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)

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.

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 instanciated 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.

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 eecuted) 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

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 is 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.

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)]).
```