Using Constraint Programming to Generate Timetables

Gonçalo Oliveira and Miguel Simões

FEUP-PLOG, Turma 2, Grupo Planeamento de Estudo_2

Abstract. It is often necessary for students to create a timetable where they specify when their focus will change from subject to subject or from group project to group project. Implementing such a technique is imperative to guarantee academic success, but it is often hard to do properly. For that reason, this paper describes how Constraint Programming can be used to generate timetables by taking into account study hours and time reserved for work done in pairs.

Keywords: Timetable · Constraint Programming · SICSTUS Prolog.

1 Introduction

The present paper concerns a project done in the Logic Programming course of the Integrated Master in Informatics Engineering and Computing course at FEUP, focusing on problem solving using constraint programming with SICSTUS Prolog's clpfd (constraint logic programming over finite domains) library.

The problem in analysis consists of taking a universe of students enrolled in a variety of modules and information about the number of study hours as well as eventual group work required by the module.

The present paper has the following structure:

- **Problem Description:** Detailed description of the optimization or decision problem in analysis.
- **Approach:** Description of the modeling of the problem as a constraint satisfaction problem.
  - **Decision Variables:** Description of decision variables and respective domains.
  - **Restrictions:** Description of rigid and flexible restrictions of the problem and respective implementation.
  - **Evaluation Function:** Description of the predicate involved in evaluating the solutions found by the solver.
  - **Search Strategy:** Description of the labeling strategy used.
- **Solution Presentation:** Description and explanation of the predicates involved in presenting the solution to the end user.
- **Results:** Demonstration of examples of application of the solver for instances of the problem with different degrees of complexity and analysis of the results.
- **Conclusions and Future Work:** Conclusions of this project, advantages and limitations of the proposed solution and possible improvements.

2 Problem Description

The timetable optimization problem is, in essence, a time slot allocation problem which, in this instance, consists of finding a time table for each student involved that reserves them time to study a certain module individually for the recommended number of hours and do group work for modules requiring it while delivering all required work on time. The main goal is to optimize the timetable, in order to maximize the free time for the each student.

3 Approach

In order to solve this optimization problem, we used a list to represent the students who will have their schedule optimized. To be easier to work, we created a small database, with the curricular units and their work time, with the students and the curricular units they are enrolled in, and the groups of work for the different subjects. The database can be updated anytime we want to work with different students and curricular units. So, if we have an input with the list ['Asdrubal', 'Bernardete', 'Cristalina', 'Demetrio' 'Eleuterio', 'Felismina'], we will optimize the schedule for these six students, which may have the curricular units they are enrolled in and their work groups inserted in the database.

3.1 Decision Variables

The solution of the problem is represented in two lists, one that indicates the list of Students inserted in the program, and other with a list of Tasks for each student, representing the time they will spend with the individual or the group work of each curricular unit. The indexes are the same for both lists (they have the same length), so the first element in the Students list has the first element in the Tasks lists has his list of tasks for the represented period. So, if the list of Students is the same as mentioned before, we will have a list of Tasks like this, [[task(X,Y-X,X,1,'Module'-'Type', task(...), ...], [...], [...], [...], [...], [...]], in which Module represents a curricular unit and Type represents if is an individual or a group work.

3.2 Constraints

As main restriction, we have that each work group should be working in their project at the same time, so we get a list of all the existent groups, and then we search for the respective tasks for each student to guarantee that the start time for that task is the same. The predicate constrain_groups will receive the Students list, the Groups list and three auxiliar lists, two with the StartTimes and other with the Tasks, as we can see in the image below. Also, the start times have a domain from 1 to 60, thinking in a way where the students have then available slots for day, and six days in a week available for study and work. The end times have a domain from 2 to 61, indicating that the when the slot 60 finishes, the work in that week is over.

3.3 Evaluation Function

The approach to solving the problem in the present paper does not involve any filtering or evaluation of the possible solutions returned by the labeling predicate.

```
constrain_groups(_, [], [], [], _).
constrain_groups(Students, [[Module, Student1, Student2]|Groups], [StartT1|Starts1], [StartT2|Starts2], Tasks) :-
    nth0(Index, Students, Student1),
    nth0(Index, Tasks, Tasks1),
    nth0(Index2, Students, Student2),
    nth0(Index2, Tasks, Tasks2),
    get_task(Tasks1, Module, StartT1),
    get_task(Tasks2, Module, StartT2),
    StartT1 #= StartT2,
    constrain_groups(Students, Groups, Starts1, Starts2, Tasks).
```

Fig. 1. Group Constraint Function

3.4 Search Strategy

Our approach to solving the problem involves a search strategy that minimizes the the latest time a task can end (MaxValue), so as to guarantee that all tasks are completed in the shortest amount of time. In order to achieve this the labeling predicate looks like the following:

```
labeling([minimize(MaxValue)], AllStartTimes).
```

4 Solution Presentation

The solution to the problem is printed as text to the SICSTUS Prolog CLI. It describes with detail the timetable for each of the students present in the knowledge base, including time reserved for individual study and group work as well as the slots reserved for such activities. The following is an example of an output using a knowledge base with two students:

```
John Doe
LTW - Individual: slots 1 to 4
LTW - Group work: slots 4 to 10
ESOF - Individual: slots 10 to 13
ESOF - Group work: slots 13 to 21
PLOG - Individual: slots 21 to 25
PLOG - Group work: slots 25 to 30
RCOM - Individual: slots 30 to 33
RCOM - Group work: slots 33 to 36
LAIG - Individual: slots 36 to 37
LAIG - Group work: slots 37 to 44

Jane Doe
ESOF - Individual: slots 1 to 4
ESOF - Group work: slots 13 to 21
PLOG - Individual: slots 4 to 8
PLOG - Group work: slots 8 to 13
LTW - Individual: slots 21 to 24
LTW - Group work: slots 24 to 30
RCOM - Individual: slots 30 to 33
RCOM - Group work: slots 33 to 36
LAIG - Individual: slots 36 to 37
LAIG - Group work: slots 37 to 44
```

The main predicate involved in displaying this output is write_schedules/2 which takes a list of students and tasks generated by the labeling process. This predicate is responsible for writing each of the student's name and passing the list of tasks to the write_tasks/1 predicate which is responsible for displaying them. Each of this algorithms recursively iterate lists passed as arguments.

```
write_schedules([], []).
write_schedules([Student|Students], [Task|Tasks]):-
    write(Student), nl,
    write_tasks(Task), nl,
    write_schedules(Students, Tasks).

write_tasks([]).
write_tasks([task(StartT, _, EndT, _, Module-'Group')|Tasks]):-
```

write(Module), **write**('␣−␣Group␣work:␣slots␣'), **write**(StartT), **write**('␣to␣'), **write**(EndT), **nl**, write_tasks(Tasks).

write_tasks([task(StartT, _, EndT, _, Module−'Individual')|Tasks]):−
write(Module), **write**('␣−␣Individual:␣slots␣'), **write**(StartT), **write**('␣to␣'), **write**(EndT), **nl**, write_tasks(Tasks).

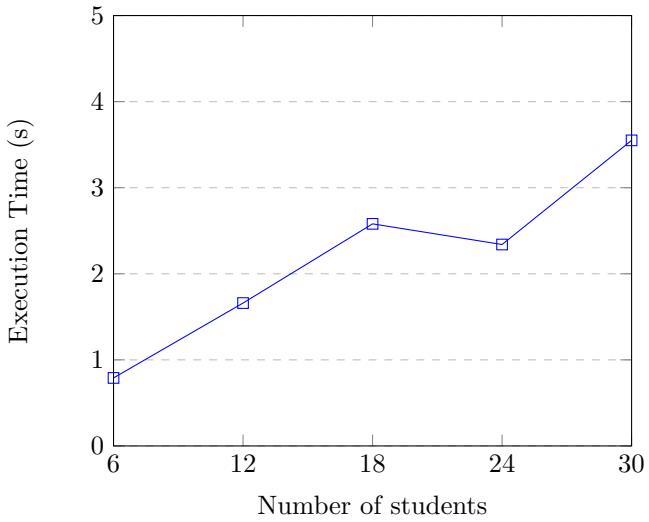
5 Results

In order to analyze the efficiency of the developed solver, we ran a series of tests with knowledge bases of varying sizes. The results of such tests showed that the execution time sharply increases with an increase in in the number of students, which also implies an increase in the number of groups. The number of modules stayed constant at 5 per student.

Table 1. Growth of execution time with an increase in number of students and groups

Number of students	Number of groups	Execution Time (seconds)
6	15	0.79
12	30	1.66
18	45	2.58
24	60	2.34
36	75	3.55

Growth of execution time with an increase in number of students and groups



These results show that the execution time of the solver grows in a quasi-linear way, dropping when the the number of students is equal to 24 and resuming afterwards.

6 Conclusions and Future Work

The main goal of this project was to apply SICSTUS Prolog's constraint programming library over finite domains, learned in the Logic Programming (PLOG) classes, in order to solve a decision or a optimization problem.

As we were developing this project, the big obstacle we had, was the available time, and that was critical to us, because with some more days we could have done a better solution to this problem.

After an analysis to the material of this curricular unit, we decide that the best predicate to use in the case was 'cumulative', because of the possibility of using the time, which is essential to create a schedule, and associate it with tasks. However, some parts of our program are not perfect, for example the free time between tasks, that could be used to work in part of the other tasks. Besides, the library in use revealed itself as very effective in handling a variety of sizes of knowledge bases and complexity of the network of modules, groups and students.

To conclude, the project was well developed for the time we had, helping us understand the subject contents in a better way, even with the management mistakes we had, we believe we did what was supposed to achieve the goals of this project.

7 Annex

7.1 Source Code

```
:− use_module(library(lists)).
:− use_module(library(clpfd)).

get_type(1, 'Individual').
get_type(2, 'Group').

make_tasks([], [], [], []).
make_tasks([[Module, Type]|T], [Start|Starts], [End|Ends], [Task|Tasks]) :−
    study_hours(Module, Type, Time),
    get_type(Type, TypeString),
    Task =.. [task, Start, Time, End, 1, Module−TypeString],
    make_tasks(T, Starts, Ends, Tasks).

all_tasks([], [], [], []).
all_tasks([Student|Students], [Task|Tasks], [Start|Starts], [End|Ends]) :−
    %find all modules where the student is enrolled in
    findall([Module, Type], (enrolled_in(Student, Module), study_hours(Module, Type, _)), Modules),
    length(Modules, Length),

    %create lists for start and end times with as many
    %elements as modules the student is enrolled in
    length(Start, Length),
    length(End, Length),
    domain(Start, 1, 60),
    domain(End, 2, 61),

    %generate list of tasks to use in the cumulative predicate
    make_tasks(Modules, Start, End, Task),
    all_tasks(Students, Tasks, Starts, Ends).

get_task([task(StartT, _, _, _, Module−'Group')|Tasks], Module, StartT).

get_task([Task|Tasks], Module, StartT):−
    get_task(Tasks, Module, StartT).

constrain_groups(_, [], [], [], _).
constrain_groups(Students, [[Module, Student1, Student2]|Groups],
                 [StartT1|Starts1], [StartT2|Starts2], Tasks) :−
    nth0(Index, Students, Student1),
    nth0(Index, Tasks, Tasks1),
    nth0(Index2, Students, Student2),
    nth0(Index2, Tasks, Tasks2),
```

```
    get_task(Tasks1, Module, StartT1),
    get_task(Tasks2, Module, StartT2),
    StartT1 #= StartT2,
    constrain_groups(Students, Groups, Starts1, Starts2, Tasks).

group_time(Students, Tasks):-
    findall([Module, Student1, Student2], group(Module, [Student1, Student2]), GroupList),
    length(GroupList, Length),
    length(Starts1, Length),
    length(Starts2, Length),
    constrain_groups(Students, GroupList, Starts1, Starts2, Tasks).

acc_constraints([], [], _, AllStartTimes, MaxValues):-
    maximum(MaxValue, MaxValues),
    labeling([minimize(MaxValue)], AllStartTimes).

acc_constraints([StartTimes|StartTimess], [EndTimes|EndTimess], [Task|Tasks], AllStartTimes, MaxValue) :-
    append(StartTimes, AllStartTimes, StartTimes2),
    append(EndTimes, MaxValue, MaxValues),
    cumulative(Task),
    acc_constraints(StartTimess, EndTimess, Tasks, StartTimes2, MaxValues).

write_schedules([], []).
write_schedules([Student|Students], [Task|Tasks]):-
    write(Student), nl,
    write_tasks(Task), nl,
    write_schedules(Students, Tasks).

write_tasks([]).
write_tasks([task(StartT, _, EndT, _, Module-'Group')|Tasks]):-
    write(Module), write(' - Group work: slots '), write(StartT), write(' to '), write(EndT), nl,
    write_tasks(Tasks).

write_tasks([task(StartT, _, EndT, _, Module-'Individual')|Tasks]):-
    write(Module), write(' - Individual: slots '), write(StartT), write(' to '), write(EndT), nl,
    write_tasks(Tasks).

study(Students):-
    all_tasks(Students, Tasks, StartTimes, EndTimes),
    group_time(Students, Tasks),
    acc_constraints(StartTimes, EndTimes, Tasks, [], []), nl,
    write('Slots 1 to 10 represent day 1, 11 to 20 day 2, and so on!'), nl, nl,
    write_schedules(Students, Tasks).
```