By being separate constraint implementations, it allows for easy extension and configuration on the constraints to be applied. It also has a ConstraintName (lines 11-43), that includes a description and weight, which are used to identify and prioritize constraints during the evaluation process. Weights already defined range from 1 to 10, with 10 being the most important constraints, but can be adjusted as needed.

```
1  using System.ComponentModel;
2
3  namespace SchoolScheduler.Core.Constraints
4  {
5      public class Constraint
```

```
6     {
7         public ConstraintName Name { get; set; }
8         public Delegate Func { get; set; }
9     }
10
11    public enum ConstraintName
12    {
13        // Hard Constraints
14
15        [Description("Teacher Conflict")]
16        [Weight(9)]
17        TeacherConflict,
18
19        [Description("Room Capacity Exceeded")]
20        [Weight(9)]
21        RoomCapacityExceeded,
22
23        [Description("Teachers Max 8 Hours Day")]
24        [Weight(8)]
25        TeachersMax8HoursDay,
26
27        // ...
28
29        // Soft Constraints
30
31        [Description("Preferred Room Type Constraint")]
32        [Weight(3)]
33        PreferredRoomTypeConstraint,
34
35        [Description("Continuous Class Gaps")]
36        [Weight(5)]
37        ContinuousClassGaps,
38        [Description("Students Conflict")]
39        [Weight(7)]
40        StudentsConflict,
41
42        // ...
43    }
44
45    // Attribute to define weight for constraints
46 }
```

Listing 5.7: Constraint.cs

Two main types of constraints are implemented separately: Hard Constraints and Soft Constraints, Listings 5.8 and 5.9 respectively. Despite hard constraints representing non-negotiable rules that must be strictly followed, the implementation allows for some flexibility by applying a high penalty when violated, rather than rejecting the schedule. This approach enables the algorithms to explore a wider solution space, potentially leading to better overall schedules.

Hard constraints implemented:

- **AssignmentLoadViolation:** Penalises schedules where the number of assignments for a subject does not match the required load defined in the class type. Weight: 9.
- **TeacherConflict** (lines 24-28)**:** Penalises each instance where a teacher is assigned to more than one class at the same time. Weight: 9.
- **StudentsLunchBreak:** Penalises when students do not have a lunch break between 12:00 and 14:00, if they have assignments in the morning and afternoon periods. Weight: 7.

- **StudentsDinnerBreak:** Penalises when students do not have a dinner break between 18:00 and 19:00, if they have assignments in the evening and night periods. Weight: 7.
- **TeachersLunchBreak:** Penalises when teachers do not have a lunch break between 12:00 and 14:00, if they have assignments in the morning and afternoon periods. Weight: 6.
- **TeachersDinnerBreak:** Penalises when teachers do not have a dinner break between 18:00 and 19:00, if they have assignments in the evening and night periods. Weight: 6.
- **TeachersMax8HoursDay** (lines 34-44)**:** Penalises when teachers have more than 8 hours of assignments in a single day. Weight: 8.
- **StudentsMax8HoursDay:** Penalises when students have more than 8 hours of assignments in a single day. Weight: 8.
- **RoomCapacityExceeded** (lines 30-32)**:** Penalises when the number of students assigned to a room exceeds its capacity. Weight: 9.
- **RoomConflict:** Penalises when two assignments overlap in the same room. Weight: 9.

```
1  using SchoolScheduler.Models;
2  using SchoolScheduler.Utils;
3
4  namespace SchoolScheduler.Core.Constraints
5  {
6      public class HardConstraints
7      {
8          public static List<Constraint> GetNamedConstraints()
9          {
10             return new List<Constraint>
11             {
12                 new Constraint { Name = ConstraintName.TeacherConflict, Func = TeacherConflict },
13                 new Constraint { Name = ConstraintName.RoomCapacityExceeded, Func =
       RoomCapacityExceeded },
14                 new Constraint { Name = ConstraintName.TeachersMax8HoursDay, Func =
       TeachersMax8HoursDay },
15                 // ...
16             };
17         }
18
19         public static List<Delegate> GetConstraints()
20         {
21             return GetNamedConstraints().Select(c => c.Func).ToList();
22         }
23
24         // Teacher cannot be in two places at once
25         public static readonly Func<List<Assignment>, int> TeacherConflict = assignments =>
26             assignments
27                 .GroupBy(a => new { a.TimeSlot, a.Teacher })
28                 .Sum(g => g.Count() > 1 ? g.Count() - 1 : 0);
29
30         // Room capacity exceeded
31         public static readonly Func<List<Assignment>, int> RoomCapacityExceeded = assignments =>
32             assignments.Count(a => a.Class.Students > a.Room.Capacity);
33
34         // No more than 8 hours a day for teachers
35         public static readonly Func<List<Assignment>, int> TeachersMax8HoursDay = assignments =>
36         {
```

```
37            int violations = 0;
38
39            var groups = assignments.GroupBy(a => new { a.Teacher, a.TimeSlot.Day });
40
41            foreach (var group in groups) if (group.Sum(a => a.TimeSlot.Time) / 60 > 8) violations++;
42
43            return violations;
44        };
45    }
46
47}
```

Listing 5.8: HardConstraints.cs

Soft constraints implemented:

- **StudentGaps:** Penalises when students have gaps between classes where classes could be scheduled. It ignores meal time breaks. Weight: 1.
- **IncorrectOrderOfClasses:** Penalises when the order of classes does not follow the T -> TP -> PL order. Weight: 2.
- **OutOfDepartmentRoomUsage:** Penalises room assignments to departments other than the department assigned to the subject. Weight: 2.
- **PreferredRoomTypeConstraint** (lines 25-37)**:** Penalises room assignments to room types other than the preferred room type assigned in the class type. Weight: 3.
- **StudentsMin3HoursDay:** Penalises when students have less than 3 hours of assignments in a single day. Weight: 2.
- **ContinuousClassGaps** (lines 39-60)**:** Penalises when classes are interrupted by a gaps or other classes. Weight: 5.
- **ContinuousClassDifferentTeachers:** Penalises when assignments have a change of teacher in the middle of the class. Weight: 5.
- **ContinuousClassDifferentRooms:** Penalises when assignments have a change of room in the middle of the class. Weight: 5.
- **StudentsConflict** (lines 62-82)**:** Penalises when students have multiple assignments in the same timeslot. Weight: 7.

```
1  using SchoolScheduler.Models;
2  using SchoolScheduler.Utils;
3
4  namespace SchoolScheduler.Core.Constraints
5  {
6      public static class SoftConstraints
7      {
8
9          public static List<Constraint> GetNamedConstraints()
10         {
11             return new List<Constraint>
12             {
13                 new Constraint { Name = ConstraintName.PreferredRoomTypeConstraint, Func =
       PreferredRoomTypeConstraint },
14                 new Constraint { Name = ConstraintName.ContinuousClassGaps, Func =
       ContinuousClassGaps },
15                 new Constraint { Name = ConstraintName.StudentsConflict, Func = StudentsConflict },
16                 // ...
17             };
```

```
18      }
19
20      public static List<Delegate> GetConstraints()
21      {
22          return GetNamedConstraints().Select(c => c.Func).ToList();
23      }
24
25      // Room type preference
26      public static readonly Func<List<Assignment>, int> PreferredRoomTypeConstraint = assignments
        =>
27      {
28          int violations = 0;
29
30          foreach (var assignment in assignments)
31          {
32              if ((assignment.ClassType.PreferredRoomType & assignment.Room.Type) == 0)
33                  violations++;
34          }
35
36          return violations;
37      };
38
39      // Avoid gaps in continuous classes
40      public static readonly Func<List<Assignment>, int> ContinuousClassGaps = assignments =>
41      {
42          int gaps = 0;
43
44          var groups = assignments
45              .GroupBy(a => new { a.Class.Name, a.TimeSlot.Day })
46              .Where(g => g.Count() > 1)
47              .Select(g => g.OrderBy(a => a.TimeSlot).ToList());
48
49          foreach (var group in groups)
50          {
51              var allGaps = ConstraintUtils.GetGaps(group);
52
53              foreach (var gap in allGaps)
54              {
55                  gaps += (int) Math.Floor(gap / TimeSpan.FromMinutes(30));
56              }
57          }
58
59          return gaps;
60      };
61
62      // Classes with multiple assignments in the same time slot
63      public static readonly Func<List<Assignment>, int> StudentsConflict = assignments =>
64      {
65          int violations = 0;
66
67          var groups = assignments
68              .SelectMany(a => a.Class.ClassNames.Select(letter => new
69              {
70                  Key = (a.Subject.Course, Letter: letter, a.TimeSlot),
71                  Assignment = a
72              }))
73              .GroupBy(a => a.Key)
74              .Where(g => g.Count() > 1);
75
76          foreach (var group in groups)
77          {
78              violations += group.Count() - 1;
79          }
80
81          return violations;
82      };
```

```
83     }
84 }
```

Listing 5.9: SoftConstraints.cs

FitnessEvaluator

The Fitness Evaluator, described in Listing 5.10, calculates the fitness of a schedule, based on the constraints defined in Setup Service. The evaluator implements three methods:

- **NameConstraintsViolated** (lines 25-39)**:** Lists the constraint and the number of occurrences each constraint is violated.
- **CountViolatedConstraints** (lines 15-18)**:** Returns the sum of the number of violated constraints.
- **EvaluateWithWeights** (lines 20-23)**:** Returns the fitness value, which is determined by the sum of the violated constraints with the respective weights applied by multiplication.

With this implementation, the lower the fitness value, the better the schedule is evaluated.

```
1  using SchoolScheduler.Core.Constraints;
2  using SchoolScheduler.Models;
3
4  namespace SchoolScheduler.Core
5  {
6      public class FitnessEvaluator
7      {
8          private readonly SetupService _setup;
9
10         public FitnessEvaluator( SetupService setup)
11         {
12             _setup = setup;
13         }
14
15         public int CountViolatedConstraints(List<Assignment> assignments)
16         {
17             return NameConstraintsViolated(assignments).Values.Sum();
18         }
19
20         public int EvaluateWithWeights(List<Assignment> assignments)
21         {
22             return NameConstraintsViolated(assignments).Sum(kv => kv.Key.Name.GetWeight() * kv.Value)
       ;
23         }
24
25         public Dictionary<Constraint, int> NameConstraintsViolated(List<Assignment> assignments)
26         {
27             var result = new Dictionary<Constraint, int>();
28             foreach (var constraint in _setup.Constraints)
29             {
30                 int violations = constraint.Func switch
31                 {
32                     Func<List<Assignment>, int> f1 => f1(assignments),
33                     Func<List<Assignment>, List<Subject>, int> f2 => f2(assignments, _setup.Subjects),
34                     _ => 0
35                 };
36                 result[constraint] = violations;
37             }
38             return result;
39         }
```

```
40     }
41 }
```

Listing 5.10: FitnessEvaluator.cs

Correction Module

The Correction Module is designed to perform LS operations that adjust schedules to reduce or eliminate constraint violations, as it is shown Listing 5.11. It systematically analyzes the schedule for conflicts, such as overlapping assignments, room capacity exceeded, or teacher conflicts, and applies targeted corrections. These corrections use techniques that reassign, swap, and relocate the resources, such as teachers, rooms and timeslots. By iteratively applying these local changes, the module improves schedule overall fitness, serving as a post-processing step or as an integrated part of the algorithms.

The implementation of the Correction Module follows a structure similar to the Constraints module, with each correction having its function and connection to the constraint it attempts to correct. It allows for easy extension and configuration on the corrections executed, adjusting to current needs. Correction functions were implemented to reduce hard constraints, but there can be functions to work on soft constraints.

In order to reduce processing resources, violated constraints are identified beforehand to target the correction module to those issues (line 30).

```
1  using SchoolScheduler.Core.Constraints;
2  using SchoolScheduler.Models;
3  using SchoolScheduler.Utils;
4
5  namespace SchoolScheduler.Core.Correction
6  {
7      public class CorrectionModule
8      {
9          private readonly SetupService _setup;
10         private readonly FitnessEvaluator _evaluator;
11
12         private readonly Dictionary<ConstraintName, Delegate> _correctionMethods;
13
14         public CorrectionModule(SetupService setup, FitnessEvaluator evaluator)
15         {
16             _setup = setup;
17             _evaluator = evaluator;
18
19             _correctionMethods = new Dictionary<ConstraintName, Delegate>
20             {
21                 { ConstraintName.TeacherConflict, TeacherCorrections.FixConflicts },
22                 { ConstraintName.RoomCapacityExceeded, RoomCorrections.FixCapacity },
23                 { ConstraintName.TeachersMax8HoursDay, TeacherCorrections.FixMaxHoursPerDay },
24                 // ...
25             };
26         }
27
28         public List<Assignment> ApplyCorrections(List<Assignment> assignments, bool progressReport =
       false)
29         {
30             var violated = _evaluator.NameConstraintsViolated(assignments);
31             int totalViolations = violated.Sum(v => v.Value);
32             var reporter = new ProgressReporter(totalViolations);
33
34             int current = 0;
35
```

```
36          foreach (var (constraint, count) in violated)
37          {
38              if (_correctionMethods.TryGetValue(constraint.Name, out var correctionMethod))
39              {
40                  // Execute Correction function
41              }
42
43              current += count;
44              if (progressReport)
45                  reporter.Report(current, $"Fixing constraint: {constraint.Name.GetDescription()}");
46          }
47
48          return assignments;
49      }
50  }
51  }
```

Listing 5.11: CorrectionModule.cs

As examples of implementation, the correction module includes functions with direct influence on both teacher and room assignments. For teachers, 5.12, one function resolves scheduling conflicts where a teacher is assigned to multiple classes at the same time (lines 8-55), and another ensures that no teacher exceeds a maximum of 8 hours per day (lines 57-109).

- **TeacherConflict:** to resolve the conflicts, the function tries different alternatives to all the conflicting assignments except the first
  1. Try assigning another available teacher for the same timeslot
  2. Try moving the assignment to another timeslot where teacher and class are both free
  3. Try moving the first assignment to another timeslot where teacher and class are both free if at least one change fails
- **TeacherMax8HoursDay:** starting from the extremities working inwards, the function tries different alternatives until the maximum 8 hours are met or all assignments of the day were attempted to alteration
  1. Try assigning another available teacher for the same timeslot
  2. Try moving the assignment to another timeslot where teacher and class are both free

```
1  using SchoolScheduler.Models;
2  using SchoolScheduler.Utils;
3
4  namespace SchoolScheduler.Core.Correction
5  {
6      public static class TeacherCorrections
7      {
8          public static void FixConflicts(List<Assignment> assignments, List<TimeSlot> allTimeSlots)
9          {
10             var conflicts = assignments
11                 .GroupBy(a => new { a.TimeSlot, a.Teacher })
12                 .Where(g => g.Count() > 1);
13
14             foreach (var group in conflicts)
15             {
16                 var excess = group.Skip(1).ToList();
```

```
17              bool slotNotFound = false;
18
19              foreach (var assignment in excess)
20              {
21                  // Try assigning another available teacher for the same time slot
22                  var availableTeacher = assignment.ClassType.TeacherNames
23                      .FirstOrDefault(t =>
24                          t != assignment.Teacher &&
25                          !assignments.Any(a =>
26                              a.Teacher == t &&
27                              a.TimeSlot == assignment.TimeSlot));
28
29                  if (availableTeacher != null)
30                  {
31                      assignment.Teacher = availableTeacher;
32                      continue;
33                  }
34
35                  // Try moving the assignment to another time slot where teacher and class are both free
36                  var newSlot = CommonCorrections.FindNextAvailableSlot(assignment, assignments,
    allTimeSlots);
37                  if (newSlot != null)
38                  {
39                      assignment.TimeSlot = newSlot;
40                  }
41                  else
42                      slotNotFound = true;
43              }
44
45              // Try moving the first assignment to another time slot where teacher and class are both free
    if at least one change fails
46              if (slotNotFound)
47              {
48                  var newSlot = CommonCorrections.FindNextAvailableSlot(group.First(), assignments,
    allTimeSlots);
49                  if (newSlot != null)
50                  {
51                      group.First().TimeSlot = newSlot;
52                  }
53              }
54          }
55      }
56
57      public static void FixMaxHoursPerDay(List<Assignment> assignments, List<TimeSlot> allTimeSlots
    )
58      {
59          var groups = assignments.GroupBy(a => new { a.Teacher, a.TimeSlot.Day })
60              .Select(g => g.ToList()).ToList();
61
62          foreach (var group in groups)
63          {
64              if (group.Sum(a => a.TimeSlot.Time) / 60 <= 8)
65                  continue;
66
67              var orderedGroup = group.OrderBy(a => a.TimeSlot).ToList();
68
69              int left = 0;
70              int right = orderedGroup.Count - 1;
71
72              while (orderedGroup.Sum(a => a.TimeSlot.Time) / 60 > 8 && left <= right)
73              {
74                  var candidates = new[] { orderedGroup[left], orderedGroup[right] };
75
76                  foreach (var assignment in candidates)
77                  {
78                      // Skip if already under max
```

```
79                      if (group.Sum(a => a.TimeSlot.Time) / 60 <= 8)
80                          break;
81
82                      // Try to find another teacher for the same slot
83                      var replacementTeacher = assignment.ClassType.TeacherNames
84                          .FirstOrDefault(t =>
85                              t != assignment.Teacher &&
86                              !assignments.Any(a =>
87                                  a.Teacher == t &&
88                                  a.TimeSlot == assignment.TimeSlot));
89
90                      if (replacementTeacher != null)
91                      {
92                          assignment.Teacher = replacementTeacher;
93                      }
94                      else
95                      {
96                          // Try moving to a new time slot
97                          var newSlot = CommonCorrections.FindNextAvailableSlot(assignment, assignments,
      allTimeSlots);
98                          if (newSlot != null)
99                          {
100                             assignment.TimeSlot = newSlot;
101                         }
102                     }
103                 }
104
105                 left++;
106                 right--;
107             }
108         }
109     }
110
111 }
112 }
```

Listing 5.12: TeacherCorrections.cs

For rooms, Listing 5.13, a correction function addresses cases where the assigned room's capacity is exceeded by the class size (lines 7-14), reassigning rooms to meet capacity and preference constraints.

- **RoomCapacityExceeded:** to resolve the conflicts, a function to improve room assignment (lines 16-131) is executed on the assignments that do not fulfill the capacity requirements, trying different alternatives to reassign them.

    1. Try assigning another room that is available, has enough capacity, follows the room type preference and is in the same department as the subject (lines 32-44)
    2. Try assigning another room that is available, has enough capacity, and follows the room type preference or is in the same department as the subject (lines 46-57)
    3. Try swapping with another room that has enough capacity, follows the room type preference and is in the same department as the subject for both assignments (lines 59-81)
    4. Try swapping with another room that has enough capacity, and follows the room type preference or and is in the same department as the subject for both assignments (lines 83-96)
    5. If capacity not met, try to find a room with increased capacity (lines 98-116)

6. If a change is forced, to resolve conflits, assign any available room (lines 118-130)

```
1  using SchoolScheduler.Models;
2
3  namespace SchoolScheduler.Core.Correction
4  {
5      public static class RoomCorrections
6      {
7          public static void FixCapacity(List<Assignment> assignments, List<Room> rooms)
8          {
9              var assignmentsWithCapacityIssues = assignments
10                 .Where(a => a.Class.Students > a.Room.Capacity)
11                 .ToList();
12             foreach (var assignment in assignmentsWithCapacityIssues)
13                 ImproveRoomAssignment(assignment, assignments, rooms);
14         }
15
16         public static void ImproveRoomAssignment(Assignment assignment, List<Assignment> assignments,
       List<Room> rooms, bool forceChange = false)
17         {
18             var timeSlot = assignment.TimeSlot;
19             var classSize = assignment.Class.Students;
20             var requiredType = assignment.ClassType.PreferredRoomType;
21             var preferredDept = assignment.Subject.Department;
22
23             var usedRooms = assignments
24                 .Where(a => a.TimeSlot.Equals(timeSlot))
25                 .Select(a => a.Room)
26                 .ToHashSet();
27
28             var availableRooms = rooms
29                 .Where(r => !usedRooms.Contains(r))
30                 .ToList();
31
32             // STEP 1: Available + Enough Capacity + RoomType + Dept
33             var preferredRooms = availableRooms
34                 .Where(r => r.Capacity >= classSize &&
35                             r.Department == preferredDept &&
36                             r.Type.HasFlag(requiredType))
37                 .OrderBy(r => r.Capacity)
38                 .ToList();
39
40             if (preferredRooms.Any())
41             {
42                 assignment.Room = preferredRooms.First();
43                 return;
44             }
45
46             // STEP 2: Available + Enough Capacity + (RoomType || Dept)
47             var fallbackRooms = availableRooms
48                 .Where(r => r.Capacity >= classSize &&
49                         (r.Department == preferredDept || r.Type.HasFlag(requiredType)))
50                 .OrderBy(r => r.Capacity)
51                 .ToList();
52
53             if (fallbackRooms.Any())
54             {
55                 assignment.Room = fallbackRooms.First();
56                 return;
57             }
58
59             // STEP 3: Swap with other assignments – Enough Capacity + RoomType + Dept
60             var sameTimeAssignments = assignments.Where(a =>
61                     a.TimeSlot.Equals(timeSlot) &&
62                     a.Class.Students <= assignment.Room.Capacity &&
63                     (assignment.Room.Type.HasFlag(a.ClassType.PreferredRoomType) ||
```

```
            a.Room.Department != assignment.Subject.Department))
        .ToList();


    var preferredRoomsToSwap = sameTimeAssignments
        .Where(r => r.Room.Capacity >= classSize &&
                    r.Room.Department == preferredDept &&
                    r.Room.Type.HasFlag(requiredType))
        .OrderBy(r => r.Room.Capacity)
        .ToList();

    if (preferredRoomsToSwap.Any())
    {
        var temp = assignment.Room;
        assignment.Room = preferredRoomsToSwap.First().Room;
        preferredRoomsToSwap.First().Room = temp;
        return;
    }

    // STEP 4: Swap with other assignments – Enough Capacity + (RoomType || Dept)
    var fallbackRoomsToSwap = sameTimeAssignments
        .Where(r => r.Room.Capacity >= classSize &&
               (r.Room.Department == preferredDept || r.Room.Type.HasFlag(requiredType)))
        .OrderBy(r => r.Room.Capacity)
        .ToList();

    if (fallbackRoomsToSwap.Any())
    {
        var temp = assignment.Room;
        assignment.Room = fallbackRoomsToSwap.First().Room;
        fallbackRoomsToSwap.First().Room = temp;
        return;
    }

    // STEP 5: If capacity not met, try to find a room with increased capacity
    if (assignment.Room.Capacity < classSize)
    {
        var betterOrFallbackRooms = availableRooms
            .Where(r => r.Capacity > assignment.Room.Capacity)
            .OrderBy(r => r.Capacity)
            .Concat(
                availableRooms
                .Where(r => r.Capacity < classSize)
                .OrderByDescending(r => r.Capacity)
            )
            .ToList();

        if (betterOrFallbackRooms.Any())
        {
            assignment.Room = betterOrFallbackRooms.First();
            return;
        }
    }

    // STEP 6: If forceChange is true, to resolve conflits, assign any available room
    if (forceChange)
    {
        var anyAvailableRoom = availableRooms
            .OrderByDescending(r => r.Capacity)
            .FirstOrDefault();

        if (anyAvailableRoom != null)
        {
            assignment.Room = anyAvailableRoom;
            return;
        }
```

```
130             }
131         }
132     }
133 }
```

Listing 5.13: RoomCorrections.cs

Other correction functions implemented:

- **TeacherCorrections:**
  - **TeachersLunchBreak:** If the minimum lunch break is not met, whenever the teacher has morning and afternoon classes, tries to move assignments during lunch times to other timeslots.
  - **TeachersDinnerBreak:** If the minimum dinner break is not met, whenever the teacher has evening and night classes, tries to move assignments during dinner times to other timeslots.
- **RoomCorrections:**
  - **RoomConflict:** Executes the function to improve room assignment on all conflicting assignments except the first.
  - **OutOfDepartmentRoomUsage:** Executes the function to improve room assignment on all conflicting assignments.
- **StudentCorrections:**
  - **StudentsLunchBreak:** If the minimum lunch break is not met, whenever the students have morning and afternoon classes, tries to move assignments during lunch times to other timeslots.
  - **StudentsDinnerBreak:** If the minimum dinner break is not met, whenever the students have evening and night classes, tries to move assignments during dinner times to other timeslots.
  - **StudentsMax8HoursDay:** Similar procedure as TeacherMax8HoursDay.
- **AssignmentCorrections:**
  - **AssignmentLoadViolation:** Ensures that all curriculum is defined in the schedule, by removing extra assignments, adding missing ones, and, whenever possible, reusing the teacher, room, and/or timeslot from the removed extra slots.

Schedule Viewer

Schedule viewer is a simple web page to display the schedules, as a console print out would be too large and unintelligible. It shows the schedules through different perpectives, grouping assignments by room, class, subject and teacher, and provides a few filters, such as department, course, day, hour, depending on the selected view. Figure 5.2 shows the Schedule Viewer with assignments grouped by class.

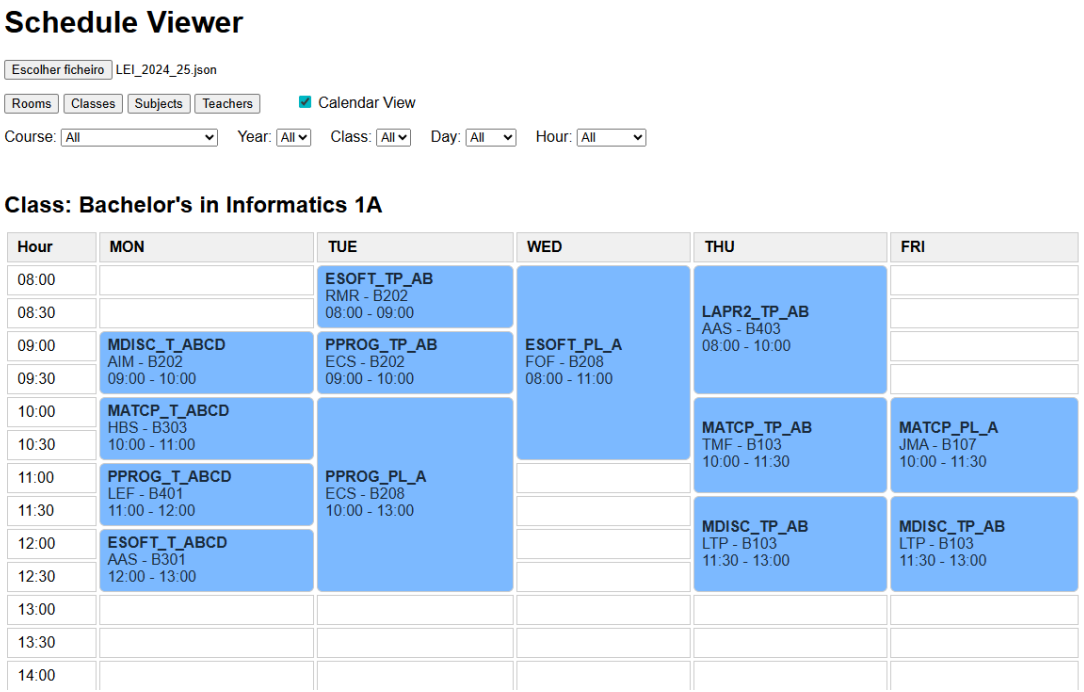

Figure 5.2: Schedule Viewer

The web page also provides a functionality to highlight conflicts, helping the user identify problems in the schedule that can be later resolved/minimized, as it is shown in Figure 5.3.

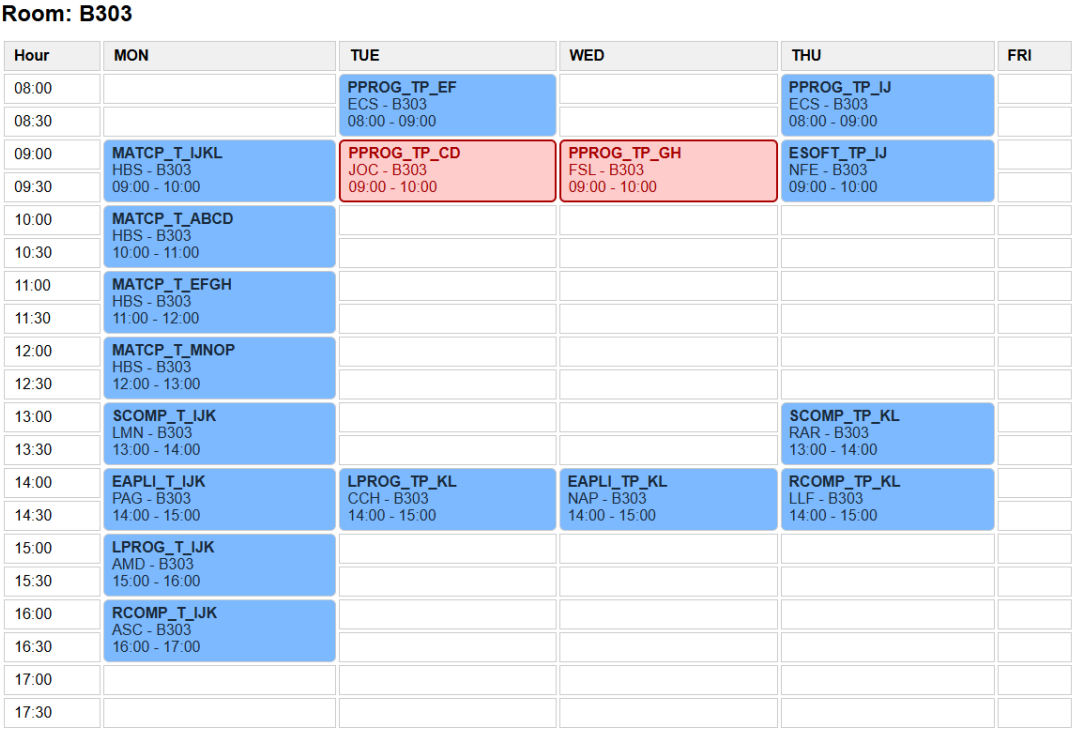

Figure 5.3: Schedule Viewer with highlighted conflicts

Scheduler

The Scheduler module is responsible for orchestrating the configuration and execution of the scheduling algorithms. It provides an interactive interface that guides the user through the necessary steps to set up the scheduling process, including selecting the algorithm (PSO or Genetic), configuring parameters such as population size and number of iterations/generations, and initializing the scheduling environment with the required data.

Once the configuration is complete, the Scheduler invokes the chosen algorithm, monitors its progress, and displays real-time feedback using the progress reporter utility. Upon completion, it presents the resulting schedule, along with relevant statistics such as fitness values, constraint violations, and other performance metrics. The Scheduler also offers options to export the schedule to JSON format and visualize it using the Schedule Viewer.

This modular approach ensures that users can easily experiment with different configurations and algorithms, facilitating comparative analysis and optimization of the scheduling solution.

5.2.3 PSO Algorithm

The PSO algorithm developed for university course scheduling is designed to efficiently search for high-quality schedules by simulating a swarm of candidate solutions, called particles. Each particle represents a possible schedule, and the swarm collectively explores the solution space, guided by both individual and global best solutions.

Particle Representation

Listing 5.14 shows the implementation of the Particle class.

```
using SchoolScheduler.Models;

namespace SchoolScheduler.PSO
{
    public class Particle
    {
        public Schedule Schedule;
        public Schedule BestSchedule;

        public Particle()
        {}

        public Particle(List<Subject> subjects, List<Room> rooms, List<TimeSlot> slots, Random rand)
        {
            Schedule = new Schedule(subjects, rooms, slots, rand);
            BestSchedule = Schedule.Clone();
        }
    }
}
```

Listing 5.14: Particle.cs

- Each particle encapsulates a Schedule object, which contains all assignments for subjects, rooms, teachers, and timeslots.
- Particles also maintain a BestSchedule, representing a clone of the best solution found by that particle so far.

Engine

The PSOEngine class, Listing 5.15, manages the execution of the PSO algorithm.

```
using SchoolScheduler.Models;
using SchoolScheduler.Utils;
using SchoolScheduler.Core;
using SchoolScheduler.Core.Correction;

namespace SchoolScheduler.PSO
{
    public class PSOEngine
    {
        private readonly SetupService _setup;
        private readonly FitnessEvaluator _evaluator;
        private readonly CorrectionModule _correctionModule;
        private readonly Random _random = new();

        public Schedule Execute(out List<GraphPoint> stats, int swarmSize = 30, int iterations = 100, List<Particle> initialSwarm = null, bool useCorrectionModule = true, bool progressReporting = true)
        {
            stats = new List<GraphPoint>();
            var swarm = new List<Particle>();

            initialSwarm ??= new List<Particle>();
            swarm.AddRange(initialSwarm);

            Schedule globalBest = null!;

            for (int i = swarm.Count; i < swarmSize; i++)
            {
                var particle = new Particle(_setup.Subjects, _setup.Rooms, _setup.TimeSlots, _random);
                particle.Schedule.Fitness = _evaluator.EvaluateWithWeights(particle.Schedule.Assignments);
                particle.BestSchedule = particle.Schedule.Clone();

                if (globalBest == null || particle.Schedule.Fitness < globalBest.Fitness)
                    globalBest = particle.BestSchedule.Clone();

                swarm.Add(particle);
            }

            var reporter = new ProgressReporter(iterations);
            var sw = System.Diagnostics.Stopwatch.StartNew();

            for (int iter = 0; iter < iterations; iter++)
            {
                foreach (var particle in swarm)
                {
                    foreach (var assignment in particle.Schedule.Assignments)
                    {

                        if (_random.NextDouble() < 0.1)
                            assignment.Teacher = assignment.ClassType.TeacherNames[_random.Next(assignment.ClassType.TeacherNames.Count)];
                        if (_random.NextDouble() < 0.1)
                            assignment.Room = _setup.Rooms[_random.Next(_setup.Rooms.Count)];
                        if (_random.NextDouble() < 0.1)
                            assignment.TimeSlot = _setup.TimeSlots[_random.Next(_setup.TimeSlots.Count)];
                    }

                    if (useCorrectionModule && _random.NextDouble() < 0.001)
                        particle.Schedule.Assignments = _correctionModule.ApplyCorrections(particle.Schedule.Assignments, progressReport: progressReporting);

                    particle.Schedule.Evaluate(_evaluator);
```

```
59
60                  if (particle.Schedule.Fitness < particle.BestSchedule.Fitness)
61                  {
62                      particle.BestSchedule = particle.Schedule.Clone();
63
64                      if (globalBest == null || particle.Schedule.Fitness < globalBest.Fitness)
65                          globalBest = particle.BestSchedule.Clone();
66                  }
67              }
68
69              var time = sw.Elapsed.TotalSeconds;
70              var meanFitness = swarm.Average(g => g.BestSchedule.Fitness);
71
72              stats.Add(new GraphPoint(time, iter, globalBest.Fitness, meanFitness));
73              if (progressReporting)
74                  reporter.Report(iter, $"Global Best Fitness:{globalBest.Fitness} | Mean Fitness: {
      meanFitness}");
75          }
76
77          return globalBest;
78      }
79
80      // Constructor and othe polymorphic overloads
81  }
82 }
```

Listing 5.15: PSOEngine.cs

- **Initialization**
  - The algorithm begins by creating a swarm of particles (default size 30).
  - Particles can be initialized randomly, from a provided initial swarm, or a combination of the two approaches, allowing for flexible starting conditions (lines 18-35).
  - The fitness of each schedule is evaluated and the best schedule found by any particle is tracked as the global best (line 28).
  - Other local variables are also instantiated to aid in the algorithm execution.
- **Main Iterative Process**
  - The algorithm runs for a specified number of iterations (default 100).
  - In each iteration (lines 25-75), every particle updates its schedule:
    - With a small probability, the teacher, room, and/or timeslot of the assignments can be randomly changed (lines 44-53).
    - Optionally, the correction module is applied to the schedule with a very low probability, targeting constraint violations and improving the fitness value (lines 55-56).
  - After modifications, the fitness of the schedule is re-evaluated (line 58).
  - If a particle discovers a schedule with better fitness than its previous best, it updates its BestSchedule (lines 60-66).
  - If any particle finds a schedule better than the current global best, the global best is updated (lines 64-65).

- **Other processes**
  - The algorithm records statistics at each iteration, including elapsed time, iteration number, global best fitness, and mean fitness to create graphs that illustrate evolution on the schedules generated (lines 69-72).
  - Optionally, the progress can be reported in real-time, providing insight into convergence and performance of the algorithm (lines 73-74).

5.2.4 Genetic Algorithm

The GA implemented for university course scheduling is an evolutionary approach that iteratively improves a population of candidate schedules through selection, crossover, and mutation. The algorithm is designed to efficiently explore the solution space and converge towards high-quality, feasible schedules.

Gene Representation:

Listing 5.16 shows the implementation of the Gene class.

```
1  using SchoolScheduler.Models;
2  using SchoolScheduler.Core;
3
4  namespace SchoolScheduler.Genetic
5  {
6      public class Gene
7      {
8          public Schedule Schedule;
9
10         public Gene()
11         {}
12
13         public Gene(List<Subject> subjects, List<Room> rooms, List<TimeSlot> slots, Random rand)
14         {
15             Schedule = new Schedule(subjects, rooms, slots, rand);
16         }
17
18         // Crossover: single point
19         public static Gene Crossover(Gene parent1, Gene parent2)
20         {
21             int point = new Random().Next(parent1.Schedule.Assignments.Count);
22             var childAssignments = parent1.Schedule.Assignments.Take(point)
23                 .Concat(parent2.Schedule.Assignments.Skip(point)).Select(a => a.Clone()).ToList();
24             return new Gene
25             {
26                 Schedule = new Schedule(childAssignments, int.MaxValue)// Fitness will be evaluated later
27             };
28         }
29
30         // Mutation: randomly change one assignment
31         public void Mutate(SetupService setup)
32         {
33             if (Schedule.Assignments.Count == 0) return;
34             int idx = new Random().Next(Schedule.Assignments.Count);
35             var assignment = Schedule.Assignments[idx];
36             assignment.Teacher = assignment.ClassType.TeacherNames[new Random().Next(assignment.
       ClassType.TeacherNames.Count)];
37             assignment.Room = setup.Rooms[new Random().Next(setup.Rooms.Count)];
38             assignment.TimeSlot = setup.TimeSlots[new Random().Next(setup.TimeSlots.Count)];
39         }
40     }
```

```
41 }
```

Listing 5.16: Gene.cs

- Each individual in the population is represented by a Gene object, which has a Schedule containing all assignments for subjects, rooms, teachers, and timeslots.
- **Crossover and Mutation:**
  - Crossover is implemented as single-point crossover, combining assignments from two parent genes to produce a child gene (lines 18-28). Both schedules are split at a random index and swapping the second portion of the gene between parents.
  - Mutation randomly alters one assignment in a gene, introducing diversity and helping the population escape local optima (lines 30-39).

Engine

The GeneticEngine class, Listing 5.17, manages the execution of the genetic algorithm.

```
1  using SchoolScheduler.Models;
2  using SchoolScheduler.Core;
3  using SchoolScheduler.Core.Correction;
4  using SchoolScheduler.Utils;
5
6  namespace SchoolScheduler.Genetic
7  {
8      public class GeneticEngine
9      {
10         private readonly SetupService _setup;
11         private readonly FitnessEvaluator _evaluator;
12         private readonly CorrectionModule _correctionModule;
13         private readonly Random _random = new();
14
15         public Schedule Run(out List<GraphPoint> stats, int populationSize = 30, int generations = 100,
           double mutationRate = 0.2, List<Gene> initialPopulation = null, bool useCorrectionModule = true,
           bool progressReporting = true)
16         {
17             stats = new List<GraphPoint>();
18             var population = new List<Gene>();
19
20             initialPopulation ??= new List<Gene>();
21             population.AddRange(initialPopulation);
22
23             var random = new Random();
24             for (int i = population.Count; i < populationSize; i++)
25             {
26                 population.Add(new Gene(_setup.Subjects, _setup.Rooms, _setup.TimeSlots, _random));
27             }
28
29             foreach (var gene in population) gene.Schedule.Evaluate(_evaluator);
30
31             var reporter = new ProgressReporter(generations);
32             var sw = System.Diagnostics.Stopwatch.StartNew();
33             for (int gen = 0; gen < generations; gen++)
34             {
35                 // Variation: mutation and correction
36                 foreach (var gene in population)
37                 {
38                     if (random.NextDouble() < mutationRate)
39                         gene.Mutate(_setup);
40                     if (useCorrectionModule && random.NextDouble() < 0.001)
```

```
41                      gene.Schedule.Assignments = _correctionModule.ApplyCorrections(gene.Schedule.
      Assignments, progressReport: progressReporting);
42                  gene.Schedule.Evaluate(_evaluator);
43              }
44              // Selection: top 50%
45              population = population.OrderBy(g => g.Schedule.Fitness).Take(populationSize / 2).ToList();

46              // Crossover to refill population
47              while (population.Count < populationSize)
48              {
49                  var parents = population.OrderBy(_ => Guid.NewGuid()).Take(2).ToList();
50                  var child = Gene.Crossover(parents[0], parents[1]);
51                  child.Schedule.Evaluate(_evaluator);
52                  population.Add(child);
53              }
54              var bestFitness = population[0].Schedule.Fitness;
55              var meanFitness = population.Average(g => g.Schedule.Fitness);
56
57              var time = sw.Elapsed.TotalSeconds;
58              stats.Add(new GraphPoint(time, gen, bestFitness, meanFitness));
59
60              if (progressReporting)
61                  reporter.Report(gen, $"Global Best Fitness:{bestFitness} | Mean Fitness: {meanFitness}")
      ;
62          }
63          return population[0];
64      }
65
66      // Constructor and othe polymorphic overloads
67  }
68 }
```

Listing 5.17: GeneticEngine.cs

- **Initialization:**
  - The algorithm starts by creating a population of genes (default size 30).
  - Genes can be initialized randomly, from a provided initial population, or a combination of the two approaches (lines 20-27).
  - The fitness of each schedule is evaluated using the fitness evaluator (line 29).
- **Main Evolutionary Process:**
  - The algorithm runs for a specified number of generations (default 100).
  - In each generation (lines 33-62), every gene undergoes variation:
    - With a given mutation rate (default 0.2), a gene may be mutated by randomly changing the teacher, room, or timeslot of an assignment (lines 38-39).
    - Optionally, the correction module is applied to the schedule with a very low probability, targeting constraint violations and improving the fitness value (lines 40-41).
  - After variation, the fitness of each schedule is re-evaluated (line 42).
  - Selection is performed by retaining the top 50% of the population based on fitness (line 45).

  - The population is then refilled by generating new genes through single-point crossover between randomly selected parents, followed by fitness evaluation (lines 46-53).

- **Other processes**
  - The algorithm records statistics at each iteration, including elapsed time, iteration number, global best fitness, and mean fitness to create graphs that illustrate evolution on the schedules generated (lines 57-58).
  - Optionally, the progress can be reported in real-time, providing insight into convergence and performance of the algorithm (lines 60-61).

5.2.5 Other Components

In addition to the primary models and core modules, several other components were developed to enhance the functionality and usability of the UCSP solution. These components include utility classes, a schedule editor, and a statistics module, each serving a specific purpose within the system.

Utils

- **JSONScheduleSerializer:** Implemented to handle the serialization and deserialization of schedules to and from JSON format, facilitating easy storage and retrieval of scheduling data.
- **GraphGenerator:** Implemented to run a predefined set of configurations for each algorithm and generate performance graphs based on the results, for the next chapter analysis.
- **Progress Reporter:** Created to report the progress of the scheduling algorithms, providing feedback during long-running operations. It provides a console progress bar, with estimated time remaining, percentage completed and comments about the current operation. Figure 5.4 shows an example of the progress reporter in action.

Figure 5.4: Progress Reporter

ScheduleEditor

A simple console-based schedule editor was developed to allow manual adjustments to the generated schedules, providing flexibility in fine-tuning the results.

- **Display the schedule**, using the Schedule Viewer
- **Edit an assignment**, allowing changes in the teacher, room, and timeslot, with error prevention techniques
- **Edit all assignments**, allowing for quicker changes in multiple assignments, with key words to navigate the schedule, saving the current state, and duplicate the teacher and room information onto consecutive assignments, assigning consecutive timeslots
- **Show fitness** and other statistics

- **Save schedule**

Statistics

A statistics module was implemented to analyze certain aspects and compare generated schedules. Some statistics were implemented as an example of the possibilities of the module.

- **Constraint Violations:** Lists each violated constraint, its number of occurrences and global fitness value.
- **Room Inefficiency:** Calculates the percentage of room occupation throughout the schedule.
- **Average Student Gaps:** Calculates the average duration of gaps between classes for each student across all days, providing insight into schedule compactness and student waiting times.
- **Average Hours Per Day for Students:** Calculates the average time a student spends in the school premises.

Figure 5.5 shows an example of the statistics module, comparing the generated schedule before and after applying the Correction Module, showing a significant reduction in constraint violations and improvement in overall fitness.

```
Corrected Schedule (Fitness: 11741 (-4094))
--------------------
==== Assignment Statistics ====
Constraint Violations Summary: 3990 (-607)
        Student Gaps: 1478 (+0)
        Continuous Class Gaps: 1236 (+0)
        Incorrect Order Of Classes: 701 (-16)
        Multiple Assignments In Same TimeSlot For Classes: 225 (-1)
        Preferred Room Type Constraint: 187 (-208)
        Students Min 3 Hours Day: 106 (-1)
        Continuous Class Different Rooms: 25 (+0)
        Continuous Class Different Teachers: 20 (+0)
        Teacher Conflict: 12 (-46)
        Room Capacity Exceeded: 0 (-193)
        Room Conflict: 0 (-142)
Percentage of room use: 35,3% (+4,8%)
Average student gaps: 5,94 (+2,97)
Average hours per day for students: 3,03 (+0,00)
```

Figure 5.5: Statistics comparing the application of the Correction Module to generated schedule

5.3 Summary

This chapter provided a detailed overview of the implementation of the University Course Scheduling Problem (UCSP) solution. It covered the technologies and programming languages used, the structure of the codebase, and the main components of the system, including models, core modules, utility functions, and the algorithms employed for scheduling. The implementation details highlighted the modular and extensible design of the solution, facilitating future enhancements and adaptations to meet evolving requirements. The next chapter will present an analysis of the results obtained from applying the implemented algorithms to various scheduling scenarios.

Chapter 6

Case Study

The purpose of this case study is to analyse the performance of the developed scheduling system and to evaluate the results it produces under different experimental settings. Multiple configurations of algorithms were tested, and their performance was compared across various metrics, including fitness values and runtime performance.

6.1 Experimental Setup

The experiments were designed to evaluate and compare the efficiency and effectiveness of two implemented algorithms: the Genetic Algorithm and the Particle Swarm Optimization. Both algorithms were developed with configurable parameters, enabling systematic analysis under different conditions. Performance metrics included fitness value progression and execution time per generation/iteration. The data collected is a result of the average of 5 runs for each configuration, executed on a Windows 11 machine, equipped with an Intel Core i7 10th Generation processor.

6.1.1 Object of study

The case study focused on generating schedules for the 2nd semester of the Bachelor's Degree in Informatics Engineering at ISEP, excluding evening and night classes. The list below provides an overview of the curricular structure and resources considered in the scheduling experiments.

- **First Year:**
  - Number of subjects: 5
  - Total weekly hours: 20
  - Number of classes: 16
  - Number of teachers: 38
- **Second Year:**
  - Number of subjects: 5
  - Total weekly hours: 24
  - Number of classes: 14
  - Number of teachers: 47
- **Third Year:**

  - Number of subjects: 3
  - Total weekly hours: 9
  - Number of classes: 12
  - Number of teachers: 22
- **Overall Totals:**
  - Number of subjects: 13
  - Total weekly hours: 53
  - Number of classes: 42
  - Number of teachers: 96
- **Available Resources:**
  - Rooms: 27
  - Weekly hours: 55 (from 8:00 to 18:00, Monday to Friday, 8:00 to 13:00 on Saturdays)

6.1.2 Configuration

To ensure clarity and consistency, each configuration was named according to the following convention:

Algorithm_Population/SwarmSize(_Initial(_LEI))(_Correction)

The elements of this naming scheme are defined as follows:

- **Algorithm:** identifies the algorithm applied, either GA or PSO.
- **Population/Swarm Size:** denotes the number of individuals (in GA) or particles (in PSO) present during the execution of the algorithm. Two values were tested: 30 and 100. A smaller size (30) allows for faster computation but may limit the diversity of solutions, potentially leading to premature convergence. A larger size (100) increases diversity and the chance of finding better solutions, but requires more computational resources and time.
- **Initial:** indicates that the configuration uses schedules, generated by runs without this tag, as part of the initial population or swarm.
  - **LEI:** Licenciatura em Engenharia Informática (Bachelor's Degree in Informatics Engineering) specifies that, in addition to the generated schedules, the initialization includes a manually created real-world schedule for the academic year 2024/25.
- **Correction:** indicates whether the Correction Module was applied.

In addition to these parameters, the following settings were consistently applied across all experiments:

- **Number of Generations/Iterations:** 1000 generations for GA and 2000 iterations for PSO. These values were chosen to ensure that both algorithms had sufficient time to collect long term execution data, allowing them to converge towards optimal or

near-optimal solutions, while also allowing for a comparison of their performance over an equivalent computational effort.

- **Mutation Rate:** 20% of the population was subject to mutation in each generation. This rate was selected to maintain a balance between exploration and exploitation, ensuring that the algorithm could explore new areas of the solution space while still refining existing solutions.
- **Correction Rate:** 0.1% of the population/swarm was processed by the Correction Module per generation/iteration. The correction module was configured with a low probability of execution, ensuring that it did not overshadow the core search performed by the evolutionary algorithms. Instead, it functioned as a supportive mechanism, periodically enhancing solution quality without becoming the dominant search component.
- **Initialization:** with random schedule generation to complete the initial populations/swarms.

As previously mention, LEI is a manually created real-world schedule for the academic year 2024/25 for the Bachelor's Degree in Informatics Engineering at ISEP. Figure 6.1 displays the statistics calculated for the schedule.

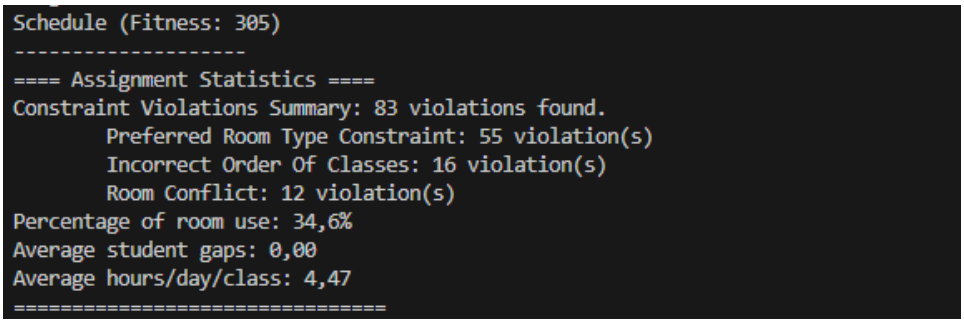

Figure 6.1: Statistics of LEI

6.2 Results and Analysis

This section analyses the outcomes of the experiments, separated into two dimensions of evaluation: fitness evolution over time, and execution time evolution over generation/iteration.

6.2.1 Fitness Evolution Over Time

The fitness value is calculated by the sum of violations of hard and soft constraints, weighted accordingly. Therefore, the lower the fitness value, the better the quality of the schedule. Figures 6.2 and 6.3 illustrate the fitness evolution over time for all configurations of GA and PSO, respectively. Both algorithms demonstrate a general trend of decreasing fitness values over time, indicating that they are effectively improving the quality of the schedules as they progress through generations/iterations.

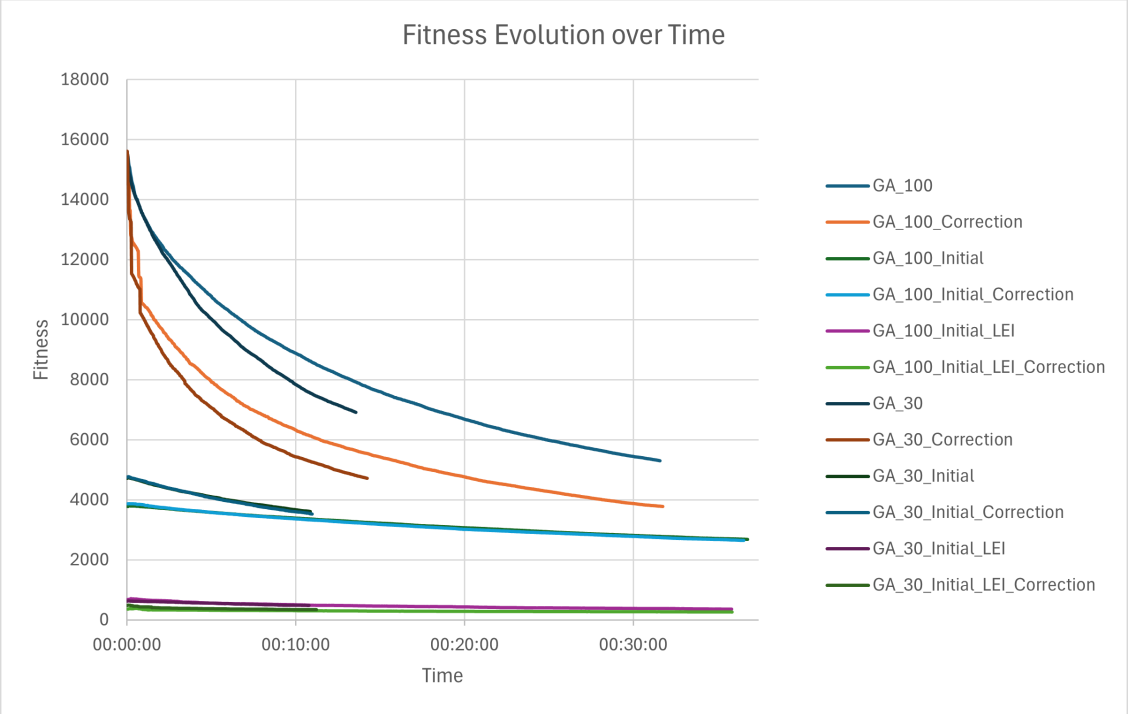

Figure 6.2: Fitness evolution for GA

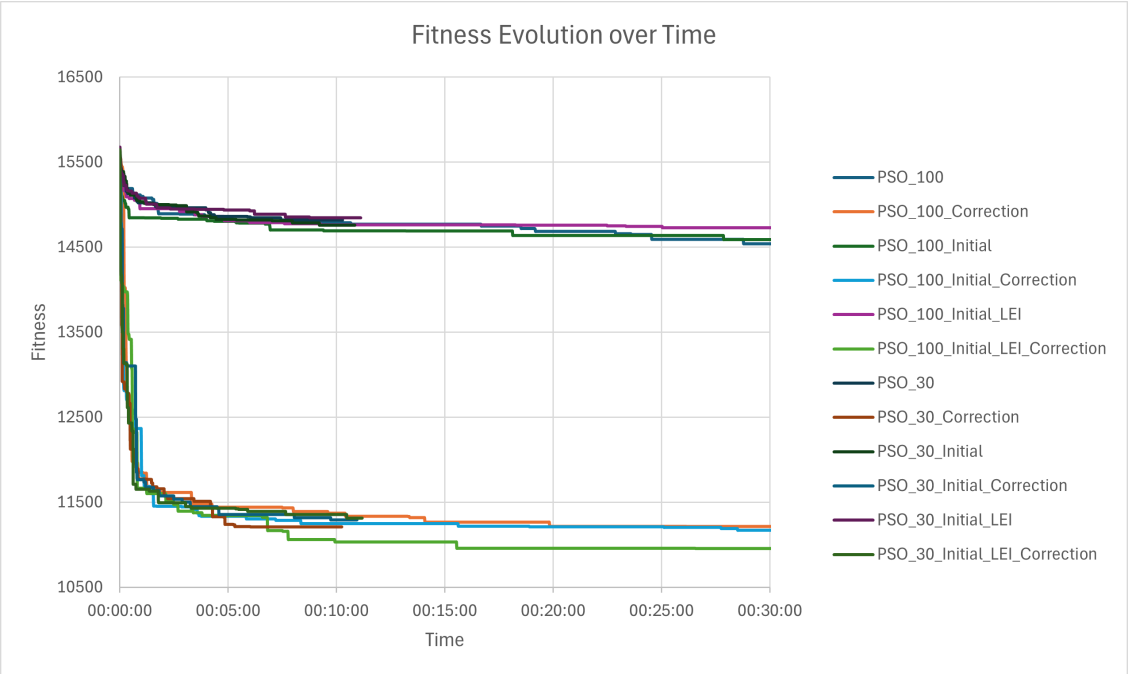

Figure 6.3: Fitness evolution for PSO

Algorithms and Correction Module

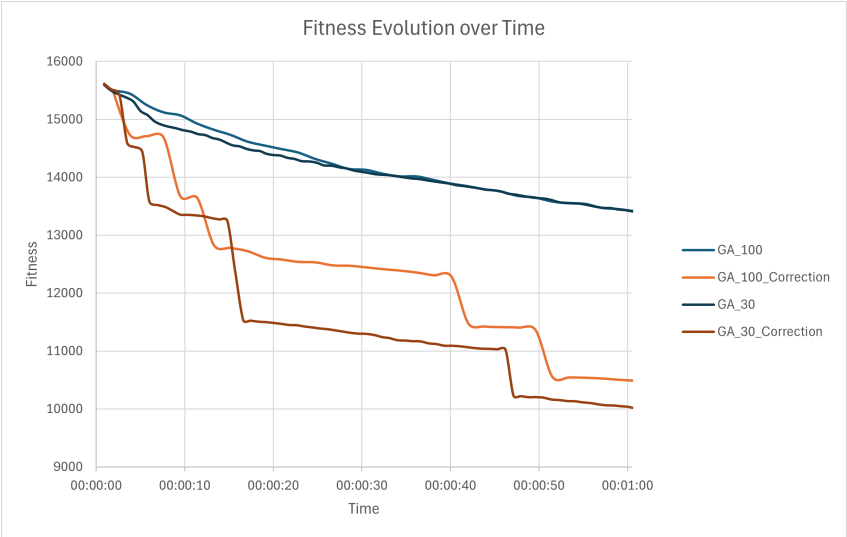

Figure 6.4: Fitness evolution for GA across population sizes and correction module settings

Figure 6.4 shows the effect of both population size and correction on GA, revealing a steeper slope when the algorithm has a smaller population, thus producing a lower overall fitness. Partnering the corection module, slope is further improved, observing rapid drops in the first minute of execution, which are larger the higher the fitness value is. After the first minute, the fitness value continues to improve, converging as the population homogenizes.

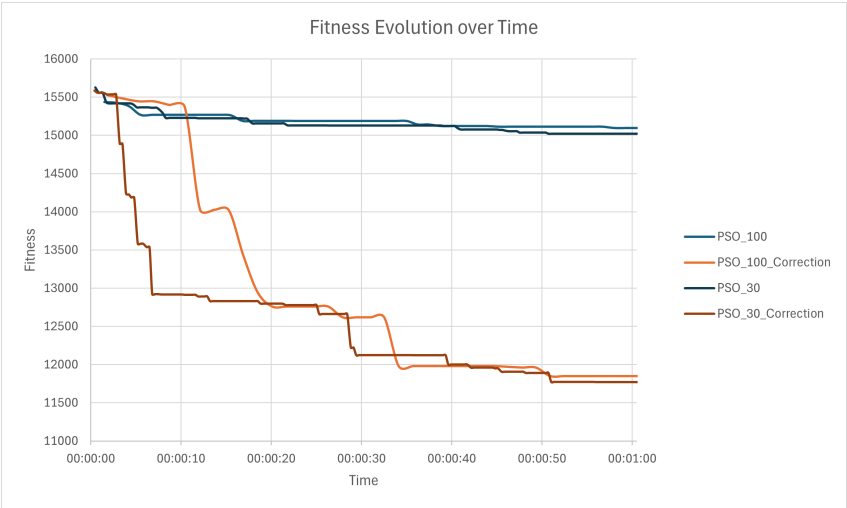

Figure 6.5: Fitness evolution for PSO across swarm sizes and correction module settings

In Figure 6.5, PSO shows a fast convergence compared without the use of the correction module. The fitness often maintains its value, with only small improvements from time to time, apart from sudden improvements caused by the correction module. Smaller swarms present better fitness values.

Algorithms with Initialization and Correction Module

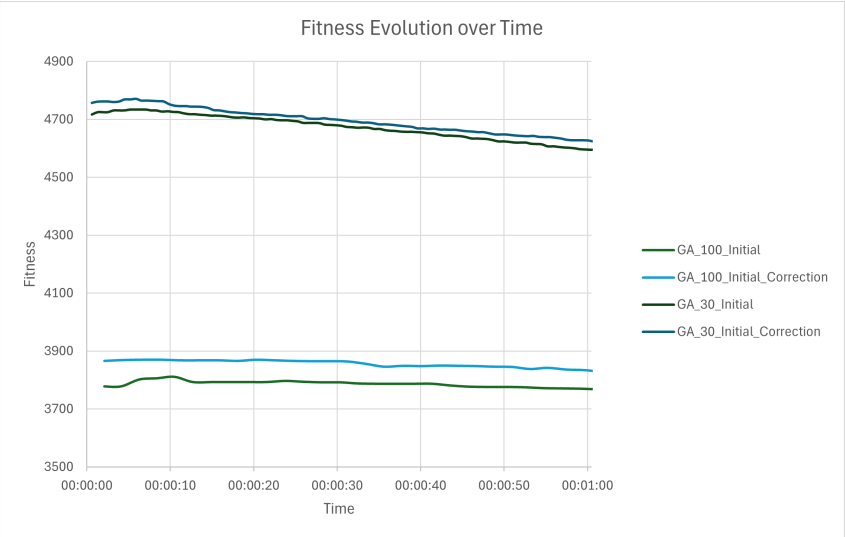

Figure 6.6: Fitness evolution for GA with initialization, across population sizes and correction module settings

As shown in Figure 6.6, initialization strategies strongly affect GA performance. Despite observing the same results in slope variation between different population sizes from the previous analysis, the addition of previously generated schedules to new population improved drastically the start fitness values, leading to a premature convergence and low variation in fitness value. The correction module as low effect improvements as it is reaching its limits, displaying a small increase in fitness value at the start that was not removed during the execution.

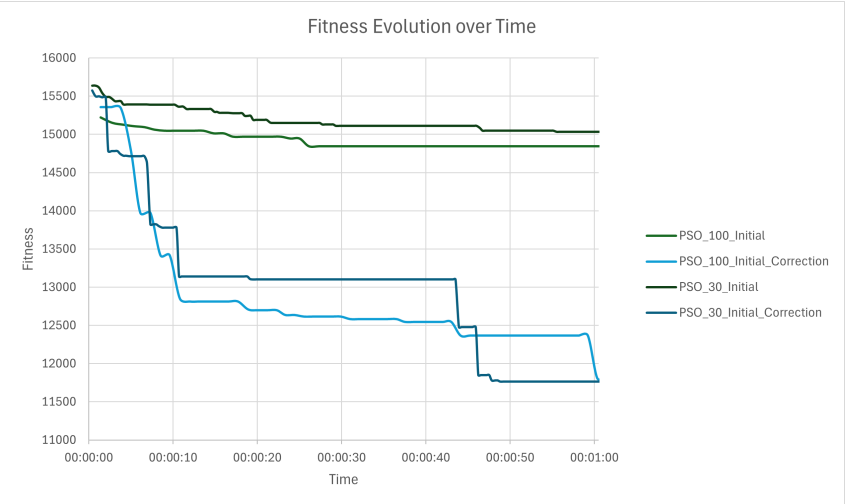

Figure 6.7: Fitness evolution for PSO with initialization, across swarm sizes and correction module settings

Figure 6.7 displays similar fitness evolutions to those found in runs without manual initialization, as the algorithm does not present the initial fitness of particles created from previously

generated schedules, only improvements are made. Larger swarms benefit from initialization, leading to slightly better fitness values.

Algorithms with Special Initialization and Correction Module

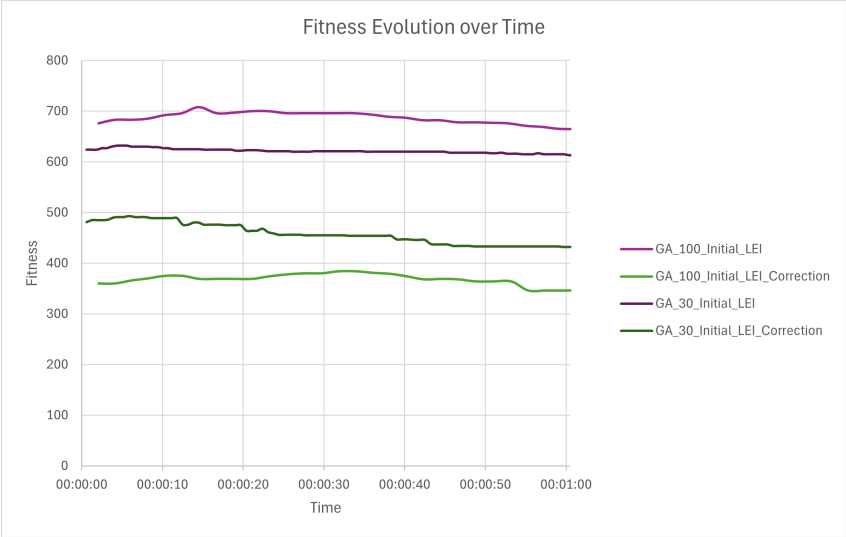

Figure 6.8: Fitness evolution for GA with special initialization, across population sizes and correction module settings

The addition of LEI to the initial population continues to lower the initial fitness value, as shown in Figure 6.8, while maintaining the same slope. Only at this stage is a schedule found with a better fitness value than the manually generated one, achieved through the GA_100_Initial_LEI_Correction configuration, as shown in Figure 6.9.

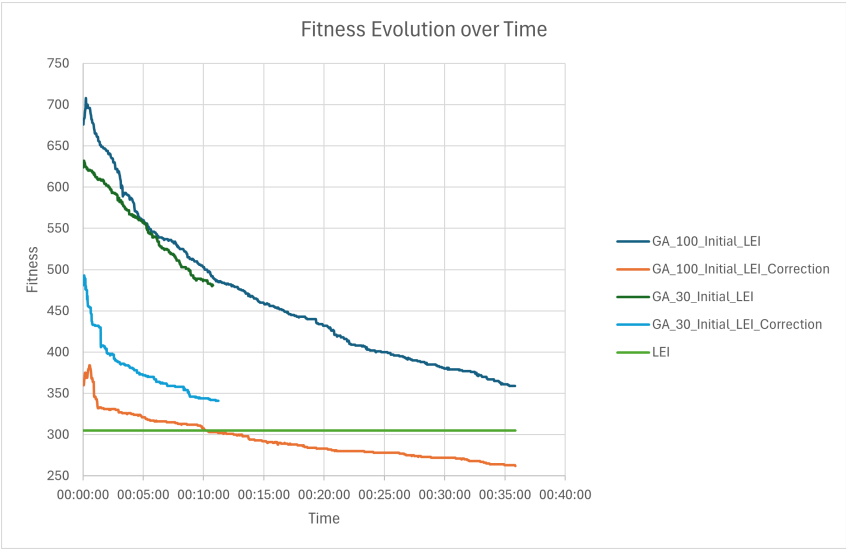

Figure 6.9: Fitness evolution long term for GA with special initialization, across population sizes and correction module settings

Figure 6.10: Fitness evolution for PSO with special initialization, across swarm sizes and correction module settings.

As illustrated in Figure 6.10, the addition of LEI to the initial swarm does not impact severely the fitness evolution in the short term. PSO_100_Initial_LEI_Correction displayed the best results after approximately seven minutes of runtime, but the difference between the point of convergerce, compared with other configurations, is almost insignificant due to the high fitness value.

GA Across All Configurations

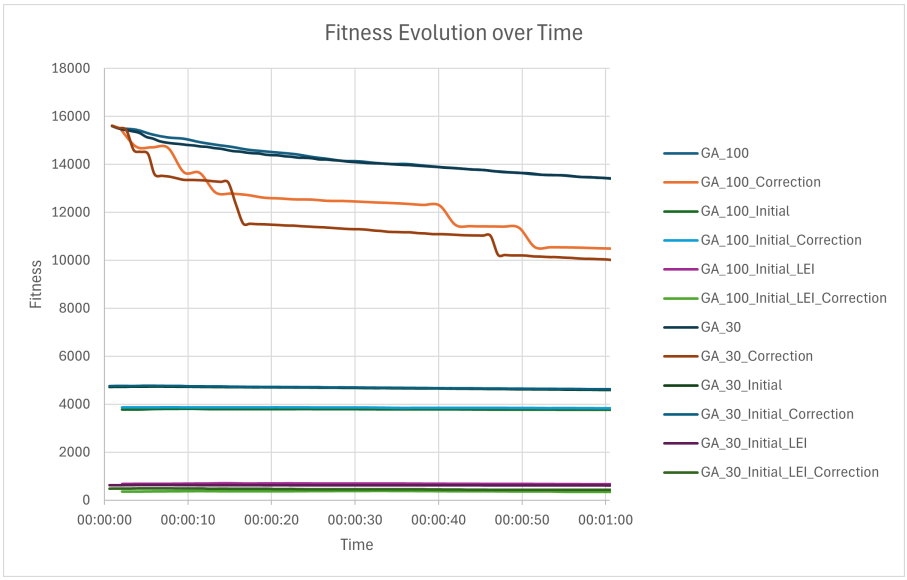

Figure 6.11: Overview of GA fitness evolution

Figure 6.11 summarizes all GA runs. The overall pattern confirms the main differential configuration is the initialization, displaying the three different tiers of configurations, and large benefits from the usage of the correction module on the continuous improvement of fitness.

PSO Across All Configurations

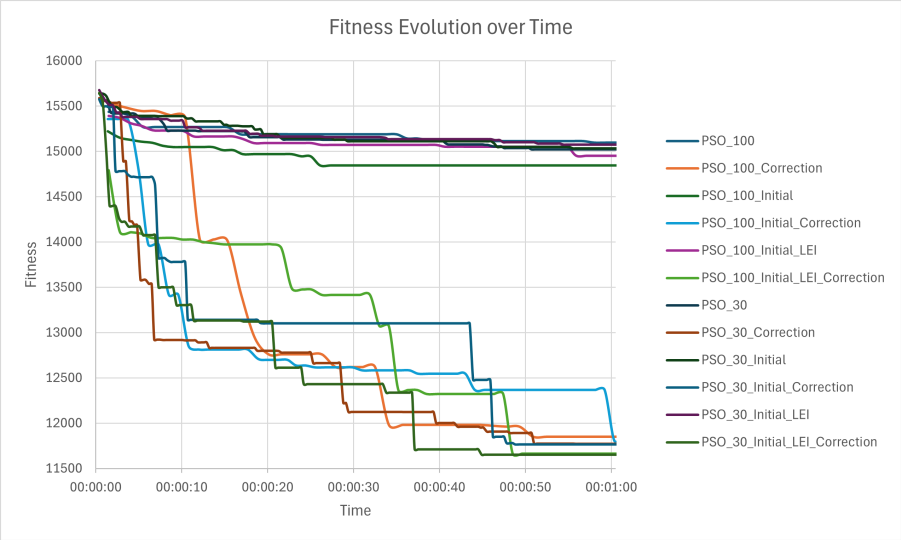

Figure 6.12: Overview of PSO fitness evolution

Figure 6.12 details PSO performance across all cases. It highlights two main groups of configurations defined by the presence or absence of the correction module, which delays the fitness convergence. Despite the multiple changes in the configuration with best results in the initial phase, the convergence points are all similar for each group. Larger swarm sizes present better fitness value.

Comparison between GA and PSO

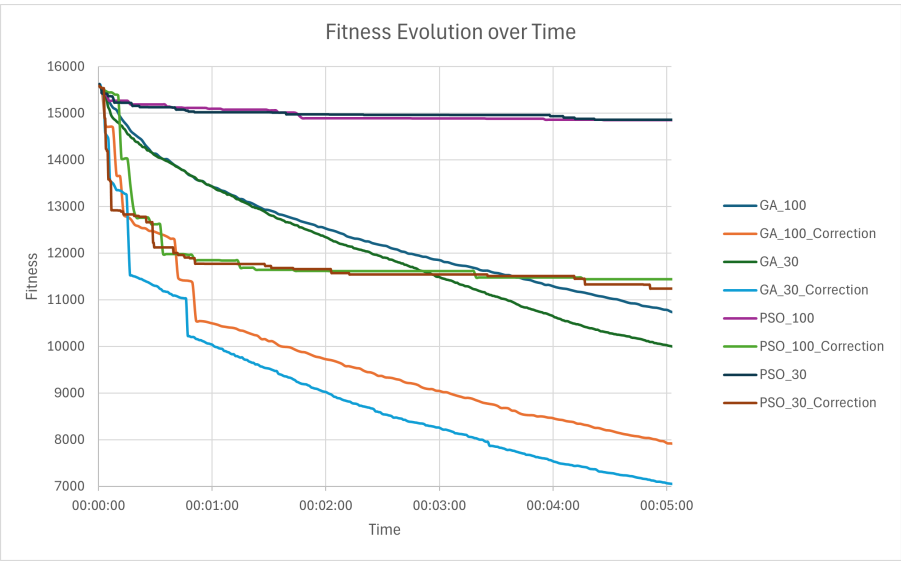

Figure 6.13: Comparison of GA and PSO fitness evolution, across population/swarm sizes and correction module settings

The Genetic Algorithm consistently outperforms Particle Swarm Optimization in terms of final fitness values, as it is illustrated in Figure 6.13, especially when previously generated

schedules or the manually created LEI schedule are initialized. Even with the benefits from the correction module in the early stages, that benefit similarly both algorithms, the PSO cannot match the consistent improvements GA shows long term, even with the absence of the correction module. Genetic Algorithm maintains steady progress throughout the generations, continuing to improve previously generated schedules, ultimately achieving superior solutions.

6.2.2 Evolution of Time Over Generations/Iterations

Previous analysis evaluated the algorithms against time of execution. It is due to the different computational cost the algorithms demand for each generation/iteration.

Execution Time for GA

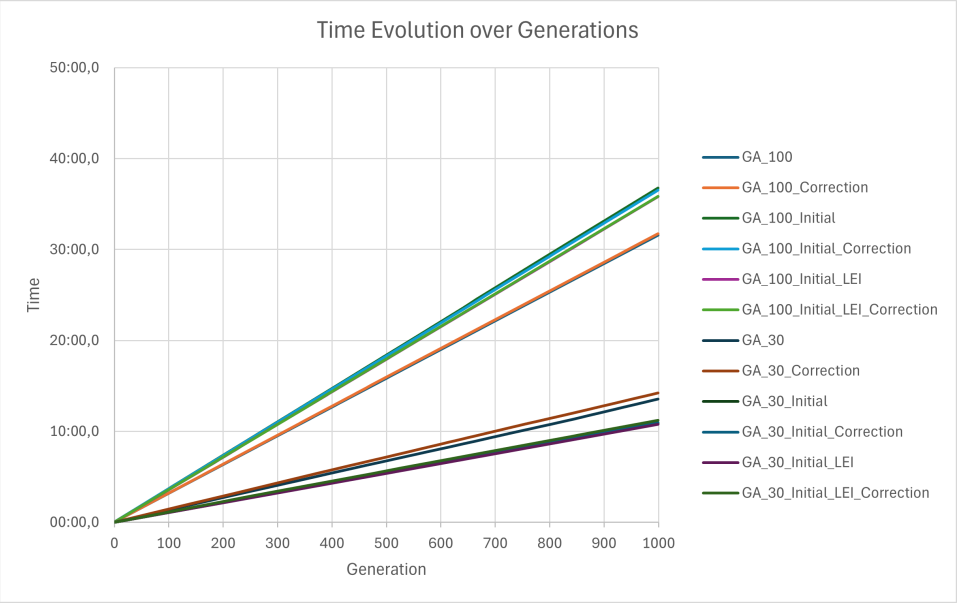

Figure 6.14: Execution time per generation for GA
fu

Figure 6.14 shows the runtime for each configuration of GA. The algorithm presents a linear progression, with steeper slopes the greater the population size. The application of the correction results in a slight increase in slope, but insignificant.

Execution Time for PSO

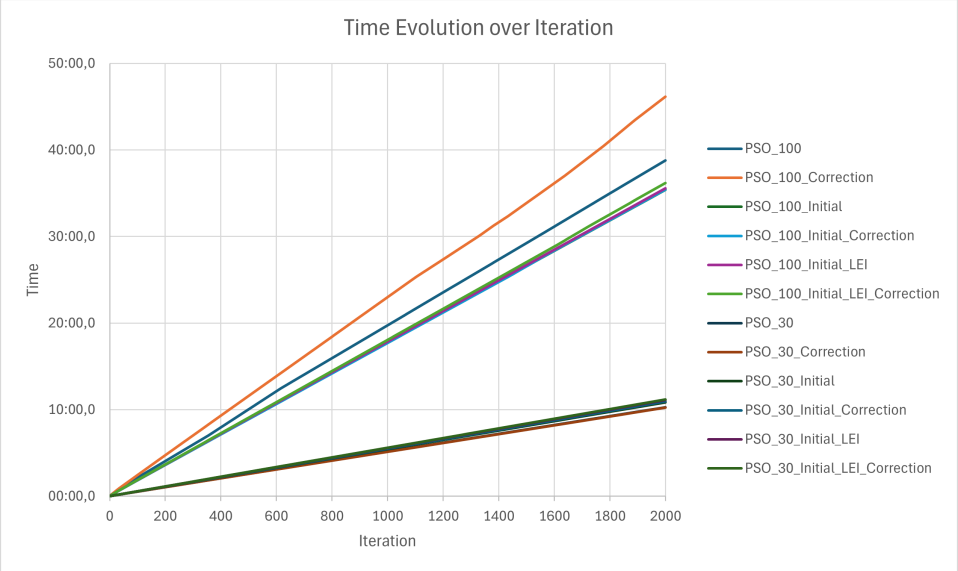

Figure 6.15: Execution time per iteration for PSO

Similar to GA, PSO has a linear time progression in execution, with steeper slopes for larger swarm sizes, as it is shown in Figure 6.15. Correction module provokes a light increase in slope.

Comparison between GA and PSO

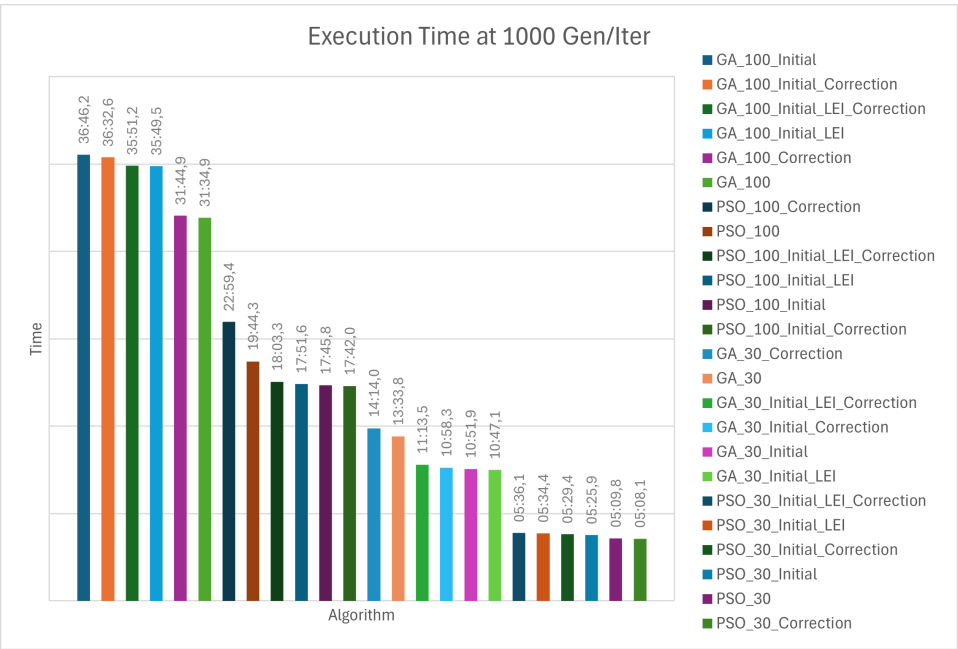

Figure 6.16: Comparison of GA and PSO execution time at the 1000 generations/iterations

As illustrated in Figure 6.16, PSO is generally faster per iteration than GA per generation, as both algorithms present linear time progressions. Due to this fact, PSO execution ran for 2000 iterations, so that cumulative runtimes are identical to runtimes of GA. Full runtime for PSO is displayed in Figure 6.17.

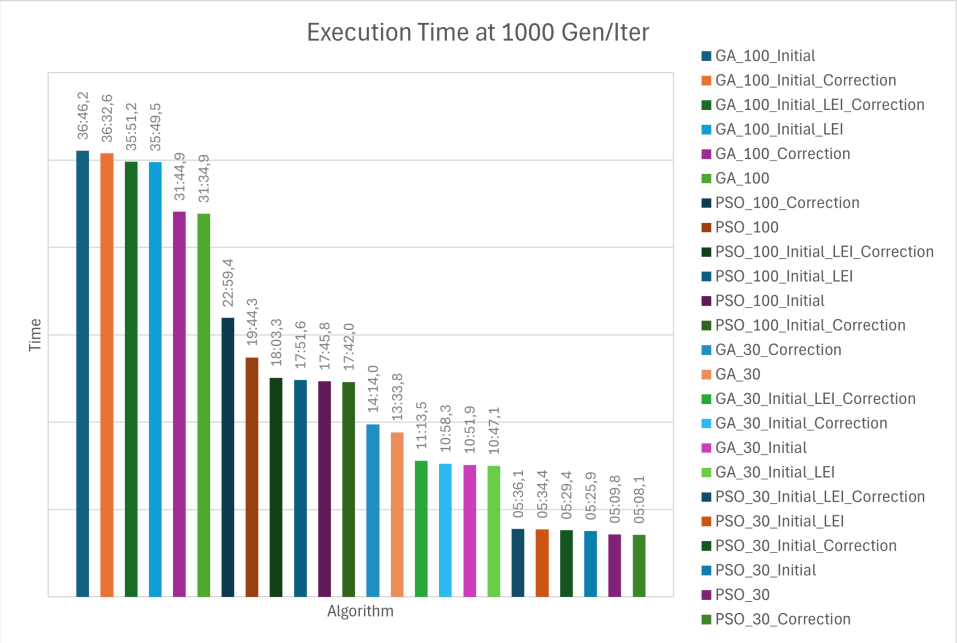

Figure 6.17: PSO execution time at the 2000 iterations

6.3 Summary

The case study demonstrates that GA and PSO each have distinct advantages. GA is more effective for achieving high-quality solutions given sufficient time, whereas PSO excels in speed and early convergence. The correction module enhances both algorithms without impacting heavily in execution time. Larger population/swarm size runs provide advantages and disadvantages to the schedule quality, which should be taken into consideration and addressed to the specific needs and priorities when generating the schedule.

Chapter 7

Conclusion

This dissertation presented a comprehensive approach to solving the University Course Scheduling Problem, a combinatorial optimization problem faced by academic institutions. The work began with an analysis of the requirements, constraints, and practical challenges inherent to university scheduling, including teacher availability, room capacity, curriculum structure, and institutional policies. The review in the state of the art revealed various algorithmic strategies, highlighting the strengths and limitations of existing methods. Based on this foundation, a modular and extensible scheduling system was designed and implemented in C#, leveraging the .NET framework for robustness and maintainability.

The developed system was focused on two metaheuristic algorithms: Particle Swarm Optimization and Genetic Algorithm. These algorithms were tailored to the UCSP, incorporating domain-specific models for subjects, classes, teachers, rooms, and timeslots. A correction module was developed to systematically address constraint violations, using targeted Local Search and repair strategies to improve schedule quality and validity. The system was evaluated using real-world data from the Bachelor's Degree in Informatics Engineering at ISEP, with extensive experimentation across multiple configurations, including variations in population size, initialization strategies, and correction module integration.

The results demonstrate the capabilities of the proposed approach, with the generation of schedules with higher quality than manually created ones, based on the implemented criteria. The dissertation concludes with a critical analysis of the system's strengths, limitations, and opportunities for future enhancement, providing valuable insights for both academic research and practical deployment in educational scheduling.

7.1 Goals Achieved

The main objectives of this dissertation were:

- **Primary Goal:** Improve the schedule generation process, either by reducing the time and resources needed, improving the quality of the generated schedules, or both outcomes.
- **Secondary Goals:**
  - Explore and apply cutting-edge algorithms and AI methods to automate and optimize the scheduling process.
  - Design a proposed solution capable of accommodating dynamic institutional changes, such as varying class sizes, faculty availability, and infrastructural constraints.

  - Establish and evaluate metrics to assess the performance of the proposed scheduling solution in comparison to traditional methods. Metrics include time savings, resource utilization, conflict resolution, and stakeholder satisfaction.

All goals were successfully achieved, reached when the system produced schedules with higher quality than manually created ones, through the implementation of Genetic Algorithm and Particle Swarm Optimization algorithms, along with a dedicated correction module. Several configurations and strategies were tested, demonstrating the effectiveness of the approach in generating schedules, capable of adapting to various constraints and requirements.

7.2 Key Findings

The evaluation and analysis of the developed system led to several key findings:

- The modular design of the system facilitated experimentation, comparative analysis, and future extension, allowing for easy integration of new algorithms, constraints, and correction strategies.
- The use of real-world data and systematic experimentation provided valuable insights into the practical challenges and trade-offs involved in automated scheduling.
- Both PSO and GA algorithms are capable of improving the quality schedules, each with its own strengths and weaknesses, depending on the specific configuration and problem instance.
- PSO demonstrated rapid convergence and efficiency, making it well-suited for scenarios requiring quick solutions, while GA provided greater long term continued improvements, often producing better results.
- The correction module plays a crucial role in improving schedule feasibility and fitness, effectively resolving constraint violations that arise during algorithmic search, without significantly increasing computational overhead.
- The choice of initialization strategy, population size, and algorithm parameters significantly impacts the performance and quality of the generated schedules, highlighting the importance of careful configuration and tuning to the current needs and context.

7.3 Limitations

Despite its strengths, the system has some limitations:

- Reliance on static input data, which limits adaptability to dynamic changes in scheduling requirements, teacher availability, or curriculum changes.
- Manual configuration required for some parameters, data sources, and initialization strategies, which may hinder usability for non-technical users.
- The correction module, while effective for hard constraints, requires further development to address soft constraints.
- Scalability challenges when handling very large or highly complex scheduling scenarios, which may require further optimization or parallelization of algorithms.

- The system was primarily tested on data from a single institution and curriculum, which may limit the generalizability of the findings to other contexts or educational settings.
- The system currently does not support evening and night classes, which may be a requirement for some institutions.

7.4 Future Work

Building on the foundation established in this dissertation, future work may address the following directions:

- Continued development and refinement of the system, focusing on the listed limitations.
- Exploration of hybrid and ensemble approaches, combining multiple algorithms and correction strategies to further improve solution quality and robustness.
- Use of parallelization and distributed computing techniques to enhance scalability and reduce computation time for large and complex scheduling problems.

7.5 Final Remarks

This dissertation has presented a comprehensive approach to addressing the University Course Scheduling Problem, demonstrating the potential of metaheuristic algorithms and modular system design in generating high-quality schedules. The findings contribute to both academic research and practical applications, providing a foundation for future advancements in automated scheduling systems. Despite prior research highlighting the advantages of PSO, the results were lightly disheartening, due to the rapid convergence, which led to high fitness values, especially when compared to GA.

Bibliography

Babaei, Hamed and Amin Hadidi (2017). "Generating an optimal timetabling for multi-departments common lecturers using hybrid fuzzy and clustering algorithms". In: *journal of Artificial Intelligence in Electrical Engineering* 6.21, pp. 9–25.

Can, E., O. Ustun, and S. Saglam (July 2023). "Metaheuristic approach proposal for the solution of the bi-objective course scheduling problem". In: *SCIENTIA IRANICA* 30.4, pp. 1435–1449. issn: 1026-3098. doi: 10.24200/sci.2021.55005.4044.

Devi, M Uma et al. (2024). "Automated timetable generation for academic institutions - AIP Conference Proceedings". In: 3075.1.

Engenheiros, Ordem dos (Nov. 2016). *Código de Ética e Deontologia*. Accessed: Jan. 4, 2025. url: https://www.ordemengenheiros.pt/fotos/editor2/regulamentos/codigo_ed.pdf.

Erdeniz, Seda Polat and Alexander Felfernig (2018). "OCSH: optimized cluster specific heuristics for the university course timetabling problem - Proceedings of the 8th International Conference on Information Systems and Technologies". In: ICIST '18. doi: 10.1145/3200842.3200858. url: https://doi.org/10.1145/3200842.3200858.

Feutrier, Thomas, Marie-Éléonore Kessaci, and Nadarajen Veerapen (2021). "Investigating the landscape of a hybrid local search approach for a timetabling problem - Proceedings of the Genetic and Evolutionary Computation Conference Companion". In: GECCO '21, pp. 1665–1673. doi: 10.1145/3449726.3463175. url: https://doi.org/10.1145/3449726.3463175.

Hossain, Sk Imran et al. (Aug. 2019). "Optimization of University Course Scheduling Problem using Particle Swarm Optimization with Selective Search". In: *EXPERT SYSTEMS WITH APPLICATIONS* 127, pp. 9–24. issn: 0957-4174. doi: 10.1016/j.eswa.2019.02.026.

Jamal, Ade (Sept. 2017). "Multiple Scattered Local Search for Course Scheduling Problem - 2017 International Conference on Soft Computing, Intelligent System and Information Technology (ICSIIT)". In: pp. 114–119. doi: 10.1109/ICSIIT.2017.22.

Jiang, Cun-bo and Hao Liu (June 2019). "A course scheduling algorithm based on improved genetic algorithm with multi-objective constraints - 2019 Eleventh International Conference on Advanced Computational Intelligence (ICACI)". In: pp. 202–206. issn: 2573-3311. doi: 10.1109/ICACI.2019.8778493.

Luo, Yanrui and Peiyuan Niu (Nov. 2024). "Physical education teaching scheduling technology based on chaotic genetic algorithm". In: *SCIENTIFIC REPORTS* 14.1. issn: 2045-2322. doi: 10.1038/s41598-024-79646-y.

Al-Mahmud and M. A. H. Akhand (May 2014). "ACO with GA operators for solving University Class Scheduling Problem with flexible preferences - 2014 International Conference on Informatics, Electronics & Vision (ICIEV)". In: pp. 1–6. doi: 10.1109/ICIEV.2014.6850742.

Muklason, Ahmad et al. (2023). "Flexible Automated Course Timetabling System with Lecturer Preferences Using Hyper-heuristic Algorithm - Proceedings of the 7th International Conference on Sustainable Information Engineering and Technology". In: SIET

'22, pp. 258–262. doi: 10.1145/3568231.3568273. url: https://doi.org/10.1145/3568231.3568273.

Pillay, Nelishia and Ender Özcan (2019). "Automated generation of constructive ordering heuristics for educational timetabling". In: *Annals of Operations Research* 275, pp. 181–208.

Porto, Instituto Politécnico do (Oct. 2020). *Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto*. Accessed: Jan. 4, 2025. url: https://www.ipp.pt/comunidade/missao-equidade-diversidade-inclusao/DespachoP.PORTOP0402020CodigodeBoasPraticasedeConduta.pdf/view.

Pothitos, Nikolaos and Panagiotis Stamatopoulos (2016). "Piece of Pie Search: Confidently Exploiting Heuristics - Proceedings of the 9th Hellenic Conference on Artificial Intelligence". In: SETN '16. doi: 10.1145/2903220.2903242. url: https://doi.org/10.1145/2903220.2903242.

Ramdania, D. R. et al. (2019). "Comparison of genetic algorithms and Particle Swarm Optimization (PSO) algorithms in course scheduling - 4TH ANNUAL APPLIED SCIENCE AND ENGINEERING CONFERENCE, 2019". In: Journal of Physics Conference Series 1402. Ed. by AG Abdullah et al. issn: 1742-6588. doi: 10.1088/1742-6596/1402/2/022079.

Rashmi, KR and MB Abhishek (2021). "Automated University Timetable Generation using Prediction Algorithm". In.

Rodprasert, Nampetch, Unchalisa Taetragool, and Khajonpong Akkarajitsakul (2023). "Online/offline Course and Multiple Lecturers Scheduling Using Meta-Heuristic Approaches - Proceedings of the 2023 9th International Conference on Computer Technology Applications". In: ICCTA '23, pp. 166–171. doi: 10.1145/3605423.3605440. url: https://doi.org/10.1145/3605423.3605440.

Romaguera, Dexter et al. (2024). "Development of a Web-based Course Timetabling System based on an Enhanced Genetic Algorithm". In: *Procedia Computer Science* 234, pp. 1714–1721.

Sakal, James, Jonathan E. Fieldsend, and Edward Keedwell (2021). "Learning assignment order in an ant colony optimiser for the university course timetabling problem - Proceedings of the Genetic and Evolutionary Computation Conference Companion". In: GECCO '21, pp. 77–78. doi: 10.1145/3449726.3459534. url: https://doi.org/10.1145/3449726.3459534.

Saptarini, Ni Gusti Ayu Harry, Putu Indah Ciptayani, and Ida Bagus Irawan Pumama (Feb. 2020). "A Custom-based Crossover Technique in Genetic Algorithm for Course Scheduling Problem". In: *TEM JOURNAL-TECHNOLOGY EDUCATION MANAGEMENT INFORMATICS* 9.1, pp. 386–392. issn: 2217-8309. doi: 10.18421/TEM91-53.

Schneider, David, Michael Leuschel, and Tobias Witt (Sept. 2018). "Model-based problem solving for university timetable validation and improvement". In: *Form. Asp. Comput.* 30.5, pp. 545–569. issn: 0934-5043. doi: 10.1007/s00165-018-0461-7. url: https://doi.org/10.1007/s00165-018-0461-7.

Shao, Xiaorui, Su Yeon Lee, and Chang Soo Kim (Apr. 2024). "A Novel and Effective University Course Scheduler Using Adaptive Parallel Tabu Search and Simulated Annealing". In: *KSII TRANSACTIONS ON INTERNET AND INFORMATION SYSTEMS* 18.4, pp. 843–859. issn: 1976-7277. doi: 10.3837/tiis.2024.04.002.

Shuguang, LIU and BA Lin (June 2019). "Research on Complex Curriculum Arrangement Problem Based on Novel Quantum Genetic Evolutionary Algorithm - 2019 Chinese Control And Decision Conference (CCDC)". In: pp. 3783–3787. issn: 1948-9447. doi: 10.1109/CCDC.2019.8832728.

Tan, Joo Siang et al. (2021). "A survey of the state-of-the-art of optimisation methodologies in school timetabling problems". In: *Expert Systems with Applications* 165, p. 113943.

Teoh, Chong Keat, Antoni Wibowo, and Mohd Salihin Ngadiman (June 2015). "Review of state of the art for metaheuristic techniques in Academic Scheduling Problems". In: *ARTIFICIAL INTELLIGENCE REVIEW* 44.1, pp. 1–21. issn: 0269-2821. doi: 10.1007/s10462-013-9399-6.

Thepphakorn, Thatchai and Pupong Pongcharoen (July 2023). "Modified and hybridised bi-objective firefly algorithms for university course scheduling". In: *SOFT COMPUTING* 27.14, pp. 9735–9772. issn: 1432-7643. doi: 10.1007/s00500-022-07810-5.

Tung Ngo, Son et al. (2021). "A Genetic Algorithm for Multi-Objective Optimization in Complex Course Timetabling - Proceedings of the 2021 10th International Conference on Software and Computer Applications". In: ICSCA '21, pp. 229–237. doi: 10.1145/3457784.3457821. url: https://doi.org/10.1145/3457784.3457821.

Vázquez-Ingelmo, Andrea, Alicia García-Holgado, and Francisco J. García-Peñalvo (2020). "C4 model in a Software Engineering subject to ease the comprehension of UML and the software". In: *2020 IEEE Global Engineering Education Conference (EDUCON)*, pp. 919–924. doi: 10.1109/EDUCON45650.2020.9125335.

Yang, Yanming, Weituan Wu, and Yue Teng (2016). "An Intelligent Course Scheduling System of Military Academy Based on Improved Genetic Algorithm - PROCEEDINGS OF 2016 INTERNATIONAL CONFERENCE ON MODELING, SIMULATION AND OPTIMIZATION TECHNOLOGIES AND APPLICATIONS (MSOTA2016)". In: ACSR-Advances in Comptuer Science Research 58. Ed. by A Petrillo, B Acherjee, and K Weller, pp. 303–306. issn: 2352-538X.

Yazdani, Mehdi, Bahman Naderi, and Esmaeil Zeinali (Sept. 2017). "ALGORITHMS FOR UNIVERSITY COURSE SCHEDULING PROBLEMS". In: *TEHNICKI VJESNIK-TECHNICAL GAZETTE* 24.2, pp. 241–247. issn: 1330-3651. doi: 10.17559/TV-20130918133247.

Zheng, Huijun et al. (June 2022). "Course scheduling algorithm based on improved binary cuckoo search". In: *JOURNAL OF SUPERCOMPUTING* 78.9, pp. 11895–11920. issn: 0920-8542. doi: 10.1007/s11227-022-04341-6.