Scheduling problems at a university: a real-world example

Marko Čupić* and Tin Franović

Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, 10000 Zagreb, Croatia
Fax: +385-1-6129653
E-mail: Marko.Cupic@fer.hr
E-mail: Tin.Franovic@fer.hr
*Corresponding author

Abstract: In this paper, we discuss the development and usage of various kinds of educational activity schedulers used at faculty level. Problems solved by such schedulers are of paramount importance for organisation of educational activities during the semester. Implementations of adequate evolutionary computation-based algorithms are being developed as part of the *Ferko* project. Various parts of the *Ferko* project have already been used for several years by thousands of students and faculty staff members at our institution.

Keywords: exam scheduling; laboratory scheduling; make-up scheduling; assigning students to lecture groups.

Reference to this paper should be made as follows: Čupić, M. and Franović, T. (2011) 'Scheduling problems at a university: a real-world example', *Int. J. Knowledge and Learning*, Vol. 7, Nos. 1/2, pp.51–69.

Biographical notes: Marko Čupić received his MSc in 2006 at the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. His current research focuses on university scheduling problems, e-learning and knowledge assessment and various areas of artificial intelligence.

Tin Franović received his BSc in 2010 at the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. He is currently pursuing his MSc in Computer Science at the same university.

This paper is a revised and expanded version of a paper entitled 'Ferko project – intelligent support for course management at faculty level' presented at the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2010), Opatija, Croatia, 24–28 May 2010.

1 Introduction

When talking about the role of computers in education, today's mainstream research topics are focused on helping students to learn by creating various formats of learning materials (e-learning, m-learning, intelligent tutoring systems, etc.) and helping teachers

to better assess students knowledge (e-assessment systems providing adaptive and intelligent assessments techniques). However, even today, the majority of courses are still being taught in classrooms, while assessments are still being conducted on-campus under staff supervision. For all but small-sized universities, creating high-quality lecture, assessment, and laboratory exercise schedules poses a serious challenge.

Being part of faculty administration is definitely not an easy job. This is even more so considering all the tasks needed to be done in light of the new Bologna reform. Out of all challenging tasks, we will single out several that are performed at faculty level: creating a semestral lecture schedule for all courses and enabling students to exchange groups in case of conflicting schedules, creating weekly laboratory schedules for all courses, creating exam schedules for all courses, and supporting course staff to manage make-up or deferred exams more easily. With regards to computational complexity, all of those problems are combinatorial NP (i.e., non-deterministic polynomial time), and solving them by hand – even in an attempt to obtain a best-effort quality schedule – is unfeasible.

Fortunately, adequate algorithmic techniques have been developed for dealing with NP class combinatorial problems. Most prominent candidates are biologically inspired meta-heuristics researched under the evolutionary computation umbrella, such as evolutionary algorithms (e.g., genetic algorithm) (De Jong, 2006; Deb, 2009; Affenzeller and Wagner, 2009), swarm algorithms (e.g., particle swarm optimisation and ant colony optimisation) (Eberhart and Kennedy, 1995; Montes de Oca et al., 2009; Dorigo and Stützle, 2004), artificial immune systems (De Castro and Von Zuben, 2000; De Castro and Timmis, 2002; Cutello and Nicosia, 2005), and differential evolution algorithms (Feoktistov, 2006; Price et al., 2005).

The goal of *Ferko*, an open-source course management and support system we are developing, is to improve the quality of studying and to leverage faculty administration and course staff of many of the common and tedious tasks. In this paper, we present work carried out as part of the development of *Ferko*, addressing most of the above-mentioned problems by providing intelligent optimisation methods based on evolutionary computation algorithms, capable of automatically producing good-quality schedules. We comment on the performance of developed algorithms, and discuss how they can be used to provide an intelligent support for course management at faculty level.

2 Related work

The possibilities of using computers to solve scheduling problems at faculty level have been studied extensively throughout the years. One of the first attempts at solving scheduling problems dates back to 1959 and is attributed to Blakesley (1959).

Modern approaches to the scheduling problem usually rely heavily on evolutionary computing and heuristics because of the NP-hardness of the task. These approaches can be divided into several categories, depending on the algorithms used for solving the task. One of the most popular algorithms are genetic algorithms, whose application to scheduling problems has been studied by Adamidis, Beligiannis, Dorigo, Mamede, Salwatch (Adamidis and Arakapis, 1997; Beligiannis et al., 2008; Colorni et al., 1990; Mamede and Rente, 1997; Salwach, 1997) and most notably by the automated scheduling and planning group at the University of Nottingham, UK. Memetic algorithms have also been used, such as the ‘neeps and tatties’ (Cumming, 1997) system used at Napier University. Other approaches include simulated annealing (Thompson and Dowsland,

1996), tabu search (Abdullah et al., 2006) and constraint logic programming (Kambi and Gilbert, 1996; Gueret et al., 1995). One of the most recent approaches includes the use of reinforcement learning in determining a suitable heuristic based on the problem presented (Ozcan et al., 2010).

Since timetable scheduling is a big problem for educational institutions, a number of open-source and commercial software applications have been developed in order to serve as help with scheduling. Most of these applications, including the *Ferko* system, have been developed at universities to suit their scheduling needs first, and published later on.

One of the examples of university timetabling software is the UTTS system (Lim et al., 2000) developed at the National University of Singapore. It is a complete solution for scheduling exams at university level and has been successfully used at a university with more than 21,000 students. The algorithm used for the version described in Lim et al. (2000) is the arc consistency algorithm (Mackworth, 1977) and is basically applied to a constraint satisfaction problem. The constraints used include two hard constraints (each student can attend just one exam at a time and the venue assigned to an exam must be large enough to accommodate all students) and a number of soft constraints which are used in fine-tuning of the schedule. The main drawback of the system is that it works best if each of the faculties is computed separately, with minimal communication, but improvements to this problem are already being researched.

Arguably, the most state-of-the-art implementation of an automatic scheduler is the *UniTime* (UniTime | University Timetabling, 2010) system, the winner of the 2007 International Timetabling Competition. *UniTime* is a comprehensive open-source academic timetabling solution which covers all university timetabling needs such as course scheduling, student sectioning, examination and event scheduling. It has been successfully applied at Purdue University, which has more than 39,000 students, 9,000 classes and 2,400 examinations which generate more than 190,000 course requests. The underlying algorithmic support consists mainly in the iterative forward search algorithm (Müller, 2009), implemented in Java.

Other commercial solutions include *Evolvera TimeEdit* (Evolvera, 2010), which is currently used by more than 30 universities in Northern Europe, including the KTH Royal Institute of Technology in Stockholm, *Infosilem EnCampus* (Infosilem, 2010) and *CELCAT Timetabler* (Celcat online, 2010).

The main problem with all commercial solutions is that it is difficult to adapt to all the specific scheduling needs a university can have. Another problem can be the integration with room-booking applications used at a university, but that can be solved by providing a room-booking engine included in the software package, as is the case with the *CELCAT Timetabler*.

3 Scheduling problems through the semester

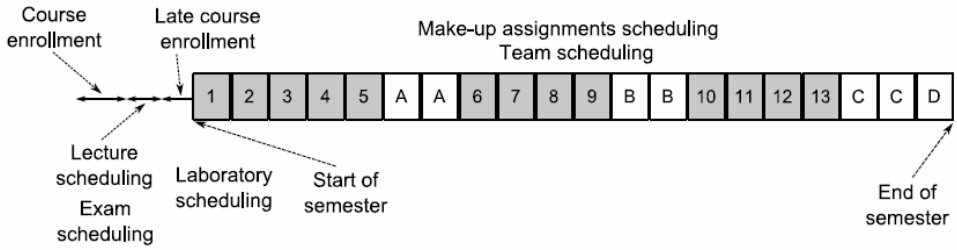
In this section, we will briefly describe the organisation of educational activities at our institution: Faculty of Electrical Engineering and Computing (FER), University of Zagreb, Croatia, having several thousand of students, between 130 and 140 courses per semester spanned over five years (i.e., 3 + 2).

Since we are not directly involved with lecture scheduling, we will briefly describe only the outcome. The lecture scheduling process results with a lecture timetable and for each course students are divided into lecture groups. On some courses, several lecture

groups may be combined into a bigger group and assigned to a lecture room (depending on the lecture room capacity). For each course, students may have one or more lectures during one week. For example, on the first-year *Digital logic* course the number of enrolled students is around 800. These students are divided into ten lecture groups. Each lecture group has four hours of lectures (two hours in two days). Having limited room resources, lectures are scheduled during the whole day. However, the lecture timetable is created in such a way that during a single week each group has lectures either in the morning or in the afternoon. The following week the schedule is reversed, so that groups which had lectures in the morning the previous week now have lectures in the afternoon, and vice-versa. However, there are exceptions to this pattern. Some courses on the third year and all of the courses for graduate students do not alternate.

Exam scheduling is performed for four periods. After lecture week 5 there is a two-week period for *first midterm exams* (Figure 1, weeks *A*). After lecture week 9 there is another two-week period for *second midterm exams* (Figure 1, weeks *B*). Finally, after lecture week 13 there is a two-week period for *final exams* (Figure 1, weeks *C*). For students who failed the course and for students who are not satisfied with the obtained results, make-up exams are organised in the week following the final exams (Figure 1, week *D*). More information on exam scheduling is provided in Section 4.

Figure 1 Timeline for semester-wide organisational activities



Notes: Each box represents one week. 1–13 are lecture weeks, *A–D* are exam weeks.

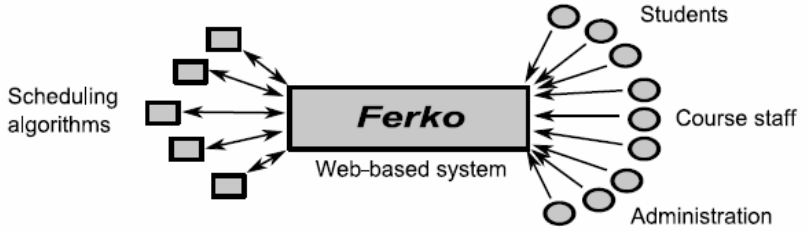
Once the lecture and exam timetables are created, course enrolments are resumed. However, during this period, students' freedom to choose courses is restricted: they cannot enrol to courses which have exams at the same time. Additionally, they cannot enrol courses which have only one group of lectures, and these lectures are at the same time. This period is known as *late enrolments*, and the number of such students is relatively small (although not insignificant) compared to the number of students enrolled during the regular enrolment period.

Late enrolled students are also not automatically assigned to lecture groups. Namely, since these students enrolled after the lecture schedule has been completed, free space in the lecture groups is limited and each late enrolled student potentially competes with other late enrolled students for that space. This is especially true if the enrolled course has multiple groups with multiple lecture terms. Instead, group assignment for late enrolled students is performed periodically, only after a number of enrolments have been processed, in an attempt to ensure that lecture schedules for all of these students remain without collisions. The algorithm solving this problem is further described in Section 5.

To help students to keep track of all activities, we have developed a web-based system named *Ferko* (see Figure 2). This system integrates the functionality of a course

management, learning management and timetable system. After logging in, students are provided with a personalised calendar showing all the student's educational activities (lectures, laboratories, exams, etc.). *Ferko* is also equipped with a schedule analysis module which allows us to easily obtain schedule activities for a given period of time for specified students, which is then used as input for other scheduling algorithms which create additional schedules which must not conflict with the existing ones. These schedules can then be published in *Ferko* which will make them visible to students and to other scheduling algorithms. For each user (student or a member of teaching staff), *Ferko* also provides the activities in *iCal* format, which can then be used for live synchronisation with any time-management system (e.g., one of the most popular among our students is Google Calendar, followed by a number of calendar readers on mobile devices).

Figure 2 *Ferko* is a system which combines CMS-capabilities and timetabling



Once the semester begins, a centralised laboratory scheduling is performed, which must create laboratory timetables for lecture weeks 3 to 13, taking into account previously created lecture timetables. During lecture week 1, laboratories are not organised on any course, and if a course requires laboratories in lecture week 2, these are scheduled as make-up assignments, as described in Section 7. The algorithms used for centralised laboratory scheduling are described in Section 6. All of those schedules are published in *Ferko*.

Finally, once the centralised scheduling activities are finished, a number of sporadic scheduling processes take place. On some courses, some lecture terms will be skipped due to absence of the lecturer. Some students will skip assigned laboratory terms due to medical (or other) reasons. Some students will not attend first or second midterm exams. For some courses additional lecture terms will be requested. Some courses will have requirements on laboratory scheduling that cannot be expressed in the form of the supported constraints. These (and similar) requests are left to course staff to handle. However, we have developed a number of tools and algorithms which are used by course staff for the creation of appropriate schedules and for their publishing in *Ferko*. Details are provided in Section 7.

4 Exam scheduling

One of the earliest scheduling activities for a semester is the creation of exam timetables. This activity is started after the regular course enrolments period has finished and the information about which students enrolled which courses is available. Due to the fact that our institution has prerequisite-based course enrolments and offers a number of

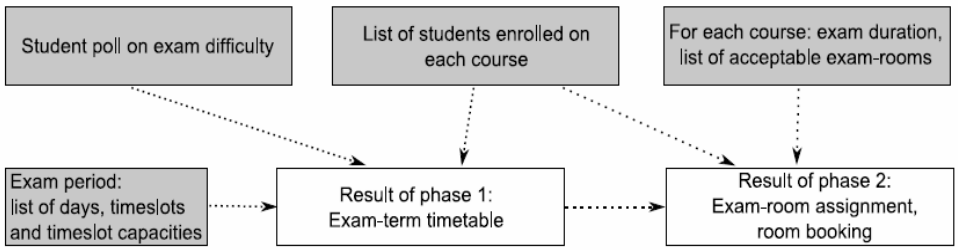
specialisation and elective courses, students are enrolled in many different combinations of courses making the exam scheduling rather difficult. At the same time, we are not interested in just finding some possible timetable that would allow all students to take exams for the enrolled courses. Instead, we are interested in finding a *good-quality* timetable, which for as many as possible students has exams dispersed through the entire exam period and which interleaves more difficult exams with the less difficult-ones. The final result of this timetabling process is a term for each course in which the exam will be held and a set of assigned and booked exam-rooms. This way, students will have enough time to review the course material before taking the exam, and course staff will only have to produce and publish a schedule of students for assigned exam-rooms (for which *Ferko* offers a built-in support, so the whole process requires several mouse clicks).

To create such a schedule, the scheduling process is divided into two phases:

1 exam-term allocation

2 exam-room assignment, as shown on Figure 3.

Figure 3 Exam-scheduling process is divided into two phases: first results with exam-term timetable and second with room-allocations



4.1 Course exam-term, allocation

In order to create course exam-term timetables, we need a list of days which make the exam period, divided into available time slots, a list of students enrolled on each course and an estimate of the exam difficulty for each course. The first two are obtained from faculty administration. To obtain the third, we have setup a web-based system which enables students to fill in a poll after they finish the semester in which they can express their opinion on the difficulty of each exam they took. This data is then used in the following year as estimation of course exam difficulty.

Having all the necessary data, we define the problem of finding a course exam-term timetable as combinatorial optimisation problem of assigning single time slot (i.e., term) to each course, with additional hard and soft constraints. We define the following hard constraints:

- no exams should be scheduled at the same time if they share enrolled students (student conflicts)
- for each available term, multiple course exams can be scheduled, but term capacity cannot be exceeded

and the following soft constraints:

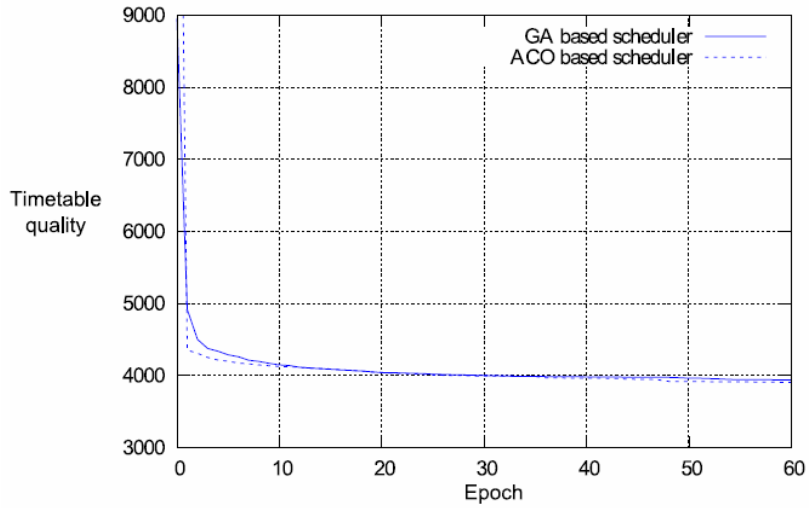
- it is bad for a student to have more than one exam on the same day
- it is bad for a student to have more exams on two (or more) consecutive days
- it is bad for a student to have more difficult exams close to each other.

Soft constraints are handled by multiplying the number of violations with the appropriate factor, and then summing them up. For the difficulty of exams, we have defined an interval [0.2, 2] which determines the exam difficulty based upon obtained student poll results. This measure is then used as a factor in the exam penalty calculation. Since the measure of difficulty is not a binary feature, it could only be included as a soft constraint. The goal of scheduling is to find such a timetable (which satisfies all of the hard constraints) from a set of all valid timetables which has the best quality (minimal violation sum).

To solve this problem, we have developed two implementations based on evolutionary computation algorithms. One is based on the genetic algorithm (Cupic et al., 2009), and the other one on ant colony optimisation. At our institution, exam schedules created this way have now been in use for two years. In each exam period, two exam weeks (see Figure 1) are separated in a total of 40 non-overlapping terms in which 130 to 140 courses are scheduled.

A typical run of both schedulers is shown on Figure 4. The quality of the obtained schedules is reasonably high. In the shown example, the majority of students on the first year of the undergraduate study programme are enrolled in five courses, and the created schedule had exams distributed on Wednesday and Friday of the first week, and on Monday, Wednesday and Friday of the second week.

Figure 4 Average progress of genetic algorithm compared to ant colony optimisation algorithm for exam-term timetabling problem (see online version for colours)



Note: Both algorithms are capable of finding good-quality solutions.

Make-up exams (scheduled at the end of semester) require separate handling. Since the available time is only half of the time available for the final exams, courses are scheduled with more density. However, considering the fact that only a moderately small number of

students is actually present on make-up exams, this is not a problem. During the make-up exam scheduling, care must be taken not to schedule a make-up exam for a course too close to the final exam of the same course.

4.2 Exam-room, assignment

Once the exam-term timetable is created and after the late enrolled students have been processed (so that the number of students on each course is stable), phase 2 is started. Its goal is to create a room assignment that will allow all exams to take place that will utilise as little as possible course staff required for exams and that will have exam rooms for each course as geographically close as possible. Additionally, for each course additional requirements of course staff must be taken into account. To this end, before starting phase 2, course staff can specify additional constraints in *Ferko*, such as the maximal number of students they will accept for each available room. This is then formulated as an optimisation problem for which a genetic algorithm based solution is implemented. Since the exam time slots in which exams were scheduled are non-overlapping, this problem can be solved one time slot at a time. Once the room assignment is done, the complete exam-schedule is published in *Ferko*, and the rooms are booked through the institution room-booking system. A detailed discussion on this subject can be found in Cupic et al. (2010).

5 Late student lecture group assignment

Assigning lecture groups to students who enrolled late to one or more courses is a combinatorial optimisation problem. Since late enrolled students and their choices were unknown at the time of the regular lecture schedule creation, lecture scheduling and division of students into groups was performed with respect to regularly enrolled students. After the lecture schedule is created, student groups, lecture times and room assignments are fixed, leaving only a small number of available places in each lecture group. Performing greedy group assignment (during the enrolment of each student) would fill those places in a non-optimal way, possibly disabling other students to enrol the required courses. To remedy for this, lecture group assignments for late-enrolled students are deferred for some time. Then, an evolutionary-based algorithm is employed with a task to find optimal lecture group assignments. This algorithm cannot move regularly enrolled students among groups. Its task is to schedule late enrolled students by adding them to existing groups. Each assignment is then evaluated with respect to several constraints.

Constraints are divided into:

1 group capacity constraints

2 no-conflicts constraint

3 no-gaps in schedule.

Group capacity constraints are the result of lecture scheduling and lecture room assignments, and are of the following form.

```
count (1 / 1.01) + count (1 / 1.03) <= capacity (r_1)
count (1 / 1.02) + count (1 / 1.04) <= capacity (r_2)
count (2 / 1.01) <= capacity (r_2)
count (2 / 1.02) <= capacity (r_2)
…
```

For example, the first constraint requires that on the course with `ID = 1` the sum of the number of students in groups 1.01 and 1.03 does not exceed the capacity of room `r_1`. The second constraint requires that on the same course the sum of the number of students in groups 1.02 and 1.04 does not exceed the capacity of room `r_2`, etc. The *no-conflicts constraint* requires that lecture group assignment must be performed in such a way which will enable all of the late-enrolled students to follow lectures of enrolled courses, i.e., there must be no conflicts in students' lecture schedules. For example, if a group must be assigned to a student that enrolled the course with `ID = 1`, there would be four possible group assignments on that course: group `1.01`, group `1.02`, group `1.03`, or group `1.04`. From the presented group capacity constraints we can conclude that groups `1.01` and `1.03` have lectures for the course with `ID = 1` at the same time; the same is true for groups `1.02` and `1.04`, so, in effect, the number of choices for that student and that course is actually only two. In order to check this constraint, an algorithm must be provided with a detailed lecture schedule for all lecture groups and all courses.

Finally, the *no-gaps in schedule*-constraint requires that lecture group assignment must be performed in such a way that the resulting lecture schedule for each student does not contain gaps during any day (for example, to have scheduled lectures on Monday from 8 AM to 11 AM and then from 1 PM to 3 PM with a gap between 11 AM and 1 PM).

Unfortunately, for the problems we faced, all of those constraints could not be satisfied. So we defined a vector [`OCM`, `Gaps`] representing the quality of lecture group assignment. The pseudocode on Figure 5 explains the evaluation procedure. For lecture schedule conflicts a period of two consecutive lecture weeks has been analysed (since lecture schedules alternate).

As it can be seen from the pseudocode, the first component of the evaluation vector is the sum of conflicts in students' lecture schedules and additional overflows caused by the optimisation algorithm. Namely, in rare cases, it is possible that after the lecture scheduling some of the groups are already over-capacitated. For each group `g`, the function `initial_overf low(g)` returns this overflow (or zero, if no overflow occurred). When evaluating the quality of group assignment, for each group we calculate only the increase of overflow (denoted `addedOverflow`) and if that value is positive, its square is added to the total amount of overflows. Non-linear mapping is chosen in order to encourage more smaller overflows over few large overflows. The second component of the evaluation vector is a value which is proportional to the total amount of conflicts in lecture schedules (denoted as `Gaps`).

Based on the described evaluation procedure, we developed a multi-population tournament-based genetic algorithm with ring-topology and migration of best solution into next population after each epoch. One epoch was defined as 200,000 reproductions (applications of crossover and mutation operators). Each population was run in isolation for one epoch. Then, a best solution from each population migrated into its neighbouring

population, and the process was repeated. We typically used eight populations, each consisting of 2,000 individuals. Solution comparison is implemented by first comparing the first component of evaluation vector. Only if first components are identical, the second component was inspected. This way algorithm will first try to find such assignment in which a number of conflicts and overflows is minimal, and only then gaps will be minimised.

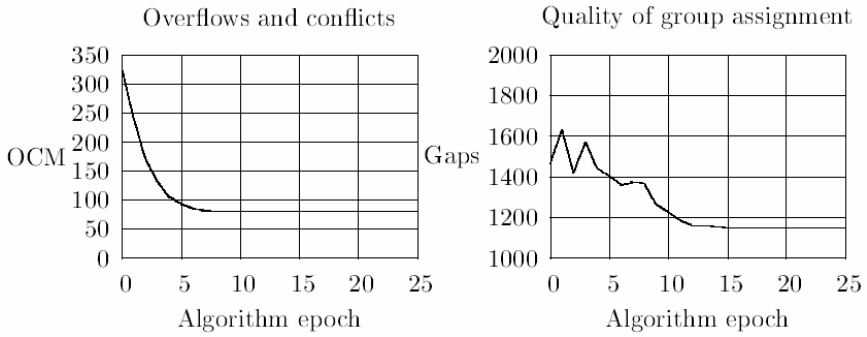
For illustration, Figure 6 shows the run of this optimiser on the problem of assigning lecture groups to 82 late enrolled students which compete for 20 different courses, with 133 requirements for lecture group assignments. For this problem, there were 107 active group capacity constraints.

As can be seen from Figure 6, the algorithm finished with `OCM = 80`. Detailed analysis showed that there were two group capacity constraints where the algorithm increased the overflow by 1 and three group capacity constraints where the algorithm increased the overflow by 2 (giving in total 14). There were 33 hours of conflicts (giving $2 \cdot 33 + 14 = 80$) distributed among 11 students in a period two weeks. In this assignment, there are 575 hours of gaps distributed among 62 students over a period of 10 days, giving an average of 0.93 hours per student per day. The lecture schedules for the remaining 20 students contained no gaps.

Figure 5 Pseudocode of the function for evaluation of the lecture group assignment problem

```
conflicts = 0, overflows = 0, gaps = 0
for each assigned group g
   overflow = number_of_students(g) - capacity (g)
   addedOverflow = overflow - initial_overflow (g)
   if addedOverflow > 0
      overflows += power(addedOverflow, 2)
   end if
end f or
for each late enrolled student s
   for each analysed day d
      for each conflict c of s in d
         conflicts += 2 * duration ( c)
      end for
      for each gap g of s in d
         gaps += 2 * duration(g)
      end for
   end for
end for
OCM = conflicts + overflows
Gaps = gaps
```

Figure 6 Progress and final results for group assignment scheduler, (a) amount of group overflows and student schedule conflicts (b) amount of gaps in created schedules



(a) (b)

6 Laboratory scheduling

Laboratory scheduling is one of the most complex scheduling problems tackled by us so far, due to the large number of constraints. To enable each course to express its requirements, a new module was added to *Ferko*, which is a wizard-like web-based application in which course staff can specify the number of laboratory exercises, acceptable laboratory rooms, number of teaching staff members required per room, number of students per room, acceptable days for each exercise, the maximal number of rooms and the maximal number of teaching-staff members which can be allocated in parallel, requirements for specific programme licences, required ordering of laboratory exercises if more than one exercise must be scheduled in the same week and many others. Additionally, since laboratory exercise scheduling is performed after the lecture schedules are created, the lab-exercise scheduler must be equipped with individual lecture-schedules of each involved student, and the produced lab-exercise schedule must have no conflicts with lecture schedules.

Based on those requirements, two algorithms were developed for schedule creation: one based on the genetic algorithm and one based on ant colony optimisation, which are described in Bratkovic et al. (2009) and Matijas et al. (2010). The time needed for a single person to operate these algorithms and to create the required lab-schedules for lecture weeks 3 to 13 is typically between 12 and 15 working days, so the creation and publishing of the lab-exercise schedule for the entire semester is typically done in the fifth lecture week. Once created, lab-schedules are published in *Ferko*, where course staff can use them to obtain a list of students for each lab-term.

7 Make-up assignments scheduling

In order to assist course staff members in creating student's schedules for make-up assignments, such as laboratory exercises, lectures, quizzes, etc., a number of additional tools have been created: a simple scheduler, a multi-term scheduler, a team scheduler and a new *Ferko* module called *FerSched*.

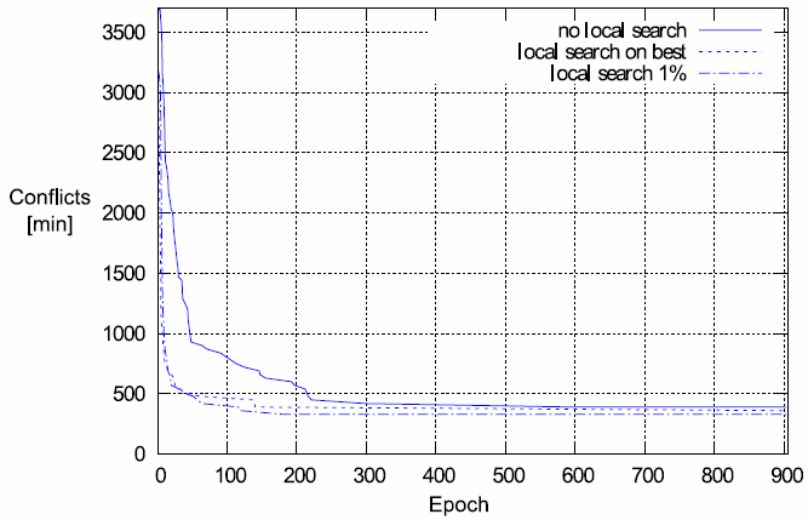
7.1 Console-based schedulers

To tackle scheduling problems which are not directly supported by previously described schedulers, we have developed three command-line tools that offer an easy way to experiment with possible schedule requirements. All of those tools take schedule requirements as input from a textual file and a list of students' prescheduled activities from *Ferko* and generate output which can be published in *Ferko*.

Simple scheduler requires a user to specify a set of terms with term capacities (then number of students that can be assigned to that term) and a set of students to be scheduled. Each student is scheduled into one of the provided terms taking into account students preexisting schedule. This tool is one of the simplest ones, and is implemented as a simple random-based iterated search.

Multi-term scheduler can be used when a set of students must be split into a predefined number of groups and when each group must be assigned to multiple predefined terms. For example, let us assume that a set of 300 students must be divided into three groups: *Gl*, *G2* and *G3*. *Gl* will have two lectures: Monday from 8 AM to 9 AM and Tuesday from 8 AM to 9 AM, *G2* will have two lectures: Monday from 9 AM to 10 AM and Tuesday from 9 AM to 10 AM, and finally, *G3* will have two lectures: Monday from 10 AM to 11 Am and Tuesday from 5 PM to 6 PM. This tool is also implemented as a simple random-based iterated search.

Figure 7 Average progress of ClonAlg algorithm for team-scheduling problem (see online version for colours)



Note: Scheduler stabilises below 500 minutes of conflicts.

Team scheduler is used in the following situation. Students on a course are divided into smaller teams and each team has a course staff member assigned. Each team has assigned only one course staff member, but a single course staff member can be assigned to multiple teams. All of this is determined in advance. Given this information, students' preexisting schedules and a list of possible time slots for laboratory exercises (or other activities) a schedule must be created in which every team is assigned one available time

slot, multiple teams can be assigned into the same time slot (up to its capacity) and no two teams having the same preassigned course staff member can be assigned into the same time slot. This is a rather difficult combinatorial optimisation problem for which an algorithm based on the clonal selection algorithm (a member of artificial immune algorithms) (De Castro and Von Zuben, 2000; De Castro and Timmis, 2002; Cutello and Nicosia, 2005) is created. For the last several semesters, this scheduler has been successfully used on several courses requiring such scheduling.

The progress of the *team scheduler* used to schedule 47 teams in a single day is shown on Figure 7. In this problem, the average number of students per team was 8, and there were only 5 course staff members, each having an average of 9 teams assigned. The best found solution had 330 minutes of conflicts (11 students each having 30 minutes of conflicts).

7.2 FerSched module

The *FerSched* module is composed of two main components: a Java Applet (Java Applet Technology, 2010) for defining the schedules, and an application which can be run locally in order to create schedules from the input data.

In the applet, the user has the possibility to define detailed constraints of each schedule. These constraints can be defined on the following three levels, listed hierarchically: the plan, event and term levels. On each of these levels, there are three main constraints which can be defined, and regard the availability of students, time spans and rooms. Naturally, if a constraint is defined on one of the levels it becomes locked, and therefore cannot be defined on the other two levels, and levels of lower hierarchy inherit in a way the constraints from the hierarchically higher level.

In order to provide an illustrative example, the scheduling of laboratory exercises for student assistants for the *Digital logic* course will be used. Firstly, the user defines all the affected students on the plan level (in the example, there are 51 of them). As there will be two events and both events should take place in the same week, the available time span could also be defined on the plan level. Secondly, the user creates two events: the theory event, and the practice event. Since these events differ in length, it is necessary to define their durations (two hours for theory, and one hour for practice). As for every student the theory event should occur one day before the practice event, we set this as preconditions for the events. The next step is to define the available rooms for each of the events. Since we do not want to overcrowd the computer rooms with students we can limit the room capacity for the rooms assigned to practice to 18 students. Finally, the user manually defines the number of terms allowed for each of the events (later referred to as the given distribution), selecting 1 term for theory and 3 or practice. This yields a valid plan, ready for processing. If any errors are found, they are reported to the user before the creation of the final plan.

The plan level serves to represent the schedule as a whole, and thus every solution will eventually be presented in form of a plan. A plan is composed of events that is make-up assignments which have their own duration, distribution and preconditions, which are in turn composed by terms. A term is an actual realisation of an event, meaning that it is uniquely defined by its start time, duration, assigned room, and a list of assigned students.

The plan level offers the user the possibility to define only the main constraints such as the selected students, the available time-span and the available rooms. If some of these

constraints are set on the plan level, they are inherited by the levels lower in the hierarchy. The event level offers the possibility of defining more specific constraints, apart from the main ones. These include the duration, distribution and preconditions of each event. The duration denotes the maximal duration of any term which is the realisation of the event. The distribution can be set as random or given. A random distribution enables the user to define the range of values which can be used as the number of terms and lets the algorithms choose the adequate number of terms, while the given distribution offers the possibility to manually create the terms for an event and also define term-specific constraints. The preconditions determine all the events whose terms must occur before the terms of this event, and also the minimal time margin between events. The term level, similarly to the plan level, allows only the definition of the main constraints. It is possible to define the constraints at term level only if the given distribution is used for the event.

After a schedule has successfully been defined, the applet proceeds to retrieve from *Ferko* all the needed data concerning the student and room occupancies. When this process is successfully completed, the module create an executable Java application (.jar) which can be downloaded and run locally to create the schedule.

The schedule creation application processes the occupancy data provided by *Ferko* and attempts to create a valid schedule (the term will be defined later in the text) with the aid of evolutionary algorithms. The algorithms following algorithms are implemented thus far: bee colony optimisation (Chong et al., 2006), clonal selection algorithm (Brownlee, 2005), genetic algorithm (Bratkovic et al., 2009), harmony search (Geem, 2009), particle swarm optimisation (Clerc, 1999) and stochastic diffusion search (Meyer et al., 2006).

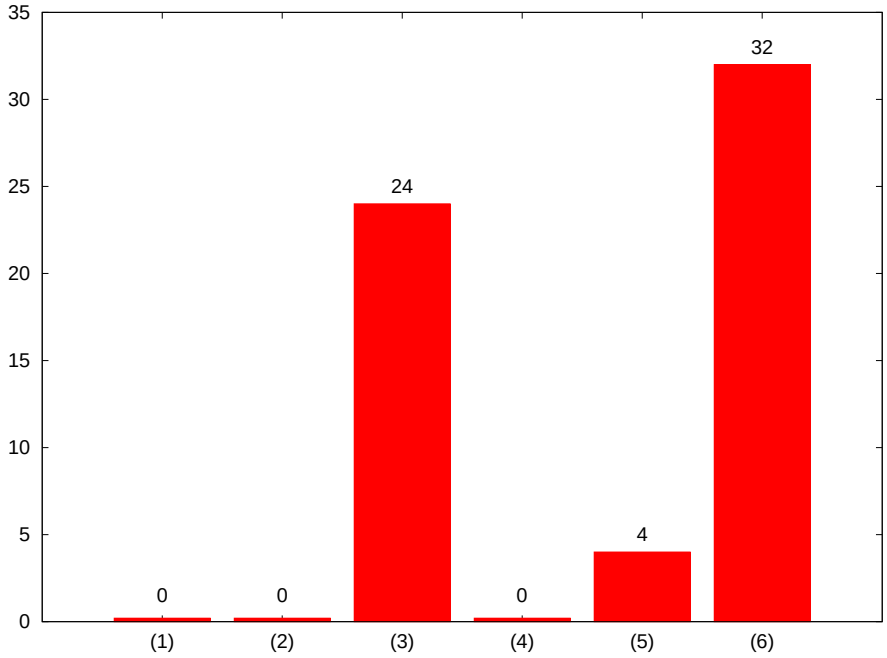
In order for the different algorithms to be able to exchange partial solutions, they must all implement the same interface. The interface defines common operations which every algorithm has to be able to perform, such as start, stop, return the currently best solution, include a solution from another algorithm into the population, etc. Solution exchange is a mechanism which provides the algorithms with quality solutions which enhance and diversify their population. The algorithm which is used to execute the next iteration of the schedule generation process is randomly chosen based on roulette wheel selection, where the intervals corresponding to the probability of each algorithm to be selected are set by the user when starting the schedule generation process.

During the schedule creation process, the user is provided with live feedback on the progress of the solutions by means of a *JFreeChart* (2010) bar chart (as shown in Figure 8) whose bars represent each of the following constraints used to rank the solutions: unsatisfied preconditions, room conflicts, student conflicts, overcrowded rooms, number of terms and vacant places. The constraints are divided into hard and soft constraints. The hard constraints (unsatisfied preconditions, room conflicts, student conflicts, overcrowded rooms) represent the constraints which must be satisfied. A schedule is considered valid if and only if all hard constraints are satisfied. The soft constraints (number of terms, vacant places) are a measure of quality of the solution, and are used to further rank all valid schedules.

Unsatisfied preconditions are expressed by events which do not satisfy their preconditions, i.e., their terms do not respect the minimal time margin from the terms of the event which is the precondition. Room conflicts are represented by the number of 15-minute intervals in which a single room is occupied by two distinct terms. Similarly, student conflicts are expressed as the number of 15-minute intervals during which a

student is assigned to two distinct terms. The overcrowded rooms constraint expresses the number of rooms where the number of assigned people exceeds the room capacity. The numbers of terms and vacant places, as their names say, denote the total number of terms in the schedule and the total number of vacant places in all assigned rooms. The order in which the hard constraints are presented also defines their priority when comparing, i.e., a schedule with fewer unsatisfied preconditions and some room conflicts is considered better than a schedule with more unsatisfied preconditions but no room conflicts. If all of the hard constraints for two schedules have the same values, the ranking is then performed by comparing the weighted sums of the soft constraint values.

Figure 8 An example of the feedback provided to the user during FerSched schedule creation (see online version for colours)



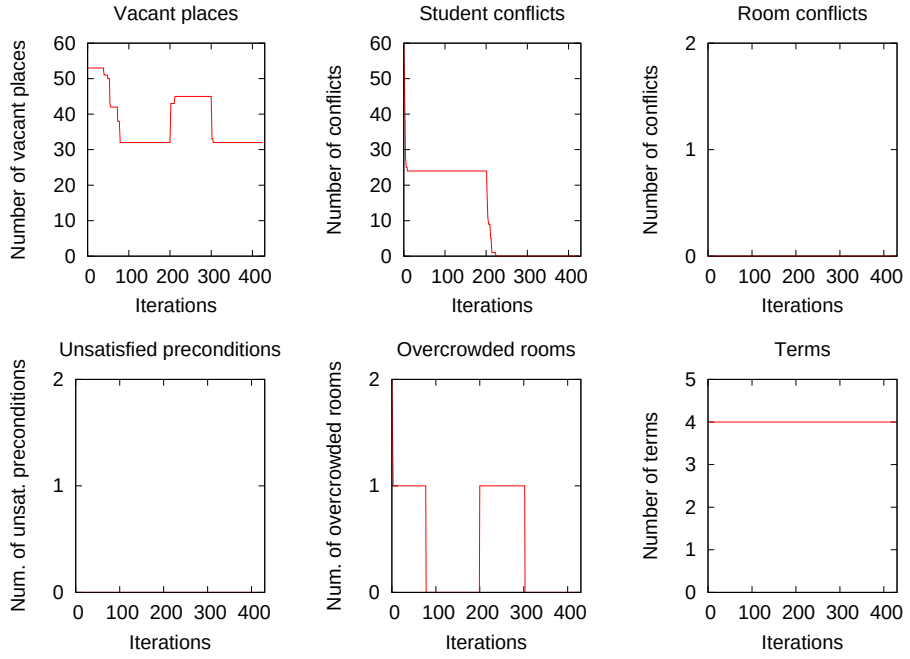
Notes: 1 = Unsatisfied preconditions; 2 = room conflicts; 3 = student conflicts; 4 = overcrowded rooms; 5 = number of terms; 6 = vacant places; [l–4] = Hard constraints; [5, 6] = Soft constraints

The user can stop the scheduling process as soon as he is satisfied with the result shown as feedback on the screen, and is then provided with an overview of the schedule, including the remaining student conflicts in case a valid schedule could not be generated. The user is also offered the option to export the generated schedule into XML, which can then be transferred to *Ferko* and thus create a new student assignment for all of the assigned students, which will immediately be available in the students' personal calendars. The room reservations are also handled automatically.

Figure 9 shows a graph representing the change in constraint values through iterations for the example problem. The most notable decrease is present in the number of student conflicts, which usually decreases exponentially.

The algorithms which have proved to work successfully together in terms of solution exchange are the genetic algorithm, clonal selection algorithm and harmony search. The remaining algorithms have longer initialisation times, and thus produce better results when used alone.

Figure 9 Constraint change through iterations during a single run of FerSched schedule creation (see online version for colours)



As far as the conduced tests show, the algorithms are relatively successful in schedule creation. The distribution of 51 students in 2 events, having 1 and 3 terms, respectively takes approximately 8 seconds to complete, while a schedule containing more than 600 students and 4 events with 4 terms each presents a notable decrease in conflicts in approximately 45 seconds.

8 Conclusions and future work

In this paper, we have described a number of situations constantly being presented to faculty staff members which are, in fact, various cases of scheduling problems. Formally, most of those problems are, from the computation complexity point of view combinatorial NP-hard problems.

As part of the *Ferko* project, a number of algorithms are being developed in order to tackle these problems and to help create better conditions for students, and faculty and course staff members. Some of the developed algorithms are presented in this paper. Although scheduling problems can be tackled by a number of different algorithms, evolutionary computation offered us an elegant, consistent and exciting framework for

the implementation of various schedulers which are now regularly used at our institution, and which provide good-quality schedules.

It is important to realise that schedules created this way significantly helped us to conduct the alignment with the Bologna process, to organise laboratory exercises in prerequisite-based course enrolment and to try to create better and less stressful exam timetables.

From the students' standpoint, the development of such a scheduling system is beneficial because it enabled students to have all of their activities scheduled in one place. This is important because it gives students the possibility to overview their activities and assignments at all times. Many students at universities which have not adopted a similar scheduling model have a hard time following their activities on multiple systems in use by the university. Also, the use of a comprehensive scheduling system at university level allows for the creation of higher quality schedules with respect to all of the student's obligations, which would not be possible if department-specific systems were used for different courses.

At a single course level, significant efforts were also made in order to help course staff to better and more easily organise all course-related events. Specifically, algorithms were developed and integrated into *Ferko* which now allow course staff members to relatively easily schedule various kinds of make-up assignments without causing conflicts with previously scheduled students' obligations.

As part of future work, there is much room left for enhancements: either from the standpoint of expressiveness, or from the standpoint of efficiency, quality, and better and more usable integration with *Ferko*. All of those are areas of future research.

Acknowledgements

The authors would like to acknowledge the students Mihej Komar and Toni Pivcevic for their work on the application for gathering laboratory schedule specifications and new exam-term scheduler, the students Zlatko Bratkovic, Tomislav Herman, Vjera Omrcen, Vatroslav Dino Matijas and Goran Molnar for their work on the laboratory schedulers, the team of students who created the *FerSched* module, and all the others participating in the development of *Ferko* and accompanying modules.

References

Abdullah, S., Ahmadi, S., Burke, E.K., Dror, M. and McCollum, B. (2006) 'A tabu based large neighbourhood search methodology for the capacitated examination timetabling problem', *Journal of Operational Research Society*, Vol. 58, No. 11, pp.1494–1502.

Adamidis, P. and Arakapis, P. (1997) 'Weekly lecture timetabling with genetic algorithms', in *Proc. of PATAT-97*, pp.115–122.

Affenzeller, M. and Wagner, S. (2009) *Genetic Algorithms and Genetic Programming, Modern Concepts and Practical Applications*, CRC Press, Boca Raton.

Beligiannis, G.N., Moschopoulos, C.N., Kaperonis, G.P. and Likothanassis, S.D. (2008) 'Applying evolutionary computation to the school timetabling problem: the Greek case', *Computers and Operations Research*, Vol. 35, No. 4, pp.1265–1280.

Blakesley, J.R. (1959) 'Automation in college management', *College and University Business*, Vol. 27, No. 5, pp.39–44.

Bratkovic, Z., Herman, T., Omrcen, V., Cupic, M. and Jakobovic, D. (2009) 'University course timetabling with genetic algorithm: a laboratory exercises case study', in *Proceedings of the 9th European Conference, EVO COP 2009'*.

Brownlee, J. (2005) 'The clonal selection classification algorithm (csca)', Technical report, Centre for Intelligent Systems and Complex Processes, Faculty of Information and Communication Technologies, Swinburne University of Technology, Victoria, Australia.

Celcat online (2010) Available at http://www.celcat.com.

Chong, C., Low, M.H., Sivakumar, A. and Gay, K. (2006) 'A bee colony optimization algorithm to job shop scheduling', in *Proceedings of the 2006 Winter Simulation Conference*, pp.1954–1961.

Clerc, M. (1999) 'The swarm and the queen: towards a deterministic and adaptive particle swarm optimization', in *Proceedings of the 1999 ICEC*, pp.1951–1957.

Colorni, A., Dorigo, M. and Maniezzo, V. (1990) 'A genetic algorithm to solve the timetable problem', Technical Report 90-060, Politecnico di Milano.

Cumming, A. (1997) 'Timetabling with neeps and tatties', available at http://www.soc.napier.ac.uk/~andrew/dneeps/.

Cupic, M., Golub, M. and Jakobovic, D. (2009) Exam timetabling using genetic algorithm, in *Proceedings of the 31st International Conference on Information Technology Interfaces*, Cavtat, Croatia, pp.357–362.

Cupic, M., Golub, M. and Jakobovic, D. (2010) 'Applying AI-techniques as help for faculty administration – a case study', in *Central European Conference on Information and Intelligent Systems 2010*, Varazdin, Croatia.

Cutello, V. and Nicosia, G. (2005) 'The *Clonal Selection Principle for In Silico and In Vitro Computing*, Edited by L.N. De Castro and F.J. Von Zuben, Idea Group Publishing, Chapter 4, pp.104–146.

De Castro, L.N. and Timmis, J. (2002) *Artificial Immune Systems: A New Computational Intelligence Approach*, Springer-Verlag, Great Britain.

De Castro, L.N. and Von Zuben, F.J. (2000) 'The clonal selection algorithm with engineering applications', in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00), Workshop on Artificial Immune Systems and Their Applications*, Las Vegas, Nevada, USA, pp.36–39.

De Jong, K.A. (2006) *Evolutionary Computation*, MIT Press, Cambridge.

Deb, K. (2009) *Multi-Objective Optimization using Evolutionary Algorithms*, Wiley, New York.

Dorigo, M. and Stiitzle, T. (2004) *Ant Colony Optimization*, MIT Press, Cambridge, MA.

Eberhart, R. and Kennedy, J. (1995) 'A new optimizer using particle swarm theory', in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, pp.39–43.

Evolvera (2010) Available at http://www.evolvera.se.

Feoktistov, V. (2006) *Differential Evolution. In Search of Solutions*, Springer, New York.

Geem, Z.W. (2009) *Music-inspired Harmony Search Algorithm Theory and Applications*, Springer-Verlag, Berlin.

Gueret, C., Jussien, N., Boizumault, P. and Prins, C. (1995), 'Building university timetables using constraint logic programming', Selected papers from the *First International Conference on Practice and Theory of Automated Timetabling*, pp.130–145.

Infosilem (2010) Available at http://www.infosilem.com.

Java Applet Technology (2010) Available at http://java.sun.com/applets/.

JFreeChart (2010) Available at http://www.jfree.org/jfreechart/.

Kambi, M. and Gilbert, D. (1996) 'Timetabling in constraint logic programming, in *Proc. 9th Symp. on Industrial Applications of PROLOG*, pp.320–334.

Lim, A., Ang, J.C, Ho, W.K. and Oon, W.C. (2000) 'A campus-wide university examination timetabling application', in *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pp.1020–1015.

Mackworth, A.K. (1977) 'Consistency in networks of relations', *Artificial Intelligence*, Vol. 8, No. 1 pp.99–118.

Mamede, N. and Rente, T. (1997) 'Repairing timetables using genetic algorithms and simulated annealing', in *Proc. of PATAT-97*, pp.187–204.

Matijas, V.D., Molnar, G., Cupic, M., Jakobovic, D. and Dalbelo Basic, B. (2010) 'University course timetabling using ACO: laboratory exercises case study', in *14th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, Cardiff, Wales, UK.

Meyer, K.D., Nasut, S.J. and Bishop, M. (2006) 'Stochastic diffusion search: partial function evaluation in swarm intelligence dynamic optimization', *Studies in Computational Intelligence*, Vol. 31, pp.185–207, Springer, Berlin/Heidelberg.

Müller, T. (2009) 'Itc2007 solver description: a hybrid approach', *Annals of Operations Research*, Vol. 172, No. 1, pp.429–446.

Montes de Oca, M.A., Stiitzle, T., Birattari, M. and Dorigo, M. (2009) 'Frankenstein's PSO: a composite particle swarm optimization algorithm', *IEEE Trans, on Evolutionary Computation*, Vol. 13, No. 5, pp.1120–1132.

Ozcan, E., Misir, M., Ochoa, G. and Burke, E.K. (2010) 'A reinforcement learning – great-deluge hyper-heuristic for examination timetabling', *International Journal of Applied Metaheuristic Computing*, Vol. 1, No. 1, pp.39–59.

Price, K.V., Storn, R.M. and Lampinen, J.A. (2005) *Differential Evolution. A Practical Approach to Global Optimization*, Springer-Verlag, Berlin.

Salwach, W. (1997) 'Genetic algorithms in solving constraint satisfaction problems: the timetable case', *Badania Operacyjne i Decyzje* (Poland), Vol. 2, pp.55–63.

Thompson, J. and Dowsland, K.A. (1996) 'Variants of simulated annealing for the examination timetabling problem', *Annals of Operations Research*, Vol. 63, No. 1, pp.105–128.

UniTime | University Timetabling (2010) Available at http://www.unitime.org.