Declarative Programming project: Exam timetabling

Report

Arno Moonens
Studentnr.: 500513
arno.moonens@vub.ac.be

1. Introduction

For this project, it was necessary to find a schedule for a set of exams, students, lecturers, etc. such that there are no conflicts when they take place in a room at on a certain day and at a certain time. Students and lecturers also have preferences. If these are violated, we get a certain penalty. Using these, we can calculate the cost of a schedule. It is also the aim to find the optimal schedule and to find the best one using heuristics. Finally, print predicates allow us to see the schedule in a more readable and ordered way.
In this report, we will first discuss how each part of the implementation of my project was implemented.

2. Implementation

2.1. Preprocessing

Before certain things can be calculated, preprocessing is executed to reduce the number of calculations. The things that we preprocessed are then asserted for easy retrieval.
The first thing that is asserted is for each exam the list of students that take that exam. These are simply the students that follow the course of that specific exam.
We also assert combinations of exams that collide if their day, start and end time overlap. If those exams are taught by the same lecturer of some students have both exams, we say that there is a possible collision and assert the combination of exam ID's.

2.2. Validity

Implements `is_valid(?S)`.

One predicate is used to both check the validity of a schedule and generate a valid schedule. Both components are explained below.

2.2.1. Checking the validity schedule

To see if a schedule is valid, we first find all the exam ID's of the exams for which we need to find a good room, day and start hour (from now on called an event) and put them in a list. After this, we check if the exam of the first event in our schedule is included in the previously mentioned list and delete it. We then continue with checking the validity of the rest of the events using the remaining list. Thus, we see that we start at the end of the list to check for validity. An empty schedule for which the list with exam ID's is also empty is always a correct schedule.
When other events are checked, we see if the room in which the exam is held is big enough for the number of students that take the exam. After this, we see if the exam takes place during the exam period and that an exam can takes place in that room during the exam's duration on the specific day.
When this is checked, we check if there isn't a conflict with an event that we already checked. For every event that we we already checked, we see that the day and time of that event and the event we currently handle don't overlap or that they take place in a different room and there isn't a possible collision (which was asserted in the preprocessing phase, see 2.1). If there isn't a collision, we say that the event is valid.
A schedule is valid when all the events in the schedule are valid.

2.2.2. Generating a valid schedule

As was already explained, the same predicate can be used to generate valid schedules.
We start again at the end of the list and gradually add an event. Thus, the variables in our current event are unbound at first but get bound when searching for a room, day and start hour. After this we check again if there isn't a conflict with the other events that were already generated. If there is a conflict, other rooms, days and start hours will be tried using backtracking until the event is valid.
The schedule is valid when the first event in the list is generated and is valid.

2.3. Cost

Implements: `cost(+S,-C)` and `violates_sc(+S,-SC)`.

To calculate both the cost of a schedule (using the predicate `cost(+S,-C)`) and the soft constraints that were violated (using the predicate `violates_sc(+S,-SC)`), we use the same predicate: `violated_sc_cost(+S, -Violated, -Cost)`. When calculating the cost or violated constraints, we only use that variable from this predicate. For the cost predicate, the `Cost` variable can be already instantiated

or not instantiated. If the cost was already instantiated, the predicate will succeed if the calculated cost can be unified (i.e. is the same) as the cost given.
We now explain how the `violated_sc_cost` predicate works. Note that when we talk about "applying a penalty", we mean that the penalty will be added to the total cost and the associated constraint that was violated (in the same form as listed in the project description) will be added to the list of violated constraints

First preprocessing takes place, which was explained in Section 2.1. After that, we put all the students in a list and calculate their cost and violated constraints. After that, we do the same for the lecturers.
To calculate the cost of students, we handle each student separately. Before we loop over the student, we sort the events in ascending order, which is necessary to compute study time penalties. We first calculate if the student has exams on the same day and apply a penalty and add a violated constrained if necessary. To calculate the study penalty, we remember each the the previous exam that the student had, the day that it took place and how many days to learn are currently available. With each event, this number is increased by the difference between the current day and the day of the student's last exam (0 at first). We then calculate the difference between the available study days and the number of days that are needed to learn for the exam. If this number is negative, we need to multiply the absolute of that number by the penalty of `sc_study_penalty(SID, Penalty)`. If this number is positive, we still have days available and pass that number so it can be used for the next event.

In this predicate, we also calculate the lunch break penalty. This penalty takes place when their is a collision in hours between those of the event and 12 and 13 hours. The back-to-back (*b2b*) is also calculated here. Remember that we always keep track of the previous event that we handled. If this event takes place at the same day as the current event and the start hour of one of the events is the same as the other's end hour, we apply a penalty. The last penalty that is calculated is the penalty for when the event takes place in a certain period, as defined in our problem instance. If such penalty has been defined for that student and course and the day and time it takes place collides with the current event, we apply the penalty.

We also handle each lecturer separately when handling the cost of the lecturers. On beforehand, the events were also sorted, but in the opposite order. We do this because, in contrary to the calculation of study time, we need days after the exam took place instead of before. By reversing the order of events, the calculation of correction time is more or less the same as the one for study time.
Like with the students calculation, we also calculate the back-to-back penalty, lunch break penalty and the penalty for when a person has a certain exam on day and period defined in our problem instance.
Apart from these, lecturers can also have a penalty for when any exam of which

the lecturer teaches the course has takes place in certain days and periods.

2.4. Optimal

Implements: `find_optimal(-S)`, `is_optimal(?S)`.

We start with `is_optimal(?S)`. Here, all valid schedules (generated using `is_valid(?S)`) and the associated costs (calculated using `cost(+S,-C)`) are put in a list. After that, we go over this list to find all schedules with the minimum cost.

At each moment, we have both the list of schedule-cost pairs that we loop over and a list of schedule-cost pairs of the schedules with currently the lowest cost. If a schedule we encounter has a lower cost than the one in the list, we continue with a new list of best schedules, only containing the schedule and cost of the new best schedule. If the cost was the same, we add the schedule and cost to the list and continue with this new list. Else, we just go to the next schedule-cost pair in the list of generated schedules and their costs.

When this is done, we use the `member` predicate to check if the schedule given is one with the lowest cost, or to unify `S` with a schedule that has the lowest cost. Using backtracking, all schedules with the lowest cost can be obtained.

For `find_optimal(-S)`, we simply call `is_optimal(?S)` and use a cut afterwards so only 1 of the best schedules is returned.

2.5. Heuristical

Implements: `find_heuristically(-S)` and `find_heuristically(-S,+T)`.

In my implementation, `find_heuristically(-S)` just calls `find_heuristically(-S,+T)` with `T=100`.

The predicate `find_heuristically(-S,+T)` starts by generating a valid schedule and copying it along with its cost 3 times into a list. Then, for each schedule in this list, a new (still valid) schedule is made and is added to the list along with its cost. This new schedule is made by picking a random event out of the schedule, possibly changing the room and changing the day, start hour or both. We also check that there isn't a conflict with the other events left in the schedule. When this is done, we get a list with double the amount of schedules that we had before.

After this, we first filter this list by keeping 3 schedules that have the lowest cost. Then we also add one random schedule from the schedules that wasn't good enough. We then repeat the process and make new schedules again. Because we add one random schedule to the list, which is also mutated, it is possible that all schedules are seen after some time. If the mutated schedule of the previous randomly chosen schedule is (randomly) chosen itself, it is possible that all possible schedules will be generated. As a result, the solution can be probabilistically approximately correct and it is possible to get the optimal schedule.

This process is repeated until it has ran for `T` seconds. After this, the schedule with the lowest cost in the list is unified with `S`.

2.6. Printing

Implements: `pretty_print(+S)` and `pretty_print(+SID, +S)`.

To print a schedule using `pretty_print(+S)`, we use a `findall` to backtrack over every possible result of the `print_day` predicate. In this predicate, `setof` is used to group the events by day and sort these days. As such, we can print the events for each day. For each day, a `findall` is again used to print the events of each room on a certain day. For each room, the name of the room is printed. Afterwards, the associated events are sorted and printed with their start and end hour, name and lecturer.

For `pretty_print(+SID, +S)`, we filter the events in the schedule to only keep the events of the exams that the student with ID `SID` has. Afterwards, `pretty_print(+S)` is called with a schedule of this filtered list of events.

3. Strengths and weaknesses

For the small instance, all required predicates were tested, worked successfully and finished in less than 2 minutes. For the other 2 instances, only `find_optimal(-S)` wasn't tested, because of it didn't finish in a reasonably time. All other predicates worked as well for these instances and finished on time. For the predicate `cost(+S,-C)`, all soft constraints were taken into account to calculate the cost of a schedule. Because in my implementation of `find_heuristically(-S,+T)` predicates that unify variables in a random way were used, the resulting schedule isn't always the same and thus the best found cost may also vary.
Every predicate that was part of the extensions was implemented and was explained in Section 2. These predicates are:

- `is_valid(?S)`
- `violates_sc(+S,-SC)`
- `is_optimal(?S)`
- `find_heuristically(-S,+T)`
- `pretty_print(+SID,+S)`

Because Prolog modules were used, different parts of the implementation can be found in different modules and files. 5 modules and files were made: *printing*, *valid*, *optimal*, *heuristically* and *cost*. A 6th file is used to load the modules and problem instance.
The code was also written to work for all problem instances and has predicates that are used in different parts of the program in order to avoid duplication. The code was written in declarative style as much as possible. It was sometimes necessary to avoid backtracking and to avoid doing a lot of unnecessary work an thus improve performance.

4. Experimental results

4.1. Small instance

For the small instance, 2 schedules were found (using `is_optimal(-S)`) that had both the lowest cost of 1.875. Those schedules are:

- `schedule([event(e1, r2, 2, 10), event(e2, r2, 5, 10), event(e3, r1, 4, 10), event(e4, r2, 4, 10), event(e5, r2, 3, 13)])`
- `schedule([event(e1, r2, 2, 10), event(e2, r2, 5, 10), event(e3, r2, 4, 10), event(e4, r1, 4, 10), event(e5, r2, 3, 13)])`

4.2. Large instances

For the large instance with a long exam period, a schedule with a cost of 0.42710526315789477 was found using `find_heuristically(-S)`. This schedule is:

```
schedule([event(e1, r3, 14, 14), event(e20, r2, 17, 12), event(e27, r1, 10, 15),
event(e11, r2, 18, 14), event(e9, r2, 7, 15), event(e12, r2, 5, 15),
event(e25, r1, 19, 9), event(e33, r2, 12, 9), event(e5, r3, 21, 9),
event(e8, r3, 12, 13), event(e16, r1, 5, 10), event(e34, r2, 11, 14),
event(e14, r1, 18, 15), event(e28, r3, 11, 14), event(e7, r2, 20, 10),
event(e4, r2, 7, 13), event(e22, r2, 20, 13), event(e31, r2, 17, 10),
event(e26, r1, 4, 10), event(e6, r2, 10, 9), event(e17, r1, 3, 9),
event(e30, r2, 5, 13), event(e23, r2, 11, 10), event(e24, r1, 12, 15),
event(e21, r2, 13, 10), event(e2, r2, 14, 9), event(e29, r3, 18, 10),
event(e15, r1, 17, 15), event(e18, r3, 13, 14), event(e13, r3, 17, 10),
event(e3, r2, 14, 13), event(e19, r3, 19, 13), event(e10, r3, 20, 13),
event(e32, r2, 3, 13)]).
```

For the large instance with a short exam period, a schedule with a cost of 17.567894736842106 was found. This schedule is:

```
schedule([event(e7, r1, 5, 10), event(e4, r1, 5, 13), event(e27, r1, 6, 11),
event(e32, r1, 5, 15), event(e6, r1, 6, 15), event(e2, r2, 7, 14),
event(e23, r1, 7, 9), event(e13, r1, 6, 9), event(e18, r3, 3, 14),
event(e5, r1, 6, 13), event(e19, r3, 6, 9), event(e15, r2, 7, 10),
event(e1, r1, 7, 13), event(e17, r1, 4, 10), event(e12, r2, 4, 13),
event(e10, r1, 4, 13), event(e22, r1, 3, 10), event(e14, r1, 3, 15),
event(e24, r2, 5, 9), event(e20, r3, 5, 13), event(e3, r3, 7, 13),
event(e26, r2, 6, 9), event(e16, r1, 4, 15), event(e28, r2, 5, 15),
event(e9, r2, 5, 13), event(e11, r2, 3, 13), event(e8, r3, 4, 9),
event(e21, r2, 4, 15), event(e25, r1, 3, 13), event(e29, r3, 3, 9),
event(e30, r2, 4, 9), event(e31, r2, 3, 15), event(e33, r2, 3, 11),
event(e34, r2, 3, 9)]).
```

5. Conclusion

We started by seeing how my solution of the exam timetabling was implemented. We then looked at the strengths an weaknesses of my implementation and ended with experimental results, which included schedules with the lowest cost that we found and with the cost itself.