# ExamScheduler - A System for Scheduling Faculty Exam Dates

Dor Bartov, Tal Gelber, Yosef Bader, Rephael Azoulay, Moisei Vainbaum, Roey Ashkenazy Supervised by: Tanya Brokhman

## ABSTRACT

Every student has faced at least once (and usually many times) a dense exam schedule, whether by exams in consecutive days, exams in the same day or even in the same hour.

The problem is that the current exam scheduling process is old-fashioned, inefficient, done manually and usually does not answer all of the different constraints that exist.

Therefore, we as a team built the ExamScheduler, an automatic exam scheduling system for the Computer Science faculty that is capable of handling a large number of courses and various constraints thereby answering the needs of the students, the semester representatives and the faculty staff.

It will the faculty staff to build exam schedules much faster than before (by not repeating the same steps every semester) and the students to have an all-around better exam schedule.

## I. INTRODUCTION

Our project aims to help the Computer Science faculty schedule exams in a quick and efficient manner.

Building a ‘good’ exam scheduler is important for both students and teaching staff of the faculty:

* The students will have time to study to each exam in order to succeed. The period of time between exams should not be too long since the students might forget the exam material.
* The teaching staff will be able to add restrictions to the schedule, i.e. a lecturer that cannot be present in certain dates can hand in a request and the exam for his course will not be scheduled in those dates.

Scheduling exams for the Computer Science faculty is currently done manually-

* Ullman issues predefined exam dates for certain first-year courses such as Algebra, since these courses are taken by students from many different faculties.
* The secretary of the Computer Science faculty receives these predefined dates and starts working on an exam schedule by iterating over many papers with information regarding the courses, constraints regarding certain exams and dates, requests from Semester Representatives (students that represent all other students of their semester) and Course Representatives (representatives for a certain course) and more.
* The secretary tries to build a good exam schedule that fits with all the constraints presented above.

This manual process is **very** inefficient: there is lots of information to process for each particular exam and complying with the different constraints can be an extremely challenging task that takes weeks to complete.

Furthermore, this process has to be repeated from scratch every semester and the resulting exam schedule is far from perfect due to human errors.

We wish to utilize computers in order to automate the process. The computing power will allow us to build a dynamic exam schedule that uses an advanced scheduling algorithm.

## II. RELATED WORK

Some of the existing algorithms try to handle scheduling with collisions or other constraints (as in our case) by using graph coloring – an assignment of colors to elements of a graph subject to certain constraints. In our case such an algorithm might try to define the colors as exam days and the elements as the courses.

According to **[1]**, the researchers reduced a general scheduling problem into a problem that has numerous constraints. Their algorithm attempts to:

* Shorten the exam period as much as possible.
* Allow more than one exam per day (depending on the input parameters).
* Solve collisions.

To define the weight of an edge between two courses, they used the number of students that are enrolled to both of these courses. (the size of the intersection of the groups of students enrolled in one of the courses).

This definition means the weight of an edge defines the priority of scheduling both exams in different days.

We took lots of ideas from the article. The ultimate reason we chose **not** to use graph coloring as described in their article is that we could not access the number of students who take two courses together. The article also tries to achieve the shortest exam period which we do not.

## III. DESIGN AND IMPLEMENTATION

### GUI Workflow

The ExamScheduler program is built in a modern design layout that features one major screen that has all of the needed functionalities to build an exam schedule. We will now explain each and every component of the GUI layout and then explain more thoroughly each component’s backend.

#### Course List Menu

The courses that are being displayed to the user are loaded automatically from a predefined course database with courses that are being taught this semester.

In this menu you can:

* Checkmark (✔) an exam for a certain course.

This is needed because some courses such as seminars do not have exams.

* Add a minimal number of days needed to study for this course’s exam. Some exams have huge amounts of study material and so the students need a minimal period of time to study.

The default option for the number of days is: *floor(course points)*. To clarify we take the course’s academic points (3,3.5 etc.) rounded down as the minimal time to study for this exam.

* **Pick a scheduling preference**: some courses tend to be scheduled early in the exam period (for example Programming Languages) where other exams are scheduled very late in the exam period (for example Operating Systems). Therefore, we added an option to choose an early/late exam, with an “automatic” option that has no particular preference.
* **Adding a fixed exam date**: if the exam for a certain course has to be scheduled in a specific day due to staff needs it is possible to drag the course name with the left mouse button unto that day in the exam period.

This fixed exam is taken into consideration during the algorithm run which means it will not be scheduled again in a different day and that the minimal time to study will be enforced by the algorithm (Note: this course has to be check-marked).

* **Course relations**: courses A and B will be defined as ‘related’ if they are taken by lots of students in the same semester. For our purpose we define related courses as those that are taken in the same “recommended” semester by the faculty since most of the students follow the recommendation.

We wouldn’t want to schedule related courses too close to each other in order to give the students time to study for both exams. If the Moed A exams have to be close to each other due to scheduling needs, the algorithm will try to space the exams as much as possible during Moed B.

To allow the user to input the ‘relation’ between courses, each course has a button that allows him to add multiple courses that are related to it (additionally this course is listed as ‘related’ to them automatically).

* **Add/Remove Course buttons**: as their name implies these buttons allow the user to add a new course to the course list or to remove a course from the list.

Upon adding a new course, the following input is needed:

* + course name
  + course number
  + number of academic points
  + ‘recommended’ semesters that feature this course.

#### Schedules

There are two big schedules for both Moed A exams and Moed B exams. These schedules are changed visually after the algorithm is executed.

Before the algorithm, this screen contains available exam days for a wanted time period.

It is possible to change the dates of the two exam periods and the screens will change accordingly.

Additionally, a day in the displayed schedule can be ‘locked’ which means no exams will be scheduled in that day. This is critical for days such as holidays or big faculty events.

Locking too many days will result in a very condensed exam schedule (some days will contain many exams) and so the user should avoid locking days as much as possible.

Once the exam schedules are filled with the wanted fixed exams (in the above-mentioned method) and the course list is configured and ready, the ‘Schedule Algorithm’ button will execute the scheduling algorithm (the in-depth explanation of the algorithm will be given later) and the resulting exam schedule will appear on both screens.

It is now possible to make adjustments to the schedules - for example the user can drag an exam from one date to another if wanted.

These manual changes are made by the user without the algorithm and so they can result in ‘related’ exams being too close to each other. In such a case a warning message will be given to the user during the manual action.

It is also possible to remove an exam by clicking the right mouse button on the exam itself.

If the user wants to view the exams scheduled for a particular course, we added a neat feature that allows him/her to move the mouse button over the wanted course in the course list menu. The scheduled exams for that course will be highlighted which allows him to see more clearly where they were scheduled.

#### Top Menu

This menu features the main functions of the ExamScheduler program:

* **‘Schedule Exam’** button: this button executes the main algorithm. All of the configurations mentioned in the article (wanted courses, fixed exams, locked days and more) will be taken into consideration by the algorithm.

The algorithm will try to schedule Moed B exams in a similar structure to the Moed A exams.

* **‘Save’** button: saves the current user configurations to the database. The next time the program will be launched the data entered by the user will be automatically loaded to allow repetitive exam scheduling every semester.
* **‘Clean’** button: cleans the schedules by removing all fixed exams. (DOES NOT cancel any course menu changes made by the user)
* **‘Export’** button: allows for easy exportation options for the resulting exam schedules, to allow the user to share/distribute them as he wishes.

We support numerous popular formats such as CSV and XML. In addition, we support Calendar-like file format (CSV file, displayed as calendar) and a screenshot for the schedules.

* **‘User Guide’** button: even though the program was built with the user in mind in a user-friendly way, we built a short user guide that is aimed to help the user get used to the various features of the ExamScheduler system.

### The Algorithm

In the field of Algorithms, scheduling is the method by which work is distributed among resources that complete it during a timeline.

Scheduling problems are considered NP-hard, which means the problem of scheduling limited resources in an optimal way cannot be done efficiently (in polynomial time).

There are a number of goals a scheduler can aim to achieve, such as maximum throughput (the total amount of work complete per time slot), minimizing wait time and more.

In our case, we aim to ensure there is sufficient time for students to prepare before an exam. As some courses (and, consequently, their exams) are more difficult than others, most students would prefer to have more preparation time before them. Usually this kind of information (course difficulty) is transferred by the semester representative to the faculty secretary. Also, we try to derive this information for the courses that were not classified by the representative. This can be done in a number of ways: number of credit points for the course, average grade, number of students that retake the course etc. As we are limited in available information, we chose to use the number of credit points as an indicator for the course difficulty. Thus, we define the amount of days needed for preparation as the number of credit points, which generally fits pretty well, but sometimes can be wrong (if 2 courses have same credit points weight but differ in real difficulty) or lead to unsolvable task (for example, an average student takes courses with mutual weight of around 20 credit points, while exam period can last for 2 weeks). Our algorithm takes this into account.

In our context, since the user story specified that the exam rooms will not be assigned by our scheduler, there is no theoretical limit on the number of exams in a given day.

Practically we cannot have too many exams in a short period of time since the students will not be able to study efficiently and due to room limitations.

Thus, input for the algorithm is:

* The courses (name, credit points, list of courses that share same semester with the course in the recommended course schedule, should be the course’s exam scheduled in beginning/end of an exam period or it doesn’t matter). The courses list will be sorted as follows:

Given two courses, C1 and C2, C1 will be considered greater than C2, if one of the following exists:

* + C2 needs to be scheduled at the start or end of the exams period, while there is no such constraint on C1.
  + C1 is required course while C2 is optional.
  + C1 number of collisions greater than C2 number of collisions.

C1 and C2 will be considered equals if one of the following exists:

* + The user has defined constraints for both courses.
  + Both courses have the same status (Both are required or Optional) and having the same number of collisions.

If none of the above accomplished, then C2 will be considered greater.

* Length of the exam period
* The list of courses with manually assigned exam dates (if any)
* *(Optional)* In case there is scheduling for the Moed B (retry of an exam) we want to ensure that between 2 exams for a course was sufficient gap. For this the algorithm for the Moed B need a complete schedule of the Moed A.

Important observations that guided us in algorithm development:

* The number of courses is relatively small (below 100 courses need to be scheduled by the faculty).
* While there are courses that are recommended to be taken in different semesters according to recommended systems for specializations, this is rare case and most courses are to be taken in same semester for different specializations.

These observations guarantee us that in practical case there exists solution for the task under our requirements.

The conflicts between courses are symmetric: that is, if after an exam of course A is needed time for preparation for course B, then after exam of course B is needed time for preparation for course A. It means, every time we schedule a particular exam to a day, we need to check exams both before the day and after.

Difficulties that were encountered:

* Straightforward solution (sort courses by number of conflicts with other courses, iterate over the list in descending order and in every iteration assign a course to the first suitable day) doesn’t work because of manually assigned exam dates - we can’t guarantee that there will be suitable day for a course, as it can be engaged by a manually assigned course that conflicts with the scheduled course.
* Generally speaking, we can’t even guarantee that solution that fulfills all our requirements exists, because one can manually arrange some of courses in such way that will prevent any other course to be scheduled without overlapping with conflict courses. Still, our observations (described above) give us confidence that in many cases it won’t happen, but if will, we have to weaken our requirements until the solution will be found, as we search for best possible schedule in general, while our requirements just help to define what it is.

Based on previous reasoning, the main idea of the algorithm is: produce legal schedule under main restriction (number of days for preparing; weaken restriction if the producing of the legal schedule is stuck on the course, when by weakening is meant possible reducing of days needed for the preparation for the exam, but not more than reducing in a half and not let to be this value less than 1) and improve it afterward while possible or until all other requirements are met. Here by “legal schedule” we mean that no courses with conflicts can be assigned to same day.

The most challenging task here is producing a legal schedule. As the scheduler assigns only one exam every iteration, we have to ensure that after every iteration we can proceed forward and won’t end in situation when an exam can’t be scheduled because of overlapping with conflict course. The way to ensure it is to choose date for exam in smart way. In our case, we define that the best day for an exam is such day, that will minimize amount of “busy days” for the exam period, when by “busy day” we define day when an exam is held or day for preparation. Thus, we define a heuristic function for a course:

when sum goes over all specializations that related to the course and is number of busy days for the specialization p. (Note: here, when we speak about specialization, we mean pair “specialization, semester”). This value should be minimized when the algorithm assigns an exam to a day.

The algorithm uses the following data structures:

* An array list for schedulable days (days that an exam can be held on them).
* A class that represent a day (contains date of the day and mapping “a course number” => “days to exam of the course”. Can hold both negative and positive values).
* A mapping between a pair “Specialization name, order number of a semester” and Boolean array that shows in what days will be exams for this semester held.

The exact flow of the algorithm is:

1. Initiate data structures for the algorithm (a day initiated to the appropriate date and empty mapping, array list of days will contain all the initiated days and the mapping is also empty in time of initiation)
2. Assign exams that were scheduled manually to appropriate days. Assigning is done in next way: go to a day of date “date of the exam - amount of days that needed for preparation for the course”, insert in its mapping pair “the course number, -number of days before the exam will be hold”. Repeat the insertion for all other days from this day till end of the exam period. (Important note: here we don’t check if there are overlapping of conflicting courses- that is done when a user provides input). Also, update the array of busy days for the relevant pair “Specialization, semester” for further heuristic computations.
3. Delete courses that were assigned manually from the list of courses for scheduling
4. Iterate over sorted list of remaining courses in descending order, when sorting is done by tuple (a, b, c) and:
   1. “a” is a Boolean value which indicates if amount of days for preparation is assigned manually or derived automatically
   2. “b” is a Boolean value that indicates if the exam should be scheduled in beginning/end of the exam period.
   3. “c” is number of conflict courses (that their exams shall not overlap with the course’s exam)
   4. We compare two tuples in such way: first compare “a” values, the tuple with biggest value wins. In case of equivalence proceed to “b” etc. Thus, we guarantee that manual definitions for a course prevail over automatic deriving of parameters.
5. In every iteration find the best day for the exam of the course (i.e. day that minimizes heuristic value of the course) and assign the exam to the day.
6. If the algorithm is stuck on a course (there is no free space for preparation for the exam) - decrease number of days needed for preparation for the exam. If the number can’t be decreased, produce error message (basically, it is only place when we give up on scheduling, because this case means that parameters that were provided manually prevents producing of legal schedule under given restrictions).
7. After the legal schedule is ready, optimize it. Basically, it can be done in different ways, up to customer needs. In our case, we define “optimized schedule” to be the schedule, which has roughly the same number of exams in a day (as mentioned above, there are practical limitations of number of exams in days, because of number of controllers, classrooms etc.). Optimization is done by iteration over exam days, finding day that has more exams than other and attempting to reassign an exam from the day under saving of schedule legality.

### The Database

The database of the system is a list of folders, where each one has a name that identifies it to a single semester. Each directory contains several XML files that describe the relevant information of this specific semester for exam scheduling.

XML language was chosen for our database as it offers both portability and convenient ways of checking the structure and the values of the data stored in the files. Moreover, Java offers a built-in library for XML handling.

The files include:

* The courses taken by Computer Science students, which are taught this semester.

The information stored for each course contains the name, course number (based on the faculty catalogue), amount of academic points for the course and more.

We also note which "recommended semesters" contain this course and for which degrees. For example, Introduction to Systems Programming (MATAM) is present in the recommended 2nd semester for all computer science related degrees.

* The various degrees handled by the faculty (Computer Science, Software Engineering and more)
* Collisions between courses. Initialized to an empty file.
* Two files for constraints - Moed A and Moed B. Initialized to empty files.
* Two files for the output exam dates – each course has the exam date of Moed A and the exam data of Moed B in different files.

During the initial loading of the program, there is a transformation of all courses and degrees from the DB into relevant Java objects (At the current state of the project, it is only possible to load and save a single semester using the GUI system, although the code of the database already has an API allowing to create, remove and modify new, distinct semesters).

Next there is an iteration process where the system deducts which courses collide with one another. As a result, each course has a list of colliding courses in one or more recommended semesters of the same degree.

Additionally, each course has a 'days before' attribute that signifies the number of days needed to study for this exam (as described above). The algorithm tries to adhere to this number as much as possible for all exams.

As default options we do not give first/last priority to course exams (exams that are usually scheduled early/late in the exam period). Also, we assume all courses have an exam until told otherwise by the user using the GUI.

After the data is loaded, it is accessed by the scheduling algorithm through a dedicated API that ensure the data stays valid and legal (more on that later).

After the scheduling algorithm is executed and all of the manual corrections are done by the user, there is an option to save the state of the schedule for future use by the program.

This is performed at the back-end by:

* Saving the relevant constraints for Moed A and Moed B.
* Saving new faculty-degrees to the database.
* Saving new courses added by the user to the course database.
* Removing unwanted courses from the database.
* Saving all the scheduled exams and their dates for both periods.
* If the user added any collision by manually changing the output of the algorithm, it will be saved among all other collisions and constraints.

All of this information will be loaded during the next time the program is executed.

The other major role of the database is to ensure all the data stays legal and valid during the usage of the program by the user.

* Legality of the data means the user cannot input an “illogical” value in the system, such as a number containing letters in the ID of a course, or a course with an empty name. Those problems are generally mistakes made by the user and are easy to detect.
* Validity of the data means the user cannot create contradictions in the existing data that might impair the scheduling algorithm, for example by assigning the same ID to more than one course, or by defining an exam date to a course that is outside of the exam period. This is more difficult to handle as it requires to analyze and compare the user input to all the existing data.

Legality of the data is handled by using XSD Schemes. Each of the XML files of the database has a XSD Scheme that define how the file must be structured, and which values and types of values are allowed in each field. The legality checking is done when loading data from the XML files, and directly through the GUI by checking the user input.

* Validity of the data is done at two points in the database. When the data is loaded at the start of the program, the following checks are performed:
* There is no duplicate course, study program, conflicts or constraint in the database.
* There is no reference to courses that don’t exist in the schedule, the conflict list or the constraint list.
* All the existing constraints and schedules have logical dates: each constraint has a date, each schedule has a start date and an end date, each constraint date is between the start and the end of the schedule.
* There are no contradictions between the flags and other data relevant only to the implementation of the scheduling algorithm.

When the data is modified, either by adding or removing some data, or by editing already existing data, those checks are also performed.

It is important to note that the saving of the data is not verified, which means we don’t check if the writing of the data into XML files was done successfully. This is due to the fact that the way we check legality and validity of the data allows us to assert that data is correct when written to XML file. Thus, if the data is not saved correctly into XML files, it can only come from the underlying XML library and not from our code.

## IV. EVALUATION

We planned to check the database functionalities:

* Ensure the correct loading of the stored data.
* Ensure the correct saving of the data into XML files.
* Ensure the correct manipulation of the data.
* Prevent the introduction of contradictions in the data when manipulated by the user.

We wrote several unit tests that we can split into 2 categories:

* Tests that check that a specific error is correctly handled. For each one of those tests, a dummy database was created, containing minimal data with one precise error. The test should try to load/modify the database, detect and report the corresponding error, and abort the operation without modifying the existing data (or preventing loading of bad data in the system).
* Tests that check that loading, saving and manipulations allowed on the database are performed correctly when they are valid. Each test tries to execute several actions, and check that no errors were encountered and that the operations were performed successfully.

As for algorithm, we were up to ensure that the algorithm provides legal, optimized schedule.

First, was performed unit testing of every unit to ensure that every part is working correctly independent of others.

Next, we proceeded for complete algorithm testing. For this purpose, we took real exam period (of Winter 2017). Important note: a winter semester has much shorter exam period than a spring semester, so the checking was done for the hardest case. All exams of courses that belong to faculties, others than CS were assigned manually to their real dates. Afterward, the algorithm was run on the input.

To be sure that algorithm really produced legal schedule, some tests were run on the schedule:

* Checking that every manually assigned course remained in place
* Checking that every exam was scheduled
* Checking that there is no course that overlaps with conflict courses

The ran tests showed that the algorithm performed its task correctly.

## V. CONCLUSION

We learned an important lesson in software development - how important teamwork and communication are, how to manage time properly, how to present our work to the class and how integration of the different program parts (GUI, Business logic and Database) can take time and has to be done carefully.

We believe the resulting program we made accomplished most if not all of the faculty demands, and the resulting exam schedules made by the system will be better than the current schedules while being made much faster and more easily.

Future work- when confronted with various faculty members and students there was an interesting suggestion that repeated itself multiple times - the program should allow different faculty members to access the system concurrently in some way.

This will allow them to make recommendations or even change the schedule configurations themselves.

## REFERENCES

1. **A New Exam Scheduling Algorithm Using Graph Coloring, by Mohammad Malkawi, Mohammad Al-Haj Hassan, and Osama Al-Haj Hassan, published in The International Arab Journal of Information Technology, Vol. 5, No. 1, January 2008.**
2. **Computer Science Department - Course List,**

http://www.cs.technion.ac.il/courses/all/by-number/index.html

1. **Computer Science Department - Catalogue,**

http://www.cs.technion.ac.il/he/undergraduate/programs/cs\_catalog\_2017\_2018.pdf

1. **Computer Science Department secretariat - Mika.**