EUROPA-UNIVERSITÄT VIADRINA FRANKFURT (ODER)

Wirtschaftswissenschaftliche Fakultät

Masterarbeit

# Metaheuristic algorithms for the automated timetabling of university courses

# List of Figures

# 1. Introduction

The creation of timetables is an important task in many areas of daily life: timetables are necessary in such different fields as education, transportation, healthcare (e.g. nurse rosters) or entertainment. However, timetabling is not a trivial task – it has been shown to be among the NP-hard problems. As such the use of computers is almost a necessity when it comes to dealing with complex timetabling problems. The main goal of this thesis is consequently the development and evaluation of an automated timetabling solver program for a specific field of timetabling problems, namely university course timetabling. Because of the NP-hardness of timetabling, the solver is not implemented as an exact algorithm, but rather utilizes metaheuristic algorithms to find a good solution.

The thesis is structured as follows:

Chapter 2 introduces the timetabling problem to the reader. First the university course timetabling problem is described in detail in chapter 2.1; then the constraints, which need to be satisfied, are explained in chapter 2.2. Finally, in chapter 2.3, a mathematical formulation of the course timetabling constraint satisfaction problem is given.

Chapter 3 is focused on the description of the benchmark datasets: chapter 3.1 gives a broad overview over the publicly available datasets for course timetabling problems; in chapter 3.2 the problem dataset used at the second International Timetabling Competition (ITC-2) is presented in detail – the problem instances of this dataset will be used in the following chapters to conduct the experiments and evaluate the performance of the solver.

The aim of chapter 4 is to give a brief introduction into the literature on the previous utilization of metaheuristics for course timetabling. At first the different approaches to the solution of timetabling problems are defined in chapter 4.1. In the following chapter 4.2, the usefulness of several metaheuristics for timetabling is assessed using sources from the recent literature on course timetabling and the results of the ITC-2; based on those findings the two metaheuristics Tabu Search and Simulated Annealing are deemed to be the most promising and are briefly described.

In chapter 5 the implementation of the solver is explained with graphical examples and pseudo-code. At first the internal classes of the solver and the way the timetabling data is represented in the program code are introduced in chapter 5.1. Afterwards chapter 5.2 takes a look at the neighborhood structure, which is used by the metaheuristics. Chapter 5.3 shows the workings of the construction heuristic, which is used to build an initial, partially complete timetable. Chapter 5.4 describes how Tabu Search and Simulated Annealing are used to find a complete and feasible timetable and goes into more detail as to the function of the two metaheuristics. In chapter 5.5 a brief explanation is given how the two metaheuristics can be utilized to improve the solution quality. Finally chapter 5.6 is concerned with how the solution timetable can be validated.

The computational experiments with the working solver program are described in chapter 6. First it is attempted to find the optimal parameter settings for both Tabu Search and Simulated Annealing. Using these parameter settings, several experiments are conducted in chapter 6.2 to answer the following questions:

- Which of the two metaheuristic - Tabu Search or Simulated Annealing - is better at finding a feasible timetable?
- Is Tabu Search better than a random assignment by the construction heuristic at finding a feasible timetable?
- Which of the two metaheuristics is better at improving the feasible timetable?
- Which neighborhood structure leads to the best performance?
- How does the runtime of the solver affect the solution quality?

With the optimal settings determined by the previous experiments, the solver is then used to create timetables for all 21 problem instances from the ITC-2 dataset and the quality of those solutions is compared with the quality of the ITC-2's winning algorithm's solutions.

Finally chapter 7 gives a summary of the results and introduces some problems for future research. The appendix, the bibliography and a summary of the thesis in German language are given after the final chapter.

# 2. Problem Description

## 2.1 University Course Timetabling

Timetabling is described by (Lewis, 2008) as the assignment of entities to a limited number of resources within a certain timeframe, such that given requirements or constraints are met. Adapting this definition to the topic of this thesis, the university course timetabling problem is defined as the assignment of events (e.g. lectures or tutorials) or persons (e.g. teachers or students) to a limited number of timeslots and rooms, subject to a number of constraints, which ensure that the timetable is feasible. As an example the assignment: "the lecture Math1 is scheduled to take place in room 7 on the second period of Monday" might be valid for a certain timetable, if no other lecture is taking place in room 7 at the same time.

According to (Schaerf, 1999), the timetabling problem can be formulated both as a *search problem* or an *optimization problem*. Formulated as a search problem, the goal is simply to find a feasible timetable, i.e. one that satisfies all of the mandatory (so-called hard) constraints. Formulated as an optimization problem - additionally to finding a feasible timetable - the goal is to minimize (or maximize) an objective function which embeds non-mandatory (so-called soft) constraints. An example for a search problem is finding a timetable, which does not have two lectures assigned to the same room at the same time. An example for an optimization problem is finding a timetable, which does not have two lectures assigned to the same room at the same time and also has as little lectures scheduled on Friday as possible (because nobody likes to have lectures on Friday, but it is not forbidden to have them anyway); in this example the objective function, which needs to be minimized, is the number of lectures on Friday.

However, the task of creating a timetable is not trivial. Timetabling is known to be among the NP-complete problems; this has been shown - among others - by (Cooper & Kingston, 1996), who proofed - for several known NP-complete problems such as graph k-colorability - the existence of a polynomial time reduction to different versions of the course timetabling problem.

Thus an exact algorithm can be expected to perform very poorly, given reasonably large problem instances. Especially for university course timetables - where often hundreds of different events must be scheduled to dozens of rooms and thousands of students have to be taken into account - an exact algorithm is not able to provide a solution within an acceptable timeframe. In this case it is necessary to rely on approximate algorithms, which may not be able to find the overall best solution, but at least can provide a good solution in reasonable time.

Specifically tailored heuristics to solve the timetabling problems at certain universities have been around for many years; unfortunately they didn’t have a wide application apart from the researchers’ home institution, because these heuristics were fine tuned to the specific problem at hand. Therefore in the recent decades the use of metaheuristics for timetabling has found a broader appeal: the goal was to create an algorithm which could be used - with only minimal configuration - for a wide variety of different problem instances.

The application of timetables can be found in many different areas, such as transportation, healthcare or education. In the transportation sector, timetables need to be created for buses or trains; in healthcare nurse rosters or operation schedules for surgeons are necessary. The most famous timetabling problems are naturally found in the educational sector: the creation of timetables for schools and universities. Universities need timetables for examinations and courses, schools need them for classes. The structure of all the educational timetabling problems is largely similar; however there are some important differences that distinguish them from one another. As a result an algorithm that can find a solution for one of the educational timetabling problems (e.g. for course timetabling, which is the focus of this thesis) can also be used to find solutions for the other problems, but only with some minor modifications. To better understand the particularities of course timetabling, it is worthwhile to have a look at the distinctions between it and the other educational timetabling problems.

The main difference between university course timetabling and school timetabling is that students at universities have the possibility to choose among the offered courses; they will typically have individual, personal timetables that differ from the timetables their peers have. Courses at universities are often attended by students from different departments or faculties.

Contrary to students at universities, students in school are aggregated in classes. All classmates have the same courses at the same times. Furthermore in school timetabling the assignment of rooms is only a minor problem, as usually every class will have their dedicated classroom. Another difference is that teachers at schools typically have very dense teaching schedules with little time in-between courses, whereas lecturers at universities have less tightly packed teaching schedules.

There are also important differences between examination timetabling and course timetabling: The first difference is that only one lecture can be held per room at the same time, while on the other hand it is certainly possible to hold two or more exams in the same room simultaneously. Furthermore, it is possible to split an exam over two or more rooms, if one room is too small to accommodate all of the students. With a lecture this is not possible, because the lecturer can only be in one place at the same time; consequently all of the students attending this lecture need to fit into a single room.

Another difference concerns the maximum number of timeslots: courses are normally scheduled for one week and the weekly schedule is then repeated until the end of the term. Because all the courses need to fit into one week, there is a tight upper bound for the number of available timeslots, e.g. a five-day week with nine timeslots per day results in an upper bound of 45 timeslots per week. The exams, on the other hand, usually take place only a single time at the end of the term (the schedule is not repeated) and can be scheduled over a timeframe of more than one week. Upper bounds for the number of timeslots are usually less strict than in course timetabling, e.g. the default exam period at a certain university is two weeks, but if that’s not long enough to schedule all of the exams, the exam period can be easily extended by a few days – in contrast extending a weekly course schedule by a few days is usually not an option.

Finally the timetabling often occurs at different points in time for course- and examination timetables: the courses are often scheduled before the beginning of a new term, i.e. before the student enrolment is known. The exams, on the other hand, take place at the very end of the term, so they can be scheduled after the students have been enrolled in the courses and it is precisely known which student has to take which exam (Burke, et al., 1997).

When speaking about the timetabling of courses, it is actually the case that individual events must be assigned to timeslots and rooms. A course is typically made up of several weekly events - such as lectures, tutorials, laboratories or discussion groups - which could take place at multiple times per week (i.e. the same event could be repeated two or more times, to give students who can't attend one of events the chance to visit it at a different time). All of these different components of a course can have different requirements - they might have differently sized groups of students, require different special equipment or special rooms or are taught by different teaching staff (Murray, et al., 2007). E.g. the course "Math1" may consist of one lecture for 50 students on Monday and two (identical) tutorials on Tuesday and Thursday for 25 students each. The lecture is given by a professor while the tutorials are held by two teaching assistants.

## 2.2 Constraints

There are usually two main types of constraints which need to be considered for timetabling problems: hard constraints and soft constraints. The so-called hard constraints are mandatory for a feasible timetable; if it is not possible to fulfill all of the hard constraints, the timetable is infeasible and normally can't be used in practice. The most common hard constraints of the course timetabling problem, adapted from (Lewis, 2006), are:

- No lecturer (or in some variants no student) can be in two places at the same time, or - expressed more formally - can be assigned to two different places (rooms) in the same timeslot (e.g. if Mr. Smith is teaching Math1 at 9 a.m. on Mondays, he can't also teach Physics3 at 9 a.m. on Mondays).
- The number of students, who are enrolled in a lecture, must be smaller or equal to the maximum capacity of the room in which the lecture is taking place; if there are special features required by the lecture (e.g. computer access or a projector), then only rooms equipped with these features are allowed.
- Only one lecture is allowed per room and per timeslot; it is forbidden to schedule two or more lectures to the same room during the same timeslot.

There may be additional hard constraints - depending on the requirements of the university for which the timetable is created - but these are the most fundamental constraints, which apply for basically all course timetabling problems. If only one of the hard constraints is violated by only a single event, then the timetable is deemed infeasible. Infeasible timetables are commonly regarded as useless, because they can’t be implemented in practice; e.g. if Math1 and Biology2 are both assigned to room 5 on Tuesday 11 a.m., the timetable is infeasible and not useable in practice.

Apart from the hard constraints, the timetable creator also has to deal with the soft constraints. Their fulfillment is not necessary to produce a feasible timetable, but the degree to which the soft constraints are implemented determines the quality of a timetable: the less soft constraints are violated, the better the timetable is. In practice the quality of a timetable is usually measured by its penalty score (although there are other measurements as well, see chapter 2.3). The penalty score is a numerical value, which is computed by a penalty function and depends on the number of soft constraint violations. Usually the penalty function is simply a linear combination of the (weighted) number of violations of each soft constraint. For example, a penalty function could take as inputs the number of lectures scheduled on Friday and the number of lectures scheduled before 11 a.m.; for the sake of example the former soft constraint is considered twice as important as the latter. A timetable with 10 lectures on Friday and 15 lectures before 11 a.m. would have a penalty score of 35 (= 2*10 + 1*15).

Improving a timetable can thus be considered a minimization problem, where the goal is to minimize the penalty score by changing the assignment of events in the timetable. A feasible timetable with a penalty score of zero is called a perfect timetable. However, a perfect timetable often doesn’t exist in practice, because some soft constraints might have opposing requirements; e.g. the soft constraints “no lectures on Friday” and “at least one lecture on every day of the week” can never both be satisfied at the same time – every possible timetable will always violate at least one of the two soft constraints. Common examples for soft constraints, adapted from (Burke, et al., 1997) and (Lewis, 2006), are:

- A lecturer prefers to have his lectures on specific days (e.g. only Mondays and Wednesdays) or to have a number of lecture-free days
- A course should take place before or after another course
- There should be no events assigned to the last timeslot of the day
- A lecturer prefers to hold his lectures in a particular room
- Students shouldn’t have only a single course on one day, and not more than three consecutive courses per day

Additional or deviating soft constraints may be introduced by the timetabling institution, to fit their specific needs. It is important to remember that even a timetable, which does not fulfill a single soft constraint, can still be feasible and perfectly usable in practice. Soft constraints only describe wishes or conveniences for the students and teaching staff, not mandatory requirements. To further adapt the soft constraints to the specific needs of the university, they may be given different weights, making some soft constraints more important than others.

A typical strategy, to tackle timetabling problems at large universities, is to split the timetable of the whole university into smaller sub-timetables for each faculty or each department. Every faculty or department then creates their own timetable, while trying to incorporate the timetables of the other departments. This works best if one department first commits a possible timetable; the next department then tries to build a timetable while taking into account the previously committed timetable and - if successful - they in turn commit their timetable to be used by the next department. A problem with this approach is that the timetabling problems of the later departments become more and more constrained and thus more difficult to solve. If a department can’t find a feasible solution, the previously committed timetables would consequently have to be revised. E.g. a business administration faculty has the departments “Management”, “Finance” and “Accounting”. First year students are required to enroll in the courses Management1, Finance1 and Accounting1. At first the management department creates a timetable for all of their courses; they decide to schedule Management1 on Tuesday 11 o’clock. Afterwards the finance department creates a timetable for their courses, but with the additional hard constraint that Finance1 can’t be scheduled on Tuesday 11 o’clock, because that would cause a conflict with Management1. In the example Finance1 is scheduled on Monday at 8 o’clock. Finally the accounting department creates a timetable, where Accounting1 can neither be scheduled on Monday at 8 or Tuesday at 11 o’clock.

Arguably the most fundamental of all constraint is the so called “event-clash” constraint. It states that if a singular resource (i.e. a resource that exists only once, such as a person or a unique room) is assigned to an event, then the resource cannot be assigned to another event in the same timeslot - the resource can’t be in two places at once (Lewis, 2008). E.g. if Mr. Smith is scheduled to hold a lecture on Monday at 10 o’clock, none of the other lectures involving Mr. Smith are allowed to take place on

Monday at 10 o’clock. The event-clash constraint allows drawing a parallel between timetabling and the graph k-colorability problem: in graph coloring connected nodes are prohibited to have the same color; in timetabling connected events (i.e. events that utilize the same resource) are prohibited to take place in the same timeslot.

A more detailed description of the relation between timetabling and graph coloring problems is given in (Lewis, 2008). However, it is not possible to directly apply a graph coloring algorithm for a timetabling problem: timetabling problems also have other hard constraints besides the event-clash constraint, which all need to be fulfilled to produce a feasible timetable. Furthermore graph coloring does not take soft constraints into account at all. Consequently graph coloring could be considered a special case of a timetabling problem.

According to (Corne, et al., 1995), both the hard- and the soft constraints can be categorized into five main constraint classes: Unary-, binary-, capacity-, event spread- and agent constraints.

- Unary constraints involve only a single event, e.g. “event X must be scheduled on Wednesday” or “event X mustn’t be assigned to a timeslot in the morning”.
- Binary constraints involve exactly two events; an example for this kind of constraint would be “event X must follow immediately after event Y”. The event clash constraint can also be classified as a binary constraint.
- Capacity constraints typically concern the maximum size of the rooms, e.g. “event X must be assigned to a room with at least 80 seats”.
- Event spread constraints handle the distribution of events over a timespan. It could require events to be close together (“clumping-together”) or it could require events to be as far apart from each other as possible (“spreading-out”). As an example, a teacher, who prefers to finish his required teaching hours on a single day, might propose the constraint: “the events X, Y and Z should all be scheduled with minimal time in-between”. The spreading-out of events could, for example, benefit students by reducing their daily workload.
- Agent constraints are used to endorse the preferences or requirements of the involved individuals (i.e. students or teaching staff). “Mrs. Smith prefers to hold courses in the morning” or “Mr. Wesson doesn’t want to teach tutorials on Mondays and Fridays” are examples of agent constraints.

## 2.3 Mathematical Formulation

A prototypical problem formulation for the hard constraints of a certain variant of the university course timetabling problem – namely curriculum-based course timetabling - is given by (de Werra, 1985) and (Schaerf, 1999):

> "There are q courses $K_1$, $K_2 \ldots K_q$ and for each i, course $K_i$ consists of $k_i$ lectures. There are r curricula $S_1$, $S_2 \ldots S_r$ which are groups of courses that have common students. This means that courses in $S_l$ must be scheduled all at different times. The number of timeslots is p, and $l_k$ is the maximum number of lectures that can be scheduled at timeslot k (i.e. the number of rooms available at timeslot k)."

In mathematical notation, the formulation for the hard constraints of the course timetabling problem can be written as follows, after (Schaerf, 1999):

$$find\ y_{ik} \qquad (i = 1 \ldots q; k = 1 \ldots p) \qquad (1)$$

subject to

$$\sum_{k=1}^{p} y_{ik} = k_i \qquad (i = 1 \ldots q) \qquad (2)$$

$$\sum_{i=1}^{q} y_{ik} \leq l_k \qquad (k = 1 \ldots p) \qquad (3)$$

$$\sum_{i \in S_l} y_{ik} \leq 1 \qquad (l = 1 \ldots r; k = 1 \ldots p) \qquad (4)$$

$$y_{ik} = \{0,1\} \qquad (i = 1 \ldots q; k = 1 \ldots p) \qquad (5)$$

$$y_{ik} = \begin{cases} 1, & \text{if a lecture of course } K_i \text{ is assigned to timeslot k} \\ 0, & \text{otherwise} \end{cases} \qquad (6)$$

In other words, the goal is to find an assignment for each course $K_i$ to some timeslot k, which doesn't violate any of the above constraints. In total there are q different courses, which need to be scheduled to a maximum number of p timeslots (Equation

(1)); e.g. $y_{3,7} = 1$ signifies that course $K_3$ is assigned to any room in timeslot 7 (which room is not important, because the room assignment is only relevant for the soft constraints).

The constraint described in equation (2) states that for all of the q courses, each course has to be made up of the correct number of individual lectures. If any lecture of a course is not assigned to a timeslot, the result would be a violation of this constraint.

Constraint (3) is a capacity constraint; for any timeslots, the number of lectures, which are scheduled to this timeslot, can be no bigger than the number of available rooms $l_k$ in that timeslot; e.g. if there are five rooms available in timeslot 12 ($l_{12} = 5$), then at most five lectures can be assigned to timeslot 12.

Equation (4) implements the idea of curriculum-based course timetabling: for any timeslot and for any curriculum, there can be at most one lecture of every curriculum assigned to the same timeslot, i.e. two lectures which are part of the same curriculum cannot be scheduled in same timeslot. The idea of curriculum-based timetabling can be illustrated with the following example: the curriculum of second year engineering students requires them to enroll in the courses Mechanics2, Physics1 and Math1; consequently Mechanics2, Physics1 and Math1 must all be scheduled to different timeslots.

Finally, the equations (5) and (6) infer that the assignment must be a binary variable; y has the value 1, if the lecture is assigned to the respective timeslot and it has the value 0, if it is not assigned.

Another problem formulation, also given by (Schaerf, 1999), uses a conflict-matrix instead of curricula: a quadratic matrix $C_{q*q}$ with length q (= the number of courses) is used to record whether the courses have students in common and are thus conflicting. If the courses $K_i$ and $K_j$ have at least one common student, then $c_{ij} = 1$, otherwise, if they are non-conflicting $c_{ij} = 0$. Using the conflict matrix, constraint (4) can be reformulated, such that for all of the timeslots, the sum of all conflict values must be zero (i.e. there can never be a conflicting assignment); an example for a conflict matrix (which is using integers instead of binary values) is shown below in Figure 1.

In the original formulation of the curriculum-based course timetabling problem by (de Werra, 1985), the solution quality is measured by a function, which computes the “desirability” of an assignment (equation (7)). The goal is to maximize this function by making changes to the timetable, while preserving the above hard constraints.

$$\max \sum_{i=1}^{q} \sum_{k=1}^{p} d_{ik} y_{ik} \qquad (7)$$

In the above function, the weight $d_{ik}$ reflects the desirability of having a lecture of course $K_i$ scheduled in timeslot k (the more desirable it is to have course $K_i$ assigned to timeslot k, the higher the value for $d_{ik}$ will be). The aim of the algorithm is to maximize the value of the objective function, by creating a timetable, where the courses are scheduled to the most desirable timeslots. For example when $d_{2,8} = 100$ and $d_{2,9} = 50$, then it is more desirable to assign a lecture of course 2 to timeslot 8, then it is to assign the same lecture to timeslot 9.

The formulation of a timetabling problem by (Tripathy, 1992) combines a conflict-matrix with a penalty score: the conflict-matrix is built with integer (instead of binary) values, where each integer $c_{ij}$ represents the number of students who are attending both of the courses $K_i$ and $K_j$. Student conflicts are not a violation of a hard constraint in this interpretation of the course timetabling problem, but rather only a violation of a soft constraint; the goal is to minimize the number of students who are involved in a conflicting assignment of lectures. A graphical depiction, adapted from (Tripathy, 1992), is given in Figure 1:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | Course |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 169 | 41 | 30 | 28 | 135 | 14 | 70 | 33 | 144 | 132 | 164 | 79 | 50 | 21 | **1** |
|  | 41 | 3 | 9 | 34 | 1 | 19 | 7 | 34 | 32 | 39 | 15 | 9 | 3 | **2** |
|  |  | 30 | 8 | 14 | 0 | 4 | 1 | 27 | 24 | 28 | 11 | 12 | 5 | **3** |
|  |  |  | 28 | 16 | 0 | 5 | 1 | 22 | 20 | 28 | 9 | 11 | 4 | **4** |
|  |  |  |  | 135 | 12 | 66 | 29 | 111 | 103 | 133 | 69 | 35 | 12 | **5** |
|  |  |  |  |  | 14 | 6 | 4 | 10 | 7 | 14 | 8 | 0 | 1 | **6** |
|  |  |  |  |  |  | 70 | 27 | 58 | 51 | 68 | 21 | 7 | 4 | **7** |
|  |  |  |  |  |  |  | 33 | 28 | 21 | 31 | 6 | 2 | 2 | **8** |
|  |  |  |  |  |  |  |  | 144 | 113 | 140 | 63 | 43 | 16 | **9** |
|  |  |  |  |  |  |  |  |  | 132 | 128 | 61 | 37 | 15 | **10** |
|  |  |  |  |  |  |  |  |  |  | 164 | 79 | 49 | 20 | **11** |
|  |  |  |  |  |  |  |  |  |  |  | 79 | 21 | 10 | **12** |
|  |  |  |  |  |  |  |  |  |  |  |  | 50 | 10 | **13** |
|  |  |  |  |  |  |  |  |  |  |  |  |  | 21 | **14** |

**Figure 1 – Example of a conflict matrix with integer value entries, Source: (Tripathy, 1992)**

The cell in row i and in column j of the matrix is described with the variable $m_{ij}$; if $m_{ij} = 0$, then course i and course j are not conflicting; if $m_{ij} > 0$, then the respective courses have exactly $m_{ij}$ students in common; if i = j, the cell $m_{ij}$ indicates the total number of students who are enrolled in course i. For example the entry in row 4 and column 10 of the above table shows that the courses 4 and 10 have exactly 20 students in common ($m_{4,10} = 20$). Thus it should be avoided to assign those two courses to the same timeslot. Courses 3 and 6 on the other hand don't have any common students ($m_{3,6} = 0$), and can be safely assigned to the same timeslot.

A mathematical formulation of pre-assignments and unavailability constraints is given by (Müller, 2005):

$$\forall\, i = 1 \dots q \quad \forall\, k = 1 \dots p \qquad p_{ik} \leq y_{ik} \leq a_{ik} \tag{8}$$

$$p_{ik} = \begin{cases} 1, & if\ a\ lecture\ of\ K_i\ must\ be\ assigned\ to\ timeslot\ k \\ 0, & otherwise \end{cases} \tag{9}$$

$$a_{ik} = \begin{cases} 1, & if\ a\ lecture\ of\ K_i\ can\ be\ assigned\ to\ timeslot\ k \\ 0, & otherwise \end{cases} \tag{10}$$

Keeping in mind that $y_{ik}$ is a binary variable, then a value of 1 for $p_{ik}$ means that course $K_i$ has to be scheduled to timeslot k, i.e. $y_{ik}$ must be 1. Similarly a value of 0 for $a_{ik}$ forces $y_{ik}$ to be equal to zero as well, so course $K_i$ cannot be scheduled to timeslot k. In case when $p_{ik} = 0$ and $a_{ik} = 1$, the variable $y_{ik}$ remains unrestrained.

If, for example, $a_{3,9} = 0$, then a lecture of course 3 cannot be assigned to timeslot 9; if $a_{3,9} = 1$, then it is possible (but not required) to assign a lecture of course 3 to timeslot 9. If $p_{3,8} = 1$, then a lecture of course 3 must be assigned to timeslot 8; if $p_{3,8} = 0$, then it is not necessary to assign a lecture of course 3 to timeslot 8.

The formulation for the Udine University benchmark timetabling problem (see chapter 3.2), which is used to evaluate the developed solver algorithm in chapter 6, can be best described as a mixture of the curriculum-based course timetabling formulation by (Schaerf, 1999), enhanced with the pre-assignments and unavailability constraints formulation by (Müller, 2005). The quality of the timetables is measured using a penalty function, as described in the previous chapter. However, the solver should – in principle - be able to find solutions using other formulations (according to the requirements of the user) as well.

# 3. Benchmark Data

## 3.1 Overview of Course Timetabling Datasets

Often a motivation for researchers to develop an automatic timetabling algorithm comes from a concrete timetabling problem at the researchers’ own home institution; the solution is then tailored to the specific problem at that institution, with all its special constraints and requirements. As a result it might be difficult or even entirely impossible to use these algorithms to create timetables at other institutions. About a decade ago the timetabling community began to build a number of benchmark problems, which could be used by other researchers to test their own timetabling programs; having a common benchmark problem with a common problem formulation made the different algorithms comparable to one another – researchers could more accurately judge the quality of their algorithms and further improve them.

Of course there is no single problem formulation which suits all use-cases, because the individual requirements of different universities are frequently incompatible with each other. One major difference between universities is, for instance, the enrolment scheme: at some universities the students have to enroll in the course before the start of the new term, in other cases they are allowed to enroll at the beginning of the term or they might not need to enroll at all. This difference already makes it impossible to have a common set of hard constraints suitable for all course timetabling problems; e.g. student conflicts cannot be a criterion for the feasibility of a timetable, if the student enrollment is unknown at the time of the timetable creation.

These problems aside, it is still desirable to provide a means of measurability for the performance of different timetabling algorithms. After all, adapting a general problem formulation to a specific problem at a certain institution can generally be achieved by changing some hard and/or soft constraints or changing the weightings of the soft constraints.

The first public benchmark datasets for course timetabling were published about a decade ago. They feature easy and small problem instances and are drastically simplified compared to their real-world counterparts. A random instance generator for these simplified problem versions was developed by (Burke, et al., 2007).

The most common of the earlier benchmark datasets was adapted from a timetabling problem at Napier University in Edinburgh, Scotland and designed by (Paechter, et al., 1998) for the Metaheuristics Network; it was used during the first international timetabling competition (ITC-1) in 2002 (the actual datasets, rules, input format and constraint descriptions are available online[1]). All of the 20 problem instances have a feasible solution and at least one perfect solution, i.e. a solution that satisfies all of the given soft constraints; a problem formulation is given by (Lewis, 2006).

The benchmark problems provided by the ITC-1 were criticized by many researchers for being too abstract and too different from real-world instances. Consequently, in the next iteration of the International Timetabling Competition the organizers provided two, more complex and more realistic benchmark datasets for course timetabling: one of the datasets was aimed at post-enrolment course timetabling (i.e. the enrolment data for each student is available, see the conflict-matrix in Figure 1 for an example) and one at curriculum-based course timetabling (i.e. a set of curricula is used to prevent conflicting assignments, see chapter 2.3 for an example).

Like other recent benchmark problems, the benchmark datasets used at the second International Timetabling Competition in 2007 (ITC-2) were not artificially created; instead they are derived from real-world timetabling problems, which were anonymized and purged of institution specific constraints. The benchmark dataset for the curriculum-based course timetabling track of the competition was adapted from Udine University in Italy; it is described in further detail in the next section of this thesis. Another well-known - although not as widely used - benchmark problem uses anonymized real data from Purdue University (USA). The problem instances in this dataset are considerable larger than the benchmark problems used during the ITC-2 (Purdue University has about 39,000 students and 9,000 courses). A detailed problem description of the Purdue timetabling problem[2] is given in (Rudová, et al., 2011).

[1] http://www.idsia.ch/Files/ttcomp2002

[2] The datasets are available at http://www.unitime.org

## 3.2 Udine University Dataset

The ITC-2[3] featured three tracks in total, two of which were dedicated to course timetabling:

- Track 1: Examination timetabling,
- Track 2: Post Enrolment based Course Timetabling and
- Track 3: Curriculum based Course Timetabling – the the main focus of the thesis.

In Post-Enrolment-based course timetabling the students are associated with courses before the timetable is created, i.e. it is exactly known in advance which student will be taking which course – thus the goal in this problem version is to minimize the number of student conflicts. This model is a further development of the original model used at ITC-1.

Curriculum-based Course Timetabling (CB-CTT) takes a slightly different approach: every student belongs to a certain group and has a corresponding associated curriculum, i.e. a set of distinct courses which she is enrolled in. For example a first year business administration student is required to take the courses "Accounting I", "Microeconomics" and "Supply Chain Management II" – these courses then form curriculum 1. As a consequence none of the previously mentioned three courses should be scheduled in the same timeslot – assigning two courses of curriculum 1 to the same timeslot would result in a curriculum conflict. The goal is to minimize the overall number of curriculum conflicts for all of the different curricula.

The CB-CTT benchmark data was adapted from a real-life timetabling problem at *Università degli studi di Udine* in Italy (the benchmark datasets are available online at the University's website[4]). The problem formulation of the 2007 version is provided by (Di Gaspero, et al., 2007); an updated version (which includes a few additional problem instances) is described in (Bonutti, et al., 2012). Furthermore (Burke, et al., 2012) gives an integer programming formulation for this timetabling problem.

The hard constraints in the original ITC-2 problem formulation are:

[3] http://www.cs.qub.ac.uk/itc2007/index.htm

[4] http://tabu.diegm.uniud.it/ctt

- **Lectures:** A course consists of multiple lectures; all lectures of a course must be scheduled to distinct timeslots.
- **Room Occupancy:** There can be only one lecture per room in the same timeslot.
- **Conflicts:** Courses of the same curriculum or taught by the same teacher must be assigned to different timeslots.
- **Availability:** When the teacher is unavailable during a timeslot, no courses by that teacher can be scheduled to this timeslot.

The mathematical formulation of the above hard constraints is available in chapter 2.3. The soft constraints of the ITC-2 problem formulation are:

- **Room Capacity (RCap):** When a lecture is scheduled to a room, the number of seats in that room must be at least as big as the number of students enrolled in that lecture; each student who doesn't have a seat counts as 1 penalty point.
- **Event spread (TSpr):** The lectures of each course should be spread out over a given minimum number of days. E.g. if the minimum number of days is set to three, then a course consisting of five lectures might have two lectures on Monday, two on Wednesday and one on Thursday. However it should not have three lectures on Tuesday and two on Friday, because in this case the lectures would only be spread out over two days. If less than the minimum number of days is used, each day below the minimum counts as 5 penalty points.
- **Time compactness (TCom):** Lectures that are part of the same curriculum should be assigned to consecutive timeslots. Each lecture, which has not at least one preceding or following lecture of the same curriculum, counts as 2 penalty points.
- **Room stability (RStb):** All lectures of a single course should be assigned to the same room. When a course uses more than one room, each additional room counts as 1 penalty point.

A mathematical formulation of the above soft constraints is available in (Lü & Hao, 2010). The penalty function, which needs to be minimized by the solver, can be written as a linear combination of the above mentioned penalty terms (Burke, et al., 2012). A mathematical formulation is given below:

$$\begin{aligned}\min f_{penalty\ score}(RCap, TSpr, TCom, RStb) \\ = RCap + 5 * TSpr + 2 * TCom + RStb\end{aligned} \tag{11}$$

The penalty function takes as input the number of occurrences of each soft constraint violation; e.g. RCap is the total number of students without a seat, TSpr is the total number of days below the minimum, etc. The number of occurrences of each soft constraint violation is then multiplied by a weight factor, which reflects the preferences of the timetable creator: a higher weight factor results in a higher importance of the soft constraint. In the formulation used at the ITC-2 (the formulation is called "UD2" in (Bonutti, et al., 2012)), the weights are set (arbitrarily) to (1, 5, 2, 1) respectively; i.e. the weight of RCap is 1, the weight of TSpr is 5 etc.

The penalty score computed by the penalty function is used by the solver to determine the quality of a timetable: among two complete and feasible timetables, the one with the lower penalty score is better. When two different timetables have the same penalty score, they are both considered equally good.

The original benchmark dataset, used in track 3 of the ITC-2, includes 21 problem instances; each instance is of different complexity and size. A feasible assignment is known to exist for all of the problem instances; timetables with an optimal penalty score have been discovered for most of the problem instances. In (Bonutti, et al., 2012) an updated benchmark dataset, which includes 7 additional instances, is given. The organizers of the ITC-2 also provided the source code of a solution validator[5], which can be utilized to compute the feasibility and penalty score of the produced timetable for each of the given problem instances. An overview of the different problem instances is presented by (Bonutti, et al., 2012), shown in Figure 31 (Appendix, page 70).

The actual problem formulation used in practice at the University of Udine includes some additional features, which were stripped from the formulation for the ITC-2. Among others, these were a penalty for rooms that are too big for a class, a lunch break cost component (at least one free timeslot is required around noon) and a maximum daily student load. These kinds of constraints were not included in the formulation for the benchmark dataset, in order to make the problem not unnecessarily complex.

[5] The validator and solution timetables for all instances are available at http://tabu.diegm.uniud.it/ctt/

# 4. Metaheuristics for Course Timetabling

## 4.1 Different Approaches to Timetabling

According to (Lewis, 2008), metaheuristics for timetabling can generally be categorized into the following three categories:

- **One-stage optimization algorithms:** a single objective function is used for both hard and soft constraints; the optimization process consists of a single run.
- **Two-stage optimization algorithms:** during the first stage only hard constraints are considered; after a feasible timetable has been constructed, the amount of soft constraint violations is minimized during the second stage, without breaking the feasibility of the timetable.
- **Algorithms which allow relaxations:** violations of hard constraints are prevented by relaxing the problem (e.g. opening additional timeslots); the algorithm then tries to simultaneously minimize the amount of soft constraint violations and eliminate all relaxations.

Among these strategies, the one-stage approach is the most straight forward one: The metaheuristic uses a single penalty function to find a satisfactory solution for both the hard and the soft constraints at the same time. To achieve this, hard and soft constraints are given different weights, indicating their respective importance; i.e. hard constraints are assigned much higher weights than soft constraints. As a result, a violation of a hard constraint has a much higher effect on the penalty score than a violation of a soft constraint. E.g. the penalty function might assign a weight of 1000 to every occurrence of an event clash (a hard constraint), while exceeding the room capacity (a soft constraint) only costs 3 penalty points per occurrence. Clearly the penalty score of an infeasible timetable, which has 1 event-clash and no soft constraint violations (penalty score: 1000) is higher than the score of a feasible timetable with 100 soft constraint violations (penalty score 3*100). Thus, by minimizing the penalty score, the algorithm will usually be able to produce a feasible timetable.

The major advantage of the one-stage approach is that it is comparatively easy to implement and easy to modify: changes to the constraints can simply be reflected by changing the weights (i.e. turning a hard constraint into a soft constraint is accomplished by decreasing its weight).

However, this approach hasn’t been very successful at finding solutions for complex problems (such as the ITC-2 instances), because of several inherit disadvantages:

- The choice of weights is often arbitrary; there is no general applicable scheme for setting the weights and they are often very specific to the problem at hand.
- A small change of the timetable often results in a large change of the penalty score (i.e. when a hard constraint has been violated), making the fitness landscape more difficult to navigate.
- The incorporation of soft constraints could take the solution further away from attractive (feasible) regions of the search space.

For these reasons the utilization of a two-stage approach is more common when trying to solve harder problems. The two-stage approach better reflects the nature of the timetabling problem of having both hard and soft constraints: During the first stage the only goal is to find a feasible timetable, while the soft constraints are completely ignored by the algorithm. Once a feasible timetable has been created, the second stage begins, where the soft constraints are finally taken into account. It is important that no hard constraint violations are allowed during the second stage of the search process; only those neighborhood operations are permitted that leave the timetable intact (feasible).

The success of this approach hinges on two criteria:

- It must be possible to find an initial, feasible timetable reasonably fast; if feasibility is very hard or even impossible to achieve, the algorithm might never reach the second stage; in this case a one-stage approach could produce at least a solution with a suitable compromise between hard and soft constrain violations.
- The feasibly-only search space must allow for a large enough amount of possible neighborhood moves; a strongly constrained search space (i.e. when only a few feasible solutions exist) might make it difficult or even impossible to reach another possible solution. E.g. for some timetabling problem, there exist only two feasible timetables. During the first stage the one timetable with the higher penalty score has been constructed; in order to reach the second feasible timetable, at least two consecutive changes to the first timetable are necessary

during the second stage. However, because every intermediate timetable (the result of a single change to the initial timetable) is infeasible, the algorithm can never reach the second (superior) timetable, starting from the first timetable.

Nevertheless, for the particular problem instances presented at both ITC-1 and ITC-2, the two-stage approach proofed to be more suitable than the one-stage approach, as shown in (Chiarandini, et al., 2006).

The third kind of algorithm, where relaxations of the problem are allowed, also gave promising results when used with the ITC benchmarks. These algorithms relax the problem by either initially leaving those events unassigned, which cannot be feasibly scheduled anywhere, or by opening additional timeslots, when an event cannot be assigned to any of the existing timeslots. The produced timetables consequently never contain any hard constraint violations, but don’t necessarily include all events or have the required number of timeslots.

The goal of the relaxation-algorithms is to minimize the soft constraint violations as well as simultaneously trying to remove all of the relaxations. Whereas the goodness of a candidate timetable during the first stage of the two-stage approach is measured by the number of hard constraint violations, the goodness of the relaxed timetable is measured by the distance to feasibility, i.e. the number of unassigned events or the number of excess timeslots opened.

The solver algorithm implemented as part of this thesis utilizes a mixture of the two-stage approach and the relaxation approach: in the first stage a feasible timetable is constructed, while hard constraint violations are prevented by relaxing the timetabling problem (i.e. by leaving events unassigned); the solution quality during this stage is measured by the distance to feasibility (how many events are left unassigned). The second stage starts when the solver has found a timetable with a zero distance to feasibility. In the second stage a penalty function (see chapter 3.2) measures the solution quality; only feasible timetables are considered during this stage.

## 4.2 Comparison of Metaheuristics for Timetabling

The five main metaheuristics considered by the Metaheuristics Network [6] are Evolutionary Algorithms, Ant Colony Optimization, Iterated Local Search, Simulated Annealing and Tabu Search. Using a two-stage approach, a performance comparison of these algorithms regarding their usefulness for timetabling problems was conducted by (Rossi-Doria, et al., 2003). As a general result, it was concluded that the population based algorithms (Evolutionary Algorithms and Ant Colony Optimization) were consistently outperformed by the local search metaheuristics, on all of the tested problem instances. Among the local search heuristics some appeared to work better at finding feasibility, while others performed better at reducing the soft constraint violations: Overall Iterated Local Search was shown to be the best metaheuristic at finding a feasible timetable and Simulated Annealing gave the best results when used during the soft constraint optimization stage. A similar result was found by (Chiarandini, et al., 2006), where Tabu Search and Iterated Local Search were best at reaching feasibility and Simulated Annealing at minimizing soft constraint violations.

An analogy, why the population based algorithms would produce worse timetables than the local search algorithms, can be drawn from an example presented by (Lewis, 2006) for a genetic algorithm: During the crossover phase of the genetic algorithm an infeasible timetable is produced almost every time; the resulting offspring timetable contains duplicate events, while other events are entirely missing from the timetable. As a consequence the offspring timetable has to be repaired by a computationally expensive rebuilding procedure – thus most of the information from the parents is lost in the process. As an example, the two (feasible) "parent" timetables are represented by the vectors (1, 2, 3, 4, 5, 6) and (6, 5, 4, 3, 2, 1), where the number signifies the event and the position in the vector signifies the timeslot (e.g. in the second timetable event 6 is assigned to timeslot 1, event 5 is assigned to timeslot 2 etc.). The cross-over point is after the 3rd timeslot; the offspring timetable produced is thus (1, 2, 3, 3, 2, 1). Clearly this timetable is infeasible: the events 1, 2 and 3 are assigned twice, while the events 4, 5 and 6 are not assigned anywhere.

[6] http://www.metaheuristics.net

Generalizing from the above example, population based algorithms seem to perform worse, because good timetables cannot be easily inherited – the derived timetables are almost always infeasible. A local search, where only small neighborhood changes occur one at a time appear to be the better choice for the construction of a timetable. This assumption is further supported by the conclusions drawn by (Chiarandini, et al., 2006), who observed that Ant Colony Optimization and Genetic Algorithms performed very bad, when they were not hybridized (i.e. used in conjunction) with local search. Another explanation for the poor performance of population based algorithms is given in the same article: "handling multiple solutions has a cost in terms of time which is not paid off in terms of quality" (Chiarandini, et al., 2006).

The winner of the 3rd track of the ITC-2 was a constraint-based solver, developed by (Müller, 2009). The algorithm uses Iterative Forward Search during the construction phase (the first stage) of the timetable creation. For the improvement phase (second stage) the Hill Climbing, Great Deluge and Simulated Annealing metaheuristics are utilized. Remarkably this solver also won the 1st track (examination timetabling) of the same competition.

The second most successful algorithm was provided by (Lü & Hao, 2010). It works by first using a greedy heuristic to build an initial timetable and then applying a Tabu Search/Iterated Local Search hybrid algorithm to reduce the soft constraint violations. The third place was achieved by a Japanese group using a general CSP solver; the fourth place was awarded to an algorithm developed by (Geiger, 2008), which first builds an initial feasible timetabling with a greed heuristic and later improves the timetable with a Threshold Accepting metaheuristic.

The two metaheuristics, which were finally implemented and tested with the automated timetabling solver, are Tabu Search and Simulated Annealing. Those two metaheuristics were selected, because they were shown - in the previously mentioned literature - to be well suited for timetabling problems and likely to give satisfying results. This chapter gives a brief description of both Tabu Search and Simulated Annealing; a detailed description follows in chapter 5.4.

The basic idea of Tabu Search (TS) is to use a tabu list to prevent cycling back to previously visited solutions in the search space, when performing a local search. The tabu list records the previously performed neighborhood moves; e.g. when the neighborhood move consists of swapping events A and B, in the next iteration the move to swap A and B again would be forbidden, as this would lead back to an already explored solution.

A pseudo-code description of a Tabu Search algorithm can be found in (Brownlee, 2011). Tabu Search typically utilizes a Hill Climbing acceptance criterion, i.e. only improving neighborhood moves are accepted (if the new solution is worse than the previous solution, it is always discarded). A metaheuristic, which sometimes also accepts worsening neighborhood moves, is Simulated Annealing.

Annealing is a technique in metallurgy, which involves the controlled heating and subsequent slow cooling of materials, as a means to reduce defects in the material. The idea of Simulated Annealing (SA) as a metaheuristic is that worsening neighborhood moves are accepted with a certain probability. This probability depends on the "temperature" variable – a value that is reduced over time according to a so called cooling schedule.

An example for a pseudo-code SA algorithm is given in (Gendreau, 2010). In the beginning the temperature is set to a high value, meaning that a lot of worsening moves are accepted – with the goal of escaping from local minima and traversing a broad range of the search space. The temperature is gradually lowered, reducing the number of accepted worsening moves and narrowing the search to an increasingly smaller area of the search space. Finally, when the temperature is at zero, only improving neighborhood moves are allowed to let the algorithm converge on a local optimum.

# 5. Implementation of an Automated Timetabling Solver

The proposed timetabling solver algorithm can be divided into three distinct major phases. In the first phase an initial solution is constructed using a construction heuristic; the goal is to build a feasible timetable. If the construction heuristic fails to find a feasible timetable, the second phase is initiated. Here it is attempted to make the timetable feasible with the help of several neighborhood moves guided by a metaheuristic. When a feasible timetable has finally been found, the number of soft constraint violations is reduced in the third and final phase; again a metaheuristic is used to guide the neighborhood moves through the search space. The difference between phases two and three is that the search space in phase three is limited to only those moves, which preserve feasibility; i.e. once a feasible timetable has been build, the algorithm is not allowed to destroy the feasibility of the timetable, in hope of reducing the penalty score.

In the following, first a brief description of the data representation is given in chapter 5.1. Afterwards chapter 5.2 explains the neighborhood structure in detail and presents some examples for neighborhood moves. The next chapters then describe in depth the construction (chapter 5.3), feasibility (chapter 5.4) and improvement (chapter 5.5) phases of the algorithm. Finally the validation of the output is discussed in chapter 5.6.

## 5.1 Data Representation

The input data for the solver is a text file in .ectt file format, as specified by (Bonutti, et al., 2012). An example for an input data file is shown in Figure 32 (Appendix, page 72). The solver extracts the raw input data from the file, creates class instances from the data and writes them into a list. The names of the classes are *Header, Course, Event, Room, Curriculum, Unavailability* and *RoomConstraint*.

The *Header* class records all the parameters of the input data, such as the name of the problem instance and the number of courses, rooms, curricula, days and periods per day. It is used to compute certain values, as well as giving an overview over the problem.

Central to the timetabling problem is the *Course* class; an instance of this class object exists for every course of the problem. The *Course* class records the ID of the course, the associated teacher, the number of lectures and students, the minimum number of working days and the number of double lectures.

*Event* class objects are an abstraction of the *Course* class; an event is created for each lecture of a course, e.g. if the course "Math1" consists of three lectures per week, then three *Event* objects are created with the ID "Math1". This makes it more intuitive to keep track of the number of assigned lectures: three events need to be assigned to one timeslot each, instead of having to assign one course to three different timeslots. Apart from the number of lectures, the *Event* class has the same parameters as the *Course* class.

The *Room* class records the ID of a room, its capacity and the location; the *Curriculum* class records the ID of a curriculum, the number of courses, which are part of the curriculum, and a list of those courses.

*Unavailability* class objects are utilized to record the unavailability constraints imposed on the timetable: the parameters of this class are the ID of a course and a timeslot, during which the teacher of the course is not available for teaching.

Finally there is a *RoomConstraint* class, which records the ID of a course and the ID of a room, meaning that the course should not be assigned to this room.

The timetable itself is represented by a dictionary, which maps a (room, timeslot) tuple to an event. The (room, timeslot) tuples are also called "positions" in the timetable. The size of the dictionary is the number of rooms times the number of timeslots. Initially all positions of the timetable are empty, i.e. they contain "None" type objects. Additionally two lists are used to keep track of the current state of the timetable: a list of empty positions (initially contains all positions of the timetable) and a list of forbidden positions (initially empty). Whenever an event is assigned to a position in the timetable, the position is removed from the list of empty positions and added to the list of forbidden positions.

As an example, when an event "Math1" is assigned to position (2, 13) – the 2nd room in the 13th timeslot: the timetable dictionary now maps (2, 13) to "Math1", the position (2, 13) is removed from the list of empty positions and the position (2, 13) is added to the list of forbidden positions.

## 5.2 Neighborhood Structure

### 5.2.1 Description

The neighborhood used throughout the search process consists of three moves: swapping two timeslots, swapping two events and swapping the rooms of two events. Depending on whether the neighborhood moves are applied during the feasibility- or the improvement phase, they may or may not be required to preserve the feasibility of the timetable. If a neighborhood move is applied during the improvement phase, the move is only allowed, if it does not result in a hard constraint violation; moves during the feasibility phase are always allowed. The constraints are defined by the "UD2" problem formulation of the 3rd track of the ITC-2 (see chapter 3.2).

Of the three possible neighborhoods, only the swapping of timeslots and the swapping of events can be effectively applied during the feasibility phase; swapping the rooms of two events, which are part of the same timeslot, does not affect the feasibility of the timetable at all, because the room capacity constraint is only a soft constraint in this formulation.

When two timeslots are swapped a hard constraint violation could occur, if one of the events is moved to a timeslot in which the designated teacher of this event is unavailable (**Availability**). Furthermore two soft constraints are affected by a timeslot swap: the **Time Compactness** (the number of events without any neighboring events of the same curriculum) and the **Event Spread** (the number of days over which a course is spread out) constraints.

The room swap neighborhood does not affect any of the hard constraints. The affected soft constraints are the assignment of an event to a room with an insufficient number of seats (**Room Capacity**) and the assignment of an event to a different room than the other events, which are part of the same course (**Room stability**).

Swapping two events can result in two possible hard constraint violations: the designated teacher of one or both of the events could be unavailable in the new timeslot (**Availability**) or the event could be part of the same curriculum as another event in the new timeslot (**Conflicts**). All of the soft constraints are affected by an event swap, as the event swap is basically a combination of a timeslot swap for two events and a room swap. When selecting the two events randomly it could also occur that they are both already assigned to the same timeslot – in this case the event swap has the same effect as a room swap.

A pseudo-code representation of the event swap is given below:

```
Function: SwapEvents
Input: position1, position2, boolean(preserve_feasibility)
Output: (True, a tuple of events + positons) OR (False, None)
1:   Create a backup of the events in both positions
2:   if (both events are the same):
3:      return False
4:   if (preserve_feasibility == True):
5:      if one of the events can’t be assigned to the new timeslot:
6:              return False
7:   assign event 1 to position 2
8:   assign event 2 to position 1
9:   return True, backup of the events in their original positions
```

The input to this function are two (randomly) chosen positions in the timeslot and a parameter which determines if the feasibility of the timetable should be preserved; the function returns “False”, if the swap was unsuccessful or “True” and a backup of the events, if the swap was successful.

If the preserve_feasibility parameter is set to True, the function ensures that no hard constraint violations occur during the swap (lines 4-6). The backups are necessary if the swap needs to be reversed (i.e. the original state of the timetable should be restored). The pseudo-code for the reverse event swap is given below:

```
Function: ReverseEventSwap
Input: backup_event1, backup_event2
Output: None
1:   get original positions and events from backup
2:   assign event 1 to its original position 1
3:   assign event 2 to its original position 2
```

Inputs for this function are the backups of two events; the function has no return value. The function SwapTimeslots and SwapRooms are very similar in design to the function SwapEvents; thus an explicit description of the former two functions is omitted.

### 5.2.2 Graphical Examples

A simple example timetable is used in order to better illustrate the neighborhood moves. The initial (feasible) timetable is shown in Figure 2. For the sake of example it is assumed that the timetable is already complete (i.e. has a distance to feasibility of zero, all events are assigned) and that the search space is limited to the set of all feasible timetables, i.e. the neighborhood moves are not allowed to break the feasibility of the timetable.

| | Monday | | | Tuesday | | |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Math1 | | Acc1 | | | |
| Room2 | | Math2 | | Law3 | | Acc1 |
| Room3 | Law2 | | | Biology1 | | |
| Room4 | | Biology3 | Law1 | | Math1 | Biology3 |

**Figure 2 – The example timetable in its initial configuration**

The timetable includes events from three different curricula: one curriculum for law students, one for biology students and one for business students (who have to take both Math and Accounting courses). Courses of the same curriculum may not be assigned to the same timeslot (e.g. Law1 and Law2 must always be scheduled to different timeslots; Math2 and Acc1 must also be in different timeslots, etc.). Timeslot 1 is the first period on Monday, timeslot 4 is the first period on Tuesday, etc.

First an example for the event swap move is given: by random choice position (4, 2) – room 4 and timeslot 2 – and position (3, 4) are selected by the algorithm; it is then checked if the event in position (4, 2) – Biology3 - can be feasibly assigned to timeslot 4, and that the event in position (3, 4) – Biology1 - can be feasibly assigned to timeslot 2. As this is the case, the swap is performed, resulting in the new timetable shown in Figure 3.

|  | Monday |  |  | Tuesday |  |  |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Math1 |  | Acc1 |  |  |  |
| Room2 |  | Math2 |  | Law3 |  | Acc1 |
| Room3 | Law2 |  |  | Biology3 |  |  |
| Room4 |  | Biology1 | Law1 |  | Math1 | Biology3 |

**Figure 3 – The example timetable after a successful event swap**

Another event swap is attempted: this time the positions (3, 1) and (2, 6) are randomly selected. The event in position (3, 1) – Law2 – fits into timeslot 6; however, the event in position (2, 6) – Acc1 - cannot be assigned to timeslot 1, because there is already another event, which is part of the same curriculum (Math1 and Acc1 are both part of the curriculum for business students). This time the event swap fails and no changes are made to the timetable.

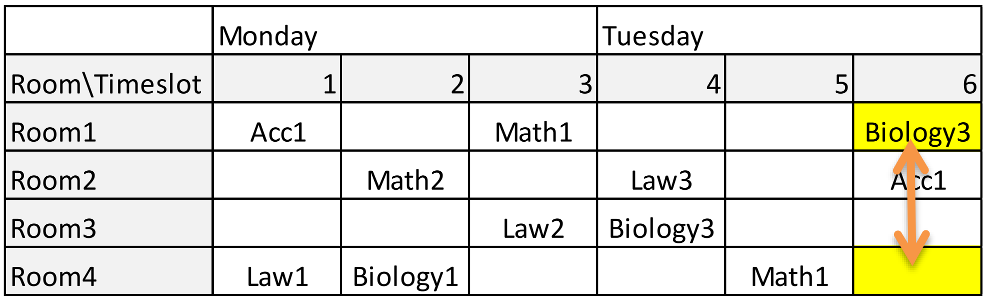
Swapping two timeslots can be illustrated with the following example: initially the algorithm selects two random timeslots, for example the timeslots 1 and 3. Next, the algorithm checks whether all of the events in timeslot 1 can be scheduled to timeslot 3 and vice versa. If there are any unavailability constraints, which prohibit the assignment of Math1 or Law2 to timeslot 1, then the swap fails. Likewise the swap would also fail, if there were any unavailability constraints, which prohibit the assignment of Acc1 or Law1 to timeslot 3. A successful timeslot swap would result in the new timetable in Figure 4.

|  | Monday |  |  | Tuesday |  |  |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Acc1 |  | Math1 |  |  |  |
| Room2 |  | Math2 |  | Law3 |  | Acc1 |
| Room3 |  |  | Law2 | Biology3 |  |  |
| Room4 | Law1 | Biology1 |  |  | Math1 | Biology3 |

**Figure 4 – The example timetable after a successful timeslot swap**

Swapping the room assignments of two events can be depicted by the following example: first the algorithm selects a timeslot and two different rooms at random; in the example timeslot 6 and the rooms 1 and 4 are chosen. Because the room capacity constraint is only a soft constraint, changing the room assignments can never result in a hard constraint violation; i.e. the room swap is always successful. The event in position

(1, 6) – in the example this position is empty, thus the event is of type "None" - is moved to position (4, 6) and vice versa. The result of the room swap can be observed in Figure 5.



| | Monday | | | Tuesday | | |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Acc1 | | Math1 | | | Biology3 |
| Room2 | | Math2 | | Law3 | | Acc1 |
| Room3 | | | Law2 | Biology3 | | |
| Room4 | Law1 | Biology1 | | | Math1 | |

**Figure 5 – The example timetable after a room swap**

## 5.3 Constructing an Initial Timetable

The first phase of building a feasible timetable is the heuristic construction of an at least partially complete timetable. During the construction process it is attempted to prioritize those events, which are likely to be difficult to assign to a position in the timetable and to choose for each event the position, which is most suitable for the event.

Before the construction heuristic itself can be explained, first two helper functions need to be introduced. The first function orders the given events by their respective priority; the aim of this function is to assign those events at an early stage of the construction, which are likely to be hard to assign. The idea for the heuristic ordering of the events is taken from (Lü & Hao, 2010). They use the following notation to describe their heuristic:

- TT: an incomplete but feasible timetable (i.e. not all events need to be assigned in the timetable, but the timetable mustn't have any hard constraint violations; initially the timetable might be completely empty)
- $apd_i(TT)$: the total number of available timeslots for course $c_i$ in the timetable TT;
- $aps_i(TT)$: the total number of available positions for course $c_i$ in the timetable TT;
- $nl_i(TT)$: the number of unassigned lectures of course $c_i$ in the timetable TT;

The events are lexicographically ordered in the following way:

(1) The highest priority is given to the courses with the smallest value of $apd_i(\text{TT}) \div \sqrt{nl_i(\text{TT})}$; i.e. courses, which consist of many lectures, but have only a small number of available timeslots are given the highest priority.

(2) If multiple courses have the same values for (1), higher priority is given to those with the smaller value of $aps_i(\text{TT}) * \sqrt{nl_i(\text{TT})}$; i.e. courses, which consist of few lectures, and have only a small number of available positions are given the highest priority.

(3) If multiple courses have the same values for (1) and (2), higher priority is given to those with the higher number of conflicts. A conflict with an event is defined - in this context - as the existence of another event, which has either the same teacher or is part of the same curriculum as said event. E.g. Math1 is being taught by Mrs. Black and Math2 is taught by Mr. Decker; if Mrs. Black is teaching five courses in total and Mr. Decker is teaching three courses in total, then Math1 has higher priority than Math2, because Math1 has four conflicts and Math2 has only two conflicts. In case when multiple courses also have the same number of conflicts, the courses are ordered alphabetically.

To illustrate how the heuristic ordering works, consider the following simple example; there are five courses (Math1, Math2, Biology1, Biology2 and Law1), which must be scheduled to a timetable with six timeslots and three rooms. Initially the timetable is empty. The details for each one of the courses are shown in Figure 6.

| Course | apd | aps | nl | apd/sqrt(nl) | aps*sqrt(nl) | conflicts |
|---|---|---|---|---|---|---|
| Math1 | 5 | 15 | 3 | 2.89 | 25.98 | 1 |
| Math2 | 5 | 15 | 4 | 2.50 | 30.00 | 1 |
| Biology1 | 6 | 18 | 3 | 3.46 | 31.18 | 1 |
| Biology2 | 2 | 6 | 2 | 1.41 | 8.49 | 1 |
| Law1 | 5 | 15 | 3 | 2.89 | 25.98 | 0 |

**Figure 6 - Example data for the heuristic ordering of courses**

For the course Math1, the above table shows that

- there are five timeslots available for Math1, i.e. one timeslot has an unavailability constraint for this course (**apd**);

- there are 15 available positions (five timeslots times three rooms), because the timetable is initially completely empty – the number of available positions is reduced as the timetable is filled with events (**aps**);
- the course Math1 consists of three lectures (**nl**);
- there is one other course which is part of the same curriculum as Math1; in this example Math2 is part of the same curriculum as Math1 (**conflicts**).

According to the values computed in Figure 6, the heuristic would create the following ordering (from highest- to lowest priority): Biology2 $\succ$ Math2 $\succ$ Math1 $\succ$ Law1 $\succ$ Biology1.

The second helper function orders the empty positions by their priority for a given event. Here, the goal is to assign the event to the position, which is most suitable for the event. A pseudo-code representation of the function is given below:

```
Function: OrderPositionsByPriority
Input: an event
Output: a list of positions
1:   for position in emptyPositions:
2:      if (event fits into position):
3:        add position to good_positions
4:      elif (event fits into timeslot):
5:        penalty = compute roomCapacity penalty score
6:        add (penalty, position) tuple to feasible_positions
7:   sort the feasible_positions by their penalty (lower is better)
8:   shuffle good_positions
9:   return [good_positions + feasible_positions]
```

The input to the function is an event; the output is a list of all possible positions for this event, ordered by their priority.

The loop in line 1 checks for each empty position whether the event fits into the position, i.e. if no hard constraint violations occur and if the roomCapacity soft constraint is not violated (line 2). When this is the case, the position is added to a list of good positions for this event (line 3). If the event fits into the timeslot (no hard constraint violations), but not into the position (the room capacity is exceeded, line 4), then the roomCapacity penalty is computed and both the position and the penalty are added to a list of feasible positions (line 5-6). Positions which lead to a hard constraint violation are skipped entirely.

The next step is to sort the list of feasible positions by their penalties in ascending order. Because all good positions are assumed to be equally well suited for the event, they are shuffled to add some randomness to the algorithm. Finally the function returns both the list of good positions and the list of feasible positions concatenated together. Note that either list could be empty (meaning that there are no good or no feasible positions for the event); when both lists are empty, the event cannot be feasibly assigned anywhere in the timetable.

As an example, it is assumed there is a timetable with the following empty positions: (1, 2), (3, 4), (2, 2) and (3, 5). Room 1 has a capacity of 30 students, room 2 of 20 and room 3 of 60. The event to be assigned next is "Math1", which has an unavailability constraint in timeslot 4 and is taken by 35 students. Given the previous heuristic the solver will produce the following order of positions for the event: (3, 5) $\succ$ (1, 2) $\succ$ (2, 2). Position (3, 4) violates a hard constraint and is thus ignored; position (3, 5) fits perfectly for Math1, while an assignment to position (1, 2) would leave five students- and to position (2, 2) would leave 15 students without a seat.

Using the above helper functions, the construction heuristic can assign the selected events one by one to positions in the timetable. The pseudo-code for the construction heuristic is given below:

```
Function: ConstructTimetable
Input: list of events
1:   while (there are still events left or timelimit is not exceeded):
2:      order events by priorty
3:      for i in range(number of events):
4:        event = remove the event with highest priority from EVENTS
5:        position = get position with highest priority for this event
6:        if (there is no feasible position):
7:          move the event to unplacedEvents
```

```
8:        else:
9:          assign the event to the position in the timetable
10:  num_unplaced = count the number of unplacedEvents
11:  shuffle the unplacedEvents
12:  for i in range(num_unplaced):
13:    position = chose a random position among the occupied positions
14:    event = remove the event at that position
15:    move the event back to EVENTS
16:    add the position to newly_empty_positions
17:  for i in range(num_unplaced):
18:    event = remove an event from unplacedEvents
19:    for every position in newly_empty_positions:
20:      if the event can be feasibly assigned to the position:
21:        assign event, remove position from newly_empty_positions
22:    if the event couldn't be assigned anywhere:
23:      move the event back to EVENTS
24:  return number of events
```

The input to the construction heuristic is a list of all events; the output is the distance to feasibility, i.e. the number of events, which couldn't be feasibly assigned to a position.

The heuristic starts by checking if the list of events is not empty (line 1); if it is empty, that means all events have been feasibly assigned to a position in the timetable and the construction is finished; if the list of events is not empty, then there are still unassigned events left.

The assignment starts by first ordering the unassigned events by their respective priority (line 2). The event with the highest priority is then selected and the position with the highest priority among the empty positions for this event is computed; if there is no feasible position for this event (i.e. the position with the highest priority is of type None), then the event is moved to a list of unplaced events. Otherwise, the event is assigned to the position in the timetable (lines 4-9). The program repeats the same process for the event with the next highest priority (line 3), etc.

When the list of events is empty, any event should be either assigned to a position in the timetable or have been moved to the list of unplaced events. To add more randomness, the list of unplaced events is shuffled (line 11); then for each unplaced event, a random, occupied position is selected in the timetable and the event at that position is removed from the timetable (lines 12-16). The idea is that the currently unplaced events could not be assigned to any of the empty positions; so it is necessary to create new empty positions, hoping that the currently unplaced events a) fit into one of those newly empty positions (lines 17-23) and b) the events, which were previously assigned to those positions, can be placed elsewhere.

The program restarts in line 1 with a list of all the events, which either could not be placed anywhere or which were randomly removed from the timetable in line 14. When the list of events is empty (i.e. all events have been assigned to a position in the timetable), the program terminates and returns the distance to feasibility, which is - in this case - equal to zero. Otherwise the program loops until the time limit is reached.

For the sake of example it is assumed that the construction heuristic produced the timetable shown in Figure 7, by first ordering the events by priority and then assigning the events one after another to the respective best empty position; the algorithm is currently in line 10 of the pseudo-code. However the events Biology1 and Law2 could not be placed anywhere so far. In the next step the algorithm randomly selects two occupied positions - (1, 3) and (2, 4) - and removes the events at those positions. Then it attempts to assign Biology1 and Law2 to one of the newly empty positions. After the first loop of the construction heuristic finishes, the timetable shown in Figure 8 has been created and the events Acc1 and Law3 are currently without an assignment.

| | Monday | | | Tuesday | | |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Math1 | | Acc1 | | | |
| Room2 | | Math2 | | Law3 | | Acc1 |
| Room3 | Law2 | | | Biology1 | | |
| Room4 | | Biology3 | Law1 | | Math1 | Biology3 |

**Figure 7 - Example timetable during the first loop of the construction heuristic; the randomly selected, occupied positions are marked in yellow**

|  | Monday |  |  | Tuesday |  |  |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Math1 |  | Biology1 |  |  |  |
| Room2 |  | Math2 |  | Law2 |  | Acc1 |
| Room3 | Law2 |  |  | Biology1 |  |  |
| Room4 |  | Biology3 | Law1 |  | Math1 | Biology3 |

**Figure 8 - Example timetable after the first loop of the construction heuristic; the events in (1, 3) and (2, 4) were replaced by two unassigned events**

Potentially this algorithm is able to construct a feasible timetable, given enough runtime. However, the question arises whether it is more efficient to let this heuristic loop until a feasible timetable has been found, or whether it is faster to let a metaheuristic achieve feasibility once the initial construction has been done; an experiment trying to answer this question is conducted in chapter 6.2.2.

## 5.4 Achieving Feasibility

Whenever the construction heuristic didn’t produce a complete timetable within the time limit (there are still unassigned events left), the second phase of the algorithm - the feasibility phase - starts. During this phase of the algorithm, a metaheuristic is utilized to guide the algorithm towards a complete and feasible assignment of events. The feasible search space is explored via a series of neighborhood moves. A new timetable is considered an improvement over the old one, if the distance to feasibility of the new timetable is smaller than that of the old one; i.e. when fewer events are left without an assignment in the new timetable, the new timetable is better than the old one. E.g. if in the old timetable, the events Math1 and Law3 are not assigned to any position in the timetable and in the new timetable only the event Biology1 is not assigned anywhere, then the new timetable is an improvement.

### 5.4.1 Tabu Search

One of the metaheuristics used during the feasibility phase is Tabu Search. During the search, two neighborhood moves are exploited: swapping two timeslots and swapping two events in the timetable. A pseudo-code description of the timeslot-swapping function is given below:

```
Function: TimeslotSwapTS
Input: a tabu list of timeslots, the last distance to feasibility
Output: True OR False
1:   ts1, ts2 = randomly chose two timeslots
2:   if (the move [ts1 -> ts2] or [ts2 -> ts1] is on the tabu list):
3:      return False
4:   else:
5:      add [ts1 -> ts2] and [ts2 -> ts1] to the tabu list
6:   make a backup of EVENTS and unplacedEvents
7:   swap the timeslots ts1 and ts2
8:   if the swap was not successful:
9:      return False
10:  for each event that has been swapped:
11:     if (the assignment of the event is not feasible anymore):
12:        remove the event from the timetable and add to unplacedEvents
13:  for each event in unplacedEvents:
14:     try to assign the event to an empty position
15:  distance = compute the new distance to feasibility
16:  if (the new distance is bigger than the last distance):
17:     reverse the timeslot swap and restore the backups
18:     return False
19:  update the last distance to the new (smaller) distance
20:  return True
```

The inputs to the function are a list of tabu timeslots and the last distance to feasibility; the output is either True or False, indicating if the swap has been successful or not.

At first the algorithm randomly selects two timeslots (line 1). It then checks whether there is an entry in the tabu list for the swap of the two selected timeslots (line 2); if there is an entry, the neighborhood move has already been done before and the algorithm terminates at this point; if the move is not on the tabu list, then it is added to the tabu list, so it can be recognized when the next swap occurs (line 5).

A backup of all the necessary variables is made (line 6); afterwards the timeslots are swapped, i.e. all events in the 1st timeslot are moved to the 2nd timeslot and vice versa (line 7). For each event that has been moved, it is checked whether or not the new position is feasible (i.e. if moving an event from one timeslot to the other violates any hard constraints). If the new position is not a feasible assignment for the event, the event is removed from the timetable and added to the list of unplaced events (lines 10-12).

In the next step, for each event in the list of unplaced events, it is attempted to assign the event to one of the empty positions (lines 13-14). Afterwards the current distance to feasibility is computed. If the current distance is bigger (= worse) than the last distance, the swap is reversed and all backups are restored. Otherwise the algorithm continues by overwriting the last distance with the current distance.

As an example, it is assumed that the construction heuristic created the initial timetable shown in Figure 8. The example tabu list is of length 2 and has the entries [(2→3), (3→2)]. Furthermore it is assumed that there are currently two unassigned events (Math3 and Biology2) – the timetable has a distance to feasibility of two. In this timetable, the function TimeslotSwapTS randomly selects the timeslots 1 and 4. Because the move is not tabu, it is added to the tabu list; the tabu list now contains the entries [(2→3), (3→2), (1→4), (4→1)]. Incidentally the event Math1 cannot be feasibly assigned to timeslot 4; it is thus removed from the timetable and added to the unassigned events (Figure 9).

| | Monday | | | Tuesday | | |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | | | Biology1 | | | |
| Room2 | Law2 | Math2 | | | | Acc1 |
| Room3 | Biology1 | | | Law2 | | |
| Room4 | | Biology3 | Law1 | | Math1 | Biology3 |

**Figure 9 - Example timetable during TimeslotSwapTS; the swapped timeslots are marked in yellow, the red border signifies the removal of an event**

The algorithm tries to assign the unassigned events to empty position in the timetable. Because of the timeslot swap, the unassigned events Math3 and Biology2 can be scheduled to timeslot 4. Math1 can be scheduled to timeslot 1 (Figure 10). As a result the new timetable now has a distance to feasibility of zero.

|  | Monday |  |  | Tuesday |  |  |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Math1 |  | Biology1 | Math3 |  |  |
| Room2 | Law2 | Math2 |  | Biology2 |  | Acc1 |
| Room3 | Biology1 |  |  | Law2 |  |  |
| Room4 |  | Biology3 | Law1 |  | Math1 | Biology3 |

**Figure 10 - Example timetable at the end of TimeslotSwapTS; the green positions signify new successful assignments**

Apart from the timeslot-swapping function, the Tabu Search neighborhood also makes use of an event-swapping function. However, in order to make the thesis not overly long, no description of the eventSwapTS function is given at this point; instead the pseudo-code for the eventSwapSA function (which is mostly identical to eventSwapTS, apart from some Simulated Annealing specific details) is described in chapter 5.4.2.

Both neighborhood functions are managed by a paramount guiding function. The pseudo-code for this function is given below:

```
Function: TabuSearchHard
Input: the maximum length of the tabu lists
Output: the smallest found distance, the best found timetable
1:   while (distance to feasibility > 0 and time limit not exceeded):
2:      if (the length of the tabu timeslots list > max length):
3:        remove the oldest entry
4:      if (the length of the tabu positions list > max length):
5:        remove the oldest entry
6:      randomly choose between two possible moves:
7:        either: swap timeslots
8:        or: swap positions
9:   return (the smallest distance to feasibility, the best timetable)
```

The algorithm loops until a feasible timetable with distance zero has been found or the time limit is exceeded. When the maximum length of either tabu list is reached, the oldest entries are removed (lines 2-5). Then a random choice is made for the next neighborhood move (lines 6-8). The output of the algorithm is the smallest distance to feasibility and the corresponding best timetable.

### 5.4.2 Simulated Annealing

The second metaheuristic, which is utilized to search for a feasible timetable, is Simulated Annealing. It uses the same neighborhood structure as Tabu Search, swapping two timeslots and swapping two events. A pseudo-code description of the event-swap function is presented below:

```
Function: EventSwapSA
Input: The current temperature T, the last and best distances to
feasibility
Output: True OR False
1:   pos1, pos2 = randomly select two positions in the timetable
2:   make a backup of EVENTS and unplacedEvents
3:   swap the positions of two events
4:   if the swap was not successful:
5:      return False
6:   for both events:
7:      if (the assignment to the new position is not feasible):
8:        remove the event and add it to unplacedEvents
9:   for each event in unplacedEvents:
10:     try to assign the event to an empty position
11:  distance = compute the new distance to feasibility
12:  delta_distance = compute (distance – last distance)
13:  if (the new distance is bigger than the last distance):
14:     if (a random variable x > exp(-delta_distance/T)):
15:       reverse the event swap and restore the backups
16:       return False
17:  update the last distance to the new (smaller or bigger) distance
18:  if (the current distance is smaller than the best distance):
19:     best_timetable = make a copy of the current timetable
20:     update best_distance to the current distance
```

```
21:  return True
```

The inputs to the function are the current temperature T and the last distance to feasibility; the output is True, if the swap was successful or False otherwise.

After randomly choosing two positions in the timetable and making the necessary backups (lines 1-2), the two selected positions are swapped, i.e. the event at the 1st position is moved to the 2nd position and vice versa. The algorithm then checks whether the newly assigned position is feasible for the event; if it is not feasible, the event is removed from the timetable and added to the list of unplaced events (lines 6-8). An attempt is made to assign the unplaced events to empty positions and, following that, the current distance to feasibility, as well as a variable delta_e (the current distance minus the last distance) are computed (line 12).

While the above steps were largely the same as for Tabu Search, the next part is unique to Simulated Annealing: if the new solution is worse than the old one (i.e. the current distance to feasibility is bigger than the last distance), the event swap is reversed with probability $1 - e^{-\Delta_{distance}/T}$ and the worse solution is retained with probability $e^{-\Delta_{distance}/T}$ (lines 13-16). When the swap is not reversed, the last distance is updated to the current distance. With Simulated Annealing, it is necessary to check whether a global optimum has been found, because the algorithm also accepts worsening moves with a certain probability. If a new global optimum has been discovered, a copy of the current timetable is made and the current distance is recorded as the best distance so far (lines 18-20).

As an example, it is assumed that the construction heuristic created the initial timetable shown in Figure 8. In the example the current temperature is at 5. Furthermore it is assumed that there are currently two unassigned events (Math3 and Biology2). The function EventSwapSA randomly selects the positions (2, 2) and (4, 5). Because the event Math2 cannot feasibly be assigned to timeslot 5, it is removed from the timetable (Figure 11). The algorithm tries to schedule the unassigned events to empty positions in the timetable, but it can’t find any feasible assignments for any of the events. The new distance to feasibility is 3 (= three unassigned events). The value for Δ_distance is consequently 1; for the sake of example, the value of the random variable is 0.3. Because $0.3 < e^{-1/5} = 0.819$, the new (worse) timetable is not rejected.

|  | Monday |  |  | Tuesday |  |  |
|---|---|---|---|---|---|---|
| Room\Timeslot | 1 | 2 | 3 | 4 | 5 | 6 |
| Room1 | Math1 |  | Biology1 |  |  |  |
| Room2 |  | Math1 |  | Law2 |  | Acc1 |
| Room3 | Law2 |  |  | Biology1 |  |  |
| Room4 |  | Biology3 | Law1 |  | 🟨 | Biology3 |

**Figure 11 - Example timetable at the end of EventSwapSA; the swapped positions are marked in yellow, the red border signifies the removal of an event**

A detailed description of the TimeslotSwapSA function is not given. However, a pseudo-code description for the largely similar TimeslotSwapTS function is available in chapter 5.4.1.

Furthermore, a paramount guiding function is utilized to manage the neighborhood moves and to compute some additional variables. The following pseudo-code describes the heuristic, which is used to guide the Simulated Annealing process:

```
Function: SimulatedAnnealingHard
Input: minimum (Tmin) and maximum (Tmax) temperature, number of steps
Output: the smallest found distance, the best found timetable
1:   step = 0
2:   Precompute Tfactor for exponential cooling from Tmax to Tmin
3:   no_improvement = 0
4:   while (distance to feasibility > 0 and time limit not exceeded):
5:      if (no improvement has been made for the last 10 iterations):
6:        step = 0
7:      set the temperature to (Tmax * exp(Tfactor * step / steps))
8:      if (the temperature is still above Tmin):
9:        step += 1
10:     randomly choose between two possible moves:
11:       either: swap timeslots
12:       or: swap positions
13:     if (the new distance is worse than the last distance):
14:       no_improvement += 1
```

```
15:     else:
16:       no_improvement = 0
17:  return (the best distance and the best timetable)
```

The input values to this function are the minimum and maximum temperature, as well as the number of steps; the steps determine how fast the temperature should be decreased from the maximum to the minimum (i.e. how many intermediary values lie between Tmax and Tmin). The function outputs the smallest distance to feasibility, which has been found, and the corresponding timetable.

Initially the step counter and the no_improvement counter (which counts the number of iterations since the last improving solution has been found – this is used to determine if the search is stuck in a local optimum) are set to zero. The cooling factor for exponential cooling: $T_{factor} = \log(\frac{T_{max}}{T_{\min}})$ is also computed (line 2).

The program loops until the distance to feasibility is equal to zero or the time limit is exceeded. Within the loop, the program first checks for how many iterations there hasn't been an improvement in the solution quality. If a certain threshold is reached, the step counter is reset to zero; as a consequence the search continues again at the maximum temperature, in the hope that it will escape from a local optimum towards different regions of the search space (line 5-6).

The current temperature is computed according to equation (12) (line 7). If the temperature is still above the minimum temperature, the step counter is increased by one (lines 8-9). Afterwards a random choice is made between swapping timeslots and swapping positions (lines 10-12). Finally, if no improvement has been made (i.e. the new distance to feasibility is not smaller than the last distance), the no_improvement counter is incremented; otherwise it is reset to zero (lines 13-16).

$$T = T_{max} * e^{T_{factor} * \frac{step_i}{step_{max}}} \qquad (12)$$

## 5.5 Improving the Solution

During the improvement phase, the same two metaheuristics are used as during the feasibility phase: Tabu Search and Simulated Annealing. The program code for the improvement algorithms is mostly the same as for the feasibility algorithms, with three notable differences:

- an additional neighborhood is utilized (swapping two rooms);
- the search is limited to feasible regions of the search space, i.e. the neighborhood moves are required to preserve the feasibility of the timetable at all times;
- instead of the distance to feasibility (which must always be zero during the improvement phase), the penalty score is used to decide whether to discard the new timetable or keep it. A new timetable, with a lower penalty score than the old one, is considered an improvement.

The feasibility of the timetable is preserved by checking whether the swap leads to a feasible assignment of events before it is actually executed; if the swap is not feasible, the swap function terminates immediately - an infeasible assignment is not allowed to be made at any point. Code-wise this is done by passing a parameter preserve_feasibility to the neighborhood function (see chapter 5.2).

The only change to the two guiding functions (TabuSearchHard and SimulatedAnnealingHard) is the addition of the room swap as another possible neighborhood move; instead of randomly selecting between two neighborhood moves, the program now has the choice between three moves. Apart from these changes, the program code is identical to the program code used in the feasibility phase.

## 5.6 Validating the Output

In order to validate the output, the program makes use of the official validator for the ITC-2, which is provided by (Bonutti, et al., 2012) on their website[7]. First an output file needs to be produced from the timetable dictionary. The output format is given by the specifications of ITC-2 track 3 in (Di Gaspero, et al., 2007); an example for an output file is shown in Figure 33 (Appendix, page 74). After the output file has been created,

[7] http://tabu.diegm.uniud.it/ctt/

the official validator is used to validate the file. The verdict returned by the validator is in turn captured by the solver. Of highest interest are obviously the number of hard constraint violations and the soft constraint penalty score found by the validator. Because the solver does not allow any hard constraint violations to occur - besides having unassigned events (when an event violates a hard constraint, it is automatically unassigned from the timetable) - the number of hard constraint violations found by the validator is equal to the distance to feasibility. Furthermore, the soft constraint penalty score has already been computed by the solver during the improvement phase. The reasons for still using the official validator are the more detailed output (the validator provides a list of all events which are involved in a hard or soft constraint violation) and the added credibility.

In the next step, the validator output is written to a text document. Recorded are the number of hard constraint violations, the penalty score, the construction time, the time until feasibility is achieved and the time used by the soft constraint improvement phase.

For every problem instance, the best solution timetable and the most recent solution timetable are saved to the folder “solutions”. During each execution the solver compares the current solution to the best solution. If the current solution is better than the best solution, than it becomes the new best solution; otherwise it is saved as the most recent solution. The ranking, to determine which solution is superior, is the following:

1) The solution with the lowest distance to feasibility is better;
2) If both have the same distance to feasibility, the solution with the lowest soft constraint penalty score is better (in case of equal penalty scores, the old timetable is retained).

To ensure that the solver is able to create timetables in accordance with the UD2 formulation, a preliminary test run was started, using the official competition instances. The solver was set to run for a maximum of 60 seconds, of which 10 seconds were reserved for the construction phase. If the solver found a feasible timetable during the construction phase, the program immediately started the improvement phase; otherwise the feasibility phase was started. Tabu Search was used for both the feasibility- and improvement phases.

As a result the solver managed to find feasible timetables for 11 out of the 21 problem instances, while for the remaining problems - where no feasible timetable was found - the number of hard constraint violations (as determined by the validator) was equal to the distance to feasibility – just as expected. It was determined that the solver did in fact produce timetables in accordance with the competition rules.

# 6. Evaluation

The goal of this chapter is to provide conclusive test data, in order to answer the following questions:

- What are the optimal parameter settings for the two tested metaheuristics?
- Which of the tested (meta)heuristics is most successful at finding a complete and feasible timetable?
- Which of the tested metaheuristics is most successful at improving a feasible timetable?
- What are the effects of changes to the neighborhood structure and the runtime?
- How is the performance of the solver in comparison to other timetabling programs used at the ITC-2?

The presented graphs in this chapter should generally be interpreted as “less is better”; i.e. the shorter the runtime or the lower the penalty score, the better the performance.

## 6.1 Parameter Setting

Before the computational experiments with the solver are made, it is first necessary to determine the optimal values for several parameters. The parameters in question are: the length of the tabu list (for Tabu Search), the maximum- and minimum temperature and the number of steps (for Simulated Annealing).

To test the parameter setting during the improvement phase, five problem instances are used: instance number 1, 3, 4, 11 and 18. The reason for choosing these particular instances is that, on the one hand, an initial feasible solution for them can be found in relatively short time, while they are, on the other hand, not trivial to solve (like the test instances). First the construction heuristic is used to build a complete feasible timetable (i.e. with a distance to feasibility of zero) for each problem instance; the resulting initial timetable is then repeatedly improved by either of the metaheuristics, changing the parameters every several runs. On each run, the improvement algorithm is allowed to loop for 1000 iterations; when the iteration limit is reached the best found solution is recorded and the next run starts again with the same initial timetable.

To test the parameters during the feasibility phase, the construction heuristic first runs once (is allowed to run for one loop), and the resulting initial (possibly incomplete) timetable is then completed by the metaheuristic under test. For this test only problem instances 3 and 4 are used, because for instances 1, 11 and 18 the construction heuristic usually finds a complete solution immediately, leaving no possibility for the feasibility phase to kick in (for instances 3 and 4 the construction of an initial solution results only in a few cases in a complete timetable). At the same time, the initial construction for instances 3 and 4 is relatively fast, compared to the remaining 16 competition instances. The timetable building is repeated 30 times per instance and per tested parameter. For each run, the runtime until a complete solution is discovered during the feasibility phase is recorded. At the end, the lowest average runtime is used to determine the optimal parameter setting.

### 6.1.1 Optimal Length of the Tabu List

The results of testing tabu list lengths of 0, 100, 200, 300 and 400 during the improvement phase are given in Figure 12:

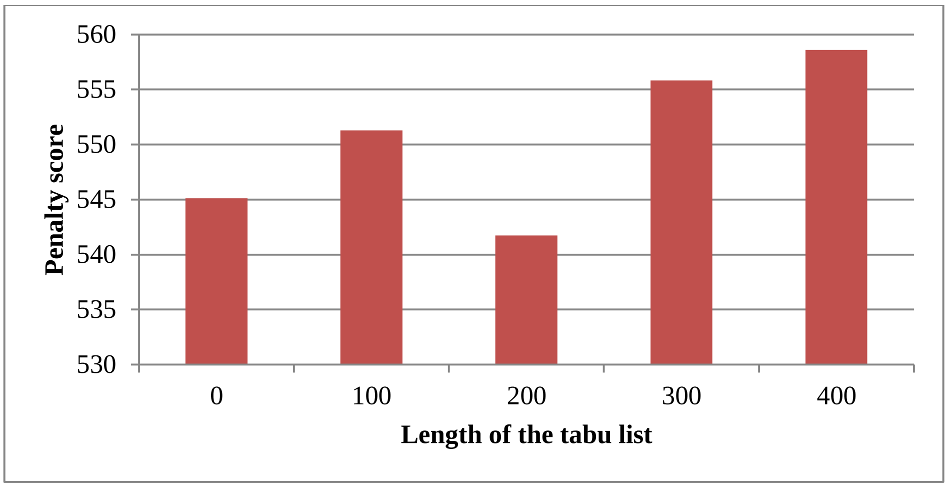


**Figure 12 - Average penalty scores for different lengths of the tabu list**

A tabu list length of 0 can be interpreted as a pure Hill Climbing algorithm. The results of the test indicate that 200 is the optimal length for the tabu list, although the differences between the various lengths are very small (e.g. only three penalty points difference between a list length of 200 and of 0). However, when looking at the results for each individual problem instance, the answer becomes less clear: some problem instances achieve lower penalty scores with a longer tabu list, while others have generally lower penalty scores with shorter tabu lists (see Figure 13). Overall it can be concluded that the length of the tabu list has – if at all – only a minimal impact on the final solution quality.

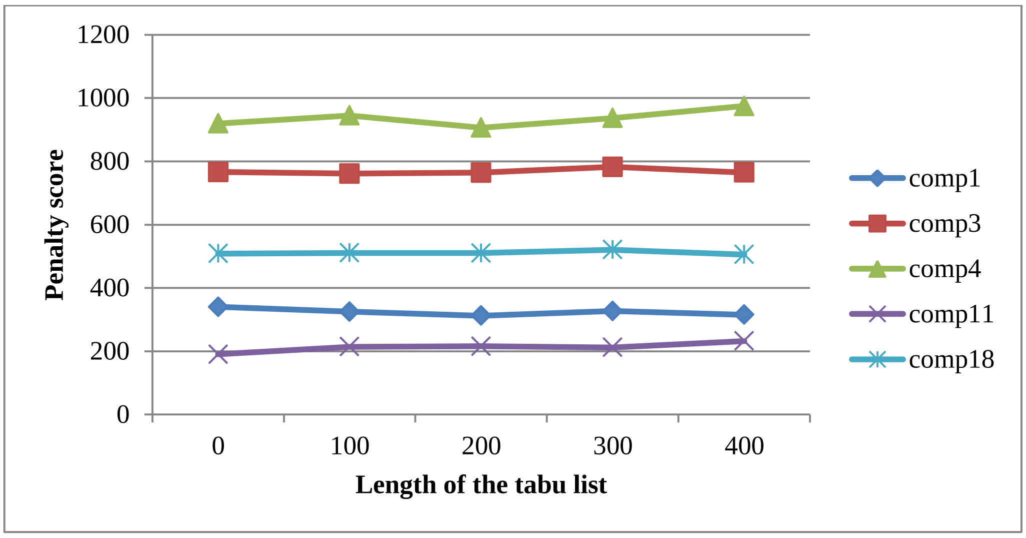


**Figure 13 - Average penalty scores for different lengths of the tabu list (detailed)**

The results of testing different tabu list lengths during the feasibility phase are summarized in Figure 14. Here, instead of measuring the penalty score, the runtime of the algorithm until it has found a complete solution is measured. In the 480 test runs for this experiment, the construction heuristic managed to produce a complete initial solution 36 times. As before, the data does not allow drawing a definitive conclusion as to which tabu length is optimal. The total average runtimes fluctuate closely around five seconds in most cases, with significant deviations only with tabu lengths of 50 and 200. The chart shows no obvious trend nor gives a clear indication, whether one of the length parameters is superior or inferior to the others. Thus a second test is deemed necessary.

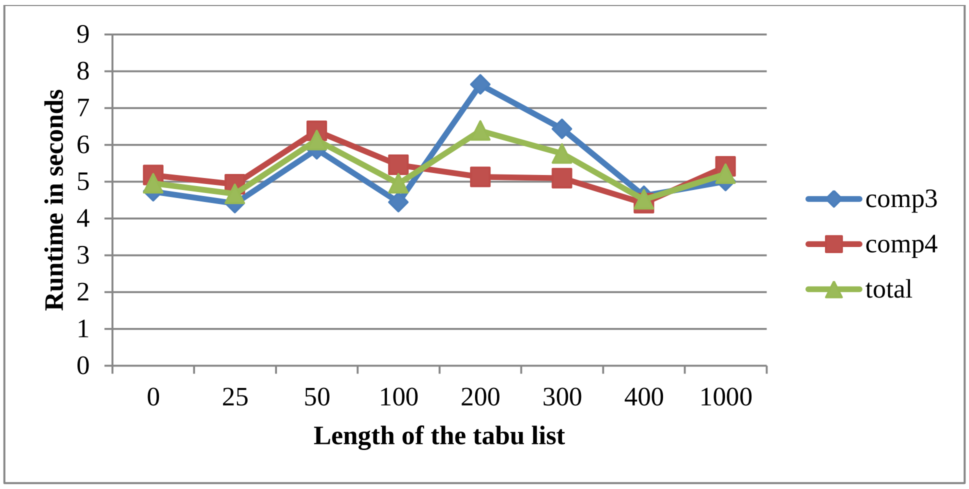


**Figure 14 - Average runtime until a complete solution is found, for different lengths of the tabu list (instance 3 and 4)**

Another test run is performed, this time using problem instances 2 - 7. Each tabu list length parameter is tested 10 times on each of the 6 instances; the resulting average runtimes are depicted in Figure 15.

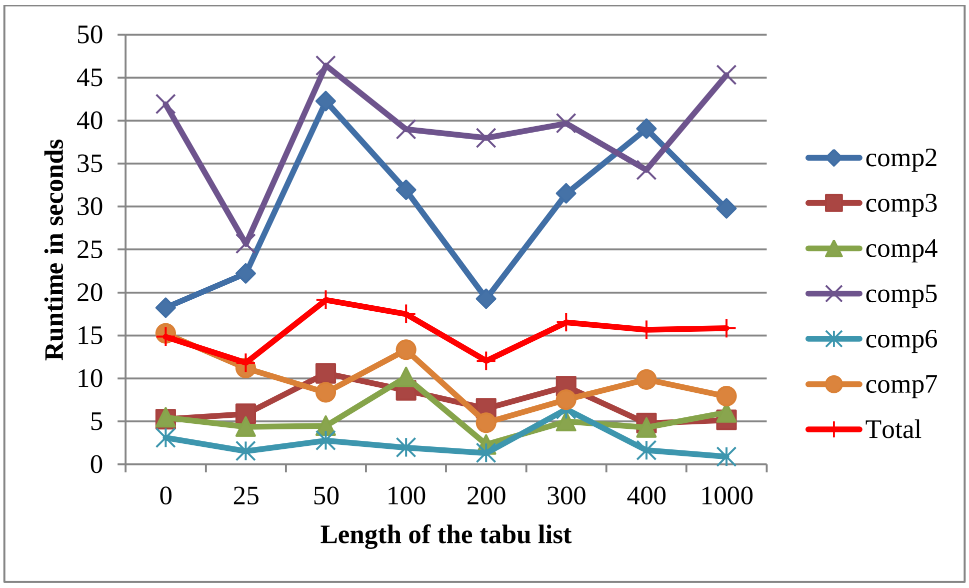


**Figure 15 - Average runtime until a complete solution is found, for different lengths of the tabu list (instance 2-7)**

This time the lowest total average runtimes are achieved with length parameters 25 and 200 – a result that can be traced back to the impact of instance 5 and 2 respectively. As before, there is no apparent trend nor clearly optimal length parameter; ultimately the length of the tabu list is set (more or less arbitrarily) to 300 for both the feasibility- and the improvement algorithms.

### 6.1.2 Optimal Temperatures and Number of Steps

The next parameter to be determined is the maximum temperature for the Simulated Annealing algorithm. The results of the first test – set during the improvement phase - are given in Figure 16.

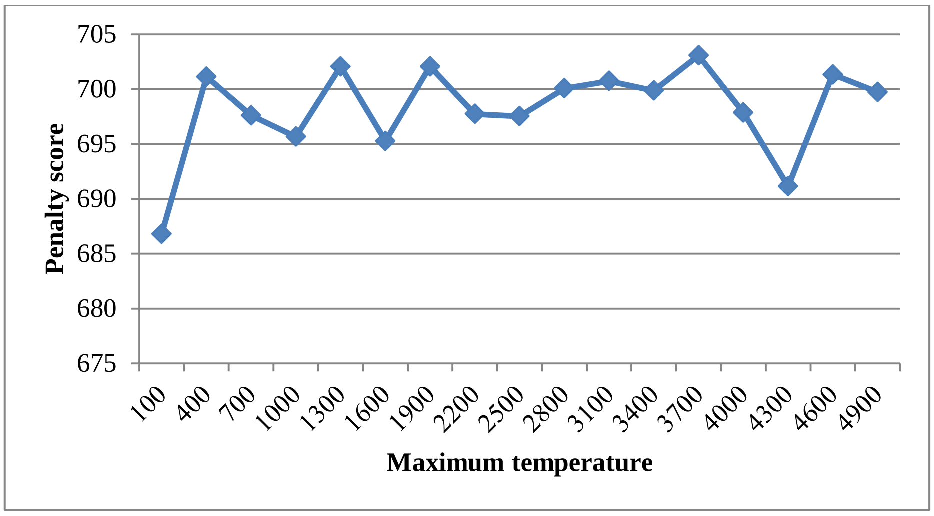


**Figure 16 - Average penalty scores for different maximum temperatures (wide range, 100 to 5000)**

The penalty scores at the various temperature levels are all very close to each other; no trend is apparent by the graph. However, the best score is achieved at the lowest tested temperature of 100 – it is concluded that a second test, using only starting temperatures between 5 and 95, is necessary. The results of the second test are depicted in Figure 17:

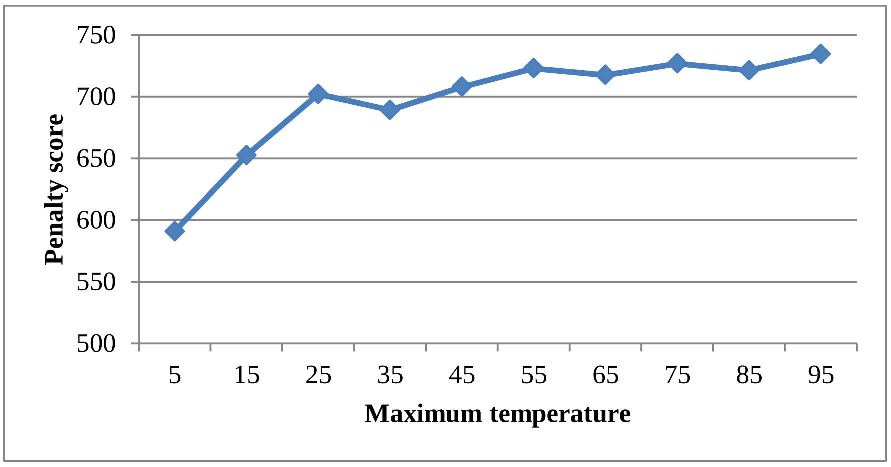


**Figure 17 - Average penalty scores for different maximum temperatures (narrow range, 5 to 95)**

The evidence suggests that the algorithm performs better at lower maximum temperatures; the best average penalty score is achieved at the lowest tested temperature of 5 and the graph shows a slight upward trend. It is interesting to remember that the probability of accepting a worsening solution is lower at lower temperatures; in the extreme case of a temperature of 0, the Simulated Annealing algorithm performs exactly like a Hill Climbing algorithm (it never accepts worsening moves). The above graph indicates that the solution quality gets better as the algorithm behaves more similar to a pure Hill Climbing metaheuristic – allowing the SA algorithm to perform a lot of worsening moves is apparently counterproductive for the final solution quality. Throughout the remainder of the thesis the optimal parameter value of the maximum temperature is assumed to be 5.

Results of the parameter test for the maximum temperature during the feasibility phase are shown in Figure 18. Of the 400 conducted runs, 31 had a distance to feasibility of zero after the initial construction and were consequently excluded from the dataset. The average runtime measurements for the remaining runs vary between 9 and 16 seconds, although no trend becomes apparent. The lowest average runtime of 8.7 seconds was found with a maximum temperature of 15. However when the experiment was repeated, the assumption that this could be the optimal parameter value could not be confirmed: with the new data the average runtime with maximum temperature 15 went up to 14.9 seconds. In conclusion the maximum temperature does not seem to have an obvious impact on the performance of the Simulated Annealing algorithm in the feasibility phase. Thus, for reasons of simplicity, the optimal maximum temperature is assumed to be 5 (the same as for the improvement phase) throughout the remainder of experiments. Furthermore, it is decided that the optimal minimum temperature and the optimal number of steps are determined only for the improvement phase; those parameter settings are then used for the feasibility phase as well.

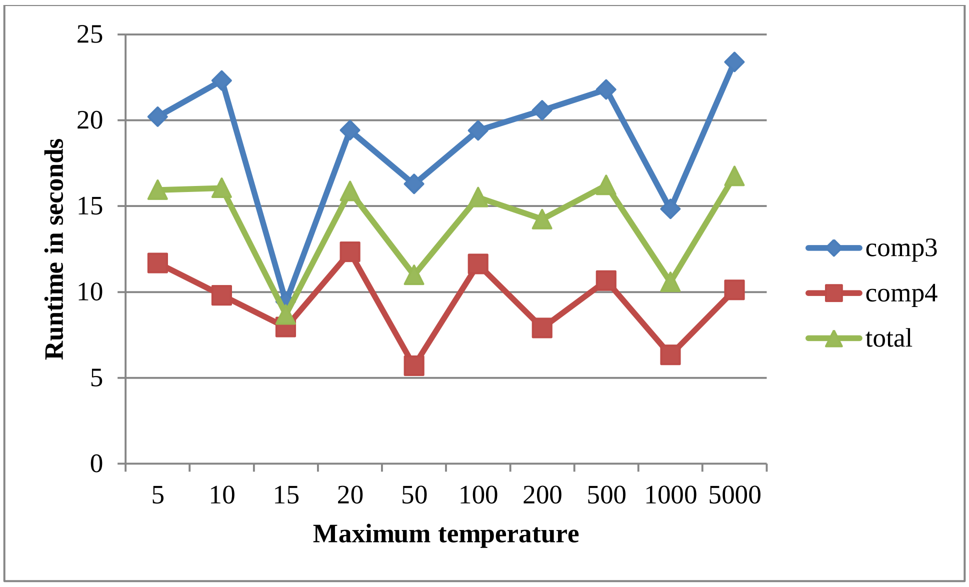


**Figure 18 - Average runtime until a complete solution is found, for different maximum temperatures; further tests could not reproduce the low average runtime at a maximum temperature of 15**

The parameter test for the minimum temperature during the improvement phase - with the maximum temperature set to 5 – is presented in Figure 19.

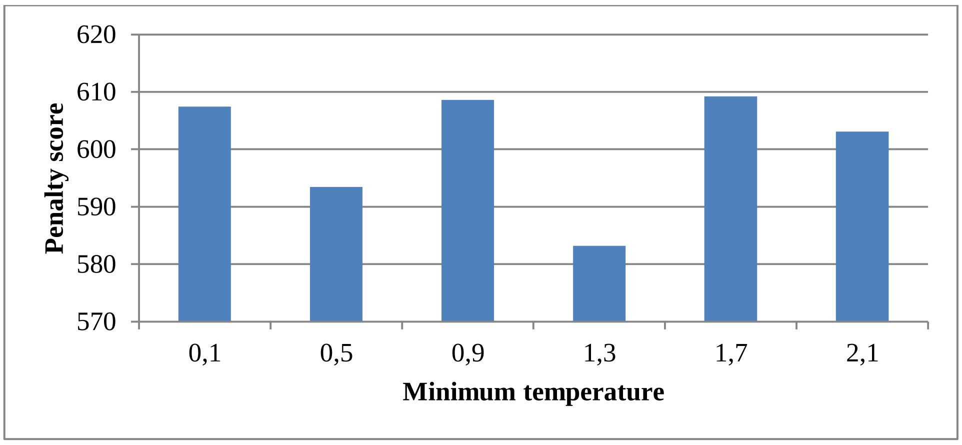


**Figure 19 - Average penalty scores for different minimum temperatures**

No trend or real outlier is apparent in the chart. The best average penalty is achieved with a minimum temperature of 1.3; upon closer examination of the test data this arises mostly from the results for a single instance – the other instances show no common behavior. In conclusion the minimum temperature doesn’t seem to have a big impact on the solution quality at all. For the remainder of the experiments the minimum temperature parameter is fixed at 1.3.

As for the optimal number of steps (i.e. the number of intermediary values between the maximum- and the minimum temperature), a plot of the test data is given below in Figure 20.

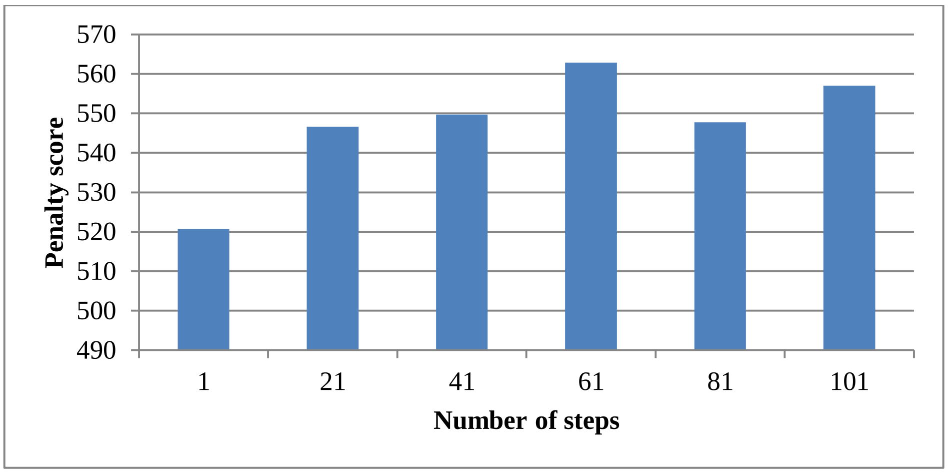


**Figure 20 - Average penalty scores for different numbers of steps**

The best average penalty is found with one step between the maximum- and minimum temperature, although the evidence is not very sound: when looking at the data for the individual problem instances, there is no clear indication as to which amount of steps is optimal (see Figure 21). The only common observation for all tested problem instances is that using 21 steps always lead to worse penalties than using 1 step. For the following experiments an optimal number of steps of 5 is assumed, to allow the Simulated Annealing metaheuristic to work and to not make its behavior too similar to a pure Hill Climbing metaheuristic.

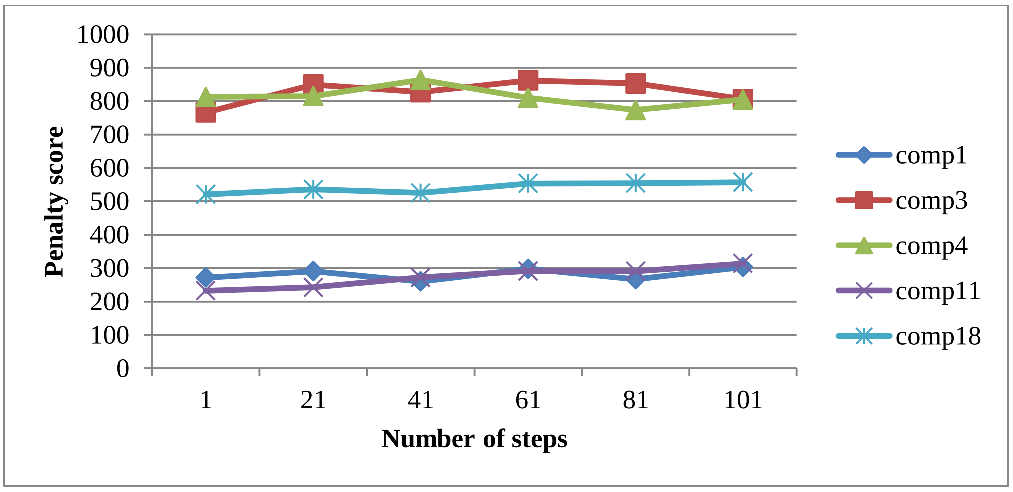


**Figure 21 - Average penalty scores for different numbers of steps (per instance)**

## 6.2 Computational Experiments

All of the computational experiments are performed on an Intel Core 2 Duo processor with 2.1 GHz clock speed, 800 MHz FSB and 4 GB RAM. The operating system is a 64-bit Version of Windows 7; the 64-bit version of the Python 3.3 interpreter is used to run the solver.

With the benchmarking program from the official website of the ITC-2[8], the maximum allowed runtime for the solver program on the author's computer is measured to be 442 seconds; i.e. if the author's computer would have been used at the ITC-2 to run the submitted programs, the programs would have been terminated after a runtime of 442 seconds. Using this maximum runtime, it is possible to more accurately compare the solution quality produced by the author's solver with the solution quality produced by the participants of the ITC-2. The experiments are conducted on the new versions of the official competition instances (labeled "comp") by (Bonutti, et al., 2012), which were already introduced in chapter 3.2.

### 6.2.1 Achieving Feasibility: Simulated Annealing vs. Tabu Search

The aim of the first experiment is to find out, which algorithm is best suited to build a complete and feasible timetable. The performance is judged by two criteria:

- If a complete and feasible timetable is found, the faster algorithm is better;
- If no complete and feasible timetable is found, the algorithm, which produced the timetable with the lowest distance to feasibility until the time limit was exceeded, is better.

The setup for the first experiment, to determine which of the two metaheuristics performs better, is the following: For each problem instance, first the construction heuristic is run for one loop; then the metaheuristic under test is given 60 seconds to find a complete, feasible timetable. Each run is repeated 30 times for each of problem instances 3, 4, 8, 11 and 18 and for each metaheuristic. The parameter settings for the metaheuristics are a length of the tabu list of 200 and a maximum temperature of 5.

[8] http://www.cs.qub.ac.uk/itc2007/index_files/benchmarking.htm

It turns out that the construction heuristic – after only looping once - is able to find a complete feasible timetable in 153 of 300 runs (51% of all runs). Upon closer inspection, the construction heuristic always (in 100% of the runs) finds a complete solution for the problem instances 11 and 18; it finds a complete solution in 42% of the runs for problem instance 8 and in 7% of the runs for both problem instances 3 and 4 (see Figure 22).

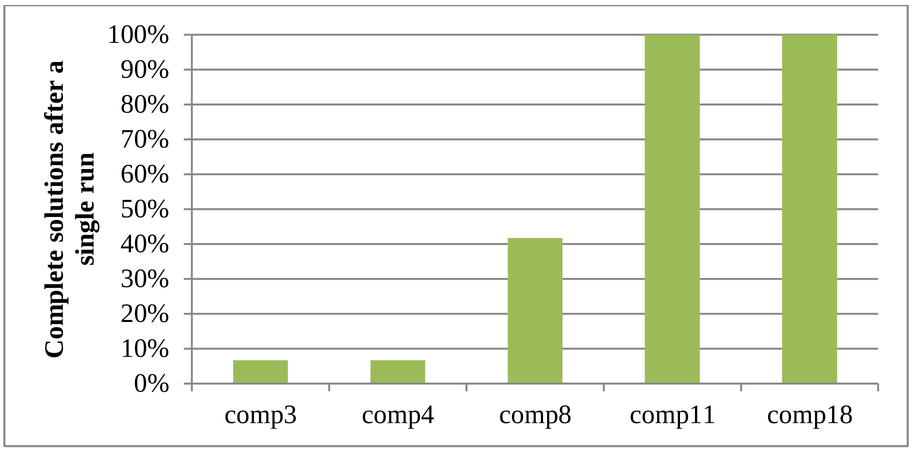


**Figure 22 - Percentage of complete timetables, found after a single loop of the construction heuristic**

Among the remaining 147 runs, where no complete solution was found after the initial construction, the solver did not manage to find a complete solution in only two runs – in both cases Simulated Annealing was used during the feasibility phase. The data from the experiment paints a clear picture, which of the two metaheuristics is better suited to achieve feasibility: the Tabu Search algorithm needed on average 3.7 seconds to find a complete solution; the Simulated Annealing algorithm on the other hand needed 10.8 seconds on average - even when the two runs, where no complete solution was found, were excluded (see Figure 23; the leftmost column shows the average runtime of SA, when the two incomplete solution were included in the calculations – both ran until the time limit of 60 seconds was reached). For all the runs which resulted in a complete and feasible timetable, Tabu Search took a maximum of 11.9 seconds to find a solution, while Simulated Annealing took a maximum of 51.1 seconds. The evidence suggests that Tabu Search is the superior of the two metaheuristics, when it comes to finding a complete and feasible timetable.

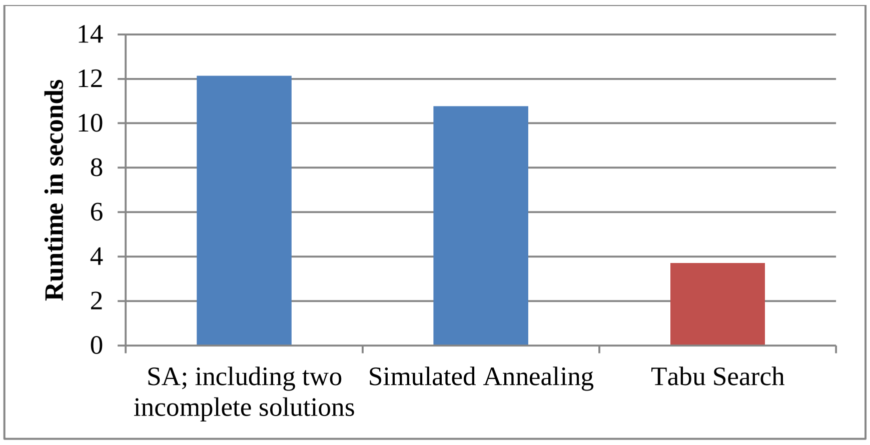


**Figure 23 - Average runtime until a complete solution is found, SA vs. TS**

A possible explanation, why Simulated Annealing does not perform very well in this case, could be that the acceptance of worsening moves is counterproductive; the search space – in this particular formulation of the timetabling problem – has potentially a lot of local optima, which are also global optima (all timetables with a distance to feasibility of zero are global optima, because there is no distance to feasibility lower than zero). It seems to be that - with the problem formulation used in this thesis - the best approach is to find the next local optimum as quickly as possible (using a pure Hill Climbing strategy), because it is likely also a global optimum with a zero distance to feasibility. The strategy of SA to first traverse a large spectrum of the search space before homing in on a local optimum doesn't appear to be well suited with this problem formulation.

### 6.2.2 Achieving Feasibility: Tabu Search vs. the Construction Heuristic

For the second experiment, the goal is to find out whether it is better to only use the construction heuristic to find a complete feasible timetable or if the construction heuristic should be used in conjunction with Tabu Search. For the experiment 35 runs for each of the problem instances 3 and 4 are conducted, while using only the construction heuristic to achieve feasibility (i.e. the construction heuristic is allowed to loop until it has constructed a complete solution). The resulting runtime data is then compared with the respective total runtimes (i.e. a single run of the construction heuristic + the following Tabu Search runtime) from the previous experiment.

Again Tabu Search comes out on top: while TS always found a complete solution in each of its 59 tested runs, the construction heuristic by itself did not find a complete solution in 4 of its 70 tested runs (6% of runs) within the time limit of 90 seconds. The average runtime further strengthens the above assumption: while the construction heuristic in conjunction with TS took on average 29.0 seconds (25.3 seconds for the initial construction and 3.7 for the subsequent Tabu Search) until a solution was found, using only the construction heuristic took 46.1 seconds on average (see Figure 24). The maximum time until a solution was found was 42.3 seconds with Tabu Search and 78.2 seconds using only the construction heuristic (excluding the 4 runs where no complete solution was found).

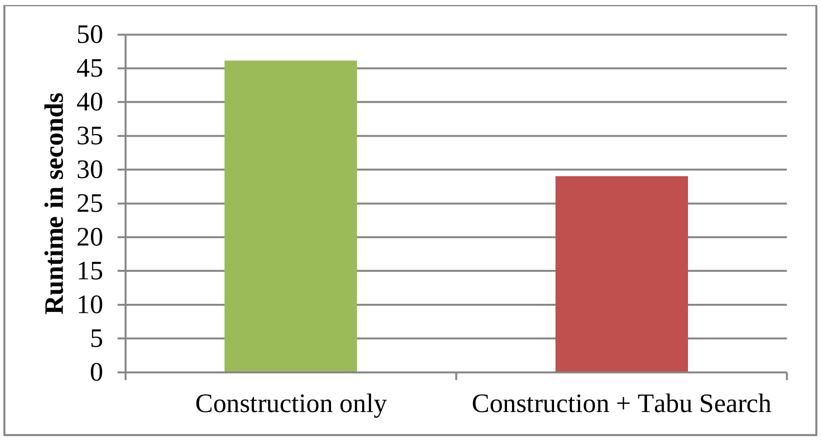


**Figure 24 - Average runtime until a complete solution is found, TS vs. construction heuristic**

Based on these findings - throughout the remainder of the experiments - Tabu Search will be used in conjunction with the construction heuristic during the feasibility phase.

### 6.2.3 Improving the Solution: Simulated Annealing vs. Tabu Search

With the next experiment, the aim is to determine which of the metaheuristics is better suited to reduce the number of soft constrain violations. In this experiment the problem instances 1, 11 and 18 are used, because they usually need the lowest construction time among all of the competition instances. Each instance is re-constructed 30 times, and each so formed initial timetable is then improved by both of the two metaheuristics; in total 180 test runs are thus performed, 90 for each metaheuristic. The maximum allowed runtime for the improvement phase is set to 60 seconds, the maximum tabu length is set to 200 and the maximum temperature is set to 5.

The average penalty scores for each metaheuristic are given in Figure 25. Tabu Search is - on average - able to find significantly lower penalty scores than Simulated Annealing for all of the tested problem instances. The average penalty score across all three problem instances is 275.7 for SA, compared to 213.4 for TS. Particularly for instance 11, the average penalty achieved by TS is only about half the average penalty achieved by SA. With instance 18 the difference between SA and TS is a lot smaller, but TS still beats SA. The best (i.e. lowest) penalty scores for each problem instance are always found by Tabu Search, beating the best scores found by SA by a wide margin.

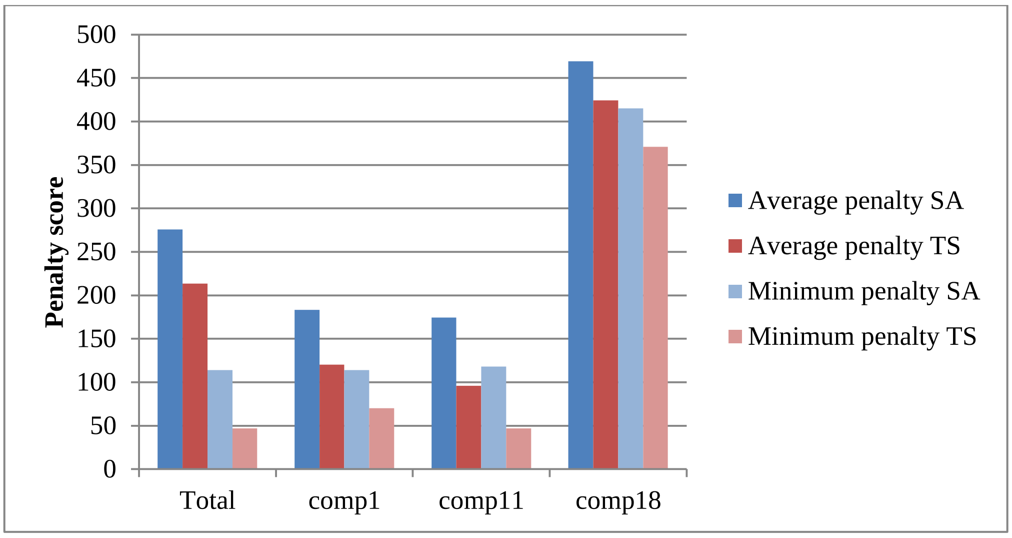


**Figure 25 – Average and minimum penalty scores, SA vs. TS**

From this experiment, it can be concluded that Tabu Search is clearly better than Simulated Annealing at reducing the soft constraint penalty score.

### 6.2.4 Changing the Neighborhood Structure

With the next experiment, the impact of the neighborhood structure on the performance of the solver is studied. The setup for the feasibility phase uses problem instances 3 and 4 for testing. For each of the problem instances, 30 initial timetables are constructed; each initial timetable is then completed by a Tabu Search in three different ways: using only timeslot swaps, using only event swaps or using a random mix of both swaps. To compare the results, the average runtime until the search algorithm finds a complete solution is measured. The maximum length of the tabu list is set to 200 for this experiment.

The setup for the improvement phase features problem instances 1, 11 and 18. For each of the instances, 20 complete timetables are constructed and Tabu Search is then utilized to minimize the penalty score, with the following neighborhood structures: using only timeslot swaps, using only event swaps, using only room swaps; furthermore the combinations timeslot/event swap, room/event swap and timeslot/room swap are tested.

The findings of the first part of the experiment are summarized in Figure 26. Apparently the usefulness of a neighborhood is strongly dependent on the problem instance: swapping only timeslots resulted in the fastest average search time for instance 3 and for the slowest average search time for instance 4; on the contrary swapping only events was fastest with instance 4 and slowest with instance 3. Using both neighborhoods (and choosing one of them at each loop of the TS algorithm at random) gave the overall lowest runtimes, while the average runtimes for each instance were only slightly slower than the fastest runtime of the respective instance. The timeslot-swap-only neighborhood was the only one that failed to produce a complete feasible timetable within the time limit on three occasions, all of them for instance 4. The maximum runtimes until a complete solution was found were 36.5 seconds for the timeslot swap, 44.8 seconds for the event swap and 22.5 seconds for the combined neighborhood.

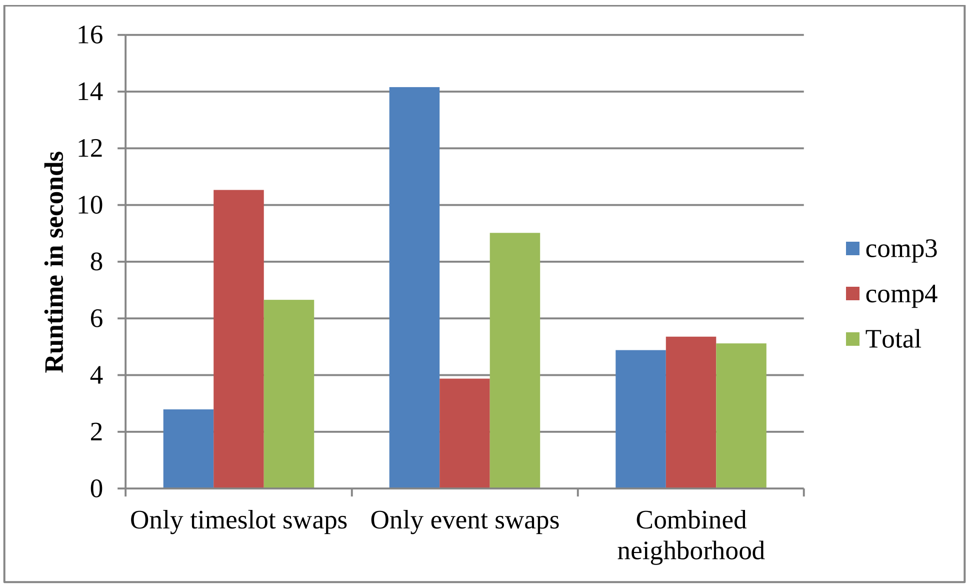


**Figure 26 - Average runtime until a complete solution is found, for different neighborhood structures**

The results suggest that the "fastest" neighborhood structure depends strongly on the problem instance. However, using the combined neighborhood, it is possible to find a complete solution almost as fast as with the "fastest" neighborhood, independent of the problem instance. Thus the combined neighborhood is arguably superior to the other two neighborhoods (for the average case) and is consequently used for the remaining experiments.

Figure 27 summarizes the result of the second part of the experiment. Again, the usefulness of a neighborhood seems to partially depend on the problem instance, especially in case of instance 18 – here the lowest average penalty was found by the timeslot/event neighborhood, which only produced mediocre results for the other instances. Overall the best performance is achieved - as expected - when using all three neighborhoods, followed closely by the timeslot/room swapping neighborhood. Surprisingly good was the performance of the event-swap-only neighborhood, which is on par with the combined room/event swap neighborhood. In general, using a neighborhood which consists of more than one move leads to better results, except for instance 18, where the opposite appears to be the case: here the timeslot-swap-only and event-swap-only neighborhoods are among the best three, outperformed only by the combination of both of them. The overall worst neighborhood is room-swap-only.

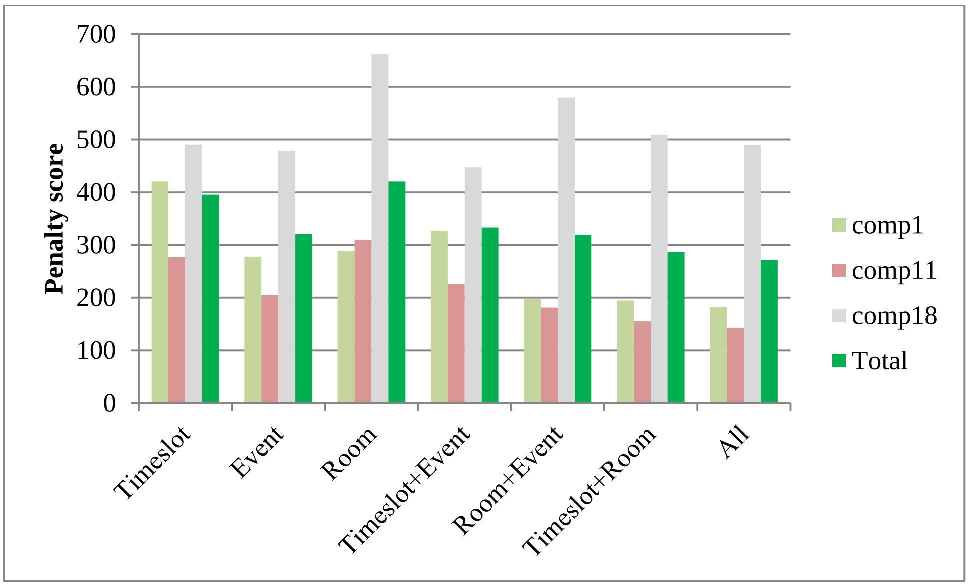


**Figure 27 - Average penalty scores, for different neighborhood structures**

### 6.2.5 Changing the Runtime

The goal of this experiment is to determine how the runtime of the solver algorithm affects the solution quality. For this experiment problem instances 2, 5, 13 and 21 are solved, with a time limit for the improvement phase of 155 seconds. At each elapsed second the current penalty score is recorded. A plot of the scores is given in Figure 28.

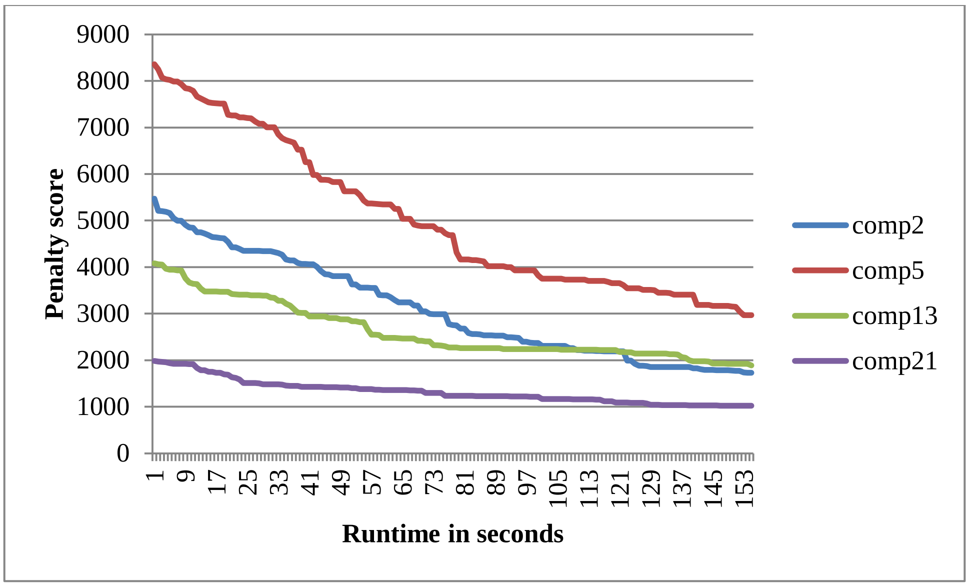


**Figure 28 –Penalty score, depending on the elapsed runtime**

All of the tested instances exhibit similar properties regarding the development of the penalty scores over time: during the beginning of the improvement phase the penalty score drops fast; the longer the algorithm runs, the slower the penalty score drops. Given enough time, the scores are expected to reach a local optimum, where no further improvement is possible. However, the above plot suggests that the optimum has not been reached after 155 seconds of runtime for any of the tested instances.

## 6.3 Assessment of Performance and Solution Quality

In this final experiment the solutions produced by the solver are compared to the best solutions (published at the official ITC-2 website[9]) produced by the winner of the ITC-2 course timetabling track (the winning algorithm is described in (Müller, 2009)). The setup for this experiment allows the solver to run for 422 seconds (the maximum allowed computing time on the author's machine, see the beginning of chapter 6) per solution attempt; each attempt is repeated 10 times per problem instance, resulting in a total of 210 runs. In the experiment Tabu Search is utilized for both finding feasibility and improving the penalty score. The results of the experiment for each problem instance are shown in Figure 29:

[9] http://www.cs.qub.ac.uk/itc2007/winner/finalorder.htm

| | Average construction time | Average Tabu Search time | Average time until feasibility | Average distance after construction | Average penalty score | Minimum distance after construction | Minimum penalty score | Minimum penalty score by Müller |
|---|---|---|---|---|---|---|---|---|
| **comp01** | 1.9 | 0.0 | 1.9 | 0 | 38.7 | 0 | 10 | 5 |
| **comp02** | 43.5 | 36.0 | 79.4 | 11.5 | 1072.8 | 6 | 725 | 51 |
| **comp03** | 27.3 | 5.1 | 32.4 | 7 | 623.7 | 4 | 432 | 84 |
| **comp04** | 40.4 | 4.3 | 44.8 | 3.2 | 578.2 | 0 | 384 | 37 |
| **comp05** | 23.0 | 34.0 | 57.0 | 5.9 | 1791.3 | 4 | 1278 | 330 |
| **comp06** | 64.7 | 0.9 | 65.7 | 1.4 | 1057.9 | 0 | 890 | 48 |
| **comp07** | 89.1 | 11.0 | 100.1 | 7.3 | 1795.2 | 4 | 1482 | 20 |
| **comp08** | 46.8 | 1.4 | 48.2 | 1.9 | 583.6 | 0 | 471 | 41 |
| **comp09** | 37.7 | 1.5 | 39.2 | 3.7 | 596 | 0 | 517 | 109 |
| **comp10** | 70.1 | 6.1 | 76.2 | 5.9 | 1168.4 | 2 | 747 | 16 |
| **comp11** | 4.0 | 0.0 | 4.0 | 0 | 19.2 | 0 | 8 | 0 |
| **comp12** | 64.0 | 12.0 | 76.1 | 4.8 | 1779.7 | 2 | 1516 | 333 |
| **comp13** | 46.5 | 4.1 | 50.6 | 4.5 | 782.6 | 2 | 579 | 66 |
| **comp14** | 42.9 | 0.1 | 42.9 | 0.2 | 546.3 | 0 | 442 | 59 |
| **comp15** | 21.2 | 5.6 | 26.8 | 7.6 | 516.8 | 4 | 404 | 84 |
| **comp16** | 59.3 | 2.0 | 61.3 | 2 | 978.4 | 0 | 718 | 34 |
| **comp17** | 45.7 | 1.2 | 47.0 | 3.2 | 792.1 | 0 | 688 | 83 |
| **comp18** | 13.9 | 0.0 | 13.9 | 0 | 261.2 | 0 | 229 | 83 |
| **comp19** | 32.9 | 11.2 | 44.1 | 7.4 | 518.3 | 3 | 402 | 62 |
| **comp20** | 68.2 | 8.9 | 77.1 | 5.8 | 2023 | 2 | 1273 | 27 |
| **comp21** | 42.6 | 14.3 | 56.9 | 11 | 890 | 9 | 726 | 103 |

**Figure 29 - Solution quality after 422 seconds of runtime**

The solver was able to produce a complete and feasibly timetable (i.e. with a distance to feasibility of zero) within the allowed runtime in 100% of the runs; for problem instances 1, 11 and 18 the construction heuristic always produced a feasible solution, for the other instances feasibility was achieved later in the following phase by a Tabu Search algorithm (the column “Minimum distance after construction” shows that the construction heuristic sometimes produced a feasible timetable for other instances as well).

The first column – “Average construction time” – gives the time measurement in seconds for the construction heuristic. The values also give a first indication about the complexity of the problem: the construction times range from 1.9 seconds for instance 1 to 89.1 seconds for instance 7.

“Average Tabu Search time” gives the time measurements for the feasibility phase of the solver; an average time of 0.0 indicates that the solver always found a complete solution during the construction phase. The lowest non-zero time measurement is 0.1 seconds for instance 14 (the construction did not produce a complete solution in 1 out of the 10 runs for this instance), the highest average time is 36.0 seconds for instance 2.

The column “Average time until feasibility” sums the previous two columns up (i.e. “Time until feasibility” = “Construction time” + “Tabu Search time”). According to the data, the hardest problem instance is instance 7, for which the solver needed on average 100.1 seconds just to find a feasible solution; the easiest problem is instance 1, with an average time until feasibility of 1.9 seconds.

The columns “Average distance after construction” and “Minimum distance after construction” give the average/minimum distance to feasibility, after only the construction phase has finished.

“Average penalty score” and “Minimum penalty score” show the average/minimum penalty score for the respective problem instance, which has been found by the solver after 422 seconds runtime (the runtime includes the time until feasibility, e.g. when the solver took 100 seconds to find a complete and feasible solution, that leaves 300 seconds for the improvement phase). The final column “Minimum penalty score by Müller” gives the minimum penalty scores for each problem instance, which were found by the winning algorithm (described in (Müller, 2009)) of track 3 of the ITC-2.

When comparing the best solutions found by the author’s solver with the best solutions found by Müller’s solver program, Müller’s solutions often have much lower penalty scores. This can partially be explained by Python being a comparatively “slow” programming language and consequently the author’s solver running slower than Müller’s solver (which is programmed in Java). Furthermore the author’s program code was written for effectiveness, not mainly for efficiency. Given more runtime - to compensate for the Python programming language’s- and the program code’s inefficiencies – the author’s solver would arguably find better solutions than the ones presented above (see chapter 6.2.5).

To test the hypothesis that the author’s solver will be able to find better solutions given more runtime, an additional test is conducted. For this test, the solver is allowed to run for 4000 seconds (about 10 times as long as in the previous experiment); because of the limited computational resources the test is conducted only once and only for instance 7 (the instance with the highest discrepancy in the penalty score between the author's and Müller's solutions). A plot of the solution quality vs. the runtime is given in Figure 30.

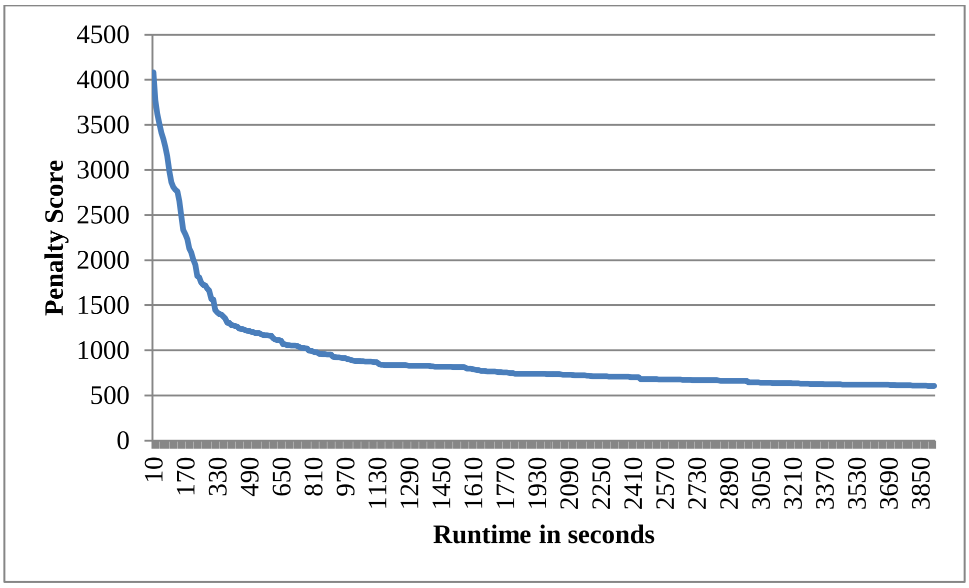


**Figure 30 – Penalty score of instance comp07, depending on the elapsed runtime**

The results of the test show that the solver was indeed able to significantly reduce the penalty score for instance 7, from 1482 (with 422 seconds runtime) down to 608 (with 4000 seconds runtime) – less than half the previous best solution found by the author and a lot closer to Müller's best solution. The slope of the above graph also indicates that there is still room for further improvements: although becoming increasingly flatter, the slope of the graph is still pointing slightly downward at the time limit of 4000 seconds. The slope suggests that the solver could find an even better solution, given additional runtime.

# 7. Summary and Future Research

An automated course timetabling solver has been created using the Python programming language. The solver takes a two-stage approach, where it first builds an initial complete and feasible timetable and then – in the second stage – improves the timetable without breaking its feasibility.

During the first stage, the solver is able to produce complete and feasible timetables for 21 real-life problem instances, which were introduced at the second International Timetabling Competition in 2007, within a reasonably short time frame (even for the hardest problem instance a feasible solution is found in about 100 seconds). Furthermore the solver – given enough time - can find a timetable with a low penalty score during the second stage.

Arguably the results of the second stage are not on par with those results produced by the winners of the competition, i.e. the penalty scores of the solutions produced by the winning algorithm are – in some cases – much lower than those of the solutions produced by the author’s solver. However, adjusting for the inefficiencies of both the Python programming language and the solver code itself (e.g. by allowing a longer runtime), the solver’s results can be improved dramatically. Ultimately, a long runtime of the timetabling solver is not extremely important in practice, because the course timetables are usually created well in advance (i.e. the timetabling is usually done days or even weeks before the start of a new semester); a runtime of a few hours for the program to finish would generally not be considered a problem.

The computational experiments have shown that the Tabu Search metaheuristic yields superior results to the Simulated Annealing metaheuristic during both the feasibility and improvement phases (at least for the given implementation). Tabu Search has also been shown to be better at achieving feasibility than a simple repeated random assignment by the construction heuristic.

Furthermore it has been shown that the neighborhood structure has a strong influence on the solution quality – some neighborhoods work particularly well for certain problem instances – and that a combined neighborhood consisting of all possible moves always yields good results.

A project for future research would be the repetition of some of the experiments. Because of a lack of computing power some of the experiments had to be limited to certain problem instances and a relatively low amount of repetitions. For some experiments, where the results were ambiguous (this was especially the case for the parameter setting), a setup including more instances and more repetitions might produce a clearer outcome.

Another future research project would be to rewrite the program code in certain parts and remove the inherent computational inefficiencies, eventually speeding up the timetabling process and make the solver more competitive with other course timetabling solvers presented in the literature.

So far the course timetabling solver has only been tested with the 21 Udine University problem instances of the ITC-2, as well as with 5 additional test instances (which were only used during the development). Further experimentation is required to determine how well the solver is able to handle other problem instances; e.g. the publicly available Purdue University or Erlangen University timetabling datasets.

# Appendix

## Description oft the Udine University dataset

| Instance | aka | C | L | R | PpD | D | Cu | MML | Co | TA | CL | RO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **comp01** | Fis0506-1 | 30 | 160 | 6 | 6 | 5 | 14 | 2-5 | 13.2 | 93.1 | 3.24 | 88.9 |
| **comp02** | Ing0203-2 | 82 | 283 | 16 | 5 | 5 | 70 | 2-4 | 7.97 | 76.9 | 2.62 | 70.8 |
| **comp03** | Ing0304-1 | 72 | 251 | 16 | 5 | 5 | 68 | 2-4 | 8.17 | 78.4 | 2.36 | 62.8 |
| **comp04** | Ing0405-3 | 79 | 286 | 18 | 5 | 5 | 57 | 2-4 | 5.42 | 81.9 | 2.05 | 63.6 |
| **comp05** | Let0405-1 | 54 | 152 | 9 | 6 | 6 | 139 | 2-4 | 21.7 | 59.6 | 1.8 | 46.9 |
| **comp06** | Ing0506-1 | 108 | 361 | 18 | 5 | 5 | 70 | 2-4 | 5.24 | 78.3 | 2.42 | 80.2 |
| **comp07** | Ing0607-2 | 131 | 434 | 20 | 5 | 5 | 77 | 2-4 | 4.48 | 80.8 | 2.51 | 86.8 |
| **comp08** | Ing0607-3 | 86 | 324 | 18 | 5 | 5 | 61 | 2-4 | 4.52 | 81.7 | 2 | 72 |
| **comp09** | Ing0304-3 | 76 | 279 | 18 | 5 | 5 | 75 | 2-4 | 6.64 | 81 | 2.11 | 62 |
| **comp10** | Ing0405-2 | 115 | 370 | 18 | 5 | 5 | 67 | 2-4 | 5.3 | 77.4 | 2.54 | 82.2 |
| **comp11** | Fis0506-2 | 30 | 162 | 5 | 9 | 5 | 13 | 2-6 | 13.8 | 94.2 | 3.94 | 72 |
| **comp12** | Let0506-2 | 88 | 218 | 11 | 6 | 6 | 150 | 2-4 | 13.9 | 57 | 1.74 | 55.1 |
| **comp13** | Ing0506-3 | 82 | 308 | 19 | 5 | 5 | 66 | 2-3 | 5.16 | 79.6 | 2.01 | 64.8 |
| **comp14** | Ing0708-1 | 85 | 275 | 17 | 5 | 5 | 60 | 2-4 | 6.87 | 75 | 2.34 | 64.7 |
| **comp15** | Ing0203-1 | 72 | 251 | 16 | 5 | 5 | 68 | 2-4 | 8.17 | 78.4 | 2.36 | 62.8 |
| **comp16** | Ing0607-1 | 108 | 366 | 20 | 5 | 5 | 71 | 2-4 | 5.12 | 81.5 | 2.39 | 73.2 |
| **comp17** | Ing0405-1 | 99 | 339 | 17 | 5 | 5 | 70 | 2-4 | 5.49 | 79.2 | 2.33 | 79.8 |
| **comp18** | Let0304-1 | 47 | 138 | 9 | 6 | 6 | 52 | 2-3 | 13.3 | 64.6 | 1.53 | 42.6 |
| **comp19** | Ing0203-3 | 74 | 277 | 16 | 5 | 5 | 66 | 2-4 | 7.45 | 76.4 | 2.42 | 69.2 |
| **comp20** | Ing0506-2 | 121 | 390 | 19 | 5 | 5 | 78 | 2-4 | 5.06 | 78.7 | 2.5 | 82.1 |
| **comp21** | Ing0304-2 | 94 | 327 | 18 | 5 | 5 | 78 | 2-4 | 6.09 | 82.4 | 2.25 | 72.7 |
| **test1** | - | 46 | 207 | 12 | 4 | 5 | 26 | 2-4 | 5.25 | 97.6 | 1.97 | 86.2 |
| **test2** | - | 52 | 223 | 12 | 4 | 5 | 30 | 2-4 | 5.57 | 86.1 | 2.11 | 92.9 |
| **test3** | - | 56 | 252 | 13 | 4 | 5 | 55 | 2-4 | 5.89 | 78.1 | 2 | 96.9 |
| **test4** | - | 55 | 250 | 10 | 5 | 5 | 55 | 2-4 | 5.98 | 76.8 | 2 | 100 |
| **toy** | - | 4 | 16 | 3 | 4 | 5 | 2 | 2-3 | 75 | 90 | 2.1 | 26.7 |

**Figure 31 - Description of the Udine University CB-CTT problem instances, Source: (Bonutti, et al., 2012)**

The instances named “compXX” are part of the original ITC-2 dataset (their alternative names are given in the column “aka”); “testX” and “toy” are intended for testing purposes. The features presented in Figure 31 are:

- **Courses (C):** the amount of courses in the problem instance; a higher number means a more difficult problem.
- **Total lectures (L):** the amount of lectures in the problem instance; a higher number means a more difficult problem.
- **Rooms (R):** the number of available rooms for the timetable; a higher number means an easier problem.
- **Periods per day (PpD):** the number of available timeslots per day; a higher number means an easier problem.
- **Days (D):** the number of days, which make up one work week; a higher number means an easier problem.
- **Curricula (Cu):** the amount of different curricula in problem instance; a higher number means a more difficult problem.
- **Min and max lectures per day per curriculum (MML):** the minimum (maximum) number of lectures of the same curriculum, which should be scheduled per day; a bigger range means an easier problem.
- **Average number of conflicts (Co):** the number of pairs of lectures, which cannot be assigned to the same timeslot, because they have a common teacher or are part of the same course or curriculum; a higher number means a more difficult problem.
- **Average teacher availability (TA) in percent:** the percentage of timeslots during which a teacher is available on average; a higher number means an easier problem.
- **Average number of lectures per curriculum per day (CL):** the amount of lectures, which are part of the average curriculum on the average day; a higher number means a more difficult problem.
- **Average room occupation (RO) in percent:** the percentage of time during which any room is occupied on average (100% means all rooms are occupied all the time); a higher number means an easier problem.

## Example for an input file

```
 Name:                       Toy
Courses:                      4
Rooms:                        3
Days:                         5
Periods_per_day:              4
Curricula:                    2
Min_Max_Daily_Lectures:       2         3
UnavailabilityConstraints:    8
RoomConstraints:              3

COURSES:
SceCosC                     Ocra        3         3         30       1
ArcTec                      Indaco      3         2         42       0
TecCos                      Rosa        5         4         40       1
Geotec                      Scarlatti   5         4         18       1

ROOMS:
rA                           32         1
rB                           50         0
rC                           40         0

CURRICULA:
Cur1                          3      SceCosC    ArcTec    TecCos
Cur2                          2      TecCos     Geotec

UNAVAILABILITY_CONSTRAINTS:
TecCos                        2         0
TecCos                        2         1
TecCos                        3         2
TecCos                        3         3
ArcTec                        4         0
ArcTec                        4         1
ArcTec                        4         2
ArcTec                        4         3

ROOM_CONSTRAINTS:
SceCosC                      rA
Geotec                       rB
TecCos                       rC

END.
```

**Figure 32 – The input file of the “Toy” problem instance**

The format for the different sections of the input file is as follows (each entry <…> represents a column in the above file):

**Courses: <CourseID> <Teacher> <# Lectures> <MinWorkingDays> <# Students> <Double Lectures>**

E.g. in the above example input file, the first entry in the section “COURSES” signifies that the course SceCosC (CourseID, column 1) is taught by Mr. or Mrs. Ocra (Teacher, column 2), consists of three lectures (# Lectures, column 3), should be spread over at least three days (MinWorkingDays, column 4), has 30 students (# Studnets, column 5) and has one double lecture (Double Lectures, column 6).

**Rooms: <RoomID> <Capacity> <Site>**

**Curricula: <CurriculumID> <# Courses> <MemberID> ... <MemberID>**

E.g. in the example input file, the first entry in the section “CURRICULA” signifies that the curriculum Cur1 (CurriculumID, column 1) consists of the three courses (# Courses, column 2): SceCosC (MemberID, column 3), ArcTec (MemberID, column 4) and TecCos (MemberID, column 5).

**Unavailability_Constraints: <CourseID> <Day> <Day_Period>**

**Room_Constraints: <CourseID> <RoomID>**

**Example for an output file**

```
ArcTec     rB     0     3
ArcTec     rB     3     3
TecCos     rB     0     2
TecCos     rB     4     0
Geotec     rB     1     1
Geotec     rA     0     1
ArcTec     rB     3     1
TecCos     rC     2     2
SceCosC    rA     1     1
TecCos     rC     1     3
SceCosC    rC     2     1
Geotec     rB     3     0
TecCos     rC     2     3
Geotec     rB     4     2
SceCosC    rB     2     0
Geotec     rC     0     3
```

**Figure 33 - A possible output file for the "Toy" problem instance**

The output file is of the format **<CourseID> <RoomID> <Day> <Day_Period>**, e.g. the last line in the above example (Geotec rC 0 3) states that a lecture of the course “Geotec” is assigned to room C on the fourth period of the first day (the day and period indices starts with 0 - period 0 is the first period, period 1 is the second period etc.).

Further details concerning the input and output files are explained in the official documentation for ITC-2 track 3 by (Di Gaspero, et al., 2007).

# Bibliography

Abdullah, S., Turabieh, H., McCollum, B. & McMullan, P., 2012. A hybrid metaheuristic approach to the university course timetabling problem. *Journal of Heuristics,* 18(1), pp. 1-23.

Bonutti, A., De Cesco, F., Di Gaspero, L. & Schaerf, A., 2012. Benchmarking curriculum-based course timetabling: formulations, data formats, instances, validation, visualization, and results. *Annals of Operations Research,* 194(1), pp. 59-70.

Brownlee, J., 2011. *Clever Algorithms: Nature-Inspired Programming Recipes.* s.l.:LULU Press.

Burke, E., Jackson, K., Kingston, J. & Weare, R., 1997. Automated university timetabling: The state of the art. *The computer journal,* 40(9), pp. 565-571.

Burke, E., Kendal, G., McCollum, B. & McMullan, P., 2007. *Constructive versus improvement heuristics: an investigation of examination timetabling.* s.l., s.n., pp. 28-31.

Burke, E. K., Marecek, J., Parkes, A. J. & Rudová, H., 2007. On a Clique-Based Integer Programming Formulation of Vertex Colouring with Applications in Course Timetabling. *CoRR,* Volume abs/0710.3603.

Burke, E., Mareček, J., Parkes, A. & Rudová, H., 2012. A branch-and-cut procedure for the Udine Course Timetabling problem. *Annals of Operations Research,* 194(1), pp. 71-87.

Burke, E. & Petrovic, S., 2002. Recent research directions in automated timetabling. *European Journal of Operational Research*, 140(2), pp. 266-280.

Chiarandini, M., Birattari, M., Socha, K. & Rossi-Doria, O., 2006. An effective hybrid algorithm for university course timetabling. *Journal of Scheduling,* 9(5), pp. 403-432.

Chiarandini, M. & Stützle, T., 2002. *Experimental evaluation of course timetabling algorithms,* s.l.: Technical Report, FG Intellektik, TU Darmstadt.

Cooper, T. B. & Kingston, J. H., 1996. *The complexity of timetable construction problems.* s.l.:Springer.

Corne, D., Ross, P. & Fang, H., 1995. Evolving timetables. In: L. Chambers, ed. *The practical handbook of genetic algorithms.* Florida: s.n., p. 219–276.

de Werra, D., 1985. An introduction to timetabling. *European Journal of Operational Research,* 19(2), pp. 151-162.

Di Gaspero, L., McCollum, B. & Schaerf, A., 2007. *The second international timetabling competition (ITC-2007): Curriculum-based course timetabling (track 3).* Rome, Italy, s.n.

Di Gaspero, L. & Schaerf, A., 2003. Multi-neighbourhood local search with application to course timetabling. In: *Practice and Theory of Automated Timetabling IV.* s.l.:Springer, pp. 262-275.

Geiger, M. J., 2008. An application of the threshold accepting metaheuristic for curriculum based course timetabling. *arXiv preprint arXiv:0809.0757.*

Gendreau, M., 2010. *Handbook of metaheuristics.* s.l.:Springer.

Halldórsson, M., 1993. A still better performance guarantee for approximate graph coloring. *Information Processing Letters,* 45(1), pp. 19-23.

Lach, G. & Lübbecke, M., 2008. *Curriculum based course timetabling: Optimal solutions to the udine benchmark instances.* s.l., s.n.

Lewis, R., 2006. *Metaheuristics for University Course Timetabling,* s.l.: s.n.

Lewis, R., 2008. A survey of metaheuristic-based techniques for university timetabling problems. *OR Spectrum,* 30(1), pp. 167-190.

Lü, Z. & Hao, J.-K., 2010. Adaptive tabu search for course timetabling. *European Journal of Operational Research,* 200(1), pp. 235-244.

Lü, Z., Hao, J.-K. & Glover, F., 2011. Neighborhood analysis: a case study on curriculum-based course timetabling. *Journal of Heuristics,* 17(2), pp. 97-118.

McCollum, B., 2006. *University timetabling: Bridging the gap between research and practice.* s.l., Springer, pp. 15-35.

Müller, T., 2005. *Constraint-based Timetabling,* KTIML MFF UK: s.n.

Müller, T., 2009. ITC2007 solver description: a hybrid approach. *Annals of Operations Research,* 172(1), pp. 429-446.

Murray, K., Müller, T. & Rudová, H., 2007. Modeling and solution of a complex university course timetabling problem. In: *Practice and Theory of Automated Timetabling VI.* s.l.:Springer, pp. 189-209.

Paechter, B., Rankin, R., Cumming, A. & Fogarty, T. C., 1998. *Timetabling the classes of an entire university with an evolutionary algorithm.* s.l., Springer, pp. 865-874.

Petrovic, S. & Burke, E. K., 2004. University timetabling. *Handbook of scheduling: algorithms, models, and performance analysis,* Volume 45, pp. 1-23.

Rossi-Doria, O. et al., 2003. A comparison of the performance of different metaheuristics on the timetabling problem. In: *Practice and Theory of Automated Timetabling IV.* s.l.:Springer, pp. 329-351.

Rudová, H., Müller, T. & Murray, K., 2011. Complex university course timetabling. *Journal of Scheduling,* 14(2), pp. 187-207.

Schaerf, A., 1999. A survey of automated timetabling. *Artificial intelligence review,* 13(2), pp. 87-127.

Tripathy, A., 1992. Computerised decision aid for timetabling - a case analysis. *Discrete Applied Mathematics,* #mar#, 35(3), pp. 313-323.

Wasfy, A. & Aloul, F., 2007. *Solving the university class scheduling problem using advanced ILP techniques.* s.l., s.n., pp. 1-5.

Wilke, P. & Killer, H., 2010. *Comparison of Algorithms solving School and Course Time Tabling Problems using the Erlangen Advanced Time Tabling System (EATTS).* s.l., s.n., p. 427ff.

# Zusammenfassung in deutscher Sprache

Das Ziel der vorliegenden Arbeit ist die Entwicklung und Evaluierung eines Programms zur automatischen Erstellung von Stundenplänen, unter der Zuhilfenahme von zwei Metaheuristiken. Hierfür wird zunächst, im zweiten Kapitel der Arbeit, das zu lösende Stundenplanproblem vorgestellt: die Erstellung von Stundenplänen für Kurse an Universitäten.

Im dritten Kapitel werden verschieden Benchmark-Datensätze vorgestellt, die in der Stundenplan-Forschung von Relevanz sind; außerdem wird der Datensatz, der bei der zweiten „International Timetabling Competition“ (ITC-2) verwendet wurde und mit dem im weiteren Verlauf der Masterarbeit gearbeitet wird, detailliert vorgestellt.

Kapitel 4 gibt einen kurzen Überblick über die Verwendung von Metaheuristiken im Bereich der Stundenplanerstellung. In diesem Kapitel wird weiterhin - anhand von Untersuchungsergebnissen aus der Literatur – ein Vergleich verschiedener Metaheuristiken hinsichtlich ihrer Eignung für die Stundenplanerstellung vorgenommen. Als Ergebnis werden die beiden Metaheuristiken „Tabu Search“ und „Simulated Annealing“ als am erfolgversprechendsten ermittelt und ihre Funktionsweise kurz vorgestellt.

Das nachfolgende Kapitel 5 liefert eine Beschreibung des erstellten Programms, wobei Pseudocode und einfache graphische Beispiele verwendet werden um die Funktionsweise des Programms zu erläutern. Dabei wird genauer auf die Nachbarschaftsstruktur des Suchraumes der Metaheuristiken eingegangen, die Heuristik zur Erstellung eines vorläufigen Stundenplanes wird beschrieben und die Implementierung der beiden Metaheuristiken für die Suche nach einem gültigen Stundenplan bzw. die Verbesserung eines vorhandenen gültigen Stundenplans wird vorgestellt. Außerdem wird erläutert, wie die fertige Lösung validiert werden kann.

Der experimentelle Teil der Arbeit ist im sechsten Kapitel beschrieben. Zunächst wird versucht anhand mehrerer Tests die optimalen Parameter für die beiden gewählten Metaheuristiken zu bestimmen; letztlich wird die optimale Länge der Tabuliste auf 300 festgelegt; die maximale und minimale Temperatur für den Simulated Annealing Algorithmus wird auf 5, bzw. 1.3 festgelegt, die Anzahl von Zwischenschritten auf 5.

Die Ergebnisse der weiteren Experimente sind die folgenden:

- Tabu Search ist sowohl für die Suche nach einem gültigen Stundenplan als auch für die Verbesserung eines vorhandenen Stundenplans besser geeignet als Simulated Annealing
- Tabu Search findet schneller einen gültigen Stundenplan als ein Algorithmus der rein nach dem Zufallsprinzip arbeitet.
- Die Wahl der Nachbarschaftsstruktur hat einen enormen Einfluss auf die Lösungsqualität, wobei manche Nachbarschaften besonders gut mit einigen Probleminstanzen funktionieren und besonders schlecht mit anderen; allgemein liefert eine kombinierte Nachbarschaft, die alle anderen Nachbarschaften in sich vereint, die besten Ergebnisse.

Das Programm ist in der Lage – in relativ kurzer Zeit - einen gültigen Stundenplan für alle 21 der getesteten Probleminstanzen zu erstellen. Im direkten Vergleich mit dem Sieger der ITC-2 zeigt sich jedoch, dass die Lösungsqualität teilweise deutlich schlechter ist. Dies lässt sich allerdings relativieren, indem man die Ineffizienz des Programmcodes berücksichtig: erhöht man die erlaubte Laufzeit des Programms, so findet dieses auch deutlich bessere Ergebnisse.

Abschließend lässt sich feststellen, dass das Ziel, ein effektives Programm zur automatischen Erstellung von Stundenplänen zu implementieren, erreicht wurde.