MEKATRONIKA
e-ISSN: 2637-0883
VOL. 5, ISSUE 2, 23 – 31
DOI: https://doi.org/10.15282/mekatronika.v5i2.9973

ORIGINAL ARTICLE

Efficiency and Accuracy of Scheduling Algorithms for Final Year Project Evaluation Management System

Loo Chang Herng[1], M. Zulfahmi Toh[1,*], Ahmad Fakhri Ab. Nasir[1], Nur Shazwani Kamaruddin[1], and Nur Hafieza Ismail[1]

[1]Faculty of Computing, Universiti Malaysia Pahang Al-Sultan Abdullah, 26600 Pekan Pahang, Malaysia.

ABSTRACT – Scheduling algorithms play a crucial role in optimizing the efficiency and precision of scheduling tasks, finding applications across various domains to enhance work productivity, reduce costs, and save time. This research paper conducts a comparative analysis of three algorithms: genetic algorithm, hill climbing algorithm, and particle swarm optimization algorithm, with a focus on evaluating their performance in scheduling presentations. The primary goal of this study is to assess the effectiveness of these algorithms and identify the most efficient one for handling presentation scheduling tasks, thereby minimizing the system's response time for generating schedules. The research takes into account various constraints, including evaluator availability, student and evaluator affiliations within research groups, and student-evaluator relationships where a student cannot be supervised by one of the evaluators. Considering these critical parameters and constraints, the algorithm assigns presentation slots, venues, and two evaluators to each student without encountering scheduling conflicts, ultimately producing a schedule based on the allocated slots for both students and evaluators.

ARTICLE HISTORY
Received: 2nd Sept 2023
Revised: 20th Sept 2023
Accepted: 6th Oct 2023
Published: 24th Oct 2023

KEYWORDS
Scheduling Algorithm
Genetic Algorithm
Hill Climbing Algorithm
PSO Algorithm

INTRODUCTION

The Final Year Project (FYP) represents a vital component of higher education, being a mandatory requirement for many courses in universities worldwide. Successfully completing an FYP contributes to the development of students' soft skills, such as creative problem-solving and effective time management [1]. Nevertheless, the substantial volume of FYPs to be managed poses a daunting challenge for instructors, particularly during the evaluation phase [2]. The manual management of student project evaluations is both time-consuming and inefficient [3]. For instance, creating an evaluation schedule involving a large number of evaluators and students is a laborious task [3,4]. This significantly impacts the efficiency of the FYP coordinator, as they must manually handle and cross-reference all schedules with the evaluators' timetables. Within a final year project evaluation management system, the management of evaluation schedules stands as one of the primary modules. Automating the scheduling of evaluation slots not only conserves a significant amount of time for the final year project coordinator but also mitigates the risk of human errors such as scheduling conflicts, unavailability of evaluators, or the repetition of students in multiple time slots.

The evaluation process for a final year project is intricate and can be assessed from various perspectives, as outlined in reference [5]. It can be divided into multiple stages, including document evaluation, final product assessment, and presentation evaluation. Furthermore, the manual arrangement of the evaluation schedule presents its own set of challenges due to numerous parameters such as venue, time slots, students, evaluators, and constraints like evaluator availability and student-evaluator relationships. These constraints and parameters contribute to a more complex algorithmic concept, necessitating extended algorithmic logic to address this issue. In a later section, we will delve into the study of three scheduling algorithms: genetic algorithm, hill climbing algorithm, and particle swarm optimization. These algorithms have been selected for their widespread use in tackling scheduling problems, as indicated in reference [6].

PROBLEM DESCRIPTION

Handling schedules that involve numerous parameters and extensive datasets can be a challenging task, especially when done manually. Therefore, the implementation of scheduling algorithms becomes essential for managing the presentation schedule within the final year project evaluation management system. This approach not only accelerates the scheduling process but also alleviates the workload of the final year project coordinator while minimizing the potential for human errors. It is imperative to consider all parameters and constraints when generating the schedule. The choice of scheduling algorithm holds great significance as each algorithm possesses its own set of strengths and weaknesses. Consequently, a comprehensive study of three scheduling algorithms, selected from among the most commonly utilized ones, becomes essential for determining the optimal scheduling algorithm to be integrated into the system.

*CORRESPONDING AUTHOR | M. Zulfahmi Toh | ✉ zulfahmi@umpsa.edu.my

Parameters and constraints serve as fundamental elements within a scheduling algorithm, as noted in reference [7]. It is crucial to account for all these parameters and constraints, and this represents the most challenging aspect of creating a schedule without conflicts, as highlighted in references [8] through [10]. The parameters within the schedule encompass students, time slots, classroom venues, and the assignment of two evaluators to evaluate a single student in a room concurrently. The schedule is subject to several constraints, which include:

1. The student undergoing evaluation cannot have one of the evaluators as their supervisor.
2. Each student must receive evaluations from two evaluators belonging to the same research group.
3. The time slot allocated to an evaluator must not overlap with their teaching timetable.
4. The schedule must remain free of conflicts in terms of venue, time slot, and the presence of two evaluators.

METHODOLOGY

Genetic Algorithm

The genetic algorithm is widely applied across various sectors due to its adaptability, as outlined in reference [11]. It employs chromosome arrangements to represent different instances of a variable and employs this approach to establish a model for machine learning. It further enhances the output by addressing any conflicts or inadequacies. The process involves the use of mutation and crossover methods to refine the combinations in the next generation, based on defined probabilities [12]. When the fitness of a particular generation reaches a value of 1, indicating a perfect fit with the expected output, the reproduction of the next generation is halted, and the output represents the best solution. As a result, the genetic algorithm guarantees the identification of the best solution, ensuring that the generated schedule is free from conflicts. The pseudocode for the genetic algorithm in presentation scheduling is provided below.

ALGORITHM 1: GENETIC ALGORITHM

```
Input: List of students, venue, timeslot, evaluator1, and evaluator2
Output: Assigned schedule for each student
Initialization of variables: Assign 50 to NUM_GENERATIONS,
POPULATION_SIZE

function Tournament_selection(parameters: population)
    Randomly select 10 individuals from population.
    foreach selected individual do
        if best is null or fitness of the individual > fitness of best do
            best = individual;
        end if
    end foreach
    return best;
end function

function fitness (parameters: individual)
    Initialize conflicts to 0;
    get global variable of students.
    foreach student in students as student1 do
        foreach next student in students as student2 do
            if (evaluator1 and evaluator 2 of student1 is same) or (evaluator1 and
            evaluator 2 of student2 is same) do
                conflicts += 1;
            else if timeslot of student1 and student2 is same do
                if (venue of student1 and student2 is same) or (evalautor1 of student1 and student2 is same) or (evaluator2 of student1 and
                student2 is same) or (evaluator1 of student1 and evaluator2 of
                student2 is same) or (evaluator2 of student1 and evaluator1 of
                student2 is same) do
                    conflicts += 1;
                end if
            end if
        end foreach
    end foreach
    return 1 / (conflicts + 1);
end function
```

```
function crossover (parameters: parent1, parent2)
    Get global variable of students.
    foreach student in students do
        if random number >= 0.5 do
            assign all properties of parent1 to a child;
        else
            assign all properties of parent2 to a child;
        end if
    end foreach
    return child;
end function

function mutation (parameters: individual)
    Get global variable of students, timeslots, venues, evalautor1, and evalu-
    ator2;
    Initialize student as random selected student from array of students;
    foreach student in students do
        Assign random selected properties to individual[student];
    end foreach
    return individual;
end function

Generation of an 2-dimension array population from array of evaluator1,
evaluator2, students, timeslot, and venue;
for generation = 0 to NUM_GENERATIONS do
    Declare an empty array, new_population;
    while count of new_population is less than POPULATION_SIZE do
        parent1 = call function tournament_selection(population);
        parent2 = call function tournament_selection(population);
        child = call function crossover(parent1, parent2);
        if random number < 5 then
            child = mutation(child);
        end if
        push child into new_population;
    end while
    population = new_population;
    foreach individual in population do
        if fitness(individual)  1 do
            break;
        end if
    end foreach
end for
Declare variable best_individual;
foreach individual in population do
    if best_individual is null or fitness(individual) > fitness(best_individual)
    then
        best_individual = individual;
    end if
end foreach
end
```

The genetic algorithm has the potential to yield optimal results, albeit at the cost of demanding substantial computational power due to its high algorithmic complexity. This complexity arises from the presence of nested for loops, which significantly extend the execution time. Conversely, the genetic algorithm is relatively straightforward to comprehend and apply to real-world problems, requiring less time and effort for implementation in scheduling tasks.

Hill Climbing Algorithm

The Hill Climbing algorithm is a commonly used local search strategy that focuses on refining the search for an optimal solution based on an initial solution, as mentioned in reference [13]. Starting with a randomly generated initial solution, it proceeds to adjust the combination and assesses the resulting solution using a fitness function. Crashes or conflicts are identified through this fitness function and are evaluated by assigning a fitness value. It's worth noting that the specific method and algorithm used for evaluating fitness can vary depending on the problem at hand, which can impact the algorithm's efficiency [14]. Hill Climbing algorithm can also be employed in conjunction with other optimization algorithms such as simulated annealing or particle swarm optimization to enhance the quality of the solutions produced [15], [16]. The fundamental concept of implementing the Hill Climbing algorithm for this problem can be illustrated in the following pseudocode.

ALGORITHM 2: HILL CLIMBING ALGORITHM

```
Input: List of students, venues, timeslots, evaluator1, and evaluator2
Output: Assigned schedule for each student
Initialization of variables: Assign zero to best_fitness, iterations, assign 20
to max_iterations

function fitness_function(parameter: timetable
    Assign zero to conflict.
    foreach student1 in timetable do
        foreach student2 in timetable do
            If (student1 not equal to student2) and (timeslot of student1 and
            student2 is same) and (venue of student1 and student2 is same) do
                conflicts += 1
            else if timeslot of student1 and student2 is same and venue of stu-
            dent1 and student2 is different do
                if (evaluator1 of student1 and student2 is same) or (evaluator1 of
                student1 and evaluator2 of student2 is the same) or (evaluator2
                of student1 and evaluator1 of student2 is the same) or (evalua-
                tor2 of student1 and student2 is the same) do
                    conflicts += 1
                end if
            end if
        end foreach
    end foreach
    return 1 / (conflicts + 1);
end function

while best fitness < 1 do
    if iterations == 0 do
        Initialize a new timetable;
        Assign the timetable to best_timetable;
        Assign fitness(best_timetable) to best_fitness;
    end if
    Initialize an empty array, neighboring_timetables;
    foreach student details in timetable do
        foreach timeslot in timeslots do
            if timeslot of the student is not equal to the timeslot then
                Assign current timetable to neighboring_timetable;
                Assign current timeslot to timeslot of the student;
                Push the generated neighbouring_timetable into neighbor-
                ing_timetables array;
            end if
        end foreach
        foreach evaluator in evaluator1 do
            foreach evaluator in evaluator2 do
                if evaluator1 is not equal to evaluator2 then
                    Assign current timetable to neighboring_timetable;
                    Assign current evaluator1 to evaluator1 of the student;
                    Assign current evaluator2 to evaluator2 of the student;
                    Push the generated neighbouring_timetable into neighbor-
                    ing_timetables array;
                end if
            end foreach
```

```
41            end foreach
42            foreach venue in venues do
43                if venue of the student is not equal to the current venue then
44                    Assign current timetable to neighboring_timetable;
45                    Assign current venue to venue of the student;
46                    Push the generated neighbouring_timetable into neighbor-
                      ing_timetables array;
47                end if
48            end foreach
49        end foreach
50        Initialize an empty array, neighboring_fitnesses;
51        foreach neighboring_timetable in neighbouring_timetables do
52            Push fitness_function(neighbouring_timetable) into neighbouring_fit-
              nesses array;
53        end foreach
54        Assign the maximum value of fitness in the neighbouring_fitnesses array
          to a variable best_neighbouring_fitness;
55        if best_neighbouting_fitness is higher than current fitness then
56            Assign the timetable with maximum fitness to best_timetable;
57        end if
58        if fitness == 1 then
59            break;
60        end if
61        Assign the best_timetable to current timetable;
62        Iterations = iterations + 1;
63        if iterations == max_iterations do
64            Assign 0 to iterations;
65        end if
66    end while
67    end
```

With some adjustments made to the original algorithm, the Hill Climbing algorithm consistently produced conflict-free timetables. However, it's important to note that the algorithm's complexity increased significantly due to the presence of nested foreach loops, and these loops repeated multiple times. This complexity leads to exponential increases in execution time as the dataset size grows.

Partical Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization technique inspired by the swarming behavior observed in nature, such as fish schooling, as described in reference [17]. This approach has proven to be effective in addressing real-world optimization problems spanning various fields, including robotics, wireless networks, operating systems, classification, and scheduling issues, as mentioned in reference [18]. In PSO, an initial solution is randomly assigned, referred to as the "position". This solution evolves and improves through velocity adjustments, which update the particle's position and generate a new solution, as outlined in reference [19]. Subsequently, an evaluation function is employed to assess this solution for conflicts or crashes. This iterative process continues until the algorithm arrives at the best solution for the given problem. The framework for updating a particle's position can be represented as follows.

$$x_{k+1}^{i} = x_{k}^{i} + v_{k+1}^{i} \tag{1}$$

where the x_{k+1}^{i} is the position of particle i at iteration $k + 1$ after the changes of the velocity vector v_{k+1}^{i}. Updated particles will be the next solution and the process continues until the best solution is generated. The fundamental concept pseudocode for PSO is provided as follows.

ALGORITHM 3: PARTICLE SWARM OPTIMIZATION

```
    Input: List of students, venues, timeslots, evaluators1, and evaluators2
    Output: Assigned schedule for each student
    Initialization of variables: Assign 10 to num_particles, 200 to max_itera-
    tions, 1.5 to c1, 2.0 to c2, 0.7 to w

1   function evaluate (parameters: position)
2       Get global variables timeslots, venues, evaluators1, evaluators2;
```

```
    Initialize an empty array, schedule;
    for i = 0; i < max count of position; i +=4 do
        timeslot = timeslots[position[i]];
        room = rooms[position[i+1]];
        evaluator1 = evaluators1[position[i+1]];
        evalautor2 = evaluators2[position[i+1]];
        Assign timeslot, room, evaluator1, evaluator2 into an array and assign
        to schedule.
    end for
    Initialize zero to cost.
    for i = 0 to max count of schedule do
        Initialize conflicts as 0.
        for j = i + 1 to max count of schedule do
            if evaluator1 and evaluator2 of $student_i$ is the same or evaluator1
            and evaluator2 of $student_j$ is the same do
                conflicts ++
            else if (evaluator1 of $student_i$ and evaluator 2 of $student_j$ is the same)
            or (evaluator2 of $student_i$ and evaluator1 of $student_j$ is the same) or
            (evaluator1 of $student_i$ and evaluator1 of $student_j$ is the same) or
            (evaluator2 of $student_i$ and evaluator2 of $student_j$ is the same) or
            (room of $student_i$ and $student_j$ is the same) do
                if timeslot of $student_i$ and $student_j$ is the same do
                    conflicts ++;
                end if
            end if

        cost += conflicts;
    end for
    return 1 / (cost + 1);
end function

function generate_particles()
    Initialize an empty array, particles;
    Get global variables timeslots, venues, evaluators1, evaluators2,
    num_particles.
    for i = 0 to num_particles do
        initialize an empty array, position;
        for j = 0 to length of students array do
            Assign random element of timeslots to position array;
            Assign random element of rooms to position array;
            Assign random element of evaluators1 to position array;
            Assign random element of evaluators2 to position array;
        end for
        Initiate an array, particles and assign an array of position = position, ve-
        locity = each number from 0 to 80, best_position = position, best_fitness
        = evaluate(position), fitness = evaluate(position) to it.
        Assign position array to particles[position] and particle[best_position]
        Assign an array of size 40 with all values of 0 to particles[velocity]
        Assign evaluate(position) to particles[fitness] and particles[best_fit-
        ness]
    end for
    return particles;
end function

Assign generate_particles() to a variable, particles.
Assign position of first array of particles to best_position.
Assign fitness of first array of particles to best_fitness.
Assign zero to iterations.

while best_fitness less than 1 do
```

```
    if best_fitness < 0 and iterations == 100 do
        Assign iterations to 0.
        Call generate_particles function and assign the result to particles.
        Assign position of first array of particles to best_position.
        Assign fitness of first array of particles to best_fitness;
    end if
    for i = 0 to num_particles do
        for j = 0 to (4 * count of students) do
            Assign 2 random number to r1 and r2.
            particles[i][velocity][j] = (w * particles[i][velocity][j]) + (c1 * r1 * (particles[i][best_position][j] – particles[i][position][j])+ (c2 * r2 * (best_position[j] – particles[i][position][j])
        end for
        for j = 0 to (4 * count of students) do
            particles[i][position][j] += particles[i][velocity][j];
            if j%4 == 0 and particles[i][position][j] >= max count of timeslots do
                particles[i][position][j] = max count of timeslots – 1;
            else if j%4 == 1 and particles[i][position[j] >= max count of venues do
                particles[i][position][j] = max count of venues – 1;
            else if j%4 == 2 and particles[i][position][j] >= max count of evaluators1 do
                particles[i][position][j] = max count of evaluators1 – 1;
            else if j%4 == 3 and particles[i][position][j] >= max count of evaluators2 do
                particles[i][position][j] = max count of evaluators2 – 1;
            else if particles[i][position][j] < 0 do
                Assign zero to particles[i][position][j].
            end if
        end for
        Assign evaluate(particles[i][position]) to particles[i][fitness].
        if particles[i][fitness] > particles[i][best_fitness] then
            Assign particles[i][position] to particles[i][best_position].
            Assign particles[i][fitness] to particles[i][best_fitness].
        end if
        if particles[i][fitness] > best_fitness then
            best_position = particles[i][position]
            best_fitness = particles[i][fitness]
        end if
        if best_fitness == 1 then
            break;
        end if
        iterations += 1;
    end for
end while
end
```

The PSO algorithm consistently delivers optimal results, ensuring the avoidance of crashes in any parameter. In terms of algorithm complexity, it shares similarities with the genetic algorithm and hill climbing algorithm, as they all involve nested for loops. However, PSO features fewer nested loops, leading to faster execution times. While PSO is more intricate than the genetic algorithm, making it somewhat challenging to grasp and apply to real-world problems, a successful implementation can lead to substantial improvements in scheduling efficiency.

EXPERIMENTAL RESULTS

The three algorithms discussed above have been compared and summarized in a Table 1, considering various factors such as their ability to guarantee the best results, computational power requirements, ease of understanding the algorithm, average execution time in seconds, and the difficulty of implementation. All three algorithms were capable of generating timetables without encountering issues with evaluators, time slots, and venues. However, their performance varied, with execution times ranging from 0.17 seconds to 0.85 seconds, as determined from the average execution time over 20 runs of these algorithms. Additionally, factors like ease of understanding and the complexity of implementation were taken into account, as these aspects can influence the timeline during the development phase.

Table 1. Comparison of the scheduling algorithms.

Properties/Algorithms	Genetic Algorithm	Hill Climbing Algorithm	PSO
Best Result Guarantee	Yes	Yes	Yes
Computational Power	High	High	Medium
Average Execution	0.8530	0.3632	0.1725
Understandibility	Easy	Medium	High
Implementation	Easy	Medium	High

In Figure 1, the Particle Swarm Optimization algorithm exhibits the shortest average execution time among the three scheduling algorithms. However, it is not as consistent as the Genetic Algorithm, with variations in execution times ranging from approximately 0.6 seconds to below 0.3 seconds. The Genetic Algorithm, on the other hand, takes more time to schedule for 10 students but demonstrates greater stability compared to the other two algorithms, with execution times ranging between 0.8 to 1 second. As for the Hill Climbing Algorithm, its execution time is generally lower than that of the Genetic Algorithm but higher than Particle Swarm Optimization. However, its performance is also less stable, with the widest execution time range, spanning from 0.1 second to 1.3 seconds.

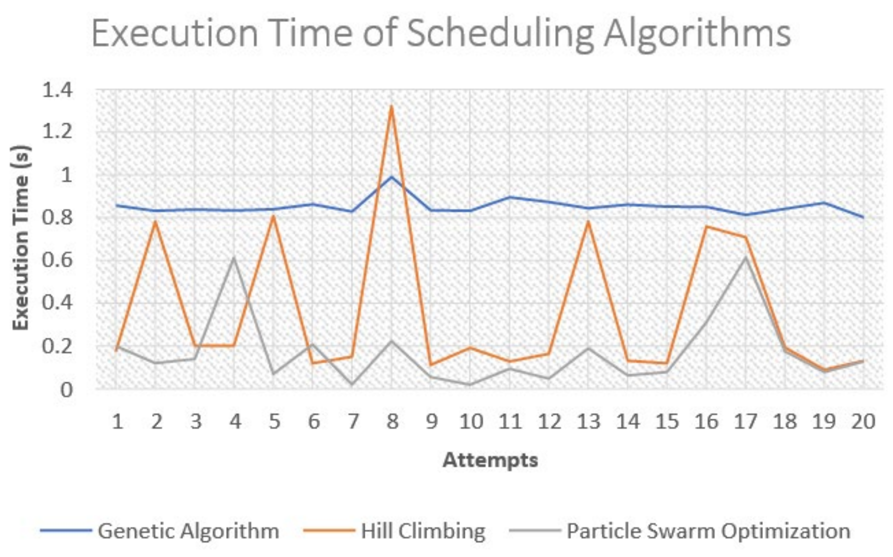

Figure 1. Execution time of the scheduling algorithms in 20 attempts.

CONCLUSION

Scheduling algorithms have had a significant impact across numerous fields, enhancing work efficiency and precision while reducing both time and labor costs. A well-designed scheduling algorithm can excel in generating schedules, even when dealing with a large number of parameters and extensive datasets, all in a remarkably short time. Among the scheduling algorithms examined in this study, the Particle Swarm Optimization algorithm emerges as the top performer, exhibiting the shortest execution time and greater stability compared to the Hill Climbing algorithm. Despite its potential complexity in understanding and implementation compared to the Genetic Algorithm and Hill Climbing Algorithm, investing effort into exploring Particle Swarm Optimization is highly worthwhile. Successfully integrating Particle Swarm Optimization into a scheduling system has the potential to enhance its overall performance significantly.

While the Genetic Algorithm and Hill Climbing Algorithm did not perform optimally in this paper, with the Genetic Algorithm having a longer average execution time and the Hill Climbing Algorithm exhibiting instability, it is essential to consider potential factors such as algorithm design. Complex algorithm structures can indeed impact execution times, or these algorithms may not be well-suited for this particular scheduling problem. Further research aimed at refining and improving these algorithms should be pursued to explore their potential applicability in addressing scheduling challenges.

REFERENCES

[1] M. W. Rodrigues, S. Isotani, and L. E. Zarate, “Educational Data Mining: A review of evaluation process in the e-learning,” Telematics and Informatics, vol. 35, no. 6, pp. 1701–1717.

[2] X. Zhao, N. Liu, S. Zhao, J. Wu, K. Zhang, and R. Zhang, “Research on the Work-rest Scheduling in the Manual Order Picking Systems to Consider Hu- man Factors,” J Syst Sci Syst Eng, vol. 28, no. 3, pp. 344–355, Jun. 2019, doi: 10.1007/s11518-019-5407-y.

[3] M. J. Pahlevanzadeh, F. Jolai, F. Goodarzian, and P. Ghasemi, “A new two- stage nurse scheduling approach based on

occupational justice considering as- surance attendance in works shifts by using Z-number method: A real case study," RAIRO - Operations Research, vol. 55, no. 6, pp. 3317–3338, Nov. 2021, doi: 10.1051/ro/2021157.

[4] M. V. Rane, V. M. Apte, V. N. Nerkar, M. R. Edinburgh, and K. Y. Rajput, "Automated timetabling system for university course," in 2021 International Conference on Emerging Smart Computing and Informatics, ESCI 2021, Insti- tute of Electrical and Electronics Engineers Inc., Mar. 2021, pp. 328–334. doi: 10.1109/ESCI50559.2021.9396906.

[5] M. W. Rodrigues, S. Isotani, and L. E. Zarate, "Educational Data Mining: A review of evaluation process in the e-learning," Telematics and Informatics, vol. 35, no. 6, pp. 1701–1717

[6] H. Alghamdi, T. Alsubait, H. Alhakami, and A. Baz, "A Review of Optimiza- tion Algorithms for University Timetable Scheduling," 2020. [Online]. Avail- able: www.etasr.com

[7] R. Ganguli and S. Roy, "A Study on Course Timetable Scheduling using Graph Coloring Approach," 2017. [Online]. Available: http://www.ripublication.com

[8] P. G. Daniel, Dr. A. O. Maruf, and Dr. B. Modi, "Paperless Master Timetable Scheduling System," Int J Appl Sci Technol, vol. 8, no. 2, 2018, doi: 10.30845/ijast.v8n2a7.

[9] C. Kalu, S. Ozuomba, and S. Isreal, "Development of Mechanism for Handling Conflicts and Constraints in University Timetable Management System," Com- munications on Applied Electronics, vol. 7, no. 24, pp. 22–32, Dec. 2018, doi: 10.5120/cae2018652804.

[10] [O. Abayomi-Alli, A. Abayomi-Alli, S. Misra, R. Damasevicius, and R. Maskeliunas, "Automatic Examination Timetable Scheduling Using Particle Swarm Optimization and Local Search Algorithm," in Data, Engineering and Applications: Volume 1, R. K. Shukla, J. Agrawal, S. Sharma, and G. Singh Tomer, Eds., Singapore: Springer Singapore, 2019, pp. 119–130. doi: 10.1007/978-981-13-6347-4_11.

[11] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," Multimed Tools Appl, vol. 80, no. 5, pp. 8091–8126, Feb. 2021, doi: 10.1007/s11042-020-10139-6.

[12] A. Amindoust, M. Asadpour, and S. Shirmohammadi, "A hybrid genetic algo- rithm for nurse scheduling problem considering the fatigue factor," J Healthc Eng, vol. 2021, 2021, doi: 10.1155/2021/5563651.

[13] S. H. Jacobson, "Analyzing the Performance of Generalized Hill Climbing Al- gorithms *," 2004. [Online]. Available: http://www.insead.edu/facul- tyresearch/tm/yucesan

[14] A. Baresel, H. Sthamer, and M. Schmidt, "Fitness function design to improve evolutionary structural testing."

[15] N. Dordaie and N. J. Navimipour, "A hybrid particle swarm optimization and hill climbing algorithm for task scheduling in the cloud environments," ICT Express, vol. 4, no. 4, pp. 199–202, Dec. 2018, doi: 10.1016/j.icte.2017.08.001.

[16] A. Lim, B. Rodrigues, and X. Zhang, "A simulated annealing and hill-climbing algorithm for the traveling tournament problem," Eur J Oper Res, vol. 174, no. 3, pp. 1459–1478, Nov. 2006, doi: 10.1016/j.ejor.2005.02.065.

[17] J. Kennedy and R. Eberhart, "Particle swarm optimization," in Proceedings of ICNN'95 - International Conference on Neural Networks, IEEE, 1995, pp. 1942–1948. doi: 10.1109/ICNN.1995.488968.

[18] A. G. Gad, "Particle Swarm Optimization Algorithm and Its Applications: A Systematic Review," Archives of Computational Methods in Engineering, vol. 29, no. 5, pp. 2531–2561, Aug. 2022, doi: 10.1007/s11831-021-09694-4.

[19] G. Venter and J. Sobieszczanski-Sobieski, "Particle Swarm Optimization," AIAA Journal, vol. 41, no. 8, Aug. 2003, Accessed: Apr. 25, 2023. [Online]. Available: https://arc.aiaa.org/doi/pdf/10.2514/2.2111?download=true