Optimized Automatic Course Timetabling Service Architecture for Integration with Vendor Management Systems

Marwah M. Alansari
Department of IT, College of Computing and Informatics
Saudi Electronic University,
Riyadh, Saudi Arabia

***Abstract*—Generating university course timetables is a complex problem, especially in large environments such as institutions. Currently, some universities in Saudi Arabia manually generate timetables for classes because they use Vendor Management Systems (VMS) for registration and management. Manually generating course timetables is time-consuming and laborious for the academic staff. Although various methods have been proposed to generate timetables, they address specific environments or systems that can be extended to or work as separate components of the university management system. In this paper, we propose a service-based system with a decentralized architecture that can fully automate the process of course timetable generation and can be easily integrated into VMS. The proposed service-based system employs a genetic algorithm to optimize the process of scheduling courses and generating timetables. The system was implemented using JAVA RESTful web services, and the algorithm was tested by generating various course timetables with various constraints. The results showed that the proposed decentralized architecture is applicable to and can be fully integrated with any VMS. Furthermore, the use of genetic algorithm set up to 200 generations and iterate 1000 times produces acceptable timetables without violating any of the defined constraints.**

***Keywords*—Courses timetable generation; genetic algorithm; course scheduling; service-based system; service-oriented architecture; optimization; web services**

I. Introduction

Some universities in Saudi Arabia use vendor management systems (VMS) such as Banner [1] to enroll and manage students. The main architecture of a VMS constitutes a data-based management system deployed on the server side and a web portal for accessing the system from the client side. However, a VMS implements only common generic requirements related to registration and management processes. Therefore, such systems can only function for general use cases, such as publishing, presenting, and manipulating student timetables. Currently, university vendor management software cannot consider specific requirements related to course management at universities. Examples of these requirements include the registration and timetable generation processes, which typically comprise many essential periodic tasks that have been performed manually thus far. Although extensive studies by artificial intelligence (AI) communities and operational research have focused on performing timetable generation through various algorithms [2] [3], VMS developers have not considered applying automatic algorithms to their systems for handling specific tasks. Extending and implementing specific tasks lengthens the development process and makes it costlier. Although such systems are being continuously developed, it is still costly to implement the specific cases and scenarios required by universities to automatically generate complete course timetables. Therefore, most universities prefer using VMS to perform general tasks and manually perform specific tasks.

Universities are encouraged to apply digital transformation and use AI solutions to reduce repetitive processes and move toward their complete automation. We discovered that generating course timetables is one of the repetitive, costly, and time-consuming tasks. Generally, generating university course timetables is considered complex and categorized as an NP-hard problem. This means that they entail exponential growth in search time and effort, wherein the problem factors such as the number of courses or students increases in size [4]. Therefore, we must pursue heuristic approaches that can handle different numbers of constraints, both hard and soft, that can vary from one institution to another. Thus, the objective of this study was to fully automate the process of preparing and allowing students to register on a timetable. This process is performed by generating the term timetable using heuristic approaches, which are elucidated in this study.

We encountered another characteristic of the generic architecture of VMS: they have a centralized architecture that primarily depends on a centralized database management system. By contrast, universities comprise several colleges, each of which comprises several departments. Therefore, a centralized architecture can result in various performance and availability problems, and the staff is unable to automate or enhance the registration process effectively. This research motivation is to improve and fully automate the registration process by overcoming the following two essential challenges:

1) Creating a service-based system that can be deployed in the form of decentralized architecture.
2) Designing and implementing an algorithm to generate course timetables effectively.

The structure of the paper is as follows. Section II includes a brief background of service-oriented architecture (SOA) and genetic algorithms (GAs). Section III covers related research on auto-generating university timetables. Section IV presents description of the problem. Section V contains an explanation of the proposed solution and introduces the design of the

proposed service-based automatic timetable generation architecture. Section VI presents the proposed GA design. Section VII describes the implementation of the architecture and the proposed GA. Section VIII discusses the results, figures, and findings, and Section IX presents our suggestions for future research directions and conclusions.

II. Background

SOA is a design paradigm employed to support distributed business applications, whereas a GA is an evolutionary algorithm that can be applied to solve optimization problems. In this section, we present a brief background of both SOA and GAs and the methods applied in our research.

A. Service-based System

The concept of SOA is applied to engineering business processes within large distributed systems. It ensures that a system has various characteristics, such as being interoperable, service-based, loosely coupled, usable, and fault-tolerant [5]. In an SOA, services are considered self-contained logical functionalities and designed through web services [6]. Furthermore, the backgrounds of the services can be encapsulated in different programming languages, wherein the service client only uses the description for the service without any knowledge of its implementation. There are two types of web service implementations: simple object access protocol (SOAP) and representational state transfer (REST). Another feature of the SOA is that it can provide an enterprise service bus (ESB), which is an infrastructure that enables high interoperability between deployed distributed services. ESBs are managed through business processes. The final feature of SOAs addressed in this section is loose coupling. SOAs enable delivering a service-based system with a reduced number of dependencies [5] [6]. It is important to remember that a single service can offer various capabilities grouped together if they relate to a functional context established by the service [6].

In an SOA, a distributed system is designed and built based on a basic software engineering concept: the theory of concerns. The strategic goals associated with service-oriented computing services indicate their purpose and capabilities through a service contract [6] that emphasizes the positioning of services as enterprise resources within agnostic functional contexts. Numerous design considerations have been proposed to ensure that individual service capabilities are appropriately defined based on an agnostic service context utilities are appropriately defined in relation to an agnostic service context [5].

The two common implementations of services are REST and SOAP. The REST service is required to identify resources that include one or more representations, either expected or provided, an address to uniquely locate the resource, a set of HTTP methods exposed at the interface-level metadata included in headers for requirements such as security tokens or caching information, and a REST-based service interaction. Standard HTTP methods are used in conjunction with HTTP response codes to establish a communication framework based on a uniform contract that can invoke service capabilities and communicate success, failure, and error conditions [6]. Resources that represent a resource in JavaScript object notation (JSON) an be considered as an alternative. In JSON, data are described in the name-value format [6].

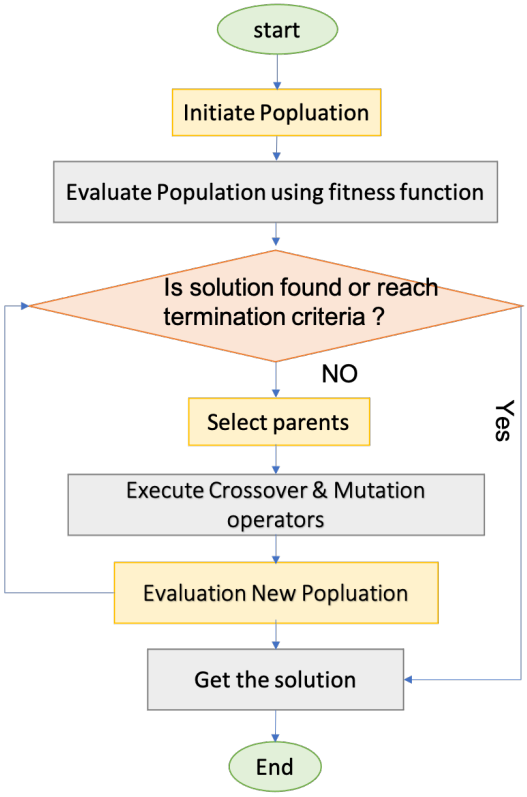


Fig. 1. General Steps of a Genetic Algorithm [7].

The specific method of encoding data “on the wire” and passing them among services is captured in SOAP services. In a SOAP service, both service requests and responses are encoded into XML documents [6]. SOAP-based web services rely heavily on the web services description language, which provides a method of expressing the service contract as a collection of operations with corresponding request/response messages [6].

B. Genetic Algorithm

GAs are evolutionary algorithms proposed by Holland in 1975 and recommended for solving optimization problems by Goldberg in 1975, who later suggested in 1989 that they can be used for solving optimization problems [7]. GAs are based on the Darwinian principle of the survival of the fittest among animals exposed to predators and environmental threats. The fittest entities are those that can adapt to evolving conditions, and their offspring inherit their characteristics and learn their skills, resulting in the creation of the best possible future generations. Furthermore, genetic mutations occur randomly among members of the same species, and some of these alterations may enhance the long-term stability of the superior individuals and their evolutionary offspring [8]. of the parent population that comprises surviving individuals (chromosomes) from previous generations and their offspring. The offspring, which represent new solutions, are generated through genetic operators such as crossover and mutation.

Parents that are selected to produce a new generation are those whose probability of selection is proportional to their fitness values. The higher the fitness value, the better the chance of surviving and reproducing [7] [8]. The algorithm begins with randomly generated initial population solutions, and the generated population gradually improves over time. It uses special criteria to select optimal individuals, who are then used to produce offspring. Offspring are generated using crossover and mutation operators [7] [8]. Fig. 1 shows the steps of a GA.

III. Related Work

The generation of university timetables is a well-studied problem in literature. It is considered a complex NP-hard problem owing to the complexity of the university environment. In this section, we describe some algorithms proposed in various studies to address the problem of generating university timetables.

A new version of the simulated annealing algorithm to address the problem of examination timetabling was proposed in [9]. The algorithm employs an acceptance criterion to move a selected exam and assess the moves by evaluating their acceptance through a temperature bin. The algorithm comprises 10 temperature bins to evaluate the number of evaluations uniformly. It uses the crystalized concept, which is assigned to the selected exam, and does not record any future acceptance moves in the temperature bins. It employs saturation degree-based heuristics combined with conflict-based statistics to eliminate the looping effect during initialization [9].

The authors of [10] proposed using particle swarm optimization (PSO) for generating university timetables. Unlike GAs, PSO simulates social behaviors to evaluate solutions. The evaluation is performed by determining the positioning and velocity of each particle by using the fitness value of the selected particle. The algorithm uses an initial step to assign time slots to the exam, whereas the remaining steps are used to assign rooms to time slots and solve exam timetabling problems at universities.

A modified event-grouping algorithm for finding the best solution by ordering events into groups was developed in [11], wherein events and conflicts are presented on an undirected graph. It should be noted that the execution time of the event-grouping algorithm increases when the number of groups increases [11].

The author in [12] proposed using an AI expert system to automatically generate a scheduling system for the course timetable problem. The auto-generated scheduling system was developed such that no conflict could occur among all the input facts, and features were provided to customize the timetable as required. The rule is executed based on the priority and ranking of the constraints. No specific information regarding the designed rule used in the expert system or definitions of hard and soft constraints were provided.

The author in [13], proposed reformulating an existing integer programming model. It employed the XML for high school timetabling (XHSTT) format to formulate a mathematical model of the problem. The authors developed a model for the problem and a set of models to formulate the constraints and operative functions to avoid clashes. The computational experiments also showed that the integer programming models resulting from the proposed formulation solved most of the instances in the XHSTT model more effectively. However, this algorithm cannot be generalized to other university models [13].

In [14], there is a suggestion to a hybrid method based on the improved parallel genetic algorithm and local search (IPGALS) to solve the course timetabling problem. The local search (LS) approach proposed in this work supports GA. The distance to feasibility (DF) criterion is applied to measure the hard constraints and ensure that they are never violated. The results showed that using DF improved performance for finding a feasible solution. A parallel approach is used to handle a higher number of constraints. The LS and elitism operators were implemented after applying the crossover and mutation operators. Therefore, the major limitation of IPGALS is that it cannot ensure the generation of a feasible timetable for large groups [14].

The author in [15] proposed applying two methods to solve the room-optimization problem in timetable generation [15]. The first method involves two-stage integer linear programming (ILP), which applies lexicographic optimization. The objective is to maximize the number of students seated and then apply it to optimize room allocation. The ILP is suitable for smaller domains; however, the computation time increases for larger domains. domains. A greedy algorithm was proposed to enhance the first approach. The lecturer is assigned to a room based on the computation of the cost function. The objective is to maximize the number of allocated students. Cost function identification enhances the performance of the greedy algorithm [15].

Guo et al. proposed a new algorithm based on the greedy method combined with a GA to solve the course timetabling problem, wherein the greedy algorithm is applied to generate the initial population [16].

In [4], a design and implementation of timetable generation based on GA were proposed using different combinations of selection algorithms and mutation types. The system uses tournament and roulette wheel selections to evaluate two cases and determine the selection technique that provides a better solution. Furthermore, the study also applied a mutation error to determine if it can retrieve a better solution faster. A similar approach was proposed in [17], wherein a GA model was applied to automatically arrange a university timetable and further study the effect of changing the crossover and mutation rates. Their simulation results showed a crossover rate of 0.70, and no hard constraints appeared in the timetable, but the authors did not mention the effect of changing the mutation rate [17].

In [18], a hybrid genetic hill-climbing algorithm with an embedded elitist mechanism was used to solve the lecturer course timetable problem. Hill-climbing optimization was implemented in the mutation phase to enable an LS. Hill-climbing optimization offers fast convergence and the capability to avoid local optima. The algorithm requires a longer time because hill-climbing optimization is frequently used. However, it does not achieve a fitness value of one, which indicates no conflict. It reaches when the population size reaches 80, but the execution time increases significantly [18].

A pragmatic algorithm implementation is presented in [19] to solve the university course timetabling problem. The algorithm was implemented as a web application deployed on the Azure cloud. The system was based on the manual assignment of courses to time slots and rooms. The objective was to engineer a timetable system programmatically as a web-based system [19]. A man–machine interaction system based on modeling the problem. The proposed algorithm uses column-generation heuristics to collect generated timetable columns by employing the relaxed integer programming method. The modeling timetable works for a reasonable number of instances; however, it may face computational complexity at a larger scale [20].

The authors in [21] presented a greedy algorithm combined with a genetic fusion algorithm to efficiently solve the course timetabling problem. It can obtain the local optimal solution using the greedy algorithm, which is efficient for generating the initial population used in GAs. A greedy algorithm includes an adaptive heuristic search combined with an evolutionary search algorithm. The approach was implemented on simulated data with a small population of approximately 20. Therefore, its effectiveness on a larger population must be evaluated and the fitness values must be compared [21].

Automatic timetable generation using GA with dynamic chromosomes was implemented in [22]. The algorithm suggests using a nonfixed chromosome size, which is adjusted according to the number of courses in the department. The algorithm implements the roulette wheel algorithm to select chromosomes. It uses harder and softer constraints to solve the room-allocated-to-time-slot problem. The performance results and implementation process of the algorithm showed that it was ineffective [22].

A hybrid genetic-based discrete PSO algorithm with two LS algorithms, LS and tabu search, was used in [23]. The objective was to enhance the performance for search solutions. The concept of PSO is based on a swarm of birds (particles) that search for food in an open space. Without any prior knowledge of the space, the birds spread and begin their search in a random space, and the position of all particles corresponds to the candidate solution for an optimization problem. Therefore, all particles are assigned a fitness function, which is determined according to the corresponding problem. Even if each particle moves to a new location in the search space, it maintains the optimal local information. Furthermore, each particle maintains its own information, shares information with the other particles, and maintains the best global information. Subsequently, each particle updates its velocity and position to correspond to the optimal local and global information, moves forward to the optimal value, and seeks the optimal solution. Tabu search is used to improve the solution quality after the LS operation is completed. the GA operation is applied (crossover and mutation) to enhance the global search. However, the execution time of the algorithm increases when a complex timetable model is introduced. This algorithm can be used for the room allocation problem [23].

IV. The Description of the Problem

The process of generating course timetables is classified as repetitive or periodic. However, we discovered that the academic departments in some universities in our region still manually generate course timetables. The entire procedure depends on humans playing the primary role in creating and maintaining such timetables.

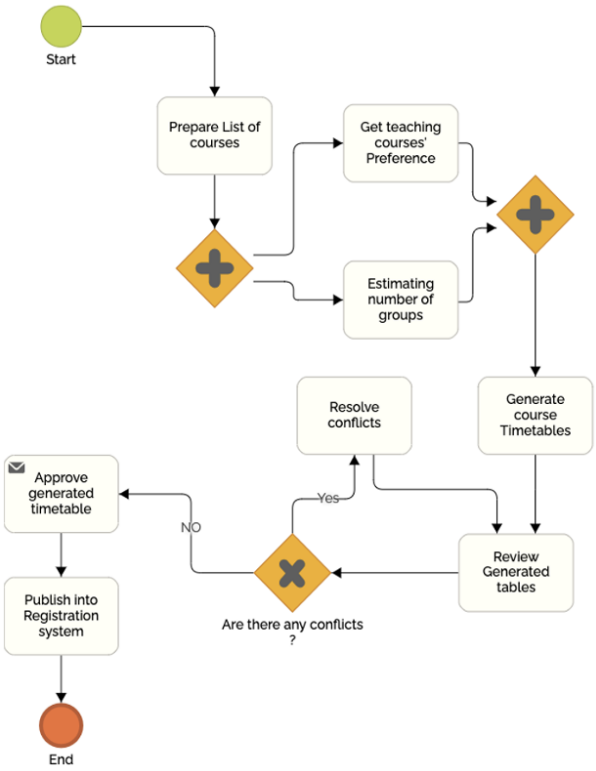


Fig. 2. The Business Process for Generating Course Timetables.

Fig. 2 shows the framework of business process modeling used to capture the common activities for producing course timetables. The process starts when the staff in a department prepare a list of courses to be taught in the next term. This is followed by collecting course preferences from teaching staff and estimating the number of students who will register for a course. Professors must also be assigned to their selected courses. After the preparation process, timetables are manually generated using spreadsheet software. This process is time-consuming because it requires several review cycles to ensure that timetables do not include any conflicts among groups of students, as shown in Fig. 2. Following the timetable generation step, the faculty administration approves the created course timetables and publishes them on a university course management system with a centralized software architecture.

The student course registration procedures also employ manual processes. It is the students' responsibility to create their entire timetable using the university registration system. At the beginning of each term, students should follow the curriculum plan published by the department after the selected courses are approved by their academic advisers. The student searches for targeted courses and selects a suitable teaching group based on their preferred time slots. If any conflict occurs in time slots, the student keeps changing the teaching groups until a conflict-free timetable is created. We noticed that manually creating and maintaining course timetables re-

sults in serious issues. One such issue is the occurrence of time-slot conflicts among courses, which affects the student registration process. Another problem is the increased number of groups in courses that contain a small number of students, because students have complete freedom to select their desired group at a suitable time slot. Hence, we have observed a misdistribution of student problems, which appears after the registration process is completed. This leads to insufficient resource allocation. After examining these common issues over several years, we conclude that manually generating timetables is time-consuming, laborious, and complex. The process creates stress among staff and students and thus needs to be addressed. Universities use VMS, wherein it is difficult and expensive to include automatic features. Thus, a system that includes optimization methods for timetable generation is required, which can be easily integrated with the registration management system.

V. Proposed Solution

This study aimed to solve the problems mentioned in the previous section by developing a service-based system comprising components responsible for publishing and generating timetables. The service-based approach is characterized by a decentralized architecture that is scalable and sufficient for the architecture used by universities. The system interacts with a university management system only to request data and publish the student and staff timetables it generates. The interaction is achieved through messages/requests and responses using RESTful web services. The data are transmitted in the JSON format. Therefore, the proposed service-based system works as a standalone feature that can be used by university departments. Each department has its own deployed service-based component to generate a timetable. In the next section, we explain service-based architecture in more detail. Furthermore, the proposed solution includes the design of a suitable algorithm for automatically generating timetables. Our objective was to eliminate the number of manual activities performed by the staff. Therefore, we designed a heuristic GA to create a departmental timetable that considers a set of hard and soft constraints. A detailed explanation of the proposed GA is presented in Section VI.

A. The Architecture of the System

Fig. 3 shows the main components of the automatic service-based system used to generate timetables. The system comprises five main components: a data notification center, notification service, data communication center, data communication service, and faculty autogen timetable service.

1) **Data Notification Center:** This component is responsible for broadcasting a scheduled and timed notification to all active notification services. It also receives replies and notification messages from notification services when the timetable generation processes are completed.
2) **Data Communication Center:** This is the communication channel between the university management system and the communication service. The communication center is responsible for filtering the requests for data from the communication services and transforming them into query passed as a message to the university management system.
3) **Notification Service:** This is an active component deployed in each university academic department. It is responsible for sending and receiving notification messages from the data notification center.
4) **Data Communication Service:** This component filters student and academic staff data and generates a timetable data format to send and receive data from the data communication center.
5) **Faculty Autogen Timetable Service:** This is the primary component responsible for automatically generating course timetables based on a set of requirements. It includes a GA to create an optimized timetable. The following Section provides a detailed explanation of this component.

B. The Architecture of the Faculty Autogen Timetable Service

The component responsible for automatically generating an optimized course timetable comprises various elements, as shown in Fig. 3. The service consists of two components: a **data filtering service** and **timetable generator**. The **data filtering service** handles received and sent data and is composed of three separate entities: course description filtering, department student filtering, and assigning student and faculty timetables.

1) *Course Description Filtering:* This entity is used to store the program course plan. The department can add courses, course details, and any updates regarding the program plan. The courses and plans are stored as JSON files.
2) *Department Students Filtering:* All the student data required to generate a timetable is received from the university management system database through the data communication service and filtered in this component. Filtering is required to determine the courses that students are expected to register for in the next term. Additionally, the component generates groups of students expected to opt for a course in the next term.
3) *Assigning Students and Faculty Timetable:* This component is responsible for assigning each student and academic staff member a timetable after the general timetable generated by the timetable generator component is approved.

The second component of the faculty autogen timetable service is a **timetable generator**. This is an essential service that creates an optimized course timetable. As shown in Fig. 4, this service comprises the following five entities:

1) *Department Criteria and Hard and Soft Constraints Configuration:* This entity configures the algorithm parameters, as well as the hard and soft constraints used in the customized scheduling algorithm.
2) *Customized Scheduling Algorithm:* This entity is the core of the timetable generator. It includes the implementation of the GA used to generate the expected timetable for the specific teaching plan of a department. Section VI describes the design of the GA in detail.

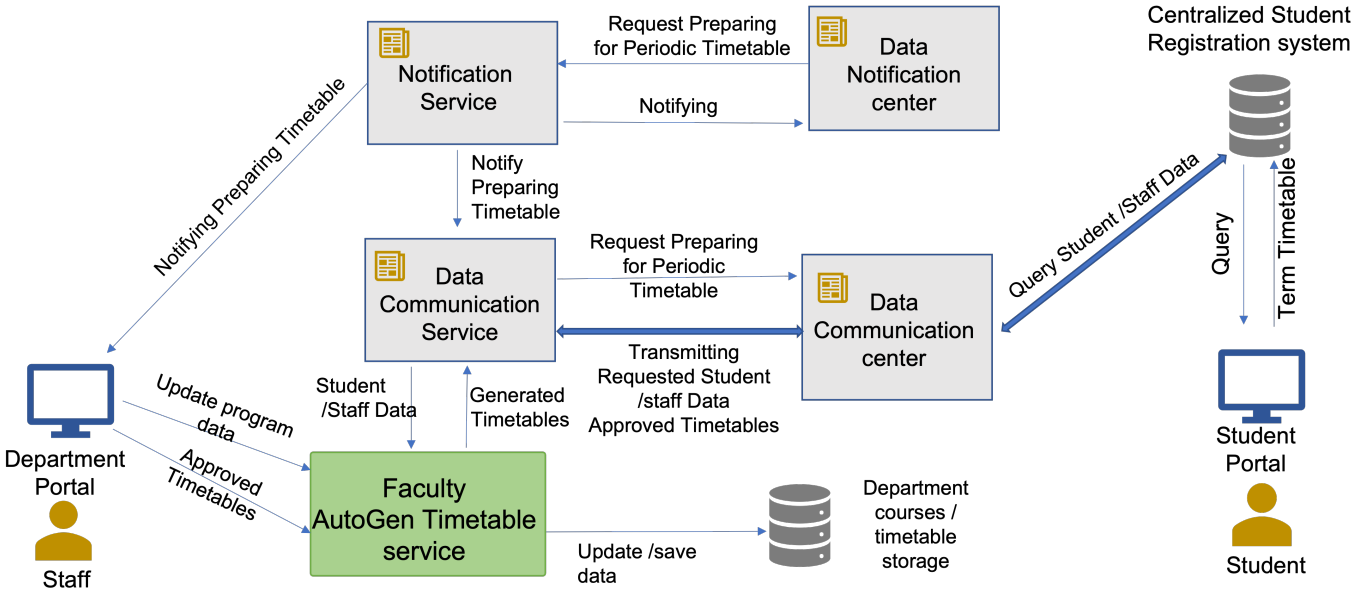


Fig. 3. The Architecture of the Service-Based System for Auto-Generating Course Timetables

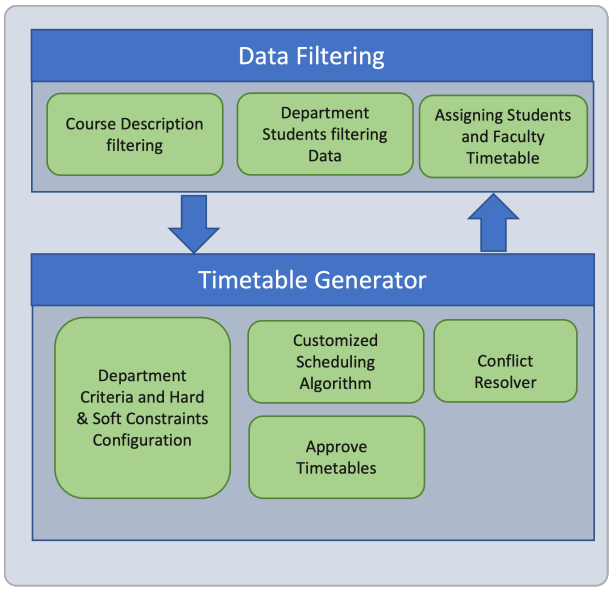


Fig. 4. The Architecture of the Faculty Autogen Timetable Service

3) *Conflict Resolver:* This component runs if the timetable generator is suitable for the department but contains several soft constraints that are violated. It allows for manual resolution of timetable conflicts.
4) *Timetable Approval:* This component obtains approvals of the generated timetables and divides them into sub-timetables based on the level to which they can be suitably applied.

VI. Genetic Algorithm Design

This section explains the design and implementation of the GA in detail. Before presenting the designed algorithm, it is necessary to mathematically model the operating environment. Furthermore, we are required to extract the possible hard and soft constraints required for optimization.

A. The Courses Timetable Mathematical Model

We mathematically modeled the courses timetable problem based on the common requirements and environments of our universities as follows :

A teaching program of a department was divided into levels. These levels comprised a set of courses. During the autumn term, all odd levels are taught, whereas during the spring term, even levels are taught. For simplicity, the summer term or special cases were not included. These levels are denoted as $L = \{l_1, l_2, l_3, ..., l_n\}$.

Any level l_i includes several courses $C = \{c_1, c_2, c_3, ..., c_m\}$, and student groups $G = \{g_1, g_2, g_3, ..., g_k\}$. Additionally, there should be no more than three groups per level. Any course c_i has one or more and no more than three sessions per week. Therefore, the sessions are denoted as $S = \{S_1, S_2, .., S_l\}$. Moreover, two types of sessions were defined, theoretical and practical, based on course type and number of teaching hours. Any course c_i is associated with teaching professor tp_i. The teaching professor tp_i who was selected from $TP = \{tp_1, tp_2, ..., tp_k\}$.

The environment model can be summarized as follows:

1) Any teaching program comprises a set of levels as follows:

$$L = \{l_1, l_2, l_3, ..., l_n\} \quad (1)$$

2) Any level l_i has several courses and groups of students and is associated with a teaching professor as

follows:

$$C = \{c_1, c_2, c_3, ..., c_m\}$$
$$G = \{g_1, g_2, g_3, ..., g_k\} \quad (2)$$
$$TP = \{tp_1, tp_2, ..., tp_r\}$$

3) Any course c_i has one or more sessions as follows

$$S = \{S_1, S_2, .., S_l\} : l \leq 3$$
$$s_i = Theorical \ or \ s_i = Practical \quad (3)$$

We want to emphasize that the algorithm proposed in this study does not consider the allocation of lecture rooms. This will constitute a further step in the system in the future, or will be handled as a separate problem. The existing timetable algorithms discussed in Section III consider room allocation to be an essential part of the course timetable allocation algorithm. However, in our case, room allocation always occurs after a suitable timetable is generated. Furthermore, our algorithm contains some factors related to building management, for which course timetables with courses and time slots assigned must be generated first.

B. The Hard Constraints

To create a suitable timetable, we defined a set of hard constraints that must be satisfied and a set of soft constraints that should not be violated. We denoted the set of hard constraints as H. The teaching professors and groups of students are defined as follows:

1) A teaching professor, tp_i must not teach two different courses in the same time slot on the same day.
2) A group of students g_i must not be assigned two different courses at the same time slot on the same day.
3) A group of students must not have two sessions of the same course on the same day. This condition has also been applied to the teaching professors assigned to the course.
4) Teaching professor tp_i who is an assistant professors or above, should have at least one day off.
5) Group of students g_i must not have more than six hours of theoretical lectures per day.

C. The Soft Constraints

A finite set of soft constraints S, which incur a small weighted plenty value when violated, is defined.

1) Teaching Professor tp_i might prefer to start the day in the morning or afternoon, according to their conditions.
2) Any practical session $s_i \in P$ for a course c_i can be started in the late morning time slot.
3) A group of students g_i should not have a gap longer than two hours between lectures.

D. Objective Function

The optimization problem is to maximize the assignment of all available classes to the sets of defined time events, sections, and teaching professors to assign the maximum number of courses possible from the preferred list. It can be expressed as:

$$f(x) = Maximize \sum_{i=1} Classes_i \quad (4)$$

The following are the defined decision variables:

1) v_1 = event day
2) v_2 = event time slot

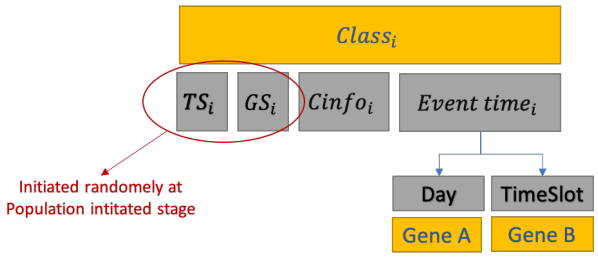


Fig. 5. Class Representation.

E. The Algorithm Design

A GA must contain chromosomes. To encode chromosomes with a course timetable, a set of classes must be generated. Using the mathematical model, specific courses and sessions at each level are defined as described in Section VI-A, and a set of targeted classes is formulated and denoted as $Classes$ as follows:

$$Classes = \{class_1, class_2, class_3, ..., class_m\} \quad (5)$$

Fig. 5 shows the class representation and its formulation. As shown in the figure, any $class_i$ is assigned a teaching professor TP_i, a group of students g_i, a type that can be theoretical or practical, and a duration d_i. Additionally, $class_i$ such as course name and number. The class type, duration, and other details are combined into class information $Cinfo_i$ as shown in Fig. 5. Moreover, these classes must be allocated to a particular time event. A time event includes assigning one of the five working days and a time slot, $ts = \{t_{start}, t_{end}\}$.

The algorithm starts with an initialization stage, where in objective is to determine the number of student groups and generate the classes possible, including sessions. Furthermore, the initialization step entails assigning teaching staff to the classes. The selection of teaching staff is based on the ranking obtained from the preference table. Based on this knowledge, a vector class that comprises all classes that can be taught for all programs at all levels is generated, as shown in Fig. 5 .

The GA must allocate the day and time slots for each class in the vector class without violating the hard constraints and no or minimum violation of the soft constraints. After formulating the real class timetable model, the chromosome and genes used in the algorithm must be formulated, as explained in the following section.

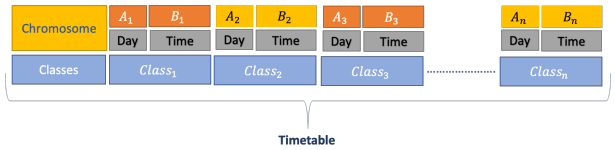


Fig. 6. The Chromosome Representation.

Fig. 7. Representation of the Crossover Operation.

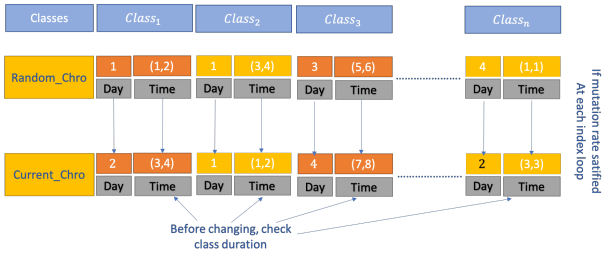


Fig. 8. Representation of the Mutation Operation.

F. The Chromosome Representation

As shown in Fig. 6, a timetable comprises a set of classes and a chromosome. The chromosome contains a vector consisting of genes. There are two types of genes. The first, denoted as A_i, contains a numerical integer that represents the day of a week. The second, denoted as B_i, is a collection of two numerical values representing the time slot. The time slot represents the start and finish times of a class. Chromosome length is twice the class vector length, which is calculated as follows:

$$chromosome_length = \sum_{i=0}^{n} Class_i * 2 \tag{6}$$

G. The Fitness Function

It is important to include a fitness function in the GA because it is used to measure the accuracy of a specific solution. In our case, a good solution should have a fitness value of one, which indicates that the generated timetable does not violate either the hard or soft constraints. Thus, the fitness value is computed as follows:

$$fitness = \frac{1}{(\sum_{i=1}^{n} \omega_i V_{hard} + \sum_{j=1}^{m} \sigma_j V_{soft}) + 1} \tag{7}$$

Both V_{hard} and V_{soft} represent the violations of hard and soft constraints, respectively, occurring in the solution. ω and σ denote the penalty weights associated with the hard and soft constraint violations, respectively. Any violation of the hard constraints is assigned a larger weight, whereas that of the soft constraints is assigned a lower weight.

H. Parent Selection and Genetic Operations

Crossover and mutation are the two basic operators of GAs. We used a uniform crossover and uniform mutation, as shown in Fig. 7 and 8, respectively. The crossover operation continues throughout the population and employs the parent selection process, which is based on the tournament selection technique [7] [24]. After parents with effective fitness values are selected, crossover is performed on them. Half of the genes from each parent are randomly selected to generate new offspring. Fig. 7 shows an example of a crossover operation, wherein genes are randomly selected from chromosome parents 1 and 2. In this example, the first gene, which represents the day for class 1, is obtained from parent 2, whereas the time slot is obtained from the time-slot gene of parent 1. We want to emphasize that the class ordering is similar for both parents. Thus, we ensured that no conflicts occurred If the chromosome does not go through crossover, it will be inserted directly into the new population.

The elitism method was applied to ensure that the best solution from the old population was not lost and the GA was optimized between class duration when the time slots are changed. If the chromosome does not undergo crossover, it is inserted directly into the new population. This method allows individual chromosomes with the best fitness values to be kept unaltered and moved to the next generation. However, we retained only a small number of elite chromosomes compared with the population size and controlled the value using the elite window size.

For the mutation operation, we used the value of mutation rate. When satisfied, a chromosome that allocates the days and effective time slots for the intended classes. The gene values of the chromosome selected for alteration were then replaced with the values generated by the random chromosome. Fig. 8 an example of replacing the gene values of the chromosome selected for mutation with those of a randomly generated chromosome.

VII. Implemnation and Expriments

The designed GA was implemented as an application programming interface (API) written in JAVA programming language. The API can be integrated into services that can be easily integrated into any university management system. The course, staff, and student data were included in a JSON file, which enables easy transmission to and from the university management system. All system components were implemented as RESTful web services when deployed on a docker container. They include the faculty autogen timetable service, data communication service, notification service, data notification center, and data communication center. However, a more challenging problem is the implementation of a customized scheduling algorithm, which depends on the GA implementation, as explained in Section VI. Therefore, this component was constructed as a standalone JAVA program for testing and experimentation, as explained in the following section. After the experiments, the program was transferred

TABLE I. THE EXPERIMENTS' CONFIGURATION

Tests	Timetable Parameters				Algorithm Parameters					Tests #
	Courses #	Level #	staff #	Groups	Population Size	Crossover Rate	Mutation Rate	Elite #	Elite Size	
Experiment 1	21	4	19	one group at each level	100,150,300,500	0.7	0.01	2	10	19
Experiment 2	21	4	19	varied groups (2,2,2,1)	100,150,300,500	0.7	0.01	2	10	19
Experiment 3	21	4	19	one group at each level	100,150,300,500	0.5	0.1	2	10	18
Experiment 4	21	4	19	varied groups (2,2,2,1)	100,150,300,500	0.5	0.1	2	10	18

Tests	Constraints		Run
	Hard	Soft	
All Experiments	5	3,2,0	15

Algorithm 1 Random Course Section Generation

Require: CoursesInfoList, ExpectedStudentList, LevelsList
Ensure: CourseInfo with Generated Sections
  for level in Level List **do**
    Extract all expected students in the level associated with their expected courses to take
    for course in CourseLevelList **do**
      Count Number of Students
      if $TotalStudent \leq MaxCapacity$ **then**
        Generate only one section
      else if $TotalStudent > MaxCapacity$ **then**
        $Sections = TotalStudents/SectionCapacity$
        for section in Sections **do**
          Pick Random Students from Expected Course List
          Add Picked Student Random in Section List
          Update Students Info with course and section
        end for
      end if
    end for
  end for

Algorithm 2 Assign Teaching Professors

Require: Teaching Prof. List, TP Preference List , CoursesInfo List
Ensure: CourseInfo Assigned TP
  Sort TP list based on Rank and Experience
  Order Preference Ranked List
  for TP_i in TP List **do**
    $Courses \leftarrow FindChoicesinCourseInfoList$
    for $choice_i$ in TP Preference List **do**
      if Check Courses with $choice_i$ is not assigned **then**
        Assign TP to Course with its sections
        Update teaching hours
      end if
    end for
    if TP remaining teaching hours $\neq 0$ **then**
      randomly picked unassigned course with sections and assigned it TP_i
    end if
  end for

Algorithm 3 Timetable Generation using GA

Require: ClassesList
Ensure: Timetable
  Initiate Population by:
  Generate Classes
  Initialize Chromosome for each Individual
  for Individual in Individuals **do**
    for $Class_i$ in Class List **do**
      Generate Random Time Slot and Day
      Assign Generated Chromosome to Individual
    end for
  end for
  Evaluate Randomly generated population
  while Termination Condition in not met **do**
    Call Crossover Population ▷ See Fig. 7 explaining used crossover operation
    Call Mutation Population ▷ See Fig. 8 explaining used mutation operation
    Call Evaluate Population ▷ by calculating fitness values using Equation 7
  end while

into a module included in timetable generator APIs. The following subsection explains the experimental environment and configuration.

A. The Experiment Environment

A MacBook Pro laptop with a 2.8 GHz Quad- core Intel Core i7 processor and 16 GB RAM was used. During the testing process, all ineffective processes were terminated. A tool was integrated within the API to measure the execution time of the algorithm and record the time required to obtain a solution. Furthermore, the algorithm was configured to enforce termination if it reached 1,000 generations without finding a solution.

B. The Configuration for Experiments

The objective of the experiment was to determine the ideal settings for the algorithm parameters (population size, crossover rate, and mutation rate) that are suitable for creating a course timetable without violating either hard or soft constraints. Furthermore, we wanted to assess the impact on the algorithm performance when the number of student groups increased. Therefore, we focus on the following two special impacts:

1) Factors that might affect the algorithm if the number of soft and hard constraints increases.
2) The extent to which an increased number of student groups might affect the execution time of the algorithm.

The experiment was divided into four sub-experiments: Experiments 1, 2, 3, and 4, each of which comprised 12 tests. Each test assumed a population configuration and tested few constraints. For example, in Test 1, Experiment 1 assumed a population size of 100 and tested eight constraints (five hard and three soft), whereas in Test 2, Experiment 1 assumed a population size of 100 and tested six constraints (five hard and two soft). The same process was followed for all the tests in each experiment. Each test was performed 15 times to ensure the accuracy. Experiment 1 assessed the execution time and average of conflicts when the mutation and crossover rates were fixed at 0.01 and 0.7, respectively, with respect to the changes in population size and the numbers of soft and hard constraints. We selected population sizes of 100, 150, 300, and 500; 3, 2, and 0 soft constraints; and five hard constraints. The experimental configurations are listed in Table I.

Experiments 1 and 2 aimed to measure the execution times and average conflicts when the mutation and crossover rates were set at 0.01 and 0.7, respectively. Both experiments included four tests, each of which was used to examine the effects of different population sizes and configurations of the number of soft constraints. Each test was performed 10 times. The difference between Experiments 1 and 2 was the number of students groups: Experiment 1 was performed with each level containing only one student group, whereas Experiment 2 was performed using various numbers of student groups, where the three levels contained two student groups. Experiments 3 and 4 followed a similar strategy as the first two experiments, but the mutation and crossover parameters were changed to 0.1 and 0.5, respectively. The objective was to determine the effect on the execution time and average conflicts when this conflict rate was used and the population size was changed. In the final two experiments, we focused on the effect on execution time and conflict rate if the number of courses was increased from 21 to 30 (a major effect on the increasing number of courses). Both experiments were performed using two population sizes, 100 and 200, and changing the number of soft constraints.

VIII. Results and Discussion

After conducting the experiments described in Section VII, we measured the average execution time and the average number of conflicts with either hard or soft constraints. The results of all four experiments are shown in Fig. 9 and Fig. 10. Fig. 9 contains four subfigures that show the average execution times in milliseconds for all experiments, whereas Fig. 10 contains four subfigures that show the average numbers of constraint conflicts that occurred in the experiments.

Fig. 9a and Fig. 9b show the average execution times of Experiments 1 and 2, respectively, indicating that the average overall execution time of Experiment 1 was lower than that of Experiment 2. Both experiments were performed using crossover and the mutation rates of 0.7 and 0.01, respectively. The main difference in Experiments 1 and 2 was the number of student groups. In Experiment 1, each level contained only one student group for each teaching course, whereas in Experiment 2, the number of student groups in each course varied. Hence, the algorithm required a longer execution time to obtain a solution. Additionally, an increase in the number of hard and soft constraints affected the execution time.

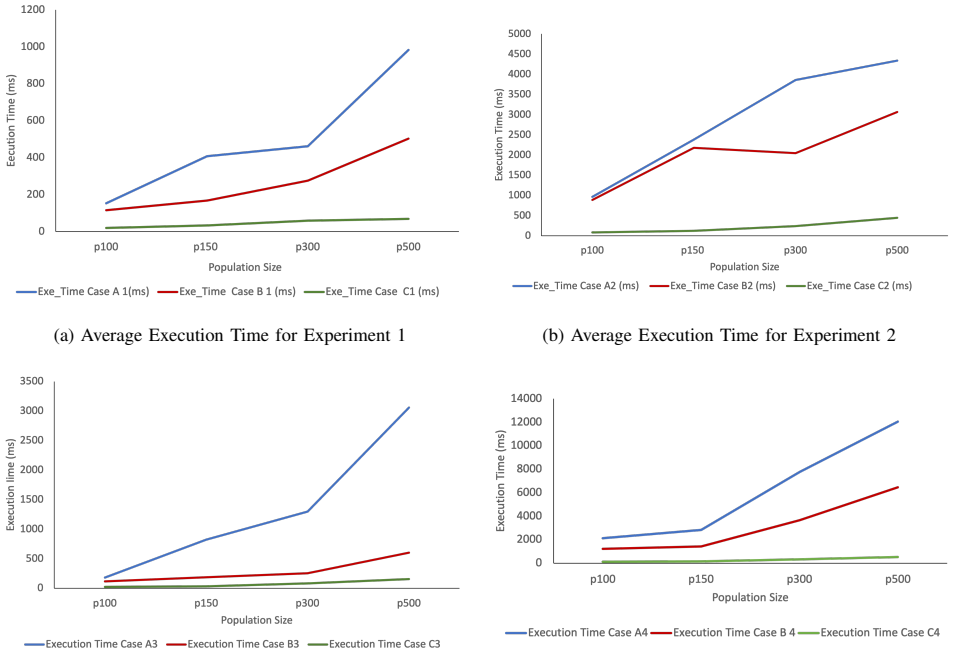
Fig. 9a shows the effect of increasing the population size and number of tested hard and soft constraints on the execution time. In Case A, the execution time resulted from testing five hard and three soft constraints; in Case B, it resulted from testing five hard and two soft constraints; and in Case C, it resulted from testing only five hard constraints.

For sub-figures Fig. 9b - 9d, we observed that the execution time for the path in Case C1 was lower than those for both Cases A1 and B1. The execution time for Case A1 path was the longest, whereas that for Case B1 was between those of the others. Looking deeply in Fig. 9b, the execution times were longer in the paths shown in these figures because the number of student groups and crossover and mutation rates were changed. Currently, we have determined that the algorithm execution time is affected by an increase in the number of tested constraints. However, if five hard constraints are considered to not have been violated, the average algorithm execution time can be maintained below 30 ms in all sub-figures. In Fig. 9c and Fig. 9d, execution time for the Case C paths was less than 100 ms, owing to the alterations in the algorithm parameters.

We observed that the execution time was affected when the population and the number of tested constraints were increased. Regarding the execution paths of Cases A1–A4 shown in Fig. 9a - 9d the execution time increases dramatically when the population exceeded 300. Thus, we conclude that if the population is set at 500, the algorithm would not obtain effective results in terms of time execution. In fact, it will take approximately 1,000 ms, as shown in Fig. 9a and close to 4,500 ms if the number of student groups is increased, as shown in Fig. 9b. These results would not be affected if the crossover and mutation rates were changed; the execution time for a population of 500 could be above 3,000 ms for a single student group in each course, as shown in Fig. 9c, and can reach over 12,000 ms for various numbers of student groups, as shown in Fig. 9d.

Therefore, it can be determined that the algorithm can generate the best results if the population size is set between 100 and 150 with single or varied number of student groups. This result can be maintained even if both the crossover and mutation rates are changed. Furthermore, the algorithm can produce effective results if a lower number of constraints is maintained, close to five. Furthermore, we assume that the algorithm can generate a reasonable solution if the number of constraints is set to six. However, if the number of constraints is set to eight or more, the algorithm can execute within a reasonable time if the population rate is maintained between 100 and 150, which will reduce the search time.

The conflict rates shown in Fig. 10 for each course are lower than those for the various numbers of student groups assigned to each course. The average number of conflicts shown in Fig. 10a is lower than those in the other figures. We observed that, if the population size was set to 100, the algorithm generated a solution without violating any of the



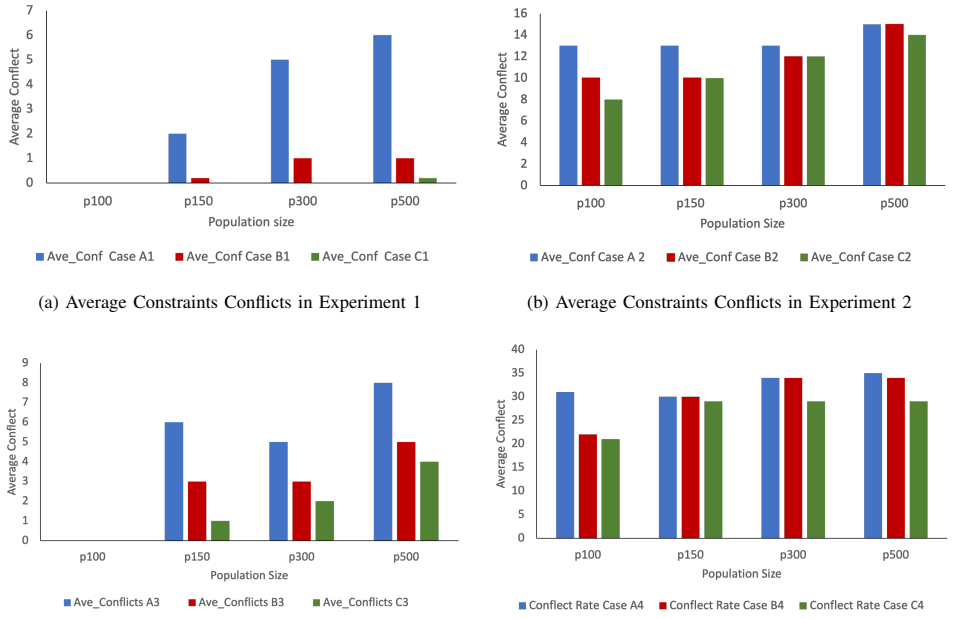
(a) Average Execution Time for Experiment 1

(b) Average Execution Time for Experiment 2

(c) Average Execution Time for Experiment 3

(d) Average Execution Time for Experiment 4

Fig. 9. The Average Execution Time in Milliseconds for All Performed Experiments Experiment 1, Experiment 2, Experiment 3 and Experiment 4



(a) Average Constraints Conflicts in Experiment 1

(b) Average Constraints Conflicts in Experiment 2

(c) Average Constraints Conflicts in Experiment 3

(d) Average Constraints Conflicts in Experiment 4

Fig. 10. Averages of Hard / Soft Constraint Conflicts in All Experiments Performed

eight tested constraints. Furthermore, relatively lower number of conflicts can occur if the number of violations is between zero and two. This could have occurred if the population size was 150

and there were six to d eight tested constraints. We observed that if the number of tested constraints was five, there would be no conflicts. A similar result was obtained e by setting the population sizes to 300 and 500. We observed that conflicts appeared six or more constraints were applied, and the number of conflicts were higher than those when the population size was set to 150. Thus, we can conclude that the algorithm can produce effective results if the population size is

set to 100, even if the numbers of courses and staff increase. Furthermore, the algorithm would obtain a reasonable result when the population size was set at 150.

The conflict rate results shown in Fig. 10b were used to test the violating number of student groups using similar settings for crossover and mutation rates, as shown in Fig. 10a. We observed that the conflict rate appeared for all quantities of tested constraints. For example, while examining the conflict rate for a population size of 100, the average number of conflicts for testing five constraints was close to eight. If the number of tested constraints was between 6 and 8, the conflict rate was between 10 and 14. Nevertheless, when the population was set at 150, the conflict rate was higher than when it is set at 100. Similar results were obtained when the number of tested constraints was five and above. As shown in Fig. 10b, if the population size is set at 300 and 500, the conflict rate for all the tested constraints is between 12 and 16. Therefore, for a situation wherein various numbers of student groups are tested, a good result can be obtained if the number of tested constraints is maintained at five or lower for a population size of 100. However, the algorithm must be improved to handle various situations more effectively by partitioning the search for each level. However, this requires further investigation and will be addressed in a future study.

The final test involved changing the crossover and mutation rates, as shown in Fig. 10a and 10c. By comparing the results shown in Fig. 10a and 10c it can be observed that there the conflict rates did not differ if the population size was set at 100 in both situations. However, the conflict rate shown in Fig. 10c is higher than that in Fig. 10a.

To summarize our findings:

1) The effective population size that will lead the algorithm to produce results without any violation of the constraints and with minimum execution time is 100, and the crossover and mutation rates are set at 0.7 and 0.01, respectively. This occurs even if both the courses and the number of teaching staff are increased at each level. However, we aim to make the algorithm work with more constraints, especially user-defined constraints. This requires further improvement, which will be the subject of future study.
2) We also discovered that if the number of student groups is increased at each level, the execution time and conflict rates increase significantly. To handle more student groups at each program level, we will develop a partitioning process for inclusion within the proposed algorithm, which will be executed only if more groups need to be allocated. However, this also requires further investigation.

IX. Conclusion

In this study, we looked at the problem for manual generation for courses timetable at universities that uses Vendor Management System (VMS) as centralized system for course registration. To solve the problem, we proposed a decentralized automatic course-timetable-generation service architecture uses Genetic algorithm as a core for generating courses timetable. The system can easily be integrated into university management systems. The architecture comprises several components: a data notification center, data communication center, notification service, and faculty autogen timetable service. The faculty autogen timetable service comprises several components, of which an essential component is customized timetable generation, which implements using Genetic Algorithm. The proposed decentralized software architecture in the paper can handle scalability issues introduced by integrating architecture with universities VMS. In addition, to make automatic generation a customized course model was used as implemented component. The customized course model includes a set of defined common hard and soft constraints.

However, the objective was to customize the constraint definitions based on the requirements of each department. The purpose of the GA is to allocate the courses that the student groups have signed up for to teaching staff in feasible time slots. During the initialization step, the class population was generated based on the number of student groups that had signed for it, and the teaching staff were randomly allocated based on their rankings. The objective was to allocate these classes to events. The algorithm was tested through a number of experiments to determine the suitable parameters for generating feasible solutions. The algorithm generated a feasible solution with a fitness value of 1 when the population number was set to 100, and the crossover and mutation rates were set to 0.7 and 0.01, respectively. Furthermore, we observed that if the number of student groups increased with no increase in the number of teaching staff, the conflict rate increased. Therefore, to improve the algorithm, we must analyze cases wherein the number of students groups taking a course is increased, which will be the subject of a future study. If a timetable is selected and the number of conflicts is constrained, the autogeneration component has a conflict resolution algorithm that either manually fixes the department staff conflicts or suggests enhancement by omitting soft constraints that can be violated.

A future direction for this study is to propose a parallel GA to enhance the component and handle a higher number of student groups taking the same course. We will also investigate the possibility of building customized hard and soft constraints to add more flexibility in generating course timetables. This requires a method to define a domain-specific language to build constraints and automatically transform them into executable constraints.

References

[1] Ellucian. (2021, 11) Ellucian student banner. [Online]. Available: https://www.ellucian.com/solutions/ellucian-banner

[2] E. A. Abdelhalim and G. A. El Khayat, "A utilization-based genetic algorithm for solving the university timetabling problem (uga)," *Alexandria Engineering Journal*, vol. 55, no. 2, pp. 1395–1409, 2016.

[3] P. Guo, J.-x. Chen, and L. Zhu, "The design and implementation of timetable system based on genetic algorithm," in *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*. IEEE, 2011, pp. 1497–1500.

[4] H. M. Mousa and A. B. El-Sisi, "Design and implementation of course timetabling system based on genetic algorithm," in *2013 8th International Conference on Computer Engineering Systems (ICCES)*, November 2013, pp. 167–171.

[5] T. Erl, *Service-Oriented Architecture : Analysis and Design for Services and Microservices*, ser. The Pearson Service Technology Series from Thomas Erl, 2, Ed. Pearson Education, 2016.

[6] N. Josuttis, *SOA in Practice: The Art of Distributed System Design*, ser. Software engineering. O'Reilly Media Inc., 2007.

[7] S. Sivanandam and S. Deepa, *Inroduction to Genetic Algorithm*. Springer-Verlag Berlin Heidelber, 2008.

[8] F. Buontempo, *Genetic Algorithms and Machine Learning for Programmers: Create AI Models and Evolve Solutions*, ser. Pragmatic programmers. Pragmatic Bookshelf, 2019.

[9] N. Leite, F. Melício, and A. C. Rosa, "A fast simulated annealing algorithm for the examination timetabling problem," *Expert Systems with Applications*, vol. 122, pp. 137 – 151, 2019.

[10] A. Ahmad and F. Shaari, "Solving university/polytechnics exam timetable problem using particle swarm optimization," in *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication*, ser. IMCOM '16, January 2016.

[11] V. S. Kralev, R. S. Kraleva, and S. Kumar, "A modified event grouping based algorithm for the university course timetabling problem," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 9, no. 1, pp. 229–235, 2019.

[12] N. I. Ilham, E. H. M. Saat, N. H. A. Rahman, F. Y. A. Rahman, and N. Kasuan, "Auto-generate scheduling system based on expert system," in *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, November 2017, pp. 6–10.

[13] G. H. Fonseca, H. G. Santos, E. G. Carrano, and T. J. Stidsen, "Integer programming techniques for educational timetabling," *European Journal of Operational Research*, vol. 262, no. 1, pp. 28 –39, 2017.

[14] A. Rezaeipanah, S. Sechin Matoori, and G. Ahmadi, "A hybrid algorithm for the university course timetabling problem using the improved parallel genetic algorithm and local search," *Applied Intelligence*, vol. 51, January 2021.

[15] A. Lemos, F. S. Melo, P. T. Monteiro, and I. Lynce, "Room usage optimization in timetabling: A case study at universidade de lisboa," *Operations Research Perspectives*, vol. 6, p. 100092, 2019.

[16] P. Guo, J.-x. Chen, and L. Zhu, "The design and implementation of timetable system based on genetic algorithm," in *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, August 2011, pp. 1497–1500.

[17] P. Khonggamnerd and S. Innet, "On improvement of effectiveness in automatic university timetabling arrangement with applied genetic algorithm," in *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*, November 2009, pp. 1266–1270.

[18] M. Yusoff and N. Roslan, "Evaluation of genetic algorithm and hybrid genetic algorithm-hill climbing with elitist for lecturer university timetabling problem," in *Advances in Swarm Intelligence*, Y. Tan, Y. Shi, and B. Niu, Eds. Cham: Springer International Publishing, 2019, pp. 363–373.

[19] R. Ferdiana, N. Ridwan, and N. F. Hidayat, "A pragmatic algorithm approach to develop course timetable web application based on cloud technology," in *2017 7th International Annual Engineering Seminar (InAES)*, August 2017, pp. 1–5.

[20] H. Komaki, S. Shimazaki, K. Sakakibara, and T. Matsumoto, "Interactive optimization techniques based on a column generation model for timetabling problems of university makeup courses," in *2015 IEEE 8th International Workshop on Computational Intelligence and Applications (IWCIA)*, November 2015, pp. 127–130.

[21] K. Wang, W. Shang, M. Liu, W. Lin, and H. Fu, "A greedy and genetic fusion algorithm for solving course timetabling problem," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, June 2018, pp. 344–349.

[22] G. Alnowaini and A. A. Aljomai, "Genetic algorithm for solving university course timetabling problem using dynamic chromosomes," in *2021 International Conference of Technology, Science and Administration (ICTSA)*, March 2021, pp. 1–6.

[23] T. Unprasertporn and D. Lohpetch, "An outperforming hybrid discrete particle swarm optimization for solving the timetabling problem," in *2020 12th International Conference on Knowledge and Smart Technology (KST)*, 2020, pp. 18–23.

[24] H. A. L. Omid Bozorg-Haddad, Mohammad Solgi, *Meta-heuristic and Evolutionary Algorithms for Engineering Optimization*. Wiley, 2017.