Solving an Exam Scheduling Problem Using a Genetic Algorithm

Dave Kordalewski
davekordalewski@gmail.com

Caigu Liu
liucaigu@gmail.com

Kevin Salvesen
kevin.salvesen@gmail.com

Abstract

This concise, one-paragraph summary should describe the general thesis and conclusion of your paper. A reader should be able to learn the purpose of the paper and the reason for its importance from the abstract.

The Problem

The problem we investigate is a sort of scheduling problem. We are attempting to find the most satisfactory choice of when and where to hold exams, given a background of student course loads.

An instance of this scheduling problem consists of a number of days on which exams can be scheduled (d), the number of time slots in which an exam can be scheduled on any day (t), a set of rooms (R), a set of courses (C), and a set of students (S), each of whom (s) has a particular course load, some subset of the courses (L_s).

A schedule, then, is a mapping from courses to rooms at times, which we can express like this:

$$C \to \{(r,\ a,\ b) \mid r \in R,\ a \in \{1..d\},\ b \in \{1..t\}\}$$

The total number of different possible schedules in any problem instance is $|C|^{|R|dt}$ which is typically far too large for brute force search. For example, one instance that we examine (scheduling the fall semester exams at the University of Toronto in 2009) involves 603 courses, 7 days, 8 times, 43 rooms, and 21945 students. This instance admits approximately 10^{6695} different possible schedules.

Typically, with relatively loose constraints on the number of rooms and particular schedules of students, it is not difficult to find some consistent schedule; that is, one where no student is asked to write 2 exams simultaneously and no room has 2 exams occur in it simultaneously.

Rather than worry about hard constraints, we prefer a framework of soft constraints, where we try to find a schedule that makes the students and invigilators happiest overall, allowing the possibility that some student or room is left with an impossible exam schedule, which can, in the real world, be dealt with on an individual basis.

The timetable that a student $s \in S$ has under some particular schedule K may be represented in this way:

$$TT_s(K) = \{(a,b) \mid \exists r \in R, \exists c \in L_s, K(c) = (r,a,b)\}$$

which may, in general, be a multiset.

Similarly, the timetable for a room $r \in R$ is

$$TT_r(K) = \{(a,b) \mid \exists c \in C,\ K(c) = (r,a,b)\}$$

We define two quality functions, mapping schedules to real numbers in [0,1], one for students and one for rooms. These are meant to capture how much they "like" the schedule under consideration. (For instance, a student will rate poorly any schedule where she has 2 consecutive exams, and very poorly any schedule which expects her to take two exams at the same time.)

$$Q_s(K) = f\left(TT_s(K)\right):\ \textit{Schedules} \to [0,\ 1]$$

$$Q_r(K) = g\left(TT_r(K)\right):\ \textit{Schedules} \to [0,\ 1]$$

We will consider later how to define these functions in a reasonable way.

The quality of a schedule, then, is given by

$$Q(K) = \frac{\left(w_s\left(\sum_{s\in S}\frac{Q_s(K)}{|S|}\right) + w_r\left(\sum_{r\in R}\frac{Q_r(K)}{|R|}\right)\right)}{(w_s + w_r)}$$

where w_s and w_r are the weightings we can use to indicate that we care somewhat more about students preferences than rooms, giving a quality (or fitness, in the terminology of genetic algorithms) in the range [0,1] for any schedule.

Evaluating this function Q can be computationally expensive when $|S|$ and $|R|$ are large. Our work examines how to find a schedule with relatively high fitness considering that we will want to do this with as few evaluations of Q as possible.

We examine, first and foremost, a genetic algorithm approach to solving this problem, but also touch on a few other methods.

Approach and Implementation

Differences From Natural Exam Scheduling Problems

The problem we have constructed here is a compromise between the most mathematically pure problem with the same sort of properties (which might ignore the rooms and related constraints completely) and the problem that must be solved in the real world when scheduling thousands of students, which has some important differences from this problem.

Tools used

The program is implemented in Java, to generate possible schedules and to calculate their fitness. A Mathematica program is also written to plot graphs, given the data generated by the Java program. Additionally, we use Google code subversion[1] and Google doc to share the work.

Fitness Function

In order to assign each schedule a fitness, we need to define the functions $q_{student}$ and q_{room}. They need to capture the idea of determining how much each student or room likes their given timetable. We do this by calculating the penalty incurred by a room, which is a measure of it having properties that are unpleasant to the student or room, and then using the formula

$$q(TT) = \frac{1}{1 + penalty_{TT}}$$

This function will be in the desired range or [0,1].

For students, we accrue a penalty of 50 for every time that is repeated in a timetable, 5 when there are exams in 2 consecutive times, 3 when 2 exams happen on the same day, and 0.5 when the student has exams on consecutive days. This accords reasonably well with real students' hopes for their exam timetables.

Rooms have the same penalty of 50 when it is used by two exams at the same time, 5 if it hosts exams in consecutive time slots (reasoning that the invigilators need time to let the students out and in and prepare for the next exam) and a penalty of 0.5 when a room is empty for an entire day between uses (reasoning that the room requires some amount of modifications to be used for exams, and it is difficult to use it for other purposes if it will still be used for exams soon.)

Genetic Algorithm Search

Although there are many readily available open source genetic algorithm software packages available (see JGAP (Meffert 2009), Jenes(CISELab 2009)) we decided to write a simple, yet general, genetic algorithm package, in Java, that can interface with data types designed by others, so long as they implement a few necessary methods. The package we wrote is more than adequate for investigating the scheduling problem under consideration here.

General Description [...]

Mutation Operator [...]

Crossover Operator [...]

Other Search Algorithms

Random Search The random search method simply generates random schedules, evaluates their fitness, and remembers the one that it has seen that has highest fitness. This method is meant as a baseline against which to compare the other methods. If a search method doesn't perform significantly better than random search with the same number of fitness evaluations, it is useless.

Hillclimbing Search The hillclimbing search that we have implemented examines every schedule in the neighbourhood of a given starting schedule, and then continues from the one with highest fitness, terminating when it has performed as many evaluations as it is allowed to, or it has found a schedule whose neighbour are all equal or inferior to it.

The neighbouring schedules are those where only one course meets in either a different room, on a different day, or at a different time. This leads to a total of $|C|(d+t+|R|-3)$ schedules in the neighbourhood of any given schedule.

It can be seen that this method is completely deterministic given a starting schedule, and will always return a local maximum

Mutate Search Mutate Search starts by generating a random schedule. It modifies this schedule (by using the same mutate operator made for the genetic algorithm) by 1% and rolls back if the mutation didn't increase the schedule's fitness. This operation is repeated until it has performed as many evaluations as it is allowed to. This algorithm is a variation of simulated annealing, where instead of accepting a change through a probabilistic function, it always accepts a change that leads to a higher fitness.

Evaluation

Success

[...] Seeing as our fitness function gives a very high penalty for schedules that have direct conflicts (either a student with two exams at the same moment, or two different exams in a same room at the same moment), if such a schedule exists given the input data, our algorithms will very quickly tend to reject all such schedules.

The Genetic Algorithm's Parameters

[...]

Comparison Between the Different Searches

[...]

Conclusion

We come up with a conclusion that in this particular problem, genetic algorithm is better than random search, and it is better than hill-climbing if the input is a large data set. But it may not be the best solution for this problem, since the MutateSearch performs way better.

References

CISELab. 2009. Jenes. http://jenes.ciselab.org.

Meffert, K. e. a. 2009. Jgap - java genetic algorithms and genetic programming package. http://jgap.sf.net.

[1] http://code.google.com/p/csc384-genetic-algorithms-project/