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

We examine the exam scheduling problem with soft constraints for problems of varying sizes using a genetic algorithm, random search, hillclimbing, and a variant of simulated annealing. This is done in the context of maximizing the fitness of a schedule, where the fitness is relatively computationally expensive to compute for any schedule. The genetic algorithm is discussed in detail. We consider selecting parameters for the GA that maximize the quality of the schedule found after a constant number of evaluations. The GA, with a naive crossover operator, performs much better than the random search, but simulated annealing is far superior in this search space.

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.)

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

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

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. For example, we don't deal with constraints such as room capacity or exams with varying length.

Tools used

Our application was entirely written in Java. We also wrote another Java application to generate problem instance files for testing our algorithms. We used Mathematica to plot all of our graphs. Additionally, we use Google code's source code repository[1] and Google docs to collaborate on this report.

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.)

We chose the weighting constants $w_s = 2$ and $w_r = 1$, which suggest that we are more concerned with the students' schedules than the rooms'.

The details of the function we used are essentially arbitrary, and we expect similar results would be attained for any other penalty-based fitness function modelling the preferences of students and requirements of rooms.

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

Genetic Algorithm Search

Although there are many readily available open source genetic algorithm software packages available (see JGAP, Jenes) 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 The genetic algorithm maintains a population of schedules and constructs subsequent generations by adding new random schedules, making copies of high quality schedules in the present generation, applies the mutation operator to schedules, or uses the crossover operator on 2 schedules. When performing mutation or crossover, the schedules are selected with a probability proportional to their fitness, allowing the schedules with high quality to be selected more frequently and make more "offspring".

The free parameters in our GA are the size of the population, number of generations to simulate, and proportions of new generations to be made by the copy, random, mutation, and crossover methods.

Representation We use the naive representation of schedules. A schedule contains lists of size $|C|$ containing the room in which each course's exam is to occur and the timing (corresponding to a pair or integers representing the day and time) of each course's exam.

We will see that this representation is far from ideal, but there is no clear alternative.

Mutation Operator The mutation operator takes a probability and a schedule. We loop over the courses, and, with the given probability, each is or is not set to a random room at a random time.

Crossover Operator Given two schedules, the crossover operator loops over the courses, and chooses at random which schedule to take the room and timing from for that course. This results in an interleaving of the original courses' data.

There is, unfortunately, no good reason to expect that the result of applying crossover to two relatively high quality schedules will produce a high quality schedule, since the quality of schedules can be very subtle, as they are a more global property of the schedule than this process captures. The crossover of two schedules where no student or room has a timing conflict is unlikely to result in a schedule with much more than average quality. There is no obvious reason to think that this sort of procedure will produce schedules with better than random fitness. Luckily, the schedules produced in practice are better than random.

If a different representation were chosen, such that the mutated schedules were different from, but likely to have quality similar to their parents, the GA would likely be able to perform much better than it does, but we could not discover such a representation and are satisfied with this method for present purposes.

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 makes an alternative schedule (by using the same mutate operator made for the genetic algorithm) with 1% of the data randomized and chooses which of the schedules is better. This operation is repeated until it has performed as many evaluations as it is allowed to.

This method can be changed to true simulated annealing by varying the degree of modification made to the current best schedule with the number of evaluations remaining. Not being the focus of our investigation, little effort was made to optimize the parameters. 1% does seem to give better performance than other nearby values, though we did not collect data to substantiate this. Finding ideal parameters for simulated annealing on this problem would make for a project-sized inquiry in itself.

Scheduling Problem Instances

We generated 2 sets of 3 scheduling problem instances to test the algorithms above with. The package `Schedule Instance Generator` in our code repository manages the generation of random problem instances with many free parameters to specify characteristics of the output instance.

The first set of instances were generated with the parameters in the table below. The rooms and courses and students were given arbitrary names and the courses that each student attended were selected with uniform probability from the complete set of courses.

The second set was generated with a sort of correlation between the courses that each student took, rendering the dataset more true to the sort of data that would be found in real-world instances of the exam scheduling problem. The set of courses that each student takes was selected in the following way. Each student was assigned, with uniform probability, a *major* and a random number of courses between 2 and 6. Each course was then selected, with probability 0.7, from among the $|S|/\#Majors$ courses in their major, or, with probability 0.3, uniformly from the set of all courses. We can expect that each student takes 70% of their courses from a small subset of the courses, representing the field that they specialize in, and 30% from other fields. This simulates the way in which we expect student bodies to choose their courses, though the true data is likely to exhibit far more structure, due to program requirements, notorious professors, etc.

The 3 sizes vary from manageable to large. The smallest could potentially be solved exactly, with methods not under consideration here, while the others have search spaces much too large for any hope of an exact solution. For the largest, which simulates a UofT exam schedule, a single evaluation of the fitness function took 0.2 seconds on the fastest machine owned by any of the authors.

The following table characterized the problem instances that we used (with SSS being an approximation of the search space size):

	S	M	L	S_{Cor}	M_{Cor}	L_{Cor}
d	7	10	7	7	10	7
t	5	8	8	5	8	8
$\|C\|$	20	200	603	20	200	600
$\|R\|$	4	10	43	4	10	43
$\|S\|$	50	300	21945	50	300	21945
$\|L_s\|$	1-5	1-6	1-5	2-6	2-6	2-6
#Maj	N/A	N/A	N/A	4	10	20
SSS	10^{182}	10^{1840}	10^{6695}	10^{182}	10^{1840}	10^{6689}

The large data set was created with parameters inspired from the University of Toronto's Faculty of Arts and Science.

Evaluation

Selecting Parameters for the GA

In order to select parameters for the genetic algorithm (that is, the population, number of generations to simulate, proportions of copies, randoms, mutations, and crossovers for the next generation) we performed tests using the small uncorrelated problem instance.

Generation Parameters We undertook a search of the space of parameters for forming new generations.

In the graph below, each data point represents the average of 5 trials. To make the scale clearer, the colour on the surface corresponds to the fitness, where the highest fitnesses are red. The number of copies is fixed at 1, and the proportions of mutations and crossovers are shown on the horizontal axes.

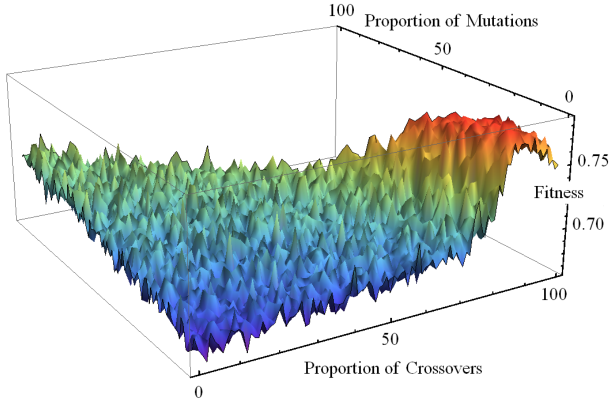

Figure 1: Effectiveness of the GA's proportion parameters

The graph above indicates that for a fixed number of evaluations, best results are achieved by selecting parameters that have the majority of individuals in the next generation be generated by crossovers of the most fit schedules in the previous generation. In order to achieve the very highest fitness, a few randoms or mutations should also be incorporated.

This test was done for only the smallest problem instance, and is infeasible for the larger instances. We chose generation proportion parameters to approximate the optimal values seen here, under the assumption that the larger problems exhibited this sort of preference.

Size Parameters We also performed tests to gauge the optimal population size and number of generations to calculate. We determined that very small values for either the population or number of generations result in very poor generated solutions. A population of around 50 is sufficient for good diversity to be retained in the population, and having a number of generations relatively large compared to the population size seemed to be effective.

We gathered data from the small and medium problem instances, but the precise qualities provided by particular parameters will not necessarily hold in larger instances. The optimal population size and number of generations (for a fixed number of fitness evaluations) is likely to vary for larger problem instances, but it seemed clear that the number of generations should be set somewhat higher than the population.

Comparison Between the Different Searches

We tested the 4 search methods on all 6 of our peoblem instances. The results for only two of the problems are shown, she small and large problems, since the medium problem exhibited characteristics very similar to the large problem.

Since all methods produced nearly indistinguishable results for the uncorrelated versions. This suggests that it may be possible to use the expected internal correlations more effectively that the methods we have proposed do.

On the graphs for the small instance, the results shown are averages over many trial runs. Individually, however, the different methods all produced similar results, in particular the ranking of the algorithms was very consistent for any particular problem instance.

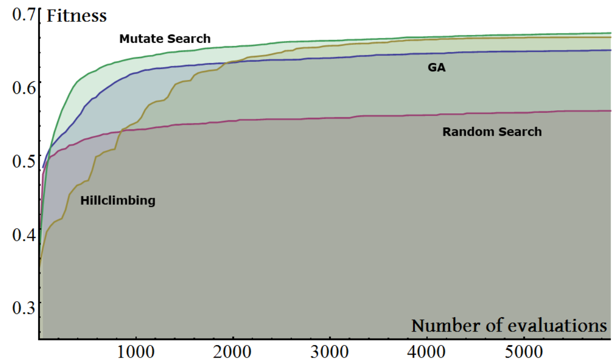

Figure 2: Comparison of the different searches on a small data set

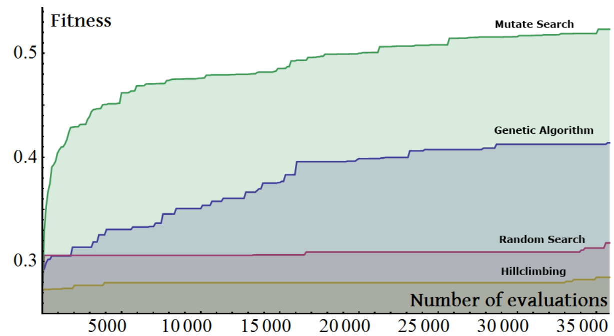

Figure 3: Comparison of the different searches on a large data set

The mutate search algorithm performed remarkably well in all tests, finding a schedule at least somewhat better than every other method for every size problem instance.

Conclusion

[. . .]

References

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

Corne, D.; Fang, H.-L.; and Mellish, C. 1993. Solving the modular exam scheduling problem with genetic algorithms. http://www.dai.ed.ac.uk/papers/documents/rp622.html.

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

UofT-ArtSci. 2009. University of toronto - faculty of arts & sciences december 2009 examination schedule. http://www.artsci.utoronto.ca/current/undergraduate/exams/dec09.

UofT. 2009. University of toronto statistics. http://www.utoronto.ca/internationalstudent/index_stats.html.