Utrecht University

Department of Information and Computing Sciences

Timetabling Lab Sessions at the Koningsberger Building

Master's Thesis

ICA-4015533

Author:
Robin de Jong
Utrecht University
Utrecht, the Netherlands

First Supervisor:
dr. J.A. Hoogeveen

Second Supervisor:
dr. ir. J.M. van den Akker

30 August 2017

Abstract

Since 2015 the *Victor J. Koningsberger building* has been used by multiple departments and faculties of Utrecht University to host their lab sessions. To help with the scheduling of the lab sessions, we created an algorithm to automatically create a timetable based on the requests. In this work we describe how we modeled the problem. We also show our algorithm, which is a version of *Simulated Annealing*, and its components. Elaborate results are shown of our research to find the best settings for our algorithm. Finally, we also take a look at the GUI application that we created to allow course coordinators to fill in their demands and the central coordinator(s) to create a timetable from those demands, using our algorithm.

Our method has already been used to create the timetables for the first two quarters of the 2017–2018 academic year, and it has been decided to use these timetables instead of the hand-made ones. More recently, it has been decided that our GUI application and algorithm will be used for the creation of the timetables for the third and fourth quarter of the 2017–2018 academic year as well.

Contents

1 Introduction

In this work, we will look at the timetabling of laboratory sessions in the *Victor J. Koningsberger building* of Utrecht University. The construction of the Victor J. Koningsberger building was completed in 2015 and the building has been used mostly by the Departments of *Biology*, *Chemistry* and *Pharmaceutical Sciences* since. The *Faculty of Medicine/University Medical Center Utrecht* and the *Faculty of Geosciences* have also been using the building for some of their lab sessions.

The Koningsberger building has 7 floors. The ground, first and second floor are mostly filled with study places for individuals and groups, and computer rooms. The third floor through the sixth floor contain 17 lab rooms and different kinds of auxiliary rooms with devices that are required for some lab sessions. The lab rooms are categorized into 4 different types. Some lab sessions need to happen in one specific type of room, others can be held in multiple room types.

Before the building existed, the Departments of Biology, Chemistry and Pharmaceutical Sciences all had their own lab rooms. Now, they have to share lab rooms, which means they have to 'merge' their timetables and resolve any conflicts. So far, this merging was done by hand. This works, but it does cost time. We will try to automate part of this process. The goal is to make the merging possible even if the building usage gets higher in the future. Another objective is to make the timetables better, with respect to things like closeness of rooms used for the same lab session, and slack in the timetable for setting up and clearing rooms.

Eventually, course coordinators should be able to request lab sessions (in consultation with the department lab coordinator). These requests will include properties like the number of required rooms, the type of the rooms and the preferred day and time the session should be held. Our solution should then take all of the requests and find a good timetable. In reality, the department lab coordinators might then have to make some adjustments, for example if a session that has been moved from its preferred day/time really cannot move there. In general though, the plan is to make lab session planning the first priority, over regular lectures (which are scheduled separately in different buildings/rooms), because lab rooms are far more scarce.

In section 2, literature regarding educational timetabling is reviewed. In section 3 we describe how we modeled our problem. Section 4 describes the approach we take to find a good timetable. Our approach is based on a *Simulated Annealing* algorithm. Then, in section 5, we describe our research to find the best settings for the algorithm and we compare timetables created by our algorithm to hand-made timetables. Section 6 contains a short

description of the GUI application we built such that the end user can easily use our algorithm. Finally, section 7 concludes this work.

2 Related Work

In this section we review literature about educational timetabling, the subject of our research. Educational timetabling is often split into high school timetabling and university timetabling. We will focus on university timetabling here. Traditionally, university timetabling has been divided into three different problems: *examination timetabling*, *post enrollment course timetabling* and *curriculum-based course timetabling*.

The difference between the latter two is the way conflicts between courses are set up. In post enrollment course timetabling, the conflicts come directly from the enrollment of students in courses. If a student attends two courses, lectures of those courses cannot be scheduled at the same time. In curriculum-based course timetabling, the conflicts come from curricula. These are groups of courses which cannot be scheduled at the same time, meaning students can attend courses in the same curriculum without having conflicts in their schedule.

The division into three problems is a result of the *Second International Timetabling Competition* (ITC-2007) [DSM07], which consisted of a contest for each of these three problems. In this literature review we will focus on curriculum-based course timetabling, because it is closest to our problem. However, it is good to mention that curriculum-based and post enrollment course timetabling are closely related. There was even a contestant who was named a finalist in each of the three competition tracks and won two of them ([Mül09]). A review of literature on all parts of educational timetabling was written by Kristiansen and Stidsen [KrS13].

More information about ITC-2007 and a review of the literature on its curriculum-based timetabling problem can be found in section 2.1.1. We start off section 2.1 with a review of some literature that was not directly based on the ITC-2007 problem formulation. In section 2.2 we will look at earlier work on the subject at Utrecht University.

2.1 Curriculum-Based Course Timetabling

In this section we review literature about curriculum-based course timetabling. First, we review two approaches that are not based on the ITC-2007 problem formulation and then, in section 2.1.1, we will look at approaches for the ITC-2007 formulation. Surveys on the subject were written by Kristiansen and Stidsen [KrS13], Bettinelli et al. [BCR15] and Babaei et al. [BKH15].

Integer Linear Programming Schimmelpfeng and Helber [ScH07] use an *Integer Linear Programming* approach to improve the timetables at the School of Economics and Management at the University of Hannover. The timetables are equal for each week during a semester and a week consists of 5 days with 6 predefined timeslots each day. The algorithm can handle multiple teachers per meeting and the most basic hard constraints are there to make sure each meeting is scheduled and no teacher is needed at two meetings at the same time. Other hard constraints force certain meetings to be scheduled in the same room directly after each other and force certain pairs of meetings to be given in a specified order during the week.

Rooms are grouped into room types and there is a soft constraint to make sure no more rooms of every type are used concurrently than are available. If this constraint is not satisfied, certain costs are associated to this that represent renting external rooms. Other soft constraints make sure some courses are not taught at the same time or some meetings are taught at the same time. Preferences of teachers regarding teaching hours are also included in the objective function.

Results are shown of planning an actual semester at the school. According to the authors the administrative staff was surprised by the extent to which the timetable was conflict-free. Finally, the authors describe how they introduced their new approach of timetabling to the teaching staff. At first, the administrative and teaching staff were skeptical, but a questionnaire among the teachers showed that after seeing the result, a large majority thought the system should be kept in use.

Variable Neighborhood Search In their paper, Nguyen et al. [NNT11] describe how they implemented and tested eight variants of *Variable Neighborhood Search* for creating timetables at the Ho Chi Minh City University of Science. For the VNS variants, they used 12 different shaking strategies, all aimed at changing the period assignment of meetings. The authors also introduced two ways of ordering the different shaking strategies and two acceptance criteria. All combinations of VNS variant, strategy ordering and acceptance criterion are tested. The Fleszar-Hindi variant with static ordering and improvement only as acceptance criterion performs best.

2.1.1 ITC-2007

In 2002, the *First International Timetabling Competition* (ITC-2002) was held. In this competition, participants had to design an algorithm to solve a *post enrollment course timetabling* problem. This competition helped to create a universal standard problem formulation, which meant research into this

area was mostly aimed at the same problem. This way, different approaches could easily be compared using a set of test instances.

Because of this success, a new competition was organized, the *Second International Timetabling Competition* (ITC-2007). This competition was divided in three tracks, to accommodate different types of timetabling at universities. The tracks were *examination timetabling*, *post enrollment course timetabling* and *curriculum-based course timetabling*. Like before, we will focus on the curriculum-based timetabling here.

The problem is formulated by Di Gaspero et al. [DSM07]. In the problem, a timetable has to be generated for one week. There are a number of days (typically 5 or 6) and each day is split in a fixed number of timeslots. A combination of a day and a timeslot is called a period. Each course has a set number of lectures to be taught in the week, a set number of students attending and a teacher teaching it. Each room has a capacity, but otherwise, all rooms are equal. There are multiple curricula, which are groups of courses. Lectures of courses in the same curriculum cannot be taught in the same period.

There are four hard constraints that solutions have to comply with to be feasible. Firstly, all lectures of each course must be scheduled, and must be scheduled in different periods. Secondly, no more than one lecture can be scheduled in a room during a period. Thirdly, lectures of courses with the same teacher or in the same curriculum cannot be taught in the same period. And finally, if a teacher is not available during a period, no lecture of their course(s) can be scheduled in that period.

Of course, there are also some soft constraints. Violations of these constraints result in penalty points. The fewer penalty points, the better a solution (as long as it is feasible with respect to the hard constraints). The soft constraints are as follows. Firstly, each lecture must be scheduled in a room with a capacity equal to or greater than the attendance. Furthermore, courses can have a minimum number of days, which means the lectures must be spread over at least that number of days. Additionally, lectures belonging to the same curriculum should be scheduled in adjacent timeslots on a day. Finally, all lectures of a course should be scheduled in the same room.

There are 21 instances, gathered from real data from the University of Udine, provided by the authors. These instances are still used today to benchmark algorithms for the curriculum-based timetabling problem. In the remainder of this section we review some of the approaches that have been taken to tackle the curriculum-based course timetabling problem, as formulated by Di Gaspero et al. [DSM07] for ITC-2007.

Genetic Algorithm In Abdullah et al. [ATM09] a *Genetic Algorithm* with local search is used to tackle the course timetabling problem. The algorithm is divided into two phases: the construction phase, in which an initial population is constructed, and the improvement phase, in which a Genetic Algorithm and local search are used with this population to find a better solution.

In the construction phase, three algorithms are used to create each individual in the population. First a *largest degree heuristic* is used (where the degree of a lecture refers to the number of conflicting lectures), then this timetable is improved using *Neighborhood Search* and *Tabu Search*.

The constructed individuals are then used in a Genetic Algorithm. Offspring is created using mutation and crossover. The authors state that the crossover used is single-point crossover, however they do not explain how the individuals are represented as chromosomes. It seems logical that a chromosome consists of the assigned period/room combination for each lecture. After this crossover, the individuals may need to be repaired, because an infeasible solution is created. After the offspring is created, local search is applied to it and the results of that are then used in the next generation of the Genetic Algorithm.

Hybrid approach One of the best performing approaches in ITC-2007 was introduced by Müller [Mül09]. He used the same approach for all three tracks (examination timetabling, post enrollment based course timetabling and curriculum-based course timetabling). In every track, this approach was named as a finalist and it won the examination and curriculum-based tracks.

The approach uses as many as four different algorithms in succession to get a good result. First, a feasible solution is found using *Iterative Forward Search*. Each iteration, an unassigned variable (i.e. a course's time and room assignment) is assigned a value in its domain. If a hard constraint is violated because of this assignment, other variables (i.e. other courses' time/room assignments) are unassigned to get rid of this violation. Every time such a violation occurs a counter is increased to make sure the algorithm does not repeat bad assignments. This phase ends when all variables are assigned.

From this initial schedule, a *Hill Climbing* algorithm is used to find the local optimum. Each iteration a random neighbor is selected and is applied if it yields an improvement. This phase ends after a set number of iterations without improvement. Then, the *Great Deluge* algorithm is applied. This algorithm uses a slowly decreasing upper bound to limit the allowed decline in solution quality. When this upper bound reaches a certain value, the *Simulated Annealing* phase is started. When Simulated Annealing is finished,

the approach goes back to the Hill Climbing algorithm and repeats the process from there. For the examination timetabling track, Simulated Annealing is never used and Great Deluge is repeatedly run (without returning to Hill Climbing). The algorithm is stopped when the maximum runtime is reached.

The used neighborhoods differ between the tracks. For the curriculum-based track, the neighborhoods consist of moving a lecture in time, room or both, moving all lectures of a course to the same room, spreading course lectures over more days and moving lectures to times adjacent to other lectures in the curriculum.

Adaptive Tabu Search In their work, Lü and Hao [LuH10] present an algorithm called *Adaptive Tabu Search*. Their approach consists of three phases. The first phase creates an initial feasible solution, using a greedy heuristic. First a course is selected based on a small number of available periods and a large number of unscheduled lectures. The lecture is then assigned to the period which yields the smallest number of conflicts with other unscheduled lectures. This is repeated until all lectures are scheduled.

The second and third phase are a combination of *Tabu Search* and *Iterated Local Search*. In the second phase a Tabu Search algorithm is used to improve the current solution. As neighborhood they use a token-ring of *SimpleSwap* (swapping the periods and rooms of two lectures) and *KempeSwap* (swapping multiple conflicting lectures from two periods) neighborhood operators. The third phase is based on Iterated Local Search. The current solution is perturbed to get out of a local optimum. These two phases are applied alternatingly until time runs out or a fixed number of iterations is reached.

During the iterations, the depth of the Tabu Search (how many iterations will be done without improvement before stopping the phase) and the perturbation strength are gradually increased as long as no new best solution is found. This way the algorithm does more and more intensification and more diversification in the hope that eventually a better solution is found. Experimental results show that the algorithm can compete with the finalists of ITC-2007, improving some of their scores, while getting close on all other instances.

Neighborhood analyses Lü et al. [LHG11] analyze the quality of multiple neighborhood operators in local search. To do this, they test several neighborhood operators on the curriculum-based course timetabling problem. They look at three known and one newly introduced operators. At first they just test the results of the local search using each of these operators

separately and report the quality of the timetables.

Then, they also look at combinations of the operators. Operators are combined in two ways: as a union, where the neighborhoods are just merged and the best neighbor is selected from this union, and as a token-ring. In token-ring search, the local optimum is first found using one neighborhood. Then the algorithm switches to the other neighborhood and finds the local optimum for that. This process is repeated until no improvement is possible anymore using either neighborhood.

The performance of the operators is analyzed with three measures: percentage of improving neighbors, improvement strength, number of improving iterations. Initially, the authors test using only a *Steepest Descent* algorithm, but in the end they also test the operators using *Tabu Search*, *Iterated Local Search* and *Adaptive Tabu Search*. With all these algorithms it turns out that the token-ring combination of the *SimpleMove* and their newly introduced *KempeSwap* operators performs best.

Artificial Bee Colony Bolaji et al. [BKA11] attempted to use an *Artificial Bee Colony* algorithm for the problem. After initialization, a set of initial solutions is generated. This is done randomly, using the saturation degree of the lectures.

Then, repeatedly, the 'employed bees' each randomly select a solution, based on its fitness. Each employed bee now performs local search on the selected solution. If the resulting solution is better than the original, the original is replaced in the population. Then, 'onlooker bees' each randomly select (using tournament selection) a solution to improve. They improve the solution using local search, but stop after a number of iterations without improvement. At this point the associated employed bee turns into a 'scout bee'. These scout bees are then used to generate new, random solutions which are added to the population. After a predefined number of iterations of repeating these three steps, the algorithm stops. The best found solution is remembered throughout the algorithm. This algorithm is designed to spend more time improving promising solutions and discard solutions when it seems they cannot be improved anymore.

It turns out the results of this method cannot keep up with the methods that reached the final of ITC-2007. The authors, however, do see promise in the results and have tried to improve the approach. We found two further articles by the authors, but the results are still not near those of the ITC-2007 finalists.

Ant Colony Optimization Thepphakorn et al. [TPH14] used an *Ant Colony Optimization* approach to solve curriculum-based timetabling. Specifically they tried the *Best-Worst Ant System* and the *Best-Worst Ant Colony System* algorithms.

An ant tour is created by going through all lectures that need to be planned, sequentially. The lectures are processed in descending order of number of students. For each lecture, one of the possible period/room combinations is selected randomly, with the probability for each combination depending on the pheromones (left behind by ants in previous iterations) and heuristic information.

The authors added to this process the use of local search. After a tour is created local search is applied. Two local search types are introduced. The first one attempts to make sure rooms are large enough and lectures of the same course are held in the same room. It does this by interchanging rooms for pairs of lectures in the same period. The second local search type attempts to spread courses over more days (if necessary) and to make lectures in the same curriculum be held in adjacent periods. Some lectures that violate these soft constraints are selected and unscheduled. Then these lectures are assigned to feasible period/room combinations that give the best local fitness.

The results of this method were compared to other Ant Colony Optimization approaches and proved to be better. The authors did not compare the results to those of other techniques.

Simulated Annealing Bellio et al. [BCD13] introduced a simple *Simulated Annealing* based approach to optimize curriculum-based course timetabling instances. They use a rather classical version of Simulated Annealing, however they do not always decrease the temperature after a fixed number of iterations, but also after a set number of successful neighborhood operations. That way more time can be spent in the later stages of the algorithm. The algorithm is stopped after a fixed number of iterations (which more or less corresponds to a fixed running time). The neighborhood operators used are *MoveLecture* and *SwapLectures*. A description of these operators can be found later in this paragraph.

Multiple parameter sets are generated using the *Hammersley point set* [WLH97]. These parameter sets are then compared using a set of generated instances, designed to resemble the ITC-2007 instances. The best parameter set is then used for the ITC-2007 and even though the method is rather simple, it comes up with good results.

In [BCD16], the same authors dive deeper into their Simulated Anneal-

ing method. First, they explain their algorithm in more detail. Unlike other methods, three out of four of the hard constraints are relaxed. The neighborhood operators used in this method are *MoveLecture*, which changes the period/room combination of one lecture, and *SwapLectures*, which swaps the period/room combinations of two lectures. The ratio in which these two operators are used is one of the parameters of the algorithm that are tuned.

After explaining the algorithm, the authors show their thorough process of tuning the algorithm parameters. Once again, they have a separate, generated training set, to prevent overfitting on the training data. Twenty different parameter sets were tested, once again generated using the Hammersley point set. Initial ranges for all parameters were determined in preliminary experiments.

The twenty parameter sets were tested on the training set to find which one was the best. In the process some parameters turned out to be either not very important to the result (in which case an arbitrary value in the range was chosen), or a very good value emerged from the results. These parameters were then fixed and three parameters remained. The authors decided to use a *classification forest*, to predict the best parameter set for a problem instance, based on instance characteristics like number of lectures and average number of conflicts.

The method using the classification forest was significantly better than always using the best parameter set (that emerged from the training set) for only a couple of the tested instances. However, it was never significantly worse. Both approaches, using the best parameter set and using the classification forest performed well on the ITC-2007 instances.

Graph Heuristics Wahid and Hussin [WaH16] focus on generating good initial populations for the problem, using (combinations of) graph heuristics. These populations are necessary in, among others, *Genetic Algorithms*. The graph heuristics they use are *largest degree*, which looks at the number of lectures each lecture conflicts with, *weighted degree*, which is like *largest degree*, but incorporates the number of students involved in the conflicts, and *saturation degree*, which takes into account the number of free valid period/room combinations for each lecture. The authors also combine these heuristics. It is not entirely clear how they do that, but it seems one heuristic is used normally and the other is used as a tiebreaker.

The heuristic values are used to order the lectures. Lectures are then each planned in a random valid slot in the heuristic's order. If a lecture cannot be planned it is revisited at the end and some undescribed methods are used to fix this. The different (combinations of) heuristics are evaluated

by attempting to generate 50 solutions for several test instances. The authors compare how many of the found solutions are feasible for each heuristic and test instance.

2.2 Previous Research at Utrecht University

Research was done into scheduling courses at Utrecht University before. Kampman [Kam13] focused on planning all meetings at the university at once. He tried to come up with a timetabling algorithm that works better than the *Syllabus Plus* software that is still used by the university today and which appears to be using a greedy algorithm for scheduling.

The method first splits up the set of meetings to be planned into two sets: the regular meetings and the incidental meetings. The algorithm uses a two-phase approach, first planning the regular meetings and then the incidental ones. Meetings can be dependent on another meeting, which means they have to be given within a certain time frame after the start of the meeting they depend on. This can be used to make sure certain meetings are held at the same time (e.g., a practical lecture of a course being held simultaneously for two or more groups of students) or directly after each other (e.g., a normal lecture followed by a practical lecture).

There are some more constraints that have to be fulfilled to get to a good schedule: the room in which a meeting is scheduled must have enough capacity for all participants to fit in, and must be part of the room set of the meeting. This room set depends on the type of meeting (e.g. normal lecture, computer practical) and the faculty that the course is part of. These room sets are used to make sure meetings get scheduled in rooms that belong to the faculty and that can host the meeting type. This also minimizes the use of rooms that are far apart from each other, for the same course.

The objective function Kampman uses consists of four terms. The first term is the quarter penalty, which penalizes scheduling meetings before 9:00 and after 17:00. It also encourages the algorithm to schedule meetings as close to 13:00 as possible, because students prefer that time. There is also an empty room penalty, which makes sure a certain amount of *buffer* rooms is always empty, so they can be used to reschedule meetings in case some rooms are unavailable unexpectedly. Then, there is the room penalty, which penalizes the use of external rooms (usually rooms with very large capacity, which the university does not have) for which the university has to pay. Finally, to allow the algorithm to keep some meetings unscheduled (temporarily) without making the solution invalid, there is a penalty for unscheduled meetings.

The algorithm created by Kampman uses the local search meta-heuristic

Simulated Annealing. It is divided into two phases. The first phase creates a 'stamp', which is a schedule for one week, which only contains the regular meetings. This stamp is created in two steps: first an initial stamp is created using a greedy algorithm. Then, Simulated Annealing is used to optimize the stamp.

Now, to create a complete timetable, consisting of all weeks, the stamp is first 'imported' into the complete schedule. This means all regular meetings are now scheduled for the weeks they are supposed to be held. Next, incidental lectures are added to the timetable using a greedy algorithm. Finally, the complete timetable is optimized using Simulated Annealing. In this step the regular meetings are fixed, which means the local search can only reschedule the incidental meetings.

The greedy algorithm to create an initial schedule, iterates over all meetings and schedules them in the first possible room and period combination they fit in. For the order in which meetings and rooms are considered, different variants are tested, based on number of participants/capacity (high number of participant meetings/low capacity rooms are considered first) and on 'popularity', which takes into account how many meetings can be scheduled in a room (according to the room sets).

For the local search, operators are needed. Kampman uses two simple and three more complex operators. The simple operators insert a meeting into the timetable and move a lecture to another room/time. The first complex operator is a *branch and bound heuristic*. It selects a couple of meetings that have the same timeslot, faculty and type, which means these meetings can potentially clash. For the selected meetings a branch and bound algorithm is used to find the best room/time assignments for them. The second more complex operator shifts all planned meetings in a room as much towards 13:00 as possible. Finally, there is the *chain reschedule operator*, which tries to move not just one but multiple dependent meetings at a time.

Using a test set obtained from the university's planning environment, Kampman tested his algorithm to find the best combination of parameters and to compare it to just a greedy algorithm. The result was very clear: the local search outperformed the greedy algorithm by a big margin. Of course, the runtime of the local search was a lot longer, but with 10–15 minutes, the improvement of the schedule is definitely worth the wait.

3 Model

In this section we will give a formal description of the problem. The goal is to find a good timetable for one quarter of the academic year (called a block from here on) for the lab sessions in the *Victor J. Koningsberger building* of Utrecht University. Lab sessions of the Departments of *Biology*, *Chemistry* and *Pharmaceutical Sciences* and the *Faculty of Medicine/University Medical Center Utrecht* and the *Faculty of Geosciences* are held in the building.

The Koningsberger building has 4 floors with lab rooms, for a total of 17 lab rooms of 4 different types. We will be looking at 15 of those rooms, as the two other rooms are scheduled separately by the Department of Chemistry.

To create a model, we first needed to get a clear view of the problem. Therefore, we attended a meeting of the responsible project manager and several lab coordinators of the involved departments and faculties. This gave us an insight in part of the problem. After this, we had several meetings with the project manager and a developer of the IT section of the *Faculty of Science*. They gave a good overview of the problem.

It was clear that it was also necessary to speak to the lab coordinators of the departments to get all the necessary details of the problem. We had a meeting with each of the lab coordinators of the Departments of *Biology*, *Chemistry* and *Pharmaceutical Sciences*. In these meetings we asked several questions regarding the problem and the coordinators took their time to answer them. The coordinators have years of experience scheduling the lab sessions and their explanations gave a great insight into the problem. One of the coordinators was nice enough to show me around the lab rooms of the Koningsberger building, which also helped in understanding the problem.

After these meetings, we drafted the model. We consulted with the project manager to make some adjustments and come up with the model that is described in this section.

The input of the system is a list of requested meetings for each course. A meeting is the term we use to indicate a lab session. The meeting requests have multiple properties, like duration, preferred start moment and possible room types. Meetings can take 1 to 4 periods (of 2 hours) on one day, or can span over multiple days. Some meetings take as much as two weeks. In this time the students are not always working, but the room cannot be used for other meetings.

In section 3.1, we will first take a look at the terminology for different units of time in our model (e.g. block, week, period). After that, in section 3.2, important specifics of the building's rooms are described. In section 3.3 we will look at the different types of meetings and dependencies between meetings. Then, in sections 3.4 to 3.7, the hard and soft constraints, the

special constraints for improving a provided solution and the objective of the model are explained.

3.1 Time terminology

In this problem we will look at the timetable for a full *block*, or a quarter of the academic year. A block consists of ten *week*s and in each week there are 5 *day*s on which the meetings can take place. Holiday and retake weeks are left out of the problem altogether. Every day consists of four *period*s of 1 hour and 45 minutes[1]. We will call the four possible periods of a day *time*s from now on. A *period* is a specific combination of a week, day of the week and a time. Simple meetings (see section 3.3) take a number of periods, between 1 and 4. Each simple meeting must be planned completely on 1 day in consecutive periods.

The periods of each week are divided in four *timeslot*s. These timeslots are:

Timeslot A Monday 9:00–12:45 and Wednesday 9:00–12:45
(4 periods each week)

Timeslot B Tuesday 9:00–12:45, Thursday 13:15–17:00
(4 periods each week)

Timeslot C Monday 13:15–17:00, Tuesday 13:15–17:00,
Thursday 9:00–12:45 (6 periods each week)

Timeslot D Wednesday 13:15–17:00, Friday 9:00–17:00
(6 periods each week)

For almost all courses that use lab rooms in the Koningsberger building, these timeslots are combined to timeslot AD and BC, resulting in two timeslots that split up the 20 periods in the week evenly. Both of these timeslots take half of the Monday and two full days.

Timeslot AD Monday 9:00–12:45, Wednesday 9:00–17:00,
Friday 9:00–17:00

Timeslot BC Monday 13:15–17:00, Tuesday 9:00–17:00,
Thursday 9:00–17:00

[1]Each day consists of four periods of 1 hour and 45 minutes. These periods are at the following times: 9:00–10:45, 11:00–12:45, 13:15–15:00 and 15:15-17:00. If consecutive periods are combined, the meeting will just go on in the time between the periods.

Courses are usually assigned a timeslot in which they must hold all of their meetings. This is done in order to allow students to choose courses that do not have meetings at the same time. Some courses hold meetings in two different timeslots (e.g., AD and BC). Meetings are then held twice, once in each timeslot, and students can then choose in what timeslot they follow the course. It follows that it is imperative that meetings are scheduled in their assigned timeslot.

3.2 Rooms

As mentioned in section 1, the Koningsberger building contains 17 lab rooms divided over four floors. All rooms but one have a capacity of 30 students. Some rooms are next to each other and only separated by retractable walls. This means they can be combined into 'clustered' rooms with a capacity of 60 or 90 students. Other rooms are connected to each other through device rooms (see below). The rooms are split into four types, named type 1 through 4. Each of these room types serves a different purpose, however some meetings can be held in multiple types of rooms.

Type 1 are the simplest rooms, they contain very little special equipment. Meetings that are usually scheduled here can normally also be held in type 2 rooms, because they do not require special equipment. There are 8 rooms of type 1, divided over two clusters of three rooms and one cluster of two rooms. There are five rooms of type 2, which are used for analysis. Compared to type 1 rooms, they contain fume hoods on the side of the rooms and they are located next to large device rooms, containing devices that are often necessary for analysis. The three rooms of type 3 are specially designed for making pharmaceutical drugs. This means they are subject to stricter hygiene measures and therefore cannot be used for most other lab sessions.

There is only one room of type 4, which is used for chemistry. This is also the only room with a capacity of 48 students instead of 30. The scheduling of this room and the other room on the sixth floor (type 2) are both left out of our problem and their timetable is handled by the Department of Chemistry. This is done, because these rooms are used differently than the other rooms. For example, multiple meetings are held in the same room and the students rotate between different spots within a room over multiple weeks to perform different experiments.

3.2.1 Device Rooms

Besides regular lab rooms, the Koningsberger building also contains device rooms. These are rooms that contain various types of devices and other

equipment. These device rooms are situated next to one or two regular lab rooms. If a meeting needs a certain type of equipment, it is important that it is held in a room next to the relevant device room, to prevent that students need to walk through the building (with samples) to the device room.

3.3 Meetings

There are two types of meetings: simple meetings and multi-day meetings. In this section we will describe both types and the room and time dependencies there can be between meetings of the same course. Both types of meetings have the following properties: the number of necessary rooms, one or more room types the meeting can be held in and the necessary devices. The meetings of a course have a predefined order, which is important for the dependencies. This order must be non-decreasing in preferred start period.

Simple Meetings Simple meetings are meetings that take a number of periods on one day. The majority of the meetings is simple. These meetings are defined by a timeslot, a preferred start period (week/day/time) and a duration. The meeting can be moved to any moment in the preferred week or the next week, as long as the entire meeting stays within the timeslot and is planned on one day.

Multi-Day Meetings Multi-day meetings are meetings that are scheduled over multiple days. During this time, no other meeting can be held in the room(s). This does not mean that the room will be in use constantly throughout this time, but rather the room is not usable between meetings of the course. This can be because of hygiene rules or because the room will be completely filled with materials for the course. Multi-day meetings are defined by a start period and an end period. They can only be moved to exactly one week later.

Room Dependencies Some meetings need to be held in the same room as the previous meeting of the same course. This prevents having to move materials to other rooms between meetings. Therefore, each meeting B can have a room dependency on another meeting A (of the same course), if A ends before B starts (with respect to preferred start/end periods). This indicates that it is preferable that the meetings are held in the same room. Each meeting can have only one room dependency on another meeting, but multiple meetings can have a dependency on the same meeting. To have a dependency between two meetings, the meetings must have at least one

possible room type in common, otherwise it is impossible for them to be in the same room.

Time Dependencies In a similar fashion to room dependencies, each meeting B can have a time dependency on another meeting A (of the same course), if A starts before or in the same period as B (with respect to preferred start periods). A time dependency indicates meeting B has to start at least a given number of periods or days after meeting A. This number of periods is typically either 0 to indicate meeting B has to start no earlier than meeting A, or the number is equal to the duration of meeting A to indicate meeting B has to start after meeting A has ended. It can also be set to 1 day, to indicate meeting B must start the day after meeting A, or later.

3.4 Hard Constraints

In this section we will go through all the hard constraints of our model. Where necessary we will add some remarks or a motivation as to why the constraint is necessary. To get a valid timetable, all hard constraints must be satisfied.

HC1: Each meeting must be scheduled in consecutive periods (and on the same day if it is a simple meeting), in the same room(s). This means it is not allowed to split a meeting into parts and hold those parts at different moments or in different rooms.

HC2a: Each simple meeting must be scheduled in (a) period(s) of its timeslot. It is very important to obey the timeslots, because otherwise students may have multiple lectures/lab sessions at the same time.

HC2b: Each multi-day meeting must be scheduled at the requested moment of the week. A multi-day meeting can only be moved by exactly one week (if not exceeding the end of the block), it cannot be moved to other periods within a week.

HC3: Each meeting must be scheduled in (a) room(s) of its possible room types. Lab sessions may need equipment that is only available in the requested room type. This means they absolutely have to be held in a room of that type, otherwise the experiments cannot be done.

HC4: Each meeting must be scheduled in the number of rooms that it needs. All students have to fit in the rooms, otherwise dangerous situations can arise and the students cannot properly perform their experiments.

HC5: Only one meeting can be scheduled in each room in each period. Even if the students of two meetings together might fit in one room, it is still not practical to do this.

HC6: Each meeting must be scheduled in the requested week, or the next week. We enforce this to make sure course schedules do not get thrown off too much. It is usually easier to hold a lab session a week later than to hold it a week earlier, because lab sessions often need to be preceded by some sort of introduction lecture. Therefore we do not allow meetings to be held a week earlier. If a meeting is requested in the second last week of the block, it must be scheduled in that week, because the last week of the block is the exam week. Of course, if a meeting is requested in the last week of the block it must be scheduled in that week.

HC7: No other meetings can be scheduled in a room when it is under ML-I conditions. ML-I conditions are in effect for safety reasons when certain types of samples are used. After these have been used, the room has to be cleaned and inspected before it can be used again. We solve this problem by making all ML-I sessions multi-day meetings, where the requested time includes the cleaning and inspection.

HC8: In given periods of the block, no meetings can be scheduled. For example, on holidays, all periods are unavailable. This does not apply to multi-day meetings, they can be scheduled during unavailable periods.

3.5 Soft Constraints

In this section we will look at all the soft constraints of our model. For each soft constraint, remarks and motivations are added where necessary. We will also explain how violations of each constraint are penalized (see also section 3.7).

SC1: All meetings must be scheduled. The first soft constraint is probably the most important one. A schedule is not complete until every meeting has been scheduled. This constraints is a soft constraint, because this allows us to have an initial feasible solution even though not all meetings have

been scheduled yet. For each meeting that is not scheduled, a penalty of $NotScheduledPenalty$ is added.

SC2: Each meeting that needs multiple rooms, should be scheduled in combinable rooms or rooms that are connected through a device room. Combinable rooms (all type 1) are rooms that are split only by a retractable wall (see section 3.2). Some other rooms (type 2 and 3) are connected through a device room, which means going from one to another does not involve walking through the hallways. If it is not possible to have all rooms adjacent (e.g., because there are simply too many rooms needed), they should be divided into balanced groups. For example, two clusters of two rooms is better than a cluster of three rooms and one separate room. If this is necessary the rooms should be on the same floor.

For each meeting, a penalty of $(\#clusters - \min(\#clusters)) \times ClusterPenalty$ is added, where $\min(\#clusters)$ is the minimum number of clusters the meeting can be scheduled in. There will also be a penalty of $ClusterBalancePenalty$, if one or more of the clusters are unbalanced, i.e. do not contain $\lfloor \#rooms/\#clusters \rfloor$ or $\lceil \#rooms/\#clusters \rceil$ rooms. For each cluster of more than one room, that is connected through (a) device room(s) instead of (a) retractable wall(s), a penalty of $ClusterDevRoomPenalty$ is applied. A penalty of $(\#floors - \min(\#floors)) \times ClusterFloorPenalty$ is added as well, where $\min(\#floors)$ is the minimum number of floors the meeting can be scheduled on.

SC3: Each meeting with a room dependency should be scheduled in the same room(s) as the meeting it depends on. For each dependency, if Δ rooms are different between the two meetings, a penalty of $\Delta \times RoomDepPenalty$ is inflicted. If the two meetings do not need the same number of rooms, Δ is calculated as $\Delta = \min(\#rooms) - \#equalrooms$, where $\min(\#rooms)$ is the minimum of the number of required rooms of both meetings and $\#equalrooms$ is the number of rooms that host both meetings.

SC4: Each meeting should be scheduled in the requested period(s). The course coordinators make preliminary schedules for their courses. It is desirable to deviate from those as little as possible, because otherwise the coordinators have to change up other parts of their schedules as well. If a simple meeting is moved to another period within the requested week, a penalty of $\#rooms \times MovePeriodPenalty$ is added (where $\#rooms$ is the number of rooms the meeting needs). If it is moved to exactly one week later,

a penalty of $\#rooms \times MoveWeekPenalty$ will be added to the total. And if it is moved to another period in the next week, those penalties are both added. If a multi-day meeting is moved to exactly one week later (which is the only move possibility), this yields a $\#rooms \times MoveMDPenalty$ penalty.

SC5: If the meeting requires one or more devices, it should be held in a room adjacent to a device room containing (one of) the device(s). For each cluster of rooms the meeting is scheduled in, a penalty of $\#rooms \times DevicePenalty$ is added if none of the rooms in the cluster are adjacent to such a device room. If at least one of the rooms in the cluster is adjacent to a relevant device room, a penalty of $DeviceSecPenalty$ is added for each room in the cluster that is not adjacent to a relevant device room. This penalty should be small compared to the former penalty, and is added to stimulate the planning of the meeting in two rooms on either side of a necessary device room, over having it in one room next to the device room and a room connected through another device room that is not necessary for the meeting.

SC6: Each simple meeting that takes 2 periods, should not be scheduled from 11:00 to 15:00. This is undesirable, because it clashes with lunch time and it may make it more difficult to plan other meetings (like lectures) for the course on the same day. For each simple meeting that is scheduled from 11:00 to 15:00 a penalty of $\#rooms \times LunchPenalty$ will be added.

SC7: For each room, there should be enough time to setup and clear it and this time should be as close before and after each meeting as possible. For some meetings, there needs to be time to put materials in the room beforehand and to clear materials after the meeting. However, there is very little information on how much time is needed for this for each meeting. Therefore, we apply general penalties for each meeting to pursue a good distribution of empty time over the rooms and over time. These penalties are used as a rule of thumb, because (as mentioned before) there is not much information available. These rules are only applied to simple meetings. Multi-day meetings take care of their setup and clear time within the requested duration.

For each meeting and each room it is scheduled in, we look at the number of periods the room is empty ($EmptyPeriodsBefore$) in the 12 periods (3 days) prior to the meeting. If this number is smaller than 6, we apply a penalty of $(6 - EmptyPeriodsBefore) \times EmptyBeforePenalty$.

If the previous meeting in the room is of the same course, we apply no penalty. We do the same for the number of empty periods after the meeting ($EmptyPeriodsAfter$). We apply a penalty of $(6 - EmptyPeriodsAfter) \times EmptyAfterPenalty$, unless the next meeting in the room is of the same course.

For each meeting and each room it is scheduled in, we look at how many days before the start of the meeting the last empty period in that room was ($LastEmptyDaysBefore$). If a meeting starts at 9:00 and the room is empty the last period of the day before, this counts as 0 days, otherwise it is just the difference in days (disregarding time of the day completely). A penalty is added of $LastEmptyDaysBefore \times LastEmptyPenalty$. If the previous meeting in the room is of the same course, no penalty is applied. We do the same for the number of days between the end of the meeting and the first empty period in the room ($FirstEmptyDaysAfter$), where (once again) the difference between the last period of a day and the first period of the next day is counted as 0 days. A penalty of $FirstEmptyDaysAfter \times FirstEmptyPenalty$ is applied, except if the next meeting in the room is of the same course.

SC8: The time dependencies of all meetings have to be respected. Often meetings of the same course have to be held during different periods, simply because they are attended by the same students and/or the same staff. The order of the meetings can be important because the results of one meeting are used in the next. See section 3.3 for more details on time dependencies. Unlike most other soft constraints, this one should absolutely not be violated in the final solution, for the reasons mentioned before. The reason this is a soft rather than a hard constraint is that it allows for more flexibility in the solution during the search process. For each time dependency that is violated, we apply a (high) penalty of $TimeDepPenalty$.

3.6 Additional Constraints for Optimizing an Initial Solution

The end user may provide the optimization program with an initial schedule. This could be a handmade schedule, or a schedule that was created by our program and then slightly tweaked. In this case we add some constraints to make sure certain elements from the provided schedule are not changed, or not changed too much.

HC9: If a meeting is locked in a certain period and/or (a) certain room(s), the period and/or room(s) cannot be changed. A meeting can be locked in place, to prevent its position from being changed. For example, if the end user has consulted with a course coordinator to find (a) suitable period(s) for a meeting in a busy week, this selected period may not be optimal. However it is necessary to keep the meeting in that selected period. For this reason, the period can be locked. The same can be done with the room(s) of a meeting.

SC9: No more than a given number of meetings can be moved. It is possible to provide the application with a maximum number $MaxChanges$ of meetings that can be moved from their position in the initial solution. If more meetings' positions have changed, a penalty of $(\#moved - MaxChanges) \times ChangePenalty$ is applied. It is possible to select whether $\#moved$ is the number of meetings with a changed period, or whether meetings with (a) changed room(s) are also included. Each meeting will only be counted once though, even if the period and room(s) have changed.

3.7 Objective

The objective of our model is to find the feasible timetable with the lowest sum of all penalty points. A valid timetable consists of an allocation of a start time and (a) room(s) for each scheduled meeting, that conform to all hard constraints. In section 5.2 an overview is given of all parameters that are used in the penalty calculations, together with the values that are used during the experiments.

4 Approach

In this section we will give an overview of the approach we take to find a good timetable. Our approach is based on the metaheuristic *Simulated Annealing*. In section 4.1 our general approach is introduced, then, in section 4.2, the main algorithm is described in detail. Finally, in sections 4.3 to 4.5 the specific components of the Simulated Annealing algorithm are explained.

4.1 General Approach

Our method is based on a metaheuristic. We choose to use a metaheuristic for several reasons. Firstly, metaheuristics performed well at the *curriculum-based course timetabling* problem, which is similar to (although simpler than) our problem. See section 2.1 and [ATM09] [Mül09] [LuH10] [BCD16].

Secondly, speed is important. The method does not have to be super fast, but it must not be very slow, because the end users will not want to wait for a long time for a solution. Especially if they want to make some changes to the input and rerun the optimization, it will be annoying to wait for extended periods of time. Besides that, it is not important to find the optimal solution at all cost. The goal is to find a good solution, and a metaheuristic should find a good solution in less time than for example an exact method.

The metaheuristic we will be using is *Simulated Annealing*. We choose for Simulated Annealing because it has been applied to many varying optimization problems with success. It has also been applied to the curriculum-based course timetabling problem with success by Bellio et al. [BCD13] [BCD16]. Even though they used a very simple version of Simulated Annealing, without restarting or reheating, the results were good. Our version of Simulated Annealing does include restarting and reheating. Kampman [Kam13] used Simulated Annealing to schedule course meetings for the entirety of Utrecht University.

In the remainder of section 4 the components of our Simulated Annealing algorithm are shown. In section 4.2 the main Simulated Annealing algorithm is described. Then, in section 4.3, we show the preprocessing that happens before the optimization is started and section 4.4 shows how the initial solution is constructed. Finally, section 4.5 shows the neighborhood operators that are used in the Simulated Annealing algorithm.

4.2 Main Algorithm

In this section we take a look at our main algorithm, which is a form of Simulated Annealing [KiM03] [AKM14] [BCD16]. The pseudocode for the

algorithm can be found in algorithm 1. The algorithm starts by getting an initial solution. How we get this initial solution, is explained in section 4.4. After that, we set the temperature to its initial value.

The algorithm consists of four nested loops. The outer loop, on lines 4–18, is the "restart loop". At the start of this loop the current solution s is set to the best solution found so far ($\bar{s}$). This loop stops (*stop condition* 1) after a set number of iterations without an improvement of $\bar{s}$. This set number is defined by the $\#Restarts$ parameter.

The second loop (lines 6–17) is the "reheat loop". At the end of this loop, the temperature is reset to the reheat temperature. This means increases of the solution's penalty value are accepted with a larger chance again, which allows the algorithm to get away from a local optimum. This reset happens at the end of each iteration to allow for the reheat temperature to be different from the initial temperature. The loop ends (*stop condition* 2) after $\#Reheats$ consecutive iterations in which the penalty of s increases or $\#ReheatsGlobal$ iterations without improvement of $\bar{s}$.

The third loop (lines 7–15) is the main loop. This loop continues its iterations (*stop condition* 3) until no improvement of the solution is found for $MainLoopCutoff$ consecutive iterations of the inner loop (lines 8–13). After every $InnerLoopIterations$ of the inner loop, the temperature is decreased by multiplying it by $CoolingRate$. In the inner loop, a random operator is applied (see below). The new solution is used from then on. If this new solution is better than the best found solution so far, we save it to $\bar{s}$. The "restart loop", "reheat loop" and the main loop are all also stopped if a time or iteration limit is provided and it is reached.

How the operators are applied is shown in algorithm 2. First we get a random operator. The operators all have an associated chance to be chosen. In section 5 we research the best distribution of these chances. We run the selected operator. This run can fail for several reasons. For example, if there are no unscheduled meetings, the insert operator will fail. If this happens, we return the previous solution. If the new found solution is better than the previous one, or if the increase in penalty is accepted, we return the new solution. If the penalty increase is declined, we return the previous solution. An increase Δ of the penalty is accepted with a probability of $\exp(\frac{-\Delta}{T})$. All neighborhood operators are described in section 4.5.

4.3 Preprocessing

Before starting the optimization (or even creating an initial solution) we will do some preprocessing on the input. For example, all rooms and all periods get an index. This means we can create a 2D array containing fields for

Algorithm 1 Simulated Annealing

```
1: function SimulatedAnnealing()
2:     s̄ ← GetInitialSolution()                    ▷ s̄ keeps the best solution
3:     T ← InitialTemp
4:     while ¬stop condition 1 do
5:         s ← s̄
6:         while ¬stop condition 2 do
7:             while ¬stop condition 3 do
8:                 for all i ∈ {1, ..., InnerLoopIterations} do
9:                     s ← ApplyOperator(s, T)
10:                    if s < s̄ then                    ▷ Compare penalties
11:                        s̄ ← s
12:                    end if
13:                end for
14:                T ← CoolingRate × T
15:            end while
16:            T ← ReheatTemp
17:        end while
18:    end while
19:    return s̄
20: end function
```

Algorithm 2 Apply Operator

```
1: function ApplyOperator(s, T)
2:     Operator ← GetOperator()
3:     s′ ← Operator(s)
4:     if Operator failed then
5:         return s
6:     else if s′ ≤ s then
7:         return s′
8:     else if Random([0, 1)) < exp((s−s′)/T) then
9:         return s′
10:    else
11:        return s
12:    end if
13: end function
```

all possible period/room combinations in which we store the meeting that is currently planned there (if any). For each meeting, we create a list of all rooms that have the requested room type(s). If a meeting's rooms are locked, only the selected rooms are added to this list, which means the meeting will always be scheduled in these rooms (HC9, see section 3.6).

We also find all possible start periods for simple meetings. These periods have to be within the requested week or the next week, and the entire meeting must fit on that day and within the provided timeslot. We also make sure that the meeting does not overlap with periods that are unavailable (HC8, see section 3.4). For each possible start period, we precalculate the corresponding penalties that only depend on the selected start period for the meeting (SC4 and SC6, see section 3.5). If a meeting's start period is locked, only the initial start period will be added as possibility to the list of start periods (HC9).

For multi-day meetings, there are always two possible start periods, namely the requested start period and the period exactly one week later (unless this means the meeting gets moved into the exam week or outside the block). After these steps of preprocessing, simple and multi-day meetings can be processed almost equally, because both now have a set of possible start periods and a duration. One exception to this is that some penalties are not applied to multi-day meetings.

Another step of preprocessing that we apply is looking at the time dependencies and eliminating all possible start periods that are not possible because of those dependencies. For example, if meeting B has to start at least a day after meeting A, all possible start periods of B that are not at least a day after the first possible start period of A can be eliminated. Time dependencies that no longer have any effect (last possible start period of meeting A does not clash with first possible start period of meeting B) are removed as well. If at the end of preprocessing there are meetings with no possible start periods left, or with fewer possible rooms than the number of rooms requested, those meetings are discarded. Of course, the penalties for not scheduling them (SC1) will still be applied.

Under normal circumstances, meetings will never have to be discarded. As long as the requested start periods respect the timeslots and the time dependencies, there will always be a possible start period for each meeting (the requested start period). Normally, the requests will respect this. However, it could be that an end user locked a certain meeting in one place and that placement makes the scheduling of another meeting impossible (HC9). In this case the other meeting will not be scheduled. During our experiments, however, this will never happen because of the way the test instances are constructed. A shortage of possible rooms can only occur if more rooms are requested than there are available of the necessary room type.

4.4 Initial Timetable

At the start of the Simulated Annealing algorithm, an initial feasible solution is needed (see line 2 of algorithm 1). Because the constraint that all meetings must be scheduled, is a soft constraint, an obvious choice as initial solution is an empty timetable. This was used during most of our experiments. However, we also wanted to see if using a different initial schedule could have a positive effect on the resulting schedule or the runtime.

Therefore, a greedy algorithm was designed. [LuH10], [BKA11] and [Kam13] all use a greedy algorithm to create one or more initial solutions, whereas [WaH16] focuses specifically on different heuristics that can be used. Our greedy algorithm is shown in algorithm 3. Every iteration, one unscheduled meeting is selected based on a certain statistic. Then, every possible start period/room(s) combination is checked for feasibility and for the effect on the schedule. This effect is measured using yet another statistic. The best combination is selected and the meeting is scheduled there. If there are no feasible start period/room(s) combinations for the meeting, it is discarded and will be left unscheduled.

Algorithm 3 Initial Greedy Scheduler

```
1:  function InitialGreedyScheduler(s)
2:      s' ← s
3:      M ← UnscheduledMeetings(s')
4:      while |M| > 0 do
5:          m ← SelectMeeting(M)
6:          M ← M \ {m}
7:          s' ← ScheduleMeeting(s', m)
8:      end while
9:      return s
10: end function
```

We will now look at what statistics were used to select the meeting on line 5. Three different statistics were tested, all aiming to select the meeting that is hardest to schedule:

Available positions For each meeting, the number of "available positions" is calculated, and the meeting with the least positions available is selected. This heuristic is related to the *saturation degree* shown by [WaH16]. Ties are broken using the *schedule load* statistic shown below. The number of available positions indicates how many feasible possible start period/room(s) combinations there are in the current

schedule. It is calculated as follows (where P_m is the set of possible start periods of m and R_m is the set of possible rooms of m):

$$\mathrm{AP}(m) = \sum_{p \in P_m} \mathrm{PAP}\left(m, \sum_{r \in R_m} \mathrm{IsFree}(m, p, r)\right) \tag{1}$$

$$\mathrm{PAP}(m, f) = \begin{cases} f / \mathrm{NumRooms}(m) & \text{if } f \geq \mathrm{NumRooms}(m) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$\mathrm{IsFree}(m, p, r) = \begin{cases} 1 & \text{if } m \text{ can be scheduled in room } r \text{ at period } p \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

For each possible start period, the number of free rooms f is divided by the number of requested rooms of the meeting. If there are not enough rooms, the start period is not counted altogether.

Schedule load The schedule load of a meeting indicates the space a meeting takes in the schedule. It is calculated as $\mathrm{SL}(m) = \mathrm{NumRooms}(m) \times \mathrm{Duration}(m)$. The meeting with the highest load is selected, because it is assumed to be the most difficult to schedule. Ties are broken using the *available positions* measure shown above.

Room load This measure is almost the same as the *schedule load*, but the number of possible rooms for a meeting is also taken into account. The room load is calculated as follows: $\mathrm{RL}(m) = \mathrm{NumRooms}(m)/|R_m| \times \mathrm{Duration}(m)$. Once again, the meeting with the highest load is selected, because it is assumed to be the most difficult to schedule. Ties are broken using the *available positions* measure shown above.

If both the main statistic and the tie breaker are equal, the tie is broken at random. We used two different measures to select the best possible start period/room(s) combination (line 7):

Penalty This one is pretty straight-forward: whichever start period/room(s) combination results in the lowest resulting solution penalty, is selected.

Total available positions This measure is based on the *available positions* measure used to select meetings. The sum over all meetings (in M) of the *available positions* for each meeting is taken and the start period/room(s) combination that results in the highest sum is selected. A higher value for this measure should indicate that there are more possibilities to schedule the remaining meetings in M.

In both cases, ties between start period/room(s) combinations are broken at random.

Of course, it is also possible to start the algorithm with an existing timetable. That timetable could, for example, result from a previous run of the algorithm, but with a few lectures' positions altered. The greedy algorithm can be applied on an existing solution as well.

4.5 Neighborhood Operators

The most important component of Simulated Annealing are the neighborhood operators. They are imperative to the quality of the algorithm. In this section we will go through all neighborhood operators we created. In section 4.5.11 we describe how the penalty of a new solution is calculated when a neighborhood operator is applied.

Each iteration of Simulated Annealing, one operator is selected randomly and then applied. In section 5 we research what the best distribution of probabilities is for the operator choice. We also show some statistics of the operators' performances, like the percentage of improvements and the average change in penalty value induced by the operators over the course of a run of the algorithm.

4.5.1 Insert operator

The *Insert* operator is a simple operator, but is unmissable if the initial solution is an empty timetable. If there is at least one unscheduled multi-day meeting, a random unscheduled multi-day meeting is selected, otherwise we select a random unscheduled simple meeting. Multi-day meetings are inserted first to make sure there is space to insert them. They are a lot harder to insert in a busy schedule than simple meetings, because they are usually longer and have only two possible start periods.

The possible start periods of the selected meeting are either shuffled randomly or sorted in order of increasing penalty for SC4 and SC6 (see section 3.5). In this order the start periods are considered. For a start period, the possible rooms that are free are considered in random order. Once we have found enough free rooms, we insert the meeting there. If there are not enough free rooms, the next possible start period is considered. This operator fails if there are no unscheduled meetings or if the selected meeting cannot be inserted.

4.5.2 Remove operator

As the name implies, the remove operator does the opposite of the insert operator. It gives simple meetings priority over multi-day meetings in its random selection, to make sure there are never unscheduled multi-day meetings while there are scheduled simple meetings. The selected meeting is removed from the schedule. Obviously, this will (almost) never result in an improvement of the solution, but it might be able to help the algorithm get out of a local optimum when the temperature is still high. This operator fails if there are no scheduled meetings.

4.5.3 Move Period operator

This operator is aimed at changing only the period a meeting starts in. Like the *Insert* operator, we consider the possible start periods in either random or sorted (by penalty for SC4 and SC6) order. For each start period that is considered, first the rooms that the meeting is currently scheduled in are checked. Each room that is unavailable for the new start period is replaced by a random free room. If there are not enough free rooms, the next possible start period is considered. Otherwise, the meeting is scheduled in the selected start period and rooms. This operator fails if there are no scheduled meetings, if the selected meeting has only one possible start period or if there is no space in any of the other possible start periods.

4.5.4 Move Room operator

The *Move Room* operator changes one room of a meeting. One of the randomly selected meeting's room is selected at random and replaced by another of the possible rooms that is free. The operator fails if there are no scheduled meetings, if the selected meeting has no more possible rooms than the number of rooms it needs or if none of the other possible rooms is free.

4.5.5 Move Cluster operator

The *Move Cluster* operator is more advanced than the previous operators. Instead of selecting one room to move, it selects an entire cluster of rooms (see section 3.2) and removes the meeting from those rooms. Then, a random (free) replacement room is selected. Then, while not enough rooms have been selected for the meeting yet, a new cluster is "grown" from the replacement room. To do this, the rooms next to the cluster are checked (if they exist) for availability and one of them is added to the cluster if available. If both

are available, one is chosen at random. If no rooms are available next to the cluster, a random room is selected and a new cluster is started from there.

4.5.6 Move Random operator

This operator is a remove and insert operator chained together. A randomly selected scheduled meeting is removed from the schedule and then inserted in the same way the insert operator works. This operator fails if there are no scheduled meetings or if the selected meeting has only one possible start period and only as many possible rooms as the number of rooms it needs to be scheduled in, which would mean the meeting will always be inserted in the place it was removed from.

4.5.7 Change Day Rooms operator

This operator is the most complicated one we implemented. This operator aims to select (a) good room(s) for each meeting on a day, without changing the period(s) in which the meeting is held. All meetings on the day that can be moved to other rooms, are removed from the timetable and divided into groups which do not conflict with each other. Then, in some order, the meetings of each group are placed back in the (combination of) room(s) that gives the best resulting penalty value.

The pseudocode of the algorithm can be found in algorithm 4. We will go through the algorithm here. Firstly, on lines 2–4, 9–11 and 19–21 the operator fails, because there are no meetings that can be moved. On lines 5–7 a copy of the solution is made (in the pseudocode, this does not happen in the actual program), a random day in the timetable is chosen and the meetings scheduled on that day are fetched. On line 8, meetings that cannot be moved to any other room(s) than the one(s) they are currently in are discarded from this set. All meetings are then backed up and removed from the timetable (lines 12–13).

In the loop on lines 14–18, each meeting's available rooms in the newly cleared schedule are checked. If these are exactly the number of rooms the meeting needs, the meeting cannot be moved and it will be put back in the schedule in the rooms it was scheduled in before (which will be the exact same rooms that are available for the meeting). This is repeated until no meetings are removed from M, because placing back a meeting may restrict the movement of other meetings.

Then, on line 22, the meetings are split into groups. Groups are split into as many groups as possible, without violating the following rule: two meetings that have at least one period and one room in common, must be

in the same group. The result is that meetings in different groups can never conflict, i.e., they cannot both be placed in any room at the same time. We can now treat all groups completely separately, because their assignments will never conflict.

Now, for each group (lines 23–36) we first order the meetings (line 24). This can be done randomly, in descending order of the numbers of rooms the meetings need or in descending order of the durations of the meetings. In this order, we go through the group's meetings (lines 25–35). For each meeting, we first get the available rooms in the schedule (line 26). If the number of available rooms is smaller than the number of rooms the meeting needs, the scheduling of this group is canceled, all of the group's meetings are reset to their original positions and we move on to the next group (lines 28–31). Otherwise, we check all combinations of the available rooms and find the one that induces the best penalty value (lines 32–33). On line 34, we place the meeting in that combination of rooms.

If none of the groups resulted in a valid change of room assignments (i.e., all of the groups were reset by line 29 at some point), the operator fails. Otherwise, s' is returned.

4.5.8 Move Best Rooms operator

The *Move Best Rooms* operator, like the *Change Day Rooms* operator, looks at all combinations of the available rooms to place a meeting. However, at first it only randomly selects one simple meeting to move. The meeting's start period is not changed, but all combinations of rooms that do not have a multi-day meeting in them during the considered meeting are tested. The combination that results in the best resulting penalty value is selected. Other simple meetings that obstruct the selected rooms, are removed from the timetable. The penalty change resulting from removing these meetings is not included in the comparison of the room combinations.

The considered meeting is then moved to the selected rooms. The operator then attempts to reinsert all removed meetings in random order. For each of the removed meetings, all possible start periods are considered, ordered either randomly or by penalty for SC4 and SC6 (like the *Insert* operator). The meeting is scheduled in the first start period that has enough rooms available. The algorithm looks at all combinations of the available rooms. The best combination of rooms is selected and the meeting is scheduled in those rooms. If there is no start period in which enough rooms are available, the meeting will be left unscheduled.

This operator fails if there are no scheduled simple meetings, if the selected meeting only has as many possible rooms as the number of rooms

Algorithm 4 *Change Day Rooms* operator

```
1: function ChangeDayRooms(s)
2:     if |scheduled meetings| = 0 then
3:         fail
4:     end if
5:     s' ← s
6:     day ← random day
7:     M ← GetMeetings(day)
8:     M ← {m | m ∈ M, |PossibleRooms(m)| > NumRooms(m)}
9:     if |M| = 0 then
10:        fail
11:    end if
12:    backup ← Backup(s', M)
13:    s' ← Remove(s', M)
14:    repeat
15:        M' ← M
16:        M ← {m | m ∈ M, |AvailableRooms(m)| > NumRooms(m)}
17:        s' ← Reset(s', M' \ M, backup)
18:    until M = M'
19:    if |M| = 0 then
20:        fail
21:    end if
22:    G ← SplitGroups(M)
23:    for all g ∈ G do
24:        M_g ← OrderMeetings(M_g)
25:        for all m ∈ M_g do
26:            R ← AvailableRooms(m)
27:            numRooms ← NumRooms(m)
28:            if |R| < numRooms then
29:                s' ← Reset(s', M_g, backup)
30:                break
31:            end if
32:            C ← Combinations(R, numRooms)
33:            r ← argmin_{c∈C} Place(s', m, c)
34:            s' ← Place(s', m, r)
35:        end for
36:    end for
37:    if all groups failed then
38:        fail
39:    end if
40:    return s'
41: end function
```

it needs to be scheduled in or if the only available rooms are the ones the meeting is already planned in. From the experiments in section 5.11 onwards and in the final version of the algorithm, not only simple meetings can be randomly selected to be moved, but multi-day meetings can as well.

4.5.9 Move Period Best Rooms operator

The *Move Period Best Rooms* operator is very similar to the *Move Best Rooms* operator. The only difference is that instead of just changing the rooms, this operator also changes the start period. All possible start periods except the currently selected one are considered, in either random order or sorted by penalty for SC4 and SC6 (like the *Insert* operator). The first start period where enough rooms are available (simple meetings will be removed to fit in the meeting) is chosen. From there on the process is the same as for the *Move Best Rooms* operator. Like for the *Move Best Rooms* operator, multi-day meetings can be selected by this operator from section 5.11 onwards as well.

4.5.10 Insert Best Rooms operator

This operator is a combination of *Insert* and *Move Period Best Rooms*. It is used from the experiments in section 5.11 onwards and in the final version of the algorithm. A random unscheduled meeting is selected and than inserted in a similar way to *Move Period Best Rooms*. Like *Insert*, unscheduled multi-day meetings are given precedence over unscheduled simple meetings. The operator fails if there are no unscheduled meetings or if there is no position in the schedule where the meeting fits (simple meetings will be removed to fit in the meeting).

4.5.11 Penalty Calculation

Every time a neighborhood operator is applied, the penalty of a solution needs to be reevaluated. However, none of the penalties are global, i.e., when moving a meeting, only penalties related to it and some other meetings have to be reevaluated. For example, the penalties for deviating from the preferred start period and for scheduling through lunch only need to be recalculated for a meeting with a changed start period. On the other hand, the penalties for SC7 (see page 23) may change for multiple meetings when just one meeting is moved.

At the start of the program, the initial penalty of the empty solution is set to $\#meetings \times NotScheduledPenalty$. All neighborhood operators are built up from *place* and *remove* function calls. Those insert and remove a

single meeting in/from the schedule, respectively. There is no function to directly move a meeting; this has to be done by removing and then inserting it. When a meeting is placed, the induced penalty change is calculated and added to the solution's penalty value. This includes subtracting the *NotScheduledPenalty*. The part of the penalties that are only dependent on the meeting's start period and room(s) assignment (the penalties for SC1, SC2, SC4, SC5, SC6, see section 3.5), is saved. This way those penalties do not have to be calculated when removing the meeting. The other penalties for removing the meeting are calculated in the same way as when inserting and then negated.

In our implementation of the algorithm, there is only one solution object. The best found solution is saved as a backup of the actual solution. When the backup is restored, all meetings are put in the correct place in the schedule and the penalty is simply reset to the backup's penalty value. Neighborhood operators use partial backups. Before placing or removing a meeting, it is added to the partial backup. If an operator fails or its result is declined, the partial backup is restored, which works similar to a full backup, only with just a (usually small) subset of the meetings.

5 Experiments

In this section we give an overview of the research we performed to find the best parameters for our Simulated Annealing algorithm. In sections 5.1 to 5.2, we first describe some of the preliminaries that had to be taken care of before the experiments could be performed. Sections 5.3 to 5.5 describe our experiments to find good parameters for the Simulated Annealing algorithm and good probabilities of selecting each neighborhood operator in an iteration.

Experiments regarding different types of initial solutions for the algorithm are shown in section 5.6. Then, in section 5.7, we show some statistics of the used problem instances and, in section 5.8, we compare the results of our approach to some handmade timetables. Section 5.9 contains a brief analysis of the neighborhood operators' performances and section 5.10 shows the results of additional experiments to tune the Simulated Annealing parameters and the neighborhood operators' probabilities. Section 5.11 shows how we improved the algorithm to make it better at moving multi-day meetings and, finally, section 5.12 describes the experiments we performed with initial schedules with locked meetings.

5.1 Generated Problem Instances

Our problem is a real-world problem, so we do have real-life instances for the first two blocks of the academic year 2017–2018. However, these instances are not very difficult to solve and we do want our method to be able to cope with more difficult instances as well. This way, the method will still be useful if the building gets used more in the future. Therefore we built an application that can generate problem instances. During our experiments we use three generated instances consisting of courses that are made to resemble different types of real courses. Of course, these generated instances have more of these courses than the real instances. Statistics of the three instances can be found in section 5.7.

5.2 Penalty Parameter Values

Another thing that needs to be taken care of before any experiments can be done, is setting the values of the penalty parameters (see section 3.5). We first set initial values based on the importance of the soft constraints as indicated by the departments' lab coordinators and using inspection of the resulting timetables.

Even though the parameters of the algorithm had not been optimized yet, we were able to use the algorithm to generate timetables for the real-life instance of the first quarter of the next academic year using multiple, slightly tweaked, penalty parameter sets. We showed these timetables to the project manager and the lab coordinators, in order to find a set of penalty parameters that represent the importance of all soft constraints.

We had a meeting with the project manager and one of the lab coordinators. With them, we tweaked the parameters to get the final parameter set. The penalty parameter values of the final set are shown in table 1. Despite the fact that the algorithm was not yet optimized, the project manager and lab coordinator were already happy with the quality of the shown schedules compared to the handmade schedules. Because the deadline of submitting the final schedules was very close at that time, it was decided to use the generated schedules (using the final penalty parameter set) for the first two quarters of the 2017–2018 academic year over the handmade schedules.

5.3 Experiments for Initial Simulated Annealing Parameters

In this section we show and discuss the results of our experiments to find good initial parameters for our Simulated Annealing algorithm. We need to find good values for two groups of parameters: the operator chances and the Simulated Annealing parameters. In the next section, we look into finding a good distribution of the operator probabilities. After that, we will look into good Simulated Annealing parameters. To perform the experiments for the former, however, we do need reasonable Simulated Annealing parameters. Therefore we first perform some experiments to find those initial parameters.

The results of these experiments can be found in appendix A. Table 24 shows the initial parameter values that we found using trial-and-error during the development of our approach. We start with these values, and change one or two parameters. We test multiple values and decide which performs best. We then use this best value in the following experiments and move on to changing another parameter.

All experiments were run on a PC running Windows 7, with an Intel Core i7 4770K (3.5 GHz, quad-core) CPU and 16 GB DDR3 RAM. To speed up the experiments, we always start three experiments at the same time. Although this affects the runtime of the algorithm, this increase is only between 1% and 4%, because of the quad-core CPU used. Since we do not look very strictly at runtime during our research, this should not affect the results significantly.

The first parameter we varied was the *CoolingRate*. The results can be

Table 1: The 18 parameters used for the penalties applied by the soft constraints. The soft constraint that uses each parameter is listed, as well as the value used during the experiments.

Parameter name	Soft constraint	Value
$NotScheduledPenalty$	SC1	6000
$ClusterPenalty$	SC2	150
$ClusterBalancePenalty$	SC2	90
$ClusterDevRoomPenalty$	SC2	50
$ClusterFloorPenalty$	SC2	80
$RoomDepPenalty$	SC3	150
$MovePeriodPenalty$	SC4	300
$MoveWeekPenalty$	SC4	400
$MoveMDPenalty$	SC4	600
$DevicePenalty$	SC5	200
$DeviceSecPenalty$	SC5	20
$LunchPenalty$	SC6	30
$EmptyBeforePenalty$	SC7	15
$EmptyAfterPenalty$	SC7	10
$LastEmptyPenalty$	SC7	15
$FirstEmptyPenalty$	SC7	15
$TimeDepPenalty$	SC8	3000
$ChangePenalty$	SC9	3000

found in table 25. These experiments were run with a time limit of 15 minutes. This time limit is applied because the final algorithm must not take hours to complete its task. Considering that the PC used for these experiments is probably quite a bit faster than those of the end users, 15 minutes is the absolute maximum time the algorithm can take. This limit will be set to more strict values later. For problem instance 1, 0.97, 0.95 and 0.90 are close in average penalty. However, 0.90 results in a way lower average runtime. For instance 2, once again the huge difference in runtime overshadows the increase in average penalty. For instance 3, 0.90 and 0.85 show a clear advantage in both average penalty and average runtime. Therefore, we select a $CoolingRate$ of 0.90 for further use.

We now look at the $InitialTemp$ and $ReheatTemp$ parameters. We decided to give these parameters an equal value during this phase of the experiments. The results of these experiments can be found in table 26. With the new value for $CoolingRate$, the runtime has gone down significantly. Therefore we use a time limit of 10 minutes from here on. However, this time limit was never hit during the remainder of the experiments in this subsection. A value of 750 for $InitialTemp$ and $ReheatTemp$ is clearly the best here, resulting in the best average and a competitive runtime for all three instances.

The next parameter we look at is $MainLoopCutoff$. One would expect that a higher value of this parameter gives a better average penalty, at expense of a higher runtime. However, the results, shown in table 27, do not show this. Values of 5000 and 10000 result in close penalty averages, but 20000 is always worse than 10000. We keep this parameter at 10000 for now, but will experiment with a larger range of possible values in the later experiments.

$\#Restarts$, $\#Reheats$ and $\#ReheatsGlobal$ are the next parameters to be checked. The test results can be found in table 28. The set with values 2, 2 and 3 is the only one that results in a competitive penalty average for every instance. Once again, one would expect that only increasing values would give a better average penalty at the expense of a higher runtime. However, there are multiple sets of values with that have one of the values increased from 2, 2 and 3 and leave the other two values untouched, yet they result in worse average penalties. We will try the larger values for $\#Reheats$ and $\#ReheatsGlobal$ again in the later experiments for these parameters. $\#Restarts$ will be set to ∞, with a stricter time limit.

The last tested parameter is $PreprocessTimeDeps$. This parameter indicates whether or not the time dependencies are considered during preprocessing. If they are, possible start periods for meetings are removed if they will always result in a time dependency violation. Disabling this preprocessing

results in more freedom to move meetings around, but in the end only the viable start periods are chosen, because violating a time dependency yields a large penalty. Table 29 shows the results for turning on or off this preprocessing. Having the preprocessing on gives a better result for all three instances.

Table 2 gives an overview of the initial and new values of the Simulated Annealing parameters. The new values will be used in the experiments in the next section, to find a good distribution of the operator probabilities.

Table 2: The initial and new values of the Simulated Annealing parameters. The new values are used during the operator probabilities experiments.

Parameter name	Initial value	New value
InitialTemp	1500	750
ReheatTemp	1500	750
CoolingRate	0.95	0.90
InnerLoopIterations	10000	10000
MainLoopCutoff	10000	10000
#Restarts	2	2
#Reheats	2	2
#ReheatsGlobal	3	3
PreprocessTimeDeps	Yes	Yes

5.4 Neighborhood Operators' Probabilities Experiments

Every iteration of the inner loop of our Simulated Annealing algorithm (see algorithm 1) one neighborhood operator is selected at random and applied to the solution. Not all operators are chosen with an equal probability. For example, the *Change Day Rooms* operator is a lot more powerful than the *Move Room* operator, but it is also a lot slower. *Change Day Rooms* moves meetings to the best combination of rooms in some order, and applying it once gives a large chance of improving the solution's penalty value (unless the room assignment is already very good on the selected day). *Move Room*, on the other hand, relies on trying a lot of random room moves. The majority of those moves will not yield an improvement of the solution, but some of them will. Therefore, it makes sense to give the *Move Room* operator a higher probability of being chosen than the *Change Day Rooms* operator.

Some operators have multiple variants. The *Insert*, *Move Period*, *Move Best Rooms* and *Move Period Best Rooms* operators all have two variants:

one where the possible start periods of a meeting are considered in a random order, and one where they are considered in order of ascending penalty value (for the SC4 and SC6 soft constraints). *Change Day Rooms* has three variants: one where meetings are placed back in the schedule in random order, one where they are considered in order of descending number of rooms, and one where they are considered in order of descending duration.

We assign a weight w_o to each operator variant o. The probability with which variant o is chosen is then $Pr(o) = w_o / \sum_{o' \in O} w_{o'}$, where O is the set of all operator variants. We are looking to find a good assignment of the operators' weights. In this section the experiments to find this assignment are discussed. The results of the experiments can be found in appendix B.

Table 31 shows the weights that were used in previous experiments and the initial weights that will be used in the next two experiments. The difference is that all variants of *Change Day Rooms* are set to be (almost) equal, keeping the total at 800. The previously used weight were set rather arbitrarily during the development of the algorithm and we chose to give all variants of each operator an equal probability to start with.

Before doing any experiments, we looked at the statistics of the operators from the previous experiments (in section 5.3) and we found that the *Remove* operator always came up with a worse schedule, and that the resulting schedule was (almost) never accepted. Therefore, some experiments were run with the weight of *Remove* set to 0. The result are shown in table 32. There is no significant difference in average penalty and the average runtime is lower. Because the *Remove* operator does not seem to add anything, we decided to not use it from here on.

Of course, if all weights are multiplied by the same factor, all probabilities stay the same. Therefore we decided to find a good value for the weights of the variants of *Insert* and keep its total weight fixed after that. The results for the different weights for the two variants can be found in table 33. At this point the weights of the two variants are kept equal. Weights of 4 and 5 are among the best for every instance. We select weights of 4 for use in further experiments, because they have a very slight edge over weights of 5.

Table 34 gives an overview of the new weights. It also shows the weight ranges that we will search for each of the operator variants. These ranges start at 0 and end at two times the current value. It is important to note that the sum of the weights of the two *Insert* variants will be kept at 8. This means there will be no symmetric weight sets, where all weights are multiplied by the same factor.

Selecting a number of possible values for each weight and then trying all possible combinations will result in way too many weight sets to test. It is also not a good idea to test multiple possible values for one or a couple of

weights at a time. This way covariance between two weights may not be exploited properly. Therefore 24 weight sets are generated using a *Halton sequence* [WLH97]. A Halton sequence is a low-discrepancy sequence, which means that its values are spread evenly across the search space, without leaving large gaps in the search space. Bellio et al. [BCD16] successfully use a *Hammersley point set*, which is closely related to a Halton sequence, to generate parameter sets.

A Halton sequence uses a prime number for each dimension of the search space to generate values. We use a leaped Halton sequence, which means we take only every Lth value of the sequence, where L is a prime larger than all primes used to generate values for each dimension. This is necessary to make sure the values are well spread when taking a low amount of values compared to the highest prime used. For example, if the prime 19 is used (which is necessary if there are 8 dimensions) and only 9 values from the normal Halton sequence are taken, that will only spread about half of the space in the associated dimension.

We tested the 24 sets of weights that were generated using the Halton sequence. The Simulated Annealing parameters were set to the new values in table 2 and the time limit was set to 5 minutes. We choose this time limit because we think, on the relatively fast hardware that was used, it should take no longer to generate a schedule. The end user might be using significantly slower hardware and it is undesirable that they have to wait for an hour to get a good schedule. Each test was repeated 9 times and once again we ran 3 tests in parallel on the same PC.

The results for each of the three problem instances are shown in tables 35 to 37. For each weight set, we calculate how far its average is above the best found average, expressed in a percentage. All these values are accumulated in table 3, where the average percentage and maximum percentage are shown for each set. For use in further experiments, we select the set with the lowest maximum percentage, which is set #03.

The weights of set #03 are shown in table 38. It also shows new search ranges, which start at 0.7 times the value of the weight in set #03 and end at 1.3 times the weight in set #03. The search is refined to this space and another 12 weight sets are generated using the Halton sequence. Together with set #03, these 12 sets are tested in the same way the first 25 sets were tested.

The results of these experiments for each problem instance can be found in tables 39 to 41 and the overview of the results in table 4. Weight set #31 had both the lowest average and maximum percentage and its weights are therefore used for all further experiments. The weights and associated probabilities are shown in table 5.

Table 3: The aggregate results of running the algorithm for all three instances, with different sets of weights for the neighborhood operators. Numbers in bold face indicate the best result for that statistic. For each weight set, the results for each of the three instances are shown, expressed as the percentage the weight set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those three percentages, and the *% above lowest, max* column shows the highest percentage. More elaborate results for each instance can be found in tables 35 to 37 in appendix B.

Weight set	% above lowest, instance 1	% above lowest, instance 2	% above lowest, instance 3	% above lowest, avg.	% above lowest, max
#01	1.15%	1.57%	0.46%	1.06%	1.57%
#02	2.73%	6.68%	0.99%	3.46%	6.68%
#03	0.52%	0.99%	0.99%	0.83%	**0.99%**
#04	0.82%	2.33%	**0.00%**	1.05%	2.33%
#05	0.98%	4.97%	4.22%	3.39%	4.97%
#06	0.79%	2.12%	6.60%	3.17%	6.60%
#07	1.46%	1.70%	6.20%	3.12%	6.20%
#08	**0.00%**	5.87%	1.43%	2.43%	5.87%
#09	0.51%	3.31%	0.01%	1.28%	3.31%
#10	1.04%	5.79%	1.15%	2.66%	5.79%
#11	0.48%	5.50%	0.39%	2.12%	5.50%
#12	0.28%	**0.00%**	1.64%	**0.64%**	1.64%
#13	1.52%	6.60%	0.65%	2.92%	6.60%
#14	0.03%	6.65%	0.96%	2.55%	6.65%
#15	0.27%	1.80%	0.16%	0.75%	1.80%
#16	1.48%	3.49%	6.51%	3.83%	6.51%
#17	2.16%	1.78%	0.66%	1.53%	2.16%
#18	1.25%	0.29%	1.57%	1.04%	1.57%
#19	1.29%	21.35%	1.77%	8.14%	21.35%
#20	0.71%	2.30%	0.91%	1.31%	2.30%
#21	0.51%	3.54%	1.12%	1.72%	3.54%
#22	0.28%	0.99%	2.28%	1.18%	2.28%
#23	1.80%	3.61%	0.76%	2.05%	3.61%
#24	0.08%	8.40%	1.56%	3.35%	8.40%
#25	0.10%	1.48%	1.77%	1.12%	1.77%

Table 4: The aggregate results of running the algorithm for all three instances, with the second group of weight sets for the neighborhood operators. Numbers in bold face indicate the best result for that statistic. For each weight set, the results for each of the three instances are shown, expressed as the percentage the weight set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those three percentages, and the *% above lowest, max* column shows the highest percentage. More elaborate results for each instance can be found in tables 39 to 41 in appendix B.

Weight set	% above lowest, instance 1	% above lowest, instance 2	% above lowest, instance 3	% above lowest, avg.	% above lowest, max
#03	0.85%	4.39%	5.36%	3.53%	5.36%
#26	2.27%	5.65%	0.37%	2.76%	5.65%
#27	0.61%	7.53%	0.30%	2.81%	7.53%
#28	0.43%	2.73%	**0.00%**	1.05%	2.73%
#29	1.95%	7.02%	0.36%	3.11%	7.02%
#30	1.59%	3.25%	0.46%	1.77%	3.25%
#31	1.20%	**0.00%**	0.69%	**0.63%**	**1.20%**
#32	0.23%	2.89%	0.40%	1.18%	2.89%
#33	1.54%	2.25%	0.21%	1.33%	2.25%
#34	0.72%	5.71%	1.55%	2.66%	5.71%
#35	0.39%	4.43%	0.50%	1.78%	4.43%
#36	**0.00%**	3.13%	6.11%	3.08%	6.11%
#37	1.24%	19.53%	0.18%	6.98%	19.53%

Table 5: The initial and new values of the neighborhood operators' weights and the probabilities induced by the new values. The new values are from weight set #31 and are used during all further experiments.

Operator name (Variant)	Initial weight	New weight	Probability
Insert (Random)	4	7	0.018%
Insert (Sorted)	4	1	0.003%
Remove	4	0	0.000%
Move Period (Random)	4000	6566	16.487%
Move Period (Sorted)	4000	3235	8.123%
Move Room	8000	19268	48.382%
Move Cluster	8000	5293	13.291%
Move Random (Random)	400	279	0.701%
Move Random (Sorted)	400	364	0.914%
Change Day Rooms (Random)	268	280	0.703%
Change Day Rooms (Sort #Rooms)	266	147	0.369%
Change Day Rooms (Sort Duration)	266	17	0.043%
Move Best Rooms (Random)	800	1379	3.463%
Move Best Rooms (Sorted)	800	1689	4.241%
Move Period Best Rooms (Random)	800	733	1.841%
Move Period Best Rooms (Sorted)	800	567	1.424%

5.5 Simulated Annealing Parameters Experiments

With the neighborhood operators' probabilities fixed, we now look at the Simulated Annealing parameters again. We did some experiments to find good initial values for the parameters, as described in section 5.3. We now hope to find an even better set of values. The initial values of the parameters can be found in table 43 in appendix C. The table also shows the search range for each parameter for the next experiment. These search ranges are based on the previous experiments in section 5.3.

There are multiple parameters that, when increased, should give an improvement of the average penalty, at the expense of a longer runtime. These parameters are $MainLoopCutoff$, $\#Restarts$, $\#Reheats$ and $\#ReheatsGlobal$. Because we do not want the algorithm to run for longer than 5 minutes (as explained in section 5.4) and because we need one clear statistic to decide which parameter set is best (the average penalty, in this case), we set a time limit of 5 minutes and set the $\#Restarts$ parameter to ∞. This way, increasing any of the $MainLoopCutoff$, $\#Reheats$ or $\#ReheatsGlobal$ parameters, will result in a decreased number of restarts within the time limit as a trade-off. This means we can make a fair comparison of the average penalties for all parameter sets.

Once again, we start with testing the initial parameter set and 24 parameter sets generated using a leaped Halton sequence. The operators' probabilities were set to the new values shown in table 5. Each test was repeated 9 times and 3 tests were run in parallel on the same PC.

The results for the three problem instances are shown in tables 44 to 46. Once again, we calculated the difference between each set's average penalty and the best average penalty for the instance, expressed in a percentage of the best average. The results for all instances are shown in table 6. The initial set of parameters (set #01) turns out to have the best average and maximum percentage.

The search ranges are reduced to the intervals shown in table 47 and 12 new parameter sets are generated using a leaped Halton sequence. These 12 sets are tested together with set #01 in the same way the first 25 sets were tested. The results per instance are shown in tables 48 to 50 and table 7 shows an overview of the results. These parameter sets produce the best average and minimum penalty for each problem instance of all configurations that were run 9 times. Parameter set #34 performs the best and is therefore used in all further experiments. The values of set #34 are shown in table 8.

Table 6: The aggregate results of running the algorithm for all three instances, with different sets of parameters for the Simulated Annealing algorithm. Numbers in bold face indicate the best result for that statistic. For each parameter set, the results for each of the three instances are shown, expressed as the percentage the parameter set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those three percentages, and the *% above lowest, max* column shows the highest percentage. More elaborate results for each instance can be found in tables 44 to 46 in appendix C.

Parameter set	% above lowest, instance 1	% above lowest, instance 2	% above lowest, instance 3	% above lowest, avg.	% above lowest, max
#01	0.41%	0.50%	0.89%	**0.60%**	**0.89%**
#02	1.48%	**0.00%**	0.90%	0.79%	1.48%
#03	1.74%	1.44%	0.40%	1.19%	1.74%
#04	0.99%	1.04%	**0.00%**	0.68%	1.04%
#05	0.93%	6.30%	1.87%	3.03%	6.30%
#06	0.48%	4.31%	0.49%	1.76%	4.31%
#07	0.68%	17.45%	0.81%	6.31%	17.45%
#08	1.02%	7.72%	0.87%	3.20%	7.72%
#09	0.91%	1.43%	0.92%	1.09%	1.43%
#10	1.80%	1.08%	1.22%	1.37%	1.80%
#11	1.76%	15.34%	0.61%	5.90%	15.34%
#12	0.65%	1.25%	0.54%	0.81%	1.25%
#13	1.68%	18.40%	2.02%	7.37%	18.40%
#14	1.36%	20.16%	1.75%	7.76%	20.16%
#15	1.15%	1.04%	1.23%	1.14%	1.23%
#16	0.67%	1.35%	2.24%	1.42%	2.24%
#17	0.31%	2.38%	0.80%	1.17%	2.38%
#18	1.91%	4.85%	0.61%	2.45%	4.85%
#19	2.10%	6.86%	0.88%	3.28%	6.86%
#20	0.30%	4.22%	0.75%	1.76%	4.22%
#21	1.10%	7.72%	10.56%	6.46%	10.56%
#22	**0.00%**	19.80%	5.25%	8.35%	19.80%
#23	1.68%	17.45%	1.15%	6.76%	17.45%
#24	0.77%	2.80%	1.32%	1.63%	2.80%
#25	0.32%	6.16%	0.44%	2.31%	6.16%

Table 7: The aggregate results of running the algorithm for all three instances, with the second group of parameter sets for the Simulated Annealing algorithm. Numbers in bold face indicate the best result for that statistic. For each parameter set, the results for each of the three instances are shown, expressed as the percentage the parameter set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those three percentages, and the *% above lowest, max* column shows the highest percentage. More elaborate results for each instance can be found in tables 48 to 50 in appendix C.

Parameter set	% above lowest, instance 1	% above lowest, instance 2	% above lowest, instance 3	% above lowest, avg.	% above lowest, max
#01	0.96%	18.69%	1.03%	6.89%	18.69%
#26	0.47%	19.65%	1.45%	7.19%	19.65%
#27	0.84%	1.63%	1.07%	1.18%	1.63%
#28	0.85%	18.17%	0.80%	6.60%	18.17%
#29	**0.00%**	3.65%	5.06%	2.90%	5.06%
#30	1.65%	1.15%	0.46%	1.09%	1.65%
#31	0.69%	3.23%	**0.00%**	1.31%	3.23%
#32	0.73%	3.42%	0.80%	1.65%	3.42%
#33	1.02%	19.49%	0.60%	7.04%	19.49%
#34	1.24%	**0.00%**	0.21%	**0.48%**	**1.24%**
#35	0.39%	5.17%	1.25%	2.27%	5.17%
#36	0.64%	4.05%	0.62%	1.77%	4.05%
#37	0.96%	2.70%	0.37%	1.34%	2.70%

Table 8: The initial and new values of the Simulated Annealing parameters. The new values are from parameter set #34 and are used during all further experiments. There is no value for *#Restarts*, because it was set to ∞ and the tests were stopped using a time limit instead.

Parameter name	Initial value	New value
InitialTemp	750	660
ReheatTemp	750	740
CoolingRate	0.90	0.917
InnerLoopIterations	10000	10000
MainLoopCutoff	10000	21461
#Restarts	2	-
#Reheats	2	2
#ReheatsGlobal	3	3
PreprocessTimeDeps	Yes	Yes

5.6 Initial Greedy Solutions Experiments

As described in section 4.4, we designed a greedy algorithm to create an initial timetable to start the Simulated Annealing algorithm with. In this section we describe the experiments that were performed to find out their effect on the initial and final solutions' penalty values.

Once again, a time limit of 5 minutes was used. The runtime of the greedy algorithm is not included in this limit. Although the greedy algorithm is not optimized for fast performance, it still never took more than 5 seconds to run during the experiments (depending on the settings used). An extra runtime of 5 seconds is not a problem when the algorithm is being used by the end user. Therefore, we decided to not optimize the greedy algorithm and just leave it out of the time limit. We are certain that optimization of the algorithm would lead to a runtime of less than a second for all settings.

We tried all combinations of statistics to select a meeting and to select the meeting's start period/room(s), which are described in section 4.4, and compared them to running the algorithm with an empty initial schedule. The results of the experiments can be found in table 52 in appendix D.

If just comparing the result of the greedy algorithms without running the Simulated Annealing algorithm, the *available positions/penalty* settings perform best. They get the best result for instance 1 and 2 and are close to the best result for instance 3.

However, this does not translate to a good penalty value after running the Simulated Annealing algorithm. For instance 1, the penalty average is very

close to starting with an empty schedule, but for the other two instances, the result is far off. Therefore, it seems best to stick to an empty schedule as initial solution. An overview of the results can be found in table 9.

Table 9: The aggregate results of running the algorithm for all three instances, with different settings of the initial greedy algorithm. The different settings refer to the measure for selecting and scheduling meetings respectively, as shown in section 4.4. Numbers in bold face indicate the best result for that statistic. For each set of settings, the results for each of the three instances are shown, expressed as the percentage the set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those three percentages, and the *% above lowest, max* column shows the highest percentage. More elaborate results for each instance can be found in table 52 in appendix D.

Greedy algorithm settings	% above lowest, instance 1	% above lowest, instance 2	% above lowest, instance 3	% above lowest, avg.	% above lowest, max
-	0.00%	0.00%	0.00%	**0.00%**	**0.00%**
AP/Pen.	1.48%	10.50%	31.69%	14.56%	31.69%
AP/TAP	2.27%	14.35%	60.23%	25.61%	60.23%
SL/Pen.	1.82%	10.17%	40.08%	17.36%	40.08%
SL/TAP	69.08%	16.01%	47.56%	44.22%	69.08%
RL/Pen.	0.69%	14.73%	28.12%	14.52%	28.12%
RL/TAP	74.56%	14.25%	55.36%	48.06%	74.56%

A possible reason for the bad performance of the Simulated Annealing algorithm after using the greedy algorithm could be that some meetings, notably multi-day ones, are very hard to move in a full schedule. Therefore, if they are in a bad position, that cannot be fixed anymore. When starting from scratch and slowly inserting meetings while executing other neighborhood operators in between, simple meetings that are inserted early can force a multi-day meeting towards a good position while there is still room to move it. To get this effect in the greedy algorithm, it would be required to optimize the schedule between insertions, which would defeat the purpose of the greedy algorithm.

Another factor is that the Simulated Annealing parameters are optimized for starting with an empty solution. Optimizing them for starting with a schedule created by the greedy algorithm might improve the performance. Unfortunately, there was no time to run experiments for this.

We also tried to run the algorithm with lower time limits. This might

show an advantage for starting with a non-empty schedule, because there is already a somewhat reasonable schedule to start with. The results of running the experiments with time limits of 5, 4, 3, 2 and 1 minutes are shown in table 53.

Once again, starting with an empty timetable is the best for any time limit. It might not be the best for even lower time limits (it certainly is not for a time limit of 0 minutes), but we do not think it is necessary to go below 1 minute. The end user will probably not mind waiting for one or a few minutes to get a good timetable and an even lower time limit is not worth the possible loss in solution quality.

These results show something interesting though: the average penalties for running the algorithm with an empty initial solution do not seem to go up much when the time limit is set to lower values. More extensive results are shown in table 10. For each instance, we compared the time limits to the time limit of 5 minutes, using an unpaired t-test. A significance level of 5% was used.

As can be seen in the table, time limits of 1, 2 and 4 minutes perform significantly worse for instance 1 than the time limit of 5 minutes. The 3 minutes time limit also has a worse average, but the difference is not statistically significant. For problem instance 2, the 5 minutes time limit did not perform best, but there are no significantly better or worse time limits here. For instance 3, all time limits are worse than 5 minutes, but only the 2 minutes time limit is significantly worse.

Because the time limits of 1, 2 and 4 minutes are significantly worse than the 5 minutes time limit and the 3 minutes time limit always has a worse average, we decided to use the limit of 5 minutes for the end user application. We will not actually use a time limit in the application, but rather an iteration limit which corresponds with the time limit of 5 minutes on the PC that was used for running the experiments. This way, the performance of the algorithm does not depend on the PC that it is used.

To verify our claim that multi-day meetings make life hard for the algorithm to improve an initial greedy solution, we ran the algorithm with edited instances. These instances are based on the three previously used instances, but all multi-day meetings have been removed. For instance 1, this meant removing 10 of the 147 meetings, for instance 2, 6 of the 175 meetings were removed, and for instance 3 this meant removing 5 of the 156 meetings. The experiments were run with time limits of 5 and 2 minutes. We used two settings for the initial greedy algorithm: Available positions/Penalty and Room load/Penalty (see section 4.4).

The results of the experiments are shown in table 11. We can see that performances with and without initial greedy algorithm are very close now that

Table 10: The results of running the algorithm for all three problem instances, with a time limit of 1 to 5 minutes. The *Significant difference?* column indicates whether the difference with the 5 minute time limit is statistically significant, according to an unpaired t-test with a significance level of 5%. All experiments were run 9 times, with the *#Restarts* parameter set to ∞.

Instance 1				
Time limit (min)	Penalty avg.	Penalty SD	Penalty min	Significant difference?
5	**32207**	243	**31970**	-
4	32698	639	32190	Yes
3	32370	338	32110	No
2	32532	117	32345	Yes
1	32634	310	32305	Yes
Instance 2				
Time limit (min)	Penalty avg.	Penalty SD	Penalty min	Significant difference?
5	43982	2231	41760	-
4	**42926**	2062	**41055**	No
3	44102	2379	41770	No
2	44740	2840	41495	No
1	42997	1789	41410	No
Instance 3				
Time limit (min)	Penalty avg.	Penalty SD	Penalty min	Significant difference?
5	**23965**	408	**23695**	-
4	23971	298	**23695**	No
3	24230	410	23755	No
2	24361	362	23755	Yes
1	24244	181	23980	No

the multi-day meetings are gone, so our claim seems correct. We performed t-tests for all instance/time limit combinations to compare both greedy algorithm settings sets to not using a greedy algorithm, and none of the differences were significant with a significance level of 5%.

Running an initial greedy algorithm has no clear advantage over starting with an empty solution for both time limits of 2 and 5 minutes without any multi-day meetings. Therefore, we see no reason to try and adapt the greedy algorithm to give better results with multi-day meetings, because it is unlikely that will give an advantage over just running Simulated Annealing with an empty initial schedule. However, in section 5.11 we will attempt to make the algorithm better at moving multi-day meetings, which is also helpful when the end user provides the algorithm with an initial solution.

5.7 Statistics of the three used instances

In this section we give an overview of some statistics of the three instances in the experiments and of the best found solution for each instance, using the selected parameters. The overview is shown in table 12. For instance 2, two meetings remain unscheduled in the best found timetable, probably because it is really hard or even impossible to schedule all meetings.

5.8 Comparison to Hand-Made Timetables

In this section we compare the timetables our algorithm produces for the first two blocks of the 2017–2018 academic year to the hand-made ones for those blocks. The hand-made schedules were made by the project manager, who consulted with department lab coordinators and course coordinators. The same request information was used for the generated timetables. Table 13 shows that all of the 9 runs for each instance resulted in better penalty values than the penalty for the hand-made schedule.

Tables 14 to 15 show a more in-depth comparison of the hand-made schedules and the best found schedules by the algorithm for both blocks. All but one statistic are in favor of the algorithm's best found solution. The hand-made schedule for block 2 is better than the algorithm's schedule at one point: it has no moved (simple) meetings, whereas the algorithm's timetable has one. However, it sacrifices the quality of other parts of the timetable.

Table 11: The results of running the algorithm for all three edited problem instances. These instances are based on the previously used instances, but all multi-day meetings have been removed. The experiments were run for different settings of the initial greedy algorithm and for time limits of 5 and 2 minutes. With a significance level of 5% none of the results with an initial greedy solution were significantly different from the results with an empty solution for the same instance and time limit. All experiments were run 9 times, with the *#Restarts* parameter set to ∞.

Instance 1 (without multi-day meetings)					
Greedy algorithm settings	Time limit (min)	Initial penalty avg.	Penalty avg.	Penalty SD	Penalty min
-	5	822000	24351	62	**24260**
AP/Pen.	5	**37724**	**24332**	53	**24260**
RL/Pen.	5	44833	24362	68	24270
-	2	822000	**24391**	101	24280
AP/Pen.	2	**37653**	24458	129	**24260**
RL/Pen.	2	44835	24399	104	24325
Instance 2 (without multi-day meetings)					
Greedy algorithm settings	Time limit (min)	Initial penalty avg.	Penalty avg.	Penalty SD	Penalty min
-	5	1014000	20728	49	20635
AP/Pen.	5	29679	**20712**	52	**20620**
RL/Pen.	5	**28504**	20731	47	20665
-	2	1014000	20804	106	**20650**
AP/Pen.	2	29580	20809	125	**20650**
RL/Pen.	2	**29404**	**20755**	76	20665
Instance 3 (without multi-day meetings)					
Greedy algorithm settings	Time limit (min)	Initial penalty avg.	Penalty avg.	Penalty SD	Penalty min
-	5	906000	**16110**	3	**16105**
AP/Pen.	5	**27119**	16117	20	**16105**
RL/Pen.	5	27233	16144	66	**16105**
-	2	906000	**16207**	122	**16105**
AP/Pen.	2	27183	16229	120	**16105**
RL/Pen.	2	**26991**	16226	139	**16105**

Table 12: Statistics of the three instances used in the experiments. The top part of the table shows the fixed instance statistics. The bottom part shows the statistics of the best found solution, using the selected parameters (see table 5 and table 8), an empty initial solution and a time limit of 5 minutes. For each instance, the algorithm was run 18 times using these settings.

	Instance 1	Instance 2	Instance 3
Total number of meetings	147	175	156
Number of simple meetings	137	169	151
Number of multi-day meetings	10	6	5
Number of time dependencies	62	112	84
Minimum number of clusters	181	192	173
Unscheduled simple meetings	0	2	0
Unscheduled multi-day meetings	0	0	0
Violated time dependencies	0	0	0
Number of clusters	182	192	176
Floor penalties	12	8	6
Room dependency penalties	6	9	6
Moved simple meetings	20	14	18
Moved multi-day meetings	1	0	0
Clusters with device penalty	1	16	2
Setup/clearing penalty total	6330	2940	3695
Total penalty	31970	41410	23695

Table 13: The results of running our algorithm on the real-life instances for the first two quarter of the 2017–2018 academic year. The penalty value of the hand-made schedules for the two quarters is also shown. For each instance, the experiment was run 9 times with a time limit of 5 minutes.

Block	Hand-made penalty	Penalty avg.	Penalty SD	Penalty min	Penalty max
Block 1	18885	12093	207	11825	12495
Block 2	10815	8604	505	8350	9495

Table 14: A comparison of some statistics of the hand-made timetable for block 1 of the 2017–2018 academic year and the best found schedule by our algorithm.

	Hand-made	Algorithm
Total number of meetings	120	120
Number of simple meetings	112	112
Number of multi-day meetings	8	8
Number of time dependencies	74	74
Minimum number of clusters	125	125
Unscheduled simple meetings	0	0
Unscheduled multi-day meetings	0	0
Violated time dependencies	0	0
Number of clusters	135	125
Floor penalties	2	2
Room dependency penalties	26	16
Moved simple meetings	3	2
Moved multi-day meetings	0	0
Clusters with device penalty	11	7
Setup/clearing penalty total	6055	4245
Total penalty	18885	11825

Table 15: A comparison of some statistics of the hand-made timetable for block 2 of the 2017–2018 academic year and the best found schedule by our algorithm.

	Hand-made	Algorithm
Total number of meetings	97	97
Number of simple meetings	72	72
Number of multi-day meetings	25	25
Number of time dependencies	57	57
Minimum number of clusters	108	108
Unscheduled simple meetings	0	0
Unscheduled multi-day meetings	0	0
Violated time dependencies	0	0
Number of clusters	121	117
Floor penalties	12	10
Room dependency penalties	10	7
Moved simple meetings	0	1
Moved multi-day meetings	0	0
Clusters with device penalty	6	5
Setup/clearing penalty total	3415	2010
Total penalty	10815	8350

5.9 Analysis of the Neighborhood Operators' Performances

In this section we give a brief analysis of the performance of each of the neighborhood operators, based on the statistics shown in appendix E. A description of the charts and tables can be found at the beginning of the appendix. The statistics are based on one run of the algorithm, namely best run for problem instance 1, using the selected settings. Instance 1 was chosen because, of the three instances, it has the median best found penalty value. In most charts, only the first 7.2 million iterations are shown, to make the charts more easily understandable.

One thing to make clear before going through the operators is that worsening the solution is not always bad. Even if an operator increases the penalty value more than it decreases it, it can still cause variation in the schedule, which allows other operators to make improvements.

Figures 1 to 4 show the penalty and temperature values throughout the algorithm run. Tables 54 to 55 (at the end of the appendix) show operator statistics summed over the entire run of the algorithm.

***Insert* operator (2 variants, figs. 5 to 8)** The first thing that stands out here, is that the results charts for both variants show a lot of the white background. This happens because the probabilities for the operators to be chosen are so low, that some times they are not called at all during the 10,000 iterations. During the beginning of the run, most insertions succeed and are an improvement. Later on, a lot of insertions fail, probably because the schedule is already very full at that point.

During the later stages of this part of the run, the result is almost always *not possible*, because all meetings have already been inserted. Some operators (*Move Best Rooms* and *Move Period Best Rooms*) can remove meetings from the schedule, which explains why some times an insertion still happens during this stage. The change in penalty caused by the operator is almost exclusively negative (negative is good, in this case), because insertions almost never cause a worsening of the schedule.

***Move Period* operator (2 variants, figs. 9 to 12)** Most of the calls to this operator result in a declined worsening of the schedule. Not very surprising, because this operator is the type of operator that is fast, but needs a high volume of calls to find improvements. We can see that during most of the run, the *Sorted* variant, on average, improves the solution. The *Random* variant does not improve the solution on average throughout the run, but it does during the second half of the first reheat loop iteration.

And one can see that this does not mean the operator only makes worsening changes, but also makes big improvements.

***Move Room* operator (1 variant, figs. 13 to 14)** This operator, compared to *Move Period*, has a lot more calls that result in an accepted change of the schedule (be it an improvement or worsening). However, if we look at the other charts, we can see that the average change in penalty is smaller than *Move Period*. One could say that this operator makes smaller but steadier steps. Although this operator makes a net positive change of the penalty (which is bad, in this case), we suspect it provides other room moving operators (e.g., *Move Best Rooms*) with the possibility to make bigger improvements.

***Move Cluster* operator (1 variant, figs. 15 to 16)** This is much the same story as the previous operator, although the average change in penalty is a bit better, probably because there are fewer destructive moves that, for example, split one cluster into three.

***Move Random* operator (2 variants, figs. 17 to 20)** The *Random* variant really just picks a random new start period/rooms combination for a meeting. Therefore, it should not come as a surprise that most of its changes are declined. The large number of calls that results in an equal penalty probably comes from cases where the starting position of a meeting is the only possible position for that meeting. The net penalty change is slightly above 0 throughout most of the algorithm, but when an improvement is made, the average change is around or below -200 for most of the run. The *Sorted* variant does not really do the name of the operator justice, because the start periods are sorted by increasing penalty. Therefore, the average penalty change is better and there are even more calls that result in an equal penalty, because the best start period was already selected before the operator was applied.

***Change Day Rooms* (3 variants, figs. 21 to 26)** This operator is much slower than all previous operators, but it has the performance to back it up. It has a very high percentage of improvements throughout the first half of every reheat loop iteration. Besides that, its average accepted change in penalty is below 0 throughout the run and the average improvement is larger than the average worsening as well. Of the three variants, the *Sort Duration* variant seems to perform best, although this could also be caused by the low

number of calls that are being made to it. A higher number of calls could result in diminishing returns.

***Move Best Rooms* (2 variants, figs. 27 to 30)** This operator is another "slow" one, although to a lesser extent than *Change Day Rooms*. Once again, this is backed up by an average improvement of the penalty value and a good portion of improvements. By far the largest number of calls results in an equal penalty, probably caused by meetings already being in the best rooms for them. The two variants of this operator show almost no difference in their statistics, because the difference between the variants only comes into play when a meeting has to be moved for the selected meeting to be placed in its best possible rooms.

***Move Period Best Rooms* (2 variants, figs. 31 to 34)** This is an operator that has a very high number of declined changes. Yet the average change in penalty for *Random* variant hovers around 0 and the average for the *Sorted* variant is below 0 almost all the time. This operator would need a lot of calls, like the *Move Period* operator, but because it is one of the slower operators, that would cause a large decrease in calls to the other operators. Yet, we think this operator is important, because, unlike any other operator, it has the ability to move a meeting to a start period even though there's no space for the meeting at that period.

5.10 Parameter Experiments with Additional Instances

So far, the neighborhood operators' probabilities and Simulated Annealing parameters have been tuned using three problem instances. To avoid overfitting, we performed additional experiments with different problem instances. We first tested 23 new weight sets for the neighborhood operators' probabilities on five new generated problem instances. Then, we tested 23 new Simulated Annealing parameter sets on five other new generated problem instances.

We generated 23 new weight sets using a Halton sequence. The set denoted as #01, is the previously found best set (see columns *New weight* and *Probability* in table 5). The other sets were generated using ranges of 0.7 to 1.3 times the weights' values in set #01. The results of these experiments are shown in table 16. Once again, we select the best weight set based on the lowest maximum percentage, in this case set #15.

With weight set #15 for the neighborhood operators' probabilities, we repeated this approach for the Simulated Annealing parameters. Parameter set

Table 16: The aggregate results of running the algorithm for the five instances 7–11, with the new group of weight sets for the neighborhood operators. Numbers in bold face indicate the best result for that statistic. For each weight set, the results for each of the five instances are shown, expressed as the percentage the weight set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those five percentages, and the *% above lowest, max* column shows the highest percentage.

Weight set	% above lowest, instances					% above lowest, avg.	% above lowest, max
	#7	#8	#9	#10	#11		
#01	0.04%	0.24%	**0.00%**	1.42%	2.68%	0.87%	2.68%
#02	0.13%	0.28%	0.06%	1.06%	**0.00%**	**0.30%**	1.06%
#03	0.05%	0.25%	0.01%	0.83%	3.67%	0.96%	3.67%
#04	0.08%	0.45%	0.09%	2.07%	2.95%	1.13%	2.95%
#05	0.07%	0.19%	0.04%	0.03%	1.68%	0.40%	1.68%
#06	0.22%	0.31%	0.04%	0.89%	4.03%	1.10%	4.03%
#07	0.05%	0.48%	0.16%	1.08%	6.12%	1.58%	6.12%
#08	0.07%	0.17%	0.02%	0.25%	3.18%	0.74%	3.18%
#09	0.08%	0.44%	0.03%	0.96%	3.09%	0.92%	3.09%
#10	0.05%	0.46%	0.04%	1.99%	4.08%	1.32%	4.08%
#11	0.09%	0.22%	0.01%	2.05%	3.14%	1.10%	3.14%
#12	**0.00%**	0.21%	0.03%	1.04%	2.41%	0.74%	2.41%
#13	0.03%	0.25%	0.01%	0.65%	4.13%	1.02%	4.13%
#14	0.06%	0.84%	0.02%	0.40%	3.87%	1.04%	3.87%
#15	0.05%	0.44%	0.02%	0.98%	0.50%	0.40%	**0.98%**
#16	0.10%	0.42%	0.07%	0.63%	2.26%	0.70%	2.26%
#17	0.09%	0.52%	0.05%	0.03%	3.67%	0.87%	3.67%
#18	0.07%	**0.00%**	0.03%	0.60%	4.34%	1.01%	4.34%
#19	0.09%	0.25%	0.08%	1.45%	3.37%	1.05%	3.37%
#20	0.05%	0.16%	0.00%	1.22%	5.13%	1.31%	5.13%
#21	0.05%	0.23%	0.11%	**0.00%**	2.40%	0.56%	2.40%
#22	0.05%	0.29%	0.07%	0.68%	4.48%	1.11%	4.48%
#23	0.05%	0.23%	0.05%	0.18%	5.67%	1.23%	5.67%
#24	0.03%	0.31%	0.05%	0.83%	3.43%	0.93%	3.43%

#01 is the previously found best set (see column *New value* in table 8). The other 23 sets were generated using the Halton sequence. For each parameter, we used the same range size as shown in table 47, but centered the range around the value in set #01. An exception to this is $MainLoopCutoff$, for which a range of $[16000, 27000]$ was used. The results are shown in table 17. We select the best parameter set based on the lowest maximum percentage, which is set #02.

With these additional tests using ten new instances, overfitting should be less of an issue. Weight set #15 and parameter set #02 were selected and parameter set #02 will be used in the final version of the algorithm that is included with the GUI application. The final weight set is selected in section 5.11, but is partially the same as weight set #15. Tables 18 to 19 show an overview of the previous and new neighborhood operators' probabilities and Simulated Annealing parameters.

5.11 Moving Multi-Day Meetings

In section 5.6 we found that moving multi-day meetings is a problem when not starting from an empty timetable. The solution in that section was to not use the greedy initial algorithm, but just stick with a empty initial timetable. However, if the end user provides a timetable to start from, the issue might still pose a problem.

To fix this issue, we edited two operators and created an additional one. Firstly, we edited the *Move Best Rooms* and *Move Period Best Rooms* operators to also work for multi-day meetings. This means multi-day meetings can be moved much more easily. Secondly, the *Insert Best Rooms* operator was created. See section 4.5 for a description of the operator. This operator makes sure it is possible to insert meetings into the schedule even if there is little space.

Initial tests indicated that these changes seem to fix the problem. Before doing some tests to show the effect, we first ran some experiments to tune some of the operator weights, namely for *Insert*, *Move Best Rooms*, *Move Period Best Rooms* and *Insert Best Rooms*. Once again we generated 12 sets using a Halton sequence. For the *Move Best Rooms* and *Move Period Best Rooms* operators we used ranges from 0.7 to 1.3 times the previously found weight. For *Insert* and *Insert Best Rooms* we set ranges of 0 to 8 for all variants.

Weight set #01 has the weights shown in table 18, with the weights for the variants of *Insert Best Rooms* set to 4. We used five (randomly selected) instances from the instances 7–16 that were used in section 5.10. The results can be found in table 20. Weight set #13 was selected as the final weight

Table 17: The aggregate results of running the algorithm for the five instances 12–16, with the new group of parameter sets for the Simulated Annealing algorithm. Numbers in bold face indicate the best result for that statistic. For each parameter set, the results for each of the five instances are shown, expressed as the percentage the parameter set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those five percentages, and the *% above lowest, max* column shows the highest percentage.

Parameter set	% above lowest, instances #12	#13	#14	#15	#16	% above lowest, avg.	% above lowest, max
#01	0.09%	1.88%	5.05%	0.56%	4.66%	2.45%	5.05%
#02	0.05%	0.73%	1.57%	0.71%	1.71%	0.95%	**1.71%**
#03	0.07%	**0.00%**	0.75%	0.59%	3.66%	1.01%	3.66%
#04	0.05%	1.08%	4.03%	0.16%	1.88%	1.44%	4.03%
#05	0.05%	0.97%	4.36%	0.35%	2.34%	1.62%	4.36%
#06	0.05%	0.68%	1.43%	0.13%	3.45%	1.15%	3.45%
#07	0.06%	1.90%	1.64%	**0.00%**	3.70%	1.46%	3.70%
#08	0.06%	0.32%	3.30%	0.29%	2.41%	1.28%	3.30%
#09	0.05%	0.37%	2.08%	0.48%	3.44%	1.28%	3.44%
#10	**0.00%**	3.09%	3.97%	0.70%	1.81%	1.91%	3.97%
#11	0.05%	1.57%	2.10%	0.13%	1.83%	1.14%	2.10%
#12	0.06%	0.37%	2.21%	0.43%	3.44%	1.30%	3.44%
#13	0.05%	2.43%	3.95%	0.03%	3.93%	2.08%	3.95%
#14	0.05%	1.21%	0.94%	0.40%	7.03%	1.93%	7.03%
#15	0.04%	1.81%	2.71%	0.13%	**0.00%**	0.94%	2.71%
#16	0.04%	1.60%	3.26%	0.50%	2.00%	1.48%	3.26%
#17	0.09%	0.95%	**0.00%**	0.86%	1.76%	**0.73%**	1.76%
#18	0.05%	0.84%	4.93%	0.11%	5.68%	2.32%	5.68%
#19	0.10%	2.51%	4.87%	0.32%	4.62%	2.48%	4.87%
#20	0.03%	1.62%	1.02%	0.31%	2.73%	1.14%	2.73%
#21	0.06%	1.93%	2.55%	0.35%	5.16%	2.01%	5.16%
#22	0.06%	0.89%	2.60%	0.11%	3.01%	1.34%	3.01%
#23	0.06%	2.46%	4.20%	0.59%	4.20%	2.30%	4.20%
#24	0.06%	0.43%	3.48%	0.21%	6.41%	2.12%	6.41%

Table 18: The previous and new values of the neighborhood operators' weights and the probabilities induced by the new values. The new values are from weight set #15.

Operator name (Variant)	Previous weight	New weight	Probability
Insert (Random)	7	7	0.020%
Insert (Sorted)	1	1	0.003%
Remove	0	0	0.000%
Move Period (Random)	6566	5619	16.315%
Move Period (Sorted)	3235	3777	10.967%
Move Room	19268	13654	39.645%
Move Cluster	5293	5302	15.394%
Move Random (Random)	279	335	0.973%
Move Random (Sorted)	364	433	1.257%
Change Day Rooms (Random)	280	321	0.932%
Change Day Rooms (Sort #Rooms)	147	112	0.325%
Change Day Rooms (Sort Duration)	17	20	0.058%
Move Best Rooms (Random)	1379	1688	4.901%
Move Best Rooms (Sorted)	1689	1960	5.691%
Move Period Best Rooms (Random)	733	806	2.340%
Move Period Best Rooms (Sorted)	567	406	1.179%

Table 19: The previous and new values of the Simulated Annealing parameters. The new values are from parameter set #02.

Parameter name	Previous value	New value
$InitialTemp$	660	690
$ReheatTemp$	740	771
$CoolingRate$	0.917	0.922
$MainLoopCutoff$	21461	19134
$\#Reheats$	2	3
$\#ReheatsGlobal$	3	3

set and will be used in further experiments and in the final version of the algorithm. The weights of set #13 are shown in table table 21.

Table 20: The aggregate results of running the algorithm for the five instances, with the new group of weight sets for the neighborhood operators. Numbers in bold face indicate the best result for that statistic. For each weight set, the results for each of the five instances are shown, expressed as the percentage the weight set's average penalty is above the best average penalty. The *% above lowest, avg.* column shows the average of those five percentages, and the *% above lowest, max* column shows the highest percentage.

Weight set	% above lowest, instances					% above lowest, avg.	% above lowest, max
	#8	#10	#11	#13	#16		
#01	0.15%	0.79%	0.48%	0.13%	0.32%	0.38%	0.79%
#02	0.17%	2.53%	1.24%	0.05%	0.26%	0.85%	2.53%
#03	0.09%	0.68%	1.17%	0.03%	**0.00%**	0.40%	1.17%
#04	0.16%	1.37%	1.39%	0.01%	0.57%	0.70%	1.39%
#05	**0.00%**	1.62%	1.27%	0.10%	0.23%	0.64%	1.62%
#06	0.03%	0.48%	0.42%	0.01%	0.23%	0.23%	0.48%
#07	0.19%	1.42%	0.29%	**0.00%**	0.59%	0.50%	1.42%
#08	0.12%	0.91%	1.36%	0.04%	0.12%	0.51%	1.36%
#09	0.30%	0.70%	0.89%	0.20%	0.42%	0.50%	0.89%
#10	0.16%	1.26%	0.23%	0.15%	0.26%	0.41%	1.26%
#11	0.11%	1.04%	0.56%	0.15%	0.81%	0.53%	1.04%
#12	0.26%	1.76%	**0.00%**	0.11%	0.13%	0.45%	1.76%
#13	0.14%	**0.00%**	0.23%	0.11%	0.42%	**0.18%**	**0.42%**

To test the effect of the changes made, we will rerun the first experiment of section 5.6 regarding the use of a greedy initial algorithm. The results of the first run can be found in table 9 and table 52 (appendix D). Using a greedy initial algorithm did not work, because the Simulated Annealing algorithm had trouble moving around multi-day meetings from their position in the initial schedule to their ideal position.

The new results are shown in table 22. It is clear that the changes to the algorithm have a big impact on these results. All averages are now very close to each other and starting with an initial empty solution is not always the best choice anymore.

Table 21: The previous and new values of the neighborhood operators' weights and the probabilities induced by the new values. The new values are from weight set #13.

Operator name (Variant)	Previous weight	New weight	Probability
Insert (Random)	7	2	0.006%
Insert (Sorted)	1	6	0.017%
Remove	0	0	0.000%
Move Period (Random)	5619	5619	16.273%
Move Period (Sorted)	3777	3777	10.938%
Move Room	13654	13654	39.542%
Move Cluster	5302	5302	15.355%
Move Random (Random)	335	335	0.970%
Move Random (Sorted)	433	433	1.254%
Change Day Rooms (Random)	321	321	0.930%
Change Day Rooms (Sort #Rooms)	112	112	0.324%
Change Day Rooms (Sort Duration)	20	20	0.058%
Move Best Rooms (Random)	1688	1776	5.143%
Move Best Rooms (Sorted)	1960	2158	6.250%
Move Period Best Rooms (Random)	806	600	1.738%
Move Period Best Rooms (Sorted)	406	408	1.182%
Insert Best Rooms (Random)	0	3	0.009%
Insert Best Rooms (Sorted)	0	4	0.012%

Table 22: The results of running the algorithm for the three problem instances 1–3, with different settings of the initial greedy algorithm. The different settings refer to the measure for selecting and scheduling meetings respectively, as shown in section 4.4. The hyphen means that an empty initial solution was used. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Instance 1

Greedy algorithm settings	Initial penalty avg.	Penalty avg.	Penalty min	% above lowest avg.
-	882000	**32121**	32025	0.00%
AP/Pen.	**50544**	32124	32045	0.01%
AP/TAP	254391	32157	32085	0.11%
SL/Pen.	59420	32147	**32000**	0.08%
SL/TAP	251615	32139	32045	0.06%
RL/Pen.	61471	32122	32025	0.00%
RL/TAP	268469	32133	32045	0.04%

Instance 2

Greedy algorithm settings	Initial penalty avg.	Penalty avg.	Penalty min	% above lowest avg.
-	1050000	41495	41380	0.10%
AP/Pen.	63715	42318	**40675**	2.09%
AP/TAP	320822	41493	41410	0.10%
SL/Pen.	73947	**41453**	41410	0.00%
SL/TAP	317284	41512	41410	0.14%
RL/Pen.	**63306**	41473	41410	0.05%
RL/TAP	317599	41523	41410	0.17%

Instance 3

Greedy algorithm settings	Initial penalty avg.	Penalty avg.	Penalty min	% above lowest avg.
-	936000	23835	**23695**	0.37%
AP/Pen.	43038	23773	**23695**	0.11%
AP/TAP	292821	23786	**23695**	0.16%
SL/Pen.	51327	23855	**23695**	0.45%
SL/TAP	299282	23775	**23695**	0.12%
RL/Pen.	**41699**	**23747**	**23695**	0.00%
RL/TAP	300473	23867	**23695**	0.51%

In section 5.12 we look at instances with existing initial schedules with part of the meetings locked. To also test how the new operators work on existing initial schedules, we ran the same experiments without the locks. The results of this are shown in table 23 in the *Avg. w/o locks* column. If we compare these results to the results with an initial empty schedule (*Normal avg.* column), we can see that the differences are small, even though the initial schedules have very high penalty values (*Start penalty* column).

Looking at these two tests it seems that editing the *Move Best Rooms* and *Move Period Best Rooms* operators to include multi-day meetings and creating the *Insert Best Rooms* operator have solved the issue of not being able to move around multi-day meetings, that was found in section 5.6. Even though a greedy initial algorithm will not be used in the end product (because there are no clear advantages over starting with an empty timetable), this is still a useful development if the end user provides the algorithm with an existing initial timetable. The algorithm should now be able to better optimize such an initial schedule.

5.12 Experiments on Optimizing Initial Schedules with Locks

We created three instances to test the algorithm when it has to improve an initial schedule with locked meetings. The instances were generated in the same way the previous instances were created. We then let the algorithm create a schedule for them, but with changed penalty parameters. Parameters like $MovePeriodPenalty$, $MoveWeekPenalty$, $MoveMDPenalty$ and $DevicePenalty$ were decreased to create a non-optimal schedule for the original penalty parameters. After creating these schedules, each meeting's start period was locked with a probability of 10%.

With these initial schedules we ran experiments to optimize them (using the original penalty parameters). The results can be found in table 23. Of course, these optimized schedules can never be as good as the "normal" optimized schedules, because of the locked meetings, which are not in the optimal position. However, the initial schedules are all greatly improved and the resulting schedules come close to the "normal" schedules.

Table 23: The results of running the algorithm with an initial schedule with 10% of the start periods of the meetings locked. Three different instances were tested. The *Normal avg.* column shows the average penalty after optimizing the schedule for the instance with an initial empty schedule (thus without locked meetings). The *Start penalty* column shows the penalty of the initial solution. The *Avg. w/o locks* column shows the average penalty after optimizing this initial schedule, without taking the locks into account. The *Avg. w/ locks* and *SD w/ locks* columns show the average and standard deviation of the penalties after optimizing the initial schedule for the instance with 10% of the start periods locked. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Instance	Normal avg.	Start penalty	Avg. w/o locks	Avg. w/ locks	SD w/ locks
#4	22522	103525	22524	26439	34
#5	26768	104535	26845	32249	61
#6	27619	133060	27145	35991	122

6 GUI application

To allow university staff to use our algorithm efficiently, we designed a GUI application. Screenshots of the application can be found in appendix F. The application can be used by course coordinators and department lab coordinators to enter the requested meetings for each course and after the requests have been entered, the application can be used to automatically generate a timetable using our algorithm and to tweak the generated timetable where necessary.

The application is divided into three tabs. The *Project* tab (fig. 35) allows the user to select the block settings. The block settings determine the number of weeks in the block, the associated calendar week numbers and the periods in the block that are unavailable (i.e., when no (simple) meetings can be held).

The *Requests* tab (fig. 36) shows, for each course, the requested meetings. Courses can be added, edited and removed. Both simple and multi-day meeting requests can be added, edited and removed as well. The requests of a course are displayed in a grid with rows for each week and columns for each period of a week.

Finally, the *Schedule* tab (figs. 37 to 38) shows the complete schedule for the block. At first all meetings are unscheduled. The tab contains three buttons to run our scheduling algorithm. The first button discards the previous schedule and makes the algorithm start from scratch. The second button runs the algorithm starting with the current schedule, trying to improve upon it. The third button does the same as the second one, but gives the option to limit the number of start period and/or room changes (see section 3.6).

Three equal instances of the algorithm with different seeds for the random number generator are always run in parallel, and the best resulting schedule is loaded into the GUI application. The algorithm is run with the selected parameters (see section 5) and with $\#Restarts$ set to ∞. To keep the quality of the resulting schedules equal across different PCs, we did not use a time limit, but rather an iteration limit. We took the average number of iterations per 5 minutes of all 99 runs that were done using the selected parameters and rounded that up to an iteration limit of 17 million.

The *Schedule* tab contains a grid for every week of the block, showing the meetings scheduled in the week, positioned according to their scheduled periods and rooms. Each meeting can be selected and then edited using the edit panel (shown at the bottom of fig. 38). In this panel the start period and rooms for the meeting can be selected. The meeting's start period and/or rooms can also be fixed (see section 3.6). The right side of the panel shows information about the request, like the timeslot and the number of rooms of

the meeting and the devices that are needed.

There is a configuration option to hide the *Schedule* tab. The application can be delivered using this configuration to (for example) the course coordinators, who are not involved with the final schedule, but rather just need to enter the requests for their own course. There is also a button to export the schedule as an HTML file, so it can be sent to people who do not have our GUI application on their PC.

7 Conclusion

Since 2015, the *Victor J. Koningsberger building* has been used by the Departments of *Biology*, *Chemistry* and *Pharmaceutical Sciences*, the *Faculty of Medicine/University Medical Center Utrecht* and the *Faculty of Geosciences* for (part of) their lab sessions. This means these departments and faculties have to work together to create a timetable for the building's lab rooms.

Our project aimed to help the people responsible in two ways: by creating an algorithm to find a good schedule automatically and by creating a GUI application in which course coordinators can request their lab sessions and which the central coordinator can use to import all these request and create a timetable using our algorithm. The GUI application can also be used to edit the timetable and to improve on an existing timetable.

Our algorithm to find a good schedule is based on *Simulated Annealing*. Our version of Simulated Annealing uses both restarting, where the current solution is reset to the best found solution and the temperature is reset, and reheating, where only the temperature is reset. We created 9 different neighborhood operators, some with 2 or 3 slightly different variants. We also created a greedy algorithm which creates an initial timetable, but this had no advantages over starting the Simulated Annealing algorithm with an empty timetable and actually made the resulting timetable quality worse.

Our research was mainly focused on finding the best settings for our algorithm. We had two groups of parameters: the neighborhood operators' probabilities to be chosen during an iteration and the Simulated Annealing parameters (e.g., starting temperatures, cooling rate, cutoff). We tested a total of 60 different Simulated Annealing parameter sets and 73 different sets of neighborhood operators' probabilities, generated using the *Halton sequence*, and we selected the best performing set to be used in the final version of our algorithm that is included with the GUI application.

Before our search for the best algorithm settings, we were already able to find good timetables for the first two quarters of the 2017–2018 academic year using our algorithm. The project manager and department lab coordinators found these timetables better than the hand-made ones that were made for those quarters, and decided to use our timetables instead. More recently, it has also been decided to use our GUI application and algorithm to gather all requests for the third and fourth quarters of the 2017–2018 academic year and to create the timetables.

8 References

[AKM14] E. Aarts, J. Korst and W. Michiels (2014). Simulated annealing. In *Search Methodologies*, Springer US: 265–285.

[ATM09] S. Abdullah, H. Turabieh, B. McCollum and E.K. Burke (2009). An Investigation of a Genetic Algorithm and Sequential Local Search Approach for Curriculum-based Course Timetabling Problems. *Proc. Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2009)*: 727–731.

[BCD13] R. Bellio, S. Ceschia, L. Di Gaspero, A. Schaerf and T. Urli (2013). A simulated annealing approach to the curriculum-based course timetabling problem. *Proc. of the 6th Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA-13)*.

[BCD16] R. Bellio, S. Ceschia, L. Di Gaspero, A. Schaerf and T. Urli (2016). Feature-based tuning of simulated annealing applied to the curriculum-based course timetabling problem. *Computers & Operations Research* 65: 83–92.

[BCR15] A. Bettinelli, V. Cacchiani, R. Roberti and P. Toth (2015). An overview of curriculum-based course timetabling. *TOP* 23.2: 313–349.

[BKA11] A.L. Bolaji, A.T. Khader, M.A. Al-betar and M. Awadallah (2011). Artificial Bee Colony Algorithm for Curriculum-Based Course Timetabling Problem. *The 5th International Conference on Information Technology (ICIT 2011)*.

[BKH15] H. Babaei, J. Karimpour and A. Hadidi (2015). A survey of approaches for university course timetabling problem. *Computers & Industrial Engineering* 86: 43–59.

[DSM07] L. Di Gaspero, B. McCollum and A. Schaerf (2007). *The Second International Timetabling Competition (ITC-2007): Curriculum-based Course Timetabling (Track 3)* (Technical Report QUB/IEEE/Tech/ITC-2007/CurriculumCTT/v1.0/1). Queen's University, Belfast, United Kingdom.

[Kam13] H.C. Kampman (2013). *Timetabling at Utrecht University* (Unpublished master's thesis). Utrecht University, The Netherlands.

[KiM03] K.H. Kim and K.C. Moon (2003). Berth scheduling by simulated annealing. *Transportation Research Part B* 37.6: 541–560.

[KoW97] L. Kocis and W.J. Whiten (1997). Computational Investigations of Low-Discrepancy Sequences. *ACM Transactions on Mathematical Software (TOMS)* 23.2: 266–294.

[KrS13] S. Kristiansen and T.R. Stidsen (2013). *A Comprehensive Study of Educational Timetabling, a Survey* (Report 8.2013). Department of Management Engineering, Technical University of Denmark.

[LHG11] Z. Lü, J.K. Hao and F. Glover (2011). Neighborhood analysis: a case study on curriculum-based course timetabling. *Journal of Heuristics* 17.2: 97–118.

[LuH10] Z. Lü and J.K. Hao (2010). Adaptive Tabu Search for course timetabling. *European Journal of Operational Research* 200.1: 235–244.

[Mül09] T. Müller (2009). ITC-2007 solver description: A hybrid approach. *Annals of Operations Research* 172.1: 429–446.

[NNT11] K. Nguyen, Q. Nguyen, H. Tran, P. Nguyen and N. Tran (2011). Variable Neighborhood Search for a Real-World Curriculum-Based University Timetabling Problem. *2011 Third International Conference on Knowledge and Systems Engineering*: 157–162.

[ScH07] K. Schimmelpfeng and S. Helber (2007). Application of a real-world university-course timetabling model solved by integer programming. *OR Spectrum* 29.4: 783–803.

[TPH14] T. Thepphakorn, P. Pongcharoen and C. Hicks (2014). An ant colony based timetabling tool. *International Journal of Production Economics* 149: 131–144.

[WaH16] J. Wahid and N.M. Hussin (2016). Construction of Initial Solution Population for Curriculum-Based Course Timetabling using Combination of Graph Heuristics. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)* 8.8: 91–95.

[WLH97] T.T. Wong, W.S. Luk and P.A. Heng (1997). Sampling with Hammersley and Halton Points. *Journal of Graphics Tools* 2.2: 9–24.

A Results of Initial Simulated Annealing Parameters Experiments

Table 24: The initial values of the Simulated Annealing parameters. *PreprocessTimeDeps* indicates whether or not time dependencies are considered in preprocessing to eliminate possible start periods that will always lead to a time dependency violation. The other parameters are explained in section 4.2.

Parameter name	Initial value
InitialTemp	1500
ReheatTemp	1500
CoolingRate	0.95
InnerLoopIterations	10000
MainLoopCutoff	10000
#Restarts	2
#Reheats	2
#ReheatsGlobal	3
PreprocessTimeDeps	Yes

Table 25: The results of running the algorithm with different values for the *CoolingRate*. All experiments were run 3 times, with a time limit of 15 minutes for each run. » indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1			
CoolingRate	Penalty avg.	Penalty SD	Time avg. (s)
0.97	32350	223	681
»0.95	32402	328	520
*0.90	32830	589	211
0.85	33790	1594	115
Instance 2			
CoolingRate	Penalty avg.	Penalty SD	Time avg. (s)
0.97	43740	1864	381
»0.95	42415	375	340
*0.90	43277	2231	96
0.85	42820	96	122
Instance 3			
CoolingRate	Penalty avg.	Penalty SD	Time avg. (s)
0.97	27323	5278	716
»0.95	27810	6658	646
*0.90	24157	547	282
0.85	24058	266	203

Table 26: The results of running the algorithm with different values for *InitialTemp* and *ReheatTemp*. All experiments were run 3 times, with a time limit of 10 minutes for each run. » indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1			
InitialTemp / *ReheatTemp*	Penalty avg.	Penalty SD	Time avg. (s)
500	32398	427	174
*750	32297	78	168
1000	33495	1376	181
»1500	32830	589	211
2000	33865	1810	231
Instance 2			
InitialTemp / *ReheatTemp*	Penalty avg.	Penalty SD	Time avg. (s)
500	46130	762	121
*750	42717	51	114
1000	43985	2925	106
»1500	43277	2231	96
2000	43827	1713	123
Instance 3			
InitialTemp / *ReheatTemp*	Penalty avg.	Penalty SD	Time avg. (s)
500	23943	188	223
*750	23880	270	236
1000	24028	304	276
»1500	24157	547	282
2000	24112	362	275

Table 27: The results of running the algorithm with different values for $MainLoopCutoff$. To test with a value of $MainLoopCutoff = 5000$, we lower the value $InnerLoopIterations$ to 5000 as well. The cutoff is only checked after every $InnerLoopIterations$ iterations, and if we do not lower that, the effect of lowering $MainLoopCutoff$ will be less apparent. To make sure the temperature goes down at an equal rate, we set $CoolingRate$ to $\sqrt{0.9} \approx 0.949$. All experiments were run 3 times, with a time limit of 10 minutes for each run. » indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1

MainLoop-Cutoff	*InnerLoop-Iterations*	*Cooling-Rate*	Penalty avg.	Penalty SD	Time avg. (s)
5000	5000	0.949	32513	171	201
»*10000	10000	0.900	32297	78	168
20000	10000	0.900	32400	208	193

Instance 2

MainLoop-Cutoff	*InnerLoop-Iterations*	*Cooling-Rate*	Penalty avg.	Penalty SD	Time avg. (s)
5000	5000	0.949	42607	449	102
»*10000	10000	0.900	42717	51	114
20000	10000	0.900	43262	2269	106

Instance 3

MainLoop-Cutoff	*InnerLoop-Iterations*	*Cooling-Rate*	Penalty avg.	Penalty SD	Time avg. (s)
5000	5000	0.949	23875	159	293
»*10000	10000	0.900	23880	270	236
20000	10000	0.900	24072	26	194

Table 28: The results of running the algorithm with different values for *#Restarts*, *#Reheats* and *#ReheatsGlobal*. All experiments were run 3 times, with a time limit of 10 minutes for each run. » indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1

#Restarts	*#Reheats*	*#Reheats-Global*	Penalty avg.	Penalty SD	Time avg. (s)
5	0	0	32495	311	202
4	1	1	32390	43	119
4	2	3	32502	134	302
3	2	2	32407	410	205
3	2	3	32350	190	267
»*2	2	3	32297	78	168
2	2	5	32315	48	221
1	5	5	32353	243	230

Instance 2

#Restarts	*#Reheats*	*#Reheats-Global*	Penalty avg.	Penalty SD	Time avg. (s)
5	0	0	45568	3420	67
4	1	1	42503	597	83
4	2	3	45882	2682	152
3	2	2	44758	3368	103
3	2	3	47760	5458	105
»*2	2	3	42717	51	114
2	2	5	45072	3060	159
1	5	5	44278	2686	182

Instance 3

#Restarts	*#Reheats*	*#Reheats-Global*	Penalty avg.	Penalty SD	Time avg. (s)
5	0	0	24042	14	201
4	1	1	28505	7526	179
4	2	3	24285	356	327
3	2	2	24018	249	235
3	2	3	23917	252	283
»*2	2	3	23880	270	236
2	2	5	23910	322	426
1	5	5	23833	190	308

Table 29: The results of running the algorithm with different values for *PreprocessTimeDeps*. *PreprocessTimeDeps* indicates whether or not time dependencies are considered in preprocessing to eliminate possible start periods that will always lead to a time dependency violation. All experiments were run 3 times, with a time limit of 10 minutes for each run. « indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1			
PreprocessTimeDeps	Penalty avg.	Penalty SD	Time avg. (s)
Yes*«	32297	78	168
No	32410	320	141
Instance 2			
PreprocessTimeDeps	Penalty avg.	Penalty SD	Time avg. (s)
Yes*«	42717	51	114
No	45697	3076	89
Instance 3			
PreprocessTimeDeps	Penalty avg.	Penalty SD	Time avg. (s)
Yes*«	23880	270	236
No	24013	248	339

Table 30: The new values of the Simulated Annealing parameters. These values are used during the operator probabilities experiments.

Parameter name	New value
InitialTemp	750
ReheatTemp	750
CoolingRate	0.9
InnerLoopIterations	10000
MainLoopCutoff	10000
#Restarts	2
#Reheats	2
#ReheatsGlobal	3
PreprocessTimeDeps	Yes

B Results of Neighborhood Operators' Probabilities Experiments

Table 31: The operator weights used previously (during the initial Simulated Annealing parameters experiments) and the new weights that will be used initially for the experiments to find good weights. The probability for each operator to be chosen in an iteration of the Simulated Annealing algorithm, is given by its weight divided by the sum of all weights. Some operators have multiple variants. For ChangeDayRooms, the variant indicates in which order the meetings are placed back into the schedule. For the other operators, the variant indicates the order in which possible start periods of a meeting are considered.

Operator name (Variant)	Previous weight	Initial weight
Insert (Random)	4	4
Insert (Sorted)	4	4
Remove	4	4
Move Period (Random)	4000	4000
Move Period (Sorted)	4000	4000
Move Room	8000	8000
Move Cluster	8000	8000
Move Random (Random)	400	400
Move Random (Sorted)	400	400
Change Day Rooms (Random)	400	268
Change Day Rooms (Sort #Rooms)	200	266
Change Day Rooms (Sort Duration)	200	266
Move Best Rooms (Random)	800	800
Move Best Rooms (Sorted)	800	800
Move Period Best Rooms (Random)	800	800
Move Period Best Rooms (Sorted)	800	800

Table 32: The results of running the algorithm with different weights for the *Remove* operator. All experiments were run 6 times, with a time limit of 10 minutes for each run. » indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1			
Remove weight	Penalty avg.	Penalty SD	Time avg. (s)
»4	32508	429	252
*0	32569	215	198
Instance 2			
Remove weight	Penalty avg.	Penalty SD	Time avg. (s)
»4	43963	2243	147
*0	43194	1296	93
Instance 3			
Remove weight	Penalty avg.	Penalty SD	Time avg. (s)
»4	24018	242	254
*0	24229	449	233

Table 33: The results of running the algorithm with different weights for the two variants of the *Insert* operator. All experiments were run 6 times, with a time limit of 7.5 minutes for each run. » indicates the initial parameter value, * indicates the value that was selected to be used in further experiments.

Instance 1			
Insert (Random) / *Insert* (Sorted) weights	Penalty avg.	Penalty SD	Time avg. (s)
1	36393	3404	179
2	32907	1427	175
3	32410	314	219
»*4	32569	215	198
5	32531	304	163
6	32392	232	169
8	32449	336	215
Instance 2			
Insert (Random) / *Insert* (Sorted) weights	Penalty avg.	Penalty SD	Time avg. (s)
1	211533	410768	135
2	45048	2434	100
3	44906	2384	121
»*4	43194	1296	93
5	43495	2246	93
6	44938	2360	120
8	44349	3574	116
Instance 3			
Insert (Random) / *Insert* (Sorted) weights	Penalty avg.	Penalty SD	Time avg. (s)
1	328856	470295	179
2	24113	245	238
3	24222	366	266
»*4	24229	449	233
5	24333	386	257
6	24253	452	231
8	24650	805	365

Table 34: The new operator weights that are used as a base for further experiments. The search range for the following experiment is also shown. Each operator's weight will be in the range from 0 to 2 times its current weight. The *Remove* operator will not be used. The sum of the weights of the two variants of *Insert* will always be set to 8, i.e., one weight is chosen in the range $[0, 8]$ and the other weight is set to 8 minus the first weight.

Operator name (Variant)	New weight	Search range
Insert (Random)	4	$[0, 8]$
Insert (Sorted)	4	$[0, 8]$
Remove	0	0
Move Period (Random)	4000	$[0, 8000]$
Move Period (Sorted)	4000	$[0, 8000]$
Move Room	8000	$[0, 16000]$
Move Cluster	8000	$[0, 16000]$
Move Random (Random)	400	$[0, 800]$
Move Random (Sorted)	400	$[0, 800]$
Change Day Rooms (Random)	268	$[0, 533]$
Change Day Rooms (Sort #Rooms)	266	$[0, 533]$
Change Day Rooms (Sort Duration)	266	$[0, 533]$
Move Best Rooms (Random)	800	$[0, 1600]$
Move Best Rooms (Sorted)	800	$[0, 1600]$
Move Period Best Rooms (Random)	800	$[0, 1600]$
Move Period Best Rooms (Sorted)	800	$[0, 1600]$

Table 35: The results of running the algorithm for **problem instance 1**, with different sets of weights for the neighborhood operators. Weight set #01 has the weights as shown in table 34. The other weight sets were generated using the Halton sequence with the ranges in table 34. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run.

Weight set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	32806	773	172	32215	1.15%
#02	33318	1476	159	32250	2.73%
#03	32603	319	177	32220	0.52%
#04	32698	640	139	32160	0.82%
#05	32753	456	167	32235	0.98%
#06	32689	558	254	32085	0.79%
#07	32908	745	131	32290	1.46%
#08	**32433**	248	279	32105	0.00%
#09	32598	837	188	32190	0.51%
#10	32772	499	143	32115	1.04%
#11	32589	272	282	32240	0.48%
#12	32524	349	201	**32070**	0.28%
#13	32926	699	205	32115	1.52%
#14	32442	451	160	32165	0.03%
#15	32522	340	220	32095	0.27%
#16	32913	1476	234	32110	1.48%
#17	33133	2384	138	32200	2.16%
#18	32838	423	182	32190	1.25%
#19	32853	658	122	32225	1.29%
#20	32664	373	268	32150	0.71%
#21	32597	296	204	32190	0.51%
#22	32524	278	241	32235	0.28%
#23	33016	1744	255	32090	1.80%
#24	32461	300	123	32125	0.08%
#25	32467	222	248	32260	0.10%

Table 36: The results of running the algorithm for **problem instance 2**, with different sets of weights for the neighborhood operators. Weight set #01 has the weights as shown in table 34. The other weight sets were generated using the Halton sequence with the ranges in table 34. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run.

Weight set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	43586	2301	105	41425	1.57%
#02	45776	3921	106	41560	6.68%
#03	43334	1913	97	41605	0.99%
#04	43912	3279	87	41480	2.33%
#05	45042	2359	99	42255	4.97%
#06	43821	1883	178	41800	2.12%
#07	43641	2286	73	41520	1.70%
#08	45431	3320	179	41835	5.87%
#09	44332	2795	111	41695	3.31%
#10	45394	3459	75	41605	5.79%
#11	45270	3436	175	41525	5.50%
#12	**42911**	1527	129	**41420**	0.00%
#13	45741	2874	89	41715	6.60%
#14	45766	2398	109	41985	6.65%
#15	43685	2247	117	41850	1.80%
#16	44409	2900	133	41940	3.49%
#17	43673	1885	90	42075	1.78%
#18	43037	1690	108	41830	0.29%
#19	52071	21073	82	41530	21.35%
#20	43897	2004	219	41930	2.30%
#21	44430	2648	110	41540	3.54%
#22	43336	2449	185	41480	0.99%
#23	44458	3189	136	41540	3.61%
#24	46514	3969	73	41605	8.40%
#25	43546	2070	150	41480	1.48%

Table 37: The results of running the algorithm for **problem instance 3**, with different sets of weights for the neighborhood operators. Weight set #01 has the weights as shown in table 34. The other weight sets were generated using the Halton sequence with the ranges in table 34. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run.

Weight set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	23994	214	257	**23695**	0.46%
#02	24119	292	270	23755	0.99%
#03	24121	421	277	23740	0.99%
#04	**23884**	194	169	**23695**	0.00%
#05	24892	2228	192	23755	4.22%
#06	25461	4202	320	**23695**	6.60%
#07	25364	3741	162	**23695**	6.20%
#08	24226	291	313	23965	1.43%
#09	23887	119	270	23755	0.01%
#10	24159	206	162	23820	1.15%
#11	23977	198	310	**23695**	0.39%
#12	24275	451	276	23935	1.64%
#13	24038	332	237	**23695**	0.65%
#14	24113	199	199	23755	0.96%
#15	23922	215	268	**23695**	0.16%
#16	25438	4139	271	**23695**	6.51%
#17	24041	140	226	23865	0.66%
#18	24259	533	275	23790	1.57%
#19	24306	253	189	24050	1.77%
#20	24102	89	311	23935	0.91%
#21	24152	308	262	23790	1.12%
#22	24428	901	300	23755	2.28%
#23	24064	389	280	**23695**	0.76%
#24	24257	465	176	23755	1.56%
#25	24307	437	301	23755	1.77%

Table 38: The operator weights of weight set #03, that are used as a base for the next experiment. The search range for the following experiment is also shown. Each operator's weight will be in the range from 0.7 to 1.3 times its current weight. The sum of the weights of the two variants of *Insert* will always be set to 8, i.e., one weight is chosen in its range and the other weight is set to 8 minus the first weight.

Operator name (Variant)	Set #03 weight	Search range
Insert (Random)	6	[4, 8]
Insert (Sorted)	2	[0, 4]
Remove	0	0
Move Period (Random)	5860	[4102, 7618]
Move Period (Sorted)	2893	[2025, 3761]
Move Room	15767	[11036, 20498]
Move Cluster	4640	[3248, 6032]
Move Random (Random)	237	[165, 309]
Move Random (Sorted)	493	[345, 641]
Change Day Rooms (Random)	375	[262, 488]
Change Day Rooms (Sort #Rooms)	191	[133, 249]
Change Day Rooms (Sort Duration)	22	[15, 29]
Move Best Rooms (Random)	1142	[799, 1485]
Move Best Rooms (Sorted)	1517	[1061, 1973]
Move Period Best Rooms (Random)	900	[630, 1170]
Move Period Best Rooms (Sorted)	635	[444, 826]

Table 39: The results of running the algorithm for **problem instance 1**, with the second group of weight sets for the neighborhood operators. Weight set #03 has the weights as shown in table 38. The other weight sets were generated using the Halton sequence with the ranges in table 38. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run.

Weight set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#03	32546	683	188	32150	0.85%
#26	33004	798	172	32205	2.27%
#27	32469	334	193	32115	0.61%
#28	32413	365	141	32105	0.43%
#29	32903	690	183	32090	1.95%
#30	32784	582	252	32170	1.59%
#31	32659	750	182	32115	1.20%
#32	32348	149	209	32190	0.23%
#33	32769	643	142	32215	1.54%
#34	32507	458	176	32075	0.72%
#35	32400	429	220	32070	0.39%
#36	**32273**	144	227	**32040**	0.00%
#37	32674	623	180	32180	1.24%

Table 40: The results of running the algorithm for **problem instance 2**, with the second group of weight sets for the neighborhood operators. Weight set #03 has the weights as shown in table 38. The other weight sets were generated using the Halton sequence with the ranges in table 38. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run.

Weight set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#03	44550	2245	124	42600	4.39%
#26	45089	3389	123	41625	5.65%
#27	45892	3375	133	42670	7.53%
#28	43842	2402	93	41750	2.73%
#29	45671	2191	93	42690	7.02%
#30	44066	3129	148	41420	3.25%
#31	**42677**	1570	94	**41100**	0.00%
#32	43911	2565	127	41470	2.89%
#33	43639	2158	108	41775	2.25%
#34	45116	2057	114	42730	5.71%
#35	44569	2796	137	41480	4.43%
#36	44012	2363	138	41715	3.13%
#37	51011	19668	127	41815	19.53%

Table 41: The results of running the algorithm for **problem instance 3**, with the second group of weight sets for the neighborhood operators. Weight set #03 has the weights as shown in table 38. The other weight sets were generated using the Halton sequence with the ranges in table 38. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run.

Weight set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#03	25199	3990	246	**23695**	5.36%
#26	24004	210	243	**23695**	0.37%
#27	23989	193	242	**23695**	0.30%
#28	**23917**	273	221	**23695**	0.00%
#29	24003	229	237	**23695**	0.36%
#30	24027	269	280	**23695**	0.46%
#31	24081	340	245	23775	0.69%
#32	24013	142	258	23755	0.40%
#33	23967	179	257	**23695**	0.21%
#34	24288	560	200	23790	1.55%
#35	24037	373	260	**23695**	0.50%
#36	25377	3817	264	**23695**	6.11%
#37	23959	273	210	**23695**	0.18%

Table 42: The operator weights of weight set #31 and the induced probabilities. These weights are used for all further experiments.

Operator name (Variant)	Set #31 weight	Probability
Insert (Random)	7	0.018%
Insert (Sorted)	1	0.003%
Remove	0	0.000%
Move Period (Random)	6566	16.487%
Move Period (Sorted)	3235	8.123%
Move Room	19268	48.382%
Move Cluster	5293	13.291%
Move Random (Random)	279	0.701%
Move Random (Sorted)	364	0.914%
Change Day Rooms (Random)	280	0.703%
Change Day Rooms (Sort #Rooms)	147	0.369%
Change Day Rooms (Sort Duration)	17	0.043%
Move Best Rooms (Random)	1379	3.463%
Move Best Rooms (Sorted)	1689	4.241%
Move Period Best Rooms (Random)	733	1.841%
Move Period Best Rooms (Sorted)	567	1.424%

C Results of Simulated Annealing Parameters Experiments

Table 43: The Simulated Annealing parameters used previously (during the neighborhood operators' probabilities experiments). The search range for the following experiment is also shown. These ranges are selected based on the experiments shown in section 5.3 and appendix A. $\#Restarts$ is set to ∞, because all runs will keep going until the time limit of 5 minutes is reached.

Parameter name	Initial value	Search range
$InitialTemp$	750	$[500, 1000]$
$ReheatTemp$	750	$[500, 1000]$
$CoolingRate$	0.90	$[0.80, 0.95]$
$InnerLoopIterations$	10000	10000
$MainLoopCutoff$	10000	$[10000, 50000]$
$\#Restarts$	2	∞
$\#Reheats$	2	$[0, 5]$
$\#ReheatsGlobal$	3	$[0, 5]$
$PreprocessTimeDeps$	Yes	Yes

Table 44: The results of running the algorithm for **problem instance 1**, with different sets of parameters for the Simulated Annealing algorithm. Parameter set #01 has the parameter values as shown in table 43. The other parameter sets were generated using the Halton sequence with the ranges in table 43. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Parameter set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	32357	328	305	32065	0.41%
#02	32703	658	305	32115	1.48%
#03	32785	797	306	32150	1.74%
#04	32543	424	307	32105	0.99%
#05	32524	599	304	32070	0.93%
#06	32379	322	303	32105	0.48%
#07	32444	335	307	32085	0.68%
#08	32553	746	302	32040	1.02%
#09	32518	511	311	32125	0.91%
#10	32804	901	304	32065	1.80%
#11	32791	910	303	32065	1.76%
#12	32433	407	305	32050	0.65%
#13	32767	692	303	32115	1.68%
#14	32663	648	309	32155	1.36%
#15	32596	363	305	32210	1.15%
#16	32440	543	302	32065	0.67%
#17	32324	192	307	32140	0.31%
#18	32838	860	305	32085	1.91%
#19	32901	615	306	32120	2.10%
#20	32321	296	305	32065	0.30%
#21	32579	684	303	32095	1.10%
#22	**32224**	132	305	**32030**	0.00%
#23	32766	832	303	32130	1.68%
#24	32474	382	307	32040	0.77%
#25	32328	361	304	32050	0.32%

Table 45: The results of running the algorithm for **problem instance 2**, with different sets of parameters for the Simulated Annealing algorithm. Parameter set #01 has the parameter values as shown in table 43. The other parameter sets were generated using the Halton sequence with the ranges in table 43. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Parameter set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	42964	1417	303	41750	0.50%
#02	**42749**	812	304	41470	0.00%
#03	43363	2102	302	41335	1.44%
#04	43193	2567	305	41375	1.04%
#05	45443	2683	303	41470	6.30%
#06	44593	2574	302	41605	4.31%
#07	50208	18546	304	41495	17.45%
#08	46048	3232	301	41480	7.72%
#09	43359	2057	305	41730	1.43%
#10	43209	1733	302	41510	1.08%
#11	49306	18751	301	41480	15.34%
#12	43283	2418	304	41410	1.25%
#13	50617	18274	303	42175	18.40%
#14	51367	19836	307	**41070**	20.16%
#15	43194	2154	303	41115	1.04%
#16	43325	2272	302	41410	1.35%
#17	43768	2218	303	41085	2.38%
#18	44821	3209	303	41540	4.85%
#19	45680	3458	305	41470	6.86%
#20	44554	2471	302	41470	4.22%
#21	46049	3491	302	41730	7.72%
#22	51211	18146	302	41510	19.80%
#23	50210	20258	302	41470	17.45%
#24	43944	2181	306	41410	2.80%
#25	45381	2622	302	41550	6.16%

Table 46: The results of running the algorithm for **problem instance 3**, with different sets of parameters for the Simulated Annealing algorithm. Parameter set #01 has the parameter values as shown in table 43. The other parameter sets were generated using the Halton sequence with the ranges in table 43. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Parameter set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	24089	319	306	23755	0.89%
#02	24090	349	309	**23695**	0.90%
#03	23972	155	305	**23695**	0.40%
#04	**23876**	149	314	**23695**	0.00%
#05	24322	863	305	**23695**	1.87%
#06	23992	175	306	**23695**	0.49%
#07	24069	352	306	**23695**	0.81%
#08	24083	249	305	**23695**	0.87%
#09	24097	330	314	23755	0.92%
#10	24168	277	307	23800	1.22%
#11	24022	160	304	23755	0.61%
#12	24004	359	307	**23695**	0.54%
#13	24358	696	307	23755	2.02%
#14	24295	488	314	23755	1.75%
#15	24169	480	305	**23695**	1.23%
#16	24410	837	304	23890	2.24%
#17	24068	357	311	23755	0.80%
#18	24022	281	306	**23695**	0.61%
#19	24086	321	311	**23695**	0.88%
#20	24054	291	306	23755	0.75%
#21	26398	4913	305	23755	10.56%
#22	25129	3766	306	**23695**	5.25%
#23	24151	391	305	23755	1.15%
#24	24192	438	314	**23695**	1.32%
#25	23981	213	306	**23695**	0.44%

Table 47: The Simulated Annealing parameters of parameter set #01, that are used as a base for the next experiment. The search range for the following experiment is also shown. These ranges are selected based on the experiments shown in section 5.3 and appendix A, but are set to smaller intervals compared to the previous experiment. $\#Restarts$ is set to ∞, because all runs will keep going until the time limit of 5 minutes is reached.

Parameter name	Set #01 value	Search range
$InitialTemp$	750	$[650, 850]$
$ReheatTemp$	750	$[650, 850]$
$CoolingRate$	0.90	$[0.88, 0.92]$
$InnerLoopIterations$	10000	10000
$MainLoopCutoff$	10000	$[10000, 25000]$
$\#Restarts$	-	∞
$\#Reheats$	2	$[1, 3]$
$\#ReheatsGlobal$	3	$[2, 4]$
$PreprocessTimeDeps$	Yes	Yes

Table 48: The results of running the algorithm for **problem instance 1**, with the second group of parameter sets for the Simulated Annealing algorithm. Parameter set #01 has the parameter values as shown in table 47. The other parameter sets were generated using the Halton sequence with the ranges in table 47. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Parameter set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	32503	367	306	32105	0.96%
#26	32345	218	305	32145	0.47%
#27	32466	407	305	32170	0.84%
#28	32467	389	304	32060	0.85%
#29	**32194**	117	304	32105	0.00%
#30	32724	510	304	32075	1.65%
#31	32415	310	306	32115	0.69%
#32	32429	368	304	32090	0.73%
#33	32524	685	306	**31995**	1.02%
#34	32595	538	307	32045	1.24%
#35	32319	283	304	32130	0.39%
#36	32399	363	306	32030	0.64%
#37	32502	408	304	32095	0.96%

Table 49: The results of running the algorithm for **problem instance 2**, with the second group of parameter sets for the Simulated Annealing algorithm. Parameter set #01 has the parameter values as shown in table 47. The other parameter sets were generated using the Halton sequence with the ranges in table 47. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Parameter set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	50102	18493	303	41760	18.69%
#26	50509	18366	303	41410	19.65%
#27	42901	1876	304	41285	1.63%
#28	49883	18622	303	41420	18.17%
#29	43754	2927	303	41520	3.65%
#30	42699	1803	303	41420	1.15%
#31	43578	2902	304	41065	3.23%
#32	43656	2531	302	41410	3.42%
#33	50443	21446	303	**41055**	19.49%
#34	**42214**	1542	303	41410	0.00%
#35	44396	2029	303	42635	5.17%
#36	43923	2441	305	41420	4.05%
#37	43352	2124	303	41345	2.70%

Table 50: The results of running the algorithm for **problem instance 3**, with the second group of parameter sets for the Simulated Annealing algorithm. Parameter set #01 has the parameter values as shown in table 47. The other parameter sets were generated using the Halton sequence with the ranges in table 47. Numbers in bold face indicate the best result for that statistic. The *% above lowest avg.* column shows how far the set's average penalty is above the lowest average penalty. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Parameter set	Penalty avg.	Penalty SD	Time avg. (s)	Penalty min	% above lowest avg.
#01	24087	293	308	23755	1.03%
#26	24187	439	308	**23695**	1.45%
#27	24098	344	309	23755	1.07%
#28	24033	328	308	**23695**	0.80%
#29	25049	2420	309	23950	5.06%
#30	23953	329	309	**23695**	0.46%
#31	**23842**	162	307	**23695**	0.00%
#32	24033	160	305	23775	0.80%
#33	23986	323	308	**23695**	0.60%
#34	23892	225	305	**23695**	0.21%
#35	24141	350	307	23755	1.25%
#36	23991	166	308	**23695**	0.62%
#37	23930	183	306	**23695**	0.37%

Table 51: The Simulated Annealing parameters of parameter set #34. These values are used for all further experiments. There is no value for *#Restarts*, because the tests were stopped using a time limit instead.

Parameter name	Set #34 value
InitialTemp	660
ReheatTemp	740
CoolingRate	0.917
InnerLoopIterations	10000
MainLoopCutoff	21461
#Restarts	-
#Reheats	2
#ReheatsGlobal	3
PreprocessTimeDeps	Yes

D Results of Initial Greedy Solutions Experiments

Table 52: The results of running the algorithm for all three problem instances, with different settings of the initial greedy algorithm. The different settings refer to the measure for selecting and scheduling meetings respectively, as shown in section 4.4. The hyphen means that an empty initial solution was used. All experiments were run 9 times, with a time limit of 5 minutes for each run and with the *#Restarts* parameter set to ∞.

Instance 1				
Greedy algorithm settings	Initial penalty avg.	Penalty avg.	Penalty min	% above lowest avg.
-	882000	**32207**	**31970**	0.00%
AP/Pen.	**52693**	32685	32205	1.48%
AP/TAP	253342	32938	32250	2.27%
SL/Pen.	59961	32793	32070	1.82%
SL/TAP	247905	54454	32280	69.08%
RL/Pen.	62076	32429	32180	0.69%
RL/TAP	266749	56219	32930	74.56%
Instance 2				
Greedy algorithm settings	Initial penalty avg.	Penalty avg.	Penalty min	% above lowest avg.
-	1050000	**43982**	**41760**	0.00%
AP/Pen.	**62996**	48599	43795	10.50%
AP/TAP	326167	50292	49130	14.35%
SL/Pen.	73743	48457	46365	10.17%
SL/TAP	313786	51024	45965	16.01%
RL/Pen.	64747	50462	46270	14.73%
RL/TAP	313587	50250	46110	14.25%
Instance 3				
Greedy algorithm settings	Initial penalty avg.	Penalty avg.	Penalty min	% above lowest avg.
-	936000	**23965**	**23695**	0.00%
AP/Pen.	48715	31558	23755	31.69%
AP/TAP	291346	38399	32960	60.23%
SL/Pen.	55299	33571	23875	40.08%
SL/TAP	297278	35364	31820	47.56%
RL/Pen.	**45722**	30705	**23695**	28.12%
RL/TAP	292852	37232	32380	55.36%

Table 53: The results of running the algorithm for all three problem instances, with different settings of the initial greedy algorithm and with a time limit of 1 to 5 minutes. The different settings refer to the measure for selecting and scheduling meetings respectively, as shown in section 4.4. The hyphen means that an empty initial solution was used. All experiments were run 9 times, with the *#Restarts* parameter set to ∞.

Instance 1					
Greedy algorithm settings	Penalty avg., 5 min	Penalty avg., 4 min	Penalty avg., 3 min	Penalty avg., 2 min	Penalty avg., 1 min
-	**32207**	**32698**	**32370**	**32532**	**32634**
AP/Pen.	32685	32952	32482	33418	33346
AP/TAP	32938	33969	33385	34062	35229
SL/Pen.	32793	32846	32977	33311	34014
SL/TAP	54454	54298	53956	59217	62341
RL/Pen.	32429	32885	32967	33099	33921
RL/TAP	56219	59382	64999	64001	59681

Instance 2					
Greedy algorithm settings	Penalty avg., 5 min	Penalty avg., 4 min	Penalty avg., 3 min	Penalty avg., 2 min	Penalty avg., 1 min
-	**43982**	**42926**	**44102**	**44740**	**42997**
AP/Pen.	48599	48773	48889	48052	47971
AP/TAP	50292	51263	51347	50868	52735
SL/Pen.	48457	48424	49859	48289	49588
SL/TAP	51024	49372	50288	50478	50604
RL/Pen.	50462	49237	50447	49412	49694
RL/TAP	50250	51447	51375	51890	50818

Instance 3					
Greedy algorithm settings	Penalty avg., 5 min	Penalty avg., 4 min	Penalty avg., 3 min	Penalty avg., 2 min	Penalty avg., 1 min
-	**23965**	**23971**	**24230**	**24361**	**24244**
AP/Pen.	31558	30658	28289	28259	27628
AP/TAP	38399	35194	36053	37428	36309
SL/Pen.	33571	28692	34031	34795	32079
SL/TAP	35364	36211	33761	36062	37932
RL/Pen.	30705	32001	34126	31801	30246
RL/TAP	37232	39864	35754	36301	37659

E Statistics of the Neighborhood Operators

In this appendix we show charts and tables with statistics from one run of our algorithm. The run we selected is the best run of the algorithm with the selected settings, for problem instance 1. Instance 1 was chosen because, of the three instances, it has the median best found penalty value. Some global stats of the instance and this specific run, can be found in table 12.

All charts in this appendix show the values of one or more statistics throughout the 16,530,000 iterations of the algorithm run. The run is divided in blocks of 10,000 iterations, and the values of a statistic in each block are aggregated into one value.

The first chart (fig. 1) shows the penalty value of the current and best found solution through the run. The best found solution was found between iterations 6,460,000 and 6,470,000. When a reheat happens, the current penalty goes up enormously, after which it goes down again. A restart is also visible in the chart: the current penalty value is first set to the best penalty value, to immediately go up, and then go down again. An example of this can be seen after just under 8 million iterations. The second chart (fig. 2) shows the Simulated Annealing temperature throughout the run. Since the vertical axis has a logarithmic scale and the temperature goes down exponentially, this graph consists of straight lines.

To make the remainder of the charts clearer, they do not show the values throughout the entire run, but rather stop at 7,200,000 iterations, which is exactly 1 iteration of the reheat loop (see section 4.2) after the best solution was found. Figures 3 to 4 show the same statistics as the first two charts, but only throughout the first 7.2 million iterations.

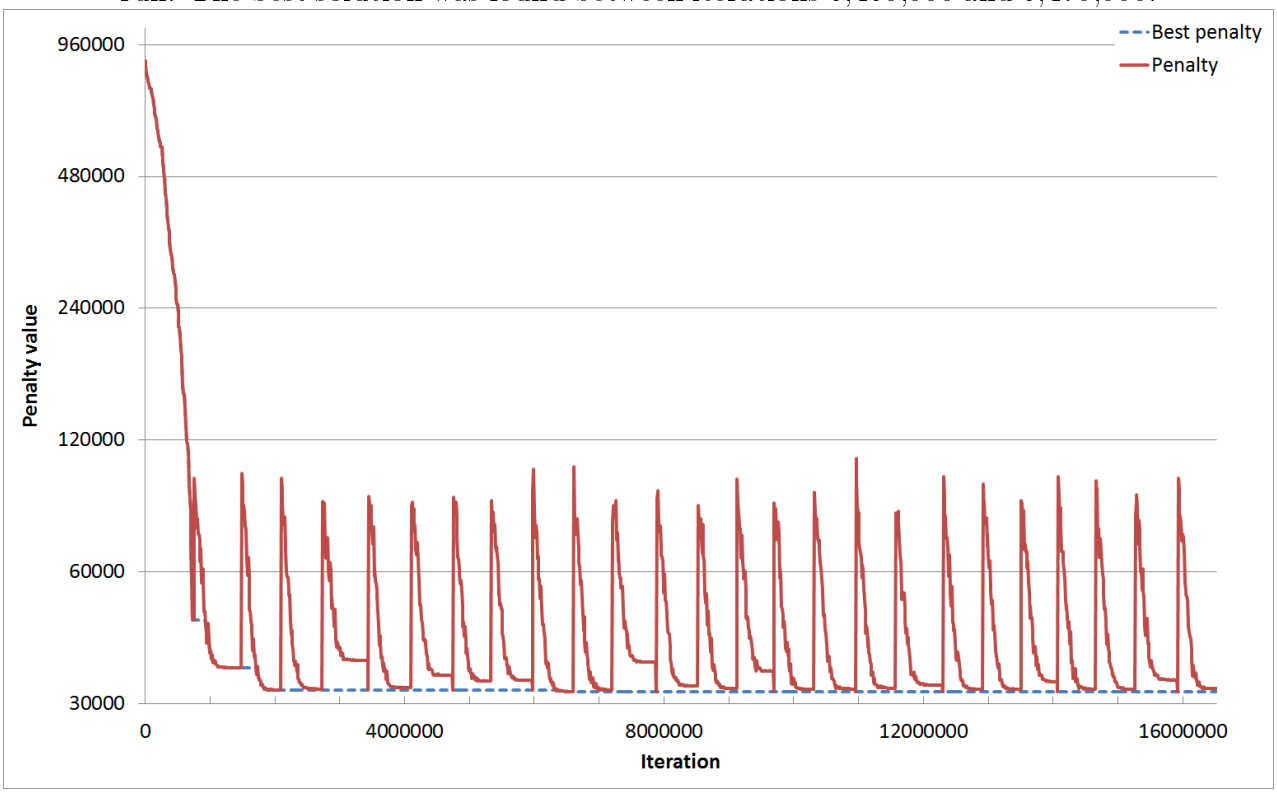
Figures 5 to 34 show statistics for each of the neighborhood operators separately. For each operator, there are two charts. The first chart shows the distribution of the results of the calls to the operator. *Improved* means that the resulting schedule had a better penalty value than before and *equal penalty* means the penalty is equal. *Worse, accepted* means the resulting timetable was worse, but Simulated Annealing accepted the change nonetheless, whereas *worse, declined* means the applied change was discarded. *Failed* means the operator selected an operand, but the operation still failed (e.g., the *Insert* operator selected an unscheduled meeting, but there was no position in the timetable to schedule it). Finally, *not possible* means that the operator could not be executed, because there were no available operands (e.g., the *Insert* operator is called, but there are no unscheduled meetings).

The second chart for each operator shows the average change in penalty induced by an accepted change made by the operator (i.e., for all accepted changes made by the operator, the change in penalty is taken and of all those

values the average is calculated). The chart also shows the average change in penalty induced by calls that resulted in improvement of the penalty and by calls that resulted in a worsening of the penalty, but were accepted by Simulated Annealing. These are not shown for the *Insert* operator, because it almost exclusively results in improvements. The values in these charts (second chart for each operator) display a sliding average over 30,000 iterations per data point, rather than the raw values for 10,000 iterations. This is done to make the graph more readable, but for peaks of just one data point, the value should be tripled.

Finally, tables 54 to 55 show these operator statistics summed over the entire run of the algorithm.

Figure 1: The penalty value of the current schedule (s in algorithm 1) and the penalty value of the best found schedule ($\bar{s}$ in algorithm 1) throughout the run. The best solution was found between iterations 6,460,000 and 6,470,000.



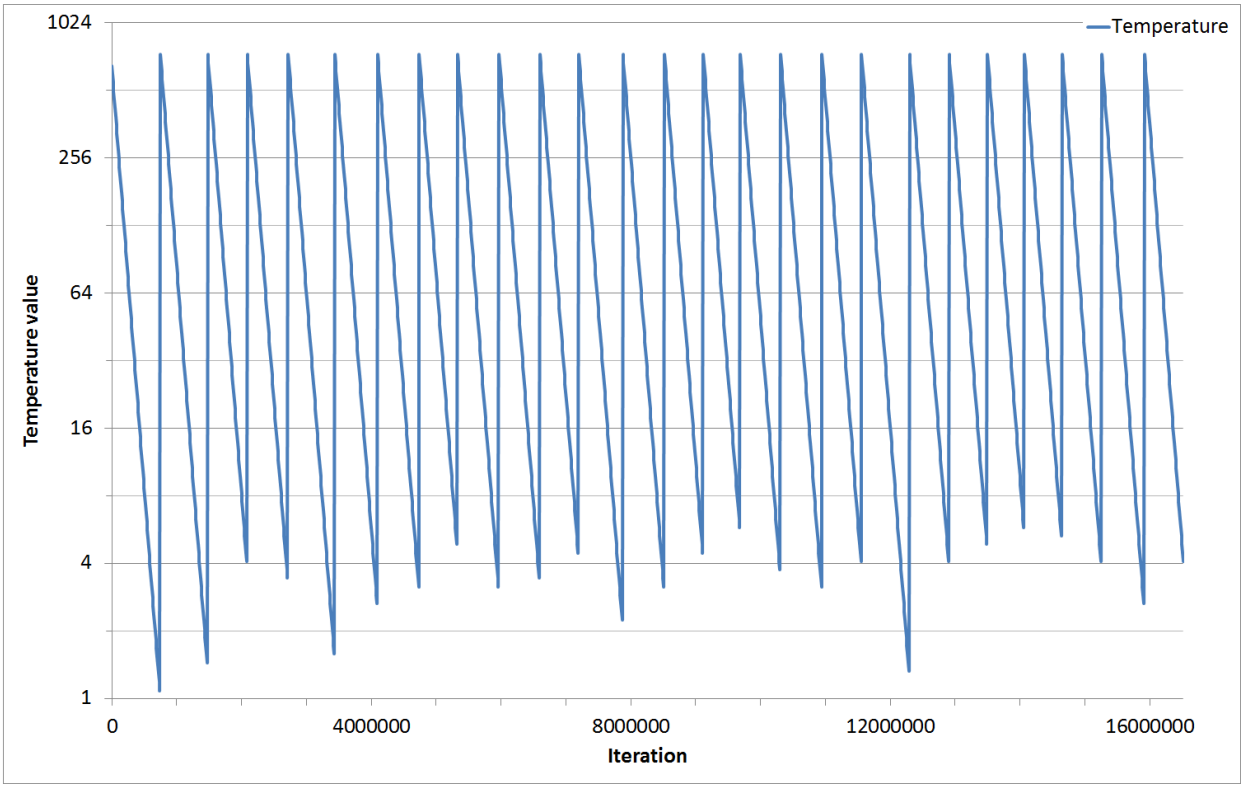
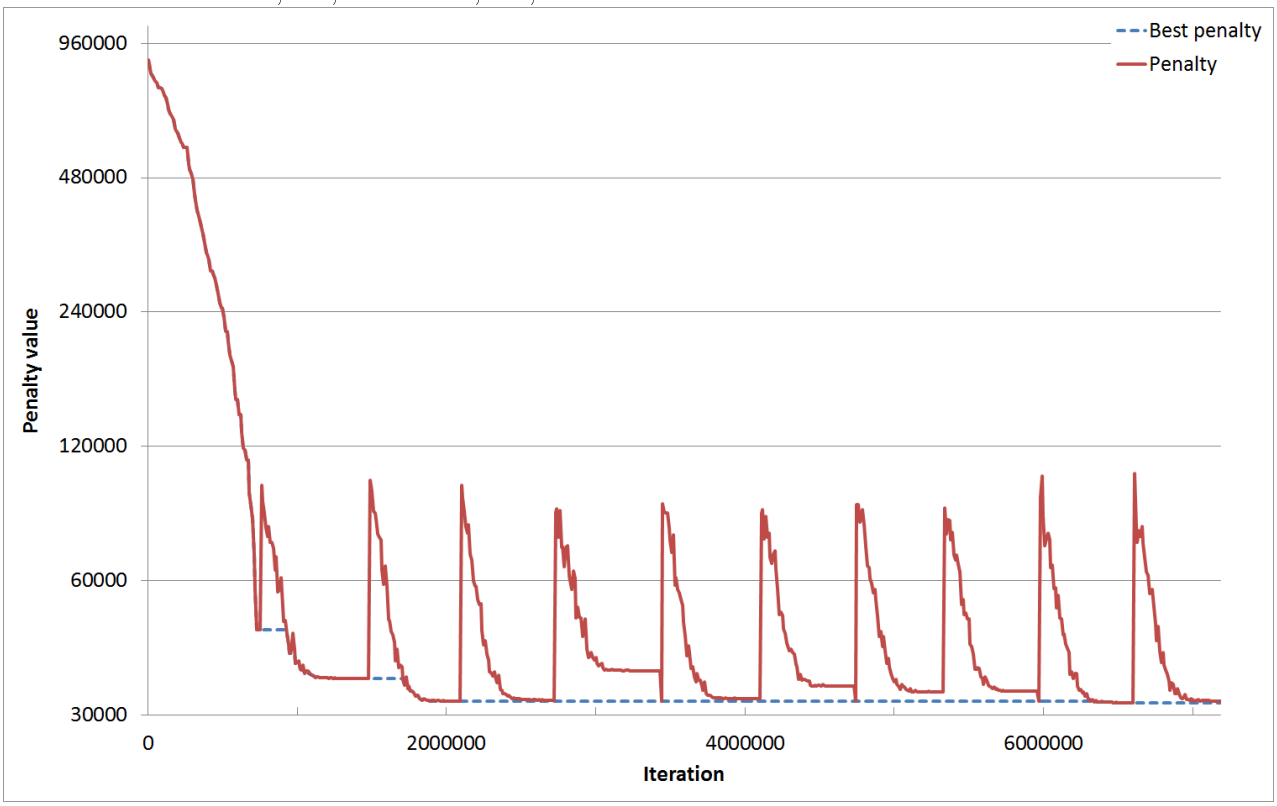


Figure 2: The Simulated Annealing temperature throughout the run. The vertical axis has a logarithmic scale, so the exponential temperature function appears as straight lines.

Figure 3: The penalty value of the current schedule (s in algorithm 1) and the penalty value of the best found schedule ($\bar{s}$ in algorithm 1) throughout the first 7.2 million iterations of the run. The best solution was found between iterations 6,460,000 and 6,470,000.



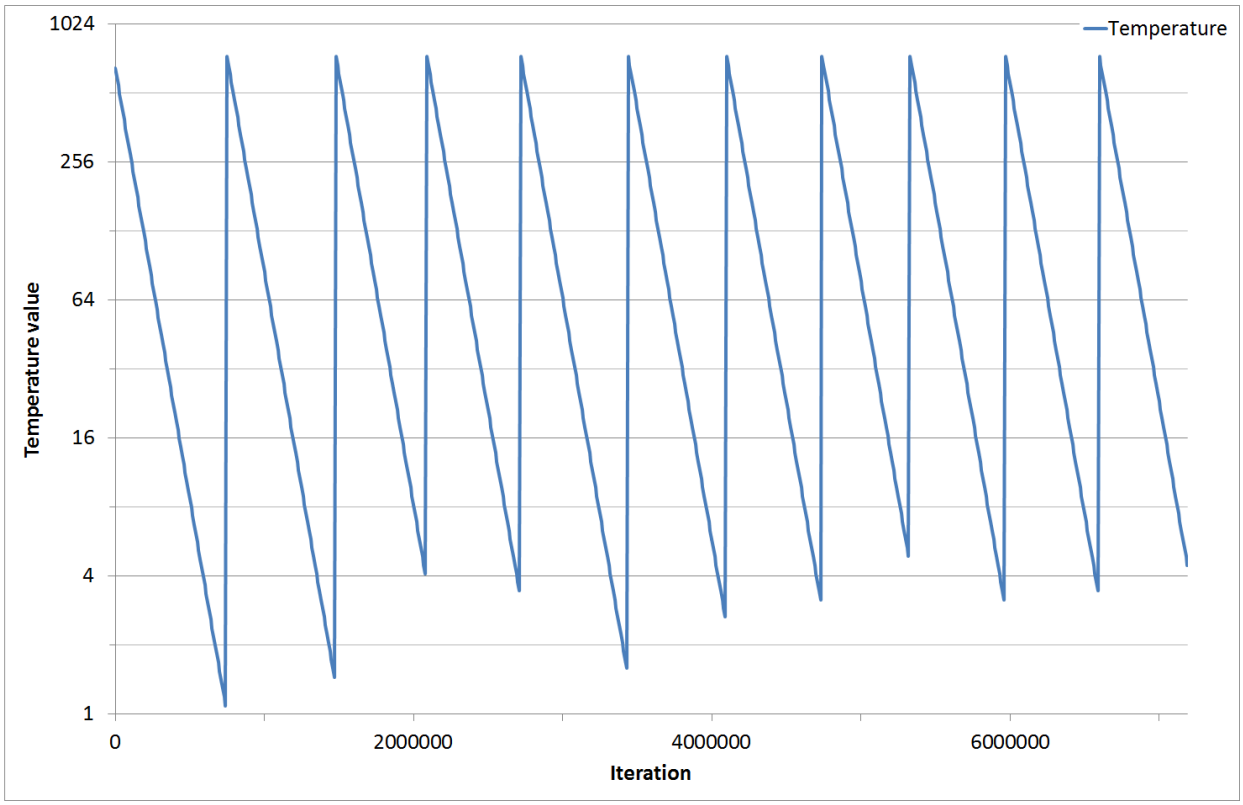
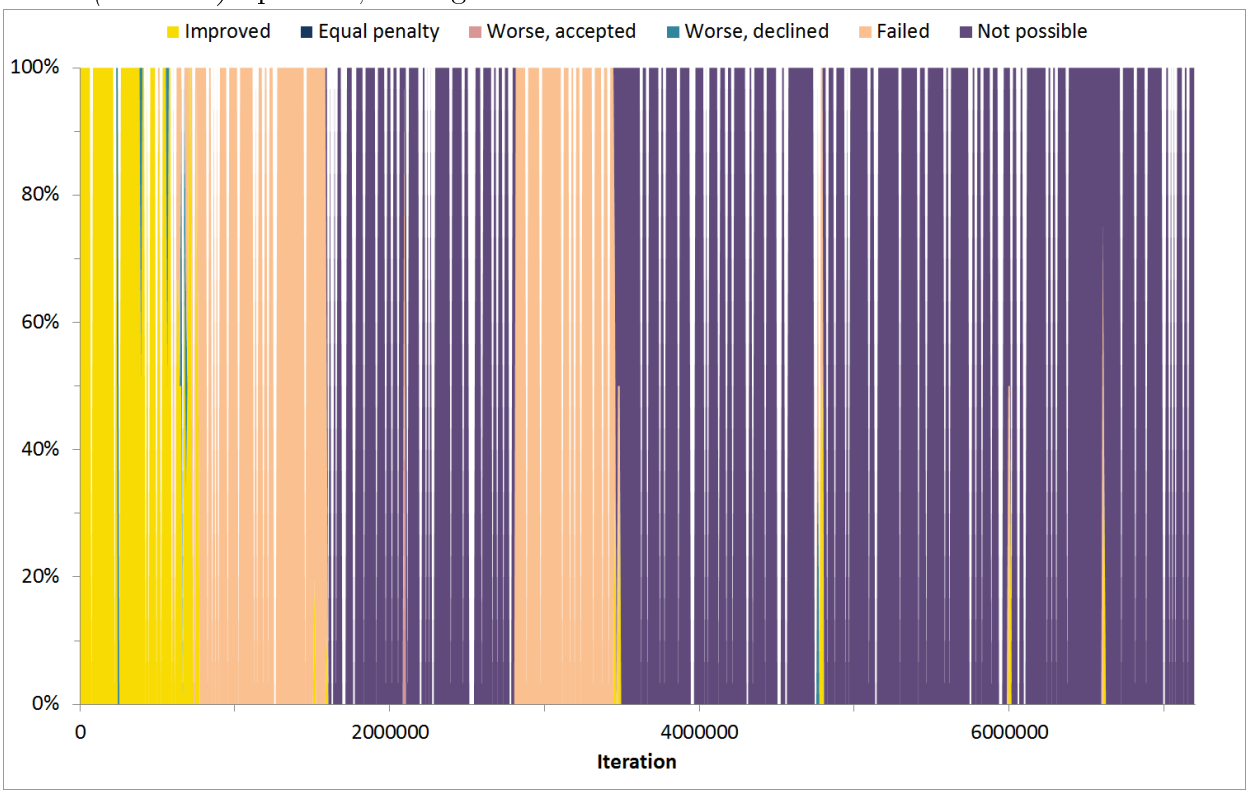


Figure 4: The Simulated Annealing temperature throughout the first 7.2 million iterations of the run.

Figure 5: The distribution (over 10,000 iterations) of the results of the *Insert (Random)* operator, throughout the first 7.2 million iterations.



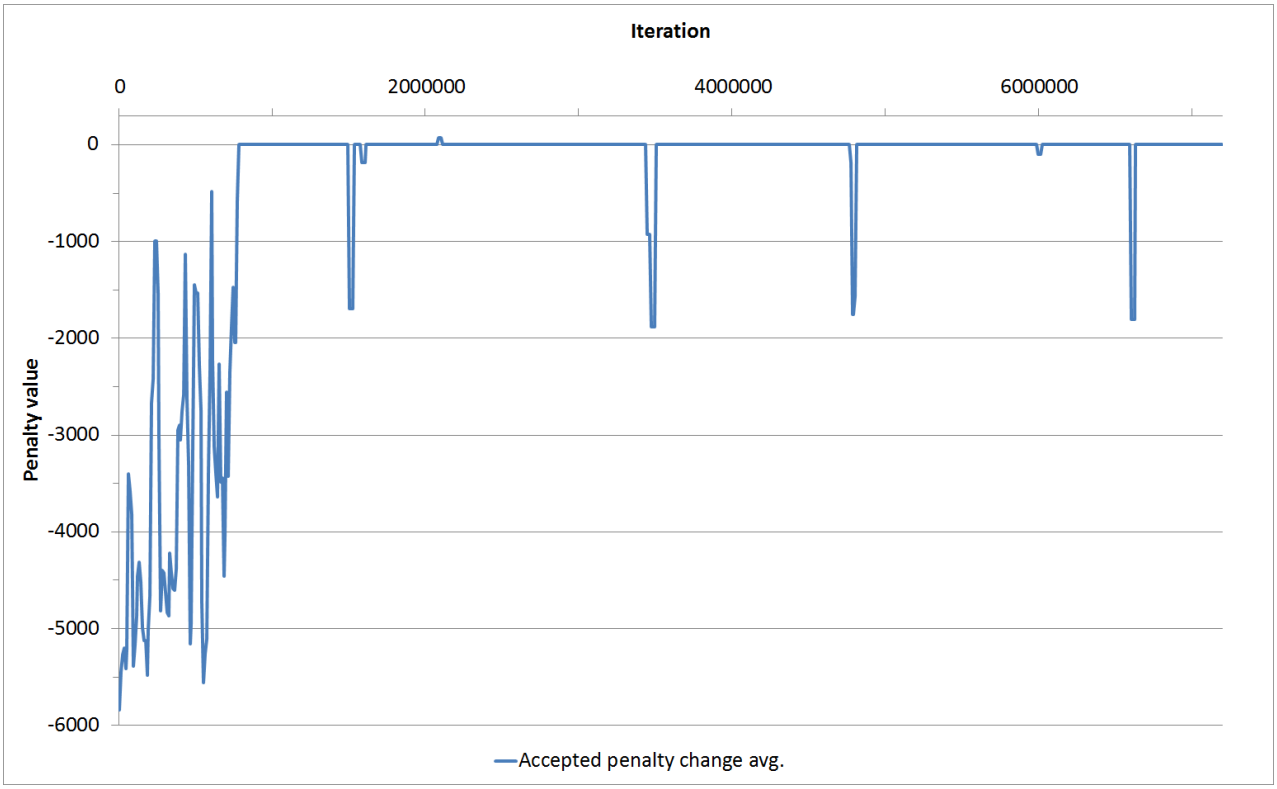
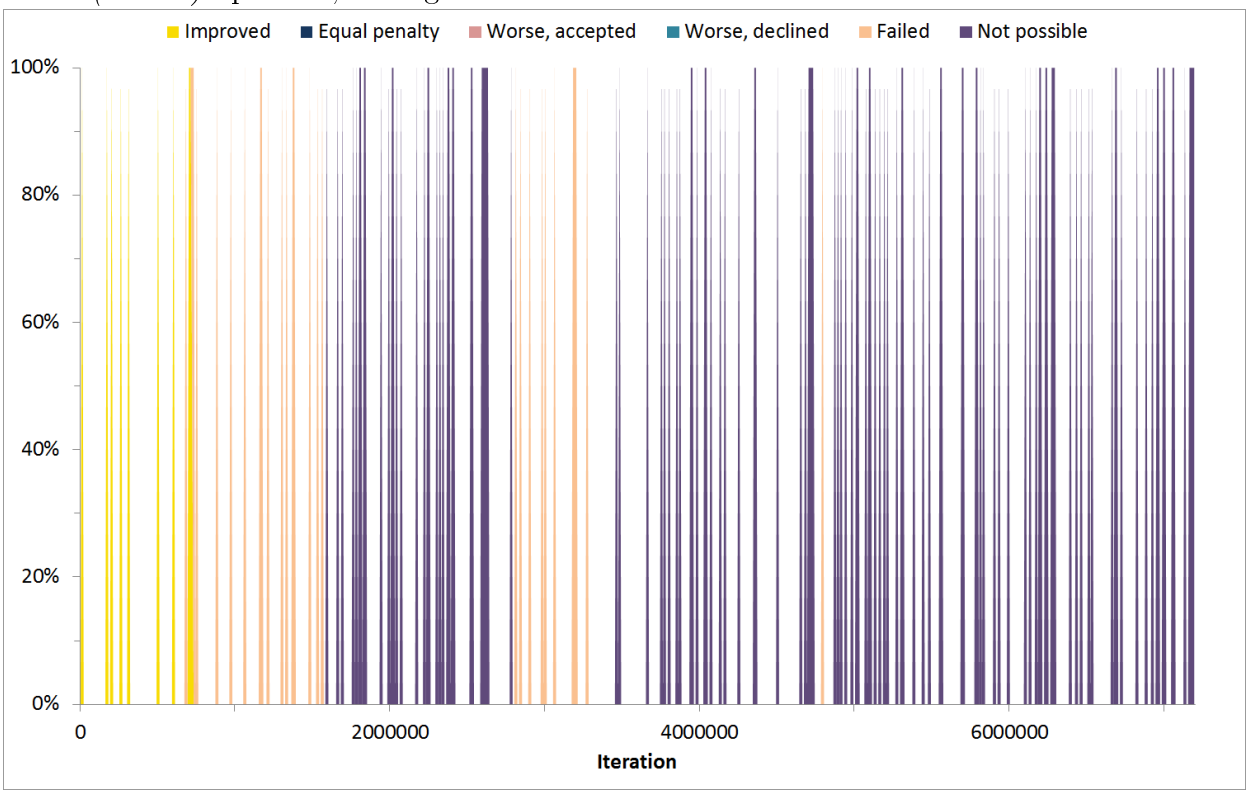


Figure 6: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Insert (Random)* operator, throughout the first 7.2 million iterations.

Figure 7: The distribution (over 10,000 iterations) of the results of the *Insert (Sorted)* operator, throughout the first 7.2 million iterations.



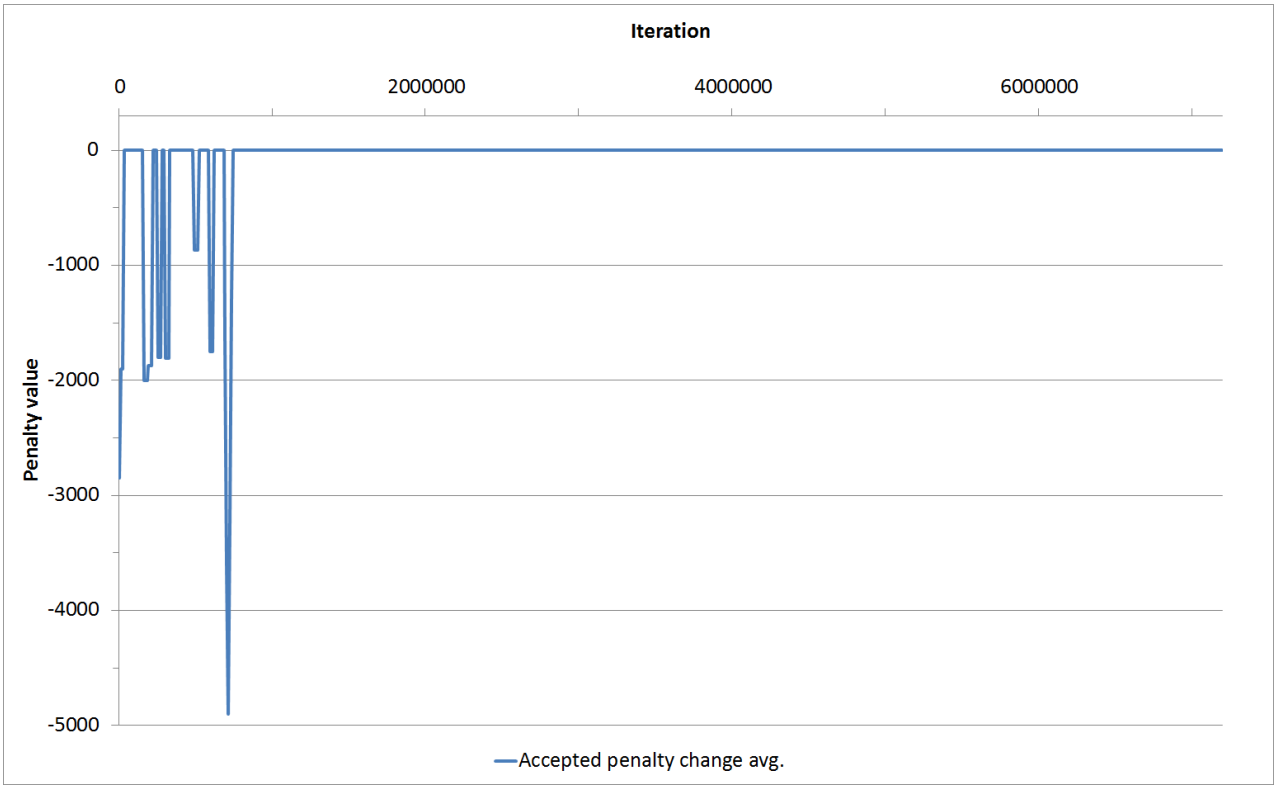
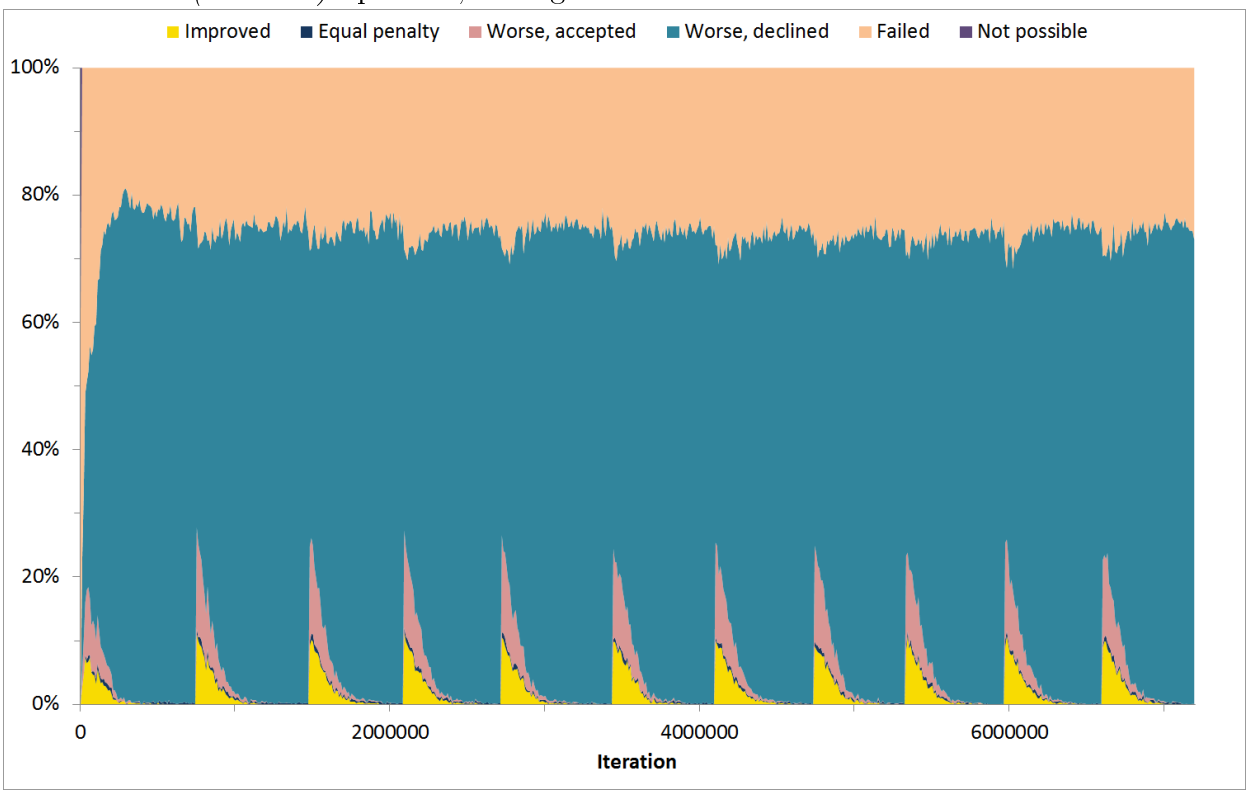


Figure 8: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Insert (Sorted)* operator, throughout the first 7.2 million iterations.

Figure 9: The distribution (over 10,000 iterations) of the results of the *Move Period (Random)* operator, throughout the first 7.2 million iterations.



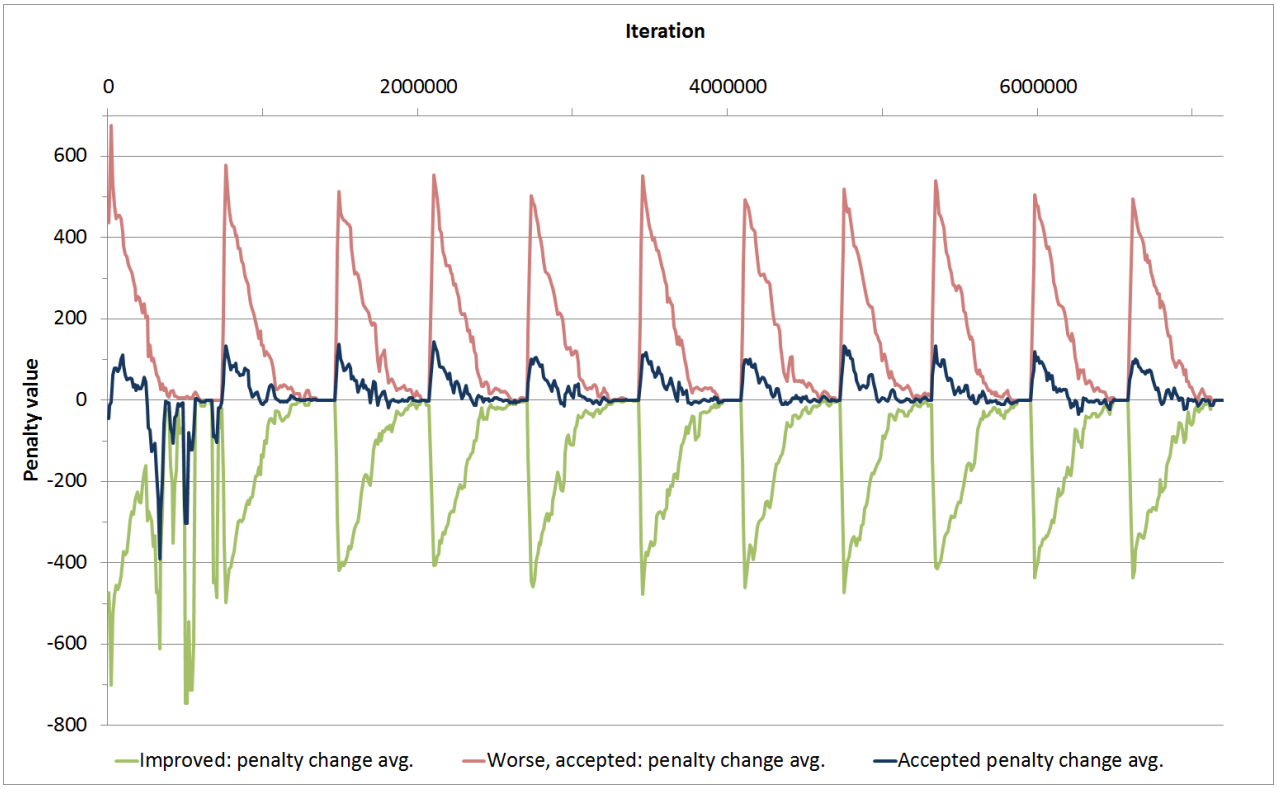
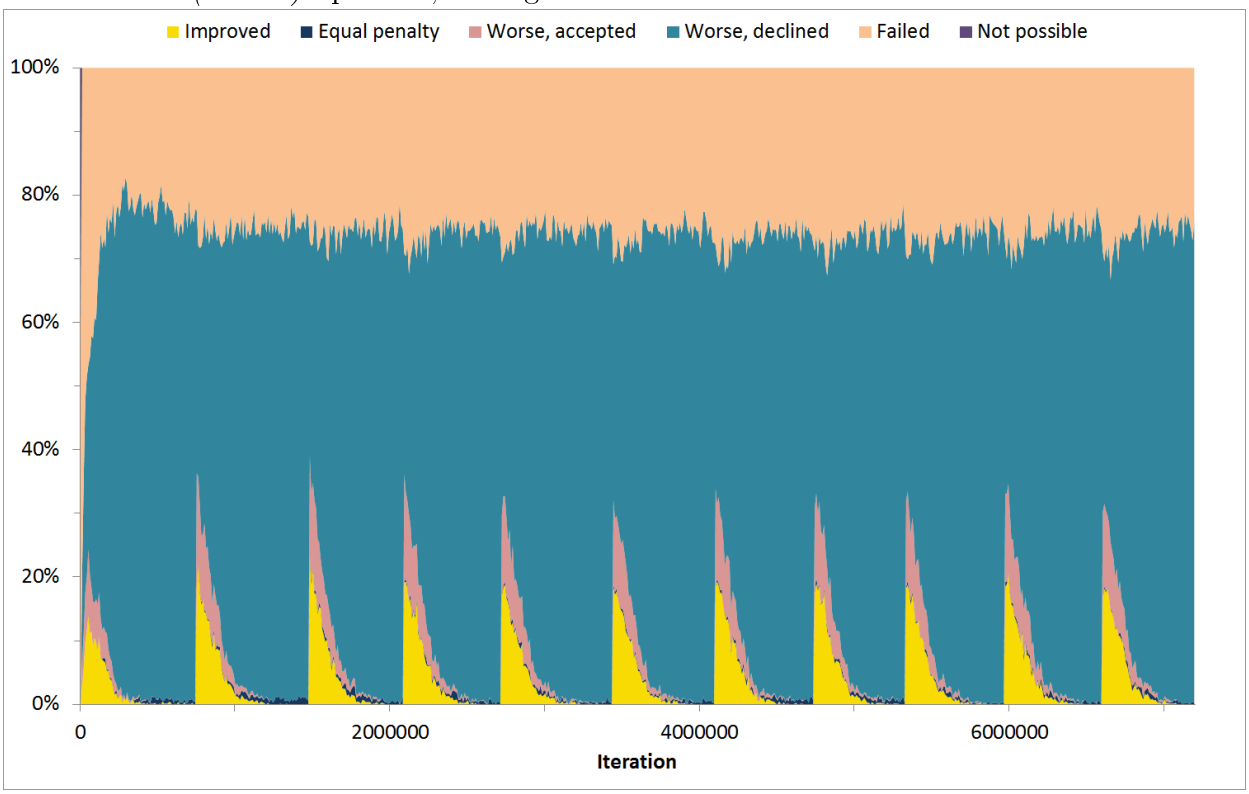


Figure 10: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Period (Random)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 11: The distribution (over 10,000 iterations) of the results of the *Move Period (Sorted)* operator, throughout the first 7.2 million iterations.



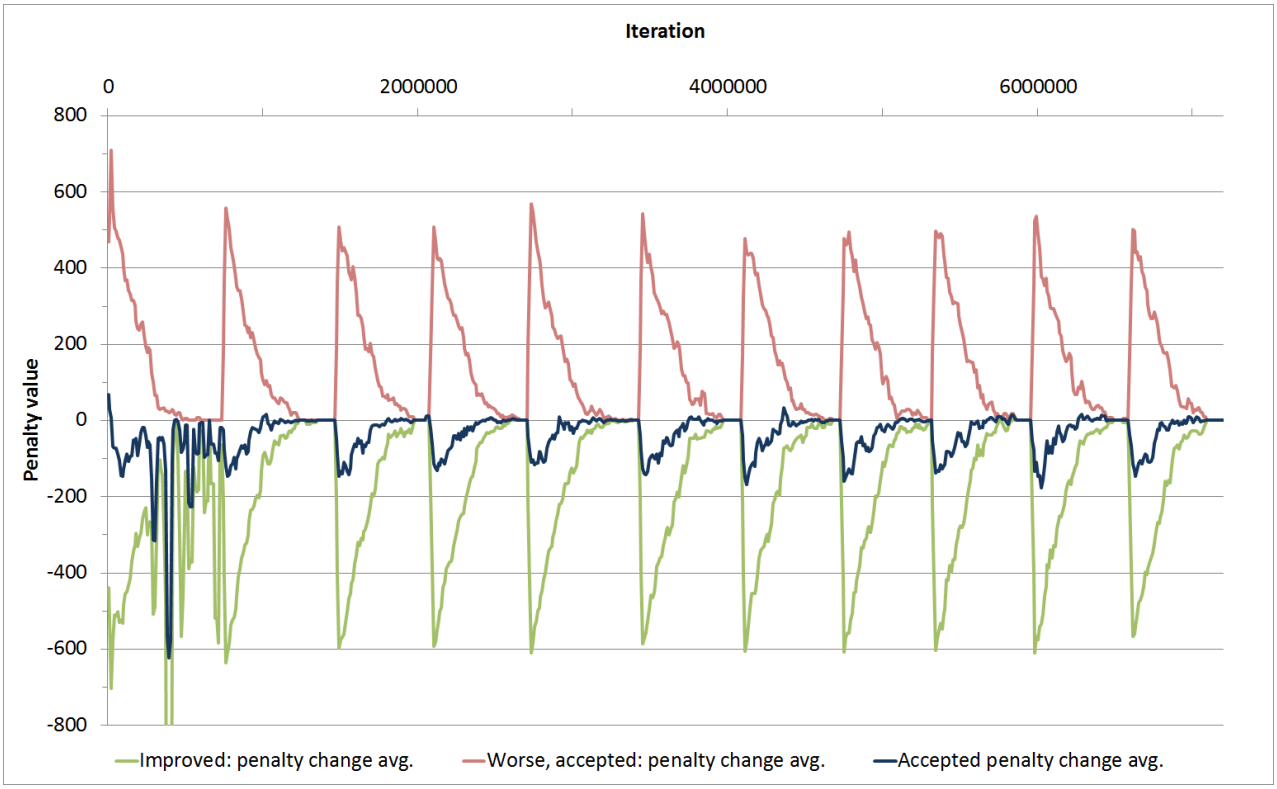
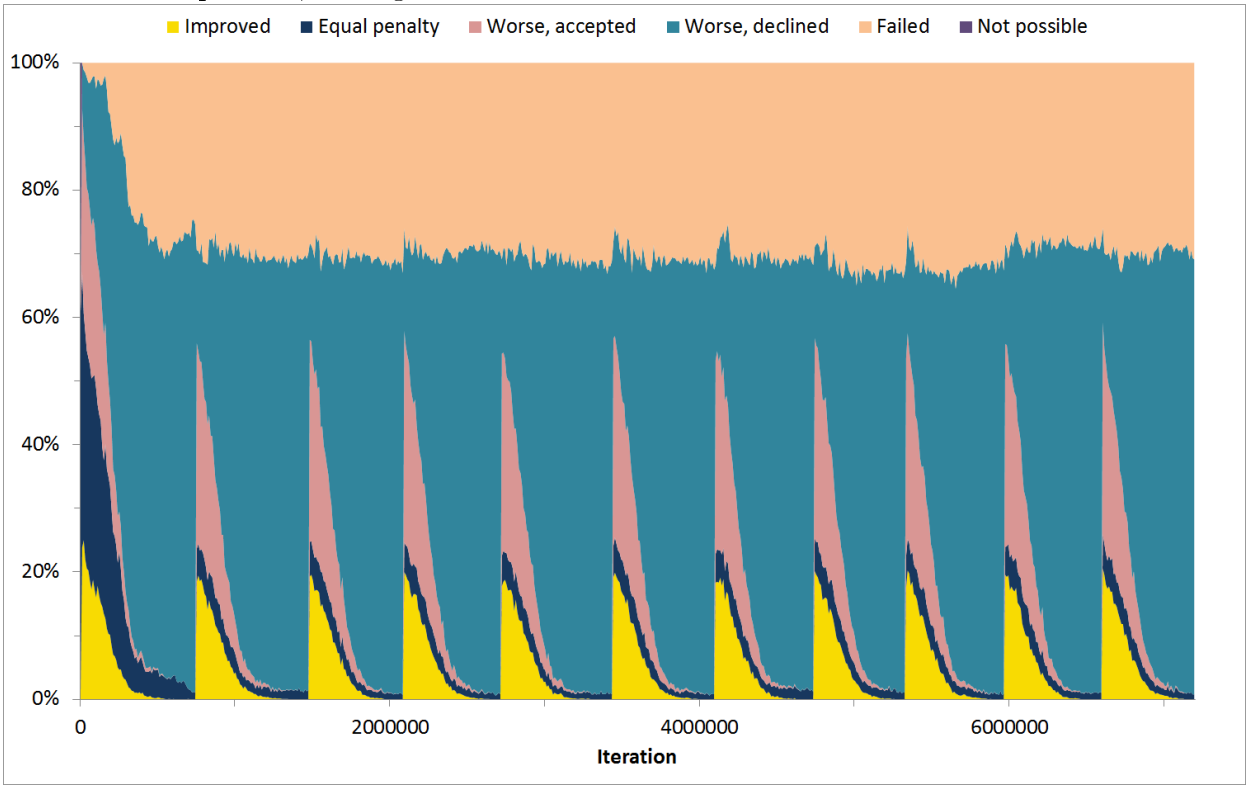


Figure 12: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Period (Sorted)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 13: The distribution (over 10,000 iterations) of the results of the *Move Room* operator, throughout the first 7.2 million iterations.



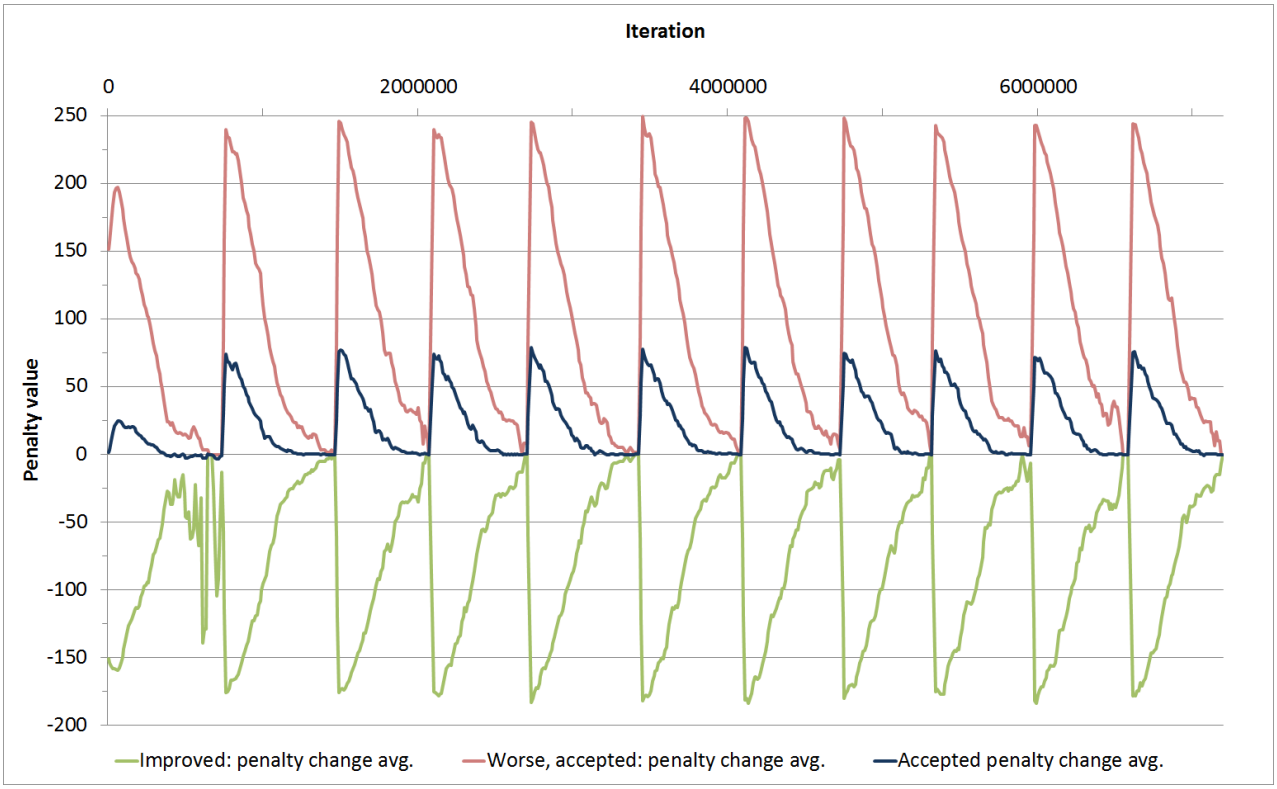
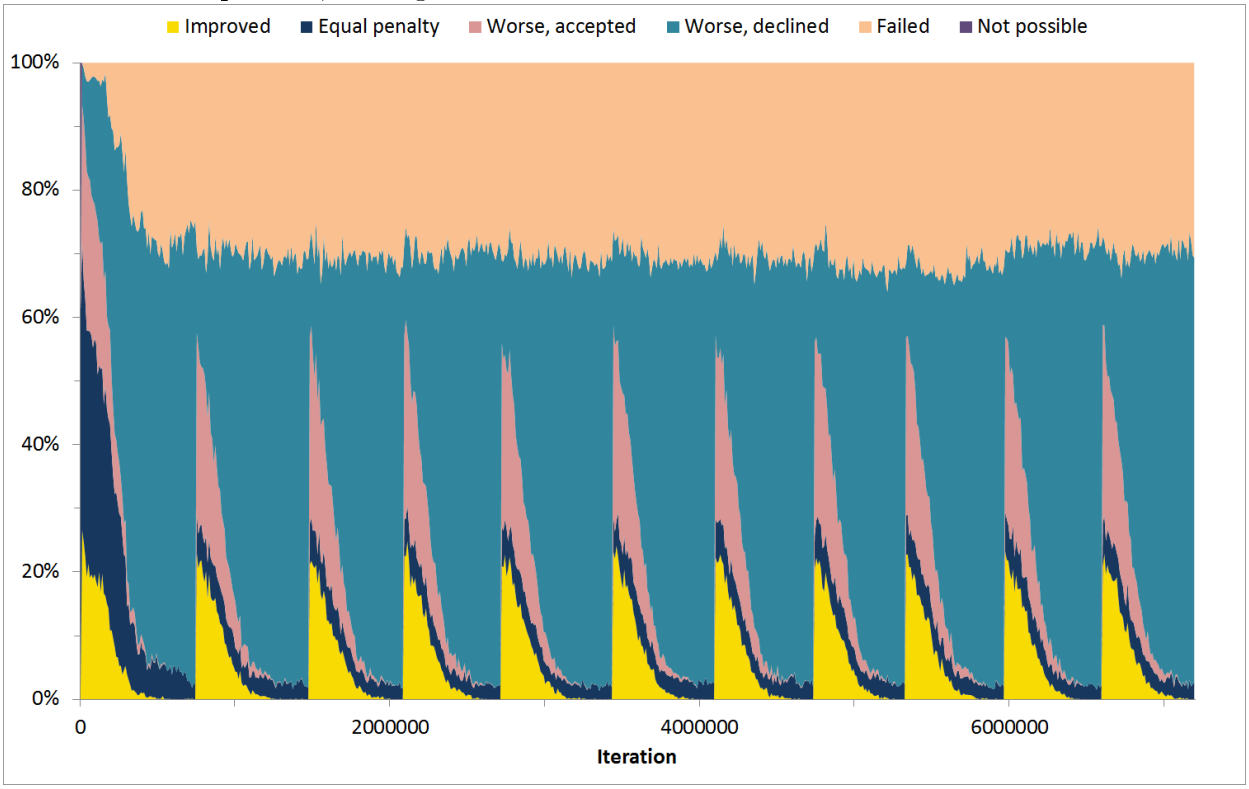


Figure 14: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Room* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 15: The distribution (over 10,000 iterations) of the results of the *Move Cluster* operator, throughout the first 7.2 million iterations.



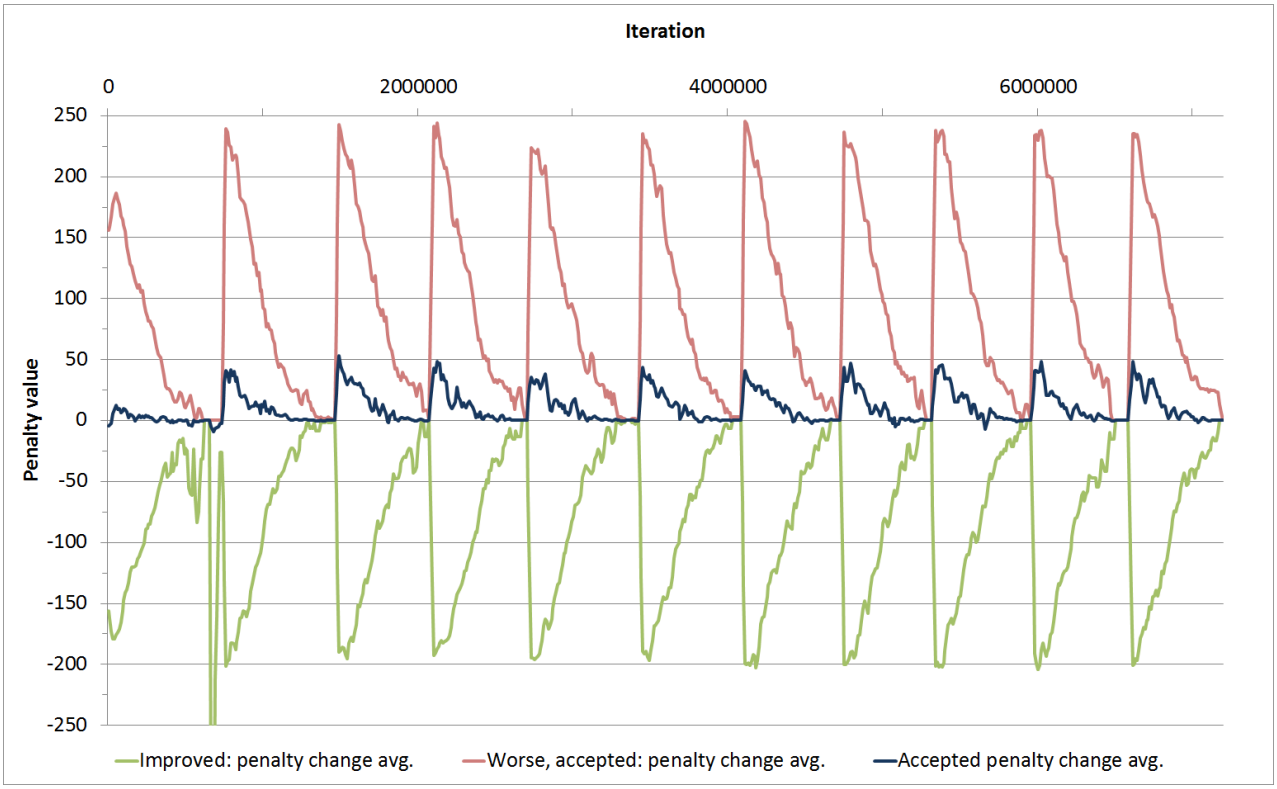
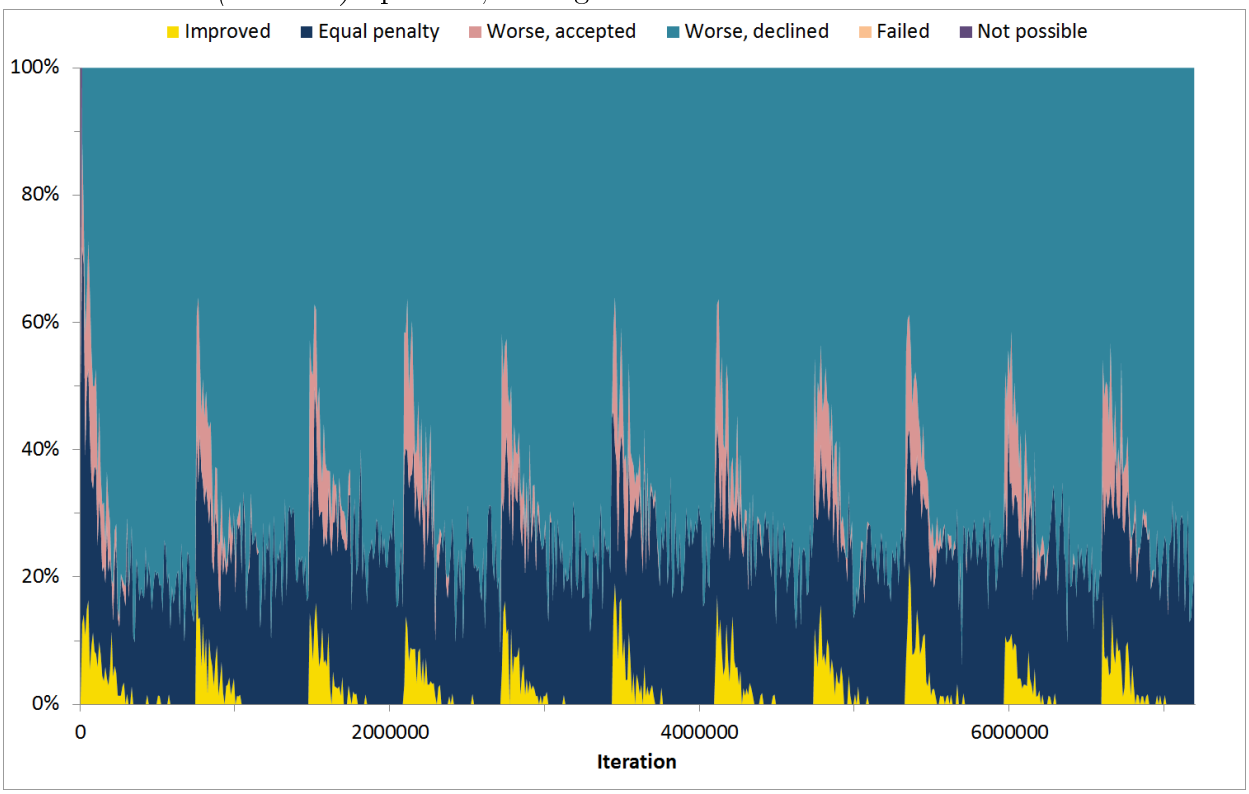


Figure 16: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Cluster* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 17: The distribution (over 10,000 iterations) of the results of the *Move Random (Random)* operator, throughout the first 7.2 million iterations.



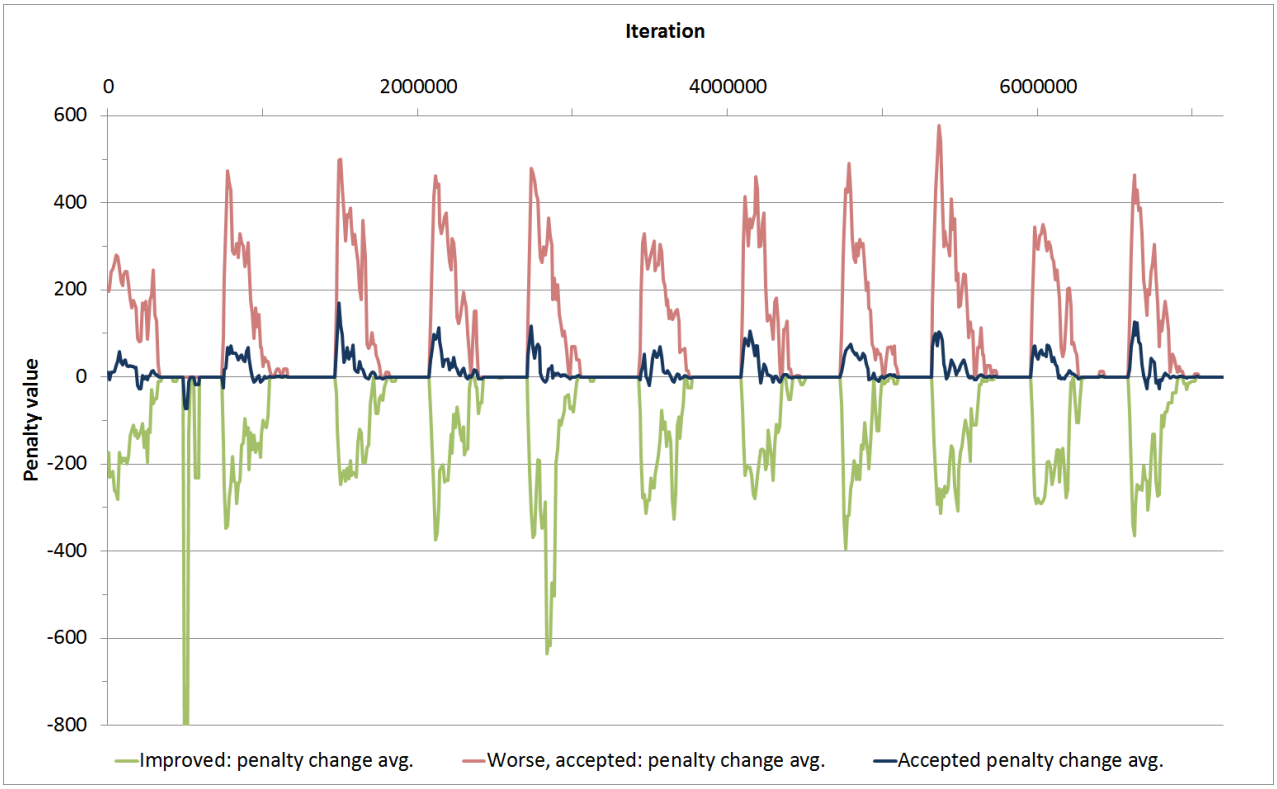
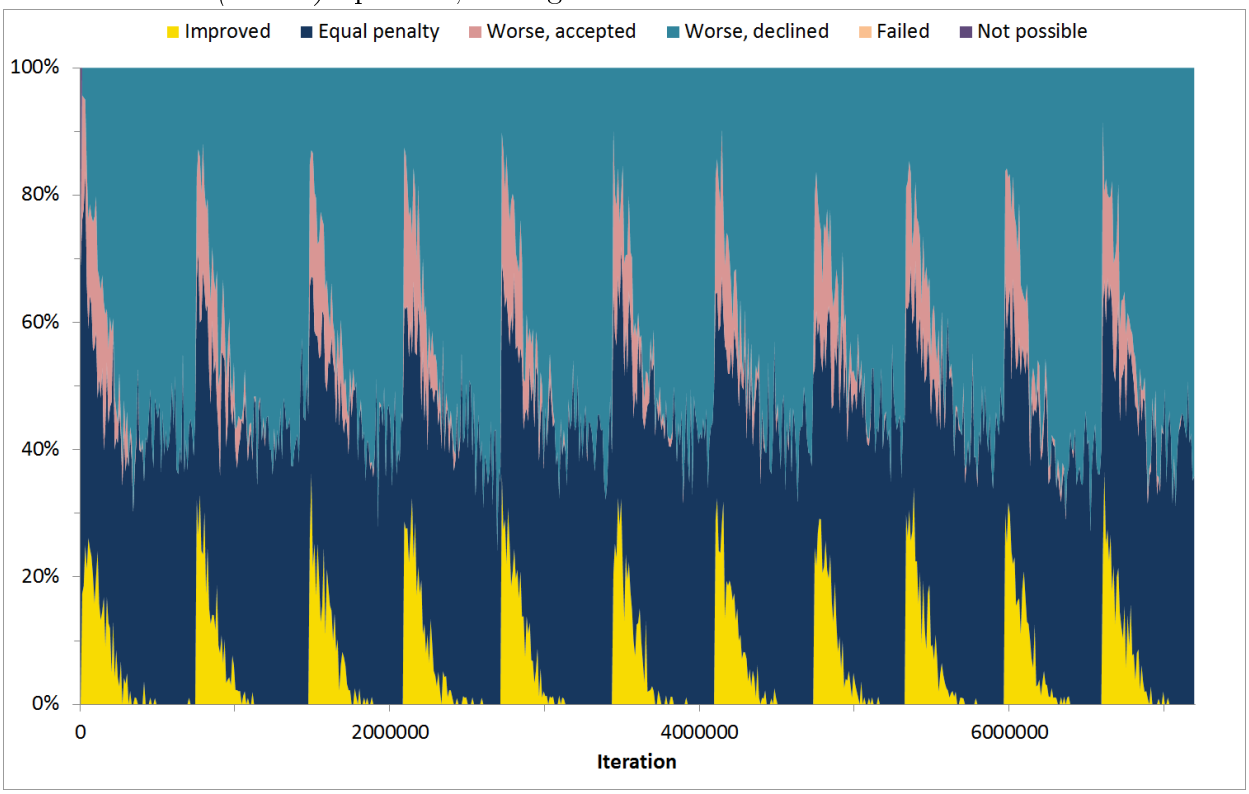


Figure 18: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Random (Random)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 19: The distribution (over 10,000 iterations) of the results of the *Move Random (Sorted)* operator, throughout the first 7.2 million iterations.



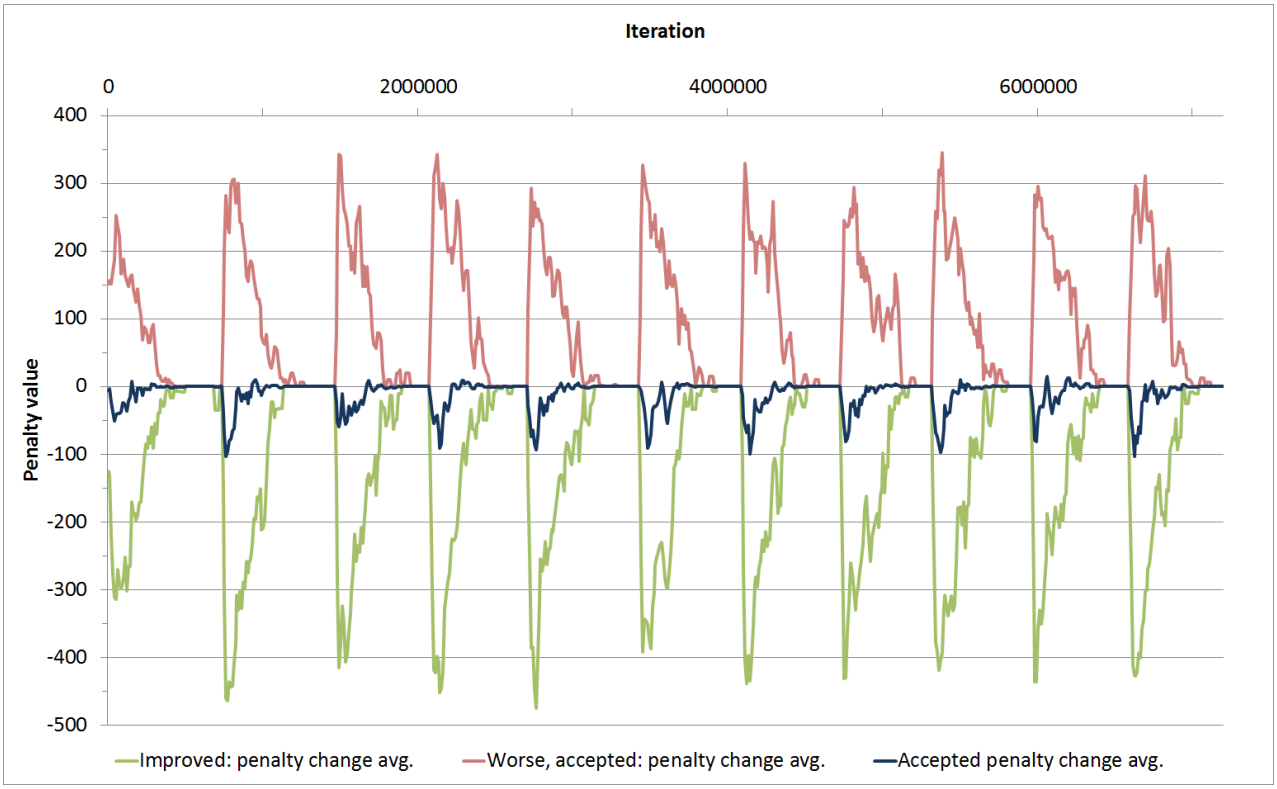
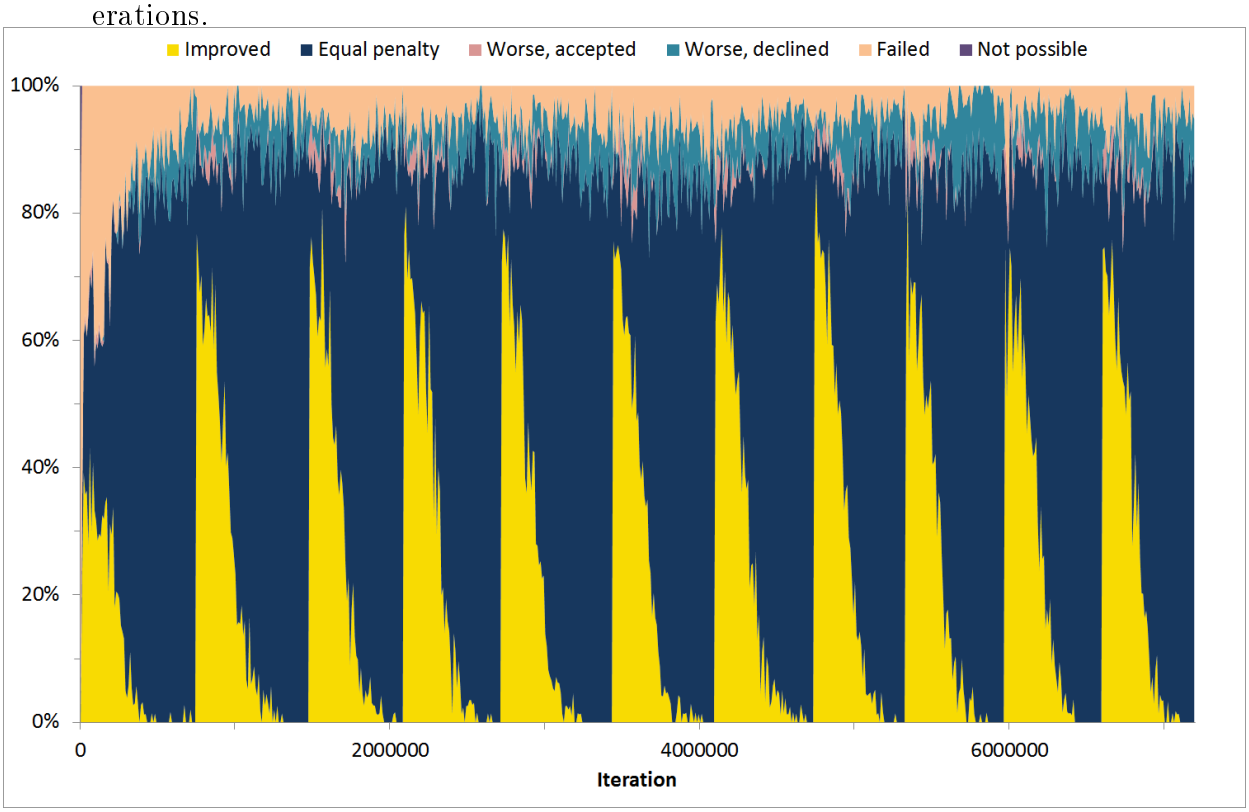


Figure 20: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Random (Sorted)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 21: The distribution (over 10,000 iterations) of the results of the *Change Day Rooms (Random)* operator, throughout the first 7.2 million iterations.



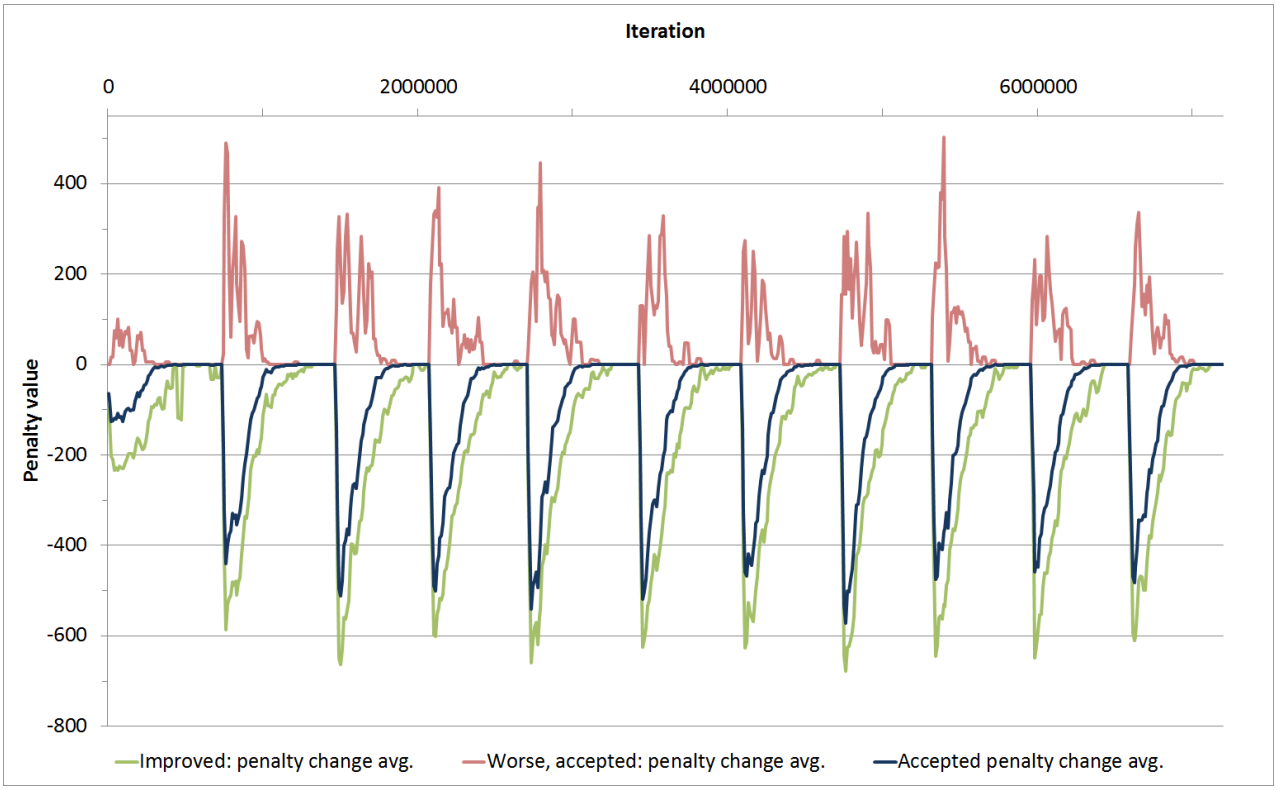
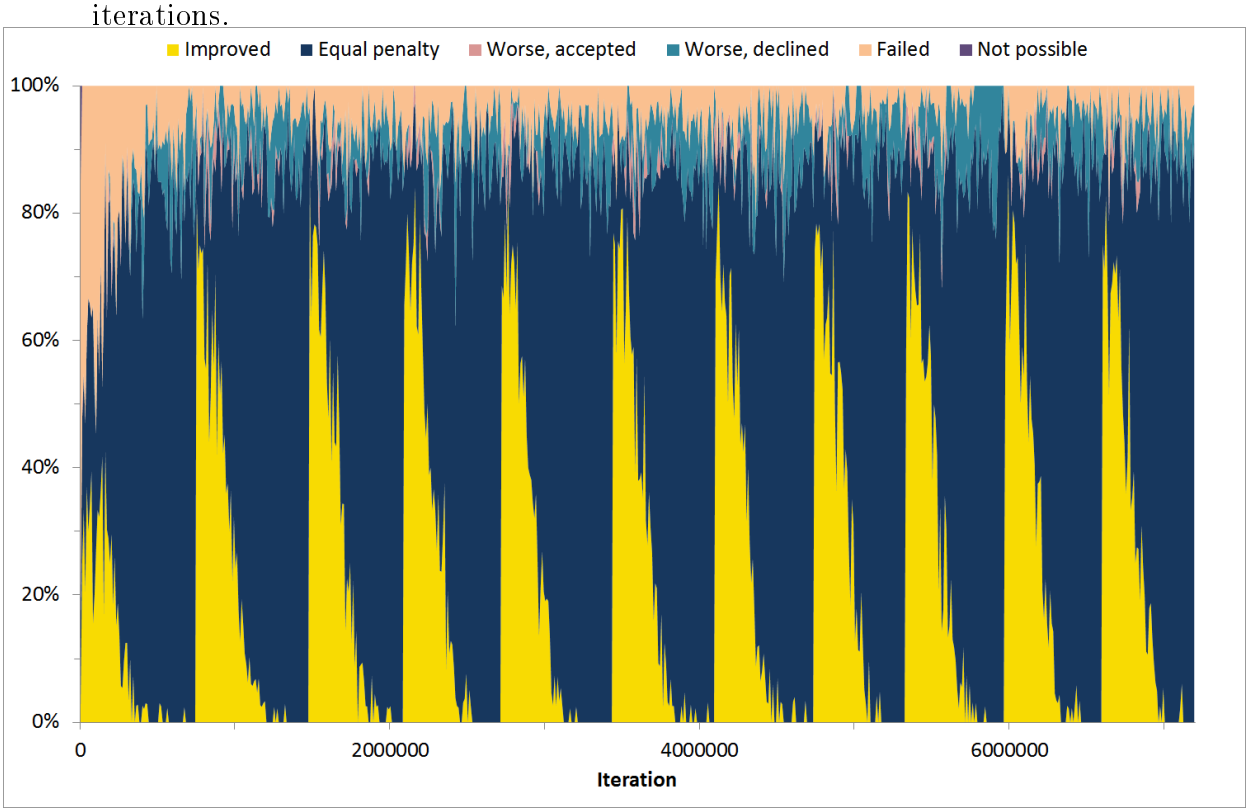


Figure 22: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Change Day Rooms (Random)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 23: The distribution (over 10,000 iterations) of the results of the *Change Day Rooms (Sort #Rooms)* operator, throughout the first 7.2 million iterations.



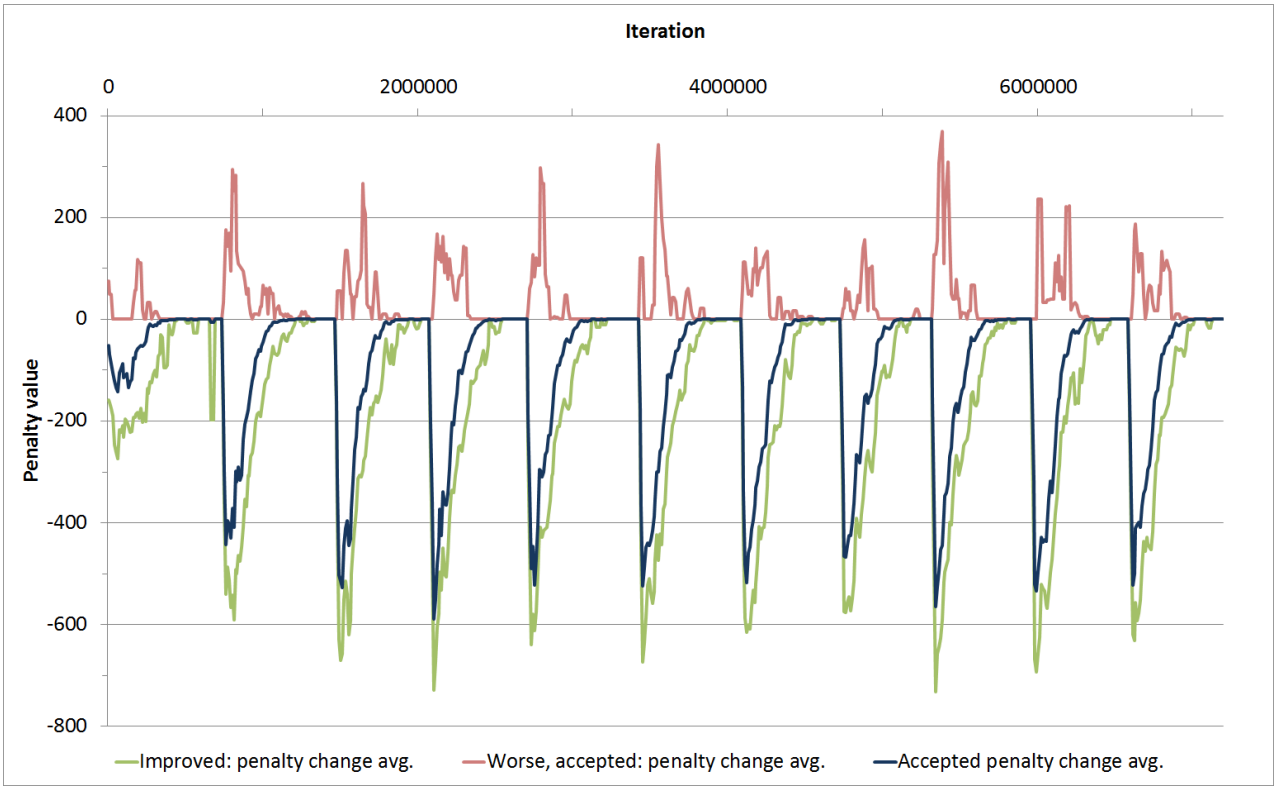
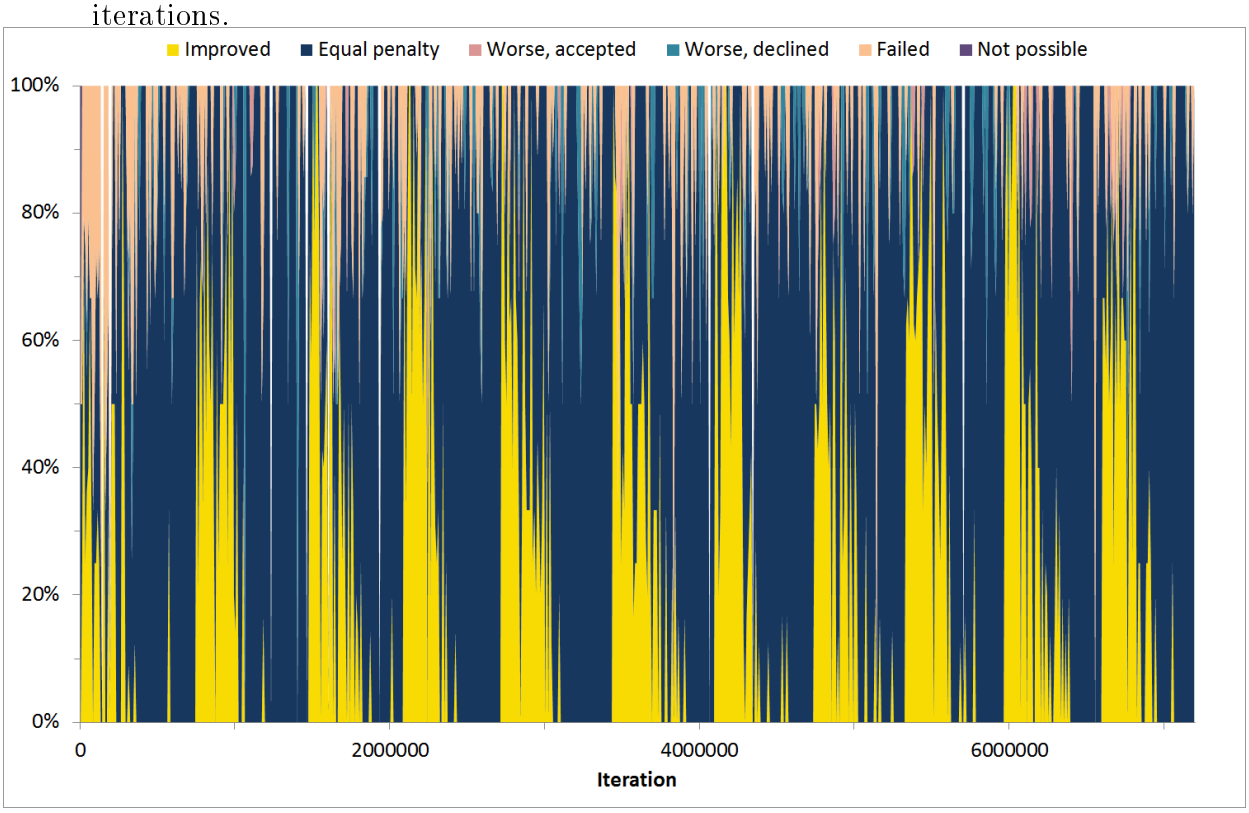


Figure 24: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Change Day Rooms (Sort #Rooms)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 25: The distribution (over 10,000 iterations) of the results of the *Change Day Rooms (Sort Duration)* operator, throughout the first 7.2 million iterations.



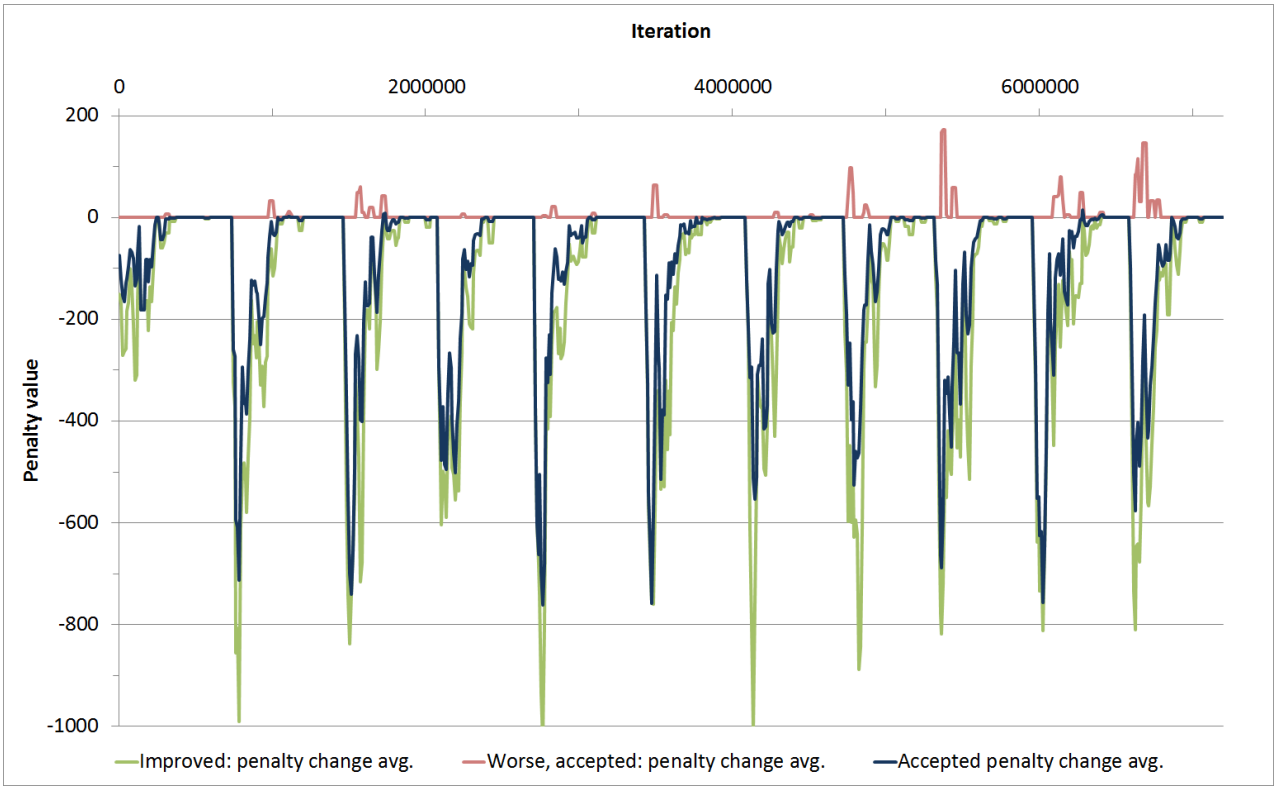
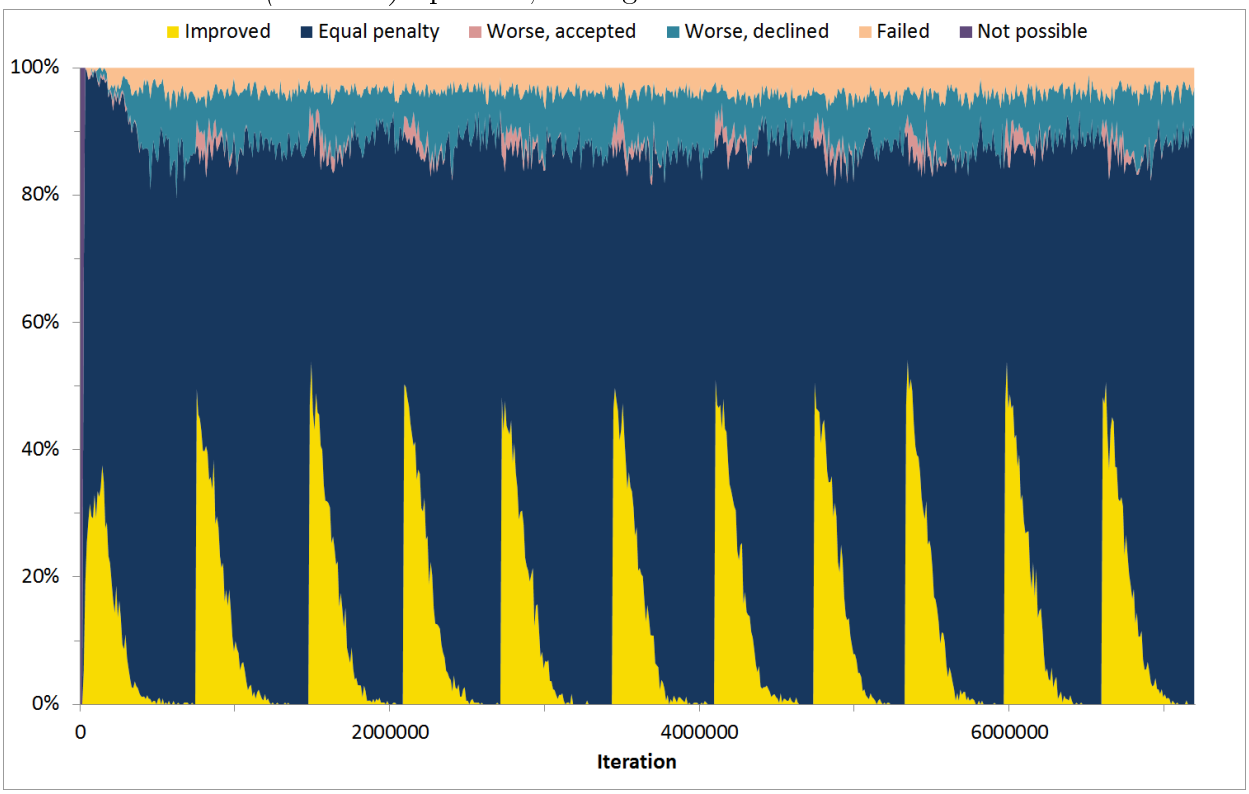


Figure 26: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Change Day Rooms (Sort Duration)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 27: The distribution (over 10,000 iterations) of the results of the *Move Best Rooms (Random)* operator, throughout the first 7.2 million iterations.



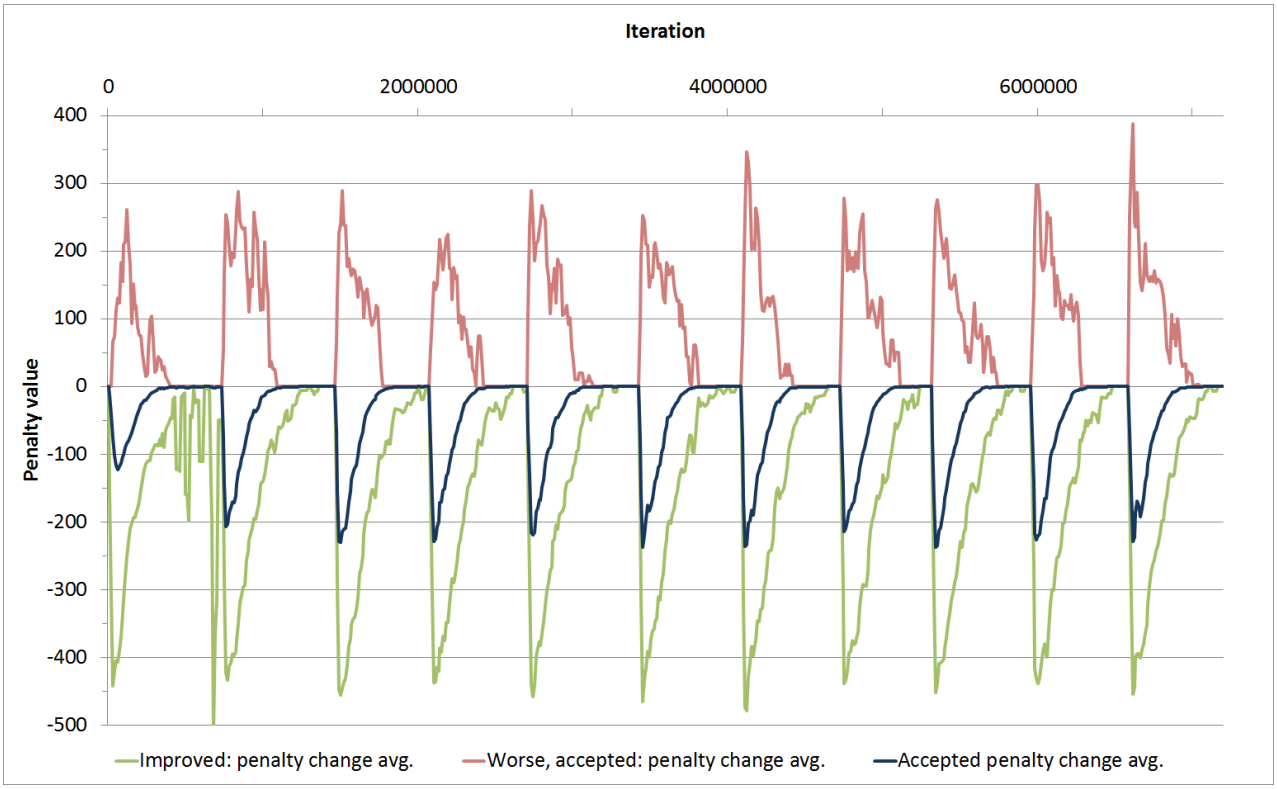
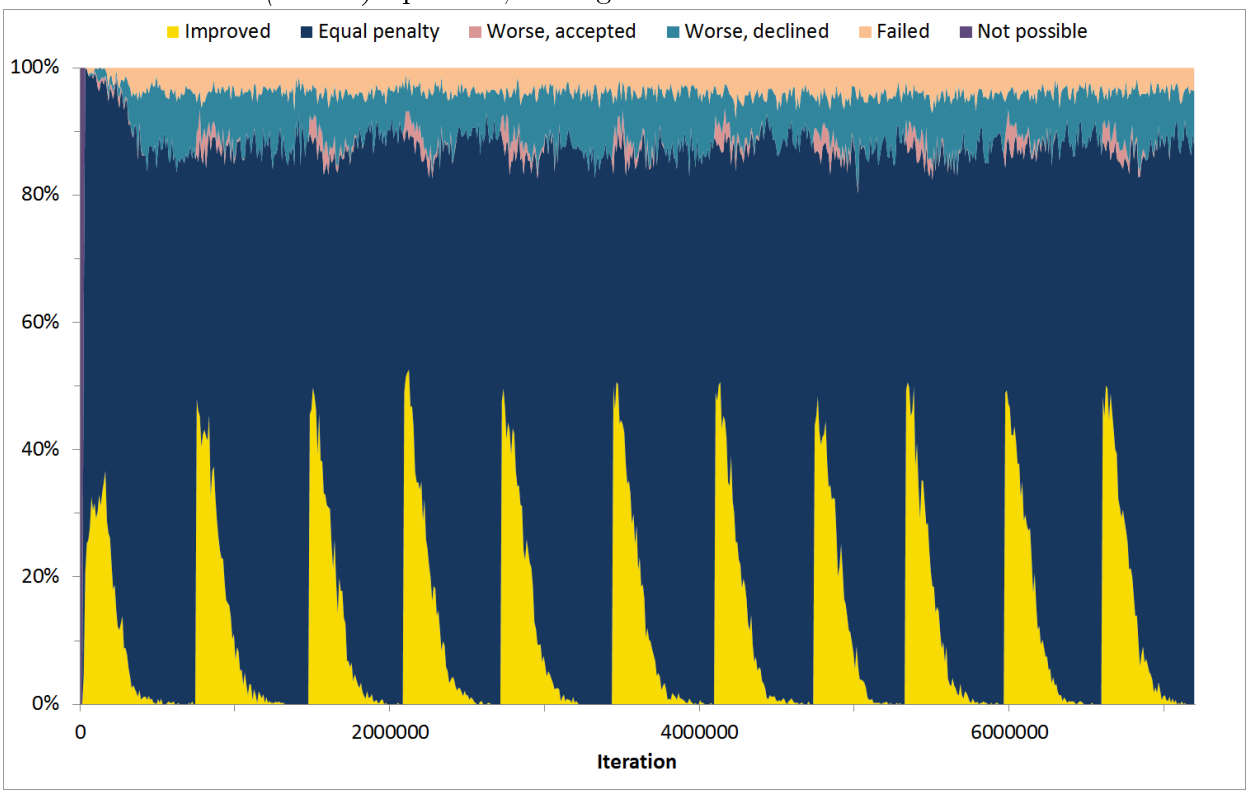


Figure 28: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Best Rooms (Random)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 29: The distribution (over 10,000 iterations) of the results of the *Move Best Rooms (Sorted)* operator, throughout the first 7.2 million iterations.



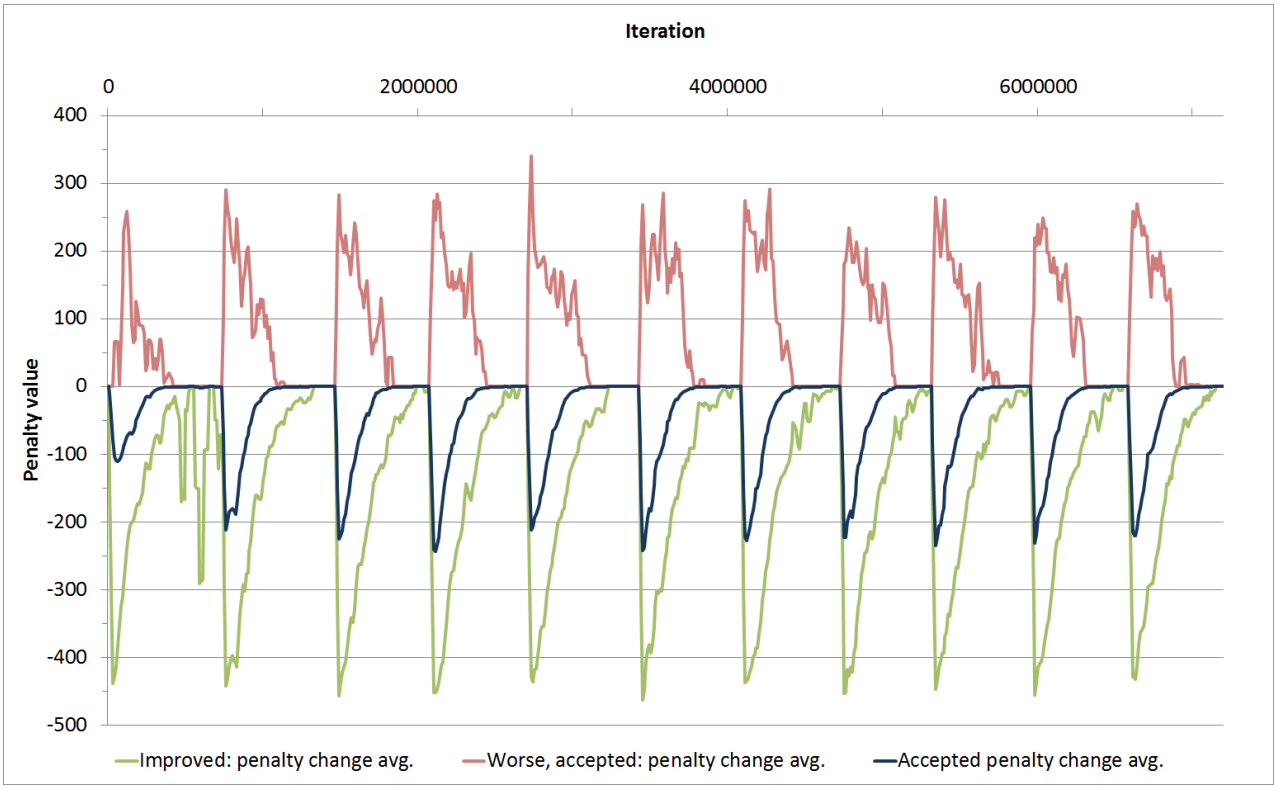


Figure 30: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Best Rooms (Sorted)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 31: The distribution (over 10,000 iterations) of the results of the *Move Period Best Rooms (Random)* operator, throughout the first 7.2 million iterations.

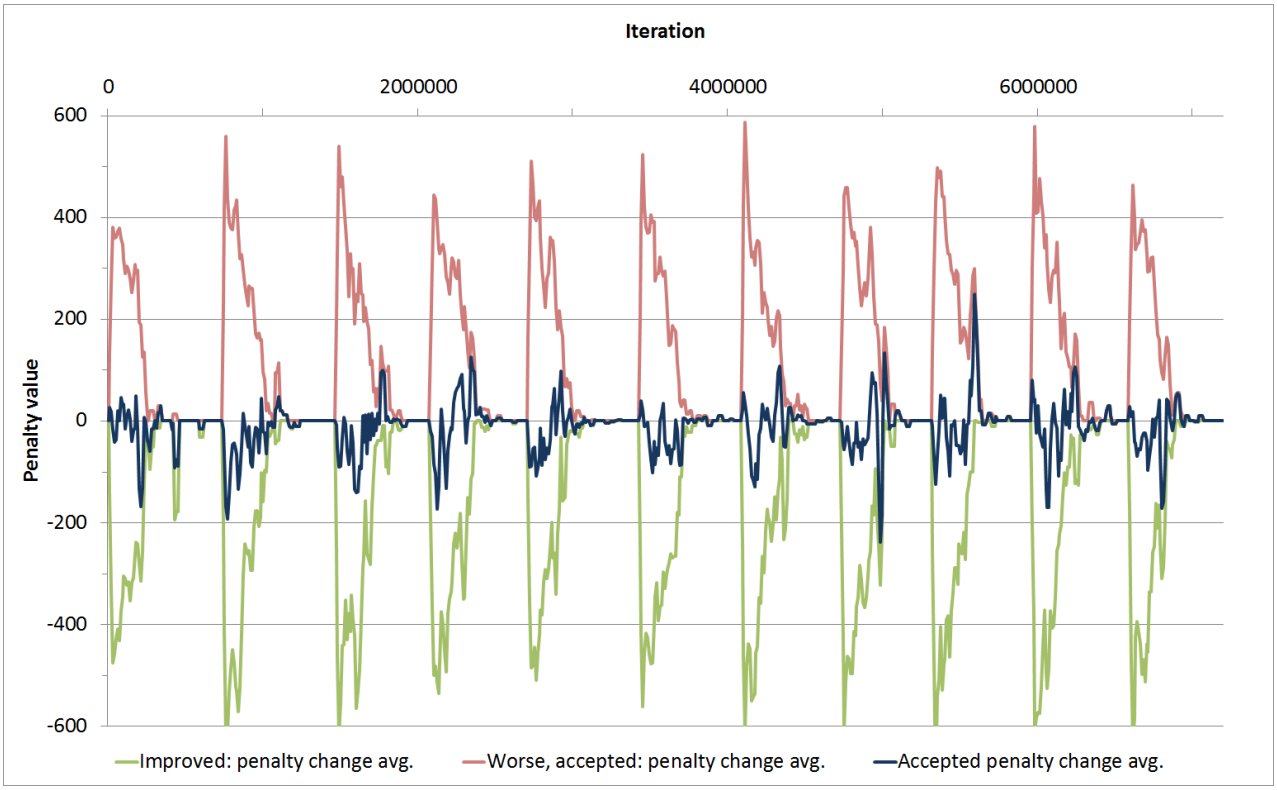
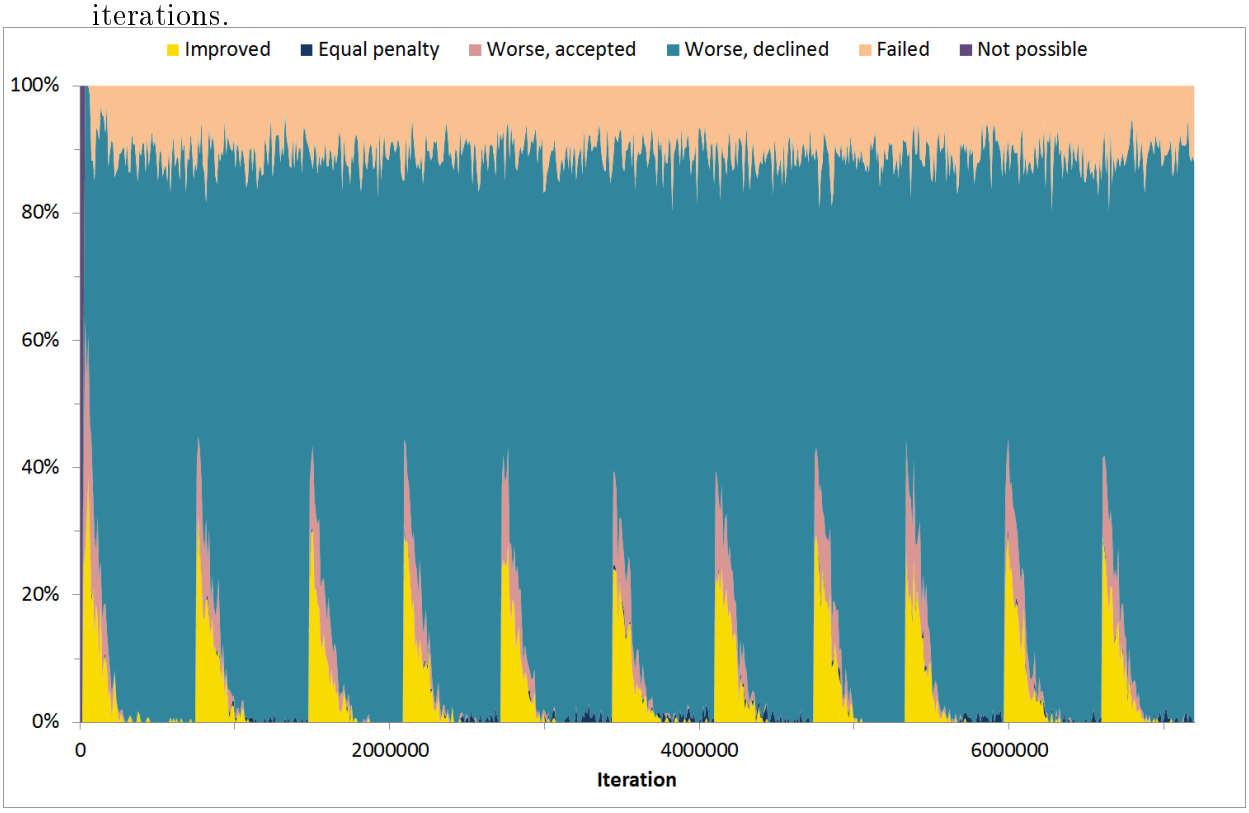


Figure 32: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Period Best Rooms (Random)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Figure 33: The distribution (over 10,000 iterations) of the results of the *Move Period Best Rooms (Sorted)* operator, throughout the first 7.2 million iterations.



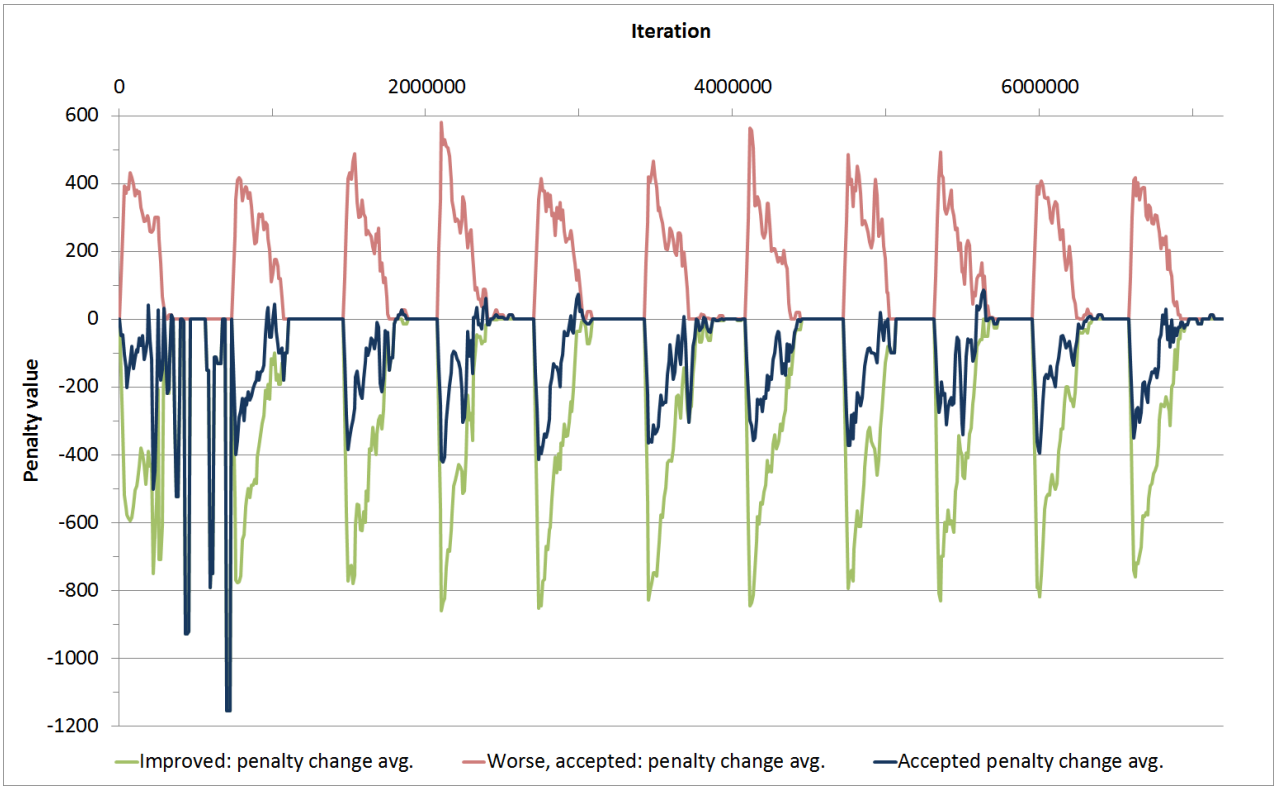


Figure 34: The average change in penalty of the timetable (over 10,000 iterations) induced by the *Move Period Best Rooms (Sorted)* operator, throughout the first 7.2 million iterations, for improving results, accepted worsening results and in total.

Table 54: The distribution of the results of all operators over all 16,530,000 iterations of the run.

Operator name (Variant)	Improved	Equal penalty	Worse, accepted	Worse, declined	Failed	Not possible
Insert (Random)	5,3%	0,0%	0,0%	0,2%	16,2%	78,2%
Insert (Sorted)	2,9%	0,0%	0,0%	0,0%	14,1%	83,0%
Move Period (Random)	1,9%	0,3%	2,5%	69,2%	26,2%	0,0%
Move Period (Sorted)	3,9%	0,4%	2,9%	66,5%	26,2%	0,0%
Move Room	5,2%	3,0%	7,9%	54,4%	29,6%	0,0%
Move Cluster	6,1%	4,2%	7,6%	52,5%	29,6%	0,0%
Move Random (Random)	2,5%	23,3%	3,8%	70,4%	0,0%	0,0%
Move Random (Sorted)	6,4%	40,4%	5,4%	47,8%	0,0%	0,0%
Change Day Rooms (Random)	23,6%	63,4%	1,0%	5,9%	6,1%	0,0%
Change Day Rooms (Sort #Rooms)	23,6%	62,9%	1,0%	6,5%	6,0%	0,0%
Change Day Rooms (Sort Duration)	24,1%	64,0%	1,2%	3,3%	7,4%	0,0%
Move Best Rooms (Random)	12,9%	74,8%	0,8%	7,6%	3,7%	0,1%
Move Best Rooms (Sorted)	12,9%	74,8%	0,9%	7,5%	3,8%	0,2%
Move Period Best Rooms (Random)	2,6%	0,3%	2,5%	83,7%	10,8%	0,1%
Move Period Best Rooms (Sorted)	4,8%	0,3%	3,1%	80,8%	10,8%	0,1%

Table 55: The number of calls to all operators and the average change in penalty induced by each operator, for improving results, accepted worsening results and in total, aggregated over all 16,530,000 iterations of the run.

Operator name (Variant)	#Calls	Improved: penalty change avg.	Worse, accepted: penalty change avg.	Accepted penalty change avg.
Insert (Random)	2900	-4380	+220	-4350
Insert (Sorted)	382	-5048	+0	-5048
Move Period (Random)	2722806	-341	+399	+75
Move Period (Sorted)	1343197	-436	+369	-88
Move Room	7998231	-150	+199	+49
Move Cluster	2198294	-162	+184	+24
Move Random (Random)	115920	-244	+319	+20
Move Random (Sorted)	151227	-303	+220	-14
Change Day Rooms (Random)	116123	-388	+177	-102
Change Day Rooms (Sort #Rooms)	60824	-394	+138	-105
Change Day Rooms (Sort Duration)	7051	-389	+143	-103
Move Best Rooms (Random)	572213	-322	+195	-45
Move Best Rooms (Sorted)	701119	-321	+196	-45
Move Period Best Rooms (Random)	304131	-434	+365	-37
Move Period Best Rooms (Sorted)	235582	-624	+350	-236

F Screenshots of the GUI Application

Figure 35: This screenshot shows the *Project* tab of the application. Here the block settings can be selected. These settings determine the number of weeks in the block, the associated calendar week numbers and the unavailable periods.

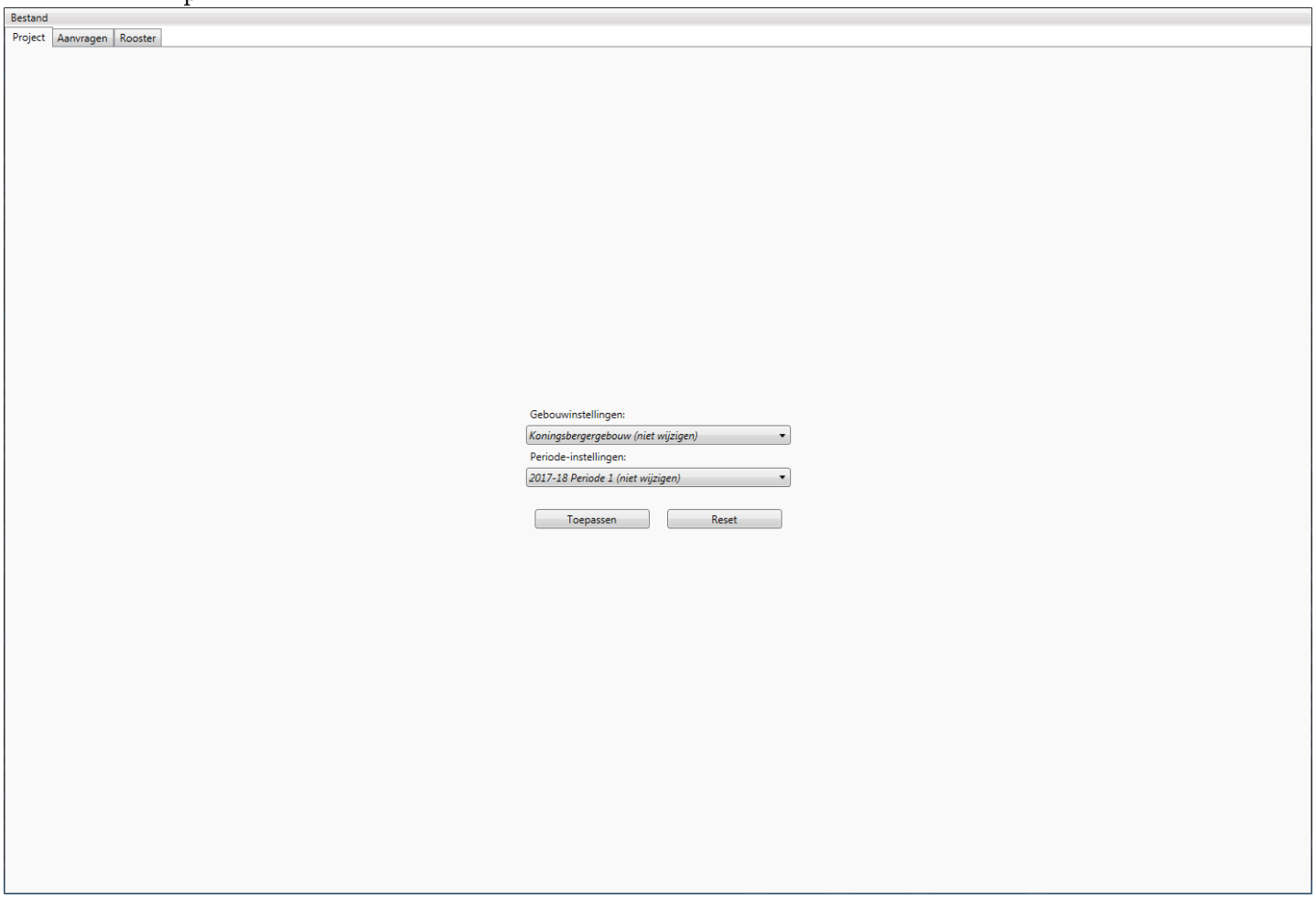


Figure 36: This image shows the *Requests* tab. Courses can be added, edited and removed and for each course, the requested meetings can be added, edited and removed. The grid shows all of the selected course's requests, positioned according to their preferred start period and duration. The bottom of the image shows the panel where the selected meeting request can be edited, in this case for a simple meeting.

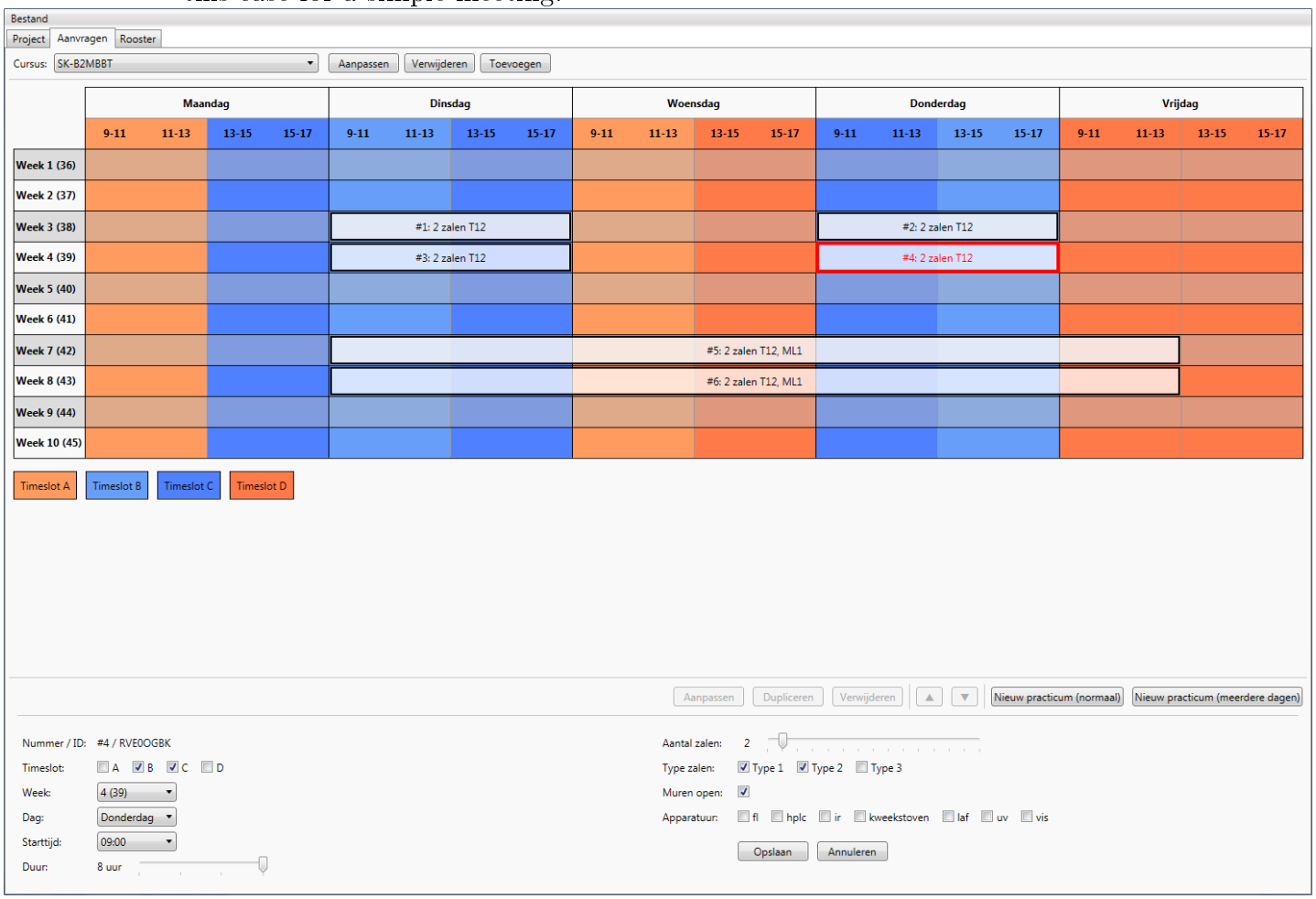


Figure 37: The *Schedule* tab of the application is shown in this figure. There are buttons to automatically generate a timetable using our algorithm and for each week a grid shows the meetings scheduled in that week, positioned according to their scheduled periods and rooms.

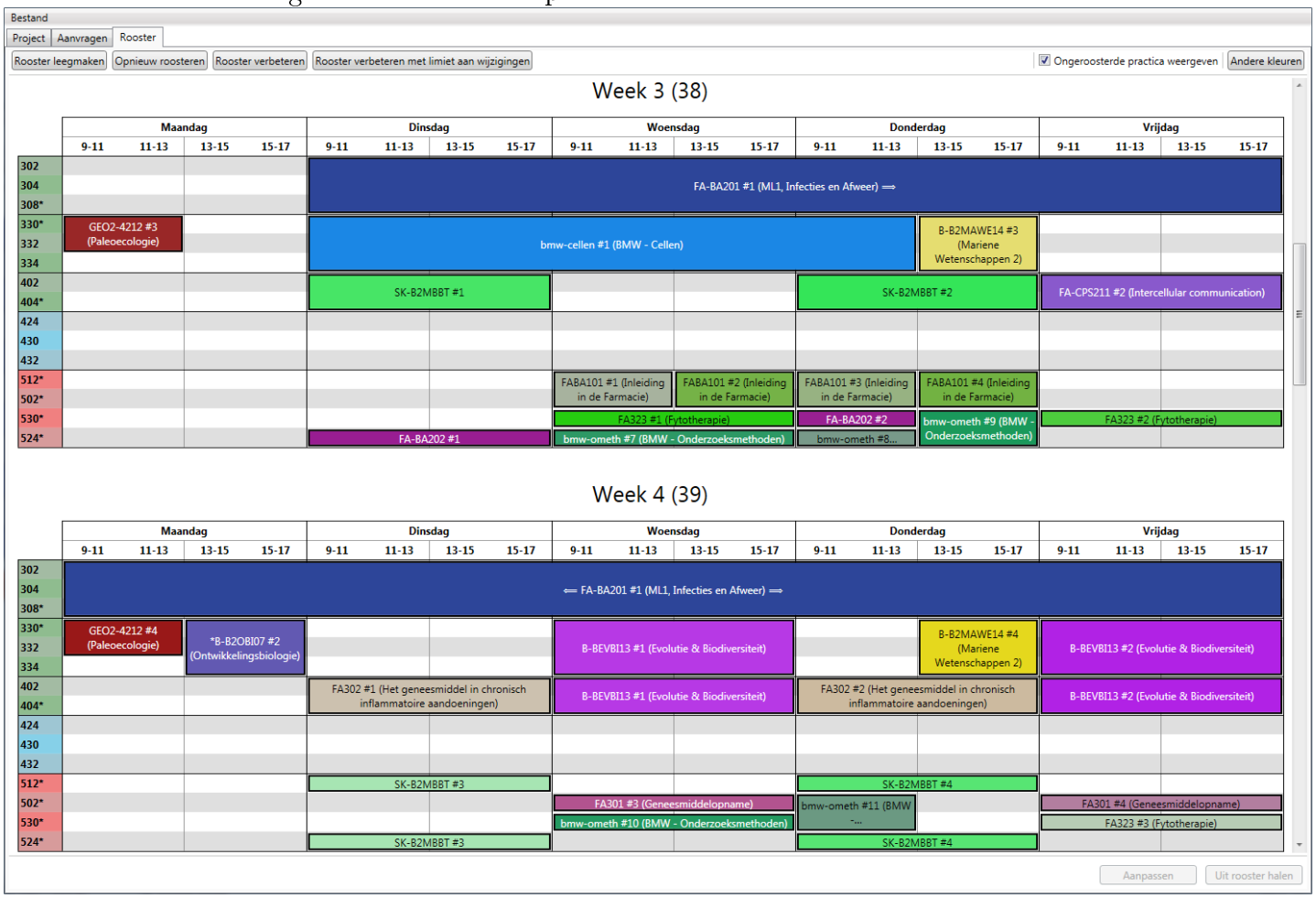


Figure 38: The is another figure of the *Schedule* tab, this time showing the panel to edit the scheduled start period and rooms of a meeting. It is also possible to fix a meeting's start period and/or rooms (see section 3.6). The right side of the panel shows information about the request.