Exam Timetabling Using Genetic Algorithm

Marko Čupić, Marin Golub, Domagoj Jakobović
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
{Marko.Cupic|Marin.Golub|Domagoj.Jakobovic}@fer.hr

Abstract. *In this paper we present a case study concerning the exam timetabling problem we faced, and its genetic algorithm based solution. Several variations of the algorithm as well as the influence of algorithm parameters are analyzed.*

Keywords. Exam timetabling, Genetic algorithm

1. Introduction and Motivation

In academic year 2005/2006 Faculty of Electrical Engineering and Computing (FER) was one of the first members of University of Zagreb to align itself with Bologna Process. This introduced many organizational issues. Offering students a number of elective courses created several timetabling problems – from lecture timetabling and laboratory timetabling to exam timetabling. Each semester was divided into 3 cycles, each containing four or five weeks of lectures followed by a week for laboratory exercises followed by a week for exams (see Figure 1.a). Previous program organization had equally distributed laboratory exercises and exams throughout semester, so this new organization increased the pressure to students. During the period of four years since the alignment with Bologna process started, organization was gradually modified to relieve created pressure. Laboratory exercises are now blended into lecture weeks, and exams are expanded into two weeks (see Figure 1.b).

The alignment with Bologna process imposed several very challenging organizational tasks. Students can specialize themselves by taking several of a number of elective courses offered. Also, there is no strict distinction among different years of study; instead, courses have prerequisites, which enables a student to enroll some of the courses offered on first, second and third year at the same time (for which the prerequisites are met). Those situations render lecture and laboratory exercise timetabling very difficult.

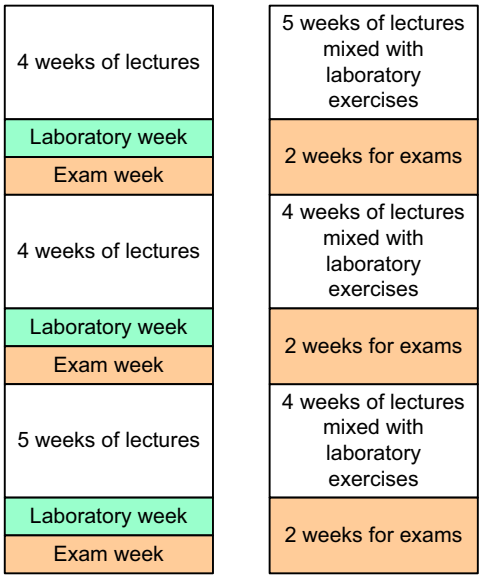

Figure 1. Semester organization. (A) first year after alignment with Bologna process, and (B) current organization.

Concentrated exams are also very demanding for students, so it is necessary to produce the exam timetable such that as many of students as possible have maximally sparse schedule (for example, more than one exam in the same day or exams in adjacent days should be avoided if possible). Creating such a timetable can not be done by hand. Simple algorithms performing exhaustive search on real case examples are also not an option, due to the complexity of the problem. There are many methods applicable for such problems (see e.g. [1]). A comparison of methods for solving such problems can be found in [7]. This paper will focus on a solution based on evolutionary inspired algorithm known as genetic algorithm [4,5,6]. We will describe its performance on a real case scenario: timetabling first mid-term exams of the academic year 2008/2009, winter semester. We will analyze several variations of genetic algorithm and the influence of its parameters on the quality of the obtained solutions. Finally, we will show that

genetic algorithm can be a powerful tool for tackling with the described problem, and that it can be easily adapted to other faculties facing the same problems.

This paper is organized as follows. In Section 2 we give a problem definition and describe the size of the problem. In Section 3 we describe the genetic algorithm based solution. In Section 4 we describe several variations of algorithm we tested. Section 5 gives experimental results.

2. Problem Definition

Specific exam timetabling problem for which we needed a solution can be described as follows.

We define $T=\{t_1,t_2,\ldots,t_n\}$ a set of distinct non-overlapping terms. A term is a time period in a specific day having a beginning time and duration. Several terms can exist in the same day (but not overlapping). Function *day(t)* for each term $t\in T$ gives its *day*. Function *capacity(t)* for each term $t\in T$ gives maximal number of students that can be assigned to that term (this reflects the number of rooms available for exams at that specific time).

There is a set of students $S=\{s_1,s_2,\ldots,s_l\}$ and a set of courses $C=\{c_1,c_2,\ldots,c_m\}$, and mapping *enrollment*: $C\rightarrow CS$, which for each course $c\in C$ gives a set of enrolled students $CS\subseteq S$.

There is also a set of fixed course terms $FXC=\{(c_x,t_y), (c_w,t_z),\ldots\}$. That means it can be required that some courses have exams at exactly specified terms. This means that the algorithm can not arbitrarily chose best terms for those courses (so in a way, the problem becomes simpler). However, the algorithm must be aware of those assignments, in order to avoid creating conflicts when assigning other courses at the same terms.

There is a set LC of disjunctive sets LC_i of linked courses: $LC=\{LC_1,LC_2,\ldots\}$, $LC_i\subset C$, and for each $i\neq j$ $LC_i\cap LC_j=\emptyset$. Linked courses LC_i are courses for which exams are required to be held at the same term. The term itself can be selected by the algorithm, but all linked courses must then be assigned to that term. This can potentially lead to ill-posed problems, if a special care is not taken to ensure that all linked courses are disjunctive with regard to enrolled students. Handling this constraint is required by similar courses held for different study programs, in order to allow the creation of one exam sheet with slight variations for each study program.

As a solution to the problem, it is required to find a mapping *solution* $C\rightarrow T$ that for each course $c\in C$ assigns one term $t\in T$ in which that course's exam should be held. Such a solution must satisfy the following constraint: let CT_i denotes a set of all courses $c_j\in C$ that are scheduled in term $t_i\in T$; then for each two courses c_x and c_y from CT_i, $x\neq y$ must be $enrollment(c_x)\cap enrollment(c_y)=\emptyset$, i.e. there should be no conflicting courses regarding enrolled students assigned to the same term.

Typically, there will be many solutions for which the imposed constraint is met. So, we further search a solution whose *quality* is better. The quality of a solution will depend on following two factors: (i) how many students are scheduled to have more than one exam at the same day and how many times this happens for those students, and (ii) how many students have scheduled exams at adjacent days and how many times this happens for those students. The first factor is also more significant than the second, i.e. it is better to have two exams at adjacent days than at the same day.

In this paper, we will consider the following case. There are more than 18000 student-course enrollments, with 102 courses giving in average 180 students per course. Of those courses, exams for 77 should be scheduled in a two week period. Furthermore, in each day there are three terms available (09h-11h, 12-14h and 15h-17h), and in each week five days are available, which gives a total of $3\times5\times2=30$ terms. A number of possible timetables with those parameters is roughly bounded by $30^{77}\approx 10^{114}$. This clearly indicates that any attempt to solve this problem that is based on exhaustive search is not an option.

3. Genetic Algorithm Based Solution

Genetic algorithm is an evolutionary inspired metaheuristic method that does not guarantee to find optimal solution. In fact, there are a number of problems known as deceptive functions that are extremely hard for genetic algorithm. However, in many cases genetic algorithm can relatively quickly find a good-enough solution.

Genetic algorithm works with a population of possible problem solutions. Each solution is often called a chromosome (in this paper we will use both terms interchangeably). For each solution a quality is measured (called *fitness*). Using some probabilistic selection method, two chromosomes are selected and combined by

appropriate crossover operator to produce a new chromosome called a child. This child can also be mutated using mutation operator. Then, the obtained child replaces some solution from the population, and the whole procedure repeats itself. Pseudocode of genetic algorithm is shown in Figure 2.

```
initializePopulation P
for each sol_i from P
   calculateFitness(sol)
repeat
   select two parents sol_1 and sol_2 from P
   child = crossover(sol_1, sol_2)
   mutate(child)
   calculateFitness(child)
   replaceSome(P, child)
until stop condition not meet
```

Figure 2. Pseudocode of genetic algorithm

In order for the algorithm to work properly, the selection operator must pick parents from population probabilistically, but in a way that better parents have a greater chance to be picked (this is called *evolutionary pressure*). This way it is expected that, in average, a child will also have a better quality, and after entering the population and replacing some solution that is worse, the average population quality will also be improved. The selection method we have used can be described as follows. We select three random solutions from the population. The solutions are sorted based on fitness. the best two solutions are uses as parents, which leaves the third (the worst) solution as a candidate for replacement (thus producing evolutionary pressure).

After the creation of a child solution from the selected parents and probabilistic mutation, the child is checked to prevent duplicate solutions. If a duplicate is found in the population, the child is discarded. Otherwise, one of the solutions from the population is replaced with newly obtained child, and the procedure is repeated, either until a good-enough solution is found, or a predetermined iteration count is reached.

3.1. Global Data Structures

In order to efficiently implement genetic algorithm for this problem, we define four globally available data structures, as shown in Figure 3. Each course contains its name, its index in *courses* array, enrolled students, a flag saying whether it should be assigned in fixed term (and which term), a number of students it shares with other courses, and a flag saying whether it is a member of some linked courses group (called CourseCluster).

```
Course[] courses;
Term[] terms;
Student[] students;
CourseCluster[] courseClusters;
```

Figure 3. Global data structures

Each term contains its time information, day index, capacity and associated index in *terms* array. Each course cluster contains indexes of courses from courses array which are members of the same group.

3.2. Chromosome Structure

There are many ways to represent a single solution that are adequate for genetic algorithm. The simplest way is to represent a solution as a string of bits rendering crossover and mutation operators extremely simple. However, decoding such a representation can be inefficient. Instead, we have used a slightly redundant representation that enabled us to implement most of the genetic operators as O(1); see Figure 4.

```
KCourse[] kcourse;
KTerm[] kterms;
int[] clusterTerms;
int[] eval;
```

Figure 4. A single chromosome

KCourse is a chromosome specific simple structure containing a pointer to global course information, index of assigned term, and pointers to previous and next KCourse, allowing the creation of circular double-linked list of KCourses which belong to same term. KTerm is also a chromosome specific simple structure containing a pointer to global term information and pointer to some KCourse object which is scheduled to that term. Having this structures set up, information such as "all courses assigned to term t_i" or "in which term is assigned course c_i" can be directly obtained with no searching. This is illustrated in Figure 5, with 8 courses and 3 terms. Arrows above courses represent *next* course relation (e.g. next(c1)=c3, next(c3)=c4,

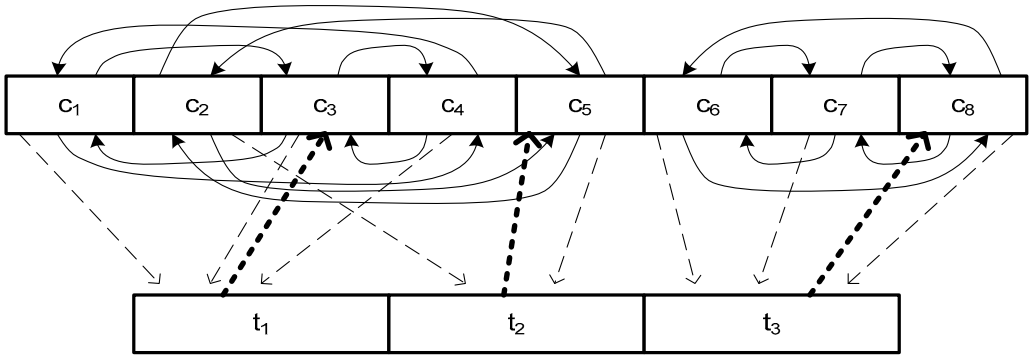

Figure 5. Chromosome structure

next(c4)=c1), while arrows below courses represent *previous* relation (e.g. previous(c1)=c4, previous(c3)=c1, previous(c4)=c3). Pointers from courses to terms are represented as dashed arrows, and pointer from each term to one term representative course is given as dotted bold arrow. Now it is obvious that moving a single course from one term to another can be implemented as O(1) operation. The array clusterTerms contains indexes that for each CourseCluster point to the term in which all courses from associated cluster are assigned. Finally, array *eval* contains components of fitness function for that chromosome. Similar structure was successfully used in [10].

3.3. Crossover and Mutation

The crossover operator is implemented as uniform crossover. The first parent is copied into the child. Then, for each course its term is updated to match the term it has in second parent with 50 percent probability.

Two mutation operators are implemented. The first mutation is *course mutation*, which with probability *probMut* randomly selects new term for a course. The second mutation is *term mutation* which with probability *probSwapTerm* randomly selects a second term and then exchanges all courses between them.

3.4. Evaluation Function

The fitness of a solution is composed of three components. To obtain the first component, for each term and for each course in that term a number of students shared with other courses assigned to the same term is counted (this represents a conflict). Total conflict count is then stored as *eval*[0] in chromosome. The second component measures the solution quality. For each term t_i and for each course c_j in that term it counts how many exams students of course c_i have at that same day (this count is then scaled by factor 4), and how many exams they have the next day (this count is added without scaling). The previous day is not considered in order to prevent double counting. This number is then stored as *eval*[1] in chromosome. Finally, for each term a number of students exceeding term capacity is counted, and a total sum is then stored as *eval*[2]. With fitness defined this way, better solution is the one having smaller numbers as components of *eval*. Also note that in order for a solution to be acceptable, *eval*[0] (number of conflicts) and *eval*[2] (number of exceeding students) should be 0.

4. Variants of the Algorithm

To analyze how this algorithm performs, several variants were implemented, all of which are the result of modification of three algorithm procedures: (i) parent selection procedure, (ii) replacement selector procedure and (iii) modulated randomness of the random number generator for term selection, as explained in the following sections.

4.1. Parent Selection Procedure

In order to assure adequate evolution pressure, when selecting parents to crossover, three solutions are selected randomly from population, and then ordered by their fitness. However, to sort selected solutions, a total ordering procedure must be defined, since *eval* is a vector, and not a scalar. Some approaches to tackle this are described in [3,8,9]. We defined two procedures. The first one, called *wsel*, uses weighted sum of components, so that we can obtain a scalar *fitness*=(*eval*[0]+*eval*[1])*w_1+

eval[2]**w*$_2$. The solutions are then ordered based on calculated scalar value.

The second approach (*hsel*) introduces a hierarchy among components of *eval*. To simplify this, we map three-component *eval* into two-component tuple (e_1,e_2)=(*eval*[0]+*eval*[2], *eval*[1]), and compare the solutions firstly by the first element, and if they are equal, then by the second element.

4.2. Replacement Selection Procedure

After a child is created, a solution from the population must be selected which will be replaced with the child. We created two procedures: *nrep* selects the third (the worst) solution from initial parent selection procedure to be replaced. *hrep* first divides the whole population with regard to child into four quadrants based on mapped *eval* tuples (e_1,e_2), see figure 6.

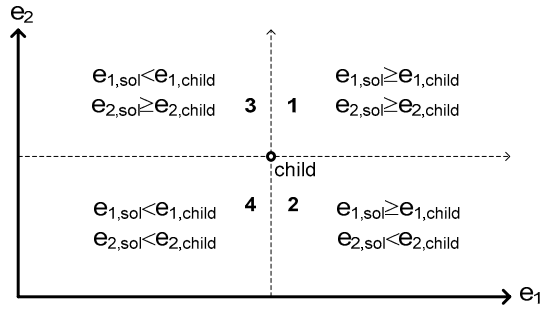

Figure 6. Population division

The procedure then finds the first nonempty quadrant (starting with quadrant marked with 1), and selects one random solution from that quadrant to replace. Such approach has been described in [2].

4.3. Modulation of Probability Function

Initially we experimented with uniformly distributed random number generator (*mod1*). However, after a number of experiments, we observed situations where some terms were relatively rarely used. To tackle this, during the optimization we periodically check frequency usage of each term in whole population, and modify probability function so that random number generator returns indexes of terms with lower usage with higher probability (*mod2*). As a third procedure (*mod3*), we used mod1 50% of the time and mod2 the other 50%.

5. Results

Based on the described algorithm variations we ran a number of experiments on real data with 77 courses that needed to be scheduled in 30 terms over a 10 days period. Each time genetic algorithm was allowed to make 4000000 iterations (each iteration is one selection / crossover / mutation / replacement cycle), which took about 2 minutes. We varied parent selection and replacement selection in three combinations (*wsel*/*nrep*, *hsel*/*nrep*, *hsel*/*hrep*), where wsel used factors w_1=1 and w_2=0.05. Each of those combinations was tested with three probability functions (*mod1*, *mod2* and *mod3*). For each of those we also varied *probMut* and *probSwapTerm* (values were taken from the set {1, 2, 5}), and population sizes (values were taken from the set {20, 50, 100, 200}), giving a total of 324 experiments. Each experiment was repeated 3 times in order to get the average solution quality.

A selection of obtained results is presented in table 1. Table shows obtained results as a function of population size, probability of mutation, probability of term swapping, selection operator and replacement type. For each probability function 5 of best solutions are given. The best found solution is obtained when probability function is modified so that rarely used terms are more likely to be generated next. In best solutions *probMut* parameter is tipically small (1 or 2). Worst solutions were generated with *probMut* set to 5. It also seems that algorithm is not too sensitive to values of *probSwapTerm* parameter.

6. Conclusion and Future Work

In this paper we presented an approach to producing exam timetables which is based on genetic algorithm. Genetic algorithm is an extremely robust method which can successfully provide good-enough solutions to many optimization problems, one of which is exam timetabling. We tested the created algorithm on a real data and obtained satisfactory results.

The behavior of the developed algorithm was examined with respect to parameter variations and different selection and replacement procedures. Best results were obtained with moderate populations (50 to 100), small *probMut*, and by using procedures *hsel*, *hrep* and *mod2*, and all of that in a rather small amount of time (less than two minutes). As part of this

Table 1. Results of parameter variations

Population size	pMut [%]	pSwap [%]	Selection	Replacement type	Number of conflicts eval[0]	Solution quality eval[1]	Capacity violations eval[2]
Best 5 solutions (mod3)							
200	1	5	wsel	nrep	0.67	1754.67	0
50	1	1	wsel	nrep	0.33	1763.33	0
20	1	5	hsel	hrep	0.00	1770.00	0
20	1	1	hsel	hrep	0.00	1789.00	0
50	1	5	wsel	nrep	0.33	1795.00	0
Best 5 solutions (mod1)							
100	1	5	hsel	nrep	0.00	1733.33	0
20	2	5	hsel	nrep	0.00	1733.33	0
20	2	2	hsel	nrep	0.00	1749.00	0
200	2	2	hsel	hrep	0.00	1757.67	0
50	2	5	hsel	nrep	0.00	1760.67	0
Best 5 solutions (mod2)							
50	1	5	hsel	hrep	0.00	1708.00	0
20	1	5	wsel	nrep	0.33	1718.67	0
100	1	5	hsel	hrep	0.00	1740.33	0
50	1	5	hsel	nrep	0.00	1742.67	0
20	1	5	hsel	nrep	0.00	1762.33	0

work, an open source application is being developed which will incorporate the developed algorithm, and enable a broader academic community to solve its exam timetabling problems.

8. Acknowledgements

This work has been carried out within projects 036-0361994-1995 Universal Middleware Platform for Intelligent e-Learning Systems and 036-0362980-1921 Computing Environments for Ubiquitous Distributed Systems both funded by the Ministry of Science and Technology of the Republic Croatia.

9. References

[1] Burke E.K, Newall J.P, WeareR.F. A memetic algorithm for university exam timetabling. In: E.K. Burke, P. Ross editors. Practice and Theory of Automated Timetabling: Selected Papers from the 1st International Conference. LNCS 1153. Springer-Verlag, Berlin, Heidelberg. p. 241-250, 1996.

[2] Chu P. C, Beasley J. E. Constraint Handling in Genetic Algorithms: The Set Partitioning Problem. Journal of Heuristics, 11, p. 323-357, 1998.

[3] Fonseca C. M, Fleming P. J. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In Forrest S, editor. Genetic Algorithms: Proceedings of the Fifth International Conference, pp. 416-423, San Mateo, CA: Morgan Kaufmann, 1993.

[4] Goldberg D.E. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, 1989

[5] Haupt, R. L, Haupt S.E. Practical genetic algorithms. New York: Wiley, 1998.

[6] Mitchell M. An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press, 1998

[7] Naji Azimi Z. Comparison of metaheuristic algorithms for examination timetabling problem. Applied Mathematics and Computation, 16(1-2): p. 337-354, 2004.

[8] Schaffer J. D. Multiple objective optimization with vector evaluated genetic algorithms. In Grefenstette J. J, editor. Proceedings of the First International Conference on Genetic Algorithms; Lawrence Erlbaum; p. 93-100; 1985.

[9] Srinivas N, Deb K. Multiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation 2(3), 221–248, 1994.

[10] Talbi E-G, Weinberg B. Breaking the search space symmetry in partitioning problems: An application to the graph coloring problem. Theoretical Computer Science (TCS), Vol.378, No.1, p. 78-86, 2007.