BABEŞ-BOLYAI UNIVERSITY FACULTY OF
MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE ENGLISH

# DIPLOMA THESIS

## Job shop scheduling algorithm optimization for generating efficient timetables

**Scientific supervisor**
**Lect. dr. Mircea Ioan Gabriel**

**Student**
**Alexis Razi**

2020

UNIVERSITATEA BABEŞ-BOLYAI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ENGLEZĂ

# LUCRARE DE LICENŢĂ

## Optimizarea unui algoritm de rezolvare a problemelor de tip "job shop" pentru generarea orarelor in mod eficient

**Coordonator ştiinţific**
**Lect. dr. Mircea Ioan Gabriel**

**Student**
**Alexis Razi**

2020

# 1 Abstract

Even with technology becoming better and better every year and older technology becoming obsolete so fast, computers still struggle with solving NP problems, such as the traveling salesman problem (TSP), which is a special case of the job shop scheduling problem (JSSP). The problem states that there are $n$ jobs, each having various job operations that have to be executed on $m$ machines. A proposed solution should be able to generate the sequence of job operations for each machine given a set of constraints.

This paper tackles the flexible JSSP (FJSSP) with partial flexibility in the context of generating efficient timetables for students. The proposed approach's aim is to generate the sequence of operations each machine can run while minimizing the makespan (denoted as $C_{max}$), which is the total time it takes for all the job operations to be completed by the machines. Particularly, in the topic of timetables, job operations represent the classes, the jobs are the subjects, and the machines represent the days in a working week, thus making the machine count a constant 5. Several heuristics such as class priority (lecture should be held before the seminary and the seminary before the laboratory) and building heuristic (trying to put successive classes in the same building) are applied.

In order to solve the flexible JSSP, the proposed approach creates maps all the job operations of a specific job to a machine which can process them and it also stores the time it takes for the operation to be processed on that specific machine. To solve the general flexible JSSP problem, another heuristic that does not apply when generating timetables was designed, based on randomness.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Razi Alexis

# Contents

# List of Figures

# 2 Introduction

In the first chapter of the thesis, different approaches to solving the job shop scheduling problem (JSSP) are described. Some of them aim to solve the classic JSSP problem, while others tackle the flexible JSSP with constraints that are closer to a production environment (e.g. some machines can break down) using various techniques such as artificial intelligence, greedy and backtracking based methods, each with promising results.

The second section of this paper describes the greedy based approach and heuristics that I've come up with in order to generate better timetables for students' classes at a university. The classes will be arranged such that a student can progress through a course in an orderly manner: the lecture comes before the seminary and the seminary comes before the laboratory for as many courses as possible. Also, classes are grouped to be held in the same building as much as possible. If this heuristic fails, at the end of the algorithm, if there is still time in the schedule, gaps will be inserted between two successive classes if they are held in another building, so that students will have time to get from one building to the other.

The next chapter contains the programming language used in the practical part of the thesis (Java), the IDE, the used dependencies and the reason behind choosing them in order to create the web application that acts as a wrapper to the proposed algorithm in the second chapter, named UBB Timetable (UBBTT). Also, use case diagrams, sequence diagrams, class diagrams and also a database diagram are presented in this chapter, giving a much clearer view of the implementation details of UBBTT. The ending subsections of the chapter present a user experience and user manual, containing screenshots of the application and there are also mentioned difficulties that were met during the implementation phase of the application and how I was able to overcome them.

The last chapter of the application compares the results of the proposed approach on a given data set against other results from the presented approaches from literature. Then, the comparison results are interpreted and conclusions are drawn. In the end, future improvements that could result in an improved algorithm are discussed.

# 3 Problem and problem context

## 3.1 The problem

More than 80 years ago the first programmable computer, named Z1, which is illustrated in 1 and was designed and created by the German Konrad Zuse in his parents' living room. It weighed a ton and it had about 20000 parts. [8] The computer supported only 8 operations: 2 for input/output, 2 for reading from memory into a register and writing to memory from a register, and 4 arithmetic operations (addition, subtraction, multiplication and division). It only had a clock running at 1 Hz (one cycle/second). The Z1 was not reliable, though, because of poor synchronization.

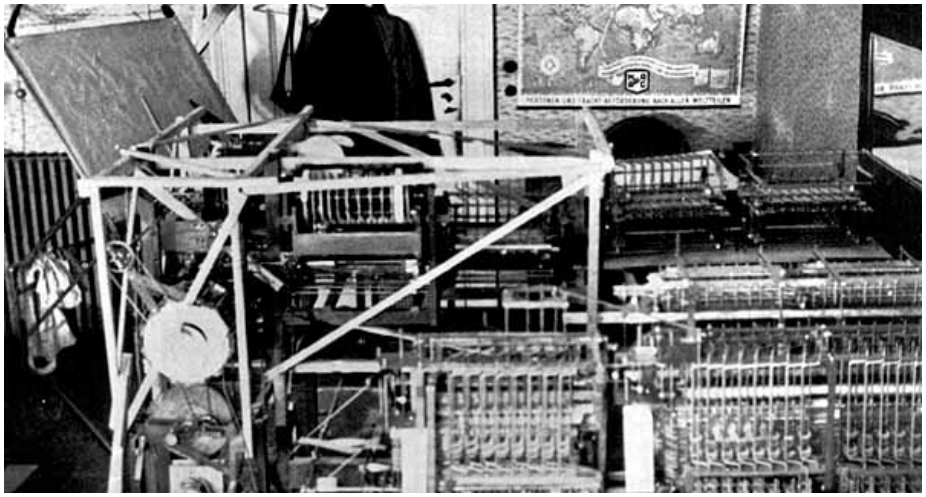

Figure 1: The Z1 computer created by Konrad Zuse. Taken from [8]

The Z2 was the first reliable computer built by Zuse, it had a clock running at 3 hertz (Hz), and featured the same operations and the mechanical memory as the Z1, but used relays for arithmetic functions. The Z3, though, is considered Zuse's masterpiece because it was made using just electrical relays, had a clock frequency of 5.33 Hz and consisted of 9 operations: the 8 Z1 and Z2 had, and calculating the square root of a number. Even so, the number of computations per second was very slow: Division and square root needed about 4 seconds, multiplication - 3 seconds, addition and subtraction - no more than a second. [6]

Moore's law, named after Gordon Earle Moore, the co-founder of Intel Corporation, states that every 18 months, the processing capacities of computers double. Now, more than 50 years later, computers are smaller, faster and cheaper than ever. Moore's statement still holds true, companies even basing investment decisions on it. [7] Nowadays, we don't talk about hertz, but rather gigahertz (GHz), as computing speed has increased exponentially in the last

decades. A central processing unit (CPU) that has a clock of 3.5 GHz means that it is able to compute 3.5 billion cycles per second. The world's fastest computer (a supercomputer) is called Summit and it analyzed data at a peak 1.88 exaops (1.88 billion billion operations per second). [13]

But even with all of that computing power that humanity benefits from in today's world, computers still struggle to solve particular problems, such as the halting problem or NP-hard problems. The halting problem should receive as input a computer program and its input and decide if the program will ever stop running or not. In 1936, Alan Turing, a mathematician who (sau whom?) is most famous for the Turing test, proved that a general algorithm for solving the halting problem for any given input cannot exist.

NP (nondeterministic polynomial time) is a class that contains decision problems that can be solved in polynomial time. An example of a decision problem is to check whether a string $S$ is part of an array $A$. NP-complete problems are the hardest problems in NP and each problem must be part of NP. [2] NP-hard is a category of problems in which every single one should be at least as hard as the hardest problems in NP (i.e. they should be at least as hard as the NP-complete problems), and they do not necessarily have to be in NP. [9] An example of an NP-hard problem is the well known traveling salesman problem (TSP), which has as input a list of cities and the distances between them, and as output it should determine the shortest route that the traveling salesman should take such that he would visit every city and return to the same starting city.

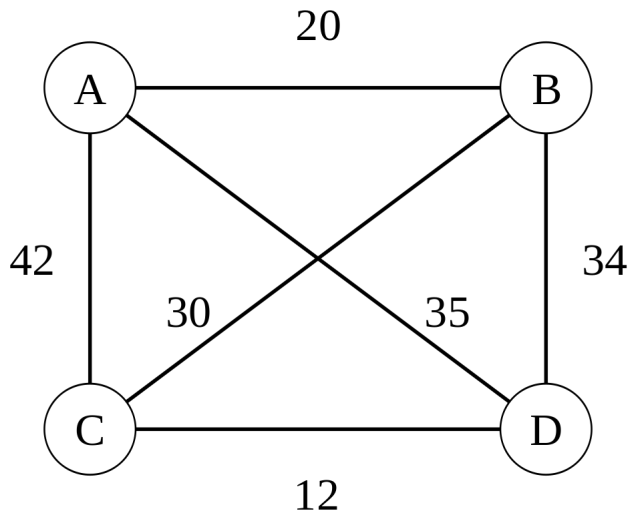


Figure 2: Symmetric traveling salesman person problem with four cities. Taken from [18]

## 3.2 State of the art

### 3.2.1 Solving the job shop scheduling problem using artificial intelligence [12]

A genetic algorithm (GA) belongs to the class of evolutionary algorithms (EA) and is a search heuristic that is based on the theory of natural selection posed by Charles Darwin. This algorithm replicates natural selection such that only a population's fittest individuals are chosen for reproduction and produce offspring.

In their paper, Mahanim Omar, Adam Baharum, Yahya Abu Hasan describe their own approach to solving the job shop scheduling problem (JSSP) through a genetic algorithm. Each job that needs to be processed on the given machines have to be executed in a particular order (i.e. a technological constraint is imposed) and no interruptions are allowed. The goal of the proposed algorithm is to reduce the makespan, which represents the total time required for all the jobs to be finished. For this research, we will assume that the number of jobs and their duration (i.e. processing time) are known and will not change and no machines break down.

A critical path is set to be the longest paths from the first executed job until the last one. The jobs included in this path are named critical operations. The set of critical operations on the same machine are called a critical block. The distance between two schedules S1 and S2 is defined as the number of differences in the order that jobs are executed on each machine of said schedules. This distance is also called the disjunctive graph (DG) distance, which will be denoted as DGDistance(S1, S2).

Neighbourhood searching is a technique used to solve a set of combinatorial problems. A schedule S marks a point in the domain through which the algorithm searches, also called a search space. The other schedules that we can arrive at from the S with one transition are denoted by N(S). The algorithm is executed based on specific criteria in order to pick a new point from the neighbouring schedules. In this approach, the neighbourhood is defined if the disjunctive graph distance between a schedule S and a point $S'$ from $N(S)$ is 1.

We define the population to be a list of individuals (i.e. a schedule), each of which have three characteristics: a phenotype, which in the context of the JSSP represents the job order on a specific machine, a genotype, which is the machine schedule and a fitness, which is represented by the makespan.

The representation used in this paper is called the permutation representation (Yamada and Nakano, 1997)[21]. JSSP can be viewed as an ordering problem just like the Travelling Salesman Problem (TSP). A job sequence matrix contains a schedule's job permutations on each machine. In figure 3, the rows indicate the machines and the columns indicate the order of the job operations.

The initial population that will be used as input to the algorithm is going to contain schedules that will be randomly generated, but also schedules generated by applying various priority sequencing rules, for example the shortest

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 1 & 3 \end{bmatrix}$$

Figure 3: A job sequence matrix for a 3x3 job shop scheduling problem

processing time (SPT) rule and the longest processing time (LPT) rule. SPT states that the next job that gets to be processed is the job that requires takes the shortest time to finish, while in LPT, the next job that gets to be processed is the one that takes the longest time to finish.

During the parent selection process, two individuals (two schedules) have to be chosen in order for them to reproduce. In this approach, the parents will be randomly selected, so as to give every individual the same chance at reproduction.

For the crossover function, this paper follows an algorithm developed by Yamada and Nakamo in their 1995 paper [20] for solving the JSSP. Their approach utilizes the DG distance and CB neighbourhood.

Parent 1 and parent 2 from the parent selection process will be denoted as $p_1$ and $p_2$. Let $x = p_1$ and $N(x)$ be the CB neighbourhood for $x$. Then, for each element $S_i \in N(x)$, the distance between Si and p2 is computed in order to generate $D(S_i, p_2)$. The next step is to sort $D(S_i, p_2)$. Finally, the chosen child (denoted by $q$) is the one with the minimal DG distance between q and its parents. Then, we iterate through $D_{sorted}(S_i, p_2)$ and we accept $S_i$ with 100% probability if the schedule's fitness is less or equal than $x$'s fitness: $V(S_i) <= V(x)$. If the schedule's fitness is not less or equal than $x$'s fitness, the probability to accept $S_i$ decreases to 50%. Then, $x$ is modified from $p_1$ to gradually resemble $p_2$. Each iteration, $x$ will start to lose $p_1$'s attributes and inherit more and more of $p_2$'s attributes, but not in the same ratio.

The most essential step of the genetic algorithm is selecting the individual that gets to be replaced by the chosen child. For this approach, the worst individual is chosen to be replaced with the child. If there are any other members with the same fitness (i.e. the makespan of the schedule), those members are also replaced with the child.

In the end, this is the final form of the algorithm:

1. Initial population consists of 10 random schedules, some of them being generated by applying priority rules

2. Two schedules $p_1$ and $p_2$ are randomly chosen from the initial population and the DG distance between them is computed

3. If the DG distance between $p_1$ and $p_2$ is less than a predefined value, algorithm 2 is executed on $p_1$, a child is created. Then, skip to the fifth step of the algorithm.

4. If the DG distance between $p_1$ and $p_2$ is larger than the predefined value, algorithm 1 is applied to both $p_1$ and $p_2$ and a child is generated.

5. The neighborhood search algorithm is run on the child in order to find the best child in the neighborhood (i.e. to find the one with the highest fitness value).

6. If the best child's makespan is smaller than the worst individual of the population (i.e. the one with the smallest fitness values) and not equal to any other individual's fitness, then the best child will replace the worst individual in that population. If, however, there is an individual with the same makespan as the best child, then that individual will be replaced with the best child.

7. Steps 2 through 6 are repeated until a certain stopping condition is met.

The results of the test runs were promising most of the time, but there is one trade-off: the proposed genetic algorithm takes more time to generate a solution. When the initial population was entirely randomized, the optimum solution schedule was only found at generations larger than 100. But, if the initial population is not entirely randomized, only part of it, and the other part contains schedules generated by priority rules, then the optimum solution schedule is found before generation 100.

Because artificial intelligence is a very wide field, there are, of course, many other research papers and approaches available. For example, Lei Wang, Jingcao Cai, Ming Li, and Zhihu Liu proposed an ant colony optimization (ACO) algorithm [17] for the flexible job shop scheduling problem (FJSP). ACO-based algorithms are influenced by the way ants, in real life, search for food. Another important aspect is that one of the goals of ant colonies is their survival, not just one individual's survival. The authors of the papers mention that ACO can get stuck in a local optimum search, so they proposed some improvements, basically modifying the probabilities of a job operation choosing a machine. Just like real ants want to take the shortest path to obtain food for their colony's survival, the job operations must identify the best machines that can execute them. The proposed improved ant colony optimization (IACO) was tested against other algorithms from literature. In a test run, the smallest makespawn was obtained by IACO for all test cases, and for the second test case, the algorithm found the optimal makespan in 9 out of 10 cases. The proposed algorithm is promising, but more research should be done as to improve it to solve variations of the job shop scheduling problem containing fuzzy (variable) due dates or dynamic scheduling.

#### 3.2.2 Solving the job shop scheduling problem through an iterative greedy insertion technique [1]

This paper presents an approach to solving the flexible job shop scheduling problem (JSSP), which is an extension of the JSSP. The term flexibility refers to the fact that a particular job operation can be assigned to a subset of machines, which means that only partial flexibility exists, or that an operation can be assigned to any available machine, which means total flexibility.

The FJSSP is comprised of two parts: the sequencing part, which serves to generate the job sequencing on all machines to obtain a solution schedule. The solution's aim is to minimize the makespan (denoted as $C_{max}$). The second part, called the routing part, is actually running the job on its alloted machine.

In the paper, it is mentioned that FJSSP, being an NP-hard problem, the speed with which the solution is generated overshadows the quality of the generated solution. This approach aims to reduce the time it takes for a schedule to generate and will produce acceptable results in a shorter amount of time rather than exact results which could take longer.

There are two approaches regarding the aforementioned parts. The first one is the sequential approach, which states that the sequencing and routing parts are viewed as two different problems and are handled individually. In the integrated approach, however, the sequencing and routing parts can be handled together. In this paper, the chosen approach is going to be the integrated one and a heuristic for the sequencing problem is proposed in order to minimize $C_{max}$.

A greedy algorithm is a technique that, as per its name, at each step makes the choice that yields the best result at that particular step (i.e. it chooses the locally optimal option at each step). The greedy technique is widely used if you want the overall time complexity of the algorithm to be reduced, but it also provides other advantages, such as the fact that it's easy to implement and to expand.

Because it is costly to search for the optimal solution schedule in the middle of all possible candidates, the Iterative Greedy Insertion Technique (IGIT) aims to solve this issue. As the name suggets, the algorithm will greedily fill the machines with jobs in order to minimize $C_{max}$.

The approach proposed will also consider the following constraints:

- Allocation constraints: each job operation can only be executed by one machine, there can be no two machines such that they have to execute the same operation. Also, operations can only be assigned to a machine that is able to execute it.

- Precedence constraints: a job operation (denoted as JO) can only begin to execute if the preceding JOs that have to be finished before are finished. If two JOs that are part of the same job will be run on different machines, transportation time comes into play: it represents the time it takes for the relevant job operations to be transferred from the machine that previously executed the job operations to the machine that will run the next job operation.

- Makespan ($C_{max}$) constraints: $C_{max}$'s value is going to be set to the last job operation finished in a particular assembly cell.

The algorithm first iterates through all the job operations so that it can find all JOs that can be run at the beginning, at time 0. At each iteration, a free machine that can process the operation that can be run at time 0 is assigned

that particular JO. Then the job index that this JO was part of is inserted in all the free slots of the machines that can run it. At the end, the best schedule that is the local optimum is chosen as the best schedule so far, and the redundant schedules are removed. In the end, if the current solution schedule is better than the best solution schedule found so far, the best schedule is replaced by the current schedule.

The proposed approach's algorithm compares against two others:

1. The Mixed-Integer Linear Program (MILP)

   MILP was developed using Cplex, considers all given constraints and generated an offline solution.

2. The Potential Fields (PF)

   PF is a technique that helps to control moving machines and their performance in undetermined environments (e.g. movement of robots), and was even used for in production. This algorithm considers both limited storage constraints for machines and transportation times.

In order to test the paper's approach, static scenarios were made, in which disturbances are not considered.

For a particular set of data, 3 jobs (the first having 10 operations, and the other two 7), IGIT's performed better than PF and near MILP's results, with only a 6.13% gap.

In another test, containing 7 jobs with 60 total operations, the results obtained by IGIT and MILP were the same.

In conclusion, the results with the presented heuristic are encouraging. Future research and development of this approach should integrate it with an iterated local search technique.

#### 3.2.3 Backtracking search based hyper-heuristic for the flexible job-shop scheduling problem with fuzzy processing time [10]

The flexible job-shop scheduling problem with fuzzy processing time (FJSPF) is an extension of the flexible job-shop problem (FJSP) and it means that a job's deadline or the time it takes for it to finish is a fuzzy variable. Research for an algorithm to solve this problem is of great importance, because the FJSPF is resembles the reality in a better way.

While FJSPFs are oftentimes solved using meta-heuristics, not all types of optimization problems benefit from meta-heuristics and not even all variations of the same problems, as stated in the No Free Lunch theorem.[19]

Hyper-heuristics is a new optimization technique and it has gained more attention in recent times. While meta-heuristics operate on the solution domain, hyper-heuristics do it indirectly through a collection of low-level heuristics.[15] Generating an optimal heuristic using the aforementioned low-level heuristics is by using backtracking search optimization (BSA).

The paper's proposed solution of solving the FJSPF is through a backtracking search hyper-heuristic (BS-HH). The algorithm's aim is to minimaze the makespan. Note that because deadlines are a fuzzy variable in the FJSPF, the makespan is also a fuzzy variable.

The problem is as follows: there are $m$ machines $M = \{M_1, M_2, ..., M_m\}$ and $n$ jobs $J = \{J_1, J_2, ..., J_n\}$ with each job $J_i$ having $n_i$ operations $O = \{O_{i,1}, O_{i,2}, ...O_{i,n_i}$. The time it takes for a job $O_{p,q}$ to finish on machine $M_r$ is denoted as a triplet $t_{p,q,r} = (t^1_{p,q,r}, t^2_{p,q,r}, t^3_{p,q,r})$, the three values having the meaning of the lowest time it takes for the job to finish, the probable time and the highest time. This is called a triangular fuzzy number (TFN). Since we've also concluded that the makespan of $O_{p,q}$ is going to be a TFN, we can represent it as $C_{p,q,r} = (C^1_{p,q}, C^2_{p,q}, C^3_{p,q})$. The purpose of the algorithm is to obtain the operations alloted on each machine and the order in which they will be executed while trying to minimize the maximum makespan $C_{max} = max_{i=1,2,...,n} C_i$. The proposed approach will also have to meet the following precedence constraints:

- Interrupting a job operation (i.e. preemption) is forbidden if the operation has begun executing
- A job operation can not be executed by more than one machine at the same time
- A machine can only executed one job operation at a given time

To generate an acceptable solution schedule, we have to define the ranking, maximum and addition operations on a TFN. Let $T_1 = (x_1, x_2, x_3)$ and $T_2 = (y_1, y_2, y_3)$. The maximum value between the two TFNs is defined as:

$$\text{if } T_1 > T_2 \text{ then } T_1 \vee T_2 = T_1; \text{ else } T_1 \vee T_2 = T_2$$

The ranking function is defined as either $F_1(T) = (x_1 + x_2 + x_3)/4$, $F_2(T) = x_2$ or $F_3(T) = x_3 - x_1$ and operates on three rules:

- First, the larger number between $F_1(T1)$ and $F_1(T2)$ is chosen
- If the two numbers are equal, then $F_2$ is used for comparing the values
- If, again, the two numbers are equal, $F_3$ is chosen

The addition between $T_1$ and $T_2$ is defined as:
$T_1 + T_2 = (x_1 + y_1, x_2 + y_2, x_3 + y_3)$
BSA is an evolutionary algorithm (EA) with two populations, hence why it's described as dual-population-based EA. It consists of five steps, which are:

1. Initialization

   The starting population (denoted as P) is determined with the help of equation 1. U is denoted as the uniform distribution operator, S is the size of the population and D is the dimension of the problem.

$$P_{i,j} = U(low_j, up_j), i = 1, 2, ..., S; j = 1, 2, ..., D \tag{1}$$

2. Selection (1)

   In the first selection algorithm, in order to establish the search direction, a historical population is generated and denoted as HP. Equation 2 describes how it will be generated:

$$HP_{i,j} = U(low_j, up_j) \tag{2}$$

   As a new generation begins, HP will be updated in equation 5 and by the random permuting operation described in equation 6. The connection between HP and the preceding generation is guaranteed by equation 5.

$$if a < b then HP = P | a, b = U(0, 1) \tag{3}$$

$$HP = permuting(HP) \tag{4}$$

3. Mutation

   Let TP be a test population generated in this step using the following formula:

$$MP = P + F * (HP - P) \tag{5}$$

   $F$ is a parameter given as input by the user which is used to control the extent of the search direction matrix (which is the result of $HP - P$). The test population is going to be enhanced by the previous generations, because we are taking into account both the population and the historical population.

4. Crossover

   The crossover step is comprised of two parts. In the first part, equation 8 builds a binary matrix (denoted as $map$) which has the size $SxD$. [...] is denoted as the ceiling function. The ceiling function maps a given input integer to the smallest integer that is greater than or equal to the input integer. For example: $[3.1] = 4$. $r_1, r_2 \in [0, 1] \in$ are generated randomly. We also define a function called $rand(x)$ which returns a random integer between 1 and $x$, including $x$. In the end, the mix rate is denoted as $rate_{mix}$, such that $rate_{mix} \in [0, 1]$. In order to generate the final population (denoted as FP), TP is modified using these rules:

   - $if map_{i,j} = 1, then TP_{i,j} is set to P_{i,j}$
   - Individuals of $TP_{i,j}$ that are not in the allowed search space are going to be recomputed

$$\begin{cases} \text{map}_{i,permuting(1:[rate_{mix} * D * U(0,1)])} = 0 & r_1 > r_2 \\ \text{map}_{i,rand(x)} = 0 & \text{otherwise} \end{cases} \tag{6}$$

5. Selection (2)

   In the last step of the algorithm, the individuals in the test population (TP) which have better fitness values than their corresponding individuals in the current population (P) will replace them. If $S \in P$ is better (i.e. contains a higher fitness value) compared to the global best, then the global best will be set to $S$.

The low-low level heuristic previously mentioned are based on the insert, inverse and swap neighborhood structures. They are presented in the list below:

1. Random swap

   Choose in a random manner two distinct job operations (JO) from the sequence of the JO execution and swap the JOs if they are not part of the same job.

   This heuristic makes use of simulated annealing, which is a technique that permits the acceptance of solutions with higher makespans (i.e. inferior solutions), albeit with a smaller probability of this happening. The reason for this is to prevent being confined into a local optimum.

2. Adjacent swap

   Choose a JO from the execution sequence and swap the JO with the next one in the sequence, if it's not the last one. If the chosen one is the last one, it will be swapped with the first operation of the sequence.

3. Backwards insertion

   Two JOs (denoted as $O_1$ and $O_2$ are selected in a random fashion. Then, take the JO that is in the front and insert it before the JO that is in the back of the sequence.

4. Forward insertion

   This is the reverse operation of the backwards insertion heuristic. As in the backwards insertion, two JOs (denoted as $O_1$ and $O_2$ are chosen randomly. The difference is that the JO that is in the back of the sequence is inserted before the one that is in the front of it.

   Simulated annealing is also used for this algorithm, which is described above.

5. Inverse

   Two indices chosen randomly in the JO execution sequence are chosen (denoted as $i, j$ and the subsequence between $(i, ..., j)$ is inversed.

6. Simplified referenced local search (RLS)

   RLS is a technique that has been used in solving a scheduling problem, but also to build a hybrid algorithm. However, the authors of the paper mention that the operation is costly. As such, they developed a simplified

version of RLS so that it is easier to implement. The simplified RLS is going to be denoted as SRLS.

In essence, the SRLS algorithm removes a job operation (denoted as $o$) from the sequence with the best makespan so far. Also, it receives as input a given sequence, denoted as $s$ (i.e. a solution schedule) and proceeds to find the best solution sequence by inserting $o$ in each available slot in $s$. If the makespan ($C_{max}$) of the newly formed sequence (denoted as $s'$) is less than $C_{max}$ of $s$, then $s$ will be set to $s'$.

In this paper's proposed approach, BSA acts as a high-level technique in the backtracking search based hyper-heuristic, its aim being to discover the heuristics that produce, for the FJSPF, solutions that are as good as possible.

Both the initial and historical populations (P and HP) are generated in a random manner, making sure that every individual of the population contains the set of the low-level heuristics. P can be illustrated as a matrix:

$$P = \begin{bmatrix} \mathrm{h}_{1,1} & ... & \mathrm{h}_{1,j} & ... & \mathrm{h}_{1,L} & \pi_{1,k} & ... & \pi_{1,N} \\ . & & & & & & & \\ . & & & & & & & \\ . & & & & & & & \\ \mathrm{h}_{1,1} & ... & \mathrm{h}_{1,j} & ... & \mathrm{h}_{1,L} & \pi_{1,k} & ... & \pi_{1,N} \\ . & & & & & & & \\ . & & & & & & & \\ . & & & & & & & \\ \mathrm{h}_{1,1} & ... & \mathrm{h}_{1,j} & ... & \mathrm{h}_{1,L} & \pi_{1,k} & ... & \pi_{1,N} \end{bmatrix}$$

$i = 1, 2, ..., S; j = 1, 2, ..., L; k = 1, 2, ..., N$

In the population matrix, $h_{i,j}$ represents the heuristic found at index $j$ of individual $i$. $L$ indicates the total number of heuristics. $N$ stands for the number of job operations and $S$ represents the size of the population (i.e. the number of individuals in the population).

The heuristics that are going to be applied to the test population (TP) presented earlier (generated in the second selection step of the BSA algorithm) will be stored in an array. Its contents are determined by applying the following formula (note that *round* represents the rounding operation): $max\{1, min\{round\{TP_{\{}i, j\}\}, L\}\}$.

To the solution schedule $\pi_i = \{\pi_{i,1}, \pi_{i,2}, ..., \pi_{i,N}$ is going to be applied the heuristic described above: $h_{i,j}$. The heuristic is applied based on the order in which it appears in the individuals of the test population.

The results of the proposed BS-HH algorithm approach for the FJSPF is compared against other algorithms (Gao et al., 2015; Lin, 2015a). The result of the comparison was that the proposed solution yields substantial, greater results than the other algorithms. What is even better is that the proposed approach managed to find best solutions that previously were not achieved by the other algorithms. Furthermore, from some test runs, it can be seen that the algorithm does not overload the CPU as much as other algorithms.

## 3.3 Overview

In this chapter, multiple approaches to various variants of the job shop scheduling problem (JSSP) were presented. The presented genetic algorithm was used to solve the classic JSSP, while the approach using greedy insertion tackled an extension of it, the flexible JSSP. The third approach used a backtracking search based hyper-heuristics for the flexible JSSP, but with an interesting constraint: fuzzy (i.e. unknown) processing times for job operations. All the algorithms proved to return promising solution schedules, even if they were not optimal for all the test cases. However, the big problem with a proposed algorithm that aims to solve JSSP or one of its derived problems is that it is hard to come up with a good (i.e. that yields optimal solutions) for all the constraints or disturbances that can occur in a system. In the real world, in production environments, new constraints and disturbances appear constantly, and it is a hard task of efficiently regenerating a new solution schedule based on these factors that can appear randomly.

# 4 Proposed solution

After careful consideration, I have decided that to develop my own approach at solving the flexible job shop scheduling problem (FJSSP). In JSSP, there are $n$ jobs $J_1, J_2, J_3, ..., J_n$, each of them containing $O_1, O_2, O_3, ..., O_n$ operations. Given $m$ machines, the how should the jobs be scheduled on those machines in such a way that the jobs will finish executing as fast as possible (i.e. the makespan, denoted as $C_{max}$ is minimized)? In FJSSP, the term flexible refers to the fact that each job operation can be executed on only a set of predetermined machines. Full flexibility means that any job operation can be executed on any of the available machines.

Thus, the first step was to get a basic algorithm capable of solving the FJSSP using a greedy-based technique.

The first version that could actually solve the problem and return a possible solution iterated through all job operations (JO) and assigned them to the first machine that could the JO at that step of the algorithm. But even if the approach was fast in terms of determining a viable sequence, the makespan ($C_{max}$) returned on a few test cases proved the sequence to be very slow.

The conclusion from the first version of the algorithm was that there needs to be a balance between the time it takes for the algorithm to determine a candidate solution and the makespan of said solution. Thus, I came up with a solution that generates better schedules.

The jobs and the number of available machines are tightly coupled, because every job contains a map that holds the necessary information as to how long each of its operations takes to finish on each available machine. The key of the map is a tuple, consisting of ¡$machine_i$, $job_j$, $operation_k$¿ and the key is the time it takes for the job to run $operation_k$ of $job_j$ on $machine_i$.

After initializing the jobs, each of them is added into a set. Then, the algorithm will iterate through the set and instantiate a new map that adds each job's map into it, called the global map.

Then, the algorithm iterates through each job and their operations and finds the best machine suitable for completing that operation and assigns it to that machine. At each step, all the entries from the map that contain the machines that can solve that particular job operation are removed, since they are no longer useful.

The problem with this approach, however, was that it would not work well in a production environment, since it would first try to complete all operations from job $job_i$ before advancing to $job_{i+1}$ instead of progressing through multiple jobs at the same time.

Having that in mind, an array with the size of the number of jobs which contains the progression of all the job operations for each job, called $jobOperations$ was instantiated with all values 1.

The algorithm still behaves the same, but instead of progressing through job $job_i$ before $job_{i+1}$ and so on, like before, this time it would go through the first operation of $job_i$, then the first operation of $job_{i+1}$ and so on.

Each time a job ($job_i$) with an operation ($operation_j$) would be assigned to

a machine, $jobOperations_i$ would be increased by one. The algorithm will keep finding the best machine that can execute the current job operation until all operations are each assigned to a machine.

To avoid getting the same solution schedule, a heuristic that has a 20% chance of triggering was introduced. If triggered, then the machine that would be assigned a particular job operation would not be the one that could finish it the fastest, but the machine that has the lowest makespan. Even though the time of running that operation on a machine $M_1$ would be lower than executing it on machine $M_2$, $M_1$ could have the highest makespan of all the machines and $M_2$ could have a lower one, thus making it a better decision to assign the previously mentioned operation to $M_2$, rather than to $M_1$. During testing, this heuristic was found to be responsible for both new lowest and highest makespans.

Figure 4 represents the flowchart of the proposed algorithm that aims to solve the flexible job sop scheduling problem.

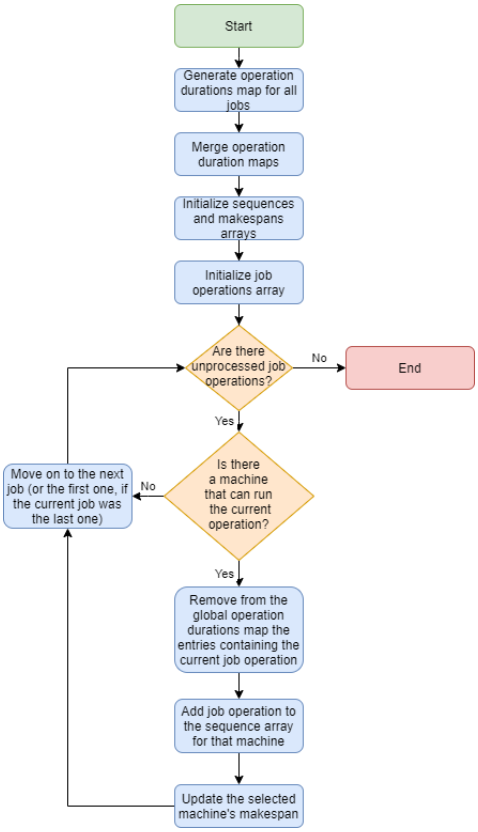


Figure 4: Proposed approach's flowchart. Created using `https://diagrams.net/`

A real world usage of this algorithm would be to generate timetables for

universities. In this context, the jobs could be seen as the subjects a student would have in a particular semester, the job operations would represent a combination of a lecture, a seminary and a laboratory. The number of machines is a constant, which is 5, because that's the number of days in a work week.

It was not hard to adapt the current algorithm to be able to generate a timetable. However, it was not very efficient because applying the same heuristics as in the general approach would not yield efficient solutions: for example, the random heuristic would only worsen a solution timetable.

Thus, new heuristics that are related to timetables are introduced:

- Building heuristic

  This heuristic tries to put as many classes in the same building on the same day

- Class priority heuristic

  This heuristic tries to put the lectures first, then the seminaries and only then the laboratories for a specific subject

Also, after the sequence is generated, the timetable's hour optimization is applied. This refers to the fact that if, for example, a class that lasts 2 hours starts at 8 (and will end at 10), if the next class in the same day will not start at 10 if the two classes are not in the same building. Instead, it will start at 12, if possible.

The flowchart of the proposed algorithm updated to generate timetables for students' classes is illustrated in figure 5.

## 4.1 Tests and specification

In order to test the proposed algorithm, the Spring framework has a Test module that can be used in combination with JUnit to create powerful and thorough tests.

I've tested the functionality of the algorithm in the `FJSSPTest` class, where I initialize the number of machines, the number of jobs along with their operations and how much each operation takes to execute on each machine. There is a function for each job that instantiates the maps that contain the operation durations. The key of the maps consist of ¡$machine_{id}, job_{id}, operation_{id}$¿ and the value is the duration of $job_{id}$

All of this happens in the function "generateJobs". After initialization, the class `FJSSPSolver` is instantiated with following parameters: job count, machine count and the set of jobs. After the instantiation of `FJSSPSolver`, certain functions used in the main function that generates the solution schedule are tested.

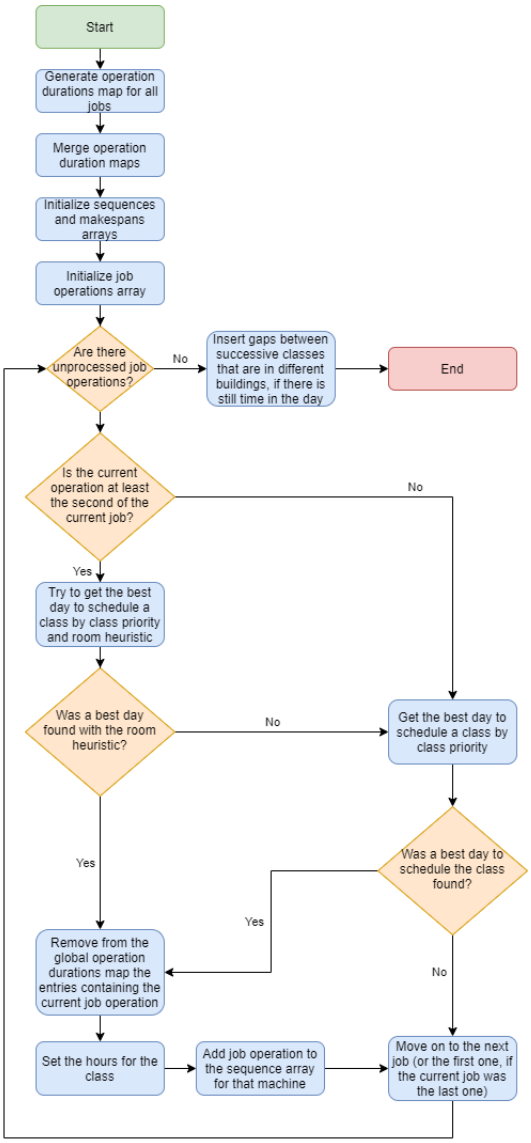


Figure 5: Timetable generator's flowchart. Created using `https://diagrams.net/`

| Preconditions and Postconditions | | |
|---|---|---|
| Function | Preconditions | Postconditions |
| jobOperationsStillNeedToBeProcessed | jobOperations ArrayList was initialized with base values of 1 | True was returned if the numbers found at index $i$ of jobOperations are less than or equal to the number of operations found at job which has the index $i$ |
| getMachineThatCompletedJobOperationBefore | sequences map was initialized, job and operation ID are valid | The ID of the machine that completed the previous operation for the job received as input is returned. If this is the job's first operation, 0 is returned |

# 5 Application: UBB Timetable

As a justification for the real-world utility that the proposed approach offers, the algorithm is integrated in a web application named UBB Timetable (UBBTT), which is directed towards both students and teachers. Students can view their timetable, and teachers can post notes regarding the fact that the scheduled class is canceled or some other information, for example the room that it is held in changes. In this application, the developed algorithm comes into play by helping create a timetable in such a manner that courses are first, then seminaries and last but not least, the laboratories.

reorganizing all the available classes as best as possible such that there are no gaps in-between classes.

## 5.1 Architectural design

The application is designed in such a way that users can have one of two roles: student and teacher. The teacher has to be logged in, but any anonymous user that does not log in is treated as a student.

The student's use case diagram is shown in figure 6. A student can view the timetable and all the notes on all their classes for the current week of college for the whole faculty, as the timetable is publicly available for everyone. Also, they can send their suggestions and feedback regarding the application anonymously. Also, they can view the personalized timetable.

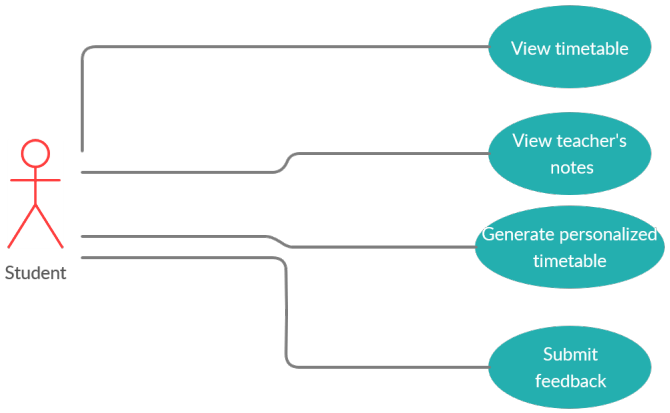


Figure 6: Student use case diagram. Created using `https://creately.com/`

In order for a student to generate a timetable, they can simply do so by making a request from the client. This process is pictured through a sequence diagram in 7.

The teacher's use case diagram is represented in figure 8. The teacher can do almost everything a student can anonymously do, like send feedback, view

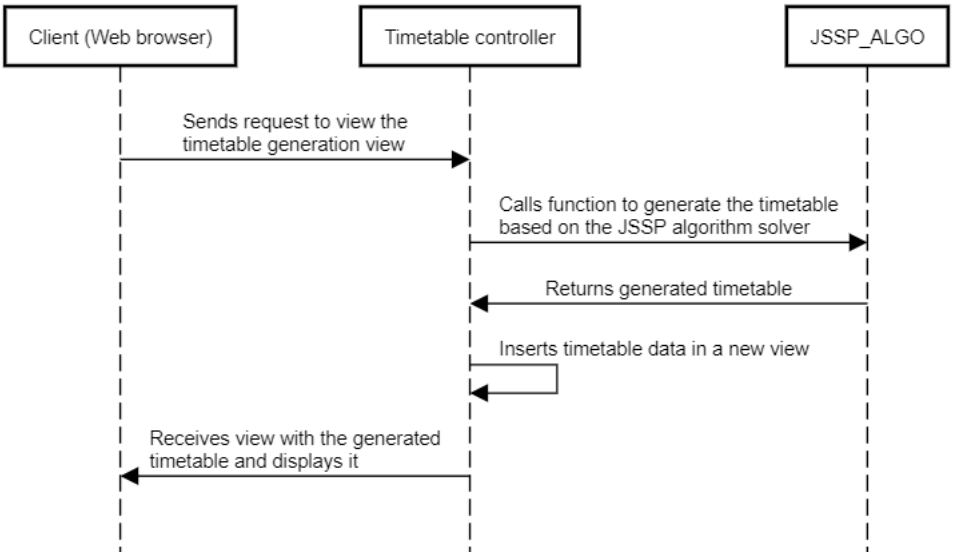


Figure 7: Generate timetable sequence diagram. Created using `sequencediagram.org`

the students' timetable, they can also generate a timetable, but that would not benefit them, as they are not a student. Of course, a teacher has to log in to the application beforehand in order to be treated as one. From there, they can create new notes regarding some of their classes, they can update existing ones or even delete them. If they are on a public computer, for example, and wish to not remain logged in, they can, of course, log out of the application.

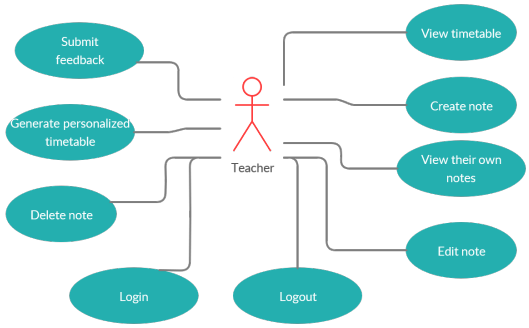


Figure 8: Teacher use case diagram. Created using `https://creately.com/`

In order for a user, be it logged in or not (i.e. a student or a teacher) to report feedback or a suggestion to the developer of the application, the process is similar to creating a note.

Figure 9 contains the sequence diagram that illustrates the process of a user submitting feedback to the developers of the application.

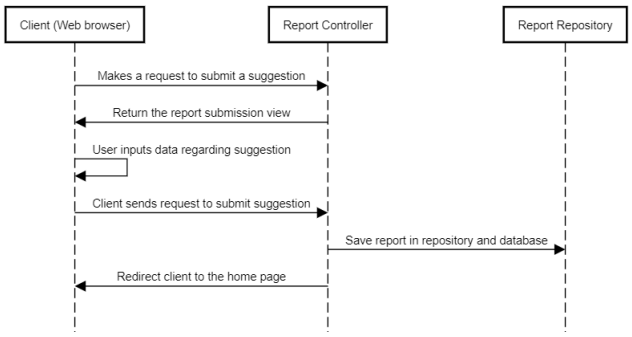


Figure 9: Sequence diagram for sending feedback to the developer. Created using `sequencediagram.org`

In short, the user fills out the information related to the suggestion they are going to make, such as the message, and the request is then sent to `ReportController`, which in turn calls the save function of `ReportRepository`, where the suggestion will be saved in the database.

In order to add a new note, the process is similar to reporting feedback, the only difference being the details that the teacher has to fill in. Indeed, there is much more information needed for a note, such as the reason, the week, the affected class and so on.

The sequence diagram for editing a note is intuitive (figure 10). The teacher needs to view a list of their notes, choose one to edit and save the updated note for others to see.

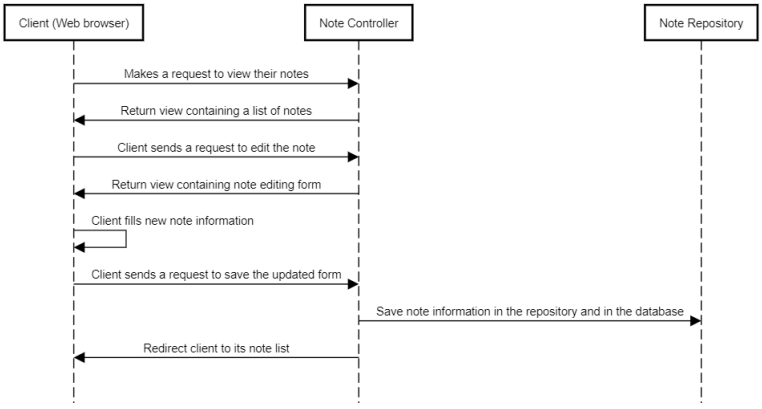


Figure 10: Sequence diagram for editing a note. Created using `sequencediagram.org`

In figure 11, the layers of which UBBTT is comprised of are illustrated.

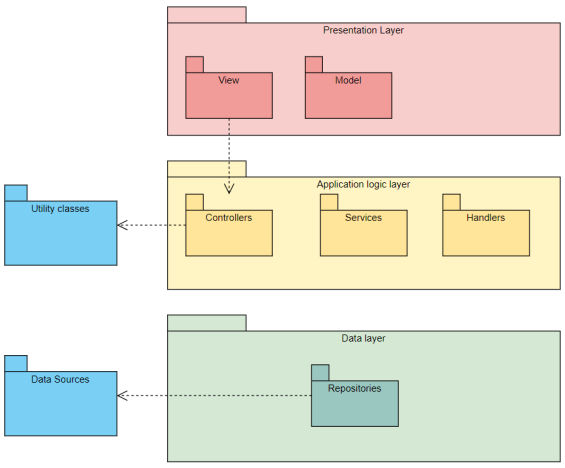


Figure 11: UBBTT's layers. Created using `https://online.visual-paradigm.com/`

## 5.2 Implementation

UBB Timetable is a web application developed using Java, and as an IDE I chose IntelliJ IDEA Ultimate, which is the paid version of the IDE, but is free for students. It has many advantages, such as the ease of integration of libraries and frameworks into projects and it offers many refactoring tools that can help identify and remove duplicate code by extracting it into a separate function or identifying redundant functions and replacing them with updated ones, renaming variables, classes and functions, but these are just a few.

As for the programming language, I chose Java for its many advantages such as the support for interfaces, automatic memory management, portability and its rich list of frameworks and libraries that it's associated with, which are listed below.

Because there are quite a few libraries and dependencies in the project, this task was automated as much as possible by Maven, which keeps all the needed dependencies alongside other information about them (such as their version) in a Project Object Model (POM) file called `pom.xml`, which is located in the top level directory of a project. If there ever are bugs or incompatibilities between different dependencies contained in the POM file, the version of the one that is causing the issue can be simply changed and IntelliJ will detect that the file has changed and ask the user for permission to download the new version of the dependency. The great part is that all that's needed is the correct name and version of a library, and Maven will automatically search in its repositories for that specific library. However, the user can also set a local path if the desired

library was already downloaded.

Maven also organizes a project's directory structure as follows (note that these are not all of them, just the most important): [5]

| | |
|---|---|
| `src/main/java` | Path containing application source files |
| `src/main/resources` | Path containing application resource files |
| `src/test/java` | Path containing application's tests source files |
| `src/test/resources` | Path containing application's tests resource files |

Figure 12 illustrates how the directory structure of UBB Timetable looks in IntelliJ IDEA Ultimate:

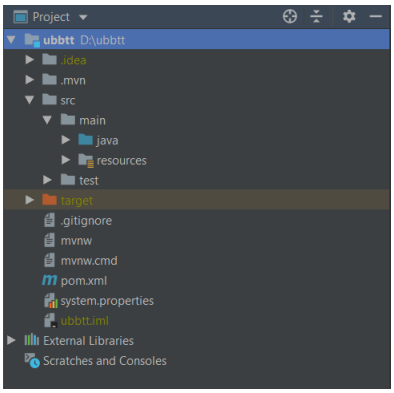


Figure 12: Directory structure of UBB Timetable as seen in IntelliJ IDEA

Using Maven, it was very easy to include all the dependencies required for this project. Some of them are: Spring Boot, Spring Security, Spring Web, Spring MVC, Lombok, Thymeleaf and Bootstrap.

Before Enterprise Java Beans (EJB) made its appearance, Java developers had to use JavaBeans (JB) to design web applications. Even though it was easier to create UI components, JB did not provide much needed features, like security, for example. EJB aimed to include features that JB did not have, but it still wasn't good enough, because the code could easily become complex and cluttered. Spring was introduced as an alternative to EJB. It is an open source framework that has an inversion of control container at its core, uses dependency injection to manage an application's components, and it also encourages good object oriented practices such as code reusability and splitting the application into layers.

The Spring framework is comprised of around 20 modules that are grouped into 7 categories [14], as depicted in figure 13.

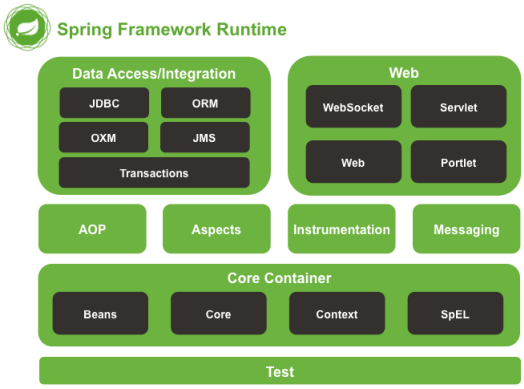


Figure 13: The structure of the Spring framework

Spring MVC (Model-View-Controller) makes it easy for applications that use this dependency to separate their own modules. The Model part will contain the plain old Java objects (POJOs) that represent the entities in the domain of the problem (e.g. for UBBTT a note or a user are such entities). The View will create the HTML code that contains the attributes provided in a `Model`. Last but not least, the Controller handles the requests received from the client (e.g. GET, POST requests), instantiates a `Model` that will be populated with attributes which will then get sent to the client (i.e. the web browser).

A very important library that I used in conjunction with the Spring framework is Lombok. While it can do many useful things, such as automate the logging process for an application or simplify the creation of the Builder design pattern, I only used it for what I think is its most important feature: removing boilerplate code. By using Lombok, I got rid of getters, setters and constructors by only using a few annotation above class declarations: @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor. The first two automatically define getter and setter functions for the variables inside the class. The @NoArgsConstructor annotation declares an empty constructor, and the @AllArgsConstructor declares a constructor that receives all fields as arguments and sets them accordingly. What I would also like from Lombok but currently it is not possible is an annotation that can declare a constructor with only some fields from the class, not all or none.

Another very useful dependency that was crucial in the development of UBB Timetable is Thymeleaf, which is a Java template engine for environments that use servlets (i.e. web environments), but also for non-web (i.e. standalone) ones. Also, it integrates with both IntelliJ IDEA and Spring [16]. It is very easy to send data from the back end of the application (e.g. a controller) to the

front end (the HTML page that the user will see displayed on their browser). Inside HTML files that use Thymeleaf, it is possible to write if-else statements, loops, and even iterate through maps in order to display them. For example, in order to display the list of notes for a teacher, I can add the `List<Note>` object to a model as a model attribute. The model alongside all its attributes will be available on the web page as soon as the request is handled and processed in its respective controller and the page that the client requested is displayed.

Another useful feature that Thymeleaf provides is that in order to create an object of some sort (a note for a specific university class, for example), it is possible to map the input fields in the HTML file in the front end of the application to the fields of the used class in the back end of the application. As soon as the client sends a POST request, the object will be already instantiated with the inputted data.

To make the graphical user interface (GUI) I've decided to use HTML with Bootstrap, because it provides an easy way to create beautifully designed web pages. It is also customizable, responsive and easy to set up. It also has a well written documentation, which includes many examples and previews of how they look on a web page.

In figure 14, "th:each" is used in order to fill an HTML table with the values from a map which was added to the model in the back-end of the application, in Spring. Each entry from the map will now have its own row in the table.

```
<tr th:each="CNMap : ${courseNoteMap926}">
    <td th:text="${CNMap.key.getDay()}">testday</td>
    <td th:text="${CNMap.key.getHours()}">testhrs</td>
    <td th:text="${CNMap.key.getFrequency()}">testfreq</td>
    <td th:text="${CNMap.key.getRoom()}">testroom</td>
    <td th:text="${CNMap.key.getType()}">testtype</td>
    <td th:text="${CNMap.key.getDiscipline()}">testdiscipline</td>
    <td th:text="${CNMap.key.getTeacher()}">testteacher</td>
</tr>
```

Figure 14: Iterating over a map in Thymeleaf

In figure 15, "th:if" and "th:switch" alongside "th:case" are used in order to display certain data on the page. For example, the buttons 1, 2 or 3 will appear based on what the function "getType()" will return, but only if the value of current mape's entry is not null.

All of the application's repositories extend the `JpaRepository` class provided by Spring. It also provides annotations such as @Query, which can be put above functions declared in repositories, and, as an argument, they receive a string containing an SQL query which specifies what to retrieve from the database. The function can also use input parameters which can be used in the query.

The login page is used to authenticate teachers into their accounts. To do so, they have to fill in their username and password in the login form contained in `login.html`. If the provided credentials are stored in the database, then they are logged in faster than the first time they try to do so. This is because the

```
<form th:if="${CNMap.value != null}" class="form-inline my-2 my-lg-0">
    <th:block th:switch="${CNMap.value.getType()}">
        <button th:case="'1'" >Button 1</button>
        <button th:case="'2'" >Button 2</button>
        <button th:case="'3'" >Button 3</button>
    </th:block>
</form>
```

Figure 15: Conditions in Thyemelaf

first time a user logs in the credentials will not be found in the database, and so an instance of `CustomAuthenticationFailureHandler` that implements the interface `AuthenticationFailureHandler` gets instantiated and the function "onAuthenticationFailure" is called and tries to authenticate the user through the faculty's server. If the server returns an OK response, then the username and password will be saved in UBBTT's database for easy access the next time. Once a user logs in, they will also be remembered through cookies, so that they do not have to log in each time they visit the website.

IntelliJ IDEA comes with a tool that can generate class diagrams for a project's packages. I've used this tool to create the diagrams for the most important packages of UBBTT.

The package that contains all the entities used in UBBTT is called "domain" and contains the following classes: `Course`, `Note`, `Week`, `User`, `Report`, `Role`. `Course` and `Week` are only used when adding new timetables to the database through their respective repositories (`CourseRepository` and `WeekRepository`) in a function named "onApplicationEvent", which executes when the application starts. The diagram only illustrates the classes' variables, without all the getters and setters.

Another useful package that contains miscellaneous classes is "utils". It is depicted in figure 17 and contains 5 classes, `ClassUtils`, which contains helper functions used to generate the timetable for every group and sorting the classes. The `GroupUtils` class contains functions that help to create arrays of strings that contain a group alongside it's subgroups and it's used for retrieving timetable information from the database for a specific group. `Pair` and `Tuple3` are two classes that have two, respectively three elements each and are used in storing data regarding the job shop problem, such as machine IDs, job IDs and the like. `HomepageTimetableUtils` generates the timetables for each needed group, which contains each class and its associated note from the teacher, if any.

In figure 18 are illustrated the 4 controllers in the application: `NoteController`, which handles the create, read, update, delete (CRUD) operations on class notes, `LoginController`, which handles the teachers' login. The login function there is called "tryLogin" because that is the function that attempts to authenticate a teacher that was already saved in the database. If this is the first login, then the function "onAuthenticationFailure" of the class `CustomAuthenticationFailureHandler` which was described above will be executed. The other controllers in this pack-

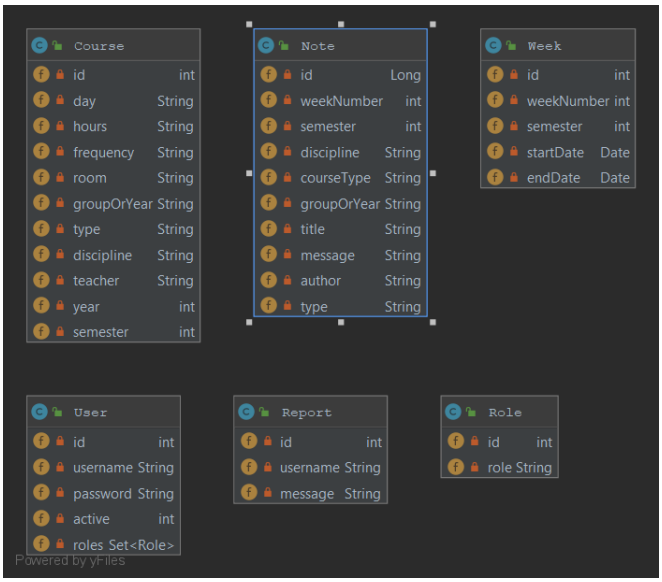


Figure 16: Class diagram of package "domain". Created using IntelliJ IDEA

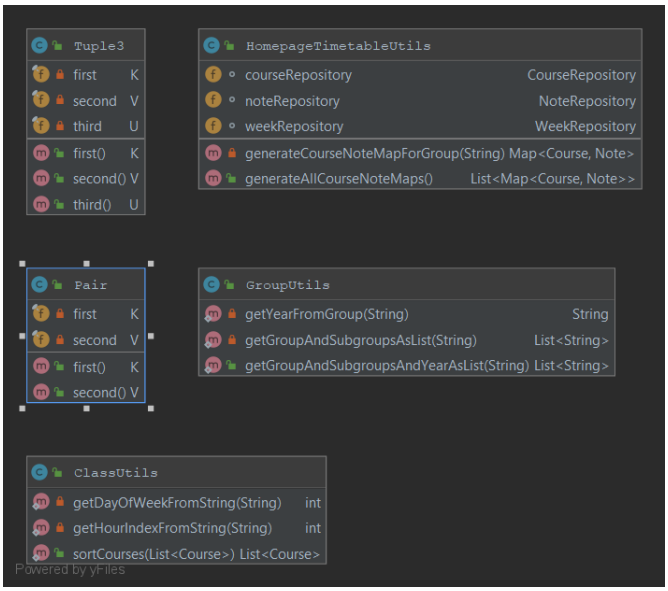


Figure 17: Class diagram of package "utils". Created using IntelliJ IDEA

age are `TimetableController`, which handles the creation of a timetable based on the flexible job shop scheduling problem algorithm and `ReportController`, which handles the feature that lets users report feedback to the developers.

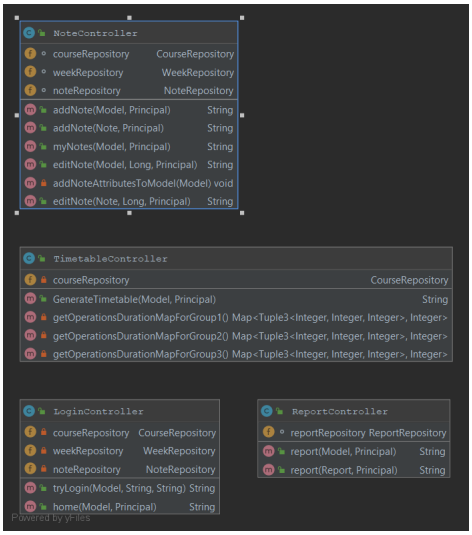


Figure 18: Class diagram of package "controllers". Created using IntelliJ IDEA

Figure 19 contains the three classes that implement the approach discussed in the previous chapter. The abstract class `JobShopSolver` serves as a base and contains a list of jobs (represented by the class `Job`), the list containing the sequences that will be generated for each machine and an array of makespans for each machine. It also contains some getters and the abstract method "solve".

`FJSSPSolver` implements `JobShopSolver`, overrides the "solve" function and contains other helper functions, such as the ones that initialize the makespan array with zeros, but also functions that are directly related to the approach, such as "getBestMachineForJobOperation". Last but not least, the function "printSequences" was used for debugging purposes and it prints the resulted sequences and makespans for each machine after the algorithm was executed.

The class `Job` contains its index, which starts from 1, the number of operations it contains, the operation duration map for its operations and some getters for the last two mentioned properties.

As for version control, I have decided to use Git [3] and the application is hosted on GitHub [4]. Besides the fact that GitHub is easy to use, intuitive and has a clean design, the main reason for choosing it is because IntelliJ IDEA provides GitHub integration. All that is needed is just to connect to the website in the IDE and then it's easy to commit, push, pull and manage branches directly from IntelliJ.

Even though UBBTT is not a large application with many lines of code, it is still open to expansion because of some principles that I have followed from the

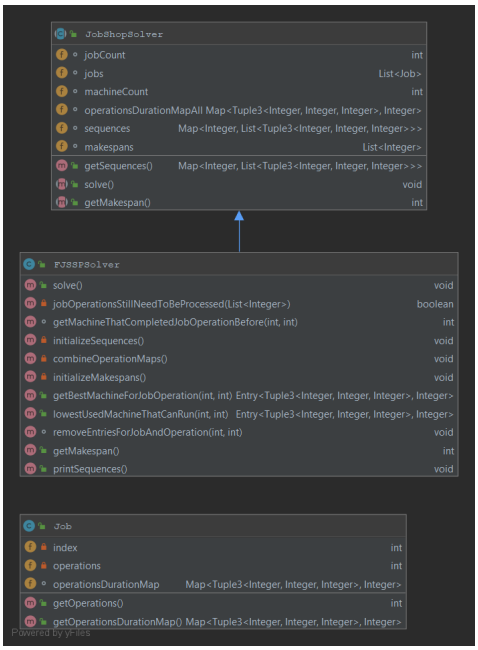


Figure 19: Job shop class diagram. Created using IntelliJ IDEA

book "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin[11], such as keeping classes as small as possible, using meaningful names for variables, functions and classes. Some other important rules are that I've applied were using as few arguments as possible for functions and also keeping the comments to a minimum, because the code should be as clear as it can be. Otherwise, if a developer changes the code and forgets to also update the comment, it will be outdated and will create confusion.

## 5.3 Persistence

All the domain entities in the UBB Timetable application were stored using a Java Database Connectivity (JDBC) Driver in a MySQL database. The connection details (such as the connection URL, username and password) are set in the `WebSecurityConfig` class, as a Spring bean. At source code level, the entities are stored in repositories, which are implemented using the Java Persistence API (JPA). In order to map each entity to a table in the database, JPA provides a very easy way to do to that: through the @Entity bean that is written just above a class name. Then, each field of the class is going to be represented in the database with the specified type by adding the bean @Column above it. There are more beans available, of course. For example, the ID of an entry in the database can be set by simply using the bean @Id, and its generation method can be set using the @GeneratedValue bean. For ex-

ample, if we want to have the ID automatically generated, all it takes is this: @GeneratedValue(strategy=GenerationType.AUTO).

In order to create a many to many relationship in the database, no additional class must be created, but instead the @ManyToMany annotation is used, alongside @JoinTable, which receives as an argument the name of the resulting table, and also the @JoinColumn annotation is used to determine which columns are included in the said table.

The database diagram of UBBTT is depicted in figure 20:

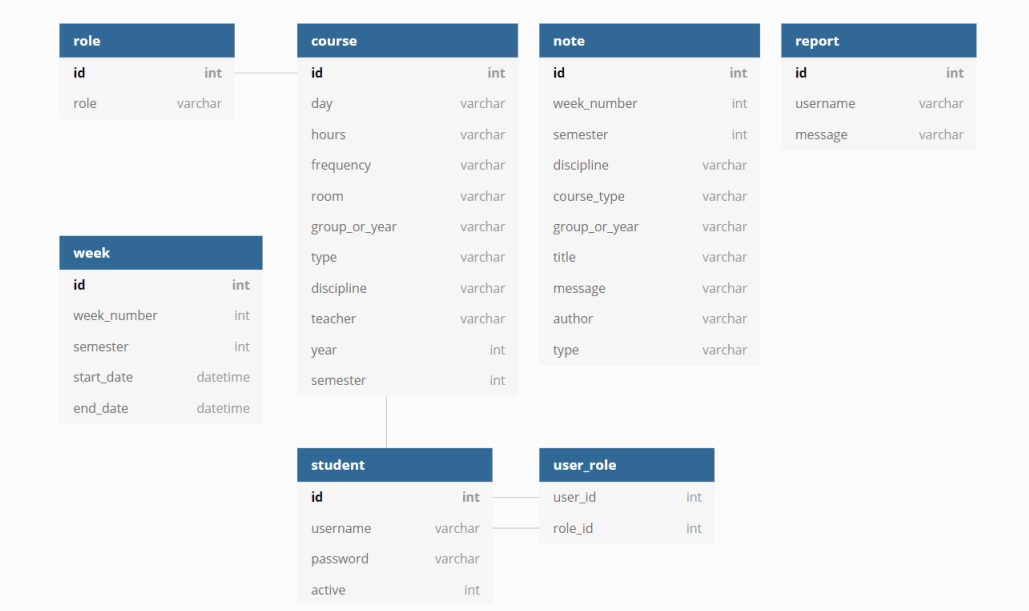


Figure 20: Database diagram of UBBTT. Created using `www.dbdiagram.io`

As it can be observed, even though the `role` table is used, all the users are automatically administrators, because only teachers can log in. But the table was kept in the database and the `Role` class was kept in the backend of the application in order for the application to be expanded to make students be able to log in, too.

## 5.4 Application Tests

To also test that Spring is working and it loads pages as intented, I've also written a number of tests aimed at checking if the correct page with its associated attributes provided in the page's model is loaded, or that the correct response is returned at page loading (e.g. 200 is the HTTP status code for indicating success).

The class `SubmitSuggestionTest` contains a test for the "Submit suggestion" feature, which allows users to send feedback to the developers of the

application. The test contains an instance of the class `MockMvc`, which, as the title suggests, mocks a model, view and controller and tries to access the page that contains the said feature, which is "/report". Then, the test checks that the status is OK (indicating a success status) and that the HTML content of the page contains strings, such as "Submit a suggestion/complaint", which is the title of the page. By doing that, the test checks that the opened page is the right one and contains the right data.

## 5.5 Implementation difficulties and overcoming them

When I started this application, I did not know exactly where it would end up, even though I had planned ahead of time how it would look. It was of much benefit to organize entities into separate directories (e.g. different folders for the problem domain, controllers, repositories, validators). Every so often, though, I had to take a step back at look at the big picture and notice that certain pieces of code were unnecessarily complex. A good example of that is the helper class `GroupUtils`, in which string manipulations were done in order to obtain the correct subgroups of a given group from college. I had a function for every group in that I was planning to make a timetable for, but there was a lot of repetitive code and it didn't hit me until later that the whole class could be cleaned up and reduced down to only 3 functions. Of course, this was just one example of the many places that I've changed the code in a manner that it became clear. A good chunk of the development time was spent cleaning classes, even if it feels that development was slowed down, it was actually sped up in the future as everything becomes easier to read.

The Spring framework is beautiful such that it makes a lot of trivial tasks simple that would otherwise take a lot more time to do, the developer not being able to be as efficient as possible. But I've hit a couple of walls in making use of all the features that were needed for the application. What helped me the most was IntelliJ Ultimate, which allowed me to install numerous libraries and set up Spring the way I liked. This IDE also helped me immensely in code refactoring, as it provides the users with many tools in order to rename variables, classes, remove duplicate code just to name a few. Another very important tool that helped me is the very well-written documentation of Spring that even contains sample code.

Another difficulty that I've come across was that in the beginning, Bootstrap did not seem to work on the web pages where I've included and used it. The problem was that access to the library's files was denied. All that was needed to fix this was to set up Spring Security in the `WebSecurityConfig` class in order to authorize access to JavaScript and CSS files.

## 5.6 User experience and user manual

Using UBBTT is a easy, straightforward and intuitive process. It has a simple design that does not look cluttered and is also responsive, thanks to Bootstrap's capabilities. When a client connects from a web browser to the

website, they will directly see the home page, which is also where the timetable for all the university's groups and their associated notes, if any, are found. At the beginning, no timetable is shown, letting the user choose.

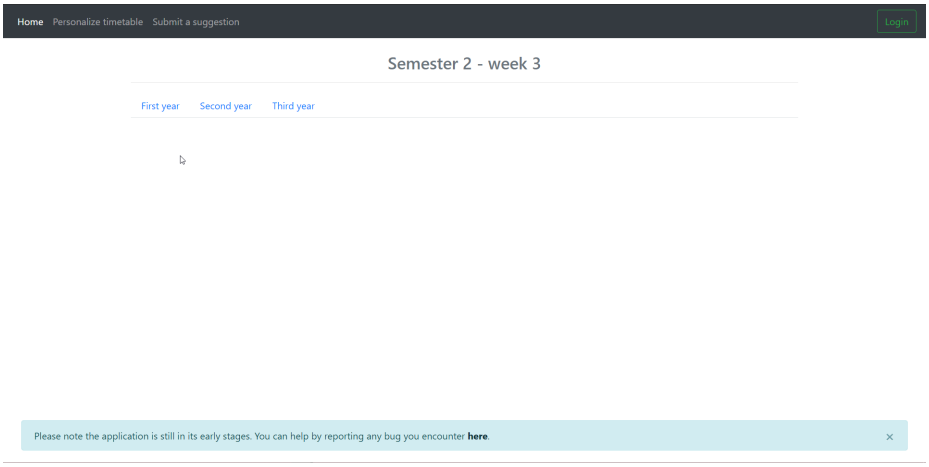


Figure 21: UBBTT's homepage and the pop-up regarding submitting feedback

In figure 22, the timetable for the group 921 is shown:

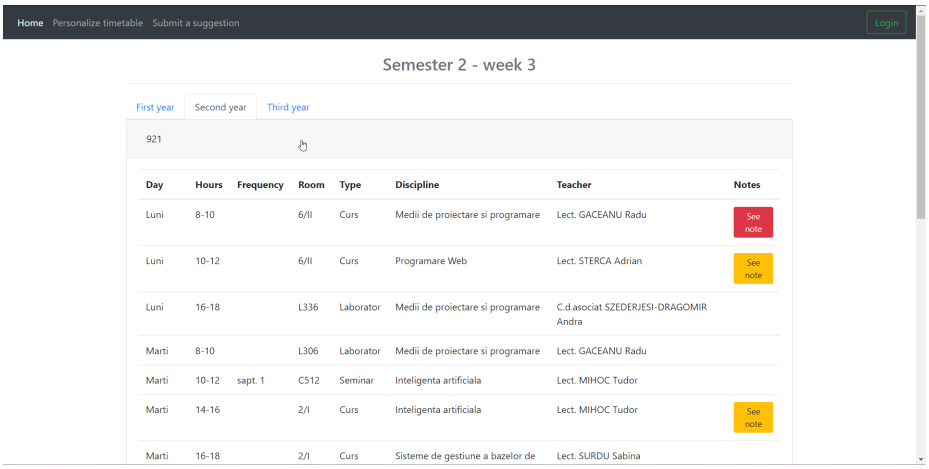


Figure 22: Timetable of group 921 shown on UBTT's homepage

At the bottom of the page, a small pop-up is shown as to indicate to the user that if they have a suggestion or want to report feedback to the website's developer they can do so by clicking on the pop-up's text. If the pop-up is closed, then the user can also press the "Submit a suggestion" button on the main menu. Then, all they have to do is enter the message they want to pass

on to the developers and hit "Send message".

If a teacher wants to log in, then they have to click the "Login" button in the right upper corner of the screen, enter their credentials that they use to log in to the faculty's server. If the login is successful, then they will be redirected back to the home page and to the main menu extra buttons will be available (such as the Notes section, where the teacher can view or edit their notes or create new ones). Figure 23 illustrates how the login page of the application looks like in the web browser:

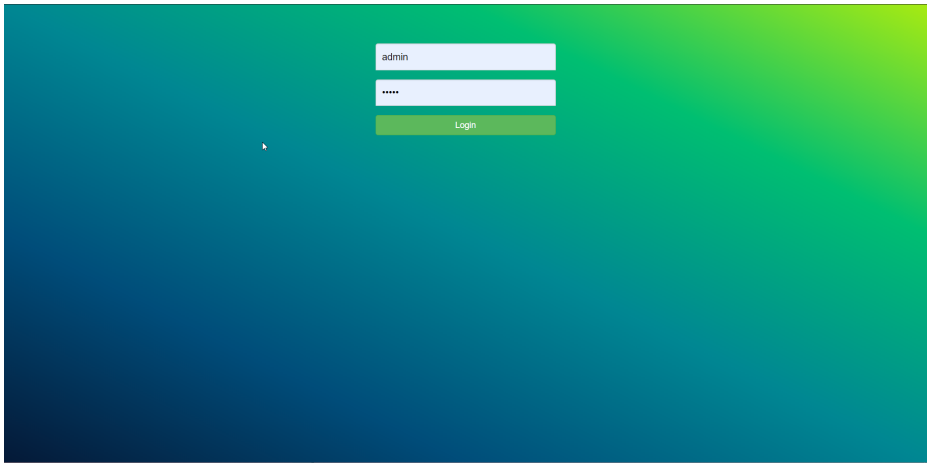


Figure 23: The teacher's login page

After logging in, teachers can go to "Notes" and then click on "Add note" to create a new note for students to see (figure 24). There, they are required to fill information such as the reason, the week, the affected group(s) and so on. When the form is filled, clicking on "Add note" will save the note into the database and will add it to the timetable on the homepage (figure 25).

Teachers can also view all their notes that they've added by going to "Notes" and clicking on "My notes". From there they can even edit or delete any of their notes, as shown in figure 26

In order for a user to see the personalized timetable, they have to click on the "Personalize timetable" button found on the main menu of the application. Then, the classes will appear in a table, sorted by day, as in 27.

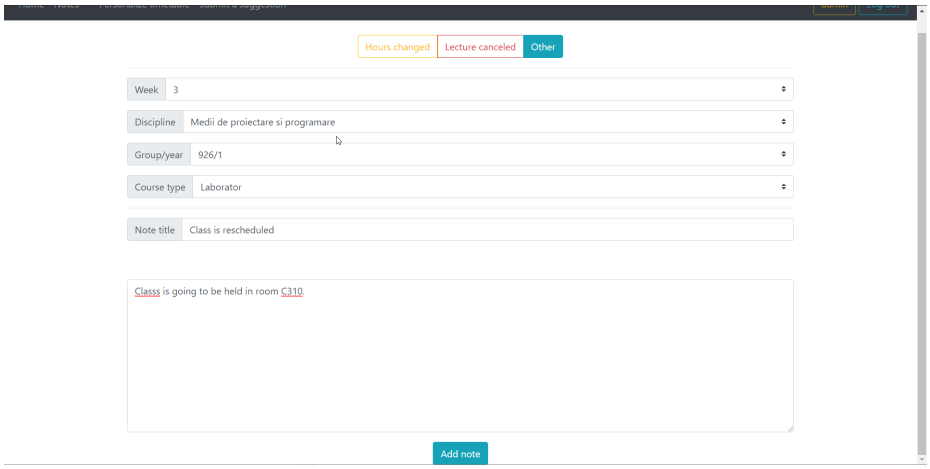


Figure 24: The "Add note" page filled with information

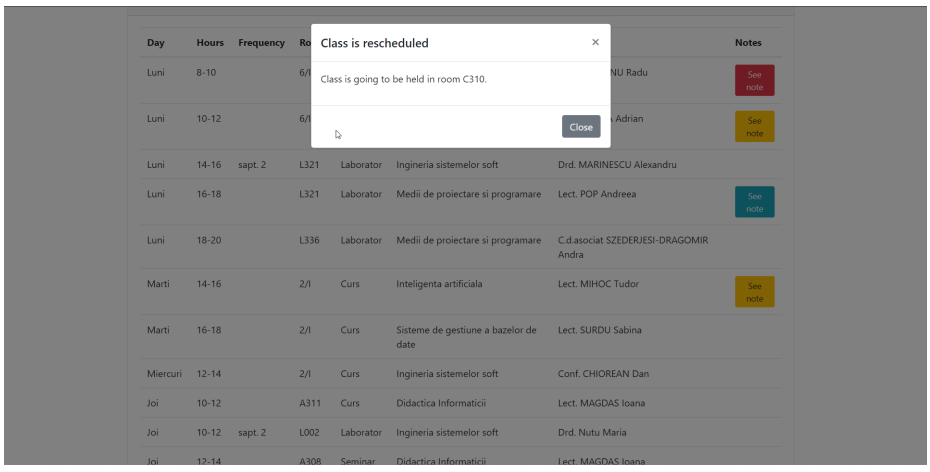


Figure 25: The newly created note placed on the timetable

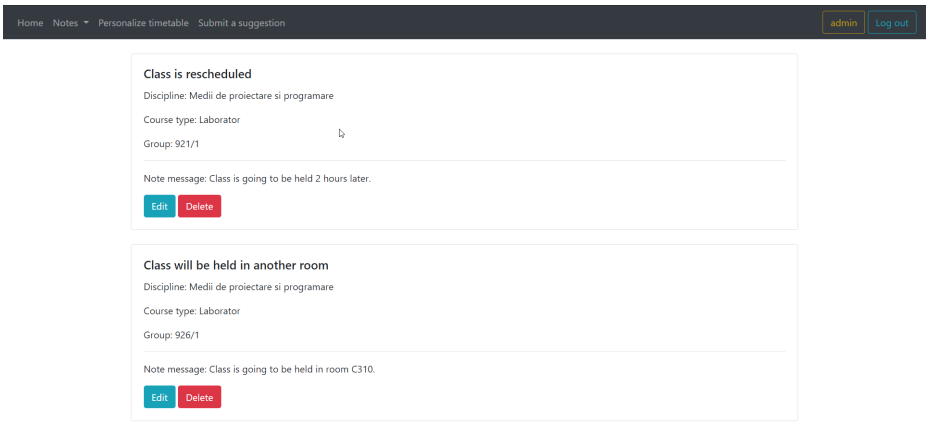


Figure 26: The "My notes" web page

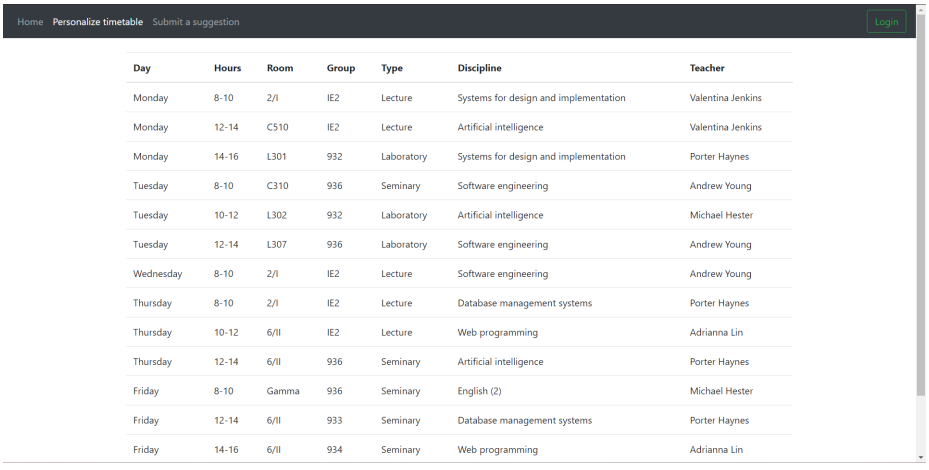


Figure 27: The "Personalize timetable" web page

# 6 Performance evaluation of proposed approach against relevant work

The proposed algorithm was tested having as input the data set from Jingcao Cai, Ming Li, and Zhihu Liu's proposed approach using an ACO-based algorithm [17] and the results between the two were compared. For this test case, there were 8 jobs, each containing 3 to 5 job operations and 10 machines. Below is the table taken from the aforementioned paper, which illustrates which machine is able to process which job, since this is a job shop problem with partial flexibility only. For a specific machine $M_i$ and for a specific job operation $o_{j,k}$ the entry can be an integer, which represents the time it takes for the $o_{j,k}$ to complete on $M_i$, or "-", which means that the job operation cannot be executed on $M_i$.

| Job | Operation | Machine | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ | $M_6$ | $M_7$ | $M_8$ | $M_9$ | $M_{10}$ |
| $J_1$ | $o_{1\text{-}1}$ | 2 | — | 4 | — | 10 | 3 | — | 8 | — | 5 |
| | $o_{1\text{-}2}$ | 5 | 7 | — | 8 | — | 9 | — | — | — | 11 |
| | $o_{1\text{-}3}$ | — | 6 | 5 | — | 10 | 9 | — | — | 4 | — |
| | $o_{1\text{-}4}$ | — | — | — | 7 | — | 5 | — | 8 | — | — |
| $J_2$ | $o_{2\text{-}1}$ | 10 | — | 4 | — | 10 | — | — | 8 | — | — |
| | $o_{2\text{-}2}$ | 9 | — | 8 | — | — | 7 | — | 6 | — | — |
| | $o_{2\text{-}3}$ | — | 6 | — | 5 | — | 8 | 7 | — | — | 4 |
| $J_3$ | $o_{3\text{-}1}$ | — | — | 9 | — | 6 | — | 4 | — | — | 5 |
| | $o_{3\text{-}2}$ | — | — | 6 | — | 8 | — | — | 7 | — | — |
| | $o_{3\text{-}3}$ | — | 9 | — | — | — | 5 | — | 6 | 4 | — |
| | $o_{3\text{-}4}$ | — | — | 4 | — | 3 | 6 | — | — | — | — |
| $J_4$ | $o_{4\text{-}1}$ | — | 8 | — | 5 | — | — | 2 | 4 | — | — |
| | $o_{4\text{-}2}$ | — | — | 6 | — | — | — | 8 | — | 5 | — |
| | $o_{4\text{-}3}$ | 5 | — | — | 8 | — | — | 4 | — | — | 9 |
| | $o_{4\text{-}4}$ | — | 3 | — | — | 11 | 6 | — | 8 | — | 10 |
| | $o_{4\text{-}5}$ | — | — | 2 | — | 5 | 8 | — | — | 8 | — |
| $J_5$ | $o_{5\text{-}1}$ | 6 | — | — | 4 | — | 3 | — | — | 8 | — |
| | $o_{5\text{-}2}$ | 3 | — | — | 7 | — | 6 | — | 10 | — | 8/9 |
| | $o_{5\text{-}3}$ | — | — | 9 | — | — | — | 4 | — | — | 11 |
| | $o_{5\text{-}4}$ | — | 7 | — | — | 6 | — | — | 7 | — | 9 |
| $J_6$ | $o_{6\text{-}1}$ | — | — | — | — | 10 | 8 | — | 5 | — | — |
| | $o_{6\text{-}2}$ | — | — | — | 3 | — | — | 2 | 7 | — | 3 |
| | $o_{6\text{-}3}$ | — | 3 | — | — | 10 | — | — | — | 6 | — |
| | $o_{6\text{-}4}$ | 10 | — | — | 6 | — | 10 | — | — | 9 | 8 |
| | $o_{6\text{-}5}$ | — | — | 7 | — | 8 | 9 | — | 10 | — | — |
| $J_7$ | $o_{7\text{-}1}$ | 9 | — | 5 | — | 7 | 9 | — | — | — | 8 |
| | $o_{7\text{-}2}$ | 2 | — | 6 | — | 8 | — | 3 | 7 | — | — |
| | $o_{7\text{-}3}$ | — | — | — | 5 | 6 | 2 | — | 9 | 3 | — |
| | $o_{7\text{-}4}$ | 9 | — | 15 | 5 | 3 | — | — | 6 | — | 5 |
| $J_8$ | $o_{8\text{-}1}$ | — | 3 | — | 6 | — | 8 | 5 | — | — | — |
| | $o_{8\text{-}2}$ | 4 | — | 6 | 3 | — | — | — | 8 | 10 | — |
| | $o_{8\text{-}3}$ | — | 5 | 8 | — | — | 6 | — | — | 5 | — |
| | $o_{8\text{-}4}$ | 2 | — | 6 | — | — | — | 6 | — | 3 | 8 |
| | $o_{8\text{-}5}$ | 3 | — | — | 5 | — | 3 | — | 5 | 9 | 8 |

Figure 28: Table containing the data set that the proposed approach received as input. Taken from [17]

| Test results | | | | |
|---|---|---|---|---|
| Algorithm | Best makespan | Average makespan | Worst makespan | Average elapsed time |
| Basic ACO | 28 | 32.4 | 36 | 9.327 |
| Improved ACO | 25 | 25.2 | 26 | 5.825 |
| Proposed approach | 35 | 53.193 | 80 | 0.385 |

Given that this is a greedy-based technique, the time it takes for the proposed approach's algorithm to run on a given test case is usually lower than for other algorithms based on artificial intelligence, because a lot of them start from random schedules and run for a various number of iterations until a good enough solution schedule is found.

However, even though the time it takes to generate a sequence for a given data set is low because of the greedy approach, the resulting makespan ($C_{max}$) is usually not as good as other algorithms' in many cases. This is a trade-off, because rather than minimizing $C_{max}$ and increasing the time for the sequence to generate, the two values even out.

But because the average time it takes for the algorithm to run is so low, it could be executed multiple times until a desirable makespan is met.

The heuristic which has a 20% chance of being applied at any step has proven to be very useful because this is what caused a new minimum makespan: 35. Even though 35 was seen as a makespan, over the course of multiple test runs (each test run solving the problem 1000 times), usually the minimum makespan ranges from 35 to 40, and the worst one from 70 to 80. The average value usually ranges from 35.1 to 35.6.

# 7 Conclusions and future improvements

My thesis started as a personal project because of my fondness for scheduling problems with practical usages in production environments, but also because of the desire to get a better grasp of Spring and both back-end and front-end technologies. While analyzing very different and unique approaches from literature, I went for an improved greedy-based algorithm.

In this project I have developed a web application aimed for making the communication between students and teachers easier by using different libraries, the most important ones being Spring and Thymeleaf. The application acted as a wrapper to my proposed practical solution for the flexible job shop problem by allowing users to generate new timetables based on various criteria.

An optimization that could be made in the future in order to tweak the proposed approach is to parallelize the algorithm such that at each step it checks the next operation that has to be completed for each job instead of processing one operation at a time. Another optimization could be to stop iterating through all the jobs if a particular job has ended. The way the algorithm executes now is that it still tries to find a best machine to run for the current job (but it will return a null value), even though all the operations have ended.

Regarding timetable generation, an improvement would be to try to even out the classes during the 5 days of the week, because sometimes, with the proposed approach, the generated timetable only makes use of 3 or 4 days.

To further improve the algorithm, support for class frequency should be designed: for example, there could be classes that are only held every 2 weeks, not every week.

To sum it up, writing this thesis and implementing the source code for the application assisted me in learning more about the job shop scheduling problem and the different proposed ideas that try to solve it while also designing my own approach, while also strengthening my abilities in developing front-end and back-end web applications.

# References


[1] Azzedine Bekkar et al. “An Iterative Greedy Insertion Technique for Flexible Job Shop Scheduling Problem”. In: *IFAC-PapersOnLine* 49 (Dec. 2016), pp. 1956–1961. DOI: 10.1016/j.ifacol.2016.07.917.

[2] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009, pp. 967–968. ISBN: 0262033844.

[3] *Git*. https://git-scm.com/. Accessed: 2020-06-10.

[4] *GitHub*. https://github.com/. Accessed: 2020-06-10.

[5] *Introduction to the Standard Directory Layout*. http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html. Accessed: 2020-05-02.

[6] Stephen H. Kaisler. *Birthing the Computer: From Relays to Vacuum Tubes*. Cambridge Scholars Publishing, 2016. ISBN: 1443896314, 9781443896313.

[7] Stephen H. Kaisler. *Together: Studies in Making Sociotechnical Order*. TODO, 1998. ISBN: 978-3-11-015630-0.

[8] *Konrad Zuse—the first relay computer*. https://history-computer.com/ModernComputer/Relays/Zuse.html. Accessed: 2020-05-02.

[9] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 5th. Springer Publishing Company, Incorporated, 2012, p. 384. ISBN: 3642244874.

[10] Jian Lin. “Backtracking search based hyper-heuristic for the flexible job-shop scheduling problem with fuzzy processing time”. In: *Engineering Applications of Artificial Intelligence* 77 (Jan. 2019), pp. 186–196. DOI: 10.1016/j.engappai.2018.10.008.

[11] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[12] Mahanim Omar, Adam Baharum, and Yahya Hasan. “A JOB-SHOP SCHEDULING PROBLEM (JSSP) USING GENETIC ALGORITHM (GA)”. In: (Jan. 2006).

[13] *ORNL researchers leverage GPU Tensor Cores to deliver unprecedented performance*. https://www.olcf.ornl.gov/2018/06/08/genomics-code-exceeds-exaops-on-summit-supercomputer/. Accessed: 2020-05-15.

[14] *Overview of Spring Framework*. https://docs.spring.io/spring/docs/5.0.0.RC2/spring-framework-reference/overview.html. Accessed: 2020-05-03.

[15] Rajni and Inderveer Chana. "Bacterial foraging based hyper-heuristic for resource scheduling in grid computing". In: *Future Generation Computer Systems* 29.3 (2013). Special Section: Recent Developments in High Performance Computing and Security, pp. 751–762. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2012.09.005. URL: http://www.sciencedirect.com/science/article/pii/S0167739X12001781.

[16] *Thymeleaf*. https://www.thymeleaf.org/. Accessed: 2020-06-05.

[17] Lei Wang et al. "Flexible Job Shop Scheduling Problem Using an Improved Ant Colony Optimization". In: *Scientific Programming* 2017 (Jan. 2017), pp. 1–11. DOI: 10.1155/2017/9016303.

[18] Wikipedia contributors. *Travelling salesman problem — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=960581811. [Online; accessed 5-June-2020]. 2020.

[19] David Wolpert and William Macready. "Macready, W.G.: No Free Lunch Theorems for Optimization. IEEE Transactions on Evolutionary Computation 1(1), 67-82". In: *Evolutionary Computation, IEEE Transactions on* 1 (May 1997), pp. 67–82. DOI: 10.1109/4235.585893.

[20] Takeshi Yamada and Ryohei Nakano. "A genetic algorithm with multi-step crossover for job-shop scheduling problems". In: Oct. 1995, pp. 146–151. ISBN: 0-85296-650-4. DOI: 10.1049/cp:19951040.

[21] Takeshi Yamada and Ryohei Nakano. "Genetic Algorithms for Job-Shop Scheduling Problems". In: *Modern Heuristic for Decision Support* (Mar. 1997).