GENERAL INFO

Home

Members

News

Contact

RESEARCH

Publications

Topics

Projects

Software

FOR STUDENTS

Courses

Thesis Proposals 2015-2016

Bachelor projects

LINKS

Evolutionary Linguistics

Declarative Programming Project: Exam Timetabling

The project presentation can be downloaded here: project presentation

For the project you'll have to implement a solver for the Exam Timetabling problem using SWI-Prolog.

Introduction

The Exam Timetabling Problem

The examination timetabling problem can be stated as follows: Where and when to schedule which exams, during the examination period? Making a good schedule is challenging due to the large number of students enrolled, often with individual programs, lecturers teaching multiple courses and the limited number and capacity of rooms. Furthermore, both lecturers and students have (often conflicting) preferences. E.g. While lecturers want sufficient correction time, students want more time to study. It therefore represents a major administrative activity for academic institutions. Partial automation of this task is an active research area.

Problem Instances

To test your scheduler, problem instances are provided. Each defines a set of courses, their exams, lecturers teaching and students following them, as well as a set of rooms and their capacity/availabilities.

Each specifies the following knowledge:

Fluid Construction Grammar
Robotics wiki
IB2 - Bioinformatics Institute
WWCS 2015
DynaMine

- A set of students:
student(SID,Name): A student with unique identifier 'SID'
and name 'Name'.

- A set of lecturers:
lecturer(LID,Name): A lecturer with unique identifier 'LID'
and name 'Name'.

- A set of courses:
course(CID,Name): A course with unique identifier 'CID'
and name 'Name'.

- A set of exams:
exam(EID,Name): An exam with unique identifier 'EID' and name 'Name'.

- A set of rooms:
room(RID,Name): A room with unique identifier 'RID' and name 'Name'.

- Which course has which exams:
has_exam(CID,EID): The course with 'CID' has the exam with 'EID'.
(Note: Each exam is related to exactly 1 course)

- Duration of each exam:
duration(EID,Duration): The exam with 'EID' takes 'Duration' hours.

- Students following a course:
follows(SID,CID): The student with 'SID' follows
the course with 'CID'.

- Which lecturer teaches which courses:
teaches(LID,CID): The lecturer with 'LID' teaches the course with 'CID'.

(Note: Each course is taught by exactly 1 lecturer)

- The capacity of rooms:
capacity(RID,Capacity): The room with 'RID' can facilitate at most 'Capacity' students.

- The first day of the study/exam period:
first_day(FirstDay).

- The last day of the correction/exam period:
last_day(LastDay).

- The availabilities of rooms:
availability(RID,Day,From,Till): The room with 'RID' is available day 'Day' from 'From' o'clock till 'Till' o'clock.

- A set of soft constraints:
See Section 'Soft Constraints'.

- The correction time required for each exam:
sc_correction_time(EID,Days): 'Days' days are required to correct the exam with 'EID'.
(Note: The day of the exam excluded)

- The study time required for each exam:
sc_study_time(EID,Days): 'Days' days are required to study for the exam with 'EID'.
(Note: The day of the exam excluded)

The following table gives an overview of the instances provided:

#	name	# students	# lecturers	# courses	# rooms	exam period length	optimal sq
1	small	4	4	5	2	5 Days	1.875
2	large_short	100	19	34	3	9 Days	???
3	large_long	100	19	34	3	23 Days	???

Hard Constraints

The following constraints must be met in order for a schedule to be admissable:

- All exams must be scheduled exactly once and start at the hour (e.g. 15:00 but not 15:30).
- Exams can only take place in a room
  - that is available for the entire period of the exam (start to end).
  - whose capacity exceeds or equals the number of students attending the exam (subscribed to the course).
- No 2 exams can take place at the same time...
  - in the same room
  - if 1 or more students are subscribed to both courses (this includes multiple exams of the same course).
  - if the same lecturer teaches both courses.

Soft Constraints

Soft constraints, unlike hard constraints, are not required for the schedule to be admissable. Rather, they are desirable (for lecturers, students or both). Each soft-constraint has a corresponding penalty, which is imposed when it is not met. One schedule is better than another if it has a lower sum of penalties.

For the project we consider the following soft constraints:

- **`sc_lunch_break(PID,Penalty)`**: A penalty of 'Penalty' is imposed for each exam a Lecturer/Student with 'PID' has during lunch break (i.e. from 12 till 13 o'clock).
- **`sc_no_exam_in_period(LID,Day,From,Till,Penalty)`**: A penalty of 'Penalty' is imposed for each exam Lecturer with 'LID' has on day 'Day', (partially) in the period from 'From' o'clock till 'Till' o'clock.

- **`sc_not_in_period(PID,EID,Day,From,Till,Penalty)`**: A penalty of 'Penalty' is imposed if the exam with 'EID' is held on day 'Day', (partially) in the period from 'From' o'clock till 'Till' o'clock. It is a constraint of Person with 'PID'.
- **`sc_same_day(PID,Penalty)`**: A penalty of 'Penalty' is imposed for each pair of exams the person with 'PID' has on the same day. E.g. if the person has 3 exams on one day, 'Penalty' is imposed 3 times.
- **`sc_b2b(PID,Penalty)`**: A penalty of 'Penalty' is imposed for each pair of exams the person with 'PID' has back-to-back (same day and consecutively). E.g. if the person has 3 exams consecutively, 'Penalty' is imposed 2 times.
- **`sc_correction_penalty(LID,Penalty)`**: A penalty of 'Penalty' is imposed for each day the lecturer with 'LID' has too little to correct all his exams. Note that a lecturer can correct exams, the same day he has another exam (just not the exam itself). E.g. assume a lecturer has 2 exams in a 1-5 day exam period, each requiring 2 days to correct. If exams are held day X and Y, Z times 'Penalty' is imposed.

X	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5
Y	1	2	3	4	5	2	3	4	5	3	4	5	4	5	5
Z	0	0	0	1	2	1	1	1	2	2	2	2	3	3	4

- **`sc_study_penalty(SID,Penalty)`**: A penalty of 'Penalty' is imposed for each day the student with 'SID' has too little to study for all his exams. Note that a student can study for exams, the same day he has another exam (just for the exam itself). E.g. assume a student has 2 exams in a 1-5 day exam period, each requiring 2 days to study for. If exams are held day X and Y, Z times 'Penalty' is imposed.

X	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5
Y	1	2	3	4	5	2	3	4	5	3	4	5	4	5	5
Z	4	3	2	2	2	3	2	1	1	2	1	0	1	0	0

Functional Requirements

Write a Prolog program to solve the Exam Timetabling problem.

- An exam schedule must be represented as a `schedule/1` functor, containing a list of `event/4` functors, one for each exam. An `event(EID,RID,Day,Start)` functor indicates that the exam with 'EID' will start at 'Start' o'clock, in the room with 'RID', at day 'Day'.

  E.g. Given the following knowledge

  ```
  exam(e1,'history').
  exam(e2,'math').
  room(r1,'2D006')
  room(r2,'10G711').
  duration(e1,2).
  duration(e2,3).
  ```

  A schedule where the history and math exam are held concurrently at day 1 at 10 o'clock, in rooms '2D006' and '10G711' respectively, can be represented as follows.

  ```
  schedule([event(e1,r1,1,10),event(e2,r2,1,10)])
  ```

  Note that since the math exam takes 1 hour longer, they end at 12 and 13 o'clock respectively.

  (also see example in slides)

- You must implement the following predicates (accounts for 18 points out of 20):

  `is_valid(+S)`: Checks whether exam schedule S is valid, i.e. satisfies all hard constraints. Note that exam events may appear in any order (i.e. all permutations are valid).

  `cost(+S,?Cost)`: Where Cost is the cost of valid schedule S.

  The cost of a schedule is determined as the total sum of penalties of violated soft-constraints. As students typically greatly outnumber lecturers, penalties are normalized as follows:

  ```
  (penalties_lecturers/#lecturers +
  penalties_students/#students)/2
  ```

  Here penalties_lecturers/students is the sum of penalties for constraints of

all lecturers/students resp. The first argument of a soft constraint predicate indicates the 'PID' of the person (lecturer/student) having this constraint.

`find_optimal(-S)`: Finds an exam schedule S, exactly minimizing cost.

`find_heuristically(-S)`: Finds an exam schedule S, approximately minimizing cost. Note that the cost of the exam schedule found, will be taken into account when grading.

`pretty_print(+S)`: Outputs an exam schedule nicely, in human readable format. For each day, exams must be listed per room, in chronological order (example see slides)

- Possible extensions (up to 3 bonus points):

  `is_valid(?S)`: is_valid/1 next to checking, also generates all valid exam schedules (without literal duplicates) in under 2 minutes. To achieve this, you're allowed to only generate semantically different solutions (i.e. exam events in fixed order). Make sure it still accepts any valid exam schedule (i.e. any permutation).

  `violates_sc(+S,-SC)`: Where SC is a list of soft constraints violated in valid exam schedule S. Use the following functors (corresponding to sc predicates) as elements of SC:

  - `sc_lunch_break(PID,EID,Penalty)`: The exam with 'EID' violates sc_lunch_break(PID,Penalty).
  - `sc_no_exam_in_period(LID,EID,Day,From,Till,Penalty)`: The exam with 'EID' violates sc_no_exam_in_period(LID,Day,From,Till,Penalty).
  - `sc_not_in_period(PID,EID,Day,From,Till,Penalty)`: The exam with 'EID' violates sc_not_in_period(PID,EID,Day,From,Till,Penalty).
  - `sc_same_day(PID,EID1,EID2,Penalty)`: The pair of exams with 'EID1' and 'EID2', violate the sc_same_day(`PID,Penalty`) constraint.
  - `sc_b2b(PID,EID1,EID2,Penalty)`: The pair of exams with 'EID1' and 'EID2', violate the sc_b2b(`PID,Penalty`) constraint.
  - `sc_correction_time(LID,DTL,TotalPenalty)`: The lecturer with 'LID' has 'DTL' days too little to correct all his exams. Where TotalPenalty equals DTL times Penalty of sc_correction_penalty(LID,Penalty).

- `sc_study_time(SID,DTL,TotalPenalty)`: The student with 'SID' has 'DTL' days too little to study for all his exams. Where TotalPenalty equals DTL times Penalty of sc_study_penalty(SID,Penalty).

If a penalty is missing or 0 for an individual, he doesn't have this constraint and therefore can't be violated (i.e. shouldn't appear in the list).

`is_optimal(?S)`: Checks whether exam schedule S is optimal. If S is a variable, is_optimal/1 generates all optimal schedules exactly once (in under 2 minutes for small).

`find_heuristically(-S,+T)`: Like find_heuristically/1, but terminates after T seconds. If T goes to infinity, the solution returned should be probabilistically approximately correct (PAC), i.e. find_heuristically/2 should eventually evaluate all exam schedules. Finally, find_heuristically(S,100) shoudn't take longer than 2 minutes.

`pretty_print(+SID,+S)`: Outputs an exam schedule for the student with 'SID' nicely, in human readable format. For each day, in chronological order, indicating room, start and end time for each exam.

Non-Functional Requirements

It also works for us...

The program has to be written in SWI-prolog and should run on the computers of the rooms of IG. Make sure that your source files are standard Unix text files, e.g. lines are separated by newline characters only. For those of you that make their project at home and/or with a different prolog, make sure that your code is fully functional on the target system. Your code must work when run using SWI-prolog in the computer rooms! Excuses of the “it worked at home but not on this system” type will not be accepted. ***If it doesn't work for us, it doesn't work.***

Generality

The same code should be able to solve all given problem instances. Furthermore, to verify

generality, your code will be tested on some additional instances. It therefore shouldn't rely on (or exploit) specific properties of these instances.

Procedural style

By using cuts, assert/retract and the build-in if-statement it is possible to write Java-like programs. This style makes hardly any use of the declarative features of prolog, and such programs are better written in Java. As this is a declarative programming course, the latter is strongly not recommended.

Efficiency

All predicates must run in under 2 minutes on the lab computers, for all instances (with exception of find_optimal/1 for the large instances, see final section).

You are thus to find a careful balance between declarative style, readability & efficiency:

E.g. Use red cuts if and only if they have a strong impact on efficiency.

Modularity

Work modular:

- Define your solution in a different file than the instances
- Divide largely independent logic for your solver over multiple files (e.g. 1 for each requested predicate).
- Use prolog modules instead of consult to combine the seperate parts of your solution.

Code Duplication

Closely related to modularity, code duplication is to be avoided as well. Having small reusable predicates will not only improve the quality of your code, it will also make things easier for you and help you avoid typical copy-paste related errors. E.g. Note how the soft constraints sc_lunch_break/2 and sc_no_exam_in_period/5 actually correspond to a set of sc_not_in_period/6 constraints. Also, sc_same_day/sc_b2b and correction_time/study_time are closely related.

Syntax Conformity

Make sure you use the same syntax as used in the assignment:

- For predicates (e.g. `is_valid` and not `isValid`)
- For schedules (e.g. `schedule([event(e1,r1,1,10),event(e2,r2,1,10)])` and not `[event(e1,r1,1,10),event(e2,r2,1,10)]`.

Internally you're encouraged to use your own predicates and data formats (whatever is most convenient/efficient), but make sure you at least provide the requested predicates, using the requested syntax. This primirarly because we'll use unit-tests to verify the correctness of your implementation.

Commenting

Comment your source code. Writing good comments is an art. Try therefore to follow the prolog-specific commenting guidelines given here as much as possible.

Reporting Requirements

Next to your implementation, you are to write a (brief) report containing:

- A brief description of your solution approach (design, data-structures, algorithms)
- Clearly state the strengths and weaknesses of your implementation. E.g.
  - Which predicates work? (For all instances?)
  - Do you take into account all, or only a subset of the soft constraints?
  - Quality of heuristic solutions to large instances?
  - Have you implemented any extensions?
  - Non-functional requirements?
- Experimental results:
  - For the small instance: An optimal solution found by your scheduler *
  - For the large instances: The cost of your heuristic solution *

* Note that any reported experimental results must be reproducable in under 2 minutes on the lab computers AND at the oral defense.

Deadline for 1st Session

The firm deadline for this project, in the first Session, is **Sunday, 10 Januari 2016**, at midnight. Dates for the project defenses will be announced later**.**

You should send me the following deliverables by email:

- Your source code files: well structured and with the necessary documentation.
- A report in PDF that briefly describes the design and functionality of your program.
- A small manual and an example run of your program: the idea is that I should be able to test your program by myself, without having to contact you for additional information.
- Everything must be sent by email to the following address any time before the deadline: **steven.adriaensen@vub.ac.be**

Some important notes

Grading

Base functionality accounts for 18 out of 20 points for the project. Implementing all extensions (those suggested, or your own) can earn you up to 3 additional points (as a reference, implementing all suggested extensions perfectly). Note that bonus points are relatively difficult to earn, so focus on meeting the basic functional and non-functional requirements first (e.g. the quality of the schedules found by find_heuristically/1 is taken into account as well). The final grade for the project will never exceed 20, but extended functionality can compensate for minor imperfections.

Schedule Cost

There are some subtle differences in interpretation of the cost of a given schedule. Read the soft-constraints section (and examples therein) carefully. Make sure you obtain the same normalized optimal cost as listed in the table above. In doubt, do not hesitate to contact me.

Optimal Solutions for Large Instances

The optimal solution must only be reported for the small instance (as solving the large instances exactly under 2 min is not feasible in general). For the large instances only the execution time found by your heuristic must be reported.

Cuts

Use cuts in a correct way and at the correct place. When a rule is not supposed to backtrack put the cut inside that rule, and not when you use the rule:

```
foo(X) :- dont_backtrack(X), !
```

vs:

```
foo(X) :- dont_backtrack(X).  bar(X) :- foo(X), !.
```

Retract after/before assert

Sometimes a predicate can be implemented more easily/efficiently using assert. While useful when done properly, it has various risks:

- It potentially creates dependencies between queries. Make sure the requested predicates work individually (without requiring other predicates to be ran first)
- While correcting your assignment, we'll be dynamically loading/unloading different instances. Instance-specific memoized data can cause havoc in this setting.

One way to avoid these problems is to make sure that each of the requested predicates cleans up after itself, i.e. retracts all asserted data. Another approach is to retract all dynamically asserted data which might interfer with a predicate, beforehand.

Vrije Universiteit Brussel - Artificial Intelligence Lab - Pleinlaan 2 - B-1050 Brussels - info@ai.vub.ac.be