

Declarative Programming Report

Bruno Rocha Pereira (0529512)

Vrije Universiteit Brussel
Master in Applied Science and Engineering: Computer Science
August 10, 2016

1 Description of the different approaches

1.1 *is_valid(+S)*

The *is_valid/1* function generates a list of all the exams and then checks if:

1. The events are indepently valid.
2. The events are valid with each other and do not create any kind of overlapping (room, time and participants). It also checks that exams take place once and only once.

1.2 *cost(+S, ?Cost)*

The *cost/2* predicate makes use of the *violates_sc/2* described below to create a list of all the violated constraints. It then iterates over that list to add the different penalties for students and teachers to apply the final cost formula.

1.3 *find_optimal(-S)*

The *find_optimal/1* predicate makes use of the *is_optimal/1* described below and cuts after finding one optimal schedule.

1.4 *find_heuristically(-S)*

The *find_heuristically/1* predicate generates a list of all the possible events and adds one by one the best event (i.e. the one that, once added, will give the lowest cost). Once all the exams are added, the validity is checked. If the current schedule is valid, we are done with technically the best schedule. If it isn't, we repair it with *correct_schedule/5*. The repair is done by changing the events one by one until reaching a valid schedule.

1.5 *pretty_print(+S)*

The first step to *pretty_print/1* is to sort the schedule. This is done by associating a value to each event to insure that they are sorted by day, then by room and then by start time. The events are printed one by one, with the previous day and the previous room passed as argument to know if the next event is on another day or in another room.

Extensions

1.6 *is_valid(-S)*

The *is_valid(-S)* predicate follows almost the same logic as *is_valid(+S)* except some data need to be instantiated during the process.

1.7 *violates_sc(+S, -SC)*

The *violates_sc(+S, -SC)* predicate splits all the soft constraints in 2 groups: the simple ones(for individual events) and the complex ones(between different events). The simple constraint violations checker iterates over the list of events and finds all the constraints violations to add them to a list. The complex constraint violations checker splits the work. A predicate was written for every *sc* and then *findall* was used to get all the violations that happens. All the different lists are then merged into one big list. *violates_sc/2* can be used with *+SC* as well, but the user must be careful as *sc_dame_day(S1, E4,E1, 2)* can be found instead of *sc_same_day(S1, E1, E4, 2)* for example.

1.8 *is_optimal(?S)*

The *is_optimal/1* predicate makes use of the *is_valid/1* and the *cost/2* predicates to generate all the valid schedules along with their cost. It then sorts them by cost and takes all the schedules with the cost of the first one (technically, the lowest cost).

1.9 *find_heuristically(-S, +T)*

This predicate makes use of the simple *find_heuristically/1* except it adds a time argument. One more clause was added on the top (to be the first executed) that checks every time if the time limit is reached.

1.10 *pretty_print(+SID, +S)*

This predicate works almost as the simple *pretty_print/1* except it filters all the event for the student *SID*.

2 Strenghts and weaknesses

2.1 Working predicates

The working predicates for all instances in under 2 minutes are:

- *is_valid*(+S)
- *cost*(+S, ?Cost)
- *find_optimal*(-S) - not tested for the largest.
- *find_heuristically*(+S)- not tested for the largest.
- *pretty_print*(+S)

2.2 Soft constraints taken into account

For all the tests, all the constraints were taken into account.

2.3 Quality of heuristic

For the heuristical search, events are added one by one to a temporary schedule and this schedule is checked everytime to be currently valid. During the implementation, it was noticed that it blocked around 18 exams inserted. This is due to the fact that it is difficult/impossible to have a valid schedule from this point. That's why the schedule correcter was added. This allowed the program to make invalid schedule and repairing them to make them valid in a second time. The time taken by the *find_heuristically* could be possibly improved by well-placed cuts. The results of the *find_heuristically* for all the instances provided can be found hereinbelow.

2.4 Extensions implemented

- *is_valid*(-S)
- *violates_sc*(+S, -SC)
- *is_optimal*(?S) - not tested for the largest.
- *find_heuristically*(-S, +T) - not tested for the largest.
- *pretty_print*(+SID, +S)

All the possible extensions were implemented

2.5 Non-functional requirements

2.5.1 Test in computer rooms

Unfortunately, this program couldn't be tested on the computer rooms since they were closed. To my knowledge, swipl v6 is running on them. The only problem is see that could happen is with sort/4, that was introduced in the v7. This could be easily transformed to a self-made sort/4 to work with the v6.

2.5.2 Generality

The implementation doesn't rely on any property of any instance. Therefore, it should be as general as possible.

2.5.3 Procedural style

Cuts were used in the program. Asserts and if-statements were avoided in order to make use of the declarative features of Prolog. The program was written as much as possible in a declarative style using cuts to avoid unnecessary backtracking.

2.5.4 Efficiency

As described 2.5.1, this program wasn't tested in the computer rooms. Although, all the different predicates were run on my laptop and all the required ones ere able to do so in less than 2 minutes.

3 Experimental Results

3.1 Small instance - find_optimal(S)

The time taken by my scheduler to find an optimal schedule for the small instance is 3.745 seconds.

```
?- time(find_optimal(X)).
% 36,484,643 inferences, 3.745 CPU in 3.745 seconds (100% CPU, 9742980 Lips)
X = 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)]).
```

3.2 Large instance - find_heuristically

The code below represent the query run on the different instances.

3.2.1 Small Instance

Finding a solution heuristically gives us a cost of 2.75.

```
?- find_heuristically(X),cost(schedule(X),Cost).
X = [event(e1, r2, 3, 10), event(e2, r2, 4, 10), event(e3, r1, 5, 10),
event(e4, r2, 5, 10), event(e5, r2, 1, 10)],
Cost = 2.75.
```

3.2.2 Large Instance Short Session

Finding a solution heuristically for the large instance and the short session time gives us a cost of 17.538947368421052.

```
?- find_heuristically(X),cost(schedule(X),C).
X = [event(e1, r3, 4, 9), event(e2, r2, 4, 9), event(e3, r2, 6, 10),
event(e4, r2, 6, 14), event(e5, r3, 7, 13), event(e6, r2, 7, 9),
event(e7, r2, 7, 13), event(e8, r3, 5, 9), event(e9, r2, 3, 9),
event(e10, r3, 3, 9), event(e11, r2, 4, 13), event(e12, r2, 5, 9),
event(e13, r3, 7, 15), event(e14, r1, 5, 13), event(e15, r1, 6, 9),
event(e16, r1, 3, 13), event(e17, r1, 4, 13), event(e18, r3, 6, 9),
event(e19, r3, 7, 9), event(e20, r2, 5, 11), event(e21, r2, 3, 13),
event(e22, r3, 6, 13), event(e23, r2, 4, 15),
event(e24, r1, 3, 9), event(e25, r1, 7, 9), event(e26, r1, 3, 15),
event(e27, r1, 5, 9), event(e28, r3, 3, 13), event(e29, r3, 4, 13),
event(e30, r2, 3, 15), event(e31, r3, 4, 15), event(e32, r2, 5, 15),
event(e33, r3, 6, 15), event(e34, r3, 3, 15)],
C = 17.538947368421052.
```

3.2.3 Large Instance Long Session

Finding a solution heuristically for the large instance and the long session time gives us a cost of 0.2631578947368421.

```
?- find_heuristically(X),cost(schedule(X),C).
X = [event(e1, r3, 4, 9), event(e2, r2, 4, 9), event(e3, r2, 6, 10),
event(e4, r2, 7, 9), event(e5, r3, 10, 9), event(e6, r2, 6, 14),
event(e7, r2, 10, 9), event(e8, r3, 10, 13), event(e9, r2, 11, 9),
event(e10, r3, 11, 9), event(e11, r2, 12, 9), event(e12, r2, 14, 9),
event(e13, r3, 12, 9), event(e14, r1, 6, 9), event(e15, r1, 12, 9),
event(e16, r1, 14, 9), event(e17, r1, 17, 9), event(e18, r3, 17, 9),
event(e19, r3, 19, 13), event(e20, r2, 11, 11), event(e21, r2, 13, 9),
event(e22, r2, 17, 13), event(e23, r2, 18, 9), event(e24, r1, 19, 9),
event(e25, r1, 19, 13), event(e26, r1, 19, 15), event(e27, r1, 20, 9),
event(e28, r3, 14, 9), event(e29, r3, 19, 9), event(e30, r2, 11, 15),
event(e31, r2, 12, 13), event(e32, r2, 14, 13), event(e33, r2, 17, 9),
event(e34, r2, 18, 13)],
C = 0.2631578947368421.
```