N.J. van der Kooy

The High School Scheduling Problem: Improving Local Search & Fairness Evaluation

Master's thesis

Supervisors:
Dr. F.M. Spieksma
Drs. B.A.F.J. van Kesteren

September 19, 2017

Mathematical Institute, University of Leiden

Contents

Abstract

The High School Scheduling Problem is the operational research problem of finding an optimal timetable for students and teachers in a secondary school. It is a minimisation problem, in which we need to satisfy as many wishes as possible while guaranteeing all demands are satisfied.

The High School Scheduling Problem has been studied for over half a century, owing to the importance of schools having a good schedule and the difficulty of finding a schedule that adequately satisfies the different constraints.

This thesis studies two new approaches to optimising solutions of a problem instance. Both of these approaches are based on transformations of the problem's search space. First, locally optimal solutions are improved by changing the penalty associated with soft constraints in specific ways. Secondly, the schedule's penalty function itself is improved, aimed at shifting focus from improving the schedule as a whole to making sure individual schedules become acceptable.

1 Introduction

Like many optimisation problems, the High School Scheduling Problem (HSSP) is one that is difficult, not because it is hard to understand, but because even the search space for a 'small' problem is so large that it is not (currently) possible to efficiently[1] find a schedule - let alone show it is the best one. Indeed, the HSSP belongs to the class of $\mathcal{NP}$-complete problems[16], a class of problems for which (currently) no efficient algorithms exist.

Typically, the creation of a schedule is split into two phases: the construction of an initial solution, and optimising this solution. In this thesis we focus on the second phase, and study the optimisation of an already constructed schedule.

1.1 Overview & contribution

In Part I, the generic HSSP is described in a conceptual and theoretical way. Also, we analyse the problem's complexity, and study the types of algorithms currently in use to optimise solutions to the problem.

In Part II we describe the model of the HSSP we will be studying in this thesis, and define the problem we will study in parts III and IV.

Since there is no efficient way to find an optimal solution, current techniques are mainly based on heuristics that take a valid solution as a starting point from which to develop better solutions. However, these 'local searches' have the drawback of getting stuck in locally optimal solutions.

In Part III, we answer the question

Can we modify a local search algorithm to automatically escape local minima?

We designed a new algorithm, not currently found in existing literature, to circumvent the drawback of local search. In addition, we study its effectiveness in practice. We show that the algorithm produces results that can improve schedules stuck in local minima and that it is not slower than the local search algorithms already in use. Finally, we study optimisations to the algorithm itself, with regard to parameters that can influence how fast the algorithm finds results.

The optimisation of a scheduling problem is done based on a penalty function describing the quality of the schedule. In real-world problems such as the HSSP, an additional challenge is determining an objective function that objectively values a schedule in a way similar to how a person would subjectively value it.

In Part IV we study this penalty function itself. We answer the question:

[1]In this thesis, we understand 'efficient' to mean 'in polynomial time relative to the input'.

How can the penalty function be chosen so that it better reflects the subjective quality of a schedule?

We first propose an extended penalty function that introduces a subjective balance or 'fairness' to the schedule. In addition, we develop a second algorithm that uses this extended penalty combined with methods proposed in Part III to solve the problem defined in Part II.

Part I

High School Scheduling Problem

2 Problem description

In the HSSP we are given a number of *lessons*, each of which consists of *students* who are taught a *subject* by one or more *teachers*. The given lessons need to be fit into a *schedule*, which usually consists of a fixed number of *time slots* a day on a fixed number of days a week. In The Netherlands for example, there are often 8 or 9 timeslots per day, 5 days a week, with a total of 40 or 45 timeslots per week. This arrangement of lessons then needs to be placed in a way that is 'as good as possible'.

How 'good' a schedule is, is calculated using different constraints imposed upon the schedule. These constraints are split into *hard constraints* and *soft constraints*. Hard constraints are not allowed to be violated, while soft constraints influence the penalty of the proposed schedule. A schedule with a lower penalty is seen as the better schedule.

If a constraint is hard or soft is a decision made when developing the mathematical model behind a scheduling problem. This can vary between different implementations, as the comparison in [16, Table 4.1] shows. The constraints which are made hard in this thesis are specified in §5.

We assume that deciding which students and teachers are tied to a lesson has been done in the preceding phase of constructing an initial schedule, and is thus outside of the scope of this thesis. In addition, we assume we are provided an initial schedule where all hard constraints are satisfied. Various research has been done on this phase of timetabling. This has produced methods like clustering which have proven effective in producing initial solutions to HSSP instances within an acceptable amount of time[14].

Due to the sheer number of possible schedules,[2] even with very liberal estimates in the ratio of invalid to valid schedules, the search space is too large for any type of brute force optimisation to be remotely feasible.[3]

3 Problem modelling

3.1 Conceptual model

Scheduling problems (and more specifically HSSPs) are not only complex (see §3.4), but also hard to define generically. This is due to how the definition of the

[2]A typically sized Dutch secondary school of between 1000 and 2000 students might have around $1500-2500$ lessons a week which needs to planned in 40 timeslots. A naive calculation gives us that there are over $1.32 \cdot 10^{3204}$ ways to place 2000 lessons into 40 timeslots. Picking the best placement randomly, assuming there is one 'best' placement, is less likely than that in a school of 1250 students every student has their birthday on January 1st.

[3]In chess, which has currently not been solved, there are around 10^{120} possible game variations - a factor 10^{3084} less than in our problem.

problem changes from country to country and even within countries, with local requirements sometimes dramatically changing the constraints within the model. Work exists in formulating a generic model that could be used internationally. An example of this is the XHSTT format [6], and some research has been done into optimising these kinds of international formats [5].

In this model, we assume that the provided lessons are fixed with regard to the teachers and students assigned to these lessons. In other words, we assume that we do not try to optimise schedules by changing which students are grouped together in a class. Allowing this would mean we create many more variables in our model, making the problem more complex.

Note that fixing lesson assignments is a reasonable decision to make: in real-life examples, this is often done when changes need to be made to the schedule halfway through a school year and we do not wish to break apart classes. In addition, a lot of optimisation with regard to class assignment is usually done prior to constructing the first schedule.

We also assume that assigning classes to rooms is done after the optimisation phase and thus not something we need to take into account[4].

The resulting structure of a lesson is shown in Figure 1.

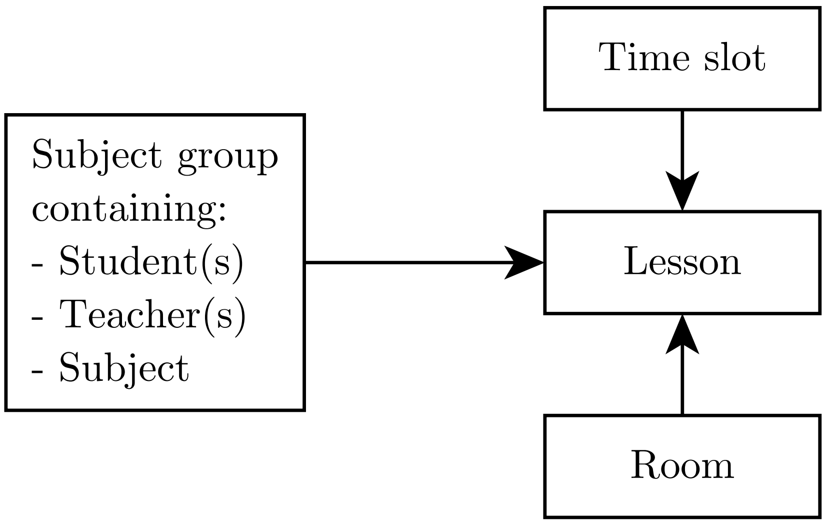


Figure 1: A visual representation of the structure of a lesson

[4]As will be explained in §5.2, in the problems studied in this thesis we do make use of soft constraints in the optimisation phase which prevent problems in the room assignment phase.

3.2 Technical model

3.2.1 Assumptions

There are several assumptions to be made on scheduling in Dutch secondary schools in general. In our model, the following will be taken into account:

1. While students are assumed to be available during every timeslot, teachers have a lot of freedom in determining which hours they prefer to teach;
2. A schedule is assumed to have a period of a week (i.e., a lesson which occurs during the third time slot on a Monday always occurs on the third time slot on a Monday). In practice, some weeks will have differing schedules due to irregular events or exams taking place, but we disregard these cases.

3.2.2 Definitions

We base our model on a number of interrelated sets, as defined in Table 1. Note that we do not define our model as an explicit optimisation problem considering every possible constraint. This has been attempted in the literature ([8], [10]), but formulations considering realistic constraints turn out to be impractical for use in heuristic solvers.

Note that in this section and the rest of this thesis, we make use of the concept of a multiset. A *multiset* is a generalisation of a set, in that it can contain the same element multiple times. We use the notation $\{\{a, a, b\}\}$

S	The set of all students
V	The set of all subjects. Note that in this model, a subject is implicitly tied to a school year and a level. Thus, German given to 4VWO is a different subject than German given to 5HAVO.[a]
T	The set of all teachers.
G	The set of all subject groups, which is a three-tuple of a subject with at least one teacher or student.[b] Thus, $G = \{(\mathcal{S}, v, \mathcal{T}) \mid \mathcal{S} \subseteq S, v \in V, \mathcal{T} \subseteq T, \mathcal{S} \cup \mathcal{T} \neq \emptyset\}$.
M	The set of all time slots.
L	The set of lessons, which is a multiset of subject groups. A subject group $g \in G$ exists in L exactly once for each time it is given during the week.

[a] The Dutch secondary school system has levels aimed at different levels of higher education: the prevocational VMBO which lasts four years, general secondary education HAVO lasting five years, and pre-university education VWO which lasts six years.

[b] It is possible to have a subject group which has either no students (like a meeting) or no teachers (like a self-study session). The first situation in particular is something that often occurs in practice.

Table 1: Definitions of sets used

Definition 3.1 (Schedule)**.** A *schedule* is the multiset[5] of lessons L whereby each lesson $l \in L$ is assigned a timeslot $m_l \in M$. Intuitively, this means we take the lessons from L, and create a schedule by scheduling these lessons into a timeslot.

Example 3.2. For a set $G = \{g_1, g_2, g_3\}$, we might have

$$L = \{\{g_1, g_1, g_2, g_2, g_2, g_2, g_3\}\}.$$

We could then have

$$\mathcal{T} = \{\{(g_1, a), (g_1, c), (g_2, a), (g_2, b), (g_2, c), (g_2, d), (g_3, c)\}\}$$

for $a, b, c, d \in M$.

Note that, according to Definition 3.1, no constraints on the schedule are taken into account. Two lessons of the same subject group could in theory even be placed simultaneously. In practice, constraints usually prohibit this.

Often we do not study the schedule as a whole, but instead consider a *restricted schedule.*

Definition 3.3 (Restricted schedule)**.** A *restricted schedule* is a subset of a schedule $\mathcal{T}$.

[5] A (sub)set based on a multiset is again a multiset

Restricted schedules are used to, for example, display the schedules for specific students, teachers and rooms. The schedule for student $s \in S$ would be $\{((\mathcal{S}, v, \mathcal{T}), m) \in \mathcal{T} | s \in \mathcal{S}\}$: all lessons where the set of students contains s. All other entity specific schedules can be defined in a similar way.

Definition 3.4. For an $a \in A$, where $A \in \{S, T, G, M\}$, define $\mathcal{T}_A(a)$ as the restricted schedule $\mathcal{T}$ where lessons contain element $a \in A$.

Note that the above definition does not include defining restricted schedules based on subjects $v \in V$. Although the definition could be extended easily to include it, such a restriction of a schedule is not usually of interest and is not used in this thesis either.

Example 3.5. For $g \in G$, $\mathcal{T}_G(g)$ is the restricted schedule of $\mathcal{T}$ containing $g \in G$: the lessons for subject group g.

Example 3.6. For $t \in T$, $\mathcal{T}_T(t)$ is the restricted schedule of $\mathcal{T}$ containing $t \in T$: the lessons taught by teacher t.

As we will see in coming sections, a schedule is tied to constraints, where the hard constraints make a schedule invalid if they are violated. We thus define a function to check the *validity* of a (restricted) schedule.

Definition 3.7. Let $\texttt{VALID} : \mathcal{P}(L \times M) \to \{0, 1\}$. $\texttt{VALID}$ is the *validity function* of a (restricted) schedule $\mathcal{T} \in \mathcal{P}(L \times M)$, whereby $\texttt{VALID}(\mathcal{T}) = 1$ means we have a valid (restricted) schedule (no hard constraints are broken), while $\texttt{VALID}(\mathcal{T}) = 0$ means it is invalid.

With no constraints imposed, we have $\texttt{VALID}(\mathcal{T}) = 1$ for all $\mathcal{T} \in \mathcal{P}(L \times M)$. The constraints introduced in later sections give us cases where $\texttt{VALID}(\mathcal{T}) = 0$.

3.2.3 Hard constraints

Hard constraints are those constraints that must be satisfied by a schedule to be a valid solution, as also defined by Definition 3.7. However, there are no universal hard constraints which are present in every model.[6] We thus do not define any hard constraints here. In the model we will study in this thesis there *are* several hard constraints which will be imposed. These constraints are explained in §5.

3.2.4 Evaluation & Soft constraints

In most HSSPs, only (some subset of) the requirements are modelled as hard constraints. However, especially as problems get larger, there are usually preferences that do not change the validity of the schedule, but do make one schedule

[6]Even constraints such as every student only being scheduled into one lesson at a time are not seen as a hard constraint in every HSSP model.

better than the other. These preferences are known as soft constraints. Examples of common soft constraints are:

- Minimising the amount of idle time[7] in teacher and student schedules, especially when it leads to gaps;
- The preference of teachers for a day off;
- The distribution of a student's lessons (making sure that there is roughly the same amount of lessons each day for example).

Assuming a schedule is `VALID` by Definition 3.7, we still need a way to compare these schedules based on their soft constraints. This is done using a penalty function, which determines the quality of a proposed schedule based on the set $\mathcal{C}$ of soft constraints.

The exact implementation of this function is generally quite specific to the choices made in the algorithms used and the soft constraints implemented (an example is shown in [11]). However, the general approach is to define some penalty for each constraint violated, and taking the sum of each of these penalties. Then, the schedule with the lowest overall penalty is taken to be the best schedule.

Throughout this thesis, we will be using $\mathcal{C}$ as the notation for the set of all possible soft constraints.

Definition 3.8. $\Phi : \mathcal{P}(L \times M) \times \mathcal{P}(\mathcal{C}) \to \mathbb{R}_{\geq 0}$ is the *penalty function* of a schedule $\mathcal{T} \in \mathcal{P}(L \times M)$ subject to soft constraints $C \subseteq \mathcal{C}$. Note that $\mathcal{T}$ must be a schedule as described by Definition 3.1.

Note that this function does not have any requirement like linearity, and as we will see it indeed often does not have these characteristics.

The details of how the model used in this thesis handles soft constraints in the penalty function is discussed in Part II (§5.2 and §5.4).

3.3 Problem definition (generic model)

Given the notation and definitions introduced in §3.2, we can now define the problem we are trying to solve as a minimisation problem.

$$\begin{array}{lll} \text{Given} & L, \mathcal{C}, M & \\ \text{minimise} & \Phi(\mathcal{T}, \mathcal{C}) & \\ \text{subject to} & \texttt{VALID}(\mathcal{T}) = 1, & \\ & (l, m_l) \in \mathcal{T} & l \in L, m \in M. \end{array}$$

In this definition, the soft constraints to which the schedule is subject are included by the minimisation of $\Phi(\mathcal{T}, \mathcal{C})$ while the hard constraints are taken care

[7] A timeslot where no lesson is planned.

of by the requirement that $\texttt{VALID}(\mathcal{T}) = 1$. The final requirement in the minimisation problem is a reflection of Definition 3.1 which requires every lesson in L to be included in the timetable.

Recall that we assume to be given an initial schedule $\mathcal{T}$ such that $\texttt{VALID}(\mathcal{T}) = 1$, so finding an initial valid solution is not a part of our problem.

3.4 Complexity

In [16], different sub-problems of the HSSP is shown to be $\mathcal{NP}$-complete. Each of these sub-problems are shown to be a special case of the HSSP, implying that the HSSP itself is $\mathcal{NP}$-complete. These sub-problems include restrictions placed on teacher availability [16, §3.4], freedom of choice in a student's curriculum [16, §3.5], educational requirements (like distribution of subjects over separate days) [16, §3.6], room assignment given room types and lessons scheduled in blocks [16, §3.7] and time slot assignments taking into account blocks and teacher availability [16, §3.8].

All of these difficulties are generally ones which are encountered in the Dutch HSSP and in particular in the problem instance discussed in this thesis. We can thus speak of the HSSP in general as being $\mathcal{NP}$-complete.

In general, the HSSP is a problem which is not linear. Examples of nonlinear constraints are given in §3.4.1.

Unlike linear programming problems, where the Simplex algorithm can in the average case solve LP problems in polynomial time, we do not (yet) know of an algorithm that can solve the HSSP efficiently. Since the problem is in essence based on the placement of lessons into discrete time slots, the problem can be seen as an (Nonlinear) Integer Programming Problem, for which currently no general polynomial algorithms exist.

Any method of finding good solutions to the HSSP is currently based on heuristic approaches, and provided $\mathcal{P} \neq \mathcal{NP}$ [7] any algorithm which guarantees to calculate the best solution brings with it the drawback of not being polynomial-time executable. In practice, this means that for any real-life problem it is not practically possible to know if a valid solution is the best possible with regard to its penalty, and given the extremely large search space, it is in fact likely that better solutions could be found given more computational time.

3.4.1 Nonlinearity of constraints

As described, the HSSP is a problem with a large amount of constraints. The types of constraints we are dealing with are of course very important. For example, if they were all linear we could see the problem as a Linear Programming Problem, into which a lot of research has already been done. The constraints used in a real world model are rarely linear. What follows are a few examples based on the specific model used in Part II to illustrate this.

Example 3.9 (Dependencies between lessons)**.** Often, the penalty of placing a lesson in a certain timeslot will depend on other lessons which have already been placed.

For instance, we might have the situation where a subject is taught for two hours a week, and the preference is for these two hours to be scheduled in a block (which often is the case for practical subjects like Physical Education or Music). Thus, moving a lesson from one time slot to another will have a different effect on the quality of the schedule depending on the placement of the other lessons of that subject for a particular group.

Another example of this is that for instance we might want at most one lesson a week during the last time slot. Thus, if a lesson being placed in a certain timeslot results in a penalty also depends on other lessons which might be placed in this timeslot. In addition, the same constraint might then cause a penalty for one student while not doing so for another student.

Example 3.10 (Severity of multiple constraint violations)**.** It is often the case that the amount of times a constraint is violated does not linearly increase the penalty on the schedule. For example, it might be the case that four gaps per week in someone's schedule is acceptable, more than four but less than seven brings about a certain penalty per additional gap, while any gap above that increases the penalty dramatically.

4 Current state of algorithms

As alluded to in §3.4, the HSSP is a problem where current solving methods are based on heuristic approaches and search algorithms based on optimisations. Modelling the problem as a (Mixed) Integer Programming Problem has been attempted, but this has mostly been unsuccessful due to the large amount of variables and constraints any realistic example has, and heuristic methods have proven themselves to be more effective in these cases[2].

Ever since research into automated timetabling began around 1963[13], numerous different approaches into finding an optimal solution have been proposed and researched. The EURO Working Group on Automated Timetabling (WATT) maintains a (non-exhaustive) list of research on educational timetabling since around 1996, with multiple search algorithms being proposed[9]. In addition, multiple surveys existing on the state of automated timetabling research ([12], [13]). These surveys show that numerous different methods have been tried over the years, most notably *Tabu search*, *evolutionary algorithms*, *simulated annealing* and several hybrid approaches that combine these mentioned methods with more local approaches like the *Hill Climbing algorithm.*

In the following subsections, these different algorithms are globally described, along with their experienced advantages and disadvantages when used to solve HSSPs.

4.1 Hill Climbing

The Hill Climbing algorithm is based on improving a solution by performing a single change (in the case of the HSSP: moving a single lesson), and accepting the change if the schedule remains valid and the quality of the schedule improves. Note that in real-life situations, a literal 'single change' often results in an invalid schedule due to broken hard constraints. §6 discusses how this is handled in the problem studied.

This type of algorithm is one that is extremely effective in finding an optimal solution in its own neighbourhood: a local minimum. For a problem like the HSSP, the Hill Climbing algorithm has the advantage of being relatively fast. However, the obvious disadvantage is that in a high dimensional problem with as many initial configurations as the HSSP, it is extremely unlikely that a locally optimal solution we find, turns out to be a global optimum. In fact, in practice it turns out that optima found by a Hill Climbing algorithm can almost always be improved by further calculations, or by an eventual new iteration of the Hill Climbing algorithm by restarting in a new initial configuration. In the literature, the algorithm is thus almost exclusively used *in combination* with other search methods[12].

4.2 Genetic algorithms

Genetic or evolutionary algorithms are one of the most popular ways to solve and optimise an HSSP in certain situations. These algorithms work with a population of solutions, and create new solutions from the best quality candidates from the original population. By crossing two good solutions, the aim is to create offspring that is of even better quality, and random mutations introduced into the new solutions are used to increase the diversity and to simultaneously try to leave local optima (which is the problem encountered in the implementation of the Hill Climbing algorithms). Thus, as the name suggests, the design of the algorithm is based on the biological evolutionary model.

One large drawback of evolutionary approaches is the significant runtime needed to find solutions[3]. While they work from a theoretical standpoint they are often not suitable to find feasible solutions to realistically sized problems.

A second problem is that they work best in situations where all constraints are modelled as 'soft', and all suggested schedules are thus `VALID`. When this is not the case, the crossing of two solutions often results in an invalid schedule, making the method impractical in situations with a lot of hard constraints.

4.3 Tabu search

Like genetic algorithms, Tabu search is a very popular search method within HSSPs. As [12, table 4] shows, this is a method that is used both on its own and in combination with other methods in a hybrid search algorithm, and it has additionally been the subject of several comparative studies. Tabu search is a variation of the Hill Climbing algorithm, which attempts to solve the problem of getting stuck in local optimal solutions by allowing solutions to get worse under certain conditions and adding the previous solution to the so-called 'Tabu list' in order to prevent the algorithm from returning to that solution. Even though Tabu search enables a potential solution to get out of a local minimum, in practise approaches using Tabu search also suffer from exceedingly high computation times[1]. The reason for this is that the search space in even 'simple' cases is so large, that the number of banned positions needed to force a solution out of a local optimum is impractically large.

4.4 Simulated annealing

One last search method often used in solving the HSSP is known as simulated annealing. Inspired by annealing in metallurgy, simulated annealing is a probabilistic process that is used to estimate the global optimum of some objective function. The algorithm works, like Hill Climbing and Tabu Search, by finding a neighbour of the current solution. When the proposed solution is better, it is accepted, but a solution that is worse is accepted based on some probability that gets lower as the difference in quality gets higher and — more importantly — as the number of completed iterations increases. In other words, as the algorithm progresses, the amount that a schedule is allowed to get worse decreases. By initially allowing large changes in the schedule and gradually restricting this, the method is ideal for estimating a global optimum in a large search space. Although this algorithm is not used or studied in this thesis, research does exist on the use of this method([2], [4], [17]).

Part II

High School Scheduling at Zermelo

In this thesis, instead of studying the most general model as described in Part I, we work on the model and constraints used at Zermelo Roostermakers, a Dutch company specialising in software for the Dutch secondary school market. As such, their model is tailored specifically to the peculiarities of the Dutch secondary school system, which brings with it constraints and specifications, when compared to the generic model.

In this part, the specification of the model studied will be described and current approaches to schedule optimisation are discussed.

5 Specification of the model

The model described in Part I was a generic one, and did not yet contain the details of the problem we will actually be studying. The following chapter makes the model of the HSSP more specific and constrained. After this chapter, any references to the HSSP will be assumed to respect both the constraints defined in Part I as well as those defined in this chapter.

5.1 Hard constraints

Recall from §3.2.3 that in the generic model no hard constraints are enforced. In practise however, there are many constraints which could be modelled as being 'hard'.

In this case, the decision was made to only model the possibility of a teacher or student physically fulfilling his schedule as a hard constraint. In other words, the only schedules explicitly disallowed are those where a teacher and/or a student is expected to be at two lessons simultaneously.

Constraint 5.1 (No overlapping of lessons for teachers or students)**.** For every teacher $t \in T$, we demand that at any given moment, personal schedule $\mathcal{T}_T(t)$ of teacher t must contain no more than one lesson at a time. In other words, for any $m \in M$, we must have that

$$|\{\{(g', m') \in \mathcal{T}_T(t) | m' = m\}\}| \leq 1.$$

In the same way, for any student $s \in S$, we must have

$$|\{\{(g', m') \in \mathcal{T}_S(s) | m' = m\}\}| \leq 1$$

for all $m \in M$. If not, the schedule is invalid.

Notice that this constraint also ensures that the same subject group cannot have two lessons planned at the same moment: any schedule such that $g \in G$ exist in the schedule more than once during timeslot $m \in M$ cannot be valid, since then the students and teachers tied to g (of which there is at least one according to

Table 1) would have to attend a lesson of subject group g multiple times at the same time.

5.2 Soft constraints

Unlike most models discussed in the literature, the model we will be studying allows for an enormous amount of different soft constraints. Moreover, it allows the scheduler to define the weight of these different constraints and whether to use them at all. These soft constraints are split into four distinct categories:

1. *Educational.* These are constraints to do with how subject groups are scheduled. For example, preference can be given to subject being taught in blocks or not, to the spread of the same subject group being taught over the week, and to how the lessons are divided over the day (whether they are taught in the morning or in the afternoon for example).
2. *Teachers.* These involve constraints based on teacher availability and their preferences for days off. Other constraints commonly found in this category are limiting the amount of gaps the teacher has in his schedule or the variation between his busiest and least busy day.
3. *Students.* For students, constraints commonly used include limiting the number of gaps in the schedule (especially for lower grades), and limiting the variation between the length of days (to make sure one day does not contain significantly more lessons than another day).
4. *"Counting groups".* As described in §1, the assigning of lessons to rooms happens as a separate process after the scheduling of lessons. However, this means that care has to be taken that lessons requiring similar types of classroom will not cause conflicts in that process. This is where the concept of "Counting groups" (Dutch: telgroepen) is used. A common example is that all subjects for Physical Education are grouped into a counting group, and this will penalise the schedule if more of these lessons are scheduled simultaneously than the number of available gymnasia.

Next to these categories of constraints, there are two other categories that can be taken into account. The first is *placements.* These have to do with constraints like the number of students assigned to subject groups and constraints ensuring that classes do not get too large or too small. However, in this case we do not take these constraints into account, since we do not allow students to be placed into different subject groups. The sixth and last category relates to classrooms. For example, a teacher can indicate his preference for classrooms, or a penalty can be placed on a change in classrooms during a block. However, these constraints are also out of scope for this research, since we are not concerned with placement of lessons in classrooms.

Note that the mentioned examples are non-exhaustive, and almost all possible preferences can be modelled as a soft constraint.

Definition 5.2. For every category described above, we use an *evaluator* to refer to the elements of the category to which we apply the soft constraint.

Example 5.3. An individual teacher is an evaluator of the Teachers category.

Example 5.4. The set of all Physical Education lessons is an evaluator of the Counting Groups category.

Definition 5.5. A *soft constraint* is a rule that can be applied to a specific evaluator in a schedule, whereby the total penalisation of a schedule increases if the rule is broken.

5.3 Softened hard constraints

As mentioned in §5.1, the model studied uses only the bare minimum of hard constraints. In practice of course, before a schedule is accepted by a school, there are many constraints that must absolutely be satisfied, and in this sense are hard constraints. An example of this is that if a teacher is absolutely not available in a certain timeslot, there cannot exist any lessons taught by this teacher during this particular timeslot. However, experience at Zermelo has taught it is most effective to have as little hard constraints as possible.

This means that in this model we have soft constraints that are in fact demands made on a schedule, and as such a schedule violating these constraints cannot be accepted. These types of constraints are what we refer to as softened hard constraints.

The obvious question is what the reason behind this is. This is due to the fact that in practise it turns out that restricting too many options in the schedule makes it a lot harder to create a `VALID` schedule, and it turns out to often be easier to optimise a schedule which violates some of these "softened hard constraints" than finding a valid schedule from scratch. In addition, local optimisation, once a valid schedule has been found, is impeded when working with too many hard constraints: with few hard constraints it remains possible to find a 'path' between solutions, which are reached by small incremental changes to the current schedule. On the other hand, with a lot of hard constraints, it becomes much harder to find `VALID` solutions in the neighbourhood of a given schedule. Analysing the effectiveness of this approach is outside the scope of this thesis.

Another motivation is that even 'demands' sometimes have flexibility in them. If, for example, a teacher is not available at a certain time but a schedule with this softened hard constraint is consistently much better in terms of its penalty, then the scheduler could try and convince the teacher to switch around his day off — a practical solution that would never have been considered if the schedule had never been shown.

In practise, the softening of hard constraints means that the penalty associated with violating one of these constraints is made significantly higher than the penalty for violating any of the actual soft constraints from §5.2. In this model for example, violated demands (a teacher teaching when he is not available, or a student in his first two years having to stay in school until late) cause a penalty of 1 000 000 being added to the penalty of the schedule, while most 'normal' soft constraints usually do not have penalties above 100 000.

5.4 Evaluation

As described in §3.2.4, the penalty of a schedule is the sum of the different penalties incurred by soft constraints being violated. These penalties are subdivided under different sections: penalties from the *educational* category are tied to subject groups, from the *teachers* category to teachers, from the *students* category to students and from the *counting groups* category to the counting groups. An example of parts of the overall penalty being assigned to teachers is shown in Figure 2.

Definition 5.6. We define several subsets of $\mathcal{C}$, namely the constraints which are tied to a single evaluator (as defined in Definition 5.2) and create a penalty in one of the categories described in §5.2. Note that we introduce the notation of C as the set of counting groups.

- $\mathcal{C}_E(g)$ for $g \in G$ apply to subject group g and contribute to a penalty in the Educational category.
- $\mathcal{C}_T(t)$ for $t \in T$ apply to teacher t and contribute to a penalty in the Teachers category.
- $\mathcal{C}_S(s)$ for $s \in S$ apply to student s and contribute to a penalty in the Students category.
- $\mathcal{C}_C(c)$ for $c \in C$ apply to counting group c and contribute to a penalty in the Counting Groups category.

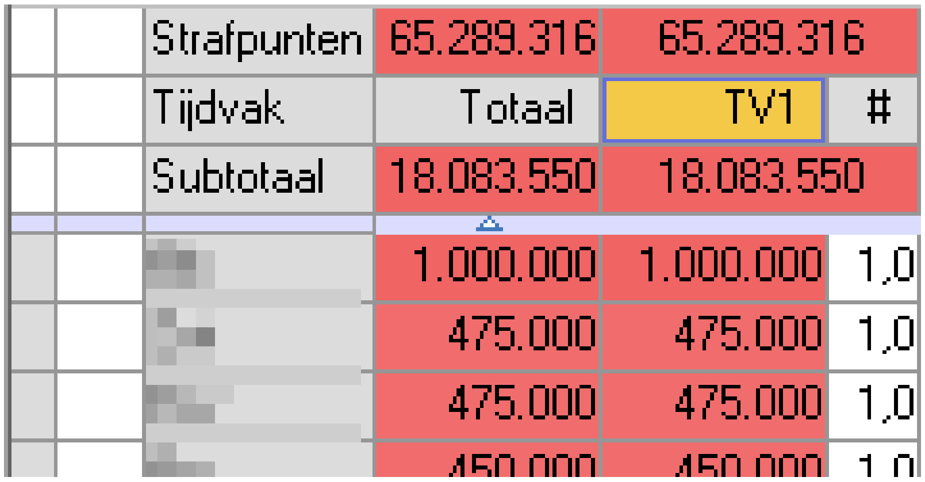
We would similarly define $\mathcal{C}_P(g)$ for placements and $\mathcal{C}_R(r)$ for rooms, but we do not study these penalties in this thesis.

Note that we use the notation $\mathcal{C}(a)$ to refer to all soft constraints that apply to some evaluator a, where $a \in A$ with $A \in \{G, T, S, C\}$. The category the soft constraints apply to (educational, students, teachers or counting groups) can be determined from the set that a comes from.

In other words,

$$\Phi(\mathcal{T},\mathcal{C}) = \sum_{g\in G}\Phi(\mathcal{T},\mathcal{C}_E(g)) + \sum_{t\in T}\Phi(\mathcal{T},\mathcal{C}_T(t)) + \sum_{s\in S}\Phi(\mathcal{T},\mathcal{C}_S(s)) + \sum_{c\in C}\Phi(\mathcal{T},\mathcal{C}_C(c)).$$

Note that since a soft constraint is tied to only a single evaluator in a single category, this sum of penalties is indeed equal to the penalty of the schedule taking into account all constraints.



		Strafpunten	65.289.316	65.289.316	
		Tijdvak	Totaal	TV1	#
		Subtotaal	18.083.550	18.083.550	
		[redacted]	1.000.000	1.000.000	1,0
		[redacted]	475.000	475.000	1,0
		[redacted]	475.000	475.000	1,0
		[redacted]	450.000	450.000	1,0

Figure 2: An example of penalty divided amongst teachers

This separating of the penalty into chunks instead of taking it all together is of vital importance to the approach taken in this thesis, which will be expanded upon in §9.

6 Current optimisation algorithms

Currently, practically all algorithms employed to optimise this specific HSSP are based on some variation of the Hill Climbing algorithm as described in §4.1. In the optimisation phase, a (partially) placed schedule is taken, and a new valid schedule is created by taking a single lesson $l \in \mathcal{T}$ and trying to schedule it in a different time slot. Then, any lesson already placed here that would make the schedule invalid (by containing a teacher or student who is also involved with lesson l) also gets placed in a new time slot. This happens either through a process of ‘switching’ (where it is placed in the timeslot that l came from) or through a process called ‘juggling’ where the lesson could be placed in any timeslot. From there, the same process is recursively applied with the conflicting lessons, until we have a new schedule. Even though more than one lesson was potentially ‘juggled’ in order to move lesson l, the new schedule can be seen as being a single change (of l to its new position) away from the original schedule.

After calculating the penalty of the new schedule, the schedule is accepted if it is not worse[8] than the original schedule.

As a result, practically all algorithms used suffer from the same drawback as the Hill Climbing algorithm does, which is that they all eventually get stuck in local optima and none of them are designed with options to get out of them.[9]

The research attempted in Part III is focused on improving this aspect of schedule optimisation.

7 Problem definition

The drawback of the Hill Climbing algorithm getting stuck in a local minimum, described in §4.1, is one which is also regularly encountered in practice at Zermelo. Usually, this is seen by schedulers as the point beyond which the schedule can no longer improve. Of course, the local minimum the schedule has converged to often is not the global minimum.

In practice, this is not always a problem. As is shown in §3.4, finding *the* global optimal solution is an $\mathcal{NP}$-complete problem, and no heuristic method guarantees that a solution which has been found is in fact the optimal solution. However, as long as the schedule is good enough for the teachers and students who need to work with it, it is not important whether the used solution is in fact optimal.

In §3.3, the concept of a `VALID` schedule was introduced: a schedule where every hard constraint is satisfied. Later, in §5.3, the concept of ‘softened hard constraints’ was introduced: constraints modelled as soft constraints even though in practice they cannot be violated. With this in mind, we observe the contradiction between what we want (a schedule where both hard and softened hard constraints are not violated) and the definition given in §3.3 which only requires no broken hard constraints. For this thesis, we thus need to extend our definition and introduce the concept of an *acceptable* schedule.

[8]We explicitly accept the schedule if it is *not worse* as opposed to *better*, to allow the algorithm to consider different schedules resulting in equal penalties in the hope that one of these schedules can be altered to allow a real improvement.

[9]There are a few algorithms which have the option built in to accept a worsened solution, but no implementation actually makes consistent use of this option to escape local minima.

Definition 7.1. Let $\texttt{ACCEPTABLE} : \mathcal{P}(L \times M) \rightarrow \{0, 1\}$. $\texttt{ACCEPTABLE}$ is the *acceptability function* of a schedule $\mathcal{T} \in \mathcal{P}(L \times M)$, whereby $\texttt{ACCEPTABLE}(\mathcal{T}) = 1$ means the schedule is $\texttt{VALID}$ *and* no evaluator has a penalty above 1 000 000. In other words:

$$\texttt{ACCEPTABLE}(\mathcal{T}) \Leftrightarrow \texttt{VALID}(\mathcal{T}) \wedge \forall \text{ evaluator } a : \Phi(\mathcal{T}, \mathcal{C}(a)) < 1\,000\,000.$$

Recall from §5.3 that softened hard constraints cause a penalty of 1 000 000 (or higher) to be added to the schedule. For a schedule to be ACCEPTABLE, we automatically require no softened hard constraints to be violated. Note that in theory it is also possible for a schedule to not be ACCEPTABLE, even if no softened hard constraints are violated, if the sum of the penalties caused by violated 'normal' soft constraints exceeds 1 000 000. However, in practice it turns out that nearly every schedule becomes ACCEPTABLE once no softened hard constraints are broken.

We can thus extend the problem definition given in §3.3 on page 9 as follows:

$$\begin{array}{lll} \text{Given} & L, \mathcal{C}, M & \\ \text{minimise} & \Phi(\mathcal{T}, \mathcal{C}) & \\ \text{subject to} & \texttt{ACCEPTABLE}(\mathcal{T}) = 1, & \\ & (l, m_l) \in \mathcal{T} & l \in L, m \in M. \end{array}$$

We retain the demand that the schedule is VALID through acceptability, but add the demand that no individual evaluator is assigned too high a penalty.

Notice that the minimisation of the total penalty $\Phi(\mathcal{T}, \mathcal{C})$ is secondary to the schedule being acceptable: we want the penalty of a schedule to be as low as possible, but in any case it needs to be acceptable.

Part III

Improving schedules using the Crowbar Method

8 Introduction

A problem that schedulers often face is that a schedule returned to them by one of the optimisation algorithms is not acceptable, due to softened hard constraints being violated. This is due to current optimisation algorithms focussing on minimising penalty Φ, end not being designed for the ultimate goal of creating an `ACCEPTABLE` schedule.

However, as a result of local search being used, this unacceptable schedule will often be one which is in a local minimum. Subsequently, any changes the Hill Climbing algorithm makes do not lead to improvement and consequently the schedule no longer improves.

One trick some schedulers use to solve this problem is a method known as the Crowbar Method (Dutch: breekijzermethode). This is a method in which a (softened hard) constraint $c \in \mathcal{C}$ that is broken is temporarily given a higher penalty artificially, by scaling the evaluator to which this constraint is tied. In this new 'scaled' situation the standard optimisation algorithms are run. The goal of this approach is to find a schedule in which the 'unscaled' quality is better than the original.

In this section, we study the automated scaling of evaluators, in such a way that we are able to escape local minima and create an `ACCEPTABLE` schedule in the process. This leads us to the question of this part of our thesis:

Can we modify a local search algorithm to automatically escape local minima?

As an extension to this question, we also study the effect on the acceptability of a schedule optimised under this algorithm.

9 Description

First, we describe scaling evaluators. To this end, we define a list of scalars, one for each evaluator, which in most cases will be 1.

Definition 9.1. For every evaluator a, we define `SCALE` as the value indicating how to scale the penalty incurred by evaluator a. This value is equal to 1, unless otherwise stated.

To use these scalars we introduce penalty function $\overline{\Phi}$, calculating the penalty of a schedule while taking the scalars into account. In other words, $\overline{\Phi}(\mathcal{T}, \mathcal{C})$ is calculated as

$$\begin{aligned}&\sum_{g\in G}\overline{\Phi}(\mathcal{T},\mathcal{C}_E(g))+\sum_{t\in T}\overline{\Phi}(\mathcal{T},\mathcal{C}_T(t))\\&+\sum_{s\in S}\overline{\Phi}(\mathcal{T},\mathcal{C}_S(s))+\sum_{c\in C}\overline{\Phi}(\mathcal{T},\mathcal{C}_C(c))\\&=\\&\sum_{g\in G}\texttt{SCALE}(g)\Phi(\mathcal{T},\mathcal{C}_E(g))+\sum_{t\in T}\texttt{SCALE}(t)\Phi(\mathcal{T},\mathcal{C}_T(t))\\&+\sum_{s\in S}\texttt{SCALE}(s)\Phi(\mathcal{T},\mathcal{C}_S(s))+\sum_{c\in C}\texttt{SCALE}(c)\Phi(\mathcal{T},\mathcal{C}_C(c))\end{aligned}$$

In this section, we discuss the algorithm used to implement the Crowbar Method. This algorithm is designed to select the evaluator that the Crowbar Method will study. The aim is to choose an evaluator on which a scaling will have the highest impact.

Algorithm 1 The Crowbar Method

```
1: Given: schedule $\mathcal{T}$, Cutoff value $m$, Max scale $s$, Number of seconds $t$ to
   run optimiser algorithm
2: Global variable PENALTIES
3: BestSchedule ← $\mathcal{T}$
4: OriginalSchedule ← $\mathcal{T}$
5: $e$ ← ChooseEvaluator($E$)
6: Complete ← FALSE
7: while Complete $\neq$ TRUE do
8:     PrePenalty ← $\overline{\Phi}(\mathcal{T},\mathcal{C})$
9:     $\mathcal{T}$ ← RunOptimiser($\mathcal{T}$, $t$)
10:    PostPenalty ← $\overline{\Phi}(\mathcal{T},\mathcal{C})$
11:    if PostPenalty < PrePenalty then
12:        BestSchedule ← $\mathcal{T}$
13:        Complete ← CheckIfComplete
14:        $e$ ← ChooseEvaluator($E$)
15:    else
16:        SCALE($e$) ← min(SCALE($e$) + 1, $s$)
17:    end if
18:    if User requested algorithm to end then
19:        Complete ← TRUE
20:    end if
21: end while
22: return $\mathcal{T}$
```

This algorithm itself uses a few other algorithms, including *ChooseEvaluator*,

$CheckIfComplete$, *RunOptimiser* and MIN. The implementations used for these algorithms and all the procedures they use in turn, are shown below.

The implementation for *RunOptimiser* boils down to running the Hill Climbing algorithm on a given schedule for a given number of seconds s, while MIN is simply choosing the smaller out of two values.

For the *ChooseEvaluator* method, we use the penalty placed on a given evaluator to calculate a probability with which a certain evaluator will be chosen in the next round. This probability equals the ratio between the penalty of the evaluator and the sum of the penalties of every evaluator above cutoff value s (this automatically means the sum of these probabilities equals 1).

```
procedure INITIALISE        # Produce evaluators sorted by their penalty
    $E \leftarrow$ The set of evaluators
    for every $e \in E$ do
        PENALTIES$[e] \leftarrow \Phi(\mathcal{T}, \mathcal{C}(e))$
    end for
    SORT(E) so that it is descending by the PENALTIES$[e]$
    return E
end procedure

procedure CHOOSEEVALUATOR($E$)
    $E \leftarrow$ INITIALISE
    PenaltySum $\leftarrow 0$
    CumulProb $\leftarrow []$
    CumulProbTot $\leftarrow 0$
    for every $e \in E$ such that PENALTIES$[e] \geq m$ do
        PenaltySum $\leftarrow$ PenaltySum + PENALTIES$[e]$
    end for
    for every $e \in E$ such that PENALTIES$[e] \geq m$ do
        CumulProb$[e] \leftarrow$ CumulProbTot + $\frac{\texttt{PENALTIES}[e]}{\texttt{PenaltySum}}$
        CumulProbTot $\leftarrow$ CumulProbTot + $\frac{\texttt{PENALTIES}[e]}{\texttt{PenaltySum}}$
    end for
    $r \leftarrow$ Random number from uniform distribution between 0 and 1
    for every $e \in E$ such that PENALTIES$[e] \geq m$ do
        if $r \leq$ CumulProb$[e]$ then
            return $e$
        end if
    end for
end procedure

procedure CHECKIFCOMPLETE
    if NoOfCrowbars >= MaxNoOfCrowbars then
        return TRUE
    end if
    temp $\leftarrow$ INITIALISE
    if $\overline{\Phi}$(FIRST(temp)) $< m$ then
        return TRUE
    end if
    return FALSE
end procedure
```

10 Experimental setup

To run simulations in which we supplement existing algorithms with the Crowbar Method, we make use of a grid-computing solution internally known as the 'Arena'. The Arena is a coordinated system which hands out tasks to networked computers known as Gladiators, prescribing an algorithm and a runtime each time. When the task is complete the Gladiator sends the result back to the Arena, at which time the Arena hands out a new task. By delegating the task of performing calculations on the schedule to multiple Gladiators, the amount of CPU-time spent on the problem can be increased by several orders of magnitude without requiring more physical time, while the Arena itself only works on coordinating tasks performed by a Gladiator.

In order to test the performance of the Crowbar Method, the Arena was modified to be able to send a Gladiator the instruction to perform the Crowbar algorithm, and to also be able to accept changes that worsen the unscaled value Φ whenever a Gladiator introduces new scaling factors.

This is of course subject to numerous parameters. These parameters include:

- the frequency at which the Crowbar Method is called;
- to what limit a deterioration of Φ is accepted by the Arena;
- the frequency at which the values of `SCALE` are set to 1 in order to 'reset' our search space;
- the parameters of the Crowbar algorithm itself.

The values chosen for these parameters could potentially have a significant impact on the results obtained by the Arena, both in the value of the result and the time it takes the algorithm to reach it.

Inherent to the HSSP is that optimising schedules is extremely time consuming, due to the huge search space and the fact that we are using heuristics. This means that the time (both CPU and real-world) it takes to measure the significance of a change in parameters is also extremely large, at least with respect to their long-term effect on the schedule. However, looking at the first results obtained in §11 we see (statistically) significant differences between using the Crowbar Method and not doing so. This will be analysed further in §12.

11 Proof of concept

To test the initial feasibility of the method discussed, we attempted to optimise the schedule of the upper years (Dutch: 'bovenbouw') of a Dutch secondary school. For schedulers, this part of the schedule is often more complex than the schedule for the lower years, because of the freedom students have in choosing their own curriculum. This means that the students in a class almost always vary per subject, so switching two lessons in a schedule is often not directly possible. This, as opposed to classes in the lower years, where students mostly follow the same subjects and usually follow all these subjects with the same classmates. This means that, providing teacher availability is not a problem, two classes with the same students can be switched easily.

Important to note on the tested schedule is that the unmodified Arena, which does not use the Crowbar algorithm, was not able to find any improvement in the schedule after running for over 3000 CPU hours (around 3 days in actual time in this instance). This is a time period after which a scheduler would have most likely already made the assumption that the schedule 'cannot be optimised further'. It is too long a time period to expect someone to wait for an improvement that might never come, due to the current schedule truly being trapped in a local minimum.

In the modified Arena, the parameters described in Algorithm 1 and in §10 were chosen on basis of a 'best guess'. This entails the following:

- When handing out the task, there is an $(h + 1)\%$ probability that the Crowbar Method is chosen. Here, h is the number of CPU hours which have passed since the last improvement. In addition, we make sure only one instance of the Crowbar algorithm is running at a time;
- Every time the Crowbar Method returns with a new value, the maximum percentage that Φ is allowed to deteriorate (and thus increase) is some value $0 \leq x \leq 15$ (picked randomly each time).
- The probability of setting all scalings (crowbars) back to 1 ($\forall c \in \mathcal{C}$, $\texttt{SCALE}(c) = 1$) is 0.5%.
- The cutoff value is picked randomly between 0 and 1 000 000, the maximum scale is set at 10, and the maximum number of scalings is a randomly picked number $1 \leq x \leq 3$.

With these described parameters, the Arena was run four times, using between 40 and 60 Gladiators at each moment depending on how many happened to be available. Note that at this stage the main goal was to see whether improving the schedule was possible at all, and the focus was less on obtaining an identical setup with regard to running time and number of Gladiators for each run. The amount of CPU time spent on each run consequently fluctuates. The progress of these four runs can be seen in Figures 3 and 4, of which the results are summarised in Table 2.

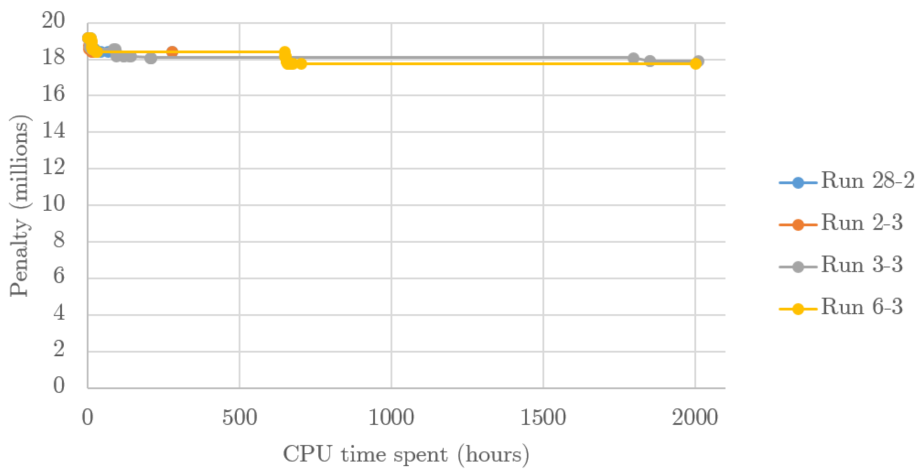


Figure 3: The penalty development of the four optimisation runs over time

Figure 3 shows that during all four runs, at least some improvement on the schedule was obtained. This already shows us an improvement over the situation where the Arena did not use the Crowbar algorithm as one of its optimisation algorithms. In addition, it can be noted that most improvements are grouped relatively closely together, separated by large periods of no change. This can be seen more clearly in Figure 4 below, where the vertical axis has been constrained in order to better show the improvements to the schedule.

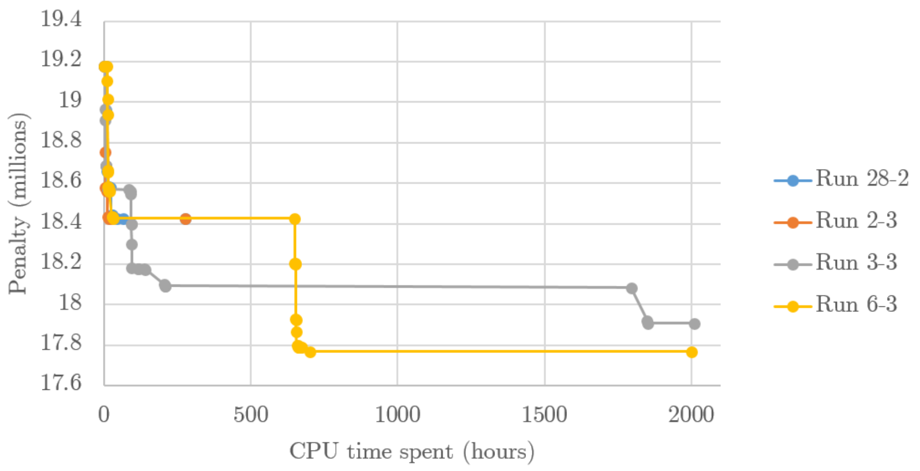


Figure 4: Same image as above, with a constrained vertical axis

The above figure shows a notable pattern, namely that in most cases an improvement is followed by more. Even if it was preceded by hundreds of hours of stagnation, a single (sometimes very small) improvement is in almost all cases followed by more improvements. Additionally, in this specific case, we find that there were improvements that were achieved extremely quickly when incorporating the Crowbar algorithm (on average in just over 7 CPU hours or under 15 ‘real’ minutes using 40 to 60 Gladiators, see also the data in Table 2) that were not found in over 3000 CPU hours of the normal Arena.

Run	Φ start	Φ end	%Δ	Runtime (CPU hours)	Time to first improvement (CPU hours)
1	19 177 132	18 426 720	−3.9%	67	8.37
2	19 177 132	18 426 720	−3.9%	278	4.67
3	19 177 132	17 907 640	−6.6%	2008	5.33
4	19 177 132	17 769 056	−7.3%	1980	10.62

Table 2: Summary of the results obtained from 4 runs

The observation made in regard to Figure 4, that small improvements made after periods of stagnation almost always lead to more improvements, reaffirms that it makes sense to look for any improvement – no matter the size – instead of trying to find the biggest improvement possible. This also makes sense in the context of escaping local minima: the fact that an improvement is made, no matter how small, shows we have escaped this local minimum and we can thus keep making improvements until we find a new local minimum.

11.1 Softened hard constraints

As described in §5.3, a primary goal of schedule optimisation is not only to make sure that the penalty Φ on the entire schedule is as low as possible, but also that no softened hard constraints are broken in the optimised schedule.

Therefore, another good way to verify the effectiveness of the algorithm is by looking at the evaluators in the different penalty categories (Educational, Teachers, Students and Counting Groups as described in §5.2) and by verifying none of these evaluators have softened hard constraints that are being violated. In practice, this means verifying that none of these evaluators endure a penalty of over 1 000 000, since this suggests the violation of a softened hard constraint. Tables 3 and 4 below show the difference in penalties for the 5 most penalised evaluators per category. We see very similar penalties occurring in each category, except that the broken softened hard constraint by one of the teachers before optimisation (in Table 3) has been resolved, reducing the highest evaluator overall from a penalty of 1 000 002 to 243 500. Note that it makes sense to study the distribution of penalties amongst evaluators in this way, because an evaluator represents one specific teacher, student, subject group or counting group. Therefore, an excessive penalty on any single evaluator would give the perception of a 'bad' schedule.

#	Educational	Teachers	Students	Counting Groups
1	260 011	1 000 002	241 500	500
2	250 011	135 012	218 050	200
3	220 431	120 002	215 550	192
4	220 423	112 002	211 550	100
5	220 423	103 002	211 250	66

Table 3: Evaluators per category with the highest penalty before optimisation

#	Educational	Teachers	Students	Counting Groups
1	220 431	135 072	243 500	500
2	220 423	126 002	217 750	200
3	220 423	120 202	215 550	192
4	220 416	112 002	210 550	100
5	220 116	100 002	203 050	66

Table 4: Evaluators per category with the highest penalty after optimisation

12 Efficacy of Crowbar Method

The main conclusion of §11 is that once a suitable scaling has been found that improves the schedule even marginally, more significant optimisations often follow. This can be seen as the result of the schedule having left its local optimum, and being able to iteratively improve until a new local optimum is reached. In addition, §11 confirms that the method employed is a fruitful one and can potentially lead to improvements that are not found without using the Crowbar Method.[10]

However, we also see that the time it takes for a schedule to be 'forced out' of its local minimum can still take large amounts of computer time. It seems convincing that being able to reduce this time would be one of the best ways in which to increase the overall performance of the algorithm.

Two questions arise from this initial exercise:

1. When optimising a schedule that does not improve without the Crowbar Method, how do we obtain optimisations using the method as quickly as possible?

[10]Inherent to the problem we are studying is that we cannot with 100% certainty guarantee a benefit to using the Crowbar Method over not using it.

2. It situations where improvements to the schedule *can* still be found without the Crowbar Method, does use of the method slow down optimisations?

The first question is studied in §12.1. Answering this enables us to make the most out of new optimisations made using the Crowbar Method.

The second question it studied in §12.2. Being able to answer 'no' to this question means there are no situations where using the new method has an adverse effect on optimising schedules which we could already improve without the Crowbar Method.

12.1 Optimisation of cutoff value

As shown in Algorithm 1, one of the parameters the Crowbar Method depends on is a cutoff value, to determine which evaluators come into consideration to be scaled by the algorithm. The reasoning behind this is, that it makes sense that the evaluators that need to be scaled in order to force the schedule out of its local optimum need to be large ones, or else the impact on the schedule will not be significant enough. On the other hand, the results in §11 show that the first improvement on a schedule need only be very slight in order to enter a sequence of subsequent improvements. This suggests that scaling 'small' evaluators might not necessarily be a waste of computing time and disregarding these might mean some improvements are missed.

In order to test this, a program has been written that repeatedly ran the Arena using 4 hyperthreaded (Intel i7-6820HQ, 2.7GHz) processor cores as Gladiators until a first improvement is found. Then, the CPU time taken for this improvement to occur has been logged, after which the entire system was reset and another run is done. In each run, a cutoff value is randomly taken from 0, 200 000, 400 000, 600 000 and 800 000.

12.1.1 First instance

For the first test run, we have taken the initial configuration of the schedule from §11, and we have done around 60 runs for each of the above mentioned cutoff values for a total of 299 runs. Each run is performed until the first improvement is made, after which the experiment is started again. Figure 5 contains a boxplot of these results, overlaid with an individual blue dot for each individual data point. Every circle corresponds to a data point outside the whiskers of the boxplot and thus qualifies as an outlier. The blue dot on the same height represents the same data point.

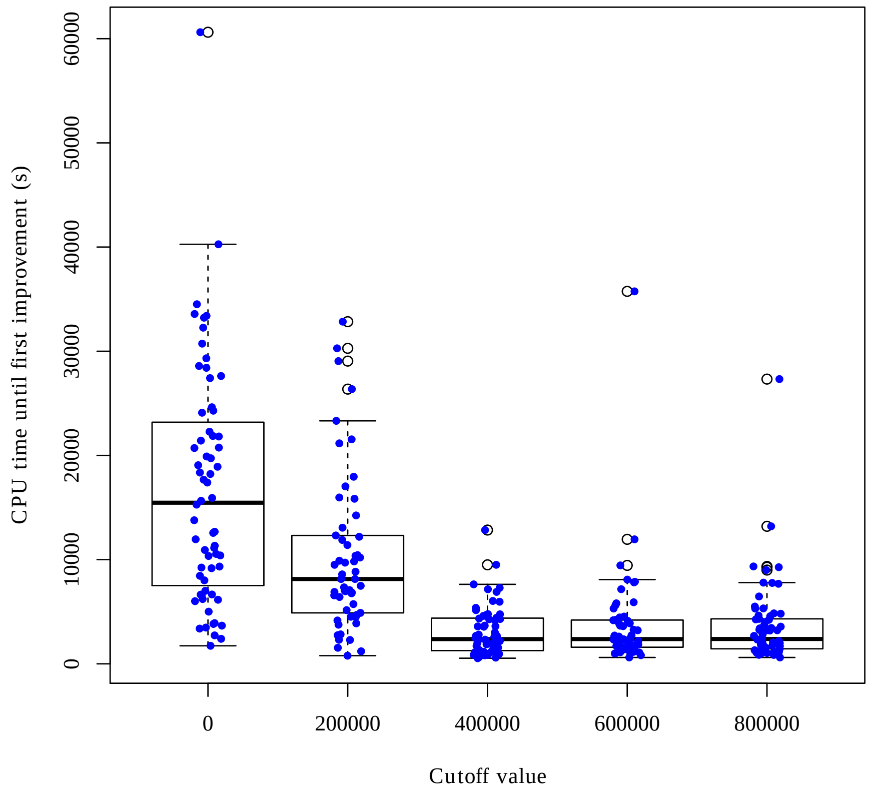


Figure 5: Boxplots of time needed to find a schedule improvement, plotted against the set cutoff value

Two things can be seen clearly from Figure 5:

1. The time taken to find the first result seems to significantly go down as the cutoff value goes up to 400 000, after which it seems to level off.
2. There seems to be a much larger variation in datapoints as the cutoff value gets lower.

Both of these results can be explained as the result of the first mentioned hypothesis at the beginning of this chapter, namely that relatively large evaluators need to be scaled in order to force a schedule into a new situation. This does not only explain the larger average time needed when the cutoff value is lower (due to more evaluators being possible and the extra time it takes for the algorithm to pick the ‘correct’ one), but also the large variance (short times are possible, if the correct evaluator happens to be scaled quickly).

A $t-$test on the data points with a cutoff value of at most 200 000 compared to those with a value of at least 400 000 shows us that the difference between these two sets is highly significant, with a $p-$value smaller than 0.0001.

12.1.2 Second instance

For the second test run, the same procedure was performed 161 times but now on the schedule from §11 in a new configuration that was reached using the Crowbar Method on the original schedule. This second configuration also stalled using the normal Arena, but it could be further optimised again using the Crowbar Method. The results of this are shown below in Figure 6.

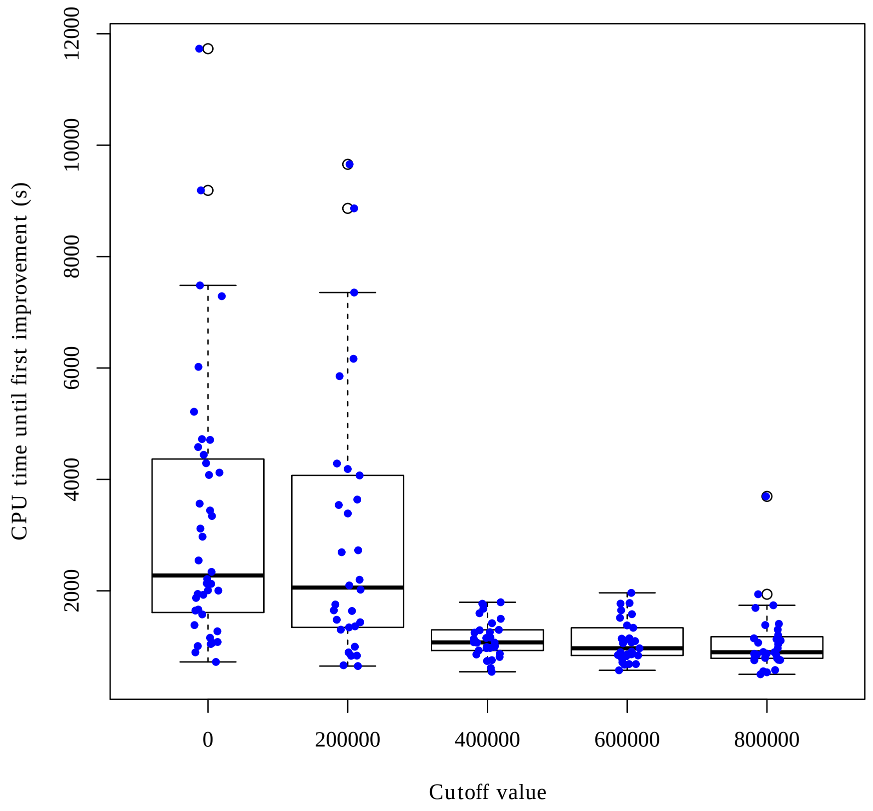


Figure 6: Boxplots of time needed to find a schedule improvement, plotted against the set cutoff margin

We again see improvements are made using the Crowbar Method when they were not found with the normal Arena, and that the time until first improvement tended to increase once the cutoff value was taken at 0 or 200 000. Like in §12.1.1, the difference between cutoff values of at most 200 000 and of at least 400 000 is highly significant.

12.2 Use of Crowbar Method on unoptimised schedules

In this subsection, we study the second question posed at the start of this section: does using the Crowbar Method have any adverse effects on the optimisation of schedules which we were still able to optimise without the method?

In an ideal situation, application of the method would also result in quicker improvements, but we at least need to ensure that improvements are not found more slowly when compared to the original situation.

12.2.1 First instance

For this first run, we took the problem from §11, but instead of taking the optimised schedule, we made an entire new `VALID` schedule that totally disregards the given soft constraints. As a result, the schedule initially had a penalty of 298 579 712 as opposed to the 19 177 132 we had before. Then we let the Arena run for 2 CPU hours (around 15 – 16 minutes in real time using 8 CPU cores) for 124 times in total, and we recorded the penalty Φ after this time.

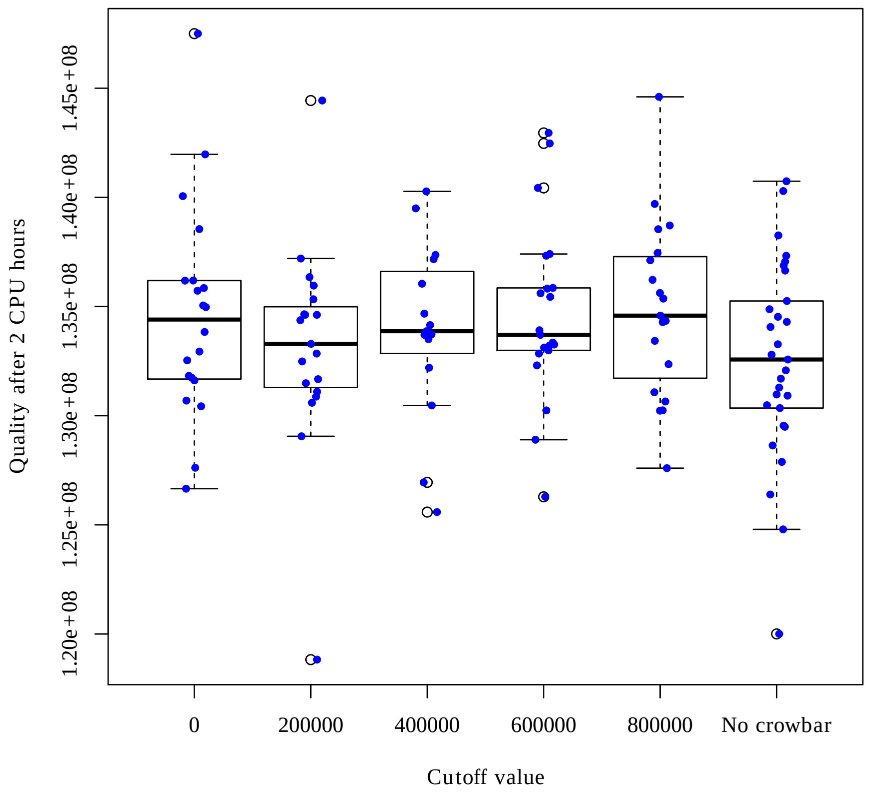


Figure 7: Boxplots of penalty after 2 CPU hours, plotted against the set cutoff margin

Figure 7 shows us that the difference in average quality for different cutoff values is quite small, seemingly indicating that in this situation the difference is not significant. Additionally the variability is large, with the median quality for every cutoff value falling within the margin of error of the other results. This means that no choice of cutoff value (or the choice not to use the Crowbar Method at all) is clearly superior.

Grouping all Crowbar runs together, we get Figure 8 below.

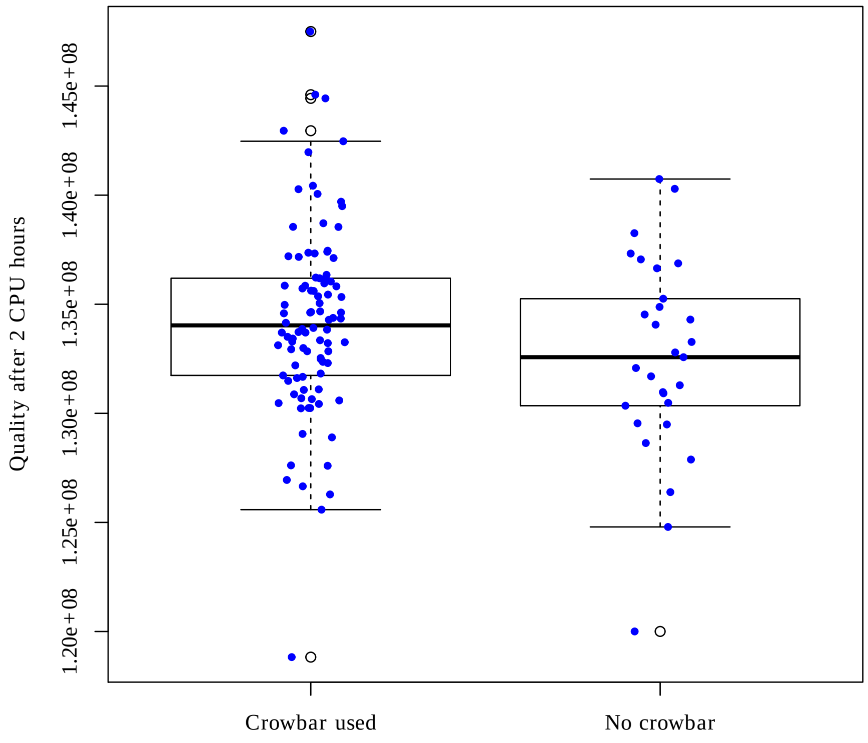


Figure 8: Boxplots of penalty after 2 CPU hours, grouped by the use of the Crowbar Method

Here, it is even more clear how small the difference in this case is, between using and not using the Crowbar Method. Indeed, performing a Student's $t-$test on the two datasets from Figure 8 shows a non-significant difference. In addition, none of the cutoff values by themselves differ significantly from the situation without the Crowbar Method.

Even though the difference is too small to be significant, we do see the average penalty after two hours of *not* using the Crowbar Method was the lowest of all the variations in this instance. A potential explanation for this is the fact that a part of the CPU time in the other cases was used to calculate a potential Crowbar, which was then never actually used since the schedule had been significantly improved in the meantime. This means that less CPU time was spent on actually improving the schedule.

12.2.2 Second instance

For the second run, we studied a completely new schedule, based on a different school than the one we were using before. We placed the lessons taking the already calculated clusters [14] into account to create a `VALID` schedule that had not yet been optimised, but some care was taken to place the lessons with soft constraints being taken into account.

Next, we again ran the Arena, switching between different cutoff values for the Crowbar Method and not using it at all. In total, 165 runs were performed. In order to increase the number of datapoints per variation we limited ourselves to setting cutoff values at 0, 400 000 and 800 000 and not using the Crowbar Method at all. This means that there are around 41 datapoints for each variation.

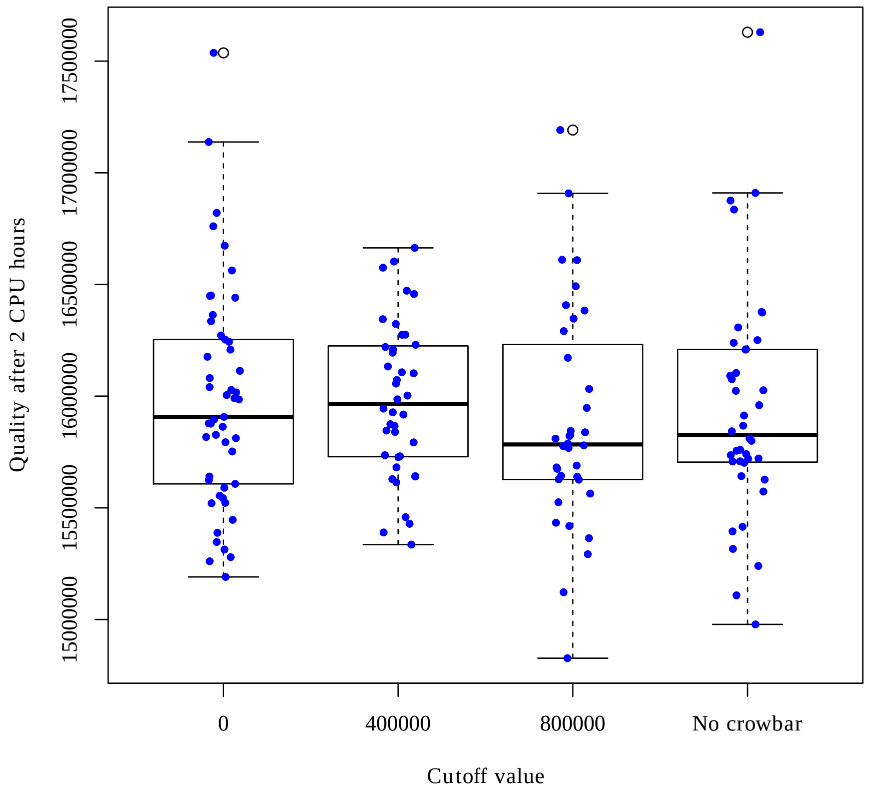


Figure 9: Boxplots of penalty after 2 CPU hours, plotted against the set cutoff margin

Here, like in the first instance above, we see that when improvements without the Crowbar Method are possible, additionally using the algorithm does not have an impact on the quality of the schedule after two CPU hours (positive *or* negative).

Grouping the datapoints where the Crowbar Method was used, like in Figure 10 below, shows this even more clearly.

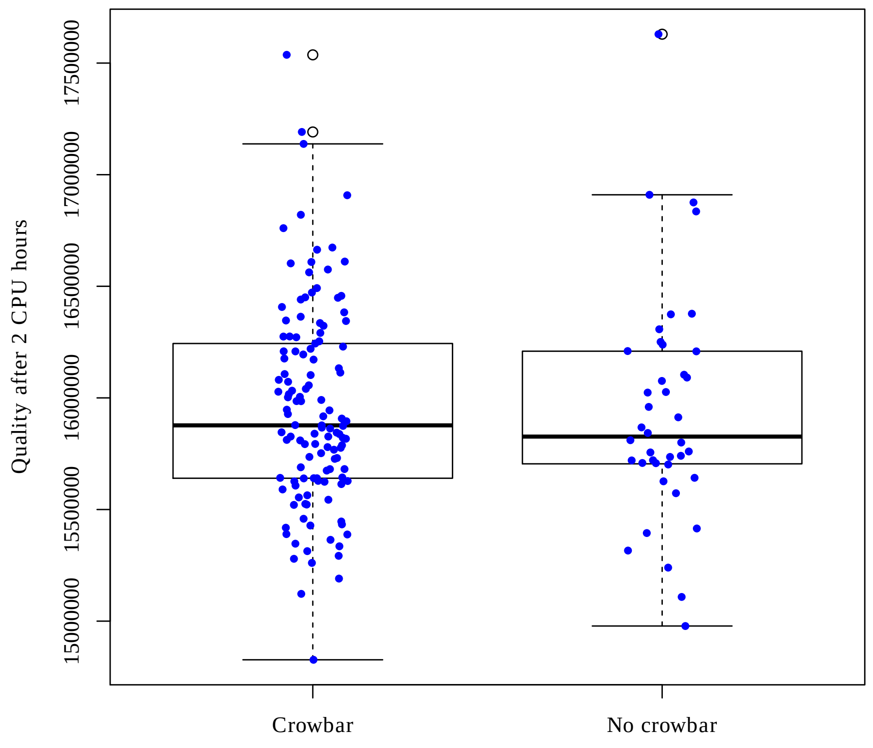


Figure 10: Boxplots of penalty after 2 CPU hours, grouped by the use of the Crowbar Method

13 Results

In this part, we tried to answer the question

Can we modify a local search algorithm to automatically escape local minima?

Looking at the data obtained in §11 and §12, one clear conclusion that can be drawn is that the Crowbar Method is able to generate improvements which were previously not possible. We saw improvements of a schedule that was not improved by standard optimisation algorithms, even though these 'standard' algorithms were given significantly more computing time. This consequently gives us a positive answer to our question. In addition, as can be most clearly seen from Table 4, the algorithm also succeeded in making the schedule `ACCEPTABLE` where this was not possible before, which was our second goal.

After an initial result in §11, we studied how to make the Crowbar Method work as efficiently as possible. We studied one of the parameters of the method, the cutoff value, and found that setting this parameters correctly has a significant impact on the efficiency of the method. In the case studied, the optimal value was found at around 400 000. While it would depend on the exact schedule what the optimal cutoff value would be, it makes sense that it is a relatively high value (in this case, only a single evaluator out of 3 158 exceeded a penalty of 400 000). Scaling an evaluator which already has a low penalty does not significantly affect the search space, and thus does not make it likely for an improvement to be found. Restricting the Crowbar algorithm to thus skip these evaluators means it is more likely for a suitable evaluator to be picked by the algorithm.

In addition, we studied the effect of applying the Crowbar Method to a schedule, even when we could also find optimisations using the standard algorithms only. In this instance, we found the average optimisation speed to be slightly lower when the Crowbar Method is employed. However, both of these decreases were not statistically significant and can thus confidently be disregarded. In any case, they were not large enough to negate the positive results obtained from schedules where optimisation without the Crowbar Method no longer seemed possible.

A final important note to be made is that next to the statistical data obtained, quite a bit of anecdotal data has been gathered from Zermelo employees in the meantime. Their experiences seem to confirm the results gathered in this chapter, which is that using the Crowbar Method generates improvements notably absent when not using the method. Even when not statistically verified, their experiences do support the assertions made in this part.

Part IV

Evaluating a schedule

14 Introduction

Imagine a schedule where the total penalty is based on the individual penalties of 2 000 evaluators (students, teachers, subject groups etc) with a total penalty of 2 000 000. It is possible that this entire penalty is based on two evaluators violating a softened hard constraint and both contributing 1 000 000 to the total penalty, forming 2 000 000. On the other hand, it could also be the case that each of the 2 000 evaluators has a penalty of exactly 1 000. This might mean that, for example, a student has one or two gaps in his schedule or something else which is relatively insignificant.

Even though in this example both schedules have an equal penalty, one could argue that the second schedule is more balanced and fairer than the first schedule. The concept of making the schedule more balanced and 'fair' for those involved is what is studied in this part. More concretely, we answer the question:

How can the penalty function be chosen so that it better reflects the subjective quality of a schedule?

This question is important when considering that the people who need to use the schedule (teachers, students etc) all need to be satisfied with the quality of their individual schedule, and consequently we cannot have someone who gets an unworkable schedule. An accurate penalty function for a schedule should ideally take this into account, and penalise schedules more heavily as a whole if they are unbalanced (i.e., when the schedule is not 'fair').

This question aligns well with our goal of making `ACCEPTABLE` schedules as defined by Definition 7.1. When a softened hard constraint is broken, one part of the schedule is unacceptably worse off than other parts of the schedule. If an objective function were able to reflect this in the penalty function, an optimisation algorithm would much better be able to work its way to a solution which could be used in the real world.

The following sections of this thesis study an altered penalty calculation method incorporating these ideas, as well as an algorithm which uses this altered penalty combined with the Crowbar Method discussed in Part III in order to quickly find `ACCEPTABLE` schedules by making the minimisation of $\Phi(\mathcal{T}, \mathcal{C})$ only secondary.

15 A fairness evaluation

15.1 Conceptually

Our goal in this section is to find a way in which to give a higher penalty to schedules which are not `ACCEPTABLE`. As such, we want to more severely penalise evaluators with a penalty over 1 000 000. More generally, in view of the above example, we want to incorporate the penalty variability amongst its evaluators into determining the quality of a schedule: we want a schedule to be *fair*.

To this end, we will define a penalty $\Phi^*(\mathcal{T}, \mathcal{C})$ which will take the concept of fairness into account.

Definition 15.1. $\Psi(\mathcal{T}, \mathcal{C}) : \mathcal{P}(L \times M) \times \mathcal{P}(\mathcal{C}) \to \mathbb{R}_{\geq 0}$ is the *fairness function* of a schedule, which measures how evenly penalties are divided amongst evaluators.

Definition 15.2. $\Phi^*(\mathcal{T}, \mathcal{C}) : \mathcal{P}(L \times M) \times \mathcal{P}(\mathcal{C}) \to \mathbb{R}_{\geq 0}$ is the *fairness evaluation* of a schedule: a penalty function which takes to into account both the violation of soft constraints, as well as the fairness defined above. We define this function as

$$\Phi^*(\mathcal{T}, \mathcal{C}) = \Phi(\mathcal{T}, \mathcal{C}) + \Psi(\mathcal{T}, \mathcal{C})$$

At this point, a decision needs to be made with regard to how exactly the fairness function should be defined, since Definition 15.1 does not address this. One standard approach here is to take the standard deviation of the values of all evaluators in the schedule, and adding this to the penalty. In §15.2, we show the comparable but slightly altered calculation that was used.

15.2 Implementation

We first need to define Ψ appropriately, taking several factors into account:

1. It has to be efficiently calculable
2. It has to accurately reflect the variability in the schedule

Starting with the standard deviation, the standard way of measuring this kind of variability, proved to be both inefficient as well as not able to effectively alter the penalty to reflect the variability of the schedule evaluated. These two problems are addressed separately below.

First, we want to be able to *efficiently* perform the calculations. Calculating the quality of a schedule is the most time-consuming part of an optimisation algorithm[11], so much care must be taken that this remains as fast as possible.

[11]The standard Hill Climbing algorithms used spend around 80-90% of their time calculating the quality of schedule variations, and only the small remainder was needed for the calculation of new schedule variations.

However, standard deviation calculations bring with it complexity which makes it difficult to efficiently implement in this case. Recall the standard deviation of a set x with N entries is given by

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2},$$

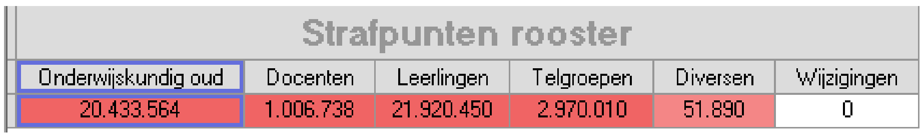
where $\overline{x}$ is the mean of the elements of x. The difficulty in this case lies in that the change of a single evaluator affects the average, and the distance to the mean changes for each evaluator, forcing substantial recalculations. The recalculating of a value for every evaluator can be prevented by using an altered version of Welford's method[15]. However, the recalculations which still needs to be done for every changed evaluator turn out to be so time consuming that the calculation of Φ^* is up to 30% slower than Φ.

The solution to this problem was to not to take into account each evaluator's distance to the mean, but consider its distance to 0. This changes the calculation to

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}x_i^2},$$

removing the need to keep track of an average over all evaluators and thus speeding up the calculation process. In addition, this measure also accomplishes our goal as described in §15.1 of more heavily penalising evaluators as their penalty gets larger.

The second requirement for our definition of the fairness function was that it should give an accurate reflection of the variability within the schedule. When defining our fairness evaluation as described above, a few initial tests show that the penalty added to a typical schedule is not of enough significance to drastically change the structure of a schedule and accomplish our goal of balancing the schedule. This is due to the penalty added to a typical schedule being less than 1% of the total penalty, as seen in Figure 11 where it is only 0.11%, resulting in the fairness evaluation being far outweighed by the traditional evaluators.



Strafpunten rooster

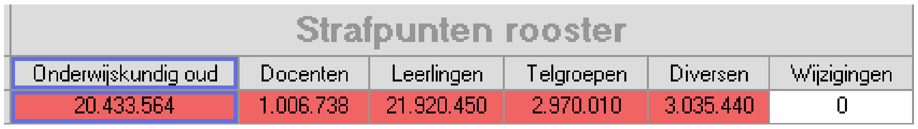
Onderwijskundig oud	Docenten	Leerlingen	Telgroepen	Diversen	Wijzigingen
20.433.564	1.006.738	21.920.450	2.970.010	51.890	0

Figure 11: The penalty on a schedule while using the $\sqrt{\frac{1}{N}\sum_{i=1}^{N}x_i^2}$ as a fairness function. The number under "Diversen" (Dutch for "miscellaneous") is the value of $\sqrt{\frac{1}{N}\sum_{i=1}^{N}x_i^2}$

In order to make the fairness function weigh up against the sum of the evaluators, the function was changed to no longer divide by the number of evaluators, but instead purely take the root over the sum of squares. Thus, we finally define:

$$\Psi(\mathcal{T},\mathcal{C}) = \sqrt{\sum_{g\in G}\Phi(\mathcal{T},\mathcal{C}_E(g))^2+\sum_{t\in T}\Phi(\mathcal{T},\mathcal{C}_T(t))^2+\sum_{s\in S}\Phi(\mathcal{T},\mathcal{C}_S(s))^2+\sum_{c\in C}\Phi(\mathcal{T},\mathcal{C}_C(c))^2}.$$

An example of this final fairness function can be seen in Figure 12:


Strafpunten rooster

Onderwijskundig oud	Docenten	Leerlingen	Telgroepen	Diversen	Wijzigingen
20.433.564	1.006.738	21.920.450	2.970.010	3.035.440	0

Figure 12: The penalty on a schedule while using the $\sqrt{\sum_{i=1}^N x_i^2}$ as a fairness function. The number under "Diversen" is the value of $\sqrt{\sum_{i=1}^N x_i^2}$

Above, we used the unscaled penalty Φ to define Φ^* and Ψ. Of course, we can also define $\overline{\Phi}^*$ as

$$\overline{\Phi}^*(\mathcal{T},\mathcal{C}) = \overline{\Phi}(\mathcal{T},\mathcal{C}) + \overline{\Psi}(\mathcal{T},\mathcal{C}),$$

with

$$\overline{\Psi}(\mathcal{T},\mathcal{C}) = \sqrt{\sum_{g\in G}\overline{\Phi}(\mathcal{T},\mathcal{C}_E(g))^2+\sum_{t\in T}\overline{\Phi}(\mathcal{T},\mathcal{C}_T(t))^2+\sum_{s\in S}\overline{\Phi}(\mathcal{T},\mathcal{C}_S(s))^2+\sum_{c\in C}\overline{\Phi}(\mathcal{T},\mathcal{C}_C(c))^2}.$$

Notice that in defining $\overline{\Psi}$, we use the scaled penalty in our calculations. This means that when scaling an evaluator, the extra penalty incurred due to the scaling is also taken into account in the fairness evaluation.

16 The Statistician

To test the concept described in §15, a new schedule optimisation algorithm (the Statistician) has been developed, that takes advantage of the new fairness evaluation. The algorithm tries to optimise a schedule towards a situation where the penalty is 'spread out' among the different evaluators as much as possible and, importantly, tries to minimise the penalty of the maximal evaluator(s).

This is done by optimising $\overline{\Phi}^*$ instead of Φ, and then punishing evaluators which are above certain target value by penalising them extra heavily through scaling.

In the subsections below, we first give a description of the algorithm. This is followed by the experiments done, and finally the results of these experiments.

Like the Crowbar algorithm described in Algorithm 1, the Statistician (described below in Algorithm 2) is a meta-algorithm, which relies on other optimisation algorithms through its call to *RunScaledFairnessOptimiser* to do local optimisations. The Statistician then makes changes to the search space through scaling of certain evaluators and restarts if the stopconditions are not met.

Notice we refer to *RunScaledFairnessOptimiser*: when running the optimisation algorithms, we use the penalty function $\overline{\Phi}^*$ in order to not only be able to use the fairness evaluation, but also of the scaling technique introduced in Part III.

Algorithm 2 The Statistician

```
1: Given: schedule $\mathcal{T}$, Number of seconds $t$, Goal margin $m$
2: BestSchedule ← $\mathcal{T}$
3: BestMargin ← GetHighestPenalty($\mathcal{T}$)
4: Complete ← FALSE
5: while Complete ≠ TRUE do
6:     $\mathcal{T}$ ←RunScaledFairnessOptimiser($\mathcal{T}$, $t$)
7:     if GetHighestPenalty($\mathcal{T}$) < BestMargin then
8:         BestMargin ← GetHighestPenalty($\mathcal{T}$)
9:         BestSchedule ← $\mathcal{T}$
10:        margin ← DetermineMargin($\mathcal{T}$, margin)
11:    end if
12:    ScaleEvaluators($\mathcal{T}$, margin)
13:    if BestMargin < $m$ then
14:        Complete ← TRUE
15:    end if
16:    if User requested algorithm to end then
17:        Complete ← TRUE
18:    end if
19: end while
20: return BestSchedule
```

Verbally, what the algorithm above does is:

1. Run a local search algorithm[12] to optimise the given schedule, evaluating using $\overline{\Phi}^*$.

[12]These are 'basic' local search algorithms, so do not make use of the Crowbar Method

2. After evaluation, determine if a new 'best' schedule has been found, where 'best' is determined from the value of the largest evaluator using Φ. If needed, a new `margin` is calculated. This `margin` is used to measure the progress of the algorithm.

3. Every evaluator with a penalty above `margin` is 'scaled', by using the Crowbar Method.

4. Unless the user stops the algorithm or all of our evaluators are below a certain target value, the algorithm repeats.

There are three subroutines that the Statistician uses:

1. *GetHighestPenalty*, which finds the value of the evaluator with the highest penalty. The Statistician uses this as a measure for the 'best' schedule.

2. *DetermineMargin*, which is used to calculate a margin which we are trying to get every evaluator below. This margin is an upper bound on penalties that evaluators are allowed to have. First, we verify if there are any evaluators above the current margin. If so, we do not yet lower it. If all evaluators are under the current margin, we set the new margin to $\mathit{GetHighestPenalty}(\mathcal{T}) \cdot 0.95$.

3. *ScaleEvaluators*, which adds a scaling factor to every evaluator *above* the current margin, in order to try and force these evaluators down. Initially 0.5 is added to the scale, but as time goes on it is more aggressively scaled by $\frac{1}{10}^{th}$ of the current scale.

```
procedure GetHighestPenalty($\mathcal{T}$)
    HighestPenalty $\leftarrow 0$
    for evaluators $e$ do
        HighestPenalty $\leftarrow max(\texttt{HighestPenalty}, \Phi(\mathcal{T}, \mathcal{C}(e))$
    end for
    return HighestPenalty
end procedure

procedure DetermineMargin($\mathcal{T}$, currentMargin)
    if GetHighestPenalty($\mathcal{T}$) $>$ margin then
        return currentMargin
    end if
    return $0.95 \cdot$ GetHighestPenalty($\mathcal{T}$)
end procedure

procedure ScaleEvaluators($\mathcal{T}$, margin)
    for evaluators $e$ do
        if $\Phi(\mathcal{T}, \mathcal{C}(e)) >$ margin then
            $\texttt{SCALE}(e) \leftarrow \texttt{SCALE}(e) + max\left(0.5, \frac{\texttt{SCALE}(e)}{10}\right)$
        end if
    end for
end procedure
```

17 Experiments

Like in Part III, we performed multiple experiments to test the Statistician, and to see how the algorithm performed under different circumstances.

The experiments were conducted in three phases:

1. The optimisation of the schedule, using only the Arena
2. The optimisation of the original schedule using the Statistician[13]
3. Optimising using the Arena *after* having run the Statistician

The first phase mentioned above is to obtain a schedule suitable for comparison with the results of phase three.

Phase two is where we really test the performance of our algorithm, and study the acceptability of the schedules afterwards.

[13]Unlike the Arena, for the Statistician all calculations have been performed 'locally' on a single computer. This is acceptable in this situation, since the CPU-time needed to obtain results was significantly lower than in the previous part.

In phase three, we optimise the schedules from phase two (where the Statistician was used) using the Arena. This is to see if using the Statistician has brought us into a new area of the search space where we can optimise using our original penalty function Φ while keeping our highest evaluator low (and the schedule `ACCEPTABLE`).

A summary of the results is given in §17.3.

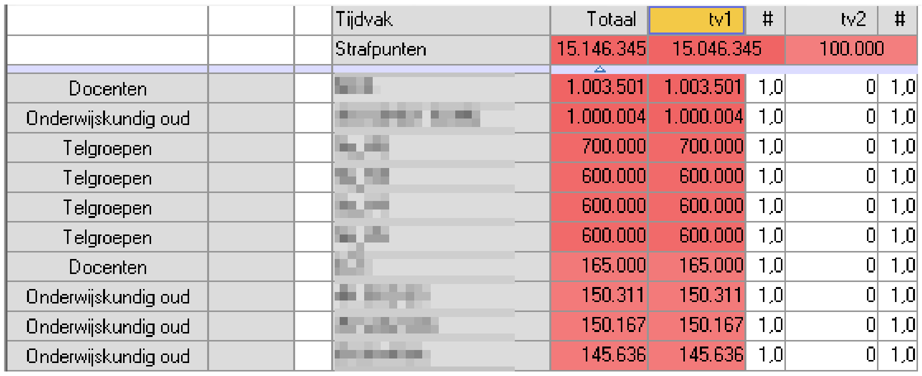
For the experiments, we used the following schedules:

1. In our first experiment, we study a situation where the higher grades (4th year and higher) are placed in the schedule, while the lower years are not.
2. For the second experiment, we used the same schedule used for testing the Crowbar Method in §11. As was seen previously, significant improvements to the penalty of that schedule were possible, at the cost of a significant amount of CPU time. Now, we look at whether the broken softened hard constraints can also be solved using the Statistician.

17.1 First schedule

17.1.1 Initial schedule

After placing the higher grades according to their clusters, a schedule was obtained where a large part of the penalty on the schedule was distributed among a relatively small portion of evaluators.



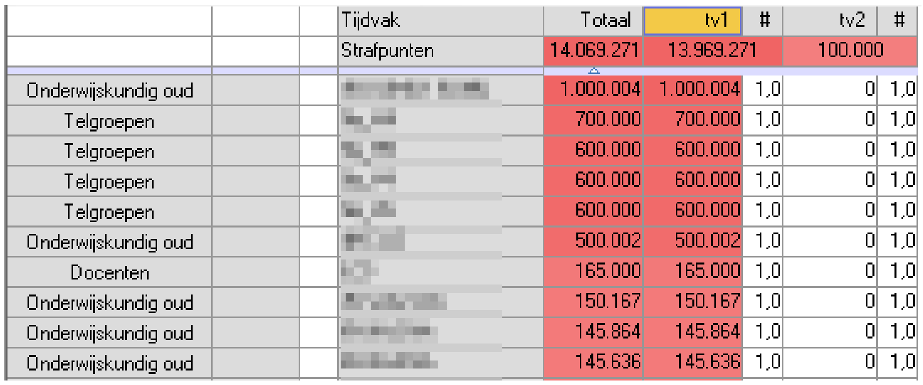
			Tijdvak	Totaal	tv1	#	tv2	#
			Strafpunten	15.146.345	15.046.345		100.000	
Docenten			[redacted]	1.003.501	1.003.501	1,0	0	1,0
Onderwijskundig oud			[redacted]	1.000.004	1.000.004	1,0	0	1,0
Telgroepen			[redacted]	700.000	700.000	1,0	0	1,0
Telgroepen			[redacted]	600.000	600.000	1,0	0	1,0
Telgroepen			[redacted]	600.000	600.000	1,0	0	1,0
Telgroepen			[redacted]	600.000	600.000	1,0	0	1,0
Docenten			[redacted]	165.000	165.000	1,0	0	1,0
Onderwijskundig oud			[redacted]	150.311	150.311	1,0	0	1,0
Onderwijskundig oud			[redacted]	150.167	150.167	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.636	145.636	1,0	0	1,0

Figure 13: The penalty on the first schedule before the Statistician, and the penalty on the top 10 evaluators

As can be seen in Figure 13, in a schedule with a total penalty of 15 146 345, 5 114 619 (33.8%) is spread over the top 10 evaluators (only 0.5% of the number of evaluators)

17.1.2 Phase one

After running the Arena for 3 full days using around 60 cores (around 4100 CPU hours), the schedule had improved somewhat, but out of a penalty of 14 069 271, 4 606 673 (32.7%) was still distributed among the top 10.

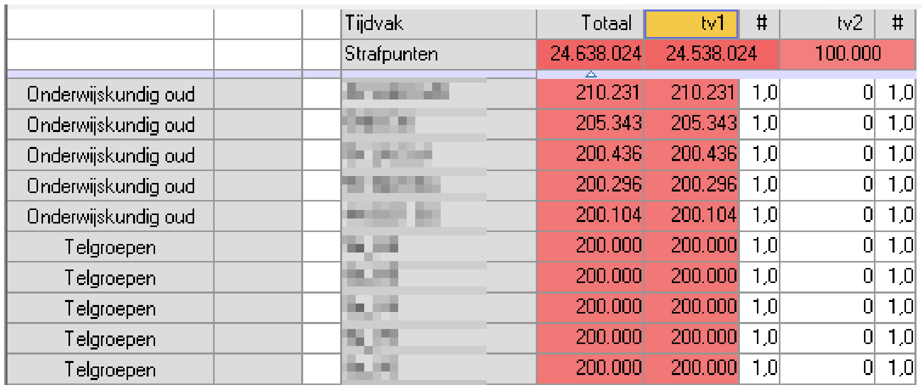


			Tijdvak	Totaal	tv1	#	tv2	#
			Strafpunten	14.069.271	13.969.271		100.000	
Onderwijskundig oud			[redacted]	1.000.004	1.000.004	1,0	0	1,0
Telgroepen			[redacted]	700.000	700.000	1,0	0	1,0
Telgroepen			[redacted]	600.000	600.000	1,0	0	1,0
Telgroepen			[redacted]	600.000	600.000	1,0	0	1,0
Telgroepen			[redacted]	600.000	600.000	1,0	0	1,0
Onderwijskundig oud			[redacted]	500.002	500.002	1,0	0	1,0
Docenten			[redacted]	165.000	165.000	1,0	0	1,0
Onderwijskundig oud			[redacted]	150.167	150.167	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.864	145.864	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.636	145.636	1,0	0	1,0

Figure 14: The penalty on the schedule from Figure 13 after running the Arena for 4100 CPU hours

17.1.3 Phase two

Now, going back to the schedule from Figure 13, we ran the Statistician for around 4 days and looked at the variability of the penalty afterwards, as shown in Figure 15.



			Tijdvak	Totaal	tv1	#	tv2	#
			Strafpunten	24.638.024	24.538.024		100.000	
Onderwijskundig oud			[redacted]	210.231	210.231	1,0	0	1,0
Onderwijskundig oud			[redacted]	205.343	205.343	1,0	0	1,0
Onderwijskundig oud			[redacted]	200.436	200.436	1,0	0	1,0
Onderwijskundig oud			[redacted]	200.296	200.296	1,0	0	1,0
Onderwijskundig oud			[redacted]	200.104	200.104	1,0	0	1,0
Telgroepen			[redacted]	200.000	200.000	1,0	0	1,0
Telgroepen			[redacted]	200.000	200.000	1,0	0	1,0
Telgroepen			[redacted]	200.000	200.000	1,0	0	1,0
Telgroepen			[redacted]	200.000	200.000	1,0	0	1,0
Telgroepen			[redacted]	200.000	200.000	1,0	0	1,0

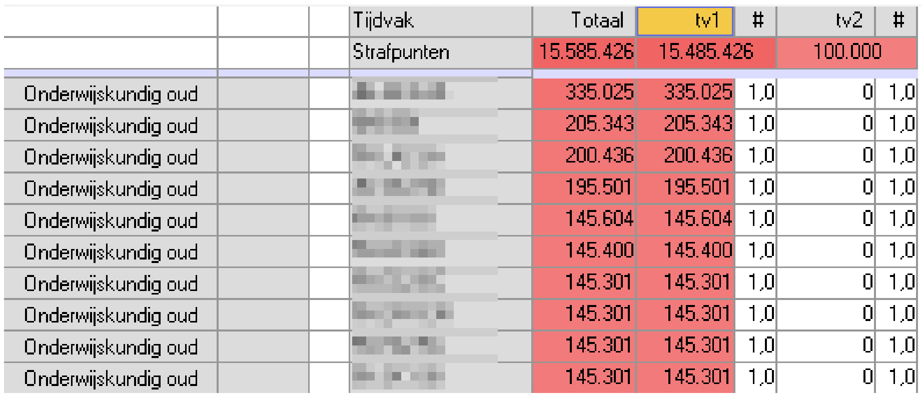
Figure 15: The penalty on the schedule from Figure 13 after running the Statistician for 4 days

Here we see how the total penalty has increased quite significantly, from 15 146 345 to 24 638 024. However, the sum of the penalties of the top 10 evaluators has decreased to 2 016 410, or 8.2% of the penalty total (down from 33.8%). In addition, the highest evaluator has a penalty of only 210 231, down from 1 003 501.

17.1.4 Phase three

After optimisation using the Statistician, we optimise the schedule using the Arena for around 1700 CPU hours.

The result of optimising using the Arena can be seen in Figure 16



			Tijdvak	Totaal	tv1	#	tv2	#
			Strafpunten	15.585.426	15.485.426		100.000	
Onderwijskundig oud			[redacted]	335.025	335.025	1,0	0	1,0
Onderwijskundig oud			[redacted]	205.343	205.343	1,0	0	1,0
Onderwijskundig oud			[redacted]	200.436	200.436	1,0	0	1,0
Onderwijskundig oud			[redacted]	195.501	195.501	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.604	145.604	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.400	145.400	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.301	145.301	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.301	145.301	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.301	145.301	1,0	0	1,0
Onderwijskundig oud			[redacted]	145.301	145.301	1,0	0	1,0

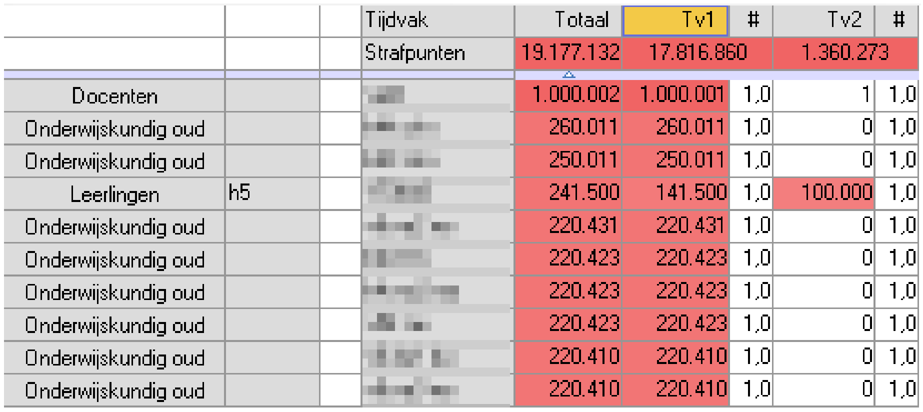
Figure 16: The penalty on the schedule from Figure 13 after running the Statistician for 4 days and subsequently the Arena for 1700 CPU hours

As we can see, the penalty has decreased by 36% since the situation directly after the Statistician, and is now only 2.9% higher than the original schedule. On the other hand, the highest evaluator is still quite low, at 335 025, which is 66.6% less than initially. Also, the total penalty of the top 10 evaluators is at 11.6% of the total penalty, which is still significantly less than the 33.8% of the original.

17.2 Second schedule

17.2.1 Initial schedule

Like the previous schedule, this instance was one where the higher grades had been placed but the lower ones had not.



			Tijdvak	Totaal	Tv1	#	Tv2	#
			Strafpunten	19.177.132	17.816.860		1.360.273	
Docenten			[redacted]	1.000.002	1.000.001	1,0	1	1,0
Onderwijskundig oud			[redacted]	260.011	260.011	1,0	0	1,0
Onderwijskundig oud			[redacted]	250.011	250.011	1,0	0	1,0
Leerlingen	h5		[redacted]	241.500	141.500	1,0	100.000	1,0
Onderwijskundig oud			[redacted]	220.431	220.431	1,0	0	1,0
Onderwijskundig oud			[redacted]	220.423	220.423	1,0	0	1,0
Onderwijskundig oud			[redacted]	220.423	220.423	1,0	0	1,0
Onderwijskundig oud			[redacted]	220.423	220.423	1,0	0	1,0
Onderwijskundig oud			[redacted]	220.410	220.410	1,0	0	1,0
Onderwijskundig oud			[redacted]	220.410	220.410	1,0	0	1,0

Figure 17: The penalty on the second schedule before the Statistician, and the penalty on the top 10 evaluators

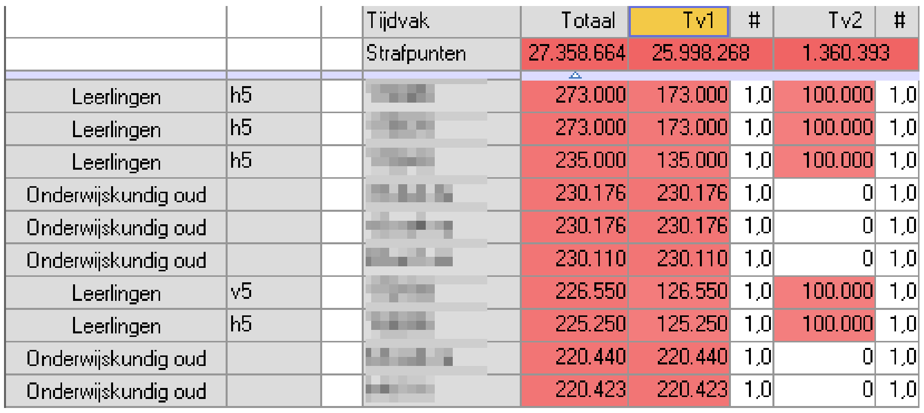
In this schedule, we see that with a total penalty of 19 177 132, 3 074 044 (16.0%) is spread over the top 10 evaluators. In a schedule with 1 787 evaluators, this is 0.6% of evaluators.

17.2.2 Phase one

As described in §11, this schedule is one which (after at least 3 000 hours of CPU time in the Arena) no longer showed any improvements. As such, the (distribution of the) penalty on the schedule after running it in the Arena remains the same.

17.2.3 Phase two

Using the schedule from Figure 17, we ran the Statistician for just under 7 hours. and again looked at the variability of the penalty afterwards. Notice that the Statistician was run for a significantly shorter period than in the first schedule. An explanation for this can be found in §17.4.



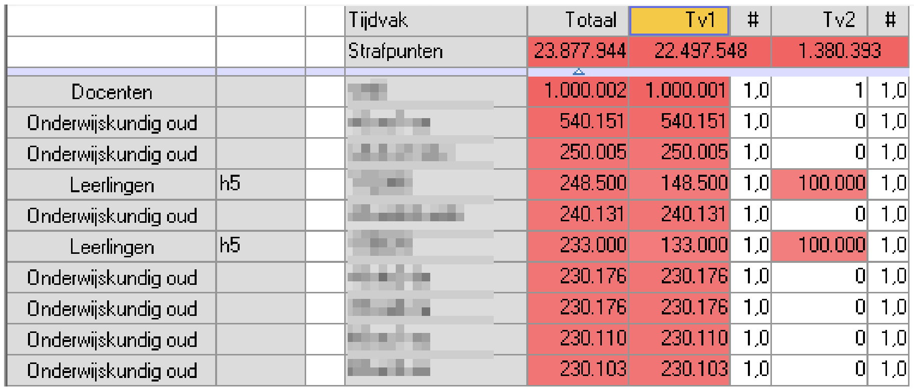
			Tijdvak	Totaal	Tv1	#	Tv2	#
			Strafpunten	27.358.664	25.998.268		1.360.393	
Leerlingen	h5		[redacted]	273.000	173.000	1,0	100.000	1,0
Leerlingen	h5		[redacted]	273.000	173.000	1,0	100.000	1,0
Leerlingen	h5		[redacted]	235.000	135.000	1,0	100.000	1,0
Onderwijskundig oud			[redacted]	230.176	230.176	1,0	0	1,0
Onderwijskundig oud			[redacted]	230.176	230.176	1,0	0	1,0
Onderwijskundig oud			[redacted]	230.110	230.110	1,0	0	1,0
Leerlingen	v5		[redacted]	226.550	126.550	1,0	100.000	1,0
Leerlingen	h5		[redacted]	225.250	125.250	1,0	100.000	1,0
Onderwijskundig oud			[redacted]	220.440	220.440	1,0	0	1,0
Onderwijskundig oud			[redacted]	220.423	220.423	1,0	0	1,0

Figure 18: The penalty on the schedule from Figure 17 after running the Statistician for 7 hours

Like in the first schedule, we see that both the sum of the penalties of the top 10 evaluators (8.6% of the total, down from 16%) and the penalty of the highest evaluator have decreased significantly, at the cost of the total penalty (from 19 177 132 to 27 358 664, an increase of 42.7%).

17.2.4 Phase three

After optimisation using the Statistician, we again optimise using the Arena. This time the Arena was run for around 16 000 CPU hours (one week in real time, over a holiday period), but since the last optimisation was done only 38 hours into the run we can be fairly confident no more optimisations would be found.



			Tijdvak	Totaal	Tv1	#	Tv2	#
			Strafpunten	23.877.944	22.497.548		1.380.393	
Docenten			[redacted]	1.000.002	1.000.001	1,0	1	1,0
Onderwijskundig oud			[redacted]	540.151	540.151	1,0	0	1,0
Onderwijskundig oud			[redacted]	250.005	250.005	1,0	0	1,0
Leerlingen	h5		[redacted]	248.500	148.500	1,0	100.000	1,0
Onderwijskundig oud			[redacted]	240.131	240.131	1,0	0	1,0
Leerlingen	h5		[redacted]	233.000	133.000	1,0	100.000	1,0
Onderwijskundig oud			[redacted]	230.176	230.176	1,0	0	1,0
Onderwijskundig oud			[redacted]	230.176	230.176	1,0	0	1,0
Onderwijskundig oud			[redacted]	230.110	230.110	1,0	0	1,0
Onderwijskundig oud			[redacted]	230.103	230.103	1,0	0	1,0

Figure 19: The penalty on the schedule from Figure 17 after running the Statistician for 2 hours and subsequently the Arena for 16 000 CPU hours

As expected, the total penalty of the schedule has decreased, this time by 12.7% compared to phase two. However, we can see that one of our main goals has failed: The highest evaluator in the schedule has gone back to 1 000 002 — higher than our goal of a maximum of 1 000 000. In addition, even though the penalty of the top 10 evaluators is at 14.4% of the schedule total compared to 16.0% originally, its actual value has *increased* compared to the original.

The example of this second schedule shows the problem between alternating optimisation between two different penalty functions (Φ and Φ^*). Doing so can lead to situations where neither goal is achieved: using penalty Φ our schedule is worse off than originally, while the reduction of the highest evaluator which we achieved in phase two using fairness evaluation Φ^* was all but undone in this phase.

17.3 Overview of results

In the table below, we summarise the results obtained from the experiments on our two schedules. We show the penalty of the schedule and of the highest evaluator, and compare this to the initial penalty (notation: Δ Start).

Ph.	Φ	Δ Start	top evaluator	Δ Start	top 10	Δ Start
Start	15 146 345	—	1 003 501	—	5 114 619	—
1	14 069 271	−7.1%	1 000 004	−0.3%	4 606 673	−9.9%
2	24 638 024	62.7%	210 231	−79.1%	2 016 410	−60.6%
3	15 585 426	2.9%	335 025	−66.6%	1 808 513	−64.6%

Table 5: Summary of the results of the first schedule

Ph.	Φ	Δ Start	top evaluator	Δ Start	top 10	Δ Start
Start	19 177 132	—	1 000 002	—	3 074 044	—
1	19 177 132	—	1 000 002	—	3 074 044	—
2	27 358 664	42.7%	273 000	−72.7%	2 364 125	−23.1%
3	23 877 944	24.5%	1 000 002	—	3 432 354	11.7%

Table 6: Summary of the results of the second schedule

17.4 Changes in highest evaluator and schedule quality

The way in which we measure the quality of a schedule in the Statistician is by looking at the value of the highest evaluator in that schedule. However, as is also the case with other optimisation algorithms, we always try to find a balance between how long to run an algorithm and how much improvement is still to be gained from this extra running time. As such, here we analyse the development of the value of the highest evaluator over time in phase two, where the Statistician was run on the schedule, to see if any conclusions can be drawn with regard to the running time of the algorithm.

17.4.1 First schedule

As we can see in Figure 20, we notice a rapid decrease in the penalty on the highest evaluator, as indicated by the orange line: Starting at 1 003 501, it decreases to less than 800 000 (under our goal of 1 000 000) in less than an hour, to under 500 000 in 12 hours and to its final value of 210 231 in 34 hours.

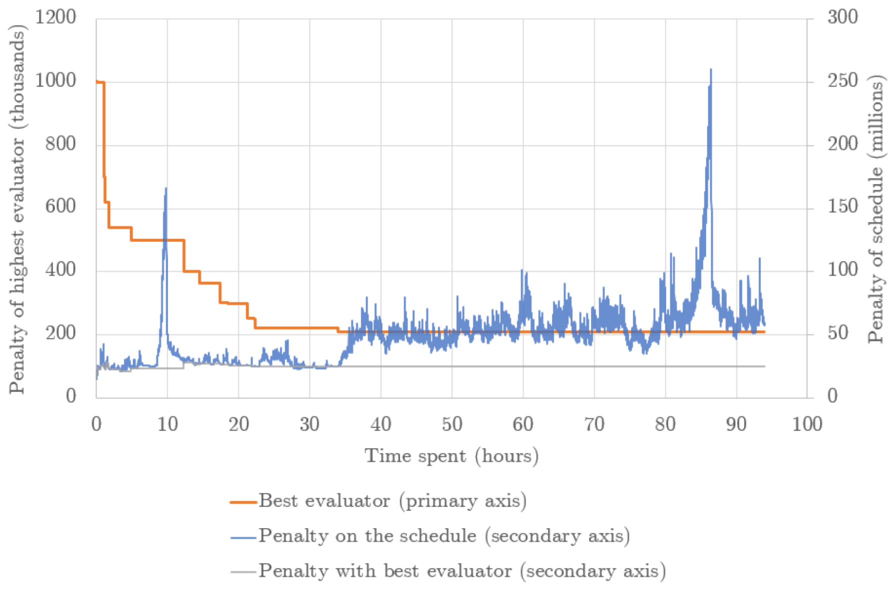


Figure 20: The development of the penalty on the schedule from Figure 13 during phase two.

The blue line indicates the quality of the schedule at 15 second intervals. As we can see, in the first 30 hours of optimisation the penalty of the schedule fluctuated relatively close to the starting penalty of 15 146 345 when we exclude the peak around 10 hours in.

After around 34 hours, immediately after the value of the ‘highest evaluator’ had decreased to 210 231, the quality of the schedule at subsequent intervals began to rapidly decrease, finally ending up above 50 000 000.

17.4.2 Second schedule

Like in the first schedule above, Figure 21 shows us how the initial lowering of the best evaluator value happens quite rapidly. Also, while less obviously visible than in the first schedule, we see that penalty Φ starts increasing significantly after the last improvement of the highest evaluator.

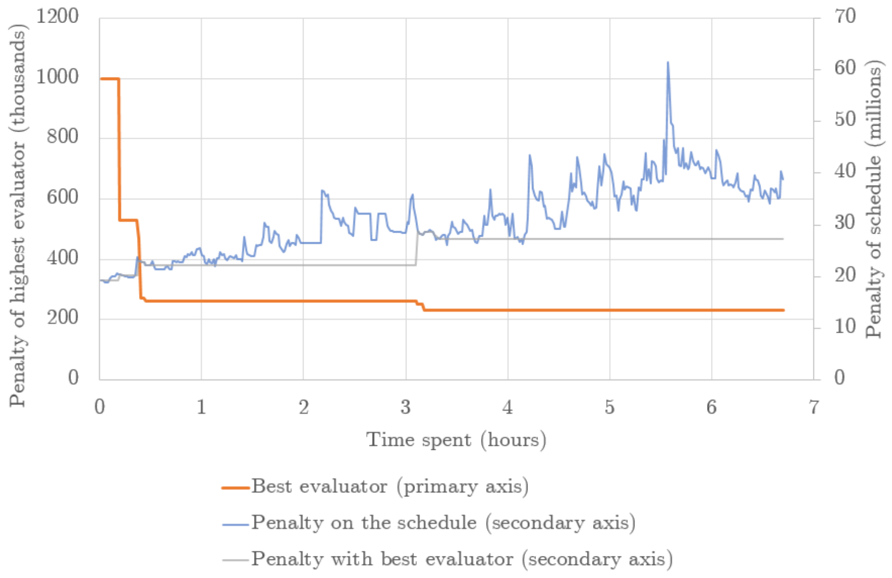


Figure 21: The development of the penalty on the schedule from Figure 17 during phase two

As can be seen from figures 20 and 21, at some point the Statistician reaches a point where the penalty on the schedule consistently stays significantly higher than what had been reached before. The moment this happens coincides exactly with the point at which the final improvement to the highest evaluator has been made.

An explanation for this can be found by studying Algorithm 2: when a new margin is calculated, this is taken to be 95% of the current highest evaluator. However, at some point, a large amount of evaluators will fall within this margin due to all evaluators being forces closer together and will thus start being scaled. This will start adversely affecting Φ, since all scaled evaluators will force themselves into new timeslots removing the structure from the schedule.

This result can thus be seen as a shortcoming in the algorithm, and it is possible that a better result might be obtained if the margin is decreased more subtly when necessary. However, the trend seen above does also hint at a logical conclusion. Namely, that at some point it becomes more and more difficult to place lessons in such a way that penalties are spread out, and that over-forcing lessons into a schedule severely degrades the quality of that schedule.

18 Results

In this part, we tried to answer the question

How can the penalty function be chosen so that it better reflects the subjective quality of a schedule?

From the above tables, we can see both the effectiveness as well as the shortcomings of using the Statistician as a way to optimise schedules using the 'fairness' of the schedule as the main focus.

In 'Phase 2', we ran the Statistician for a certain period of time, and we see that this had the desired effect on both schedules: the value of the worst evaluator was in both cases decreased dramatically, and we additionally saw decreases in the sum of the top 10 evaluators. Since acceptability was defined in Definition 7.1 as a `VALID` schedule where no evaluator had an individual penalty of 1 000 000 or higher, we see that in both the studied schedules this goal was achieved, with the highest evaluator being either 210 231 or 273 000. This means we can answer the question posed in the affirmative.

In both cases, the penalty function Φ increased significantly. Since the Statistician pays almost no attention to the quality of the schedule as-is but purely to its fairness, it makes sense to try and use a standard optimisation algorithm after the Statistician has completed in order to try and reduce Φ back. However, it can be seen from the result that this only sometimes works as intended. In the first schedule, Φ was reduced back to almost the same level as initially, while the highest evaluator stayed relatively low. On the other hand, the second schedule shows that Φ stayed significantly higher than original while the highest evaluator remained exactly the same as initially.

This shows that while the method used can be effective in some cases, it is not guaranteed to 'improve' a schedule, no matter whether Φ or the value of the highest evaluator is used as an indication of improvement.

19 Conclusion

In this thesis, we studied the High School Scheduling problem, specifically as it relates to the Dutch schooling system and as implemented by a Dutch company specialising in software related to High School Scheduling.

In parts I and II, we studied the HSSP in a general sense, and the specific choices made in the model used in this research. Particularly, we introduced the concept of softened hard constraints: constraints that are modelled as soft constraints, but in reality cannot be violated if we want our schedule to be acceptable.

In Part III, we studied the question

Can we modify a local search algorithm to automatically escape local minima?

We looked at the so called 'Crowbar Method', a method not yet found in the literature in which violated constraints were scaled in order to give local optimisation algorithms more incentive to solve these problems. We looked at examples of schedules where we could indeed see improvements using this technique. Since the scaling of constraints was most often done on those constraints which caused the highest penalties for the schedule penalty, they were most often softened hard constraints. Consequently, the method occasionally manages to reduce the penalty on a schedule and also solve the violation of softened hard constraints. We were thus able to create an algorithm based on local search, which appears able to escape local minima by transforming the search space the algorithm is working in.

In Part IV, we shifted our focus from looking at constraints to looking at the penalty function used to evaluate a schedule. We studied the question

How can the penalty function be chosen so that it better reflects the subjective quality of a schedule?

The observation was made that the penalty function used was focused on the reduction of the total penalty on the schedule, while perhaps it makes more sense to also analyse the distribution of violated constraints, and to more heavily penalise schedules when softened hard constraints are violated. This concept was expanded to extend the penalty function with an extra penalty based on how 'fair' a given schedule is: schedules where broken evaluators are distributed fairly amongst teachers, students and other stakeholders are evaluated more favourably than those where all the broken constraints are focused on a small number of evaluators.

Using this altered penalty, we found we could relatively quickly reduce the penalty on the most heavily penalised evaluator quite dramatically. On the other hand, this increased the penalty on the schedule as a whole when using the original penalty function. This means our goal with the altered penalty function was definitely reached with regard to the total penalty, which was more spread out over the different evaluators.

Using the original penalty, we found this resulted in the total penalty increasing significantly. When trying to optimise these schedules using the local search algorithms, we found that in some cases we managed to get the total penalty down to around the level we had before while still having a relatively low highest evaluator. In some other cases however, we found the highest evaluator returned to the level found originally while the total penalty also remained at a higher level than before. In other words, trying to get the 'best of both worlds' using two different penalty functions sometimes gave positive result, but in other cases it only worked counterproductively.

19.1 Inspiration for future research

The research done in this thesis can be seen as somewhat exploratory. Global proofs of concept were developed for the questions posed, based on ideas and experiences about High School Scheduling and on optimisation problems in general. These were then applied to specific problem instances where their potential benefits were shown and described. However, this process raised follow up questions which were not studied in this thesis. Potential ways in which the performed research can be further developed are described below.

- **Use of the Crowbar Method on general optimisation problems**

  The Crowbar Method is, as far as is known, a novel approach to optimisation problems. In the context of this thesis the approach was applied to the High School Scheduling problem, but the promising results obtained invite the question whether the method could also be applied against other optimisation problems. Specifically, this would entail optimisation problems which are based around minimising the penalty generated by soft constraints (as opposed to problems dominated by hard constraints). These could be other kinds of scheduling problems, unrelated to the HSSP, but also different optimisation problems altogether.

- **More specialised methods to calculate scaling factors**

  An important part of the Crowbar Method is choosing which evaluators to scale. In this case, the decision was made to choose one of the evaluators above a certain penalty threshold using a probability weighted using the penalty of the evaluator. However, no alternative ways in which to choose the evaluator to scale were studied. Choosing the correct evaluator to scale every time (and by the correct factor) would potentially increase the speed at which the Crowbar Method is effective by several orders of magnitude, so optimising this decision could be worthwhile. Examples of factors which could be taken into account are, for example, the number of classes a teacher needs to teach[14] or to what grade a subject is taught.

- **Study into matching schedule penalty with subjective quality**

  In Part IV, a 'fairness evaluation' was introduced in an attempt to create a penalty function whereby the penalty on the schedule agreed with what is subjectively found to be a 'good' schedule. In defining this penalty, we were also taking into account the problem we were attempting to solve as described in §7 where we wanted every evaluator to end up under a 'good enough' (having a penalty under 1 000 000). This goal was reached, but at the cost of having a penalty on the schedule as a whole increase significantly. The question which this result brought up is to what extent a better variability of the schedule is acceptable when taking into account the cost of an increased penalty based on broken soft constraints.

  As such, it is not immediately obvious that the penalty introduced in Part IV is one which always gives a penalty to schedules which matches its subjective quality. No more in depth research was done in this area, so studying the penalty of High School Scheduling Problems in general might be an interesting topic to explore.

- **The real-world effectiveness of the proposed methods**

  In this research, both the Crowbar Method and the work done with the 'fairness evaluation' and the Statistician was done in a way which showed its performance on a few problem instances. It must be noted that this was done with little knowledge of all the intricacies of scheduling, or the years of experience many schedulers have. This means that, although promising, it remains to be seen how much benefit will be found when schedulers use the method in practice. By the end of this research, with the scheduling season beginning for Dutch secondary schools, we have received multiple indications of useful results being obtained from the Crowbar Method in particular. However, no large scale study has yet been done into how much schedule quality has improved or how much time is saved.

[14]As someone's schedule get fuller they are usually more difficult to place correctly, so they might be given priority.

20 Glossary

Throughout this thesis, use was made of terms which often have specific meanings within timetabling. What follows is a list of these terms.

Block	A lesson for a subject group which is planned on two or more consecutive time slots on the same day, in practice becoming one long lesson.
Cluster(ing)	A method of structuring a schedule, by grouping lessons into *clusters*: sets which can be scheduled simultaneously, due to the sets of students attending each lesson being disjoint.
Constraint, hard	A restriction placed on a timetable which may not be broken.
Constraint, soft	A restriction placed on a time table which may be broken, but which does decrease the *evaluation* of the proposed timetable.
Gap	A gap in a timetable occurs when there is idle time surrounded by lessons. For teachers and students, a gap means that after a certain number of lessons, they will have to wait before being able to attend their next lesson.
HSSP	High School Scheduling Problem.
Lesson	One time-instance of a *subject group* meeting. Next to a subject group, a lesson is tied to a *room* and a *time slot*.
Penalty	The quality of a schedule. Often a numeric value based on which *soft constraints* are broken by the timetable, and how important these constraints are relative to each other.
Subject group	A collection of *students* following a certain *subject* taught by a (number of) *teacher(s)*. A subject group is assigned a number of lessons for every week, possibly with extra constraints.
Time slot	A discrete moment on the timetable where a *lesson* may take place. A timetable usually consists of an equal number of timeslots (8 to 10) on each weekday, and 0 on each day in the weekend.

21 Symbols & Notation

What follows is a list of the symbols and notation used throughout this thesis. More thorough and rigorous definitions are found in the content of the thesis.

$\{\{a, a, b\}\}$	Notation for a multiset, which may contain an element multiple times.
$\texttt{ACCEPTABLE}(\mathcal{T})$	A function which indicates if a given schedule $\mathcal{T}$ is acceptable.
$\mathcal{C}$	The set of all soft constraints.
$\mathcal{C}_A(a)$	A subset of the soft constraints, whereby A refers to the category and a to the relevant entity (Definition 5.6).
C	The set of all counting groups.
G	The set of all subject groups.
L	The set of all lessons.
M	The set of all timeslots.
$\mathcal{P}(A)$	The powerset of some set A, so the set containing all subsets of A (including the empty set and A itself).
R	The set of all rooms.
S	The set of all students.
$\mathcal{S}$	A subset of S, used as the group of students tied to a subject group.
$\texttt{SCALE}(a)$	For an evaluator a, $\texttt{SCALE}(a)$ is the value indicating how to scale the penalty incurred by evaluator a. This value is equal to 1, unless otherwise stated.
T	The set of all teachers.
$\mathfrak{T}$	A subset of T, used as the group of teachers tied to a subject group.
$\mathcal{T}$	A schedule, which is a subset of L.
$\mathcal{T}_A(a)$	For an $a \in A$, whereby $A \in \{S, T, G, R, M\}$, $\mathcal{T}_A(a)$ is a subset of $\mathcal{T}$ of lessons containing entity a which is a part of set A.
V	The set of all lessons.
$\texttt{VALID}(\mathcal{T})$	A function which indicates if a given schedule $\mathcal{T}$ is valid.
$\Phi(\mathcal{T}, \mathcal{C})$	The penalty function of a schedule $\mathcal{T}$, given soft constraints $\mathcal{C}$.
$\overline{\Phi}(\mathcal{T}, \mathcal{C})$	The penalty function of a schedule $\mathcal{T}$, given soft constraints $\mathcal{C}$ and taking into account scaling factors.

$\Phi^*(\mathcal{T}, \mathcal{C})$	The penalty function of a schedule $\mathcal{T}$, given soft constraints $\mathcal{C}$ and adding an extra penalty describing the spread of the schedule.
$\overline{\Phi}^*(\mathcal{T}, \mathcal{C})$	The penalty function of a schedule $\mathcal{T}$, given soft constraints $\mathcal{C}$, adding an extra penalty describing the spread of the schedule and taking into account scaling factors.
$\Psi(\mathcal{T}, \mathcal{C})$	A function describing the *fairness* of a given schedule.
$\overline{\Psi}(\mathcal{T}, \mathcal{C})$	A function describing the *fairness* of a given schedule taking into account scaling factors.

References

[1] Salwani Abdullah, Samad Ahmadi, Edmund K Burke, Moshe Dror, and Barry McCollum. A tabu-based large neighbourhood search methodology for the capacitated examination timetabling problem. *Journal of the Operational Research Society*, 58(11):1494–1502, 2007.

[2] David Abramson. Constructing school timetables using simulated annealing: sequential and parallel algorithms. *Management Science*, 37(1):98–113, 1991.

[3] David Abramson and J Abela. A parallel genetic algorithm for solving the school timetabling problem. 1991.

[4] David Abramson, Mohan Krishna Amoorthy, and Henry Dang. Simulated annealing cooling schedules for the school timetabling problem. *Asia-Pacific Journal of Operational Research*, 16(1):1, 1999.

[5] Leena N Ahmed, Ender Özcan, and Ahmed Kheiri. Solving high school timetabling problems worldwide using selection hyper-heuristics. *Expert Systems with Applications*, 42(13):5463–5471, 2015.

[6] Benchmarking project for (high) school timetabling. `https://www.utwente.nl/ctit/hstt/`. Accessed: 2017-02-09.

[7] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[8] Dominique de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19(2):151–162, 1985.

[9] EURO Working Group on Automated Timetabling archive of papers. `http://watt.cs.kuleuven.be/application-area/educational-timetabling/papers`. Accessed: 2017-02-13.

[10] Werner Junginger. Timetabling in Germany - a survey. *Interfaces*, 16(4):66–74, 1986.

[11] Carlos Lara, Juan J Flores, and Félix Calderón. Solving a school timetabling problem using a bee algorithm. In *Mexican International Conference on Artificial Intelligence*, pages 664–674. Springer, 2008.

[12] Nelishia Pillay. A survey of school timetabling research. *Annals of Operations Research*, 218(1):261–293, 2014.

[13] Andrea Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.

[14] Bernard van Kesteren. The clustering problem in Dutch high schools: changing metrics in search space, 1999.

[15] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[16] RJ Roy Willemen. School timetable construction: algorithms and complexity. 2002.

[17] Defu Zhang, Yongkai Liu, Rym MHallah, and Stephen CH Leung. A simulated annealing with a new neighborhood structure based algorithm for high school timetabling problems. *European Journal of Operational Research*, 203(3):550–558, 2010.