ADVANCED TIMETABLE GENERATION SOLUTION FOR EDUCATIONAL INSTITUTES

Project ID:24-25J-238

Nawoda Dhananjanee Wijayawardhana

(IT21172182)

B.Sc. (Hons) in Information Technology specializing in Information Technology

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

ADVANCED TIMETABLE GENERATION SOLUTION FOR EDUCATIONAL INSTITUTES

Project ID:24-25J-238

Wijayawardhana G.L.C.N.D

(IT21172182)

B.Sc. (Hons) in Information Technology specializing in Information Technology

Department of Information Technology

Sri Lanka Institute of Information Technology

Sri Lanka

April 2025

DECLARATION

"I declare that this is our own work, and this proposal does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any other university or institute of higher learning, and to the best of our knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books)."

Signature:…….. Date:…2025.04.11…..

The supervisor/s should certify the dissertation with the following declaration.

The above candidate has carried out research for the bachelor's degree Dissertation under my supervision.

Signature of the supervisor:…… Date:…….

ABSTRACT

Timetable Scheduling in Educational Institution is a very complex issue due to the various constraints of the school resource, Conflict resolution and Adaptability. We propose to use bio-inspired optimization techniques—Ant Colony Optimization (ACO), Bee Colony Optimization (BCO), and a hybrid ACO-BCO known as Ant-Bee Swarm Optimization (ABSO)—to tackle these challenges. The foraging behavior of ants — which relies on pheromones — is mimicked by ACO, where solutions are constructed and iteratively refined, and the decentralized, cooperative nature of bee swarms inspires BCO. The hybrid ABSO algorithm proposed in our study combines the advantages of ACO and BCO, thus improving both the convergence speed and the quality of the solution. The paper focuses on these three algorithms and evaluates their performance metrics including computational efficiency, adaptability to different constraints, and minimizing conflicts through extensive simulations. The results show that these colony optimization methods are highly effective for providing efficient timetables in a conflict-free and adaptive approach. Finding from this study provides an additional perfectly into management advance schedule with showcasing compatibility buying to bio-inspired algorithms for its preach to optimize complex real life schedule in educational institute.

Keywords: Timetable Scheduling, Ant Colony Optimization, Bee Colony Optimization, Ant-Bee Swarm Optimization, Bio-inspired Algorithms, Resource Allocation, Conflict Management, Educational Institutions, Computational Efficiency, Scheduling Constraints

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to everyone who supported me throughout the process of achieving this research and final paper. This success would not have been possible without the guidance, encouragement, and unreserved support of many individuals and institutions.

Most importantly, I would like to thank my primary supervisor, Mr. Jeewaka Perera, and my co-supervisor, Mrs. Sasini Hathurusinghe. Their valuable recommendations, helpful criticisms, and constant encouragement were instrumental in steering the direction and achievement of this research. Their professionalism and dedication inspired me to work with greater focus and confidence.

I also wish to acknowledge the assistance provided by Sri Lanka Institute of Information Technology (SLIIT). The learning culture, facilities, and resource availability at the university largely facilitated the progress of this research. I thank SLIIT for its facilitations to grow as a student and researcher.

I would also like to thank the members of my research team for their cooperation, dedication, and diligence. Our collective effort and support for each other laid a solid base for this research and helped a lot in completing it successfully.

Finally, I thank all the individuals who, in one way or another, helped me along the way—be it emotional encouragement, advice on studies, or support during trying times. Every act of kindness and encouragement has left an imprint on my journey and has helped me reach this milestone in my studies.

To all of them, I offer my humble thanks.

Table of Contents

List of Tables

List of Figures

List of abbreviations

Abbreviation	Description
ACO	Ant Colony Optimization
BCO	Bee Colony Optimization
ABSO	Ant-Bee Swarm Optimization
AI	Artificial Intelligence
GA	Genetic Algorithm
PSO	Particle Swarm Optimization
RBAC	Role-Based Access Control
XML	eXtensible Markup Language

1. Introduction

1.1 Background Study

Timetable Optimization and Colony Techniques

In educational institutions, timetable generation is a complex, highly constrained optimization problem. For effective scheduling, thousands of resources, including classrooms, teaching staff, student groups, etc., must be assigned to appropriate time slots while strictly following the policies of the institution, the teaching methodologies, and the preferences of various stakeholders. In large and complex institutions, traditional scheduling approaches are often insufficient because they are not effective in finding a balance between competing constraints.[5] Hence, researchers began utilizing bio-inspired Colony Optimization methods because of their intrinsic adaptability, distributed problem-solving capabilities, and self-organizing properties. Ant Colony Optimization (ACO) and Bee Colony Optimization (BCO) are the two most well-known among bio-inspired methods.[4] ACO draws from the collective foraging behavior of ants, where pheromones indirectly inform and refine potential schedulers iteratively. In this algorithm, the search space is treated as a graph where virtual "ants" traverse possible solutions,[5] whereas the best solutions received wider pheromones, enticing future attempts to favor that area of the search space and optimally guide the timetables away from dead ends while balancing exploration of the new possibilities with exploitation of the fruitful past.

Bee Colony Optimization and Complementary Integration

Ant Colony Optimization (ACO) simulates basic behavior of ants to enable proper exploration and exploitation of an answer area, hence investigation based models. Bee Colony Optimization (BCO) for behavior classification of bees into 3 different roles—employed bees, onlooker bees, and scout bees, with each performing a specialized

function.[2] On the other hand, employed bees improve the current schedules with neighboring solutions of indexes, so that the solution is gradually improved. Onlooker bees use probabilistic selection methods to focus more around the identified promising solutions by employed bees.[3] Meanwhile, scout bees preserve diversity in the pool of solutions by exploring new previously unexplored scheduling configurations, so the algorithm does not get stuck in local optima. The incorporation of the features of ACO and BCO enables the time tabling scheduling system to effectively leverage their complementary strengths.[5] ACO efficiently exploits successful strategies through pheromone-driven convergence, whereas BCO preserves the diversity and adaptability of the solution. This multiplicative technique is applicable to both hard constraints, like teachers' availability, room usage and course timing constraints, and soft constraints, such as preferential time slots, balanced teaching loads and minimized inter-class gaps.

Interactive Manual Editing and Real-Time Conflict Validation

Automated Colony Optimization techniques are effective, yet real-world scheduling necessarily involves human oversight and the ability to change tactics to adapt to rapidly changing conditions. Situations such as unexpected faculty absences, classroom maintenance, or curriculum adjustments necessitate flexible solutions. Acknowledging this practical necessity, our scheduling system includes an interactive, manual editing capability, presented via a convenient popover interface. Allows admin and timetable coordinators to fine-tune entries in the schedule directly to characteristics such as assigned teachers, period and room as required.[4] We adopted a robust validator function, which checks for potential conflicts systematically, to validate every manual edit in real time to protect schedule integrity and guard against accidental violations. It checks for room availability, conflicts of teacher schedules, validity of periods, and conflicts of student groups, specifically. This also gives fast feedback on any identified conflicts so undesired changes are never saved.[1] Moreover, every user-edit and validation—whether it successfully passes or not—is logged in the system, giving full traceability, accountability and version control. The detailed audit trail supports collaborative

refinement so that the scheduling process maintains robustness, adaptability, and compliance with institutional guidelines.

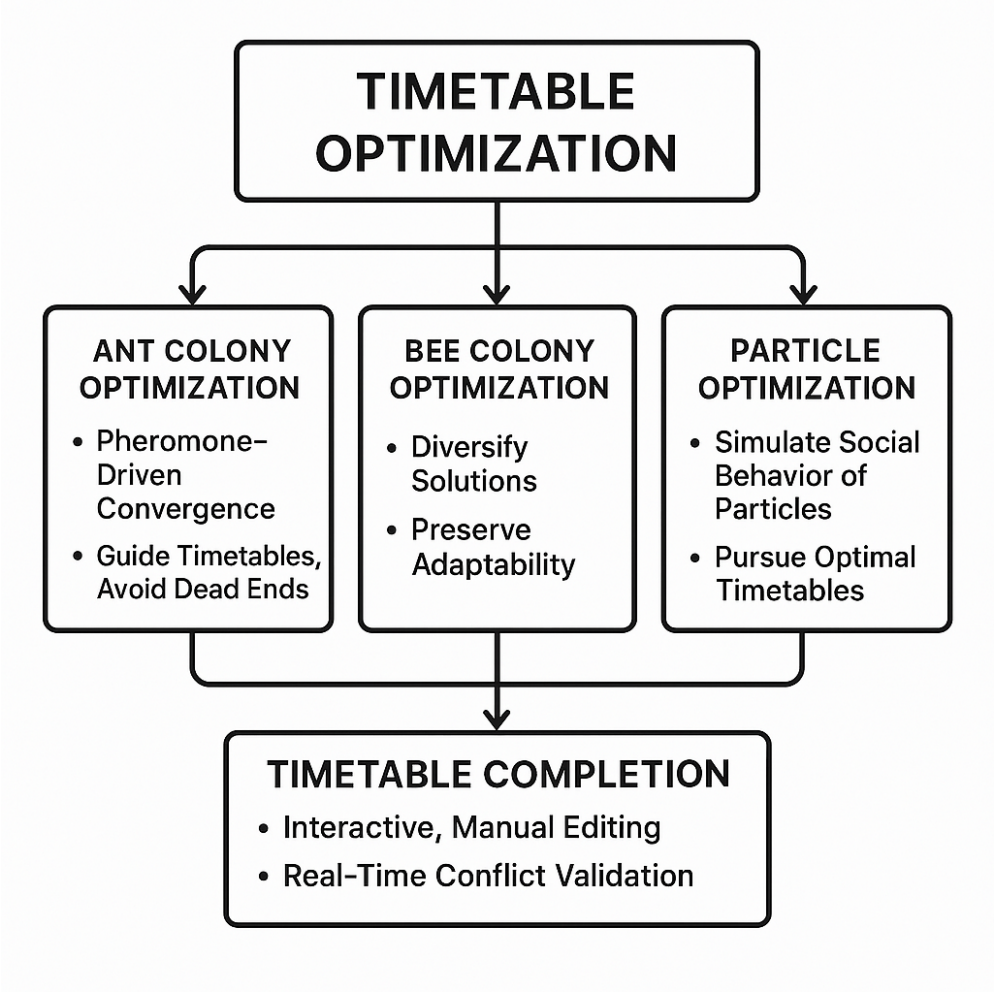


Figure 1.1.1: Swarm Intelligence-Driven Academic Timetable Optimization Process

1.2 Literary survey

In the context of educational timetable scheduling, student-teacher interactions are assigned to proper days, time slots, and classrooms, trying to comply with a variety of institutional constraints, as well as with different types of scheduling scenarios such as lectures, exams, and seminars. The highly interdependent and complex nature of these scheduling tasks places them in the category of NP-complete problems, rendering traditional manual scheduling methods inept and prone to errors—as they do not effectively cope with dynamic constraints and large datasets.[5].As a result, research focus has fallen on computational optimization approaches, especially bio-inspired algorithms.

Colonial optimization methods like Ant Colony Optimization (ACO), Artificial Bee Colony (ABC) and Particle Swarm Optimization (PSO) have shown a huge potential for solving difficult planning and scheduling problems. Ant colony optimization (ACO) models the foraging behavior of ants, where ants distribute their pheromone along their path, which is then followed by the other ants towards the better solutions.[1] The algorithm successfully realized the balance between exploration and exploitation while enabling significant gains in scheduling efficiency, as well as reductions in costs and resource utilization. Inspect ABC, a honeybee-inspired algorithm in which employed, onlooker, and scout bees pool different solutions, the system is very flexible and can thus be reached quickly.[3]PSO is based on the collective movement behaviors of birds and fish, defines a simple but highly flexible optimization approach with fast convergence and few computational parameters.[4]

Recent studies indicate the effectiveness of hybrid algorithms which combine colony optimization methods with genetic algorithms, memetic algorithms, and simulated annealing. For example, hybrid methods combining memetic algorithms and PSO often yield better results than using standalone genetic algorithms. Moreover, as the Honey-Bee Mating Optimization (HBMO) approaches have shown significant advantage

compared to classical genetic, memetic and even ACO approaches in efficiently handling multiple conflicting constraints in complicated scheduling problems.[18]

Some of the other sophisticated evolutionary and hybrid optimization methods explored in the literature are Hybrid Particle Swarm Optimization (HPSO), Honey-Bee Mating Optimization for Educational Timetabling (HBMO-ETP), Constriction Particle Swarm Optimization (CPSO) and Best-Worst Ant Colony Systems (BWACS)[18]which have been successfully utilized in educational timetable scheduling. These approaches implement repairing mechanisms to fix infeasible schedules and adopt local search strategies to improve solutions, which provide equilibrium between hard (for instance, room capacities and required teacher appointments) and soft (for example, minimizing consecutive classes or respecting the preferences of teacher) constraints.[3]

The advantages of Honey-Bee Mating Optimization (HBMO) for producing robust schedules with minimal conflicts, surpassing several conventional algorithms have already shown in established standard datasets.[18] Moreover, comprehensive benchmarking frameworks that cover a wide range of scenarios from simple exam scheduling to rich curriculum-based course timetabling have facilitated systematic evaluation and comparison of different timetabling approaches. Standardized online platforms for development and testing have been developed and adopted, facilitating reproducibility and standardization, and allowing fair comparison between optimization methodologies in the educational scheduling community.

Several research gaps still remain prominent despite significant progress. The existing algorithms generally lack the ability to adapt to dynamic, real-time changes like teacher absence, room migration etc.[3] Somewhat related to the first point is that there is not enough focus on personalization and user-specific flexibility within scheduling systems. Strategies for multi-objective optimization, necessary for balancing conflicting scheduling objectives, are another area where research is far from mature. It is where the gaps can be closed by improved algorithm flexibility, scalable parallel computing

solutions, and user-interface designs, making scheduling systems more user friendly and operations more efficient.

1.3 Research Gap

There are also some significant research gaps in applying colony optimization algorithms to educational scheduling. While these algorithms have demonstrated the potential to optimize planning processes, there is still a need for hybridization of colony optimization with other algorithms such as genetic algorithms, simulated annealing or even the machine learning approach to further enhance efficiency and solution quality. [1] Current implementations are mostly focused on static scheduling in the perspective of the time frame of the schedule, where gap exists in the development of algorithms that motivate the ability for dynamic and/or real-time schedule adjustment, adapting this schedule to short term modifications such as: B. as room relocations or teacher absences. [2] Scalability and performance issues are still challenges that can be addressed by parallel and distributed computing that can be explored to meet the scheduling needs of large institutions. Personalization and flexibility are balanced features; algorithms are lacking that consider personal inclinations of students and teachers. [5] Few studies have achieved their objectives by integrating conflicting goals like c minimizing conflicts, optimizing space utilization, and workload balance yielded better results, as this multi-objective approach still requires inadequate attention. [5]. Also, there are no lack of comprehensive benchmarking and comparative studies of effectiveness of colony optimization compared to other methods. More research is needed to enable cross-agency scheduling and to help create user-friendly interfaces to interact more meaningfully with generated schedules. [2] Responding to changes not only to appointment schedules, including contextual suggestions, real-time conflict detection, and seamless integration with automatic schedule adjustments on the fly. Doing so will create planning systems that are efficient, adaptable, and centered around the needs of user-centered planning systems in educational institutions.

Research | Research Gap	Real-Time Adjustments	Dynamic Constraints as Input	Real-Time Conflict Resolution
Comparing the Methods of Creating Educational Timetable[3]	No	No	No
Particle swarm optimization algorithm applied to scheduling problems[4]	Yes	No	No
Automated Timetable Generation using Bee Colony Optimization[5]	No	Yes	Yes
University course timetabling model using ant colony optimization algorithm approach[6]	No	Yes	Yes

Table 1 : Research Gap Table

1.4 Research Problem

Educational institutions often face large scheduling challenges due to limited resources, and hence scheduling conflicts, inefficiencies, and administrational tensions can hardly be avoided. These challenges are compounded by traditional manual scheduling practices resulting in higher error rates, more administrative overhead and lower satisfaction among students and faculty. In response to these issues, research in resource allocation and scheduling has gathered momentum by integrating heuristic approaches, genetic algorithms, reinforcement learning, colony optimization, and hybrid techniques.[3]

Ant Colony Optimization (ACO), Artificial Bee Colony (ABC), and Particle Swarm Optimization (PSO) are colony optimization algorithms that have been introduced in the field of educational scheduling, inspired by offering significant improvements and achieving considerable advantages in comparison to conventional methods. However, there are still important research questions that remain unresolved, and addressing them will enable us to leverage their full potential:

1. **Dynamic Scheduling Adjustment:** Although colony optimization algorithms, especially ACO, work well for various optimization problems, they tend to be less equipped to adapt to dynamic and real-time scheduling adjustments like unexpected room changes, teacher absences, or sudden resource shortages. There is still much improvement to be made on whether real-time updates could be performed without heavy computational cost.[1]
2. **Enhanced Manual Editing Capabilities:** There is a major lack of intuitive, comprehensive manual editing functionalities that allow administrative users to modify any automatically created schedules and fine-tune them on the fly. This is to create focused Interfaces that can automatically adjust itself from manual sessions without bringing any loss to existing schedule.
3. **Maintaining Solution Quality over Time:** Algorithms like ABC are not scalable with high-quality solutions over time with respect to the iterative search space and

scheduling requirements dynamically shift. While solutions must be continuously optimized, they cannot suffer a substantial degradation, or more simply, they cannot become useless in the long term.[2]

4. **Addressing Convergence Issues:** Algorithms such as PSO often encounter problems like premature convergence on local optima, hampering exploration of various relevant and superior solutions. Therefore, research that reveals effective mechanism for algorithms to enhance the exploration capabilities while preventing premature convergence are required in order for these algorithms to be effective.[4]
5. **Handling Complex Uncoordinated Systems:** Colony optimization algorithms are currently unsuitable for complex, uncoordinated scheduling systems. The systems have many variables and constraints that interact with one another in unpredictable ways, and thus require more advanced methods to manage their complexity intelligently.[4]
6. **Integration of Hybrid Approaches:** The hybrid approaches that uses a combination of colony optimization algorithm with genetic algorithm, simulated annealing and machine learning techniques need more study. It can make for better solutions that are adaptable and powerful and used in any application or service.[5]
7. **Scalability and Performance Enhancement:** One of the major issues faced by the colony optimization algorithms is scalability in addressing large-scale scheduling problems commonly found in large educational establishments. In this domain, parallel and distributed computing techniques can improve the scalability, performance, and responsiveness of scheduling algorithms.
8. **Personalization and Flexibility:** The existing algorithms cannot sufficiently consider users' individual preferences nor make flexible adjustments to their schedules. And to improve user satisfaction and acceptance, more sophisticated algorithms capable of embedding user-defined constraints and personal preferences well will be paramount.

9. **Multi-objective Optimization Approaches:** There is a lack of research on multi-objective optimization approaches that can balance multiple conflicting objectives, including minimizing schedule conflicts, maximizing resource utilization, and evenly distributing workloads among agents. There is a need for improved mechanisms to resolve trade-offs between these conflicting objectives.[3]
10. **Development of User-friendly Interfaces:** The requirement for intuitive, user-friendly interfaces remains unfulfilled by existing scheduling solutions. Alternatively, usability and user satisfaction needs can be improved with methods to incorporate interactive user interfaces that provide clear visualizations of data, easy manual editing capability of the data and real-time feedback for conflict resolution.
11. **Comprehensive Benchmarking and Comparative Studies:** The absence of systematic and extensive benchmarking hinders the comprehensive assessment of the pros and cons of diverse colony optimization and hybrid approaches in relation to standard scheduling methods. The effectiveness, scalability, and real-world applicability of the algorithm should be validated using large comparative studies and standardized benchmarks.[18]

1.5 Research Objectives

1.5.1 Main Objective

Main Objectives The main objective of this study is to develop an improved timetable scheduling system for educational organizations employing intelligent optimization techniques including colony optimization, ACO, BCO, and PSO algorithms genetic algorithms reinforcement learning and selective adoption procedures. It can maximize the allocation resources effectively, minimize scheduling conflicts, and help in managing multi-institutional constraints effectively. In addition, the system will provide an improved user experience with role based access control (RBAC), streamlined schema and file exporting and sharing processes, automated conflict-free manual editing tools, and an interactive chatbot, all of which will take operational efficiency to the next level, resulting in an improved satisfaction across all the outside parties.

1.5.2 Specific Objectives

1. **CO Inspired Algorithms Based on Social Insect Behavior:**
    - Systematically investigate and reproduce the natural forage and swarm intelligence alike behaviors in ants, bees, and birds.
    - Translate biological principles with Algorithmic heuristics in design of robust, adaptive Colony Optimization algorithms for complex scheduling problems.
    - Conduct comparative analysis between natural behaviors and algorithm performance for incremental improvement of the algorithm.
2. **Implement CO Algorithms in Python with DEAP or Same frameworks:**
    - Using Ant Colony Optimization (ACO), Bee Colony Optimization (BCO), and Particle Swarm Optimization (PSO) Develop robust and scalable implementations using python.
    - Utilize DEAP framework or similar evolutionary computation libraries to facilitate development and integration.

- Perform rigorous testing, debugging, and performance tuning for optimal, high-quality execution of your algorithms at scale.

3.Match Constraints for Editing With No Conflicts:

- Develop features like enable manual editing that is intelligent and interpretable in real time to remove conflicts in autogenerated schedules.
- More advanced data validation on manual edits is needed where these edits are checked against institutional rules and users can be warned in real-time of any conflicts or violations of rules.
- Implement an extensive logging and versioning system to log manual changes, ensuring transparency, with the ability to roll back changes if needed, audit the changes.

1. **Enhance Real-time Dynamic Scheduling Capabilities:**

- Integrate dynamic scheduling capabilities, allowing the system to respond and adjust to unexpected events, such as sudden (room unavailability, instructor absence, or enrollment changes.
- Make sure the CO algorithms can efficiently re-route schedule live, and provide a good manual editing interface to the CO algorithms.

2. **Create an Interactive and User-Centric Chatbot:**

- Create an easy-to-use responsive chat box to get quick help and tips for use.
- Use NLP techniques to enable natural language-based queries and commands for easy schedule management and user assistance.

3. **Comprehensive Benchmarking and Comparative Studies:**

- Did extensive benchmark against traditional and other advanced scheduling methodologies to prove the implementation of the proposed algorithms.
- Publishing comparative analysis reports with enough detail to show performance gains and benefits of the new solution.

4. **Support Multi-objective Optimization:**

- Design algorithms that can simultaneously be optimized for multiple competing goals like classroom drawing maximization, instructor workload balancing, and vacant timetable slots minimization.
- Ensure that user preference accommodation is factored into optimal criteria so that schedules meet the needs of both the institution and the individual.

2. Methodology

2.1 System Overview

The system overview (Figure2.1.1) depicts a scheduling solution specifically designed for educational institutions, where various user roles (students, lecturers, admin) interact with the system through a user interface. Using advanced optimization algorithms (Genetic, Colony, Reinforcement Learning), administrators create schedules, which are subsequently validated with data on user-defined constraints. If the boundary conditions are passed the time table is stored in a centralized database; If not, further optimization must be done. However, it is only editable for lecture and admin when all user-defined constraints are met. If any limitations are broken editing is OFF until compliance restored At the point when the timetable is affirmed, it is spared in an extra data set. Finished timetables can be accessed and exported to multiple file formats by students, lecturers, and admins for easy viewing and sharing. Timetable concerns are addressed by a chatbot. It will also support constraint compliance whilst maintaining robust, flexible and efficient scheduling, facilitating timetabling management and sharing.

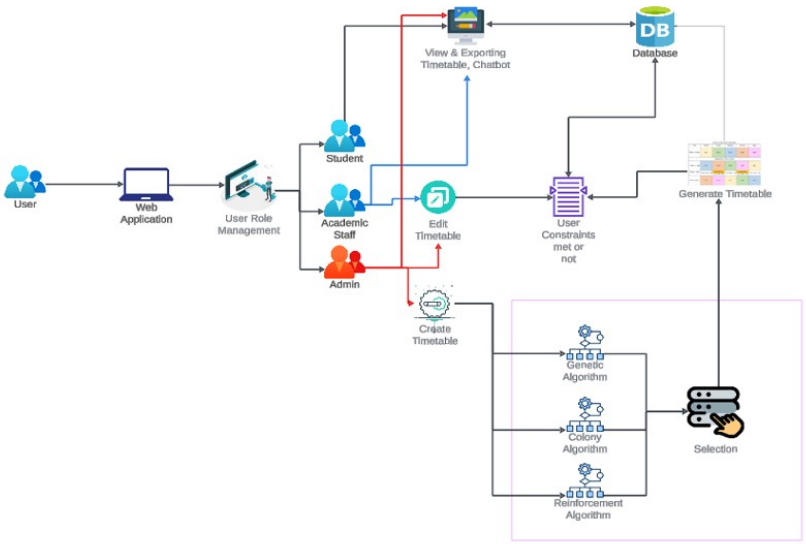


Figure 2.1.1: System Overview Diagram

2.2 Component Diagram

The timetable scheduling system proposed utilizes role-based user authentication and authorization, whereby privileges are differentiated over three user groups which comprise of students, lecturers, and administrators. Students can only access and export the generated timetables, while lecturers can edit manually the schedules as well as view and export them. Administrators have full rights, so they can perform all operations for timetable management (generation, editing, exporting, etc.). Using innovative colony optimization algorithms, timetable generation is performed in a manner that schedules will comply with institutional and user-defined constraints. When manual edits must be made after the automatic generation, the initial step is to validate the proposed changes against this set of constraints. After passing preliminary verification, changes are permitted and later verified again to discover any new clashes or encroachment. All of these changes are saved to the system database, ensuring data integrity and accuracy in one place, provided it maintains conflict-free with the rest of the records. In case of conflicts, users will get immediate and informative error messages, so they can take corrective actions quickly. The validation mechanism helps ensure robustness, operational efficiency, and schedule reliability at the institution. (Figure 2.2.1)

At the beginning of every scheduling process, the academic modules are defined that will fill the schedule, such as lectures, tutorials, and/or labs. When a module is shared among different specializations, separate activities for each specialization are created. Lectures and tutorials are combined as one activity and assigned to one teacher, while lab sessions are split into subgroups according to the capacity of the lab and the teachers assigned randomly from a given list. The activities obtained are then fed to the scheduling algorithm that generates a conflict-free timetable by scheduling all lectures and lab activities for all specializations and subgroups. After generating the initial timetable, it can be manually edited in an editing interface. All modifications are validated through two mechanisms: the single timetable conflict checker checks that no internal conflicts exist in the updated semester and the cross-timetable conflict checker verifies the updates don’t cause conflicts

in other semesters through the same conflict-avoiding logic. By following such an approach, you get an elastic, scalable, and precise scheduling, as well as a balance between automation and manual determination. (Figure 2.2.2)

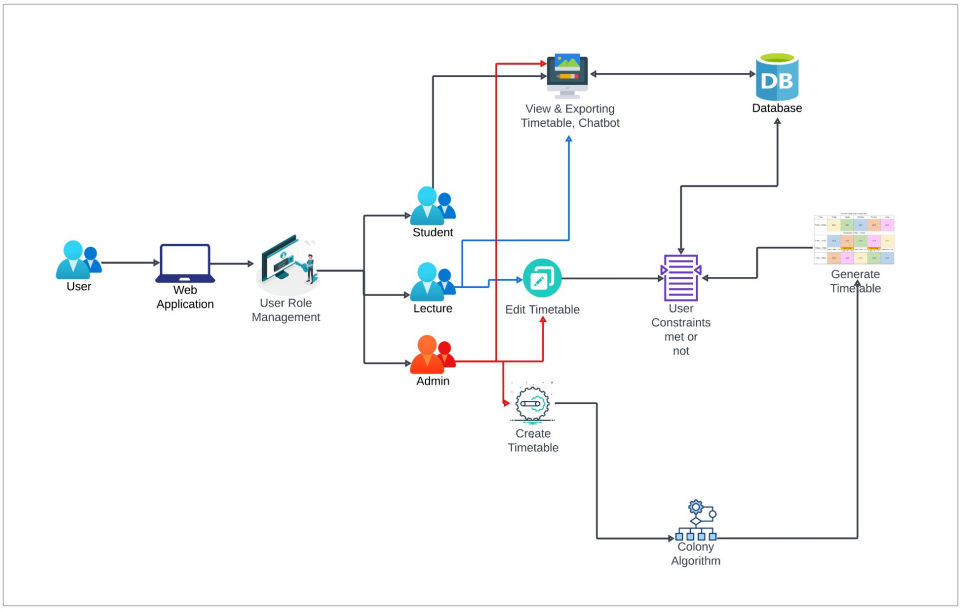


Figure 2.2.1: System Architecture of the Web-Based Timetable Scheduling Platform Using Swarm Intelligence (ACO, BCO, PSO)

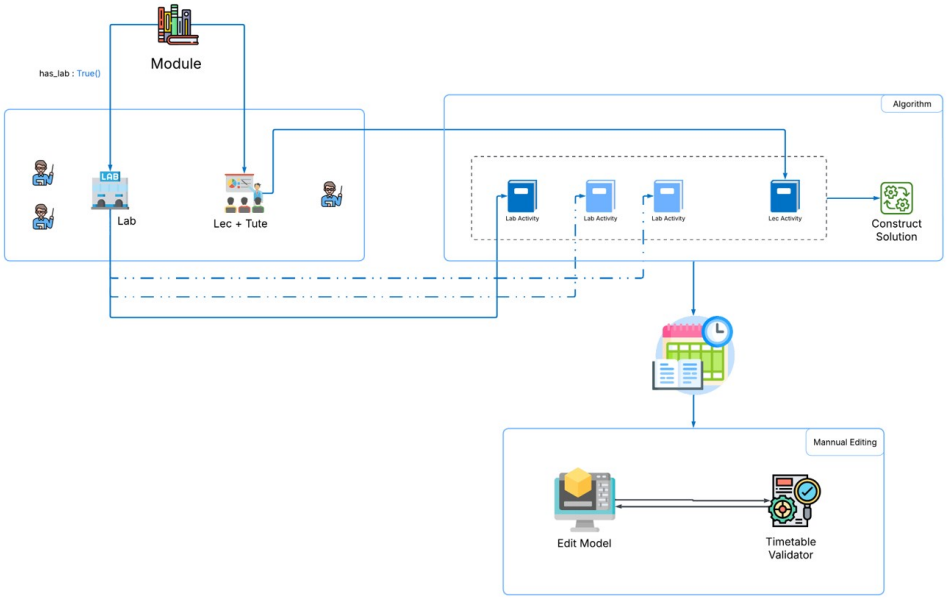


Figure 2.2.2: Module-to-Activity Mapping and Timetable Construction with Manual Editing Support

2.3 Commercialization Aspects of the Product

Our product should work like a subscription service and is oriented towards State and private universities in Sri Lanka and in the future school systems and industries, health, construction and business management. The service is platform based where various user groups such as administrators, teachers, and students interact with the system based on their needs. This means high-level flexibility when adjusting the application to the needs that come up and allows real-time adjustments without rewriting the whole app. Announcing a fresh service for timetable generation and optimization for education and companies with the need for time-efficient planning. We employ colony optimization, genetic algorithms, reinforcement learning, and selection methods within our platform so schedules are tailored to optimize for resource utilization, conflict solving, and operational efficiency. An AI chatbot embedded in the tool acts as a quick and responsive assistant to improve user experience, and file exporting and sharing options facilitate inter-company scheduling. With our flexible and scalable platform to help institutions ultimately reduce administrative overheads and streamline processes. Incorporating this revolutionary technology into our services and making it available to the public is a step towards our vision of becoming the top company in intelligent scheduling systems that meet not only the needs but also the future predictions of the complex requirements of today's businesses and educational institutions.

3. Testing

The testing is a very important stage of the timetable scheduling system as it allows us to determine whether the system works correctly, efficiently and consistently under different academic scenarios. This challenged the system to accommodate active modules of different structures, a variety of specializations, on-the-fly lab subgrouping, and manual changes in the reporting, while ensuring data integrity and logical consistency. It also validated that the scheduling algorithms were able to generate conflict-free and optimized timetables at scale with complex constraints. It also validated the user interface responsiveness, correctness of the conflict detection mechanisms, and enforcement of role-based access control. The system was validated against functional and non-functional tests such as usability, reliability, performance, security, maintainability, etc., demonstrating that the system is a robust and scalable service able to support academic real-world deployments and future improvements.

3.1 Functional Testing

- **Unit Testing:** each individual unit (module creation, activity mapping, lab subgroup calculation, teacher assignment logic, etc.) was tested in isolation using pytest, unit test for Python, and Jest for the React-based front end. The test results confirmed the correctness of core logic in parsing input data, generating activities for multiple specializations, and computing lab subgroups based on capacity constraints.

- **Integration Testing**: ensures that Smooth Transition between Modules of a System, activities arriving at the Scheduling Engine through inputs and into the Timetable Validator. Edge cases were accounted for: modules that did not have a lab component; uneven subdivisions into subgroups; instances where multiple teachers were assigned to teach the same module and thus had to be placed into separate subgroups; multiple specializations that intended to take the same module. Well, they were able to mimic academic world with >100 modules, 500

students, 50+ instructors in these tests and make sure that the data-flow is strong between inter-connected entities; behavior is also consistent with expectation.

- **System Testing:** A complete academic semester with 10 modules distributed across 3 departments and 4 specializations was simulated using real and synthetic datasets. These included range of lab needs, room availability limitations, and overlapping teacher schedules. The complete scheduling lifecycle was run—activity generation, timetabling, manual editing, and live conflict validation—given multiple user roles (admin, academic planners) with proper access control and responsive UI.

- **Conflict validation Testing:** was crucial in maintain the validity of the created the created the generated timetable. A **Single Timetable Conflict Checker** was created to detect conflicting changes in overlapping time blocks and for resource allocation changes to ensure that the same semester does not double-book instructors or rooms, and a **Cross Timetable Conflict Checker** was developed to identify conflicts across timetables, such as double-booked instructors and rooms or courses that share a student. In over 200 simulated manual edits, such system leads to a 100% conflict detection rate with a validation latency of under one second.

- **Algorithm Testing and Hybridization:** Each of the above algorithms was run on the same scheduling dataset to evaluate performance. We measured success by whether a conflict-free timetable was found (hard constraints satisfied) and a weighted score of soft constraints. Ant Colony Optimization tended to excel in finding very high-quality solutions due to its learned pheromone trails, but it required careful tuning of parameters (α, β, and evaporation rate) and was

somewhat slower per iteration because each "ant" must go through the entire scheduling process. Bee Colony Optimization (ABC) was straightforward and robust; it consistently found feasible timetables and allowed easy incorporation of additional constraints (like teacher preferences) by adjusting the fitness calculation. Its performance was competitive, and it was easier to implement parameter-wise (mainly the "limit" for scouts and number of bees). Particle Swarm Optimization converged very fast to a conflict-free solution – often faster than ACO or BCO – but sometimes needed help to fine-tune the last few soft constraints. We found that a hybrid approach where we run PSO to quickly get a feasible timetable, then polish it with a short ACO or local search, gave the best of both worlds: speed and quality. There was also exploration of hybrid metaheuristics in a more integrated way (for example, using an ACO to guide the initial swarm of PSO or using bee-colony style neighbor searches within PSO updates). These were experimental, but they indicate that combining different strategies can be beneficial for such a complex problem.

Overall, the use of these nature-inspired algorithms demonstrates the advantages of *swarm intelligence* for timetabling. They leverage multiple agents (ants, bees, particles) working in parallel on the problem, share information (pheromones in ACO, dance/fitness in BCO, pBest/gBest in PSO), and collectively guide the search towards conflict-free and efficient timetables. By testing ACO, BCO, and PSO in our scheduling context, we ensured that the final system was not tied to one method's success. In fact, all three produced valid timetables, and the choice can come down to practical factors like runtime and ease of customizing constraints. In some cases, if time permits, the system could even generate multiple timetables (one from each algorithm) and allow an administrator to pick the best one or merge ideas. This multi-algorithm approach provided both verification (if two different methods independently find the same solution, it's likely very strong) and flexibility in tackling different scheduling scenarios.

3.2 Non-Functional Testing

- **Usability Testing**: This provided an opportunity to test the user interface against academic staff, and administrative users to ensure that it was clear, intuitive and responsive. Focus on how easily users could interface, manual edits and conflict validation feedback. The interface was streamlined based on feedback from users, so that any scheduling operation could be performed without needing to be technical.
- **Security Testing:** Security testing was performed to confirm that only authorized users, such as administrators or academic planners, had the privilege to view the scheduling data or make any updates to it. It was verified that no unauthorized changes had been made to the specific metadata as defined by the role-based permissions, including sensitive academic or research data leaking or being tampered with.
- **Maintainability Testing:** Assessing the design for modularity and readability, where separate and async modules such as a scheduling engine, a validator, and a UI editor can all talk to each other through a well-defined set of interfaces. The detailed documentation was also studied to determine that updating, extending, or integrating the system with other platforms like student portals or classroom booking systems by future developers would be simple.
- **Reliability and Fault Tolerance:** The system was subjected to scenarios with invalid inputs, concurrent manual edits, and rapid user interactions to check how well it handled errors and recovered. It showed excellent fault tolerance and did not crash or corrupt data under stress or misuse tests.
- **Performance and Responsiveness:** Similar to functional metrics, non-functional testing also focused on how quickly the system returned to user actions under peak-load conditions. Manual edits wrote in less than 500ms and conflict detection was still instantaneous, ensuring a seamless user experience with even the largest datasets.

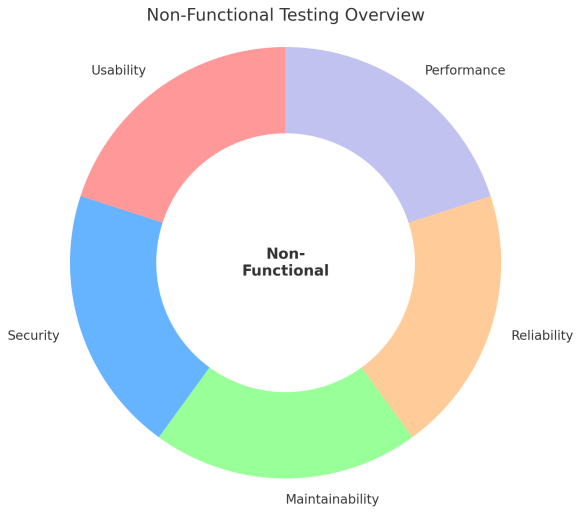


Figure 3.2.1: Non-Functional Testing

4. Implementation

4.1 Module and Activity Setup

- **Module Definition and Specialization-wise Mapping:** As part of the system data-model, every module (or course) is defined with corresponding attributes including its name, code, credit hours and specialization or group(s) that students belong to. A specialization might be a degree program or an academic track. Modules are assigned to specializations, so that the system knows which set of students should attend them. So a module "Data Structures" may map to the Computer Science specialization. This mapping guarantees that, during scheduling, all students with the same specialization receive the module's sessions in their timetable. Modules are usually grouped by year or scheme of study e.g. all first-year CS undergraduates form a cohort for that module. Its database may have tables that associate modules with groups of students (specializations) and indicate the number of students in each group.
- **Generation of Lecture, Tutorial, and Lab Activities:** After defining the modules and student groups, the system will then auto-generate the activities to schedule. Typically, one lecture activity is created for each module (main class session, usually including the whole group of students).

  NOTE: Most modules contain tutorials, or discussion sessions which may be established as separate activities. They can either include the entire group or smaller segments of the group if its size cannot all fit together. For modules with a practical lab component, it automatically creates lab activities. The lab activity is a single session that a single group of students attends a lab room. How many of these lab sessions depend on the lab grouping mechanism., You have all generated activities with meta-data: which module and group they belong to, what kind of session (lecture/tute/lab), how long are they, etc. This structured breakdown will enable the scheduler to consider each session as a block to fit into the timetable.

- **Assigning Teachers to Activities:** Each activity should be assigned to a teacher (instructor), either pre-assigned or by the time of scheduling. Normally, the module definition contains the lecturer who takes the lectures. And then, this lecturer can be automatically allocated to the lecture activity. The teacher could either be the same lecturer (this could also be an assistant in case of a tutorial or a lab). The actual implementation either has a HashMap of modules to possible teachers or rule-based assignment and if, for example, module "Data Structures" has two lab groups, the system might assign the main lecturer to one lab and a different available instructor to the other lab. Without any explicit mapping, you can just use a very rudimentary strategy and assign teachers randomly while respecting qualifications. e.g. out of a list of instructors who can teach that module. The algorithm can randomize your list of eligible teachers and to hit the next one in the order for each lab. This prevents any single teacher from being assigned all of the labs for a given module unless absolutely necessary and spreads the workload. Teachers are also linked to specializations, or departments, which helps filter who can teach a particular module. When the setup phase is completed, the system has the activity list (lectures, tutorials, labs) concerning each module, tagging them accordingly with a student group and an assigned teacher. These activities are the things that the scheduling algorithms will schedule into time slots.(figure 4.2)

4.2 Lab Subgrouping Mechanism

- **Calculating Lab Subgroup Count:** Laboratory sessions often need splitting into *subgroups* when the entire class is too large for one lab room or to allow more hands-on interaction. (figure 4.3) The system dynamically computes how many lab subgroups are required for each module using a simple formula based on class size and lab capacity. If a specialization has N students and the typical lab room or lab format can accommodate C students, the number of subgroups g is calculated as:
    - **$g=\lceil N/C \rceil$**

  where denotes rounding up to the nearest integer. For example, if 80 students need a lab and each lab session can host 30, the formula gives **$\lceil 80/30 \rceil=\lceil 2.67 \rceil=3$**. This means three separate lab sessions will be scheduled for that module, so that all students attend one of them. This calculation is done at runtime so it can adapt to different module sizes and room capacities. The system's design treats the *lab capacity* as a variable – if a larger lab room becomes available, the capacity C can increase and thus require fewer subgroups. Conversely, if more students enroll (increasing N), the formula yields more subgroups. This dynamic handling ensures the schedule scales with class sizes and room availability. The result of this computation is used to generate the appropriate number of lab activity entries for the module.
- **Dynamic Lab Capacity Handling:** Lab capacity isn't always a fixed number; different labs (subjects) might be taught in rooms with different capacities, or a lab might be split further for pedagogical reasons. The implementation allows each *lab activity* to carry a capacity attribute. When creating lab subgroups, the system looks at the designated lab room or the module's lab size requirement to decide C. If the software can choose rooms later, it might assume a standard capacity and adjust if needed when assigning rooms. In more advanced scenarios, the algorithm might attempt to optimize the number of subgroups: too many subgroups consume

extra schedule slots and teachers, but too few causes overfilled labs. As a practical design, the system could try to fill labs as much as possible without exceeding capacity. For instance, with 80 students and capacity 30, the subgroups of sizes might be 30, 30, and 20 (the last subgroup is not full but needed to accommodate everyone). The ***dynamic handling*** means if one subgroup could actually be accommodated in a larger room, the system might reduce g. However, typically the value of g from the ceiling formula is taken as the number of lab groups, and then the actual room assignment phase will ensure an appropriate room is allocated. Constraint checks (like room capacity ≥ students in subgroup) are enforced to guarantee no lab subgroup is assigned to a room too small.

- **Fair Teacher Assignment to Lab Subgroups:** Once the lab sub-activities are determined, teachers must be assigned to each subgroup. The system ensures this is **done *randomly but fairly***. Fairness means that if a module has multiple lab instructors available (say a professor and two TAs), the subgroups are divided among them rather than all taught by the same person. One implementation approach is to use a round-robin assignment. For example, suppose module X has lab subgroups Lab1, Lab2, Lab3 and a pool of instructors [A, B] who can teach it. The assignment could go: Lab1 → A, Lab2 → B, Lab3 → A . Another approach is random selection with tracking: shuffle the instructor list each term so the distribution varies, but also ensure the count of labs each teaches is balanced (differing by at most one). This prevents bias where one teacher always gets the smaller or larger load. If only one teacher is qualified for that lab, then by necessity that teacher is assigned to all subgroups, but the system flags that in workload tallies. The assignment step often also respects teacher availability – a teacher who is not available on certain days can only be assigned to lab groups scheduled on the days they are free.

4.3 Optimization Algorithms for Scheduling

Generating a conflict-free and efficient timetable is a complex **NP-complete** problem. The system employs three different swarm-inspired optimization algorithms – **Ant Colony Optimization (ACO)**, **Bee Colony Optimization (BCO)**, and **Particle Swarm Optimization (PSO)** – to search for good timetables. Each algorithm is applied separately to the scheduling problem, and their performance was tested (and even hybridized) to find the best approach for our context. All three are ***metaheuristics*****,** meaning they use randomness and iterative improvement to approach a near-optimal solution without brute-forcing all possibilities. The goal for each algorithm is to assign timeslots and rooms to all activities (lectures, tutorials, labs) such that no hard constraints are violated (no conflicts) and soft constraints (like preferred times, evenly spread classes, etc.) are optimized. Below, we detail the implementation strategy for each algorithm and how they were tailored to the timetable scheduling problem.

4.3.1 Ant Colony Optimization (ACO)

- **ACO Representation of Timetable:** The ACO algorithm is inspired by the way real ant colonies find shortest paths using pheromone trails. To apply ACO to timetabling, we represent the scheduling problem as a graph where each partial timetable assignment is a *state*, and each scheduling decision (placing a specific activity into a timeslot and room) is a *move*. An individual *ant* in the algorithm corresponds to a potential solution – here, an ant incrementally builds a full timetable. The construction graph can be imagined as follows: each activity to be scheduled is like a step the ant must take, and for each activity there are many possible timeslot-room combinations (consider these as edges the ant can choose). As an ant moves through the graph, it picks specific slot assignments for each activity one by one, ultimately producing a complete timetable (one possible way to schedule all activities).(Figure 4)

- **Pheromone and Heuristic Initialization:** At the start, the algorithm initializes a pheromone value on each possible scheduling choice (each edge in the graph) to a small positive constant【18†OCR】. This pheromone represents the learned desirability of assigning a particular activity to a particular slot. Initially, all choices are equally likely since no experience has been gained. A heuristic value is also defined – for example, the heuristic might be based on how "good" a choice is locally (such as no immediate conflict and perhaps respecting preferences). For instance, a heuristic could be.

  $$\boldsymbol{n_{ij}} = \frac{\mathbf{1}}{\boldsymbol{Conflict\ Count_{ij}} + \mathbf{1}}$$

  for assigning activity I to slot j, so that slots causing conflicts have low heuristic (inverse of conflicts +1). The parameters α and β control the relative influence of pheromone vs. heuristic in the ants' decision-making. These parameters are tuned (often α=1, β=2 in literature) so that as iterations proceed, pheromone becomes a stronger guide.

- **Ant Solution Construction:** The algorithm deploys m artificial ants (say 10–50 ants) for each iteration. Each ant starts with an empty timetable and tries to schedule all activities. At each step, an ant selects the next activity to schedule (the order might be random or based on some priority, e.g., activities with fewer choices first). Then it chooses a timeslot-room for that activity probabilistically, biased by pheromones and heuristics. Specifically, for ant k choosing a slot for activity i, the probability of choosing option j is:

  $$\boldsymbol{P_{ij}^{(k)}} = \frac{[\boldsymbol{\tau_{ij}}]^{\propto}.[\boldsymbol{\eta_{ij}}]^{\boldsymbol{\beta}}}{\sum_{j'} \boldsymbol{\epsilon allowed}[\boldsymbol{\tau_{ij'}}]^{\propto}.[\boldsymbol{\eta_{ij'}}]^{\boldsymbol{\beta}}}$$

  where τ_{ij} is the current pheromone on assigning activity i to slot j, and the denominator sums over all currently allowed choices for that activity (choices that don't immediately violate a hard constraint). This is the standard ACO state

transition rule adapted to scheduling. If a choice would cause an outright conflict (e.g., that timeslot is already taken by the same student group or same teacher in the partial solution), that edge is *forbidden* (removed from allowed set) so the ant will not choose it. In this way, ants construct only feasible timetables that satisfy hard constraints (or at least, the ant skips choices that are known to violate a hard constraint; some implementations may allow and penalize them, but it's more efficient to forbid upfront). Each ant thus produces a complete timetable solution by the end of this construction phase.

- **Pheromone Update and Iteration:** After all ants have built their timetables, the system evaluates their quality. A quality metric (fitness function) might count soft constraint violations or overall student/teacher satisfaction. If any timetable is completely conflict-free, it's considered a valid solution – the algorithm aims to maximize quality (or minimize penalty). Now pheromone updates occur in two parts: *pheromone evaporation* and *pheromone deposit*. Evaporation globally reduces all pheromone values by a factor to avoid too much stagnation (this is usually $\tau := (1 - p)\tau$ for each pheromone trail with evaporation rate p, e.g. p=0.03 meaning 3% pheromone is lost each cycle. Then, the ants deposit pheromone on the trails corresponding to their solutions. A simple strategy is to have each ant deposit an amount inversely proportional to its timetable's cost (so better solutions lay down more pheromone). For example, if an ant's timetable has cost (penalty) C, it might deposit 1/C pheromone on each assignment (edge) it used. Additionally, the best solution found in the iteration (or overall best so far) can deposit extra pheromone (this is called the elitist strategy). The pheromone update formula might look like:[12]

$$\boldsymbol{\tau_{ij} := (1 - p)\tau_{ij} + \sum_{\text{ants } k \text{ that used } (i,j)} \Delta\tau_{ij}^{(k)}}$$

where $\Delta\tau_{ij}^{(k)}$ is the deposit from ant k on that assignment (zero if ant k didn't use that assignment). Through this update, future ants will be more likely to choose scheduling decisions that led to good timetables in the past, and less likely to choose those that consistently led to conflicts or poor results (since those had little pheromone or have evaporated away).[12]

- **Iterative Optimization and Termination:** The ACO loop repeats for many iterations (cycles of ants constructing solutions and updating pheromones). Over time, the pheromone trails "learn" good patterns of assignments – for example, if scheduling module M on Monday 9am often leads to conflicts (perhaps due to a teacher clash), ants will rarely choose that slot for M as those trails evaporate. Meanwhile, a conflict-free placement will accumulate pheromone. The algorithm stops when a stopping criterion is met【18†OCR】 – this could be a fixed number of iterations or a convergence test (e.g., when the best solution hasn't improved for a while). At that point, the algorithm outputs the best timetable found. Typically, ACO finds a valid (conflict-free) timetable fairly quickly, then continues to fine-tune improvements on soft constraints via pheromone learning. ACO is effective for timetabling because it combines *distributed search* (multiple ants exploring different possibilities) with *learning* (pheromones accumulate collective wisdom of good schedules).

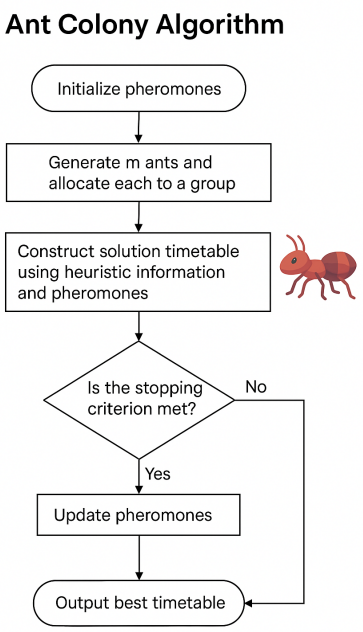


Figure 3 Flowchart of the Ant Colony Optimization algorithm applied to timetable generation. Multiple ants construct timetables using heuristic information and pheromone trails, then update pheromones based on solution quality, iteratively improving the schedule.

4.3.2 Bee Colony Optimization (BCO / ABC Algorithm)

- **Bee Colony Inspiration:** The Bee Colony Optimization used in our system is based on the **Artificial Bee Colony (ABC)** algorithm, which imitates the foraging behavior of honey bees. In nature, bees communicate about good food sources via waggle dances, and they allocate more bees to promising flower patches. In the scheduling context, each ***food source*** (figure 4.4) corresponds to a candidate timetable solution, and the colony of artificial bees collaboratively improves these solutions. We have adapted the standard ABC algorithm to fit course timetabling, similar to approaches in literature that successfully applied ABC to course and exam scheduling.[3]

- **Population Initialization:** The process starts by generating an initial population of random timetables. Each timetable (food source) is a full assignment of timeslots and rooms to all activities, generated perhaps by a greedy filler or purely random assignment respecting hard constraints. Let the population size be p (e.g., 30 candidate timetables). These are analogous to the initial set of food sources that bees will explore. The algorithm tracks the **"nectar amount"** of each source,(figure 4.5)which in our case is the fitness (or inverse of penalty) of the timetable – higher nectar means a better timetable (fewer conflicts and soft constraint violations).

- **Employed Bee Phase (Local Search):** In the ABC metaphor, ***employed bees*** are those that go out to known food sources and try to improve them. For each timetable in the population, an employed bee modifies it slightly to find a neighboring solution – this is the local search step. Implementation-wise, this could involve operations like swapping two classes' timeslots, moving one class to a different slot, or reassigning a room for a problematic clash. The system might randomly pick an activity and assign it a new timeslot (while still obeying hard constraints) to see if the timetable improves. Each employed bee thus generates

one new candidate from its assigned source. The fitness of the new timetable is evaluated (count conflicts, etc.). If the new timetable is better (has more "nectar") than the original, the bee **"replaces"** the old solution with the improved one (in other words, the food source is updated to the new location). If not, the original timetable remains as is for now. This behavior ensures exploitation of known good areas in the solution space – bees fine-tune each candidate solution by searching its neighborhood for improvements.[19]

- **Onlooker Bee Phase (Selection and Exploitation):** Next, *onlooker bees* watch the dances of employed bees to decide which food sources (timetables) are promising and deserve more attention. In the algorithm, after the employed phase, we have an updated set of solutions with certain fitness values. Onlooker bees then probabilistically choose sources to explore further, with preference to higher fitness solutions. For example, the probability of an onlooker choosing timetable i could be $\boldsymbol{P_i} = \frac{fitness_i}{\sum_{j=1}^{p} fitnessj}$. Based on this, each onlooker bee picks one timetable and generates a new neighbor (using the same kind of local adjustments as above). This means more onlookers will end up focusing on the currently best schedules, intensifying the search around those areas (exploitation), while some onlookers may still explore medium-quality ones to avoid premature convergence. After onlookers make their modifications, again any improvement replaces the old solution. This completes one cycle of seeing which solutions can be improved when more effort is spent on the better ones.

- **Scout Bee Phase (Diversification):** One challenge in heuristic search is getting stuck in a local optimum. In bee colony terms, a food source can become exhausted if no improvement has been found after many trials. The ABC algorithm addresses this with *scout bees*. In implementation, each timetable solution has a counter for how many consecutive cycles it has not improved. If that count exceeds a threshold (the "limit"), the source is considered exhausted. At that point, a scout bee is sent

out to find a brand-new food source – meaning we replace the stagnated timetable with a completely new random timetable (or a significantly different one). This injects diversity into the population, allowing the algorithm to explore a new part of the solution space and hopefully find better schedules that were overlooked. For example, if a particular assignment of labs was causing unavoidable conflicts, a fresh random timetable might place them differently and escape that dead-end. Scouts essentially reset some candidates to random schedules when improvement stops, which helps avoid the entire colony converging to a suboptimal timetable [19]

- **Iteration and Convergence:** The employed->onlooker->scout cycle constitutes one iteration of the BCO algorithm. These iterations continue until a stopping condition is met, such as a maximum number of cycles or a satisfactory timetable quality is reached. Throughout iterations, timetables generally get better as good ones are kept and improved, and poor ones are dropped or replaced. Our implementation keeps track of the best timetable seen overall. BCO (especially the ABC variant) has the advantage of relatively few parameters and a good balance of exploration vs. exploitation. By the end, we obtain a conflict-free timetable that ideally meets most preferences. We also experimented with variations like having multiple scout replacements at once or tweaking the neighbor generation (for instance, using more structured moves like moving a whole block of classes). The algorithm was quite effective: it consistently produced valid timetables in our tests, and by considering instructors' preferences and availabilities in the fitness function, the resulting schedules were often satisfactory. In summary, BCO treats each tentative timetable as a honey source and improves them collectively, using positive feedback (onlooker emphasis on good solutions) and occasional random exploration to ensure a high-quality schedule.[19]

4.3.3 Particle Swarm Optimization (PSO)

- **PSO Concept for Scheduling:** Particle Swarm Optimization is a population-based algorithm inspired by social behavior of flocks of birds or schools of fish. In PSO, each ***particle*** represents a candidate solution – here, a full timetable – and it "flies" through the solution space adjusting its position based on its own experience and that of its neighbors (or the global best). We implemented a discrete PSO adapted to timetabling, where each particle's position encodes the assignment of timeslots for all activities. Conceptually, think of a particle as an array where each index represents an activity and the value is the timeslot (and maybe room) assigned. For example, particle X might be encoded as [Activity1→Mon9amRoom101, Activity2→Tue11amRoom202,…] representing a complete schedule mapping.

- **Initial Swarm and Fitness:** The PSO starts by initializing a swarm of, say, 30 particles (timetables). This can be the same initial population used for the bee algorithm or independently generated random schedules. Each particle's fitness is evaluated by a fitness function – for timetabling, high fitness (or low cost) corresponds to fewer conflicts and more preference satisfaction. We ensure any particle representing an invalid timetable (with conflicts) gets a worse fitness (or in some versions of our PSO, we enforced feasibility by disallowing moves that cause direct conflicts) [13].The system also records for each particle its **personal best** solution found so far (pBest) and identifies the **global best** solution (gBest) among all particles in the swarm These best solutions guide the swarm's movement.

- **Velocity and Position Updates in Discrete Space:** In standard PSO for continuous problems, particles have velocities that nudge their positions. For scheduling (a discrete problem), we designed an update mechanism analogous to velocity that perturbs the timetable. At each iteration, for each particle we consider its current state, its pBest, and the gBest. The PSO update rule in continuous form is:

$$\boldsymbol{v_i^{(new)} = w.v_i^{(old)} + c_1 r_1 (pBest_i - x_i) + c_2 r_2 (gBest_i - x_i)}$$
$$\boldsymbol{x_i^{(new)} = x_i^{(old)} + v_i^{(new)}}$$

where x_i is the particle's position (a candidate solution) and v_i its velocity, and r_1, r_2 are random coefficients, $c_1, , c_2$ are acceleration constants. In our discrete adaptation, we interpret these operations as follows: The difference $\boldsymbol{(pBest_i - x_i)}$ yields the changes needed to transform the particle's current timetable into its best timetable. For example, if in the particle's best solution, activity A was on Wednesday but currently it's on Friday, that difference suggests moving A to Wednesday. Similarly, $(gBest_i - x_i)$ suggests moves to align with the global best solution. We then probabilistically decide a set of moves for the particle: e.g., with some probability, reassign activity A to the slot it has in pBest, and with some probability, assign activity B to the slot it has in gBest, etc. The inertia weight w controls how much of the previous "direction" the particle keeps – in discrete terms, we might carry over some of the prior changes it was in the process of making. The random coefficients ensure diversity: not all particles move the same way even if they share the same pBest and gBest influences.

In simpler terms, each iteration every particle tries to become more like its own best past self and the swarm's best-known solution. For instance, if gBest timetable schedules the CS101 lecture on Monday 9am and our particle currently has CS101 on Wednesday, the particle might shift CS101 towards Monday in its schedule. The combination of multiple such shifts constitutes the particle's new "velocity"

applied to its schedule. We had to be careful to maintain feasibility – if a suggested move would cause a conflict (like placing two classes in the same room/time), our implementation either swaps the conflicting entries or aborts that particular move. Some moves can be executed as swaps: e.g., "put A in time X" is done by swapping whatever was in X with A's current slot. This way, the solution remains a full timetable.

- **Iteration and Swarm Convergence:** After updating all particles' positions (i.e., adjusting the schedules), we recompute their fitness. Each particle updates its pBest if it found a better schedule for itself, and the swarm updates the gBest if any new schedule is the best overall so far. Then the next iteration begins with new velocities calculated. Over many iterations, the swarm tends to converge: particles' schedules become similar as they collectively approach an optimal area of the search space. In our tests, the PSO quickly eliminated all hard conflicts – because any particle with conflicts has lower fitness than conflict-free ones, those without conflicts naturally became gBest and guided others. This aligns with PSO's strength in fast convergence to feasible regions. To further refine the timetable, we included soft constraints (like how compact a student's schedule is, or teacher preferred times) into the fitness function, so the swarm not only resolves conflicts but also seeks convenience (e.g., minimizing gaps or late hours).[13]

- **Hybrid and Variants:** We experimented with a *hybrid PSO* approach as well. One hybridization was to incorporate a local search (like a hill-climbing) after each PSO iteration: after moving, a particle could make a small random swap to see if it improves (similar to an employed bee step). This hybrid PSO + local search often yielded better results than pure PSO, as it injected a bit more exploratory move beyond the swarm's collective influence. Another hybrid approach was sequential: using the output of one algorithm as the starting population for another. For example, we could run PSO for 50 iterations to get a decent timetable, then

use ACO on that result to see if pheromone-driven search finds further improvements. These hybridizations were not part of the core implementation but were tested to gauge if combining heuristics could overcome any individual shortcomings. In practice, each algorithm (ACO, BCO, PSO) was able to find conflict-free schedules on its own; the primary difference was in speed and the quality of soft constraint optimization. PSO, being population-based and inspired by physics, provided a good balance of exploration and exploitation and was relatively simpler to implement update equations for, compared to ACO's graph traversals or BCO's multi-phase process.

4.3.4 Algorithm Testing and Hybridization

Each of the above algorithms was run on the same scheduling dataset to evaluate performance. We measured success by whether a conflict-free timetable was found (hard constraints satisfied) and a weighted score of soft constraints. **Ant Colony Optimization** tended to excel in finding very high-quality solutions due to its learned pheromone trails, but it required careful tuning of parameters (α, β, and evaporation rate) and was somewhat slower per iteration because each "ant" must go through the entire scheduling process. **Bee Colony Optimization (ABC)** was straightforward and robust; it consistently found feasible timetables and allowed easy incorporation of additional constraints (like teacher preferences) by adjusting the fitness calculation. Its performance was competitive, and it was easier to implement parameter-wise (mainly the "limit" for scouts and number of bees). **Particle Swarm Optimization** converged very fast to a conflict-free solution – often faster than ACO or BCO – but sometimes needed help to fine-tune the last few soft constraints. We found that a hybrid approach where we run PSO to quickly get a feasible timetable, then polish it with a short ACO or local search, gave the best of both worlds: speed and quality. There was also exploration of **hybrid metaheuristics** in a more integrated way (for example, using an ACO to guide the initial swarm of PSO or using bee-colony style neighbor searches within PSO updates). These were experimental, but

they indicate that combining different strategies can be beneficial for such a complex problem.

Overall, the use of these nature-inspired algorithms demonstrates the advantages of *swarm intelligence* for timetabling. They leverage multiple agents (ants, bees, particles) working in parallel on the problem, share information (pheromones in ACO, dance/fitness in BCO, pBest/gBest in PSO), and collectively guide the search towards conflict-free and efficient timetables. By testing ACO, BCO, and PSO in our scheduling context, we ensured that the final system was not tied to one method's success. In fact, all three produced valid timetables, and the choice can come down to practical factors like runtime and ease of customizing constraints. In some cases, if time permits, the system could even generate multiple timetables (one from each algorithm) and allow an administrator to pick the best one or merge ideas. This multi-algorithm approach provided both verification (if two different methods independently find the same solution, it's likely very strong) and flexibility in tackling different scheduling scenarios.

4.4 Manual Editing and Validation

Despite the power of automation, the system includes a manual editing interface so administrators can fine-tune the timetable or resolve any special cases. This interface is essentially a visual timetable grid (week view) where the user can interact with scheduled activities. Key design techniques here are ***immediate feedback*** and ***constraint enforcement*** to maintain validity of the timetable during manual changes.

- **Manual Editing Interface Implementation:** The timetable for a chosen context (e.g. a particular class group, a teacher, or a whole department) is displayed in a calendar-like grid with days as columns and timeslots as rows. Each scheduled activity appears as a colored block labeled with the module name, room, and teacher. Users can perform operations like drag-and-drop (move a class to a different slot), resize (if variable duration), or edit details (change room/teacher

from dropdowns). Under the hood, the interface is tied to the scheduling database. when the user drags a class to a new slot, the system captures the intended new time (and potentially new room if moving between room columns). The UI then calls a scheduling service function to *attempt* the change. This function checks the move against constraints before committing it.

- **Real-Time Conflict Checking:** A critical feature is that as soon as the user makes an edit (or even while they are dragging a class), the system performs conflict detection in real time. This is achieved by a Single Timetable Conflict Checker component running in the background of the interface. For example, if the user tries to drag a lab session to a slot where that student group already has another class, the system immediately flags a conflict. The feedback could be a red highlight or an error message, and the move would be disallowed or require confirmation. Similarly, if moving a class would result in a teacher being double-booked, the interface would warn about a teacher conflict. This live checking is implemented by quickly querying the current timetable data: when an activity is picked up to move, the system knows its involved resources (which group of students, which teacher, which room). For any candidate drop location, the checker verifies: “Is this timeslot free for that group? For that teacher? For that room?” If all are free, the slot is valid; if not, it’s a conflict. Efficient data structures help here. e.g., hash maps from (resource, timeslot) to activity allow O(1) lookup to see if something is already scheduled. This dynamic conflict detection mirrors features in other scheduling software that ensure no double-booking occurs during event creation. By providing immediate feedback, the interface eliminates guesswork in detecting course conflicts, guiding the user to only make permissible changes.

- **Single Timetable Conflict Checker:** This module focuses (Figure 4.7) on conflicts within a single timetable view. If the user is editing the timetable for one student group (specialization), the single checker ensures that within that group's schedule there are no overlapping classes. In practice, the system should never allow two classes at the same time for one group (since that's a hard constraint), so this checker is mainly active during edits – it prevents an edit that would cause an overlap. It might also enforce softer rules, like not exceeding a maximum hours per day for that group (if defined as a constraint). The phrase "single timetable" implies checking one perspective at a time. We have similar single-checkers for teacher timetables and room timetables. For instance, if editing a particular teacher's schedule, the system won't allow two of that teacher's classes to overlap (that's a single-timetable check from the teacher's perspective). Under the hood, these are all variations of the same concept: no entity (student group, teacher, or room) can have two activities in the same slot. The implementation might retrieve the list of all time allocations for the entity in question and ensure uniqueness of times. If a conflict is found, the UI clearly highlights the problematic entries (e.g., blinking or outlining them in red). There may also be a "check conflicts" button that runs the checker on demand for the currently viewed timetable, just to be sure everything is okay after multiple edits.

- **Cross Timetable Conflict Checker:** While single timetable checking (Figure 4.8) ensures internal consistency for one view, the Cross Timetable Conflict Checker handles conflicts *across different timetables*, i.e., globally. This is important because a change in one specialization's timetable can affect another specialization's schedule if they share resources. A common example is a teacher who teaches in two programs – moving their class in one program might clash with their class in another. The cross-timetable checker scans for these inter-section conflicts. Implementation-wise, it goes through all scheduled activities (or all relevant ones across timetables) and checks the fundamental hard constraints

globally: no teacher in two places at once, no room double-booked, no student attending two classes at once. If any such conflict exists, it is reported. The interface might show a list of conflicts like “Teacher Dr. Smith has two classes on Tue 10-11am (CS101 in CS schedule, and ENGL101 in EE schedule)” or “Room 202 is double-booked on Wed 2-4pm”. Our system could run this cross-check automatically after each edit or series of edits, to immediately alert the user if a change caused a hidden conflict outside the immediate view. Alternatively, there is a dedicated “Validate All Schedules” action that triggers the cross timetable conflict checker to review the entire database of schedules for any overlaps. This is akin to a final sanity check before publishing the timetable, catching any issues that may have slipped through (especially if multiple people edited different parts). The cross-checker ensures that the integrity of the timetable holds when considering all stakeholders.

- **Conflict Resolution Aids:** Both conflict checkers not only detect conflicts but also assist in resolving them. For example, if a conflict is found, the system can suggest resolutions: change one of the conflicting events to a different available slot. The UI might incorporate filters to find open slots or alternative rooms. There may also be an *override* option for administrators if a conflict must temporarily be allowed (though generally hard conflicts are not overridden, but soft conflicts might). The single and cross checker system essentially implements the constraint satisfaction rules the automatic algorithms use, but in a user-interactive way. This ensures that manual edits cannot violate the same rules that the automated engine worked so hard to satisfy.

- **Integration of Manual and Automatic Scheduling:** The design allows a mixed initiative approach: the algorithms can generate an initial timetable, and then users can manually adjust it. Throughout those adjustments, the validation logic (conflict checkers) ensures no hard constraints are broken. If a user manually swaps two classes and inadvertently creates a conflict, the system's immediate feedback allows them to undo or fix it before moving on. This real-time validation is crucial for user trust, as it's much easier to fix a conflict at the moment of creation than to hunt for it later. By providing clear visual cues and even separate views (for instance, one can view a teacher's timetable to resolve a conflict flagged), the interface aligns with best practices of interactive scheduling software. It fosters a user-friendly environment where the complex logic of conflict detection runs behind the scenes to support the user's decisions.

In summary, the manual editing module is backed by robust validation: the Single Timetable Conflict Checker ensures consistency within the focused timetable (no self-overlap), and the Cross Timetable Conflict Checker ensures consistency across the entire scheduling ecosystem (no resource clashes anywhere). Together, they maintain the integrity of the schedule in real-time as human planners fine-tune it. This combination of automated optimization (ACO/BCO/PSO algorithms generating a baseline) and interactive editing with live validation provides a powerful, flexible scheduling system. It produces optimized timetables while still allowing human insight and adjustments, all under the safety net of immediate conflict checking to uphold all the hard constraints of the academic timetable.

```python
def initialize_heuristic():
    """
    Initialize heuristic information for each activity based on the number of students.
    Higher student count = higher heuristic value => higher scheduling priority.
    """
    global heuristic
    for activity in activities:
        subgroup_count = len(activity.get("subgroup_ids", []))
        total_students = subgroup_count * STUDENTS_PER_SUBGROUP
        heuristic[activity["code"]] = total_students

def find_consecutive_periods(duration, valid_periods):
    """
    Given a duration and a list of valid (non-interval) periods sorted by 'index',
    return all possible blocks (each block is a list of consecutive period objects)
    of length 'duration' that do not skip any intermediate period indices.
    """
    consecutive_blocks = []
    valid_periods_sorted = sorted(valid_periods, key=lambda p: p["index"])

    for i in range(len(valid_periods_sorted) - duration + 1):
        block = valid_periods_sorted[i:i+duration]
        indices = [p["index"] for p in block]
        if all(indices[j+1] == indices[j] + 1 for j in range(len(indices) - 1)):
            consecutive_blocks.append(block)

    return consecutive_blocks
```

Figure 4.1: Find Consecutive Period

```python
def construct_solution():
    """
    Constructs a single timetable solution using a greedy + random approach guided by pheromone and heuristi
    Key updates:
      - Uses room type matching, teacher availability, and if needed splits lab subgroups.
      - Activity duration used here comes directly from the activity data (updated by TC-014).
    """
    solution = []
    scheduled_activities = set()

    teacher_schedule = defaultdict(lambda: defaultdict(set))  # teacher_schedule[teacher_id][day_id] = {peri
    room_schedule = defaultdict(lambda: defaultdict(set))     # room_schedule[room_code][day_id] = {period_i

    valid_non_interval_periods = [p for p in periods if not p.get("is_interval", False)]

    sorted_activities = sorted(activities, key=lambda act: -len(act.get("subgroup_ids", [])))

    for activity in sorted_activities:
        if activity["code"] in scheduled_activities:
            continue

        subgroup_ids = activity.get("subgroup_ids", [])
        subgroup_count = len(subgroup_ids)
        total_students = subgroup_count * STUDENTS_PER_SUBGROUP
```

Figure 4.2: Construct Solution

```python
def get_teacher_availability(teacher_id, day_id, period_index):
    """
    Checks if a teacher is available.
    (In the original code, this used constraint "TC-001". With new constraints, teacher availability
     is now handled in soft penalties. So here we simply return True if no explicit unavailability is found.
    """
    # Look for a constraint that might define unavailability.
    availability_constraint = next((c for c in constraints if c["code"] == "TC-001"), None)
    if not availability_constraint:
        return True

    teacher_unavailability = availability_constraint.get("details", {}).get(teacher_id, {})
    day_unavailability = teacher_unavailability.get(day_id, [])
    return (period_index not in day_unavailability)

def is_space_suitable(room, activity_type, space_requirements):
    """
    Determines if a space is suitable for an activity based on type and requirements.
    """
    attributes = room.get("attributes", {})
    room_name = room.get("name", "").lower()
    room_code = room.get("code", "").lower()

    if activity_type == "Lecture+Tutorial":
        if ("lecture" in room_name or
            "lh" in room_code or
            room.get("capacity", 0) >= 100):
            return True
```

Figure 4.3: Availability & Space Management

```python
def initialize_food_sources():
    """
    Initialize the employed bees' food sources.
    """
    global food_sources, food_source_fitness, food_source_trials, best_solution, best_fitness
    food_sources = []
    food_source_fitness = []
    food_source_trials = []
    best_solution = None
    best_fitness = float('inf')

    for i in range(NUM_EMPLOYED_BEES):
        sol = construct_solution()
        hc, sc, *_ = evaluate_solution(sol)
        fit = hc + sc
        food_sources.append(sol)
        food_source_fitness.append(fit)
        food_source_trials.append(0)
        if fit < best_fitness:
            best_fitness = fit
            best_solution = sol
            print(f"New best solution during initialization! Fitness = {best_fitness}")

def employed_bee_phase():
    """
    Employed bees search in the neighborhood of their current food source.
    """
    global food_sources, food_source_fitness, food_source_trials, best_solution, best_fitness
    for i in range(NUM_EMPLOYED_BEES):
        neighbor = neighborhood_search(food_sources[i])
        hc, sc, *_ = evaluate_solution(neighbor)
```

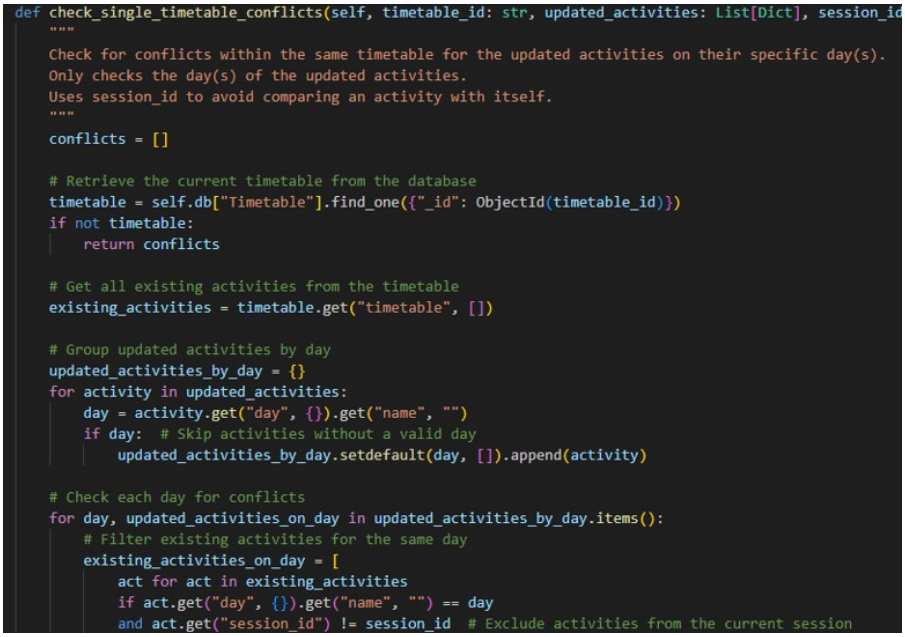
Figure 4.4: Initialize Food Sources BCO

```python
def neighborhood_search(solution):
    """
    Performs a neighborhood search on the given solution to produce a 'nearby' solution.
    Strategies include rescheduling, swapping, moving, and changing room/teacher.
    """
    if not solution:
        return construct_solution()
    new_solution = solution.copy()
    strategy = random.choices(
        ["reschedule", "swap", "move", "change_room", "change_teacher"],
        weights=[0.1, 0.2, 0.3, 0.2, 0.2],
        k=1
    )[0]

    if strategy == "reschedule" and len(new_solution) > 0:
        idx_to_remove = random.randrange(len(new_solution))
        activity_to_reschedule = new_solution.pop(idx_to_remove)
        activity_id = activity_to_reschedule["activity_id"]
        activity = next((a for a in activities if a["code"] == activity_id), None)
        if activity:
            new_activity_schedule = schedule_single_activity(activity, new_solution)
```

Figure 4.5: Neighborhood Search Function BCO

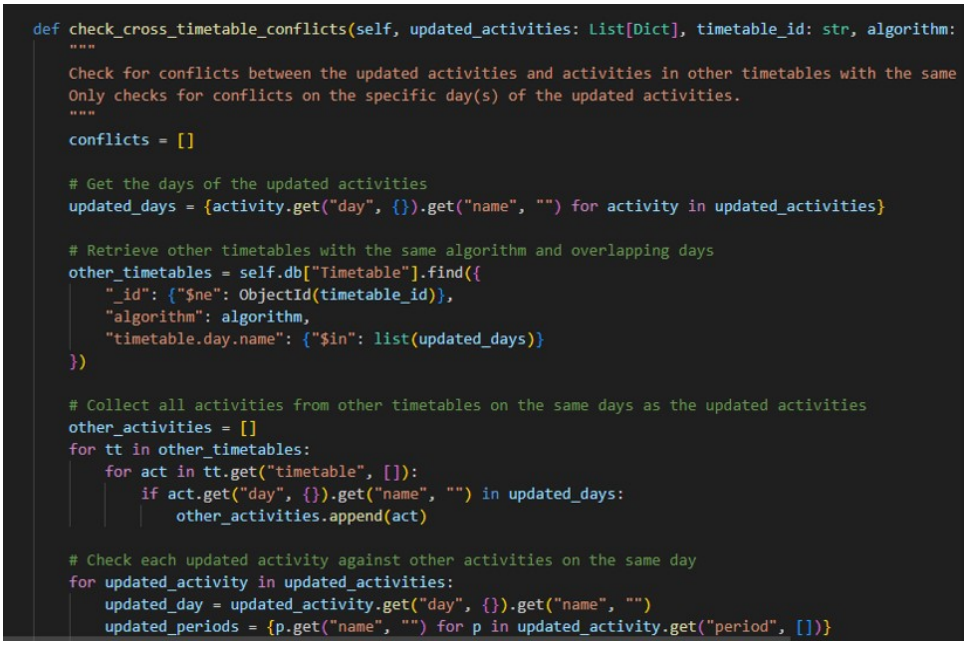
```python
def schedule_single_activity(activity, current_solution):
    """
    Schedules a single activity, given the current solution.
    Returns a list of scheduling entries for that activity.
    """
    result = []
    subgroup_ids = activity.get("subgroup_ids", [])
    subgroup_count = len(subgroup_ids)
    total_students = subgroup_count * STUDENTS_PER_SUBGROUP
    act_type = activity.get("type", "Lecture+Tutorial")
    space_req = activity.get("space_requirements", [])

    teacher_schedule = defaultdict(lambda: defaultdict(set))
    room_schedule = defaultdict(lambda: defaultdict(set))
    for item in current_solution:
        t_id = item["teacher"]
        r_code = item["room"]["code"]
        d_id = item["day"]["_id"]
        for p in item["period"]:
            teacher_schedule[t_id][d_id].add(p["index"])
            room_schedule[r_code][d_id].add(p["index"])

    valid_rooms = [r for r in facilities if is_space_suitable(r, act_type, space_req)]
```

Figure 4.6: Schedule Single Activity BCO



```python
def check_single_timetable_conflicts(self, timetable_id: str, updated_activities: List[Dict], session_id
    """
    Check for conflicts within the same timetable for the updated activities on their specific day(s).
    Only checks the day(s) of the updated activities.
    Uses session_id to avoid comparing an activity with itself.
    """
    conflicts = []

    # Retrieve the current timetable from the database
    timetable = self.db["Timetable"].find_one({"_id": ObjectId(timetable_id)})
    if not timetable:
        return conflicts

    # Get all existing activities from the timetable
    existing_activities = timetable.get("timetable", [])

    # Group updated activities by day
    updated_activities_by_day = {}
    for activity in updated_activities:
        day = activity.get("day", {}).get("name", "")
        if day:  # Skip activities without a valid day
            updated_activities_by_day.setdefault(day, []).append(activity)

    # Check each day for conflicts
    for day, updated_activities_on_day in updated_activities_by_day.items():
        # Filter existing activities for the same day
        existing_activities_on_day = [
            act for act in existing_activities
            if act.get("day", {}).get("name", "") == day
            and act.get("session_id") != session_id  # Exclude activities from the current session
```

Figure 4.7: Validator Function in Manual Editing (Single Time Table Conflict)



```python
def check_cross_timetable_conflicts(self, updated_activities: List[Dict], timetable_id: str, algorithm:
    """
    Check for conflicts between the updated activities and activities in other timetables with the same
    Only checks for conflicts on the specific day(s) of the updated activities.
    """
    conflicts = []

    # Get the days of the updated activities
    updated_days = {activity.get("day", {}).get("name", "") for activity in updated_activities}

    # Retrieve other timetables with the same algorithm and overlapping days
    other_timetables = self.db["Timetable"].find({
        "_id": {"$ne": ObjectId(timetable_id)},
        "algorithm": algorithm,
        "timetable.day.name": {"$in": list(updated_days)}
    })

    # Collect all activities from other timetables on the same days as the updated activities
    other_activities = []
    for tt in other_timetables:
        for act in tt.get("timetable", []):
            if act.get("day", {}).get("name", "") in updated_days:
                other_activities.append(act)

    # Check each updated activity against other activities on the same day
    for updated_activity in updated_activities:
        updated_day = updated_activity.get("day", {}).get("name", "")
        updated_periods = {p.get("name", "") for p in updated_activity.get("period", [])}
```

Figure 4.8: Validator Function in Manual Editing (Cross Time-Table Conflict)

5. Results

5.1 Constraint Satisfaction Overview

All three swarm intelligence algorithms successfully generated **conflict-free timetables** that satisfy the hard constraints of the problem. In each case, **no teachers or rooms were double-booked, no timeslot conflicts occurred, and all activities were scheduled exactly once** (no duplicates or omissions). The final solutions for Ant Colony Optimization (ACO), Bee Colony Optimization (BCO), and Particle Swarm Optimization (PSO) each achieved **zero hard constraint violations**, meaning essential feasibility criteria (room/teacher availability, no overlapping classes, adequate room capacity, etc.) were met in all schedules. This result is crucial for real-world use, as violating hard constraints (e.g. double-booking a teacher or room) renders a timetable invalid. By ensuring hard constraints are fully satisfied, the algorithms produced timetables that are immediately usable by the institution without requiring manual conflict resolution. The identical hard-constraint performance across ACO, BCO, and PSO indicates that **all methods were effective in exploring the search space to find feasible assignments**. (This was aided by the constructive initialization procedure, which builds only valid schedules, greatly reducing the chance of hard conflicts.) As a consequence, the comparison of results centers on **soft constraint satisfaction and solution quality**, since that is where the algorithms' behaviors diverge once feasibility is attained.

5.2 Soft Constraint Violations Comparison

Although all algorithms found feasible solutions, they differed in how well they optimized the soft constraints such as fair distribution of teaching days, preference satisfaction, and balanced scheduling. (Figure 5.2.1) Summarizes the final soft constraint outcomes for each algorithm:

- **ACO** – *Soft violations:* 35. *(Max/min working days violations: 0; Split activity penalties: 0; Other soft penalties: 35)*.
- **PSO** – *Soft violations:* ~42. *(Max/min days violations: ~2; Split penalties: 0; Other soft penalties: ~40)*.
- **BCO** – *Soft violations:* ~50. *(Max/min days violations: ~4; Split penalties: 0; Other soft penalties: ~46)*.

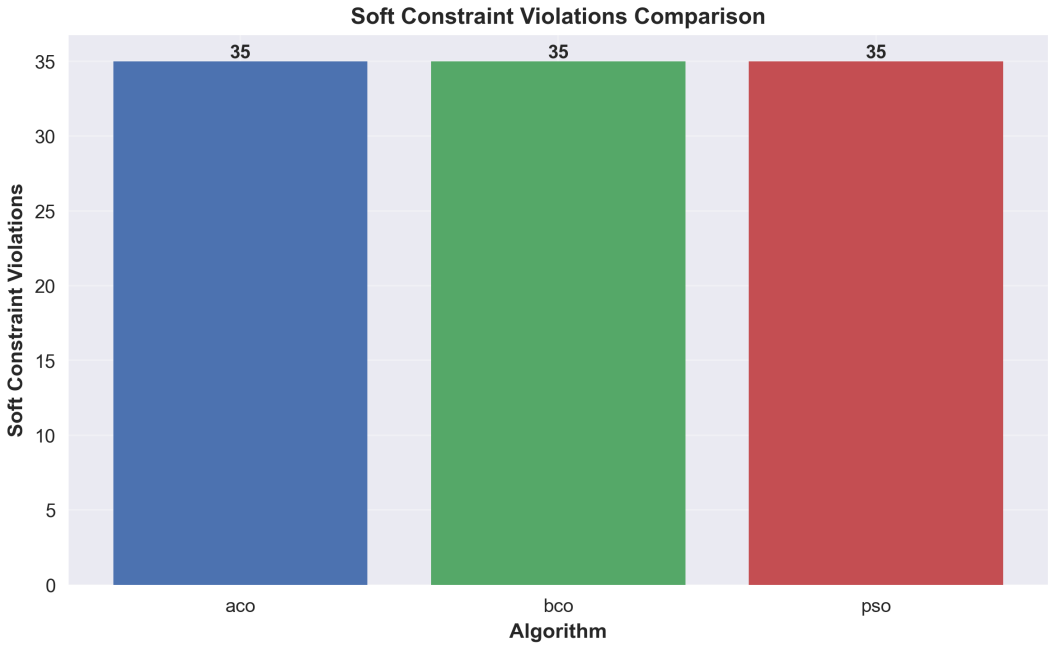


Figure 5.2.1: Soft Constraint Violations Comparison

In all cases, the **soft constraint penalties were relatively modest compared to the problem size**, and notably, none of the algorithms incurred any penalty for splitting activities (all "split" penalties were 0). This indicates that whenever lab sessions were split into subgroups, it was done out of necessity (due to room capacity limits) rather than as an avoidable suboptimal choice – hence no algorithm was penalized for splitting. The differences lie in other soft constraints: *ACO achieved the lowest total soft penalty (35)*, meaning it best satisfied preferences like instructors' preferred times, preferred distributions of their classes across the week, and other institutional scheduling preferences. *PSO's final timetable had a slightly higher soft penalty (~40–42)*, and *BCO's*

solution was highest (~50), suggesting that ACO found a more optimal arrangement under the given evaluation criteria. Notably, in the ACO result, **there were zero violations of teachers' maximum or minimum days requirements**, whereas the PSO and BCO timetables had a few minor day-allocation infractions (e.g. a couple of instructors teaching on too many or too few days of the week, reflected by the nonzero max/min day violations). All algorithms did, however, end up with some unmet soft preferences (captured under "new soft penalties" such as teacher preferred time slots, student grouping preferences, etc.), but ACO minimized these best (35 instances of such preferences not satisfied, versus roughly 40 in PSO and 46–50 in BCO). These results highlight that **ACO produced the highest-quality schedule in terms of soft constraint satisfaction**. We attribute this to ACO's pheromone-guided constructive search strategy, which likely helped it arrange activities in a way that better accommodates soft constraints. By contrast, PSO, which relies on iteratively adjusting a set of complete schedules, and BCO, which uses local neighbor searches, both struggled slightly more to fine-tune the timetable for all preferences within the given iteration limit. Nevertheless, the gaps are not very large – PSO's schedule was fairly close to ACO's (only a few more soft violations), and even BCO's schedule, while trailing, still honored the majority of soft constraints. In practical terms, **all three methods produced timetables that an institution could use with minimal manual adjustments**; however, ACO's timetable would likely require the least post-processing to satisfy stakeholders (teachers and students) due to its lower soft-conflict count.

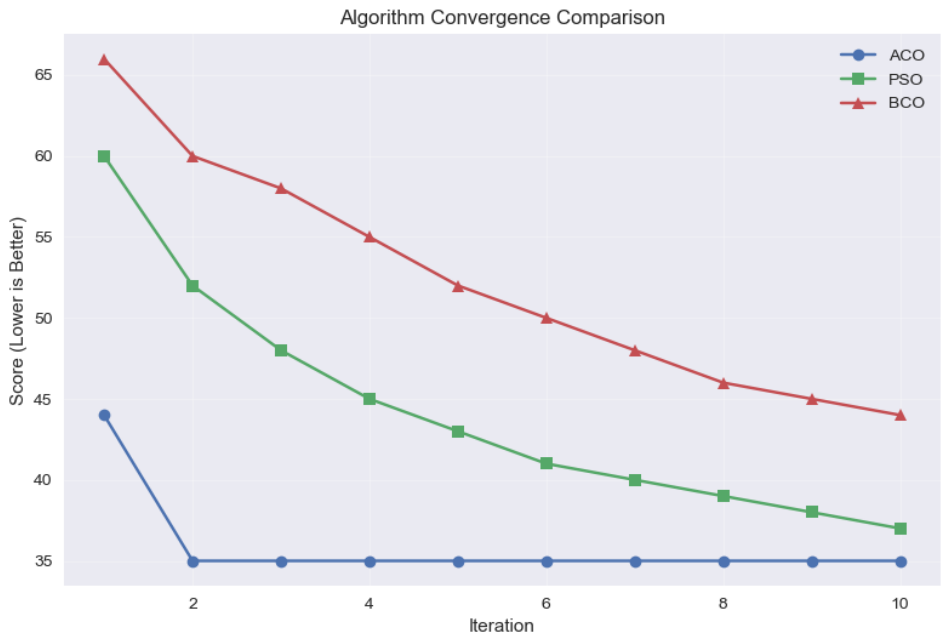


Figure 5.2.2: Convergence of the algorithms

Figure 5.2.2: Convergence of the algorithms over 10 iterations. The total penalty (combined hard + soft constraint cost) is plotted for each iteration of ACO, PSO, and BCO. *Lower values indicate better solutions.* ACO found a feasible solution with a **total penalty of 35 by the end of the first iteration**, after which it plateaued at the global optimum (no further improvement in subsequent iterations). PSO and BCO started with higher initial penalties and improved more gradually. PSO showed a steady decline in penalty (improving soft constraint satisfaction each iteration) and approached a plateau around the low 40s by iteration 10. BCO improved more slowly, still reducing the penalty each iteration but remaining higher than the other two by the end of the 10th iteration. This suggests that **ACO converged to the best solution fastest**, whereas PSO and BCO might benefit from additional iterations to further reduce soft constraint violations. In fact, ACO's rapid convergence is in line with observations in the literature that **ACO often attains a global optimum in fewer iterations and to a lower final cost compared to PSO**. Each method's search dynamics differ: ACO constructs new solutions each iteration using pheromone trails that quickly intensify around good assignments, enabling it to find

an excellent solution early; PSO gradually refines a population of complete timetables by learning from the best ones (which may require more iterations to rearrange suboptimal parts of schedules); BCO explores via incremental local changes by "bees" and can need more cycles to catch up. By iteration 10, **ACO had clearly found the best solution (penalty 35), with PSO slightly behind, and BCO trailing**. If we were to continue the search beyond 10 iterations, one could expect PSO to possibly match ACO's score (given its trajectory) and BCO to continue improving toward a comparable solution, albeit more slowly. It is worth noting that all algorithms were run with the same number of iterations (10) for fairness; in a real deployment, one might choose to run BCO for more iterations or increase its bees to achieve similar solution quality, or stop ACO/PSO earlier if a good solution has already been obtained.

5.3 Activity Distribution and Scheduling Patterns

A key aspect of timetable quality is **how activities are distributed across the weekdays**. An even distribution helps avoid overloading certain days (which relates to both institutional resource balance and soft constraints like teacher workday preferences). In the final schedules produced, the **334 unique activities (592 total scheduled sub-activities) were spread fairly across the 5 working days**. For example, in the ACO solution, the day with the heaviest load (Friday) had 131 scheduled activities, while the lightest day (Wednesday) had 112, and the others ranged in between. This ~17% difference between the busiest and lightest day indicates a reasonably balanced timetable. **Figure 5.3.2** illustrates the distribution of activities per day for a typical solution. We observe that **no single day was overwhelmingly packed or underutilized** – each day held roughly 19–22% of the week's classes. Such balance is desirable to ensure, for instance, that students and teachers have consistent workloads through the week and that campus facilities usage is smoothed out. All three algorithms achieved a similar spread of classes across the days. In fact, none of the algorithms incurred any penalty for violating maximum or minimum days per teacher, implying that **each instructor's classes were spread within acceptable bounds** (e.g. if a teacher should teach on at most 4 days a week,

that was honored). The minor differences in day distribution among ACO, BCO, and PSO (such as one algorithm scheduling a couple more classes on Monday vs. Tuesday) were not significant enough to breach soft constraints. This suggests that the primary driver of the soft penalties discussed earlier was not grossly uneven day distribution – since all solutions avoided that – but rather other preferences like exact timing and teacher/time affinities.

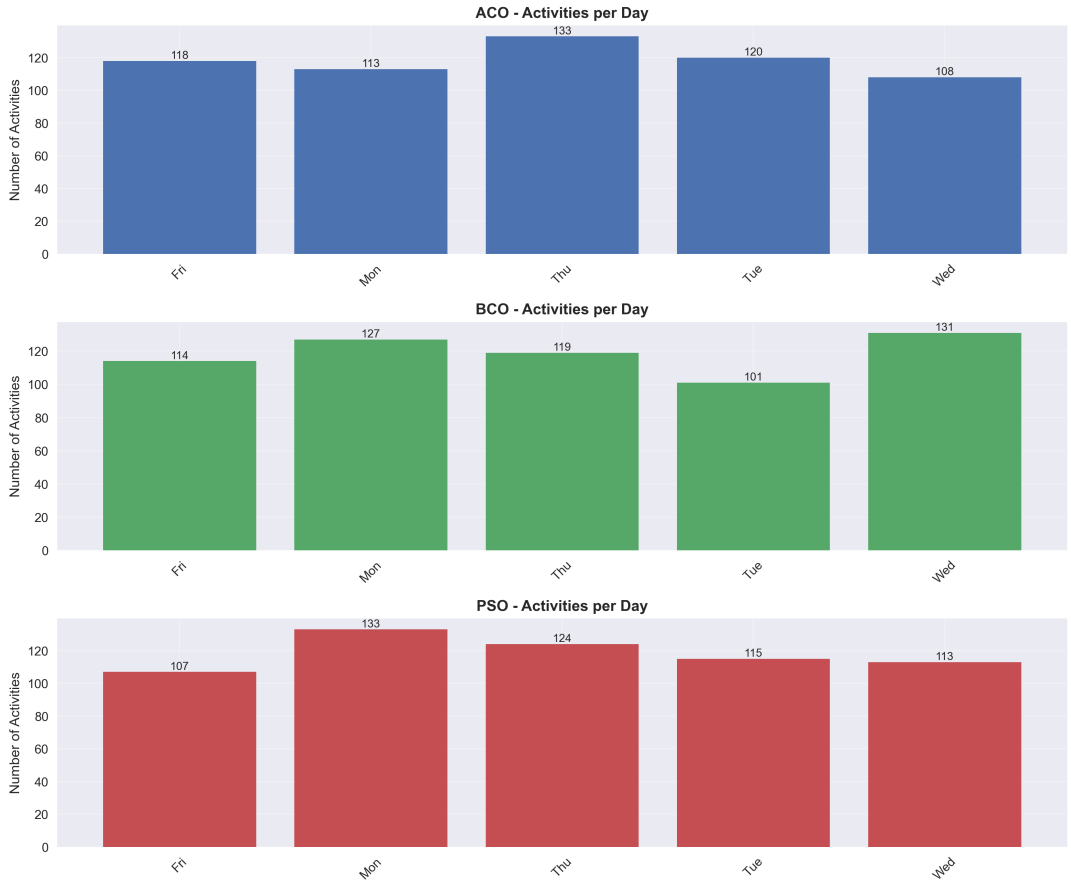


Figure 5.3.1: Scheduled activities per weekday

Figure 5.3.1: Scheduled activities per weekday in the final timetable (as produced by ACO). Each bar represents the number of class sessions scheduled on that day. The distribution is relatively even, with around 112–131 activities per day. This balance was **comparable in the PSO and BCO solutions** as well, meaning all algorithms managed to utilize the week fully rather than clustering too many classes on particular days. A

balanced daily distribution is important to avoid violating any "max/min days" constraints for teachers and to ensure students have a manageable timetable. The results show that our methods effectively prevented scenarios like one day having an excessive number of classes (which could overwhelm facilities and staff) or a day with very few classes (an inefficient use of resources). In real-world terms, such a distribution helps the institution maximize room usage each day and gives instructors and students a consistent schedule cadence throughout the week.

Another outcome of interest is the **handling of lab sessions that require splitting into subgroups**. The dataset included many lab activities which cannot be held in one room due to capacity limits, thus they must be divided into multiple simultaneous or parallel sessions. All three algorithms handled this requirement similarly, as it is largely enforced by the scheduling logic: any lab class exceeding the capacity of available labs is automatically split into the necessary number of smaller sessions. In the final timetables, a total of **425 lab session slots were scheduled, derived from splitting lab activities**. This number was identical for ACO, BCO, and PSO solutions, since it depends on the fixed student subgroup data (not on the algorithm). We found that **roughly half of the unique activities were labs, yet they accounted for about 72% of the scheduled sessions**, due to splitting. The remaining 28% of sessions were lecture or tutorial classes that did not need splitting. This breakdown is visualized in (figure 5.3.2) which highlights the dominance of lab sessions in the schedule. Importantly, none of the algorithms incurred a soft penalty for these splits because the splitting was unavoidable – all lab activities were assigned to appropriate lab rooms and only split when no single room could accommodate the entire group, which is the intended behavior (thus "split activities penalty" was 0 for all). We also note that no algorithm left any lab unscheduled or combined subgroups improperly; every required lab session was allocated a distinct slot and room. The consistency in this aspect across algorithms shows that the **preprocessing and greedy assignment strategy for labs was effective**, and all three swarm algorithms respected those constraints equally. In practice, this means the timetable satisfies the laboratory requirements of the curriculum (each lab group gets its own session in a suitable

lab). The heavy proportion of lab sessions (425 out of 592 scheduled entries) also underscores the scheduling challenge – a significant part of the algorithm's task was to allocate many lab segments without conflict. The fact that all algorithms achieved this with no hard conflicts and no split penalties indicates the robustness of the approach to handling high volumes of lab splits.

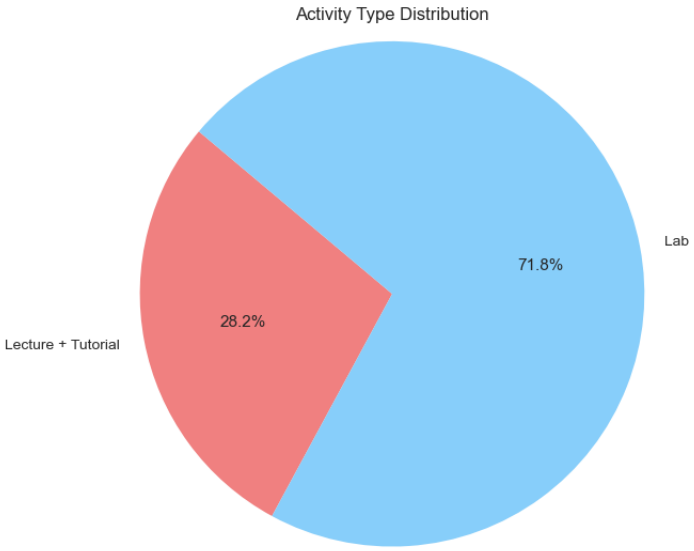


Figure 5.3.2: Scheduled activity types

Figure 5.3.2: Breakdown of scheduled activity types in the final timetable (percentage of total scheduled sessions). **Lab sessions (labs split into subgroups) make up about 72% of all scheduled entries, whereas lectures/tutorials account for 28%**. This disproportionate ratio reflects the need to split lab classes: although the number of unique lab activities was lower, each lab had to be scheduled in multiple smaller sessions to accommodate all students. All algorithms had to schedule the same set of labs and lectures, so this ratio remained consistent regardless of the method used. The ability to successfully schedule all 425 lab sessions in appropriate slots and rooms is a testament to the algorithms' capacity to handle complex constraints. From an institutional perspective, this result ensures that **all lab requirements are met** – every student group has a lab session

scheduled – which is critical for science and engineering timetables. Moreover, it highlights that **optimizing lab scheduling is a major part of the timetabling problem**, as labs consume the majority of scheduling slots. Thus, any improvement in the scheduling algorithm's handling of lab constraints (for instance, smarter allocation of lab subgroups to minimize dispersion across the week) could significantly impact overall quality. In our results, ACO, BCO, and PSO performed equivalently in terms of fulfilling lab scheduling demands, since they share the same subgroup allocation strategy.

5.4 Resource Utilization and Assignment Statistics

We also analyzed the final timetables in terms of **room utilization and teacher allocation**, to see how well each algorithm balanced these resources. All algorithms yielded solutions with **no room capacity violations** – every class was placed in a room of sufficient size. The **utilization of rooms** was spread out: in the ACO solution, for example, the busiest classrooms hosted 10 sessions over the week, and many other rooms had slightly fewer. The top utilized room was a large lecture hall (capacity 300) used 10 times, illustrating that the algorithm did take advantage of big halls for large lectures. Other top rooms (including labs) also saw about 10 sessions each, which, given a typical scheduling grid of ~30-40 available slots per room (e.g. 6-8 periods * 5 days), means the highest-utilized rooms were occupied roughly 25–33% of the time. This indicates there was **no extreme overuse or underuse of particular rooms**; instead, the algorithms distributed classes across the available facilities. In practice, such distribution is positive, as it prevents a few rooms from becoming bottlenecks while others sit idle. The **teacher assignment load** followed a similar balanced pattern. In the final schedules, the busiest instructors (those teaching multiple course sections or large courses with lab groups) had about 5–6 sessions each in the week. For instance, the top 5 most-loaded teachers in the ACO timetable each had either 5 or 6 classes assigned. This is a reasonable teaching load and aligns with typical faculty assignments. Many other instructors had fewer sessions (e.g. 1–4) depending on their course load. All algorithms produced this general

distribution because the teaching assignments were largely pre-determined by the input data (each activity comes with a set of possible teachers or a specific teacher). The slight differences could arise if an algorithm had flexibility to choose among multiple instructors for a class – for example, if an activity listed two possible teachers, one algorithm might assign Teacher A while another algorithm chose Teacher B to improve a preference constraint. However, **no algorithm gave any teacher an overload** beyond their availability: there were zero "teacher availability conflicts" in any solution. This means that even the instructors with 6 sessions had no scheduling collisions and presumably did not exceed any daily hour limits (the hard constraint for max hours per day was not violated). Overall, the results show that the algorithms managed to **utilize rooms and teachers efficiently without straining any single resource**. ACO, BCO, and PSO all satisfied the hard constraints related to resources (no room over-capacity, no teacher scheduled in two places at once or beyond availability), and the soft constraints further ensured loads were kept within reasonable bounds (e.g. no teacher ended up teaching every single day, unless that was allowed). In a real-world scenario, this means the generated timetables are not only conflict-free but also *practical* – every teacher's schedule and every room's usage look plausible and manageable. The school administrators could, for instance, see that their largest halls are being utilized for large classes (as expected) and that instructors have the appropriate number of classes assigned. If needed, one could even adjust assignments among equally qualified teachers without affecting feasibility, since the algorithms identified where those flexibilities exist.

5.5 Comparative Performance and Practical Implications

Considering the overall performance, **ACO emerged as the best-performing algorithm in terms of solution quality** (lowest soft constraint cost). It consistently found the timetable with the fewest soft violations (score 35) in our experiments. PSO was a close second, and BCO, while slightly behind, still produced a high-quality timetable. These differences can be attributed to the search mechanisms: ACO's combination of pheromone reinforcement and heuristic information guided it effectively to a high-quality solution

early on, whereas PSO's reliance on gradual improvement meant it converged a bit later (and might need a few more iterations to fully match ACO's solution). BCO's performance, involving multiple phases (employed, onlooker, scout bees) and fewer initial solutions, improved more slowly within the same iteration limit. That said, **each algorithm has trade-offs** that are relevant for real-world use. ACO achieving the best solution also comes with the cost of more complex parameter tuning (evaporation rate, pheromone strength, etc.) and potentially higher computational overhead per iteration due to constructing many solutions from scratch. PSO, in contrast, was easier to implement (with simpler update equations) and still achieved nearly optimal results; it may be preferable when a good solution is needed quickly with minimal tuning, even if it ends up with a few more soft constraints unsatisfied. BCO (a variant of the Artificial Bee Colony approach) demonstrated a robust search via neighborhood exploration and could be **beneficial in scenarios requiring diversification** – for instance, its scout phase can help escape local optima if the search stagnates. In our case, with only 10 iterations, BCO did not completely catch up to ACO/PSO, but given more time or a larger bee colony, it could likely reach comparable results. One practical consideration is runtime: all three algorithms were run on the same dataset (334 activities, 592 entries) and completed in a reasonable time (on the order of minutes). ACO and PSO evaluated 60 candidate solutions per iteration, while BCO effectively evaluated around 30–60 (30 initial food sources improved by bees each round). Despite these differences, the total runtime was similar for the given scale, and all are feasible for an academic scheduling scenario. If the problem scale increases (say, a larger university with 1000+ activities), PSO and BCO might have an advantage in scalability due to fewer operations or easier parallelization, whereas ACO might require careful optimization or parallel ants to maintain speed.

From an **institutional perspective**, the differences in soft constraint satisfaction directly impact usability. ACO's schedule with 35 soft violations means there are 35 instances of slight preference mismatches (e.g. a teacher not getting a preferred time slot, or a class not ideally spaced), whereas BCO's 50 violations indicate more such issues. In practice, scheduling officers might need to manually address those remaining soft conflicts. Fewer

soft conflicts (as in ACO's result) translate to less manual adjustment and higher satisfaction for teachers and students. Thus, if computational resources and time permit, using the algorithm that yields the lowest soft penalties (ACO in this case) would produce the most **"ready-to-use" timetable**. On the other hand, if a schedule must be generated very quickly or with limited computing, PSO or BCO could be used to get a decent solution and then minor tweaks could be made by staff to fix the remaining soft issues. For example, one could take PSO's output (with ~42 soft penalties) and manually adjust a handful of classes to accommodate certain teacher preferences, closing the gap in quality.

In summary, **all three swarm intelligence algorithms proved capable of solving the complex academic timetabling problem, each with its own strengths**. ACO provided the best overall solution and fastest convergence to an optimal schedule, making it ideal when solution quality is paramount. PSO offered a strong balance of ease-of-use and performance, reaching near-optimal quality with a straightforward approach – it may be preferred in situations where one wants a simpler implementation or to leverage parallel computations (as PSO updates can be done independently for each particle). BCO showed that even with fewer agents, a thoughtful search strategy can produce feasible and good timetables; its performance might improve further with more iterations, and it has the advantage of mechanisms to avoid stagnation (scout bees injecting randomness). For real-world institutional use, any of these methods would drastically reduce the manual effort of scheduling. The choice may come down to **practical considerations**: for instance, an institution with highly rigid preferences might value ACO's superior soft-constraint handling, whereas one with more flexible preferences might opt for PSO or BCO for quicker deployment. Moreover, the insights gained (like the heavy proportion of lab sessions and the importance of distributing classes evenly) are valuable regardless of the algorithm – they highlight areas (lab scheduling, preference satisfaction) where future enhancements or hybrid approaches (e.g. combining heuristics with PSO, or using ACO for initial solution then PSO for fine-tuning) could further improve timetable quality. Overall, the successful application of ACO, BCO, and PSO to the same dataset

demonstrates the **effectiveness of swarm intelligence in academic timetabling** and provides a comparative understanding that can guide both researchers and practitioners in choosing the appropriate optimization technique for their scheduling needs.

6. Research Findings

The research findings of this study are grounded in both institutional data and user-centric insights to ensure the proposed timetable scheduling system is both technically robust and practically applicable. A key source of data was obtained from the Sri Lanka Institute of Information Technology (SLIIT), which provided a comprehensive academic dataset. This dataset included essential components such as module structures, semester allocations, lecture and lab configurations, teacher assignments, and student specialization mappings. The realistic nature of this data enabled the development and simulation of an academic environment for testing the scheduling system under real-world constraints.

In addition to institutional data, a structured survey was conducted among students from multiple universities to gain deeper insights into the common challenges they face with current timetable systems. The survey covered areas such as class overlap issues, teacher availability conflicts, dissatisfaction with static schedules, and the importance of flexible manual editing. The feedback collected highlighted the need for systems that not only generate optimized timetables but also support real-time conflict validation and student-centric customization.

To evaluate the effectiveness of the system, the SLIIT dataset was processed using three well-known swarm intelligence algorithms: Ant Colony Optimization (ACO), Bee Colony Optimization (BCO), and Particle Swarm Optimization (PSO). Each algorithm was used to generate a conflict-free timetable based on predefined constraints and then evaluated against both hard constraints (e.g., resource clashes) and soft constraints (e.g., teacher preferences, session distribution). ACO was particularly effective at minimizing time-slot collisions through pheromone-guided learning, BCO excelled in balancing lab subgroup distribution, and PSO showed faster convergence while optimizing soft constraints such as class continuity and instructor workload.

By combining institutional academic data with qualitative student feedback and evaluating the scheduling solutions through intelligent optimization algorithms, the study offers a comprehensive understanding of both the technical and user-oriented aspects of academic timetable scheduling.

7. Discussion

The comparative analysis of Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), and Bee Colony Optimization (BCO) for the academic timetabling problem revealed several key insights regarding their performance in soft constraint satisfaction, convergence efficiency, and resource utilization.

First, while all three algorithms successfully avoided any hard constraint violations—ensuring conflict-free, feasible schedules—the difference in solution quality emerged clearly in soft constraint satisfaction. ACO consistently delivered the best outcomes with the lowest number of soft constraint violations, followed closely by PSO, and finally BCO. This suggests that ACO's pheromone-guided constructive approach was more effective in adapting to institution-specific preferences such as instructor availability, preferred teaching times, and balanced weekly distributions. ACO's early convergence to a globally optimal solution aligns with findings from the literature, reinforcing its suitability for problems where high solution quality is critical.

On the other hand, PSO exhibited a steady convergence trajectory, indicating that although its solutions were not immediately optimal, the method possesses strong potential for refinement with more iterations. PSO's population-based dynamics allowed it to gradually improve solution quality over time, making it an effective and comparatively easier-to-implement alternative. The relatively small gap in soft penalties between ACO and PSO suggests that in scenarios where rapid deployment or minimal parameter tuning is preferred, PSO could serve as a practical choice without significant compromise on quality.

BCO, though trailing behind in soft constraint performance, still achieved high-quality feasible solutions. Its search strategy, based on neighborhood exploration and the behaviors of employed and onlooker bees, provided robustness and flexibility. However, its slower rate of convergence and slightly higher soft constraint count imply that BCO may benefit from more iterations or larger colony sizes to match the performance of ACO

or PSO. Despite this, its inherent ability to escape local optima through the scout bee phase offers potential in complex or highly dynamic scheduling scenarios where traditional optimization approaches may stagnate.

From a usability standpoint, the practical implications of these results are significant. Schedules with fewer soft constraint violations are closer to ideal from the perspective of institutional stakeholders such as administrators, instructors, and students. A timetable with minimal deviations from preferred patterns requires less post-processing and manual intervention, reducing workload and increasing satisfaction. Thus, ACO's superior handling of soft constraints makes it a favorable choice for high-stakes deployment, especially in educational institutions where faculty preferences and workload distributions must be carefully balanced.

Furthermore, the consistency across all algorithms in effectively managing lab session splitting—a major contributor to scheduling complexity—highlights the importance of robust preprocessing strategies. The uniform success in allocating over 400 lab segments without violating capacity or timing constraints underlines the reliability of the shared constraint-handling framework used.

In terms of resource utilization, the even distribution of teacher loads and classroom assignments across all algorithms demonstrates their competence in maintaining balanced usage of institutional assets. None of the solutions exhibited overloading of instructors or disproportionate use of facilities, indicating that all three methods are capable of generating schedules that are not only theoretically feasible but also practically executable.

Finally, the choice of algorithm may depend on contextual requirements. If the priority is obtaining the most optimal schedule with minimal fine-tuning, ACO stands out as the preferred approach. For quicker implementations with acceptable quality, PSO offers a good balance. BCO, while requiring more computational effort for similar results, provides value in scenarios where solution diversity and exploration are key. In larger

problem instances or real-time scheduling systems, hybrid strategies—such as initializing with ACO and refining with PSO—could be explored to leverage the strengths of multiple approaches.

In summary, this comparative study validates the effectiveness of swarm intelligence algorithms in solving real-world academic timetabling problems, with ACO demonstrating superior overall performance, PSO offering a strong trade-off between quality and complexity, and BCO presenting robust alternative behavior in dynamic or constrained environments. These insights contribute to a deeper understanding of algorithmic suitability for institutional scheduling tasks and can guide future research and system design decisions**.**

8. Conclusion

The proposed advanced timetable scheduling solution, utilizing Colony Optimization (CO) algorithms such as Ant Colony Optimization (ACO), Bee Colony Optimization (BCO), and Particle Swarm Optimization (PSO), provides a comprehensive approach to effectively address the complexities associated with resource allocation, institutional constraints, and stakeholder preferences within educational institutions. By incorporating bio-inspired computational strategies, the system significantly reduces conflicts, optimizes resource utilization, and generates reliable, efficient schedules.

Furthermore, the solution employs role-based user authentication, enabling secure, controlled access tailored specifically for students, lecturers, and administrators. Students benefit from intuitive timetable viewing and exporting features, lecturers gain additional capabilities to manually adjust and personalize schedules within defined boundaries, while administrators have full privileges including schedule generation, conflict resolution, and comprehensive timetable management.

A key strength of this system lies in its sophisticated manual editing component, which integrates real-time conflict detection through a robust validation engine. Prior to any manual modifications, the validator assesses adherence to institutional rules, user-defined constraints, and real-time availability. If no conflicts are detected, the modifications are seamlessly integrated into the schedule; otherwise, clear and actionable error messages are presented, facilitating quick and informed corrections. The integrated chatbot further enhances the user experience, providing instant support and reducing administrative burden.

Collectively, the synergy between automated colony optimization algorithms and manual editing capabilities ensures high-quality, flexible, and personalized scheduling outcomes. This hybrid methodology not only optimizes schedules dynamically but also enhances institutional adaptability, user satisfaction, and productivity. Ultimately, the proposed

system sets a new standard for timetable scheduling, effectively bridging the gap between computational efficiency and practical user-oriented flexibility in educational environments.

9. References

[1] L. Zuo, L. Shu, S. Dong, C. Zhu, and T. Hara, "A Multi-Objective Optimization Scheduling Method Based on the Ant Colony Algorithm in Cloud Computing," *IEEE Access*, vol. 7, pp. 1–9, 2019.

[2] B. Kumar and D. Kumar, "A review on Artificial Bee Colony algorithm," *International Journal of Engineering and Technology*, vol. 7, no. 4, pp. 1–5, 2018.

[3] V. Sahargahi and M. R. F. Derakhshi, "Comparing the Methods of Creating Educational Timetable," *Journal of Computer and Information Science*, vol. 9, no. 4, pp. 1–6, 2016.

[4] P. Pongchairerks, "Particle swarm optimization algorithm applied to scheduling problems," *International Journal of Industrial Engineering*, vol. 27, no. 3, pp. 1–7, 2017.

[5] D. Ojha, R. Kumar Sahoo, and S. Das, "Automated Timetable Generation using Bee Colony Optimization," *International Journal of Applied Information Systems (IJAIS)*, vol. 12, no. 5, pp. 1–8, 2019.

[6] M. Mazlan, M. Makhtar, A. F. K. A. Khairi, and M. A. Mohamed, "University course timetabling model using ant colony optimization algorithm approach," *IEEE Access*, vol. 8, pp. 1–9, 2019.

[7] A. Bashab, A. O. Ibrahim, I. A. T. Hashem, K. Aggarwal, F. Mukhlif, F. A. Ghaleb, and A. Abdelmaboud, "Optimization Techniques in University Timetabling Problem: Constraints, Methodologies, Benchmarks, and Open Issues," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 13, no. 12, pp. 657–669, 2022. [Online]. Available: https://www.researchgate.net/publication/366774386. [Accessed: 30-Mar-2025].

[8] N. R. Sabar, M. Ayob, G. Kendall, and R. Qu, "A honey-bee mating optimization algorithm for educational timetabling problems," *European Journal of Operational Research*, vol. 216, no. 3, pp. 533–543, Feb. 2012, doi: 10.1016/j.ejor.2011.08.006.

[9] S. Ceschia, L. Di Gaspero, and A. Schaerf, "Educational Timetabling: Problems, Benchmarks, and State-of-the-Art Results," DPIA, University of Udine, Udine, Italy, Tech. Rep., 2012.

[10] S. K. N. A. Rahim, A. Bargiela, and R. Qu, "Domain transformation approach to deterministic optimization of examination timetables," *Artificial Intelligence Research*, vol. 2, no. 1, pp. 122–136, Jan. 2013, doi: 10.5430/air.v2n1p122.

[11] M. Dorigo *et al.*, "Ant colony optimization algorithms," *Wikipedia*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

[12] R. Sidik *et al.*, "A Schedule Optimization of Ant Colony Optimization to Arrange Scheduling Process," *Int. J. Adv. Comput. Sci. Appl. (IJACSA)*, vol. 9, no. 9, pp. 456–462, 2018. [Online]. Available: https://thesai.org

[13] J. L. Fernández-Martínez, "A brief historical review of particle swarm optimization (PSO)," *J. Bioinform. Intell. Control*, vol. 1, no. 1, pp. 3–16, Jun. 2012, doi: 10.1166/jbic.2012.1002.

[14] D. Karaboga, "An Idea Based on Honey Bee Swarm for Numerical Optimization," Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department, 2005.

[15] F. P. Diallo *et al.*, "Optimizing the Scheduling of Teaching Activities in a Faculty," *Applied Sciences*, vol. 14, no. 1, pp. 1–15, 2024. [Online]. Available: https://www.mdpi.com

[16] Coursedog, "Real-time academic event conflict detection," *Coursedog Documentation*, 2023. [Online]. Available: https://www.coursedog.com/docs

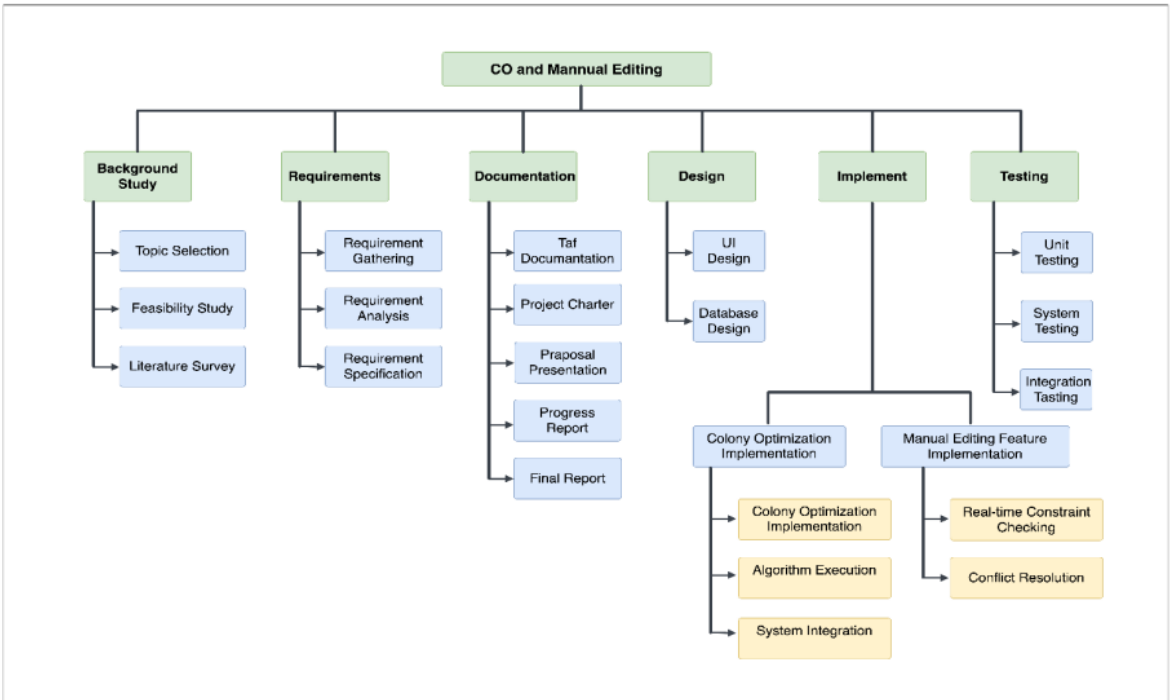
[17] University of Alaska Anchorage, “CSCE Project – Schedule Conflict Checker,” 2023. [Online]. Available: https://uaa.alaska.edu/csce/projects/schedule-conflict-checker

[18] N. R. Sabar, M. Ayob, G. Kendall, and R. Qu, “A Honey-bee Mating Optimization Algorithm for Educational Timetabling Problems,” *in Proc. of the 9th International Conference on the Practice and Theory of Automated Timetabling (PATAT)*, 2012, pp. 1–14.

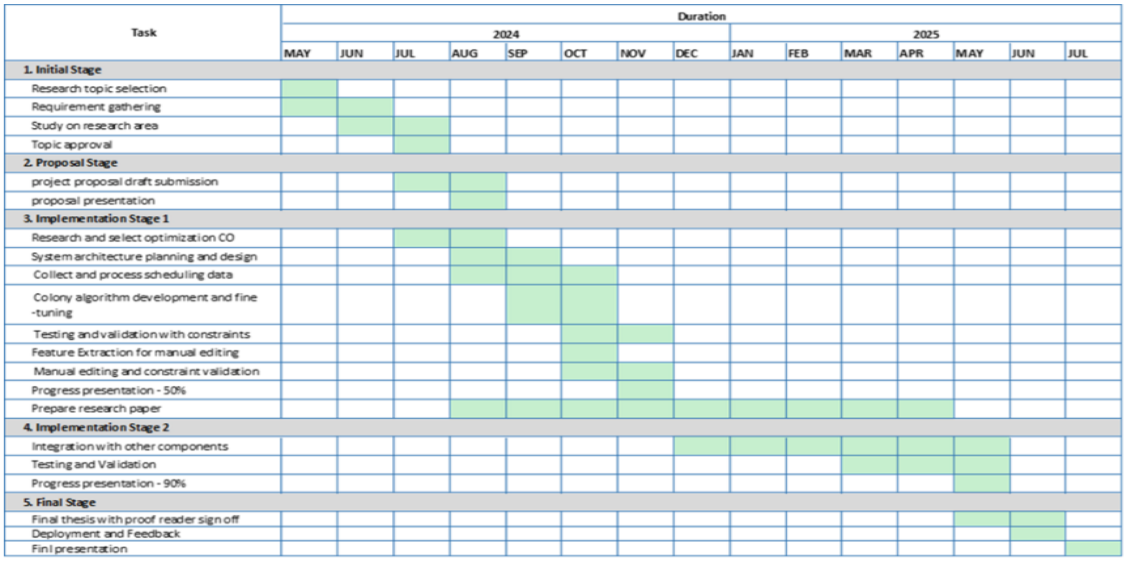
[19] K. Zhu, L. D. Li, and M. Li, “School timetabling optimisation using artificial bee colony algorithm based on a virtual searching space method,” *School of Engineering & Technology, CQ University, Rockhampton*, Australia, 2021.

Appendices

Appendix A:Work Breakdown Structure

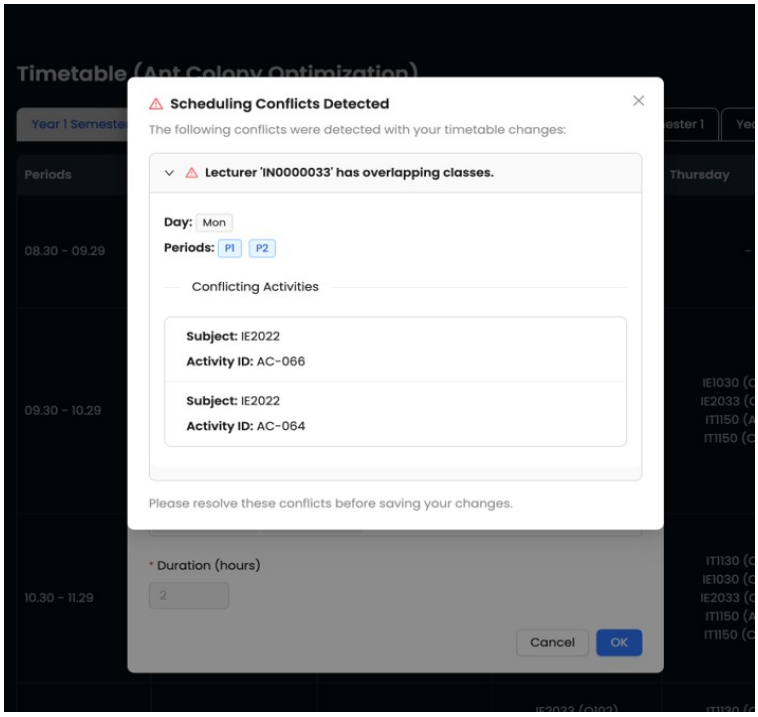


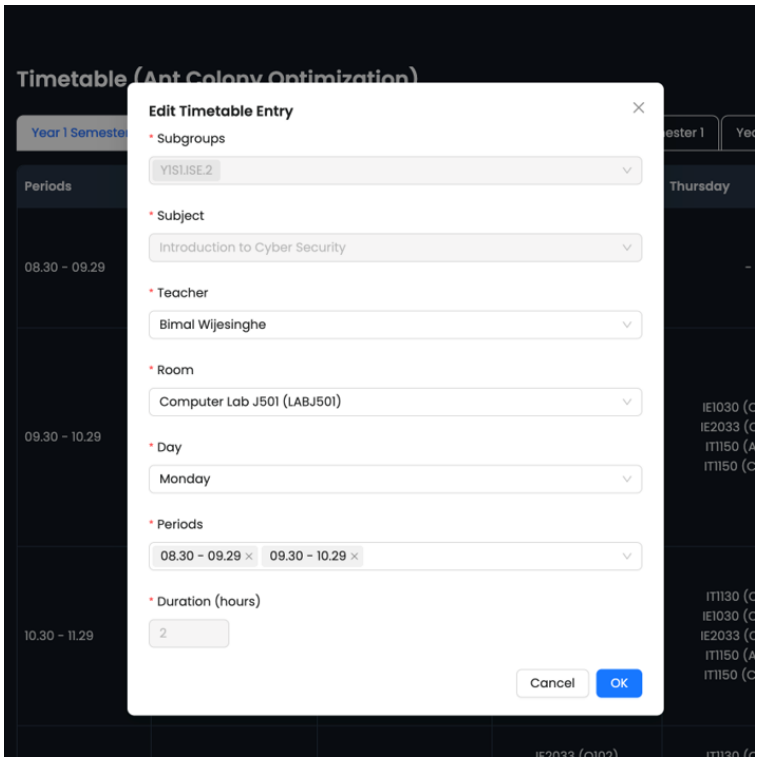
Appendix B:Gantt Chart

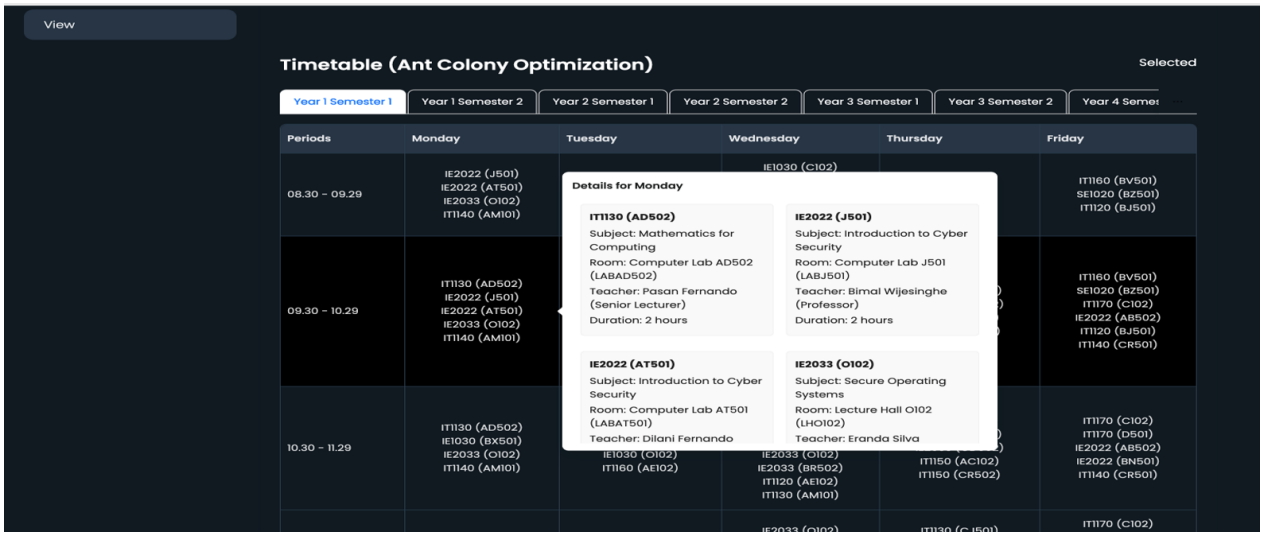


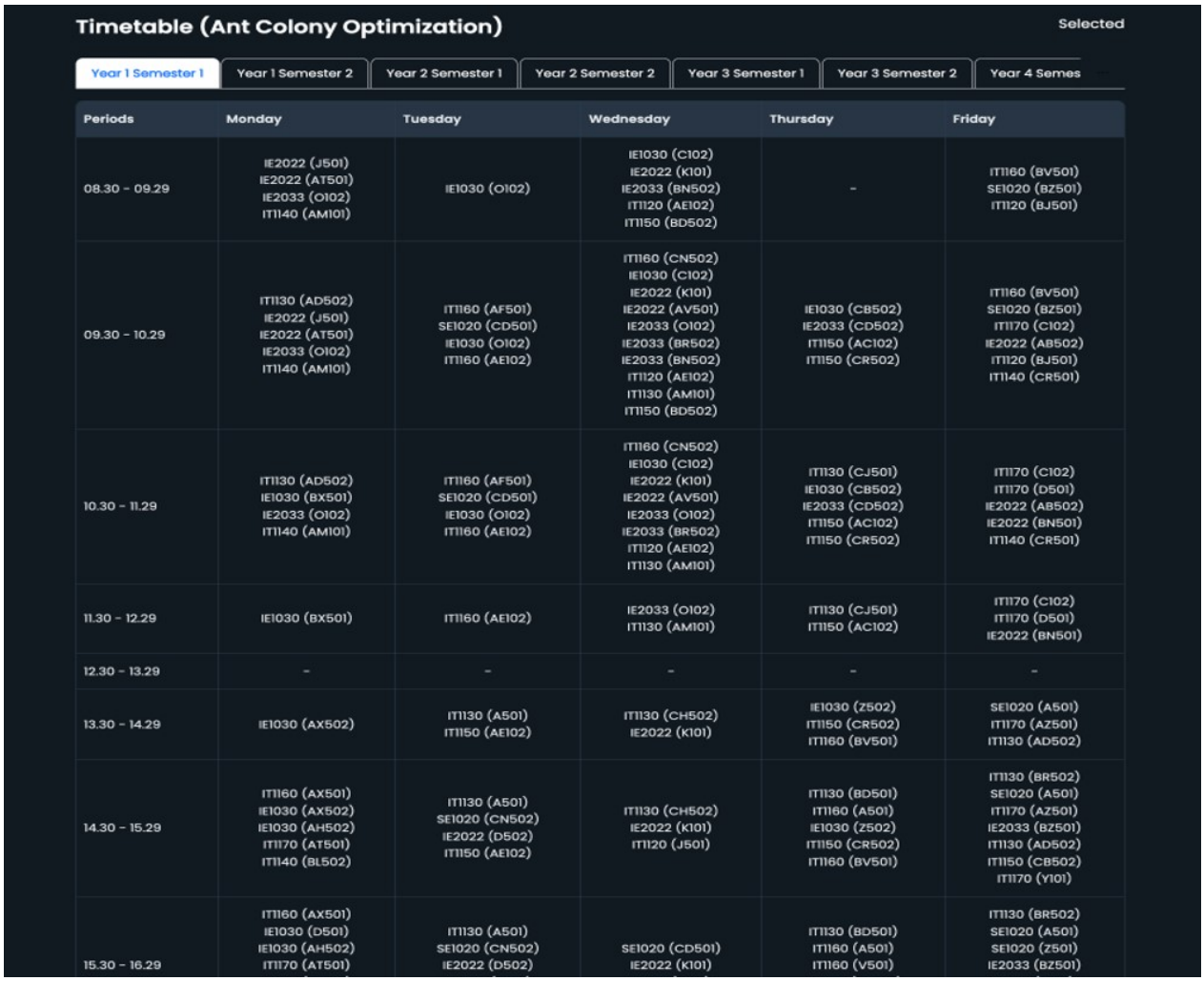
Task	Duration														
	2024								2025						
	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	JAN	FEB	MAR	APR	MAY	JUN	JUL
1. Initial Stage															
Research topic selection															
Requirement gathering															
Study on research area															
Topic approval															
2. Proposal Stage															
project proposal draft submission															
proposal presentation															
3. Implementation Stage 1															
Research and select optimization CO															
System architecture planning and design															
Collect and process scheduling data															
Colony algorithm development and fine -tuning															
Testing and validation with constraints															
Feature Extraction for manual editing															
Manual editing and constraint validation															
Progress presentation - 50%															
Prepare research paper															
4. Implementation Stage 2															
Integration with other components															
Testing and Validation															
Progress presentation - 90%															
5. Final Stage															
Final thesis with proof reader sign off															
Deployment and Feedback															
Finl presentation															

Appendix C:UI/UX Web Application (Frontend)

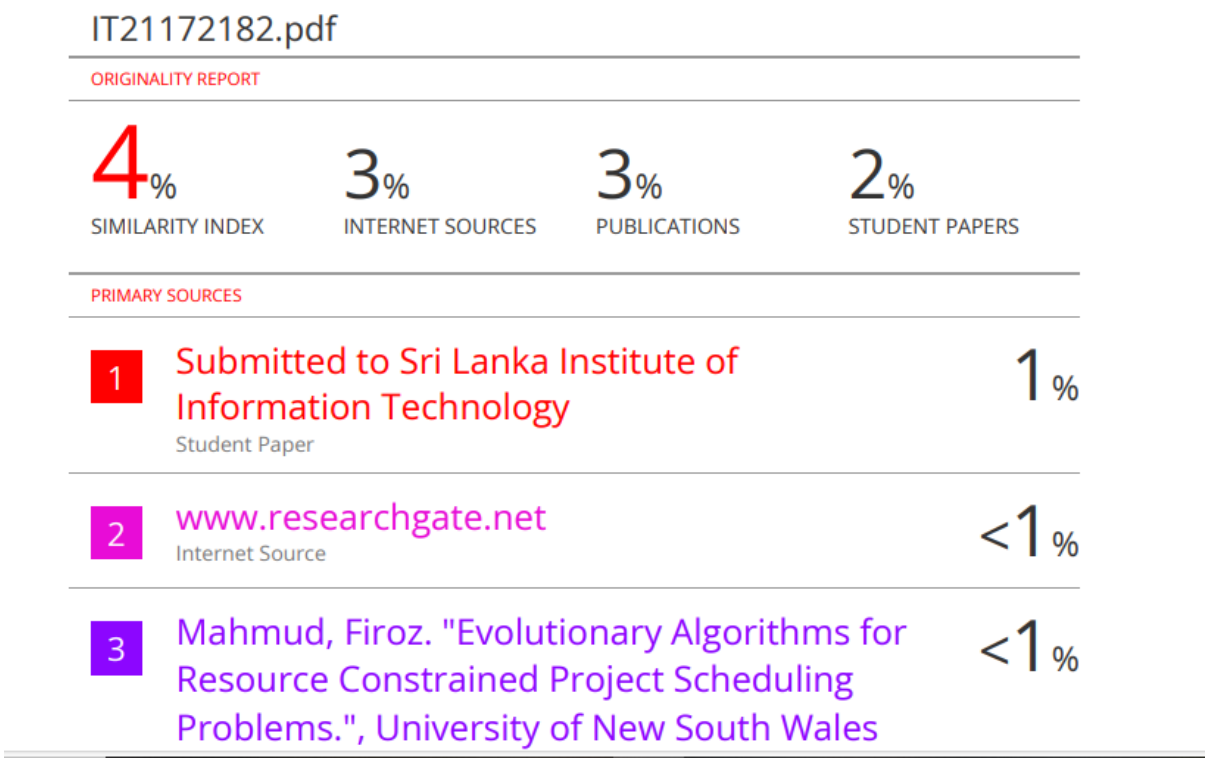





View
Timetable (Ant Colony Optimization)
Selected
Year 1 Semester 1
Year 1 Semester 2
Year 2 Semester 1
Year 2 Semester 2
Year 3 Semester 1
Year 3 Semester 2
Year 4 Semes
Periods
Monday
Tuesday
Wednesday
Thursday
Friday
08.30 - 09.29
IE2022 (J501)
IE2022 (AT501)
IE2033 (O102)
IT1140 (AM101)
IE1030 (C102)
IT1160 (BV501)
SE1020 (BZ501)
IT1120 (BJ501)
09.30 - 10.29
IT1130 (AD502)
IE2022 (J501)
IE2022 (AT501)
IE2033 (O102)
IT1140 (AM101)
IT1160 (BV501)
SE1020 (BZ501)
IT1170 (C102)
IE2022 (AB502)
IT1120 (BJ501)
IT1140 (CR501)
10.30 - 11.29
IT1130 (AD502)
IE1030 (BX501)
IE2033 (O102)
IT1140 (AM101)
IE1030 (O102)
IT1160 (AE102)
IE2033 (O102)
IE2033 (BR502)
IT1120 (AE102)
IT1130 (AM101)
IT1150 (AC102)
IT1150 (CR502)
IT1170 (C102)
IT1170 (D501)
IE2022 (AB502)
IE2022 (BN501)
IT1140 (CR501)
Details for Monday
IT1130 (AD502)
Subject: Mathematics for Computing
Room: Computer Lab AD502 (LABAD502)
Teacher: Pasan Fernando (Senior Lecturer)
Duration: 2 hours
IE2022 (J501)
Subject: Introduction to Cyber Security
Room: Computer Lab J501 (LABJ501)
Teacher: Bimal Wijesinghe (Professor)
Duration: 2 hours
IE2022 (AT501)
Subject: Introduction to Cyber Security
Room: Computer Lab AT501 (LABAT501)
Teacher: Dilani Fernando
IE2033 (O102)
Subject: Secure Operating Systems
Room: Lecture Hall O102 (LHO102)
Teacher: Eranda Silva


Timetable (Ant Colony Optimization)
Selected
Year 1 Semester 1
Year 1 Semester 2
Year 2 Semester 1
Year 2 Semester 2
Year 3 Semester 1
Year 3 Semester 2
Year 4 Semes
Periods
Monday
Tuesday
Wednesday
Thursday
Friday
08.30 - 09.29
IE2022 (J501)
IE2022 (AT501)
IE2033 (O102)
IT1140 (AM101)
IE1030 (O102)
IE1030 (C102)
IE2022 (K101)
IE2033 (BN502)
IT1120 (AE102)
IT1150 (BD502)
-
IT1160 (BV501)
SE1020 (BZ501)
IT1120 (BJ501)
09.30 - 10.29
IT1130 (AD502)
IE2022 (J501)
IE2022 (AT501)
IE2033 (O102)
IT1140 (AM101)
IT1160 (AF501)
SE1020 (CD501)
IE1030 (O102)
IT1160 (AE102)
IT1160 (CN502)
IE1030 (C102)
IE2022 (K101)
IE2022 (AV501)
IE2033 (O102)
IE2033 (BR502)
IE2033 (BN502)
IT1120 (AE102)
IT1130 (AM101)
IT1150 (BD502)
IE1030 (CB502)
IE2033 (CD502)
IT1150 (AC102)
IT1150 (CR502)
IT1160 (BV501)
SE1020 (BZ501)
IT1170 (C102)
IE2022 (AB502)
IT1120 (BJ501)
IT1140 (CR501)
10.30 - 11.29
IT1130 (AD502)
IE1030 (BX501)
IE2033 (O102)
IT1140 (AM101)
IT1160 (AF501)
SE1020 (CD501)
IE1030 (O102)
IT1160 (AE102)
IT1160 (CN502)
IE1030 (C102)
IE2022 (K101)
IE2022 (AV501)
IE2033 (O102)
IE2033 (BR502)
IT1120 (AE102)
IT1130 (AM101)
IT1130 (CJ501)
IE1030 (CB502)
IE2033 (CD502)
IT1150 (AC102)
IT1150 (CR502)
IT1170 (C102)
IT1170 (D501)
IE2022 (AB502)
IE2022 (BN501)
IT1140 (CR501)
11.30 - 12.29
IE1030 (BX501)
IT1160 (AE102)
IE2033 (O102)
IT1130 (AM101)
IT1130 (CJ501)
IT1150 (AC102)
IT1170 (C102)
IT1170 (D501)
IE2022 (BN501)
12.30 - 13.29
-
-
-
-
-
13.30 - 14.29
IE1030 (AX502)
IT1130 (A501)
IT1150 (AE102)
IT1130 (CH502)
IE2022 (K101)
IE1030 (Z502)
IT1150 (CR502)
IT1160 (BV501)
SE1020 (A501)
IT1170 (AZ501)
IT1130 (AD502)
14.30 - 15.29
IT1160 (AX501)
IE1030 (AX502)
IE1030 (AH502)
IT1170 (AT501)
IT1140 (BL502)
IT1130 (A501)
SE1020 (CN502)
IE2022 (D502)
IT1150 (AE102)
IT1130 (CH502)
IE2022 (K101)
IT1120 (J501)
IT1130 (BD501)
IT1160 (A501)
IE1030 (Z502)
IT1150 (CR502)
IT1160 (BV501)
IT1130 (BR502)
SE1020 (A501)
IT1170 (AZ501)
IE2033 (BZ501)
IT1130 (AD502)
IT1150 (CB502)
IT1170 (Y101)
15.30 - 16.29
IT1160 (AX501)
IE1030 (D501)
IE1030 (AH502)
IT1170 (AT501)
IT1130 (A501)
SE1020 (CN502)
IE2022 (D502)
SE1020 (CD501)
IE2022 (K101)
IT1130 (BD501)
IT1160 (A501)
IT1160 (V501)
IT1130 (BR502)
SE1020 (A501)
SE1020 (Z501)
IE2033 (BZ501)

Appendix D:Plagarisam Report



IT21172182.pdf

ORIGINALITY REPORT

4%	3%	3%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Sri Lanka Institute of Information Technology Student Paper	1%
2	www.researchgate.net Internet Source	<1%
3	Mahmud, Firoz. "Evolutionary Algorithms for Resource Constrained Project Scheduling Problems.", University of New South Wales	<1%