# Course Planning for University-Grade Curricula

Andrei-Mihai Ciorba, Laura Cristina Gheorghe

**Abstract.** This paper describes a near-optimal method for scheduling course placement in a university programme with regards to the number of courses per semester, inter-disciplinary prerequisites and other optional external factors. The paper also proposes a method for representing course data relevant to the main purpose of scheduling. The proposed method is to be implemented in a software application developed in Java, using predefined XML files as inputs. The application will output a description of the calculated scheduling method, as well as a graphical representation.

**Keywords:** scheduling, course, programme, critical path

## 1 Introduction

Course design is an extremely important task for every university or any other teaching institution, for that matter. Even more, the task of interconnecting courses can be broken down to the requirement for a "connection-oriented" knowledge.

As with every university, the goal of a programme is to provide the necessary knowledge in a context of constant coherence. Still, knowledge has little value without the necessary skills and practice needed in order to apply that knowledge to real-life usual situations and, most importantly, challenges. Courses must be regarded as building blocks for the skills they are meant to develop.

While considering this perspective, one must realise the importance of having a well-developed structure among different courses, especially because each course is usually focused on a specific detail, technology, situation or problem of a specific field. Even if it is a general course, spatial and temporal limitations still occur and cause the course to focus on a distinctive element that closely resembles the general aspect of its topic. This is where the problem of scheduling different courses, on various topics, while still maintaining the necessary coherence from a student perspective, starts rising question marks.

The paper at hand will offer a starting point for a solution to this problem. Initially, the first goal will be to develop a standard view that can act as an abstract model for representing the main element, a course. Then, it will provide a mechanism for interpreting not only the logical connections between these courses, but also for generating a valid result of scheduling. This result will have to meet the standard requirements of each course (prerequisites), as well as other limitations that correspond to each case in particular.

For example, a semester can only have a maximum number of courses, certain courses need to develop certain skills sooner or later in the teaching process and other courses might have to be scheduled differently because of their complexity.

Generating a valid answer (or even more) will require an application that can demonstrate how such an algorithm can be implemented. Of course, the application will have to process a limited amount of information with a limited set of requirements that might not always return a valid and possible answer. On the other hand, the application is aimed at providing several solutions to the same set of data offered for input, when possible.

Besides the utility of such a method that is to be backed up by a working implementation, the impact of such a project might address the view of a university's administrative staff. It might offer suggestions for improving the current structure of the teaching process or it might rise inadvertences regarding courses that were presumed to be relevant, well-placed in the programme and efficient from a knowledge interconnection point of view.

This paper includes the method architecture and how it relates to the subject at hand, a thorough analysis of an example set of data and its result and ideas for future development.

## 2 Related Work

No relevant open-source work that can be directly conncted to the subject at hand was found available at the time of writing of this article. The only closely related piece of software is the Mimosa Scheduling Software from Mimosa Software which, despite being a fairly complex program, only focuses on timetable generation, maintenance and optimization. It can generate a wide variety of reports and supports several open formats in order to interact with other applications, but has no utility for semester-based scheduling.

Mimosa Scheduling Software is a commercial product.

## 3 Software Architecture

The software is an application written entirely in the Java programming language, using only Java SE 1.6 classes and libraries. The IDE used for code development was Eclipse, along with NetBeans for the graphical user interface (GUI) design and debugging purposes.

A third-party open-source library (JGraph) was used to generate the graphical representation of the result. The original JGraph code was not altered in any way.

**Input**

The application takes its input from a series of XML files located under the “inputs” directory of the project. Each XML file represents a distinct course and contains all the necessary information to identify, describe and schedule the course.

Upon initialization, the appplication will read all XML files from the “inputs” folder and store them in memory. As a general guideline, it is expected that these XML file names will contain only alphanumeric characters. If a file name starts with an underscore (the “_” character), then the application will ignore it. This was implemented in order to provide a method for removing courses from the scheduling algorithm without physically deleting the corresponding files on the disk.

The number of valid input files is expected to be the real number of courses given as input to the application. It is not expected for a course to be duplicated.

The basic structure of a course input file is comprised in a <course> tag and contains the following elements (tags, basically):

- Name. This is the full title of the course and has to be unique among all input data. A name uniquely identifies a course and the same string is also used to describe dependencies. It is advisable to use the same name for the file name as well, without the white spaces.
- Description. The description is simply a text field that does not have any constraints and is not interpreted by the application in any way. It simply provides more detail about the course’s content or special conditions that don’t regard scheduling. It can also be used by other applications that accept the same input file format.
- Initials. The abbreviated name of the course. Can be used when only a short name is required. This abbreviation is not interpreted by the application in any way and does not have to be unique among all input data, although it might cause confusion when reading the output data otherwise. The course initials are also found on the final dependency graph generated by the application.
- Professor. Name of the professor, does not have any constraints.
- Requirements. The requirements tag must not contain any character data. Instead, this tag will constain zero or more <reqname> tags, each containing the course name of a prerequisite of the current course. Prerequisites are described textually, using the course name itself. These requirements must also be found in the same input directory and must be described by valid XML files in order to fulfill them and run the scheduling algorithm. There are no limits on the number of possible dependencies. Basically, a if a course is a prerequisite for another, it means that the latter one cane be scheduled only starting with the next semester, immediately following the highest dependency (latest semester, in time).
- Minimum year. Optional parameter for the scheduling algorithm. Specifies the minimum year in which the course should be scheduled.
- Maximum year. Optional parameter for the scheduling algorithm. Specifies the maximum year in which the course should be scheduled.

- Preferred Semester. Numerical value that can only store values 1 or 2, indicating the first or the second semester. This parameter, by itself, does not force any restrictions upon the year in which the course can be scheduled.
- Credits. Numerical value that stores the number of credits of a certain course. The application does not process this value, so it is used only for descriptive purposes.

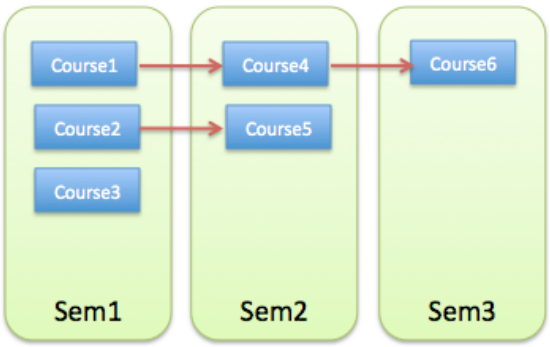
**The Scheduling Method**

After the application initializes the inputs directory is scanned for valid input files. All valid XML files (unmarked for deletion, that is) are loaded into memory, parsed and processed in order to build a graph. The resulting graph contains all courses (as nodes) and all the connectors that represent course interdependencies.

For example, the following connection indicates that Course2 has one prerequisite, Course1:


By default, it is considered that each semester has a maximum of 5 mandatory courses and can have as many optional courses as the administrator wishes to schedule.

A dependency between two courses requires the latter course to be scheduled at least one semester later after its latest dependency. This means that inter-dependencies cannot occur on the same level (semester) while scheduling. For example, the following situation displays a simple case of scheduling 6 interdependent courses without the possibility of fully completing any of the involved semester slots:



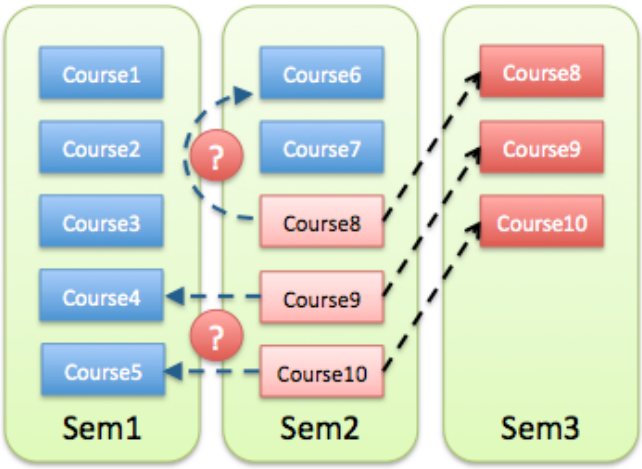

The scheduling algorithm begins by running a topological sort operation on all courses with regards to their interdependencies. The topological sort will remove the first level of courses, those that have no prerequisites and assign them a level 1 value. Further on, the same procedure is repeated until all courses have been parsed and assigned to a level.

The level assignment that takes place during the topological sort is a theoretical model that could represent the semester allocation by itself, were it not for the limit of courses per semester. Theoretically, the topological sort returns a scheduled form in the most compact form possible, since all courses occupy their lowest possible respective places. No course shifting is performed since there is no limit on how many courses can be scheduled per level.

For example, a theoretical three-semester scheduling topological sort might return the following result:


Each square represents a distinct course and the numbers represented on the squares are their associated level. Obviously enough, this wouldn't be a correct scheduling result for a maximum of five courses per semester because courses from the first level of the topological sort would have to "leak" to the next semester.

The problem that arises here is that scheduling the first seven courses (level 1) would fill the first semester and two slots from the second. Now, when attempting to schedule the three level 2 courses that follow we would be forces to jump all the way to the third semester and fill three slots from it. The disadvantage here is that the remaining three free slots in the second semester will always remain empty.




Our topological sort only gives us a sorted model based on level dependencies. That is, we know that a course from level 2 must depend on a course in level 1 so if we list all level 1 courses followed by all level 2 courses, we can be sure that iterating through the courses in the given order is not going to provide errors due to dependency inconsistencies. However, the topological sort cannot provide any information regarding exaclty which courses relate to each other.

The second semester would remain only partially completed because it contains at least a level 1 course. So, in this case, we cannot attempt to schedule a new level 2 course on the second semester without knowing whether this level 2 course depends

on a level 1 course already situated on the same level. Even more, it is likely that there would be one ore more course already scheduled on the first semester that could switch places with those on the second in order to accommodate the level 2 courses, but this would greately increases the complexity of the algorithm; in the end, it would require to generate all valid solutions in order to select the best one, so this is not the desired method.

In order to accommodate courses in all free slots, a supplementary operation is carried out before scheduling.

Whithin each level, courses are being sorted based on their number of "children", or courses that depend on them, in descending order. This way, two things are ensured:

- Courses that have the highest number of courses depeding on them will be placed as close to the "bottom" as possible; that is, the probability for them to get shifted up one or more semester along with all their children is slightly reduced, keeping the course structure more compact.
- Courses that have the fewest (or none at all) courses depending on them will end up towards the end of the sort and are much less likely to cause any further shifting.

This extra sorting step provides some optimization for the overall result but still fails to completely resolve the problem described in the previous diagram. That is, a level 2 course might still be scheduled on the second semester, along with one or more of its dependencies. To avoid this situation, another procedure is added to the scheduling algorithm.

All courses have a default level obtained through the topological sort operation. This level is used to indicate the first semester on which the course will attempt to be scheduled. This level can be increased in two cases:

- If there are no more free slots in the desired semester and the course needs to be scheduled in the next one;
- If there are conflicts between the level/semester and one or more courses located on this level, of which the scheduled course depends upon.

Increasing the level of one course has the recursive effect of increasing the levels of all courses that depend upon it, up to the last one.

Whenever a course is placed on a semester that does not correspond to its stored level (and has a higher numerical value), all subsequent courses connected to the course at hand will get their levels increased, too. Still, subsequent courses' levels will get incremented only if they are currently on a level where they should not be. Otherwise, we might face certain situations where a course that has two or more prerequisites already placed on the same semester gets its level incremented twice or more, even though one time would have been enough to accomodate both requirements.

### Output

The program outputs the result as a graphical structure. All courses are represented, along with their corresponding connections, grouped by semester. The graphical interface allows editing of each course. Modifications are dynamically updated in the graph window.

The GUI also allows the user to save the result to an XML file. The name of the XML file generated automatically as "default" followed by the current date and time.

The output file's format is as follows:

```
<year id="1">
    <semester id="1">
        <course> course name 1 </course>
    </semester>
    <semester id="2">
        <course> course name 2 </course>
    </semester>
</year>
```

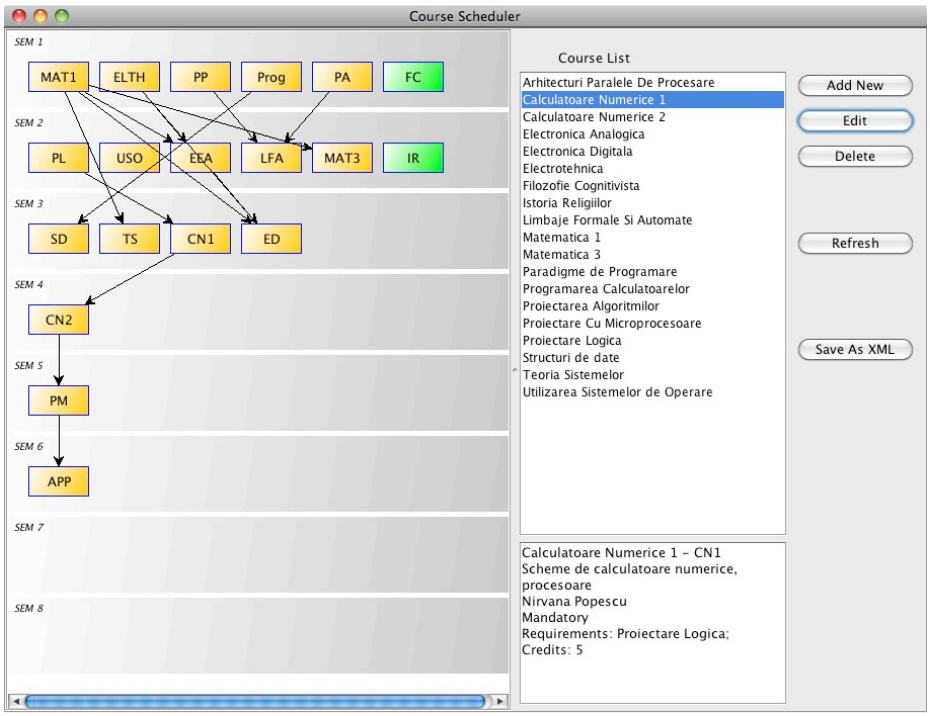
### The Graphical Interface

The GUI draws the resulting graph and allows the user to add, edit and delete courses. Deletion does not remove the physical file from the disk.

Editing an existing course allows the user to modify any of the course's parameters. The course's name, the professor's name, the description and the initials are text parameters and can be directly modified. The course's position, described by the minimum year and the preferred semester can be chosen by using two drop-down lists. The list of dependencies can also be manually edited or can be automatically generated by chosing the desired courses from a list (recommended).

## 4 Test Scenario

The test scenario that follows involves a number of 19 courses, optional and mandatory as well.




The algorithm places all courses on their corresponding semester. Optional courses are displayed on the right hand side of the graph, without any dependencies.

Arrows in the generated graph indicate the relationships between courses. For the current example, the maximum number of courses per semester has been set to 5.

The resulting XML file describing the scheduling is the following:

```
<year id="1">
          <semester id="1">
            <course>Matematica 1</course>
            <course>Electrotehnica</course>
            <course>Paradigme de Programare</course>
            <course>Programarea Calculatoarelor</course>
            <course>Proiectarea Algoritmilor</course>
            <course>Filozofie Cognitivista</course>
          </semester>
          <semester id="2">
            <course>Proiectare Logica</course>
```

```
            <course>Utilizarea Sistemelor de
Operare</course>
            <course>Electronica Analogica</course>
            <course>Limbaje Formale Si Automate</course>
            <course>Matematica 3</course>
            <course>Istoria Religiilor</course>
          </semester>
</year>
<year id="2">
          <semester id="3">
            <course>Structuri de date</course>
            <course>Teoria Sistemelor</course>
            <course>Calculatoare Numerice 1</course>
            <course>Electronica Digitala</course>
          </semester>
          <semester id="4">
            <course>Calculatoare Numerice 2</course>
          </semester>
</year>
<year id="3">
          <semester id="5">
            <course>Proiectare Cu
Microprocesoare</course>
          </semester>
          <semester id="6">
            <course>Arhitecturi Paralele De
Procesare</course>
          </semester>
</year>
```

## 5 Conclusion and Future Work

The application represents an easy to use method of storing and scheduling course information. Since it uses only XML files for inputs as well as for outputs, it can be easily extended to accommodate new features.

The scheduling algorithm can be further improved to generate multiple solutions for the same input data, when possible, compare them and let the user choose the most appropriate one. At the moment, the algorithm will generate a solution in every possible case because no limit has been imposed upon the maximum number of semesters. It is expected that when the user observes that the number of semesters does not comply with real-life constraints, they will readjust the input data. The application can be further developed to warn about impossible scenarios. At the current moment, the "maxyear" field in the input files has no meaning whatsoever, but it can be used to keep courses from being shifter too far away in the future.

The GUI can be extended to allow the user to dynamically rearrange courses (without any unwanted secondary effects on their dependencies) by simply dragging and dropping. Dragging and dropping should also be used to insert courses in their respective positions. Also, the GUI should be able to export the graphical representation to an image file on the disk.

## 6 References


1. Methods and Algorithms for Planning Course, UPB, 2009
2. JGraph library documentation, sourceforge.net/projects/jgraph/