Applying AI-techniques as Help for Faculty Administration – A Case Study

Marko Čupić, Marin Golub, Domagoj Jakobović
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3, 10000 Zagreb, Croatia
{marko.cupic, marin.golub, domagoj.jakobovic}@fer.hr

Abstract. *For every university at the beginning of each semester, there are many organizational problems which have to be solved by faculty administration. Some of them are division of late-enrolled students into lecture groups, and creation of room schedule for examinations. Both of these problems can have significant influence on the students' ability to attend enrolled courses, and on the number of teaching staff required for students examination. In this paper we will present both of these problems, and their solution using genetic algorithms.*

Keywords. student scheduling, room schedule, genetic algorithms

1 Introduction

In medium and large faculties, there are many complex organizational issues that need to be solved – lecture, laboratory and exam scheduling, just to name a few. Lecture scheduling, course enrollment administration and assignment of students into lecture groups are three interdependent processes. Allocation of rooms for exams is another example of important task, especially with densely populated terms in which available room capacities are nearly exhausted. In this paper we will show two such problems successfully tackled by genetic algorithms: division of late-enrolled students into lecture groups, and creation of room schedule for examinations. Both of those problems are instances of hard combinatorial problems. Solving them by hand is extremely hard, and solutions are often very far from being optimal. However, it is important to find solutions that are as good as possible, since the former problem influences the ability of late-enrolled students to attend lectures of enrolled courses, and the latter problem can have significant influence on a number of teaching staff needed for student examination.

Using todays computing power to assist in finding good solutions can be more difficult than anticipating at the first glance. Namely, both of those problems are hard combinatorial problems for which we can not write the exhaustive search procedure which will complete in acceptable time frame.

Evolutionary algorithms [1, 2] are metaheuristics which can rather successfully cope with this kind of problems. It is important to note that evolutionary algorithms can not provide us with a guarantee of solution optimality, when working under tight time constraints. However, they can often produce a reasonably good solution that is not far from the optimal one. Under the umbrella of evolutionary computation there are many algorithms, such as genetic algorithms [3], particle swarm optimization [5, 6], ant colony optimization [7], artificial immune systems [8, 9, 10] and many others. We have decided to tackle both of described problems using genetic algorithms, since they offer rather straight-forward means for solution representation and multi-objective optimization.

This paper is organized as follows. In section 2 we will introduce the room scheduling problem and describe the relationship between existing exam scheduling and here defined room scheduling. Implementation of algorithm for the room scheduling based on genetic algorithm will be presented and elaborated. In section 3, additional scheduling problem – assignment of late enrolled students to lecture groups – will be defined, and applied genetic algorithm based method will be described. In section 4 a conclusion and future work directions are given.

2 Room scheduling

At authors institution, each semester is divided into four examination periods: two periods are for mid-term exams, one is for final exams, and one is for make-up exams. Schedules for those periods are produced using genetic algorithms, as reported in [4]. In this paper we will focus on additional scheduling problem that is performed after the courses are assigned to exam time-slots: room scheduling. Room scheduling is an assignment problem in which for each exam time-slot rooms must be scheduled to courses while minimizing the required number of teaching staff and having adequate quality (*quality* will be defined later). This is also NP-hard problem. To exemplify, let us consider a simple room scheduling scenario in which there are 30 rooms which have to be scheduled among 10 courses. We have to check each possible scenario. The first room can be given to each of 10 courses, then the second room can be given to each of 10 courses, and so on, which gives us a total of $10 \cdot 10 \cdot \dots \cdot 10 = 10^{30}$ combinations. If it takes 1 μs to check a single combination, the exhaustive search procedure, which checks each of those combinations, will finish its work in approximately $3.2 \cdot 10^{16}$ years.

What is the relationship between room and exam scheduling, and what are the related problems at authors institution? First, the exam schedule is created and published. The generated schedule for each course contains only a time and duration of the exam. For the exam scheduling purposes, each day of examination periods is divided into 4 disjunct time-slots. Courses are then scheduled so that there is no student that is enrolled in two or more courses that are assigned into the same time-slot. Courses are assigned into time-slots taking into account time-slot capacity (a total number of students which can be accepted by all rooms which are available in that time-slot). This approach is far from ideal. For example, it is easy to imagine a situation with a time-slot where there are only 2 large rooms available (70 students each). Described scheduling process can assign three or more smaller courses in that time-slot, as long as the total sum of enrolled students is not greater than 140. However, from the standpoint of course-staff, often room-sharing is not an well-perceived option.

2.1 Exam scheduling related problems

To reduce the probability of such events, we can take two approaches: either to simultaneously generate the exam schedule up to the level of rooms, or to artificially reduce the available rooms capacity by a certain factor, so that in reality, space provided by all available rooms will not be fully occupied. The former approach is problematic, since it drastically increases the search space for an algorithm already operating on a huge search space, and trying to satisfy a number of additional constraints. Instead, we took the second approach. For a typical time-slot, actual student capacity at authors institution is about 1000 students. During the exam scheduling, time-slot capacities were all set to 80% of that number (800 students).

Unfortunately, even this approach during last few years has lead to problems when time-slots were nearly fully occupied. The problem arose when some courses reserved more rooms than expected, in order to be able to make more sparse student schedule (to make cheating more difficult). This, however, left no available rooms for other courses assigned into that same time-slot.

2.2 Increase in required number of staff members

The other important problem is the growth of teaching staff requirements needed for exams. To help with exam organization, a pool of teaching staff members is created. After the exam schedule is created, and after exam-organizers reserve required number of rooms and create student schedules, a total number of teaching staff members required for exams is calculated (denoted with T). Then, if there are N teaching staff members available in teaching staff members pool, each of teaching staff members must be present on $n = T/N$

exams. During several last semesters, we noticed the constant growth of n, which can be attributed to the increase in a number of students present in the system, as well as to misusage of the pool of teaching staff by the staff itself.

2.3 The proposed solution

To amend all of those issues, and to disburden the exam-organizers from the task of manually selecting and reserving rooms and resolving possible room conflicts, the decision was made to create a version of exam schedule that for each time-slot contains preassigned and reserved rooms for each course in that slot. This approach has many benefits.

- **Early room conflict detection.** By creating room schedules centrally, situations such as "two big rooms and three courses" can be detected early, and they can provoke a change in exam schedule, or other adjustments so that additional room required for the third course can be found. And all of that can be dealt with before the schedule is published.
- **Rational room usage.** Care can be taken in advance to create such a room schedule that will enable all courses assigned to the same time-slot to have enough assigned rooms for rational student schedules. Then, if there are available additional rooms, each exam-organizer can reserve additional rooms to make student schedules more sparse. However, initial intention that all courses can have exams can be fulfilled.
- **Rational usage of teaching staff.** During the process of room scheduling, care can be taken to minimize the total number of teaching staff required for all of the exams, based on objective assessments of room requirements. If there are rooms with various ratios s/t, where s is number of students for that room and t number of teaching staff members required in that room, a clever selection of rooms can be made that will minimize total required number of teaching staff members. It is important to note that always selecting rooms with minimal ratio s/t for one course does not guarantee optimal solution since that room is then unavailable for other course that might have a better usage for it.

Actual implementation of described scheduling techniques enabled us to include even additional wishful properties for each room schedule, which in practice generated very promising results.

2.4 Formal problem description

In this section we will provide a formal problem definition.

- Let $T = \{T_1, T_2, ..., T_m\}$ be a set of disjunct time-slots.
- Let $C_i = \{c_{i,1}, c_{i,2}, ..., c_{i,k}\}$ be a set of courses scheduled to time-slot T_i, $T_i \in T$.
- Let $stud(c_i)$ be the number of students enrolled on course c_i.
- Let $R = \{r_1, r_2, ..., r_l\}$ be a set of all existing rooms.
- Let $R_i \subseteq R$ denotes a set of rooms available to time-slot T_i, $T_i \in T$.
- Let $AR_i \subseteq R$ be a set of rooms assigned to course c_i.
- Let $stud(r_i, c_j)$ be the number of students that course c_j is willing to put in room r_i. Note that this means that different courses can decide to fill the same room with different number of students, if that room is assigned to a course.
- Let $staff(r_i, c_j)$ be the number of teaching staff members that course c_j requires to be present in room r_i. Note that this means that different courses can decide to assign different number of teaching staff members in the same room, if that room is assigned to a course.
- Let $building(r_i)$ be the building in which room is situated.
- Let $floor(r_i)$ be the floor on building in which room is situated.

Then, the basic scheduling can be formalized as follows. For each term T_i find a partition of R_i into disjunct subsets $\mathcal{R} = \{R_{i,1}, ..., R_{i,p}, R_{unused}\}$, $p = |C_i|$, $R_{i,j} \cap R_{i,k} = \emptyset, \forall j \neq k$, so that

$\forall c_j \sum_{r \in R_{i,j}} stud(r, c_j) \geq stud(c_j)$. The idea is to decompose all available time-slot rooms into a disjunct subsets of rooms – one subset for each course, and possibly to leave some of available rooms unassigned. The sum of capacities of rooms assigned to each course (as defined by that course) must be equal than or greater than the number of students on that course.

On top of that basic requirement, we added two additional ones. First, for each time-slot T_i the total number of allocated teaching staff members should be minimized:

$$minimize\, f(\mathcal{R}) = \sum_{c_{i,j} \in C_i, r \in R_{i,j}} staff(r, c_j).$$

This will automatically remove all extra rooms, which provide more capacity than needed for any of the courses. The second requirement arose from the practice of larger courses which usually allocate one additional course staff member to visit each assigned room and answer students' questions, several times during exam. Since our institution has four buildings, room schedules that would be scattered throughout the buildings are not desired. So during the room scheduling process we would like to find for each course such a room assignment that will minimize the walking distance for cyclic path that visits each assigned room once per cycle, which is in essence a TSP problem [11]. Since it is well known that TSP belongs to NP-hard class of problems, it was unacceptable to write a procedure that would, in order to evaluate the quality of room schedule, try to solve all accompanying TSP-s (one for each course). The main reason is that when using a genetic algorithm in order to solve the scheduling problem, we must be able to evaluate thousands of schedules per second. And that would not be possible if the evaluation required solutions of TSP problems.

Instead, we decided to simplify things (or to complicate it). Since for each room we had data on room's building and room's floor, we decided to measure a quality of course's room schedule by counting the number of buildings and the number of floors its rooms were located in. The idea was to favor the schedules that for a single course stay on the same floor; for bigger courses use multiple floors of the same building, and only for large courses span across multiple floors and multiple buildings.

Finally, there is one additional requirement that describes the quality of room schedule, which we decided to include - a room preferability. We have three types of rooms: flat classrooms, amphitheater rooms, and computer laboratories. During the exams, all three kinds of rooms are used. However, flat classrooms are the most preferred, since cheating in that kind of rooms is rather difficult. Amphitheater rooms are less preferable, since they allow easier student cheating during the exam. Computer laboratories are least preferable, since students are placed rather close to each other. So the room schedule should have best possible quality, when considering room preferability.

In order to collect all of required data, we enabled each course to adjust two room parameters: the number of students that the course is willing to schedule into a room, and the number of teaching staff members that the course requires to be present in the room.

2.5 Genetic algorithm for room scheduling

We implemented a steady-state genetic algorithm containing a population of 1000 chromosomes. We induced a ring topology into population, and limited genetic operators to work only on closely positioned parents. In order to do so, we defined a parameter *neighborhood* n and set its value to 10. The pseudo code of the algorithm is as follows.

```
func GA(timeslot Ti)
  initPopulation(Ti)
  while(!stoppingCondition) {
    i = selectChromosome(0,popsize);
    j = selectChromosome(i-n,i+n);
    k = selectChromosome(i-n,i+n);
    (p1,p2)=best(i,j,k);
    c = createChild(p1,p2);
    if(better_than(c,currentBest)) {
      stagnationCounter=0;
    } else {
      stagnationCounter++;
    }
    replace worst(i,j,k) with c;
  }
  return currentBest;
```

In each iteration, a random chromosome is selected and then two more chromosomes are selected from its neighborhood. The better two are selected to be parents. Crossover and mutation operators are applied on the parents. Finally, on child a local search was performed. The worst of the three chromosomes initially selected is then replaced. If the child is better than the currently best solution present in the population, stagnation counter is reset; otherwise, it is incremented. Stopping condition is set to *true* when stagnation counter reaches 1,000,000.

Function `GA` is then called once for each time-slot, in order to create time-slot room schedule, since the time slots are nonoverlapping.

The implementation of genetic algorithm for this particular problem is rather straight-forward. There is only one detail left to be explained: the originally presented problem is a clear case of multi-objective optimization (take enough rooms to allow all students to take exam, minimize total number of staff members, maximize quality of schedule in terms of number of floors and number of buildings). To handle multi-objective optimization problems, evolutionary algorithms can work with the principle of domination. Namely the problem which arises in multi-objective optimization is how to compare two solutions? In our case, is it better to have a schedule that for some course requires 10 staff members and spans over three floors, or to have a schedule that for some course requires 11 staff members and is located on a single floor? The domination principle allows such algorithms to avoid such questions, and to provide to the user a selection of various schedules each having different qualities. However, in our case there were priorities which had to be taken into account, so we decided to take another standard approach: to transform multi-objective problem into a single objective one, partially by inducing strict hierarchy among various solution quality measures, and partially by using weighting approach.

The quality of a solution is represented as a four dimensional vector q. The first component ($q[0]$) is the total number of places missing in order for all students to be able to take exam (some courses have not enough assigned rooms). This component should be minimized to 0. The second component ($q[1]$) is the total number of allocated extra-place, which are unused by students. This component should also be minimized, in order to prevent solution in which small courses (e.g., with 15 students) get large rooms (e.g., for 70 students). The third component is the total number of allocated staff members required for the schedule. Finally, the fourth component represents schedule's locational quality and preferability, and is calculated as follows.

Set *preferabilityPenalty* to 0. Then, for each room assigned in a schedule, if it is an amphitheater, increment *preferabilityPenalty* by 9, and if it is a laboratory, increment *preferabilityPenalty* by 15. Set *locationPenalty* to 0. For each course C_i in schedule calculate the number of buildings n_b and the number of floors n_f on which there are rooms assigned to that course. Increment *locationPenalty* by $70 \cdot (n_b - 1)$ and by $23 \cdot (n_f - 1)$. Finally, set $q[3]$ to the sum of *preferabilityPenalty* and *locationPenalty*. Weights and other fixed numbers used for this calculations were empirically determined for authors particular problem. For other problems (different number of buildings, floors etc.) the user can adjust this values to better suit his sense of "schedule quality".

Once the solution is fully evaluated, solution comparison is implemented as follows:

```
compare(s1, s2) {
  if(s1.q[0]!=s2.q[0]) {
    return s1.q[0]!=s2.q[0];
  } else if(s1.q[2]!=s2.q[2]) {
    return s1.q[2]!=s2.q[2];
  } else {
    return s1.q[1]+s1.q[3]
              -(s2.q[1]+s2.q[3]);
  }
```

The comparison method must return a value which is negative if $s1$ is better than $s2$, zero if they are equal, and a positive value otherwise. As can be seen from the algorithm, the most important criteria is to allocate enough space for all of the students. Only when comparing two schedules having equal number of missing places, comparison will check the total number of assigned staff members, and if there is still no difference, comparison will check the sum of extra-allocated places and *preferabilityPenalty* and *locationPenalty*.

2.6 Local search

Local search procedure is implemented as follows. For each course (in a randomly determined order),

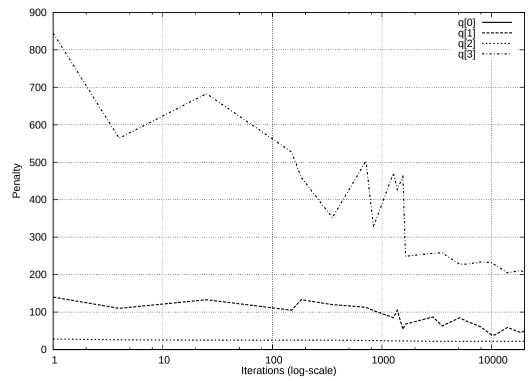

Figure 1: Algorithms performance with local search procedure enabled

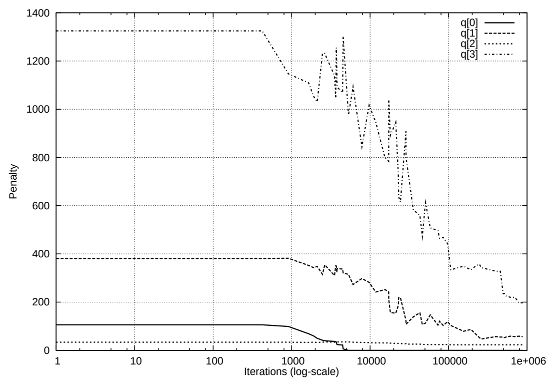

Figure 2: Algorithms performance with local search procedure disabled

if there is extra allocated place, an attempt is made to randomly deallocate some of rooms, but preserving enough places for the students (no shortage will be created). Then, for each course, if there is not enough allocated places, an attempt is made to randomly allocate additional rooms (if possible).

Difference in algorithm performance with and without local search is clearly illustrated on Figure 1 and Figure 2. To obtain even a comparable results, the algorithm with local search disabled requires about 100 times more iterations (about 10^6 iterations versus 10^4), making the search procedure a *must-have* if time-behavior is important. On Figure 1, $q[0]$ is not visible since it falls to zero in first iteration.

The results obtained for first exam period of current semester are encouraging. Compared with the same semester of previous academic year (in which the scheduling was done by hand), a number of course enrolments have risen for about 6%. However, using the schedule we generated the requirements for teaching staff members have fallen for about 3.4%, from to 524 to 506.

3 Student group assignment

At our institution, students are enrolled in a predefined set of obligatory (non-elective) courses. Apart from that, they are allowed to decide which additional elective courses they will enroll, in order to achieve better specialization. At the beginning of each semester, enrollment process is divided into two phases. During the first phase, students decide which of the elective courses they will enroll. During this phase, the majority of students submit theirs enrollment applications. Then, the enrollment process is temporarily suspended, and a lecture schedule is created, with maximal effort to allow all of the enrolled students to lake lectures. At this moment lectures begin.

Unfortunately, there is always a certain percentage of students which did not submit its enrollment application during first enrollment period, so at this point, enrollments are resumed, with one difference: since the lecture schedule bas already been fixed, enrolled students are not automatically assigned into lecture groups. Instead, a list of all late enrollments is composed. All of students on that list compete for remaining empty places in lecture rooms. In an attempt to allow all of the students to take the lectures of enrolled courses, additional scheduling problem is formed: it is necessary to blend all of the late enrolled students into existing lecture schedule so that all of them can take all of enrolled lectures, and without overcapacitying the lecture rooms, if possible, or to create a best possible group assignment with minimum number of overcapacitated rooms and schedule conflicts.

To do so, a list of required data is collected, as follows.

- A list of late enrolled student-courses that must be assigned into lecture groups.
- For each course and each course-group the lecture schedule for complete semester.

- For each course and each course-group the number of regularly enrolled and assigned students.
- For each course a set of constraints that must be satisfied with regard to number of students in lecture groups.

The need for the lastly mentioned constraints can be justified by a simple example. Let us assume that students of the first year are divided into ten groups: 1.01 to 1.10. Let us further assume that groups 1.01 and 1.03 take lectures on course c_1 simultaneously in room r_1, while on course c_2 they take separate lectures: 1.01 in room r_2 and 1.03 in room r_3. From this example, it is obvious that the following must hold:

```
count(1.01) + count(1.03) <= capacity(r_1)
count(1.01) <= capacity(r_2)
count(1.03) <= capacity(r_3)
```

In general case, there is no guarantee that additional students can be blended into existing lecture schedule, so one of the goals of optimization process is to try to satisfy as many constraints as possible.

3.1 Actual data and algorithm implementation

At authors institution this semester we had a number of late enrolled students. For courses on which all groups attend lectures at the same time, students were assigned to any of the course groups. 38 students remained on 18 courses having multiple lecture groups, that had to be scheduled (a total of 89 student-enrollments, or approximately 5 courses per student). A total number of constraints present on those 18 courses was 90).

A typical steady state genetic algorithm was employed, working with population of 1500 chromosomes. The chromosome evaluation function was implemented as follows. For all unsatisfied constraints c_i: $g_1 + g_2 + ... + g_n \leq N_i$ overflow is calculated as $o_i = g_1 + g_2 + ... + g_n - N_i$. Then, for each scheduled student s_j a total amount of conflicting lecture-hours h_j is calculated. Penalty function which is then minimized by genetic algorithm is calculated as:

$$penalty = 4 \cdot \sum_i o_i + 2 \cdot \sum_j h_j.$$

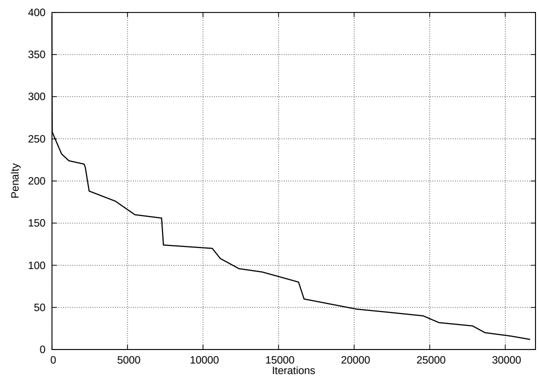

Figure 3: Algorithm's performance on scheduling unscheduled students

Weights used here were determined empirically.

The performance of the algorithm is shown in Figure 3. As it can be seen, the scheduling process finished rather successfully: only two constraints were broken by 1, producing 66% of penalty $4 \cdot (1 + 1) = 8$, and one student ended up with two hours of conflicting lectures, producing the rest 33% of penalty $2 \cdot 2 = 4$, which totals 12. Careful post analysis discovered that it was not possible for the student in question to find a better schedule.

4 Conclusion and Future Work

At the present time, more and more students are enrolled at universities. The ever-increasing number of enrollments poses serious problems for faculty administration involved in solving a variety of organizational issues. Solving these problems by hand had become an impossible mission, especially if the solution quality is considered.

In this paper we described a successful deployment of today available computation power to help with two common problems faced by many universities. The problems were tackled by two implementations of genetic algorithms, both of which proved to be very capable. Usage of those algorithms enabled us to provide a higher quality of studying to students: late enrolled students were successfully blended into existing lecture schedule, while still allowing them to enroll requested elective courses. Course staff was also deburdened, and for each course exam a non-conflicting room schedule was created, which produced additional benefit –

lowered the demands on teaching staff members.

There are still many similar problems at universities world-wide, currently solved by hand, which are ready to be tackled by evolutionary algorithms. Investigating university needs and providing adequate solutions is a part of our future work.

Also, since the focus of this paper is to show how AI-techniques can be used to help faculty administration, this paper presents a successfull application of genetic algorithm to two described scheduling problems. Using these algorithms we were able to solve real-world problems faced by authors institution. As part of future work, the focus will be shifted to obtaining better performance. This will require a more rigorous comparison of applicable (meta-)heuristic search algorithms.

5 Acknowledgments

This work has been carried out within project 036-0361994-1995 Universal Middleware Platform for e-learning Systems funded by the Ministry of Science, Education and Sport of the Republic of Croatia.

References

[1] De Jong, K. A.: Evolutionary Computation, MIT Press, Cambridge, 2006.

[2] Deb, K.: Multi-Objective Optimization using Evolutionary Algorithms, Wiley, New York, 2009.

[3] Affenzeller, M., Wagner, S.: Genetic Algorithms and Genetic Programming, Modern Concepts and Practical Applications, CRC Press, Boca Raton, 2009.

[4] Čupić, M., Golub, M., Jakobović, D.: Exam Timetabling Using Genetic Algorithm, Proceedings of the 31st International Conference on Information Technology Interfaces, 22nd - 25th June, Cavtat/Dubrovnik, Croatia, 2009, pp. 357–362.

[5] Eberhart, R.C., Kennedy, J.: A new optimizer using particle swarm theory, Proceedings of the Sixth International Symposium on Micro Machine and Human Science, Nagoya, Japan, 1995, pp. 39–43.

[6] Montes de Oca, M.A., Stützle, T., Birattari, M., Dorigo, M.: Frankenstein's PSO: A Composite Particle Swarm Optimization Algorithm, IEEE Trans. on Evolutionary Computation. 2009, 13(5), pp. 1120–1132.

[7] Dorigo, M., Stützle, T.: Ant Colony Optimization, MIT Press, Cambridge, MA, 2004.

[8] De Castro, L.N., Von Zuben, F.J.: The Clonal Selection Algorithm with Engineering Applications, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00), Workshop on Artificial Immune Systems and Their Applications, Las Vegas, Nevada, USA, 2000, pp. 36–37.

[9] De Castro, L.N., Timmis, J.: Artificial Immune Systems: A new computational intelligence approach, Springer-Verlag, Great Britain, 2002.

[10] Cutello, V., Nicosia, G.: Chapter VI. The Clonal Selection Principle for In Silico and In Vitro Computing, Recent Developments in Biologically Inspired Computing, eds. Leandro Nunes de Castro and Fernando J. Von Zuben. Hershey, London, Melbourne, Singapore: Idea Group Publishing, 2005, pp. 104–146.

[11] Johnson, D. S., McGeoch, L. A.: The Traveling Salesman Problem: A Case Study in Local Optimization, Local Search in Combinatorial Optimization, E. H. L. Aarts and J. K. Lenstra (editors), John-Wiley and Sons, Ltd., 1997, pp. 215-310.