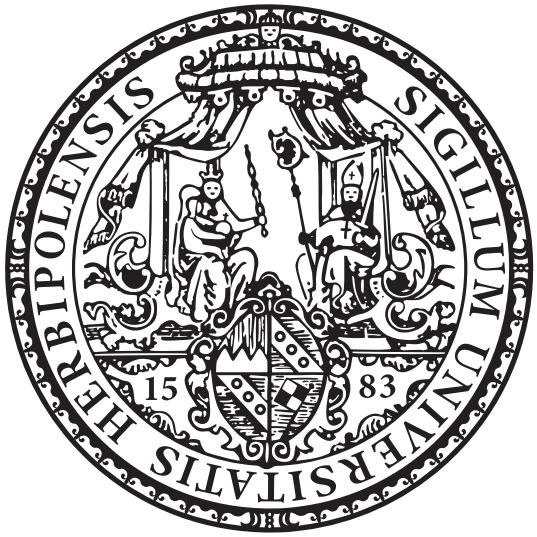
Practical Course Report

A MIP-based Very Large-Scale Neighborhood Search for Timetable Optimization

Andreas Maier

Date of Submission:
Advisor: Prof. Dr. Marie Schmidt



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

1 Einleitung

Timetabling is a problem with many different variations, based on many different real-world applications. Most people have come in contact with timetables multiple times throughout their life, and it is clear that timetables are so ubiquitous that the benefits of well-optimized ones are large. Thus, timetabling is also a popular class of optimization problems to examine and attempt to find solutions for. We take a look at one specific subtype of timetabling problem, the *High-School Timetabling Problem (HSTT)*, which is designed to model timetables in the form they are common at High Schools. We develop an algorithm with the goal of iteratively improving existing timetables that are valid, but not highly optimized. The iterative nature of the algorithm serves the purpose that it can be stopped at any point in time and will return the best found solution. This is in contrast to methods that generate a new and very optimized solution in exchange for a big block of runtime and no option to cancel the algorithm prematurely to optain a valid, but maybe not as optimal solution, and thus not being very adaptive to time and computing power constraints a user might have. In Section 2 we take a brief look at existing research related to our work. In Section 3 we define the problem and afterwards, in Section 4 we describe our algorithm. Section 6 contains some results and performance analyses of our method, using problem instances in the XHSTT format, from the dataset of the *High School Timetabling Project* from the University of Twente.

2 Related Work

For the High-School Timetabling Problem as it is relevant for our analysis, there are really two important contributions that make up the corner stones of our and many similar recent pieces of work in the field. On one side, a data format called XHSTT was created to represent problem instances and solutions [XHSa]. On the other hand, the problem of High-School Timetabling was put into spotlight in the International Timetabling Competition 2011, where the XHSTT format was established both as basis for the problem definition and for the distribution of the problem instances of the competition [PDGK$^+$16]. The collection of real-world problem instances was later extended and is maintained within the High School Timetabling Project by the University of Twente [XHSb]. All instances used in our experiments can be found there, and many pieces of recent work on the High-School Timetabling Problem are based on these data. Due to the complexity of the High-School Timetabling Problem [CK96], heuristic methods are generally more popular than exact ones [TGKS21], as the latter run into issues of obtaining solutions in reasonable amounts of computing time. Such heuristic methods include for example Variable Neighborhood searches (VNS) [SCR13], [FS14], [TSdSC19] or simulated-annealing algorithms [ZLML10], [OOOO20], and they generally consider small neighborhood definitions, while the algorithm provides ways to overcome local optima or plateaus. An example for an exact method is the publication of Kristiansen et al. [KSS15]. More recently, heuristics have also been combined with such mathematical methods to a type of hybrid approach referred to as *matheuristics* [FSC16],

[DdB14]. Especially the matheuristic part of Fonseca et al. [FSC16] served as most important inspiration for our approach: A MIP is used for storing and maintaining the current solution, and an iterative algorithm determines which variables are freed and recalculated while all others remain fixed, solving the model to local optimality at each step. Fonseca et al. choose to select all variables that refer to an element of a random subset of resources (or a random subset of times/events/any variables for alternative neighborhoods) to be reoptimized. The main change of our approach to theirs is that our algorithm is designed to identify a selection of variables whose assigned resources and times have a high chance of being exchangeable. If we think about the MIP as a three-dimensional space of Times × Resources × Events, Fonseca et al.'s selection could be interpreted as freeing variables orthogonal to the chosen random base subset, while our selection could be seen as a cuboid area. A different, more high-level interpretation of our appraoch is a comparison with Large Neighborhood Search algorithms (LNS) like [SKS+12], even though it is not inspired by these. A LNS is similar to VNS in the way that it defines a neighborhood function and iteratively improves the solution through the neighborhood, but it does so untilizing much larger neighborhoods. The selection of variables to be freed and recalculated can be viewed as a large neighborhood, and the MIP optimization process at each step as a standard hill-climbing operation.

3 Problem Formulation

Historically, there are a number of variations for the High School Timetabling Problem, which often differ in the way multiple elements "meet" or which sets are considered. The problem formulation that we use in our implementation is closely oriented on the definition of the XHSTT format [XHSa]: Not only is this useful as we want to use the available database of problem instances in this format, it is also a general and flexible enough definition so that it can be applied to many different requirements. For clarity, the description we provide here is slightly simplified and more focused on the essence of the problem and the process of solving it.

The necessary sets for the basis of the problem definition are:

- T: The set of available *times*. They are non-overlapping timeslots and indexed in order, beginning at 1.
- R: The set of available *resources*. Resources are entities which attend events, and each of them has a *resourceType*. Commonly, this set contains entities like classes (group of students), teachers or rooms.
- E: The set of required *instance events* (short: *events*). An event is a meeting between resources. Beyond the meeting resources, it specifies a *duration*, which is the number of timeslots such a meeting is supposed to take place. Commonly, this set contains courses and the duration refers to its required weekly hours.

- C: A set of constraints that describe the cost of an assignment of resources and timeslots to events. The cost is to be minimized.

Times, resources and events are commonly grouped into subsets (*timeGroups*, *resourceGroups* and *eventGroups*) in order to properly define the circumstances of a specific problem instance. The constraints stored in set C come in two different severities: *Hard constraints* are constraints that are always more costly to violate than any *soft constraint*. These terms are used slightly differently here than it is common in the context of integer programming: "Hard constraints" should be considered to be generally avoidable in practice, but it is not known from the start whether this is actually possible. Soft constraints, on the other hand, are designed for representing preferences that can be fulfilled mostly, but likely not totally. In practice, when designing a MIP model, there is a third severity *feasibility constraints* (These are what is usually called hard constraints): A timetable assignment that violates one of these is not a valid assignment.

The assignment of times, resources and events to each other is formatted in the following way:

- An event specifies several *eventResources*, which resources need to be assigned to. EventResources specify the resourceType an assigned resource is required to have. The eventResources are implicitly applied to the solution events of the event. For example, a an event called "8-grade history course" could have two eventResources, one of which requires a resource of resourceType "Teacher" and one of resourceType "Class"

- An event is divided into a number of *solution events*, which have a duration and the eventResources derived from the event, also referred to as *solution resources* (we will still refer to it as "eventResources" most of the time). The duration of the solution event refers to the number of consecutive timeslots it occupies. The total duration of all solution events of one event is supposed to be equal to the duration of the event. Solution events can be assigned a time, which is its starting time, and each solution resource can be assigned a resource.

As a note, we want to mention that in problem instances, through the XHSTT format specification, a distinction is made between times or resources that are preassigned and therefore fixed as part of the problem, and times or resources that are determined as part of the solving process. Preassignments can be viewed as fixed feasibility constraints.

In general, while hard and soft constraints are explicitly specified in the problem instance formulation through several types of constraints provided by the XHSTT format, feasibility constraints beyond the preassignments are the implicit requirements to a valid timetable: No double assignments of times and resources to solution events, durations of solution events have to sum up to their parents event duration, or links between different variables, depending on the specific implementation. The necessary feasibility constraints and also the hard and soft constraints are explained in more detail in Section 4.1, which also how we implemented our MIP.

4 MIP Formulation

As hinted at above, our method consists of two layers: At its basis, we model the problem as one big MIP, which we use to maintain our current solution. On top of that, we use a greedy selection algorithm to define a set of variables that we want to recompute an optimum for. That way, we construct a subproblem as MIP by freeing the variables of this set while all others remain fixed. The amount of recomputed variables in each iteration can be chosen so we can solve the MIP within a reasonable time. As briefly outlined above, this method has similarities with LNS algorithms, however, to better facilitate the thoughts behind the design of our method, in the following, we describe it independently and only point out analogies when they help understanding the role or purpose of a specific component. In this section, we precisely define the MIP which is the backbone of the method. Afterwards, in Section 5, we describe the selection algorithm that generates the set of variables to recompute.

4.1 MIP problem formulation

Our MIP is derived from the specification of the XHSTT format. Apart from that, for a slightly more detailed explanation of the elements of the MIP, we recommend taking a look at [FSC16], which implements a very similar MIP. In our definition, we will use entity names tied to the corresponding object in our code, which in turn means that there may be inconsistencies with the naming in aforementioned sources.

Let us start with the sets the MIP uses:

- T: The set of times (or timeslots), indexed with integers from 0 ascending
- R: The set of resources
- E: The set of events
- C: The set of constraints
- ER: The set of eventResources
- SE: The set of solution events (see construction below)
- TStart: helper set (see construction below)

Furthermore, we introduce these constants and reference attributes:

- $t_d = dummyTime$: Represents that no time has been assigned; dummyTime $\in T$
- $r_d = dummyResource$: Represents that no resource has been assigned; dummyResource $\in R$. We write $r_d(rType)$ to indicate that the slot would have had type $rType$.
- er.rType / r.rType: The resourceType of $er \in R$ / $r \in R$, respectively.

- sol.ER: The set of eventResources of $sol \in SE$
- c.appliesTo: The set of points of application of $c \in C$
- e.SE: The set of solution events belonging to $e \in E$ (including inactive ones, see construction below)
- e.duration / sol.duration: The duration of $e \in E$ / $sol \in SE$
- t.index: The index of $t \in T$. Times are indexed from 0 ascending, in the order they appear in the week.
- er.role: Attribute defined by XHSTT for each $er \in ER$, which is a unique identifier inside the respective event. It is used as reference in some constraints that apply to eventResources.

Our main set of decision variables is going to be x, with an element $x[sol][er][r][t]$ being 1 if and only if solution event sol has a eventResource er, r is assigned to er and t is the starting time of sol, and being 0 otherwise. If we construct x this way, with one element for each element in the cartesian product $SE \times ER \times R \times T$, this would lead to a large number of variables that we know cannot be 1, because sol does not have an eventResource er or r cannot be assigned to er because of different resourceTypes. Therefore, we construct a set beforehand which contains all and only the valid elements of $SE \times ER \times R$:

$$wTuples = \{(sol, er, r) | sol \in SE, er \in sol.ER, r \in R, er.rType = r.rType\} \quad (1)$$

Then we can define x like:

$$x = \{x[(sol, er, r)][t] | (sol, er, r) \in wTuples, t \in T\} \quad (2)$$

Additionally, we define some helper variables:

- $w = \{w[(sol, er, r)] | (sol, er, r) \in wTuples\}$
  with $w[(sol, er, r)] = 1 \iff sol$ has an eventResource er and r is assigned to it.
- $y = \{y[sol][t] | sol \in SE, t \in T\}$
  with $y[sol][t] = 1 \iff sol$ has starting time t
- $u = \{u[sol] | sol \in SE\}$
  with $u[sol] = 1 \iff sol$ is assigned a valid starting time (other than dummyTime)
- $v = \{v[t][r] | t \in T, r \in R\}$
  with $v[t][r] =$ how often is r scheduled for t
- $q = \{q[r][t] | r \in R, t \in T\}$
  with $q[r][t] = 1 \iff r$ is scheduled at least once for t

As mentioned, SE is constructed in a special way: Because we do not know the durations of the solution events right away, for any given event e, we include the superset of all possible combinations of durations. For example, if event e has duration 4, we need to include 4 solution events with duration 1, 2 with duration 2, 1 with duration 3 and 1 with duration 4. With these, we can represent all possible combinations of durations so that the total duration of e is not exceeded. In turn, this has the effect that several elements in SE are "inactive". We assign dummyTime to these, and $u[sol] = 0$ for them. $TStart$ is constructed the following way: $TStart[(sol, t0)] \subseteq T$ is the subset of times that sol occupies if starting time $t0$ is assigned.

These are the feasibility constraints of the MIP:

- Only 1 starting time $t \in T$ per $sol \in SE$, and only one resource $r \in R$ per eventResource $er \in sol.ER$:
  - $SolEr = \{(sol, er) | (sol, er, \_ \in wTuples\}$
  - $R'(sol, er) = \{r | (sol, er, r) \in wTuples\}$

$$\forall (sol, er) \in SolEr : \sum_{t \in T, r \in R((sol,er))} x[(sol, er, r)][t] = 1 \tag{3}$$

- The amount of assigned resources to sol equals the amount of event resources of sol $|sol.ER|$ (includes no double assignments):
  - $ErR(sol) = \{(er, r) | (sol, er, r \in wTuples\}$

$$\forall sol \in SE, t \in T : \sum_{(er,r) \in ErR(sol)} x[(sol, er, r][t] = |sol.ER| \cdot y[sol][t] \tag{4}$$

- sol cannot be longer than the remaining timeslots after its starting time t:
  - $T_+ = T \setminus \{t_d\}$

$$\forall sol \in SE, t \in T_+ \text{ with } t.index + sol.duration - 1 \geq |T_+| : y[sol][t] = 0 \tag{5}$$

- The duration of an event equals the total duration of its active solution events:

$$\forall e \in E : \sum_{sol \in e.SE} sol.duration \cdot u[sol] = e.duration \tag{6}$$

- If resources are preassigned by the problem instance, fix the respecitve variables:
  - $R_{pre} = \{(er, r)$ that are preassigned $\}$

$$\forall (sol, er, r) \in wTuples \text{ with } (er, r) \in R_{pre} : w[(sol, er, r)] = u[sol] \tag{7}$$

- Link w and x:

$$\forall (sol, er, r) \in wTuples : \sum_{t \in T} x[(sol, er, r)][t] = w[(sol, er, r)] \tag{8}$$

- Link w and u:

- $SolEr = \{(sol, er) | (sol, er, \_ \in wTuples\}$
- $R'(sol, er) = \{r | (sol, er, r) \in wTuples\}$
- $ER_{pre} = \{er \in ER | er$ has a preassignment $\}$

$$\forall (sol, er) \in SolEr \setminus ER_{pre} : \sum_{r \in R'(sol,er)} w[(sol, er, r)] \leq u[sol] \tag{9}$$

- Link y and u:

$$\forall sol \in SE : \sum_{t \in T \setminus t_d} y[sol][t] \leq u[sol] \tag{10}$$

Additionally, we add the following constraints. Theoretically, these relations are covered by above constraints, but in practice, solvers can draw a lot of value from these constraints, shortcutting more complex entanglement of variables with very basic equations:

- Exactly one assigned starting time $t \in T$ per $sol \in SE$ (via y):

$$\forall sol \in SE : \sum_{t \in T} y[sol][t] = 1 \tag{11}$$

- Exactly one resource $r \in R$ per solution resource (sol, er):
  - $SolEr = \{(sol, er) | (sol, er, \_ \in wTuples\}$
  - $R'(sol, er) = \{r | (sol, er, r) \in wTuples\}$

$$\forall (sol, er) \in SolEr : \sum_{r \in R'(sol,er)} w[(sol, er, r)] = 1 \tag{12}$$

Our hard and soft constraints are pre-established in the instance file within the constraints (set C) and generally work the same way:

- They contain an indicator whether the constraint is supposed to be a hard or a soft constraint
- They define a set of points of application (c.appliesTo), which has at least one element. The type of the element depends on the constraint. The constraint can be violated seperately for each point of application.
- The cost (called $Wcost$ = weightedCost) of violating the constraint at one point of application is calculated by the formula:

$$Wcost = weight \cdot costFunction(cost)$$

weight and costFunction are given in the instance file, and *cost* depends on the solution of the program and generally refers to how severely the constraint is violated. In the XHSTT specification, cost is called deviation. weight is an integer number and costFunction can be 'Linear', 'Quadratic' or 'Step'. As all of the instances we use only make use of Linear costFunctions, we can omit that argument and simplify to

$$Wcost = weight \cdot cost$$

- Depending on the constraint, there might be additional necessary parameters to determine the cost, which we refer to like $c.additionalParameter$

Before we list the constraints, we introduce another variable for our MIP to capture the cost:

- $cost = \{cost[(c, poa)] | c \in C, poa \in c.appliesTo\}$
  with $cost[(c, poa)] \geq 0$ being the cost of c at the point of application poa
- $Wcost = \{Wcost[(c, poa)] | c \in C, poa \in c.appliesTo\}$
  with $Wcost[(c, poa)] = c.weight \cdot cost[(c, poa)]$ being the weighted cost of c at poa.
- For several constraints we need additional helper variables, which we all store in a vector s (*subcost* in the code). Depending on the constraint, s may have different tuples to reference one variable of the vector.

These are the constraints defined by XHSTT, which can be hard or soft constraints. We denote $C' \subset C$ the subset of constraints of the respective type. All constraints are defined $\forall c \in C'$ with the respective C' and $poa \in c.appliesTo$, therefore, the names c and poa always refer to elements of these respective subsets, and we omit this expression at the constraints definition:

- AssignResourceConstraint: Each solution resource should be assigned a resource. The cost of not assigning a resource (or assigning dummyResource) at an eventResource $poa \subset ER$ is the sum of the solution events containing unassigned solution resources that refer to poa:
  - $wTuples' = \{(sol, er, r) \in wTuples | sol \in poa.SE, er.role = c.role, r \in R \setminus \{r_d\}\}$

$$cost[(c, poa)] = poa.duration - \sum_{(sol,er,r) \in wTuples'} sol.duration \cdot w[(sol, er, r)] \quad (13)$$

- AssignTimeConstraint: Each solution event should be assigned a starting time. The cost of not assigning a time (or assigning dummyTime) to solution events of the events $poa \subset E$ is the combined durations of the unassigned solution events $\in poa.SE$:

$$cost[(c, poa)] = \sum_{sol \in poa.SE} ([(u[sol] - \sum_{t \in T \setminus \{t_d\}} y[sol][t]) \quad (14)$$

- SplitEventsConstraint: The cost of this constraint at one given event $poa \subset E$ is the amount of solution events $sol \in poa.SE$ whose $sol.duration$ is outside the range $[c.minimumDuration, c.maximumDuration] = [c.minDur, c.maxDur]$, plus the amount by which $|poa.SE|$ falls short of $c.minimumAmount = c.minAmt$ or exceeds $c.maximumAmount = c.maxAmt$:
  - $dur[(c, poa, sol)]$ is the amount of solution events outside the duration limits
  - $amt[(c, poa)]$ is the amount by which $|poa.SE|$ is outside the amount limits

$\forall sol \in poa.SE$:

$$poa.duration \cdot dur[(c, poa, sol)] \geq (c.minDur - sol.duration) \cdot u[sol] \quad (15)$$

$$poa.duration \cdot dur[(c, poa, sol)] \geq (sol.duration - c.maxDur) \cdot u[sol] \quad (16)$$

$$dur[(c, poa, sol)] \geq 0 \quad (17)$$

$$amt[(c, poa)] \geq c.minAmt - \sum_{sol \in poa.SE} u[sol] \quad (18)$$

$$amt[(c, poa)] \geq \sum_{sol \in poa.SE} u[sol] - c.maxAmt \quad (19)$$

$$amt[(c, poa)] \geq 0 \quad (20)$$

$$cost[(c, poa)] \geq \sum_{sol \in poa.SE} dur[(c, poa, sol)] + amt[(c, poa)] \quad (21)$$

- DistributeSplitEventsConstraint: This constraint places limits on how many solution events sol of a given event $poa \in E$ should have given duration $sol.duration$. The cost at one poa is the amount of solution events $sol \in poa.SE$ falling short of $c.minimum$ or exceeding $c.maximum$:

$$cost[(c, poa)] \geq c.minimum - \sum_{sol \in poa.SE} u[sol] \quad (22)$$

$$cost[(c, poa)] \geq \sum_{sol \in poa.SE} u[sol] - c.maximum \quad (23)$$

$$cost[(c, poa)] \geq 0 \quad (24)$$

- PreferResourcesConstraint: PreferResourcesConstraints work similar to AssignResourceConstraints, with the added condition that a resource from a specified subset $prefRes(c) \subset R$ is assigned to $poa \in ER$. Unassigned event resources are ignored by these constraints. $prefRes(c)$ can be defined through resourceGroups or seperate resources; the set of allowed resources is the union over these. The cost at one poa is the sum of durations of misassigned solution events with poa as one of their event resources.
  - $unpref(c) = \{(sol, er, r) \in wTuples | er = poa, r \notin prefRes(c) \cup \{r_d\}\}$ is the set of disallowed resources to be assigned to $sol \in poa.SE$.

$$cost[(c, poa)] \geq \sum_{(sol,er,r) \in unpref(c)} sol.duration \cdot w[(sol, er, r)] \quad (25)$$

- PreferTimesConstraint: Analogous to PreferResourcesConstraints, PreferTimesConstraints are like AssignTimeConstraints with the added condition of a time being from a specified subset $prefTimes(c) \subset T$ when assigned to an event $poa \in E$. Like with PreferResourceConstraints, events that no time is assigned to (or dummyTime is assigned to) are ignored, and $prefTimes(c)$ is defined through a combination of

timeGroups and times as the set of allowed times, which is their union. The cost at one *poa* is the sum of durations of misassigned solution events $sol \in poa.SE$. If an optional value $c.duration$ other than 0 is given, all solution events sol with $sol.duration \neq c.duration$ are ignored.

- $unpref(c) = T \setminus (prefTimes(c) \cup \{t_d\})$
- $SE' = \begin{cases} \{sol \in poa.SE | sol.duration = c.duration\} & \text{, if c.duration} > 0 \\ poa.SE & \text{, otherwise} \end{cases}$

$$cost[(c, poa)] \geq \sum_{sol \in SE', t \in unpref(c)} sol.duration \cdot y[sol][t] \tag{26}$$

- AvoidSplitAssignmentConstraint: When an event is split into multiple solution events, it is generally allowed for different resources to be assigned to the same event resource over different solution events. For example, a course might take place in different rooms throughout the week. However, there are some event resource where it is desirable to not allow this. With the AvoidSplitAssignmentConstraint, we penalize the situations where such a split assignment occurs in an undesirable spot.
  - $s[(c, poa, r)]$ is the amount of different assignments $r \in R$ to an event resource er with $er.role = c.role$. Event resources that are preassigned are excluded, as they technically are not assigned a resource in the solution at all.

$$wTuples'(role) = \{(sol, er, r) \in wTuples | er.role = role, r \text{ not preassigned}\} \tag{27}$$

$$s[(c, poa, r)] = \sum_{(sol, er, r) \in wTuples'(c.role)} w[(sol, er, r)] \tag{28}$$

$$cost[(c, poa)] \geq \sum_{r \in R} s[(c, poa, r)] - 1 \tag{29}$$

$$cost[(c, poa)] \geq 0 \tag{30}$$

- SpreadEventsConstraint: The solution events of an event should be spread out in time. For this, an attribute $c.tgs$ ("tgs" for timeGroups) is given, which is a set of TimeGroups, and in each of them, the number of solution events with their starting time in the respective timeGroup tg should be between $c.tg.minimum$ and $c.tg.maximum$. (The $minimum$ and $maximum$ are not inherently attributes of timeGroups, so they are defined by the constraint c, but still they have to be defined per tg.)
  - $s[(c, poa, tg)]$ is a decision variable for whether the event poa has too little or too many solution events within the timeGroup $tg \in c.tgs$.

$$s[(c, poa, tg)] \geq c.tg.minimum - \sum_{sol \in poa.SE, t \in tg} y[sol][t] \tag{31}$$

$$s[(c, poa, tg)] \geq \sum_{sol \in poa.SE, t \in tg} y[sol][t] - c.tg.maximum \tag{32}$$

$$s[(c, poa, tg)] \geq 0 \tag{33}$$

$$cost[(c, poa)] \geq \sum_{tg \in c.tgs} s[(c, poa, tg)] \tag{34}$$

- LinkEventsConstraint: Linked events should be assigned the same times, so they run in parallel. A penalty arises when there are times that some of the set of linked events are occupying, but not all of them are.
  - $busyInE[(e, t)]$ is the decision variable for whether $e \in E$ is busy in $t \in T$. Events are busy not only during their assigned starting times, but during all times that are covered by their duration. The calculation of $busyInE$ is not dependent on any constraints and therefore only performed once in advance to creating the LinkEventsConstraints. Per definition, the values can be easily derived from $TStart$, which has been defined above.
  - $busyInPoa[(c, poa, t)]$ works analogously to $busyInE$, but extended to eventGroups poa
  - $busyDiff[(c, poa, t)]$ is the decision variable for whether the given time $t \in T$ is busy by poa, but not by every $e \in poa$.

  Outside the dependence on the constraint, it is:

$$\forall e \in E, t \in T \setminus \{t_d\} :$$

$$\forall sol \in e.SE : busyInE[(e, t)] \geq \sum_{t0 \in TStart[(sol,t)]} y[sol][t0] \tag{35}$$

$$busyInE[(e, t)] \leq \sum_{sol \in e.SE, t0 \in TStart[(sol,t)]} y[sol][t0] \tag{36}$$

  Note here that we place a cap on $busyInE$, because otherwise there can occur cases where the variable assumes 1 even though it is not busy, which would lead to incorrect calculations. With respect to the LinkEventConstraints, it is $\forall e \in poa, t \in T \setminus \{t_d\}$:

$$busyInPoa[(c, poa, t)] \geq busyInE[(e, t)] \tag{37}$$

$$busyDiff[(c, poa, t)] \geq busyInPoa[(c, poa, t)] - busyInE[(e, t)] \tag{38}$$

  Then is:

$$cost[(c, poa)] \geq \sum_{t \in T \setminus \{t_d\}} busyDiff[(c, poa, t)] \tag{39}$$

- OrderConstraint: Certain Events have to take place before others. Because none of our problem instances use this constraint and because it is rather complex to formulate mathematically, we did not implement this constraint. We merely mention it here for completeness.

- AvoidClashesConstraint: Certain resources should not attend two events at the same time. Any assignments that make one of these resources attend an event beyond the first are penalized.
  - $s[(c, poa, t)]$ signals the amount of clashes in $t \in T \setminus \{t_d\}$.

$$\forall t \in T \setminus \{t_d\} : s[(c, poa, t)] \geq v[t][poa] - 1 \tag{40}$$

$$cost[(c, poa)] \geq \sum t \in T \setminus \{t_d\} s[(c, poa, t)] \tag{41}$$

- AvoidUnavailableTimesConstraint: Sometimes, resources are unavailable during specific times, during which they cannot attend any event. If it happens regardless, it is penalized:

$$cost[(c, poa)] \geq \sum t \in c.unavailableTq[poa][t] \tag{42}$$

- LimitIdleTimesConstraint: A resource $r \in$ is considered *idle in* $t \in T$ with respect to a given timeGroup tg, if it is not busy in t, but is busy in a time earlier in tg and one later in tg. In other words, it is a gap between busy times where the resource is not busy. LimitIdleTimesConstraints limit the amount of idle times per timeGroup, both optionally with a minimum and and a maximum amount of idle times permitted.
  - $busyF[(c, poa, tg)]$ is the ordinal number of the *first* busy time within timeGroup tg.
  - $busyL[(c, poa, tg)]$ is the ordinal number of the *last* busy time within timeGroup tg.
  - $idleAmt[(c, poa, tg)]$ is the amount of idle times between $busyF$ and $busyL$ inside of timeGroup tg.

  We denote the ordinal number of a time within tg as $t.index(tg)$. Note that a valid timeGroup $tg \in c.tgs$ is required to consist only of subsequent times, i.e. it is not allowed to have gaps itself. Usually, these timeGroups are the days of the week.

$$\forall tg \in c.tgs :$$
$$\forall t \in tg :$$
$$busyF[(c, poa, tg)] \leq |tg| - (|tg| - t.index(tg)) \cdot q[poa][t] \tag{43}$$
$$busyF[(c, poa, tg)] \geq 0 \tag{44}$$
$$busyL[(c, poa, tg)] \geq t.index(tg) \cdot q[poa][t] \tag{45}$$
$$idleAmt[(c, poa, tg)] = busyL[(...)] - busyF[(...)] + 1 - \sum_{t \in tg} q[poa][t] \tag{46}$$

The cost is calculated as how much the amount of idle times is outside the range given by $c.minimum$ and $c.maximum$.

$$cost[(c, poa)] \geq c.minimum - \sum_{tg \in c.tgs} idleAmt[(c, poa, tg)] \tag{47}$$

$$cost[(c, poa)] \geq \sum_{tg \in c.tgs} idleAmt[(c, poa, tg)] - c.maximum \tag{48}$$

$$cost[(c, poa)] \geq 0 \tag{49}$$

- ClusterBusyTimesConstraint: Given a set of timeGroups, these constraints limit in how many of them the resource $poa \in R$ in question may be busy. Both a minimum and a maximum is given, and the cost equals how far outside the given range the number of timeGroups poa is busy in lies. An example how this constraint could be used is to enforce that a class has scheduled afternoon lessons only on at most two days per week.
  - $busyInTg[(c, poa, tg)]$ is the decision variable for whether resource $poa \in R$ is busy in at least one time $t \in T$ in timeGroup tg.

$$\forall tg \in c.tgs :$$
$$\forall t \in tg : busyInTg[(c, poa, tg)] \geq q[poa][t] \tag{50}$$
$$busyInTg[(c, poa, tg)] \leq \sum_{t \in tg} q[poa][t] \tag{51}$$

Then the cost is:

$$cost[(c, poa)] \geq c.minimum - \sum_{tg \in c.tgs} busyInTg[(c, poa, tg)] \tag{52}$$

$$cost[(c, poa)] \geq \sum_{tg \in c.tgs} busyInTg[(c, poa, tg)] - c.maximum \tag{53}$$

$$cost[(c, poa)] \geq 0 \tag{54}$$

- LimitBusyTimesConstraint: Complement to ClusterBusyTimesConstraints, where the limit is placed on the amount of timeGroups a resource $poa \in R$ may be busy in, here we limit the amount of times r is busy within a timeGroup tg. Again, a minimum to maximum range is given and the cost is calculated as the sum, over all timeGroups, of the deviations of being outside the range. Note that if poa has 0 busy times in a timeGroup, then no cost arises for that timeGroup and poa.
  - $busyInTg[(c, poa, tg)]$ is the same as in ClusterBusyTimesConstraints: It is the decision variable for whether resource $poa \in R$ is busy in at least one time $t \in T$ in timeGroup tg. We therefore do not repeat the equations that define $busyInTg$ here and refer to the paragraph about ClusterBusyTimesConstraints.

– $costInTg[(c, poa, tg)]$ is the integer variable that counts how far outside the range given by $c.minimum$ and $c.maximum$ the amount of busy times is within timeGroup tg.

$\forall tg \in c.tgs$:

$$costInTg[(c, poa, tg)] \geq c.minimum - \sum_{tg \in c.tgs} busyInTg[(...)] - |tg| \cdot (1 - busyInTg[(...)]) \quad (55)$$

$$costInTg[(c, poa, tg)] \geq \sum_{tg \in c.tgs} busyInTg[(...)] - c.maximum - |tg| \cdot (1 - busyInTg[(...)]) \quad (56)$$

$$costInTg[(c, poa, tg)] \geq 0 \quad (57)$$

Then the cost is:

$$cost[(c, poa)] \geq \sum_{tg \in c.tgs} costInTg[(c, poa, tg)] \quad (58)$$

- LimitWorkloadConstraint: These constraints limit the workload (wl) a resource $poa \in R$ may have. Workload is defined as $wl[(sol, er, r)] = \frac{sol.duration \cdot wl(er)}{e.duration}$ where $sol \in e.SE$ und $(sol, er, r) \in wTuples$. In cases where in the problem instance no workload for sol is given, it is derived from its enclosing event e, and when no workload for an event $e \in E$ is given, it is equal to $e.duration$ for that event. Because $wl[(sol, er, r)]$ is constant, it is calculated only once on the initial creation of the MIP.
  - $wl[(c, poa)]$ is the total workload across all of $wl[(sol, er, r)]$ with $r = poa$ that poa is subjected to by the assignment of resources.
  - $SolER(poa) = \{(sol, er) | (sol, er, r) \in wTuples, r = poa\}$

$$wl[(c, poa)] = \sum_{(sol,er) \in SolER(poa), t \in T \setminus \{t_d\}} wl[(sol, er, poa)] \cdot x[(sol, er, poa)][t] \quad (59)$$

Again, the cost is calculated as how much the workload of poa is outside the range given by $c.minimum$ and $c.maximum$.

$$cost[(c, poa)] \geq c.minimum - wl[(c, poa)] \quad (60)$$

$$cost[(c, poa)] \geq wl[(c, poa) - c.maximum \quad (61)$$

$$cost[(c, poa)] \geq 0 \quad (62)$$

As objective function, we want to minimize the total cost of the hard and soft constraints, which is the sum over all constraints $c \in C$. As mentioned, within the definition

of c, there is an attribute $c.required$ that tells us whether the constraint is a hard constraint ($c.required = true$) or a soft constraint ($c.required = false$).

$$objHard = \sum_{c \in C, c.required = true} \sum_{poa \in c.appliesTo} cost[(c, poa)] \tag{63}$$

$$objSoft = \sum_{c \in C, c.required = false} \sum_{poa \in c.appliesTo} cost[(c, poa)] \tag{64}$$

A single point of cost in hard constraints is always more expensive than any number of points of cost in soft constraints. One way to handle these two seperate objective functions when we solve the model for optimality, is to first solve only for $objHard$ while ignoring the soft constraints, until the optimal value is found, then fix $objHard$ to that value and solve for $objSoft$. We prefer this over a compound function that weighs every constraint in $objHard$ significantly more than the total $objSoft$ could ever be, because it significantly increases the runtime of our approach. Especially considering the fact that all problem instances we use have known solutions with $objHard = 0$, the two-stage approach is not as prone to the fact that it is hard to determine when we reached an optimal value for $objHard$; as long as it is not 0, we continue to search for a solution that has $objHard = 0$. Generally, solving this model with a generic solver is not practically feasible once the problem instance is no longer of unrealistically small size. We therefore use an iterative search to find good solutions:

5 Selection Algorithm

The main motivation for using an iterative search, which in our case turns out to have much in common with a Large Neighborhood Search (LNS), is not only the more practical runtimes to arrive at solutions that could be considered good enough. The primary use case we designed our algorithm for is one where a somewhat good, potentially hand-made solution is already known and we want to improve this solution.
Therefore, for our search, we assume a given valid and currently best known solution, and follow these steps to iteratively yield new best known solutions:

1. Fix the currently best solution in the MIP.
2. Determine some variables to free.
3. Recalculate an optimal solution, potentially changing the values of the freed, but not those of the fixed variables.
4. Repeat.

5.1 Fixing the best solution

Fix the currently best solution in the MIP. In practice, it is sufficient to do this by introducing constraints like $y[sol][t] = 1$ for all y that are actually 1 in the current solution. The variables we fix this way are y, w and u. We do not fix x directly, as we

can infer the values easily from y, w and u. However, because some relations between the variables are only defined through x, we notice a lot of time getting lost to infer the 0-values (variables with value 0) from the fixed 1-values (variables with value 1). We counteract this by introducting the following two cuts:

$$\forall sol \in SE : \sum_{t \in T} y[sol][t] = 1 \tag{65}$$

- $SolEr = \{(sol, er) | (sol, er, \_ \in wTuples\}$
- $R'(sol, er) = \{r | (sol, er, r) \in wTuples\}$

$$\forall (sol, er) \in SolEr : \sum_{r \in R((sol,er))} y[(sol, er, r)] = 1 \tag{66}$$

5.2 Freeing some variables

Generally, freeing a variable is done by removing the corresponding constraint introduced when fixing the variables. Because these constraints only exist for 1-values, we only have to determine which 1-values we want to free in the current iteration; 0-values are never explicitly fixed.
When designing the algorithm determining which variables to free, our primary goal was to keep the number of freed variables low while maximizing the amount of other feasible solutions that could be reached by changing only the freed variables. To be a little more concrete: If we choose to free 5 1-values, but they are so unrelated that resorting to the previous solution is the only solution that is feasible in the current configuration, then the iteration cannot lead to improvement. Instead we want to choose to free 5 1-values that are related in a way so that as many other solutions are feasible in the current configuration. We design the following algorithm for the purpose of determining which variables to free. We accumulate the variables to free in the sets $E_u \subset E$, $R_u \subset R$ and $T_u \subset T$. Furthermore, we define:

- $w1 = \{(sol, er, r) \in wTuples | w[(sol, er, r)] = 1\}$ contains all index tuples where w is 1.
- $R(e) = \{(er, r) | (sol, er, r) \in w1, sol \in e.SE\}$ contains all resources (paired with their respective event resource) that are assigned to event e at least once.
- $E(r) = \{e \in E | (sol, er, r) \in w1, sol \in e.SE\}$ contains all events that resource r is assigned to at least once.
- $T(e) = \{t \in T |$e is busy in t$\}$ are all times assigned to event e.
- $wRand(S)$ picks one element of S randomly by a custom weighted random algorithm. Probability to be picked is increased if there is a large intersection with T_u or if related constraints are violated.

1 $E_u, R_u, T_u = \emptyset$
2 **while** $|E_u| < params.unassignSize$ **do**
3 $\quad e_0 =$ random from E_u if possible, otherwise random from $\{e \in E || R(e)| > 0\}$
4 $\quad (er_0, r_0) =$ random from $R(e_0)$
5 $\quad$ **if** $r_0 = r_d(er_0.rType)$ **then**
6 $\quad\quad R_u = R_u \cup \{r_0\}$
7 $\quad$ **if** $T_u = \emptyset$ **then**
8 $\quad\quad T_u = \bigcup_{e \in E_u} T(e)$
9 $\quad wElems = \{\}$
10 $\quad R' = \{r \in R | (sol, er, r) \in wTuples, sol \in e_0.SE, er = er_0\} \setminus \{r_d, r_0\} \cup R_u$
11 $\quad$ **for** $r' \in R'$ **do**
12 $\quad\quad$ **for** $e' \in E(r_1) \setminus E_u$ **do**
13 $\quad\quad\quad wElems = wElems \cup (e', r')$
14 $\quad (e_1, r_1) = wRand(wElems)$
15 $\quad E_u = E_u \cup \{e_1\}$
16 $\quad R_u = R_u \cup \{r_1\}$
17 $\quad T_u = T_u \cup T(e_1)$
18 Free all $y[sol][t]$ where $sol \in e.SE$, $e \in E_u$ and $t \in T_u \cup \{t_d\}$
19 Free all $w[(sol, er, r)]$ where $sol \in e.SE$, $e \in E_u$ and $r \in R_u \cup \{r_d\}$
20 Free all $u[sol]$ where $sol \in e.SE$ and $e \in E_u$

We start the algorithm Line 1 with empty E_u, R_u and T_u. We want to find some elements to put into E_u and R_u of which we later free related variables. T_u contains all the times that any of the events in E_u is busy at, plus t_d. We add new variables to E_u until it has a certain size, given by an input parameter $params.unassignSize$.
In every iteration, we take one of the previously picked events e_0 and a resource r_0 that is assigned to it, then find the set of all resources R' that could replace r_0 (without regarding any of the MIP constraints). The important qualifying factor here is that the resourceType matches. From that, we generate all possible pairs (e', r') where r' is in R', so it has the ability to replace r_0, and where e' is another event r' is assigned to. From these pairs, we pick a random element, weighted by these two factors: If the intersection between T_u and $T(e')$ is larger, the probability for the pair to be picked is larger. The probability is also increased if e' or r' is involved in hard/soft constraints that are currently violated. With the first factor, we intent to increase the likelyhood that the newly added variables to be freed can actually be swapped or interchanged with the previously chosen in some way. The second factor aims to make iterations of our local search algorithm more likely to yield an improvement in objective value, because we try to optimize around the variables that make the objective value worse. Once we picked a pair (e_1, r_1), we add e_1 to E_u and r_1 to R_u, and we update T_u. Once we have enough elements in E_u, we proceed with freeing the variables associated with the

elements of E_u and R_u if they are also associated with elements of T_u or have dummy values. After that, we can proceed with the next step, which is solving the model again.

5.3 Summary

Now as we have outlined the components, here is how the calculation proceeds from the beginning:

1. Start the program with a problem instance, potentially with a starting solution to use, and construct the MIP.
2. Solve the MIP once with objective function $f(x) = 1$. This initial solve is necessary to load the starting solution, or, if none is given, generate a starting solution with dummy values in all assignments. The objective function is not constraining anything, most notably the cost variable, so all the feasibility constraints, but especially the preassignment and starting solution constraints can easily be fulfilled.
3. Solve the MIP once with objective function $f(x) = objHard$. Cap the current value for $objHard$ by adding the constraint $objHard \leq currentValue$.
4. Solve the MIP once with objective function $f(x) = objSoft$.
5. Fix all 1-values in the MIP.
6. Determine and free variables according to Line 1.
7. Recalculate the MIP with objective function $f(x) = objHard$. If the new objective value equals 0, change to $f(x) = objSoft$ for the rest of the calculation.
8. Repeat the steps 5.-7. until the timeLimit is reached or some other condition is fulfilled that terminates the program.

6 Experiment

Note that this section is more of a record of our experimental approaches, results and conclusions rather than a display of just the final, best or most interesting result. However, to quickly summarize beforehand, we primarily conduct two experiments, that is testing the algorithm's ability to construct a timetable from scratch, and examining how well the algorithm performs on the task of improving solutions that are already somewhat good. During the experiments, we also tested different hyperparameter values, most notably $unassignSize$, which controls the size of one MIP subproblem. We find that higher values generally perform better, leading to better solutions faster and getting stuck in local optima less frequently.

Tab. 1: Size of base sets in problem instances

	Times	Teachers	Rooms	Classes	Events	Total duration
AustraliaTES99	30	37	26	13	308	806
BrazilInstance1	25	8	–	3	21	75
BrazilInstance2	25	14	–	6	63	150
BrazilInstance3	25	16	–	8	69	200
BrazilInstance4	25	23	–	12	127	300
BrazilInstance5	25	31	–	13	119	325
BrazilInstance6	25	30	–	14	140	350
BrazilInstance7	25	33	–	20	205	500
ItalyInstance4	36	61	–	38	748	1101
SAWoodlands2009	42	40	–	30	278	1353

6.1 Code, data and computational setup

Code of this project can be found on [Cod]. The problem instances we used are available on [XHSb], where a number of real-world instances have been collected and provided in the XHSTT format. Table 1 displays the sizes of the problem instances we use. For our computations, we use the machines of the Julia 2 computing cluster of the university of Würzburg. For each run of our program with one set of parameters and one problem instances, we set a timelimit of 12h 30min on the cluster process. Another set of important parameters that we fix but are not part of the input parameters are the weights in wRand (the custom weighted random function for choosing our next element for E_u and R_u). We weigh all violated constraints by the same value of 5. For the weight for the overlap with T_u we choose a weight of 20 multiplied by $\sqrt{l}$, where l is the size of the overlap $T_u \cap T(e')$. That way, larger overlaps are weighted more than smaller ones, but this effect is not as big as the overlap grows. Other parameters are included in the input parameters, described in more detail in the code documentation, and their values can be found in the respective scripts that were used to run the experiments.

•

6.2 Experiments and Results

6.2.1 Experiment 1: Construction from scratch

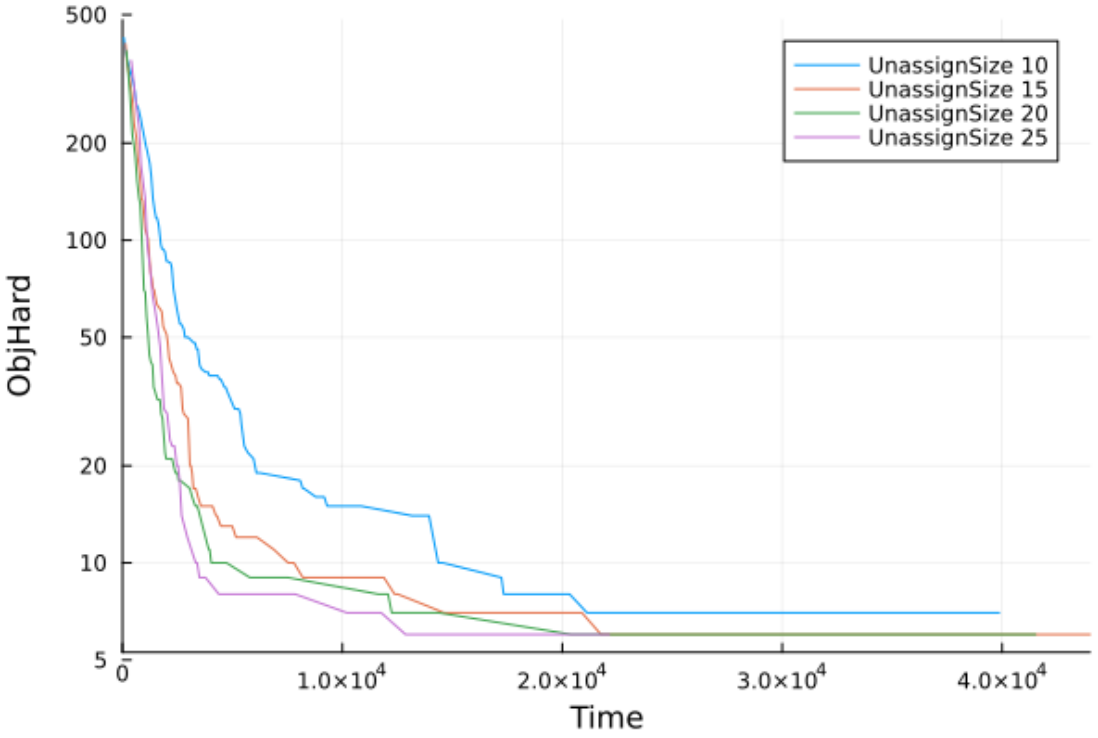
The first set of experiments that we conducted is assessing the ability to construct timetables from scratch. As mentioned, we set all assignments to dummy variables (t_d and r_d) and use that as a trivial starting solution. We initially run this experiment five times, on all instances and for $unassignSize \in \{10, 15, 20\}$. In runs 4 and 5 we also included $unassignSize = 25$. When plotting the objective value ($objHard$) against the time the program was running (example graphs Figure 1; more extensive in the appendix, Figure 6 to Figure 15), we notice the following effects: Firstly, we observe that for our

examination and scope, it is sensical to stick to the medium-sized problem instances, these being BrazilInstance 5 and 6, and adjacently 4 and 7. The smaller instances, namely BrazilInstance 1-3, reach $objHard = 0$ relatively quickly (Figure 6, Figure 7 and Figure 8), at which point improvements are no longer visible on the graph. On the other hand, the larger instances, which are the non-brazil instances, do not really reach a point where it seems as if we were close to a "good" solution, i.e. there is no sign we are near a local optimum and the objective values are still high (Figure 13, Figure 14 and Figure 15).

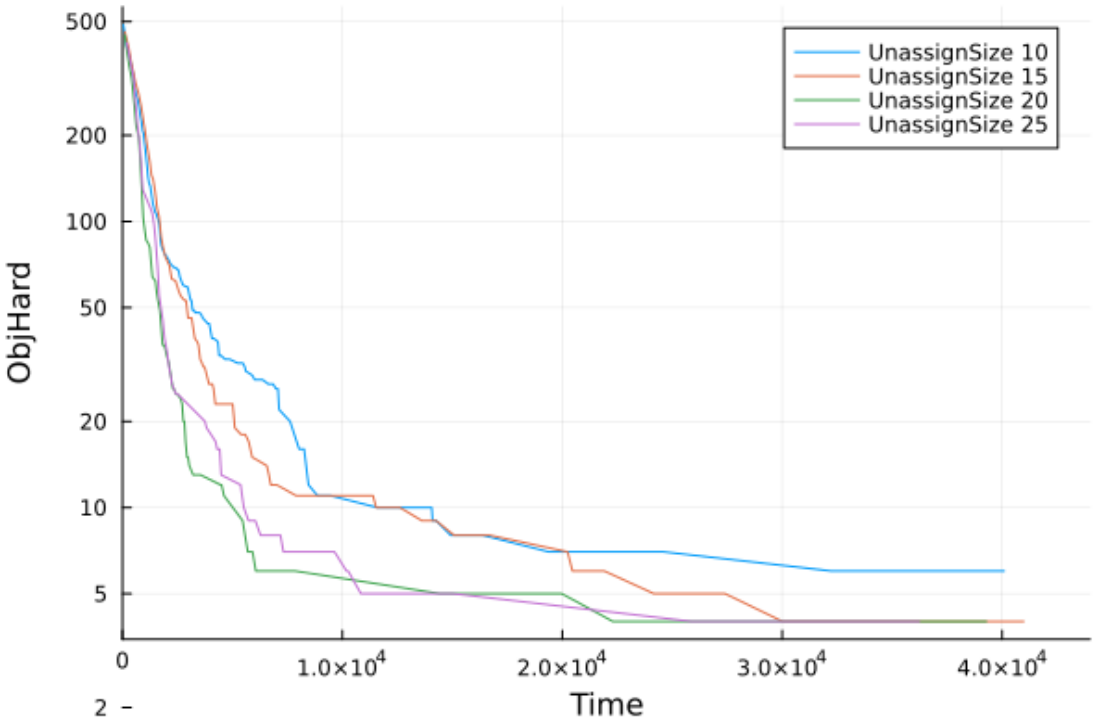
Secondly, on the remaining problem instances, the plots all follow a very similar pattern: The objective values improve very rapidly in the beginning, then at some point the improvement rate decreases to a lower value, and ultimately, the curve flattens to very little improvement being made over multiple iterations. A likely explanation for this behaviour is that initially, the MIP solver only has to replace dummy value assignments to something else (which it is encouraged to do by assignTimeConstraints and assignResourceConstraints penalizing unassigned active events), which probably does not cause collisions with any other assignments at the start. Later, when most or all dummy values on active events are assigned, the computations become harder, not as many constraint violations can be eliminated every time, but there are still enough of them available to see some improvement within every couple of iterations at least. Finally, we reach the point where we are close to local minima and it is improbable to find further improvements.
Thirdly, we observe that higher *unassignSize*s generally seem to perform slightly better. They reach steeper improvements at the beginning and run into the flat section earlier. For the final best solution found, the difference is not as significant. Randomness can be a factor for this to not always be the case. Especially, differences between different runs of the same unassignSize have fluctuations that can be roughly the same margin than the difference to neighboring *unassignSize*s. Generally, randomness appears to have some influence on how fast some improvements proceed and how much time is spent in or near local optima, but overall the behaviour is qualitatively and quantitatively similar across runs. Nonetheless, we conclude that it might be valuable to include some even higher *unassignSize*s in our experiments going forward from here.

Because these initial experiment runs were made in a rather early stage of the project, and conducting the other experiment required to adapt some pieces of code, including an instance of unintended behaviour in one of the problem instance constraints, we later revisit this experiment and perform another calculation on BrazilInstance 5 and 6 to provide another set of graphs to confirm the earlier results. We also use this opportunity to examine *unassignSize*s up to 90. As we can see in Figure 2, the results suggest that while very small *unassignSize*s like 10 and 15 tend to perform worse than 20 and 25, going into even higher values does not keep this improvement and it quickly becomes even worse than values 10 and 15.

When we plot the time each iteration takes (blue line in Figure 3 and Figure 4), we notice that for very small values, most iterations take about the same amount of time,

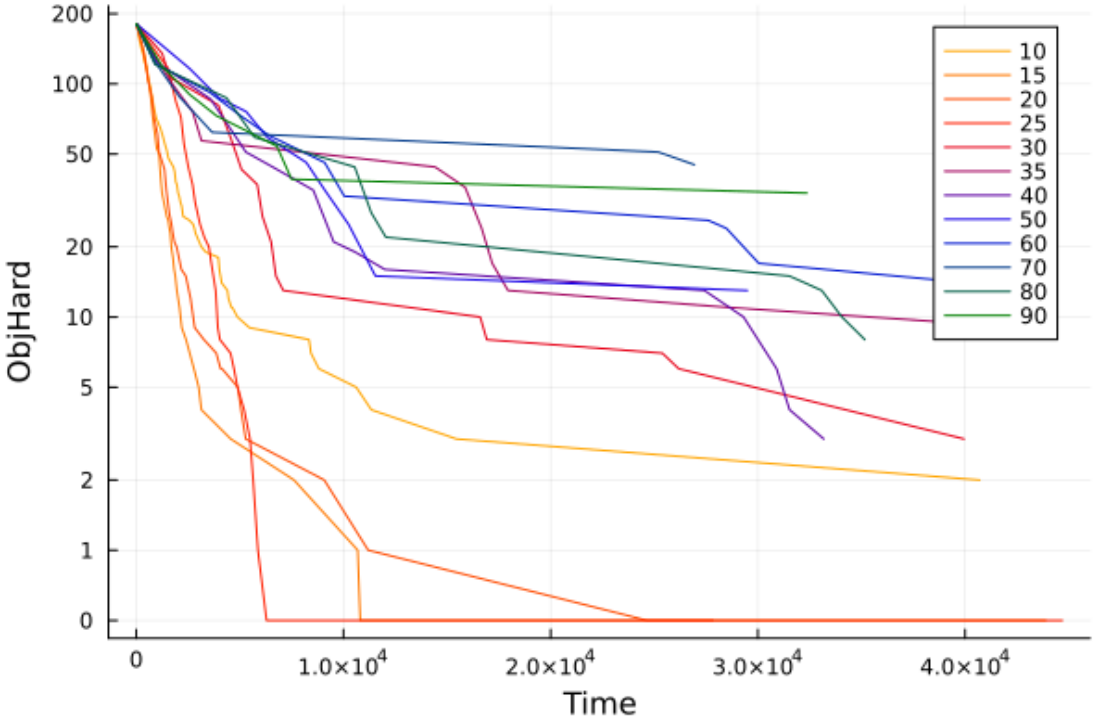


(a) Run 5 of initial runs on BrazilInstance5

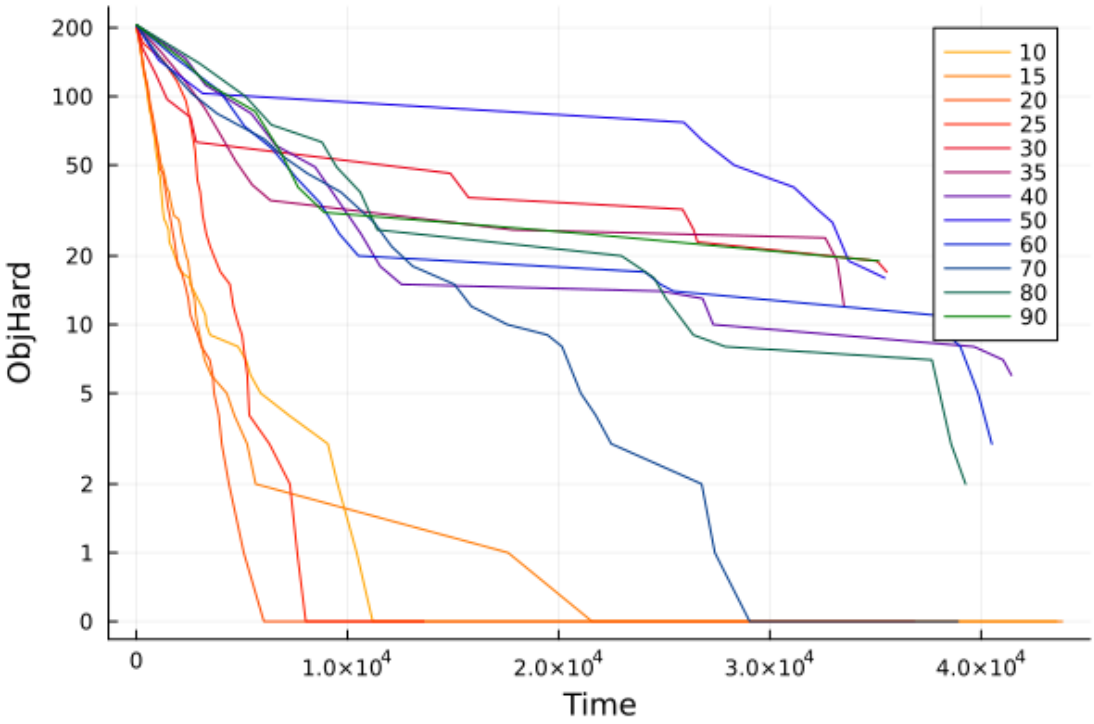


(b) Run 4 of initial runs on BrazilInstance6

Fig. 1: Example Graphs for Experiment 1 from initial runs

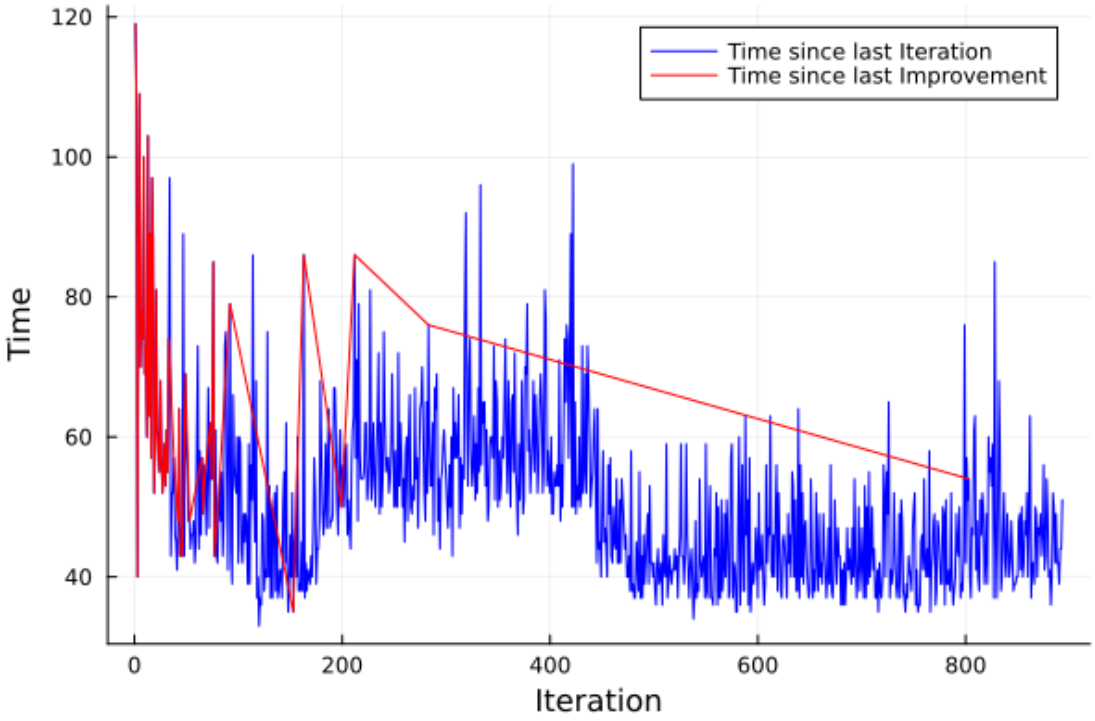


(a) Later rerun on BrazilInstance5

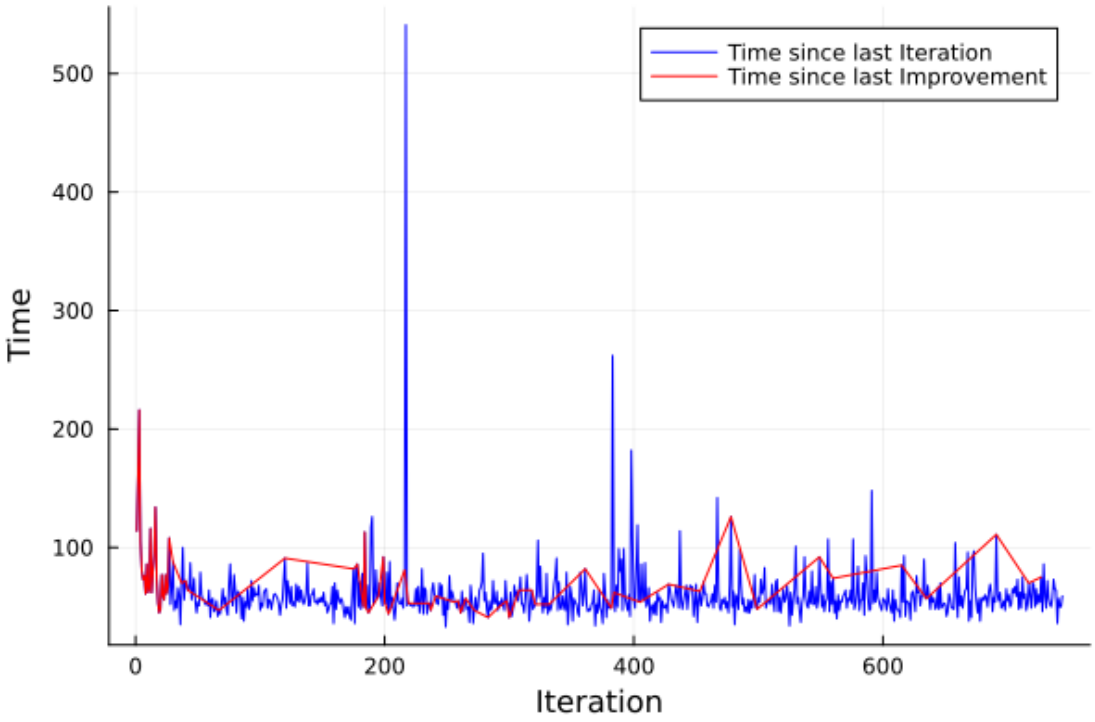


(b) Later rerun on BrazilInstance6

Fig. 2: Experiment 1 reruns (with runcreate.sh)

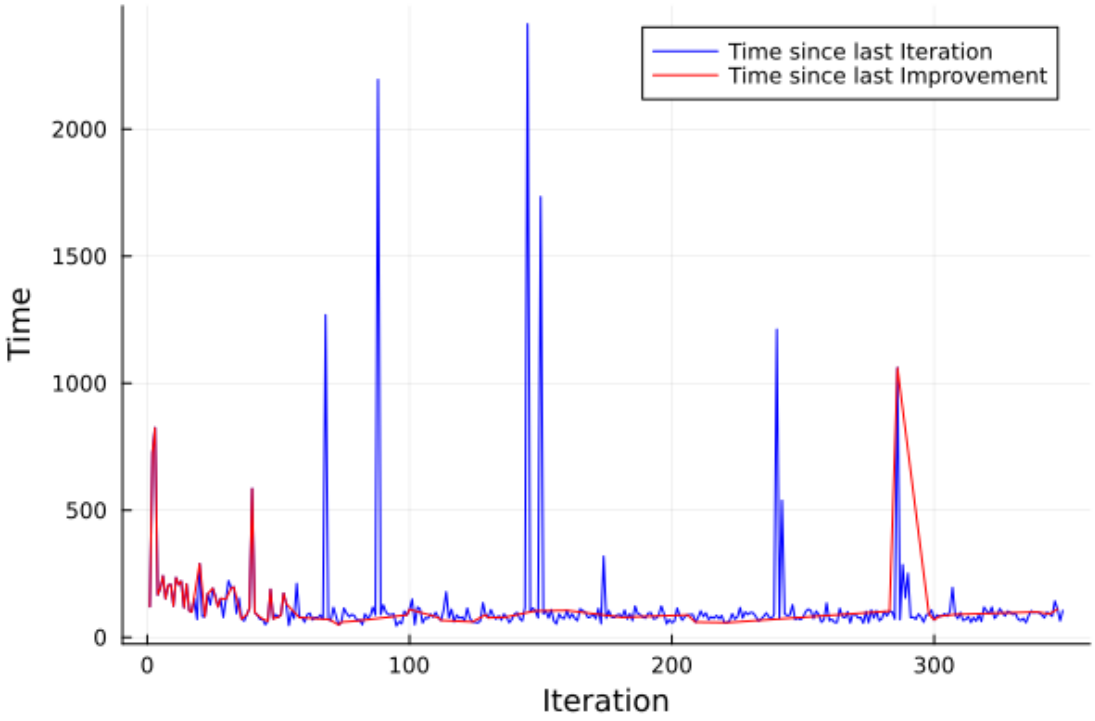


(a) Time Per Iteration: BrazilInstance5, unassignSize 10

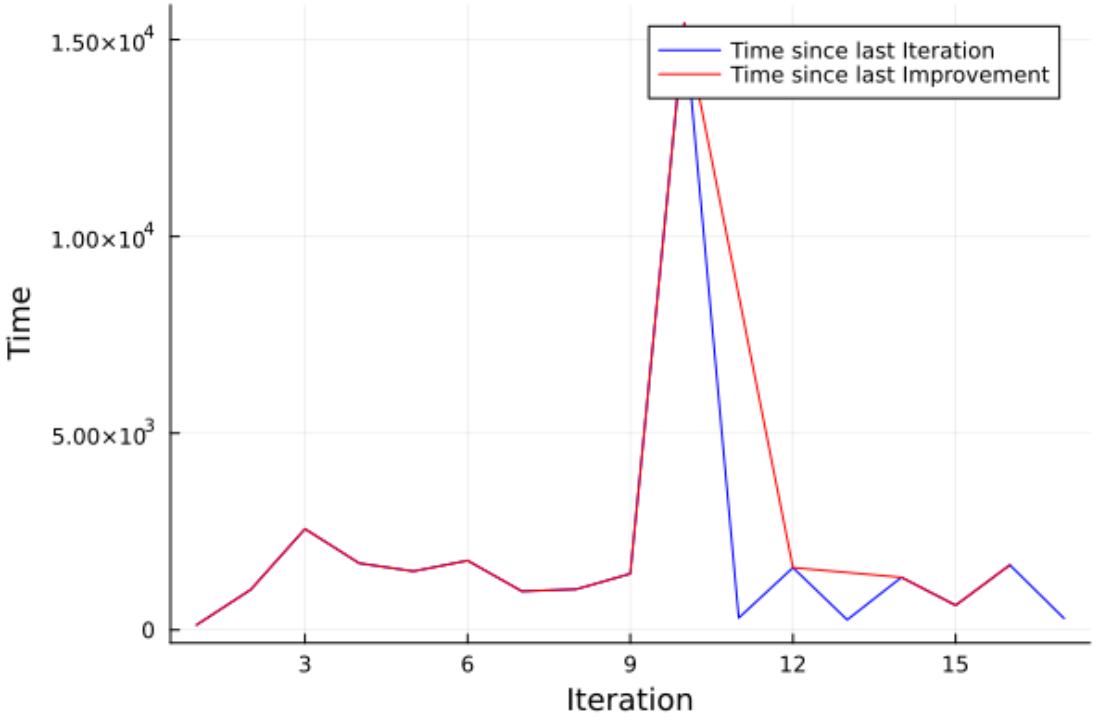


(b) Time Per Iteration: BrazilInstance5, unassignSize 15

Fig. 3: Time Per Iteration on BrazilInstance5 and different unassignSizes



(a) Time Per Iteration: BrazilInstance5, unassignSize 25



(b) Time Per Iteration: BrazilInstance5, unassignSize 40

Fig. 4: Time Per Iteration on BrazilInstance5 and different unassignSizes

Tab. 2: Iteration Durations on BrazilInstance5

UnassignSize	# Iterations	Median	Mean	Mean with IQR	"Outlier malus"
10	893	49.0	50.39	49.17	1.02
15	744	55.0	60.46	56.05	1.08
20	275	68.0	103.64	71.73	1.44
25	349	88.0	128.89	85.58	1.51
30	29	271.0	1410.17	295.46	4.77
35	12	875.5	3588.25	755.11	4.75
40	17	1337.0	1976.59	1136.0	1.74

with some variance, but not at an unexpected margin. As we get to the medium sizes, we notice that, additionally, quite extreme outliers start to appear, taking significantly longer than most iterations. The red line indicates when improvements in objective value have actually been found. We can find no obvious correlation of these points to the long-duration outlier iterations, so we are unable to conclude whether the outlier iterations contain crucial steps for the computation. However, it could be an interesting experiment to find that out, We briefly come back to this idea in Section 7.
When *unassignSize* grows even larger, we reach a point where the amount of iterations rapidly decreases to less than a couple dozen, due to the time per iteration generally increasing. Outliers seem to be still present and consume a significantly bigger amount of time as an average iteration, but with as few iterations, a statistical statement, like we made for medium *unassignSize*s, cannot really be made with high confidence. In Table 2, we display the median and mean durations of iterations for each calculation. We also show the mean duration per iteration after excluding outliers with the IQR method for outlier detection [IQR]. In the column "Outlier Malus", we note the factor by which the time per iteration increases due to outliers, as determined by the IQR methods. The jump in both the Outlier Malus and the time per iteration even with IQR applied cause the total iterations within the time limit to go down drastically. Our conclusion is that this massive drop in total iterations is likely the main reason why the results of the calculation are significantly worse for higher *unassignSize*s. That reason appears to outweigh the increase of the potential for improvement at each iteration gained by using higher *unassignSize*s and therefore optimizing a bigger part of the model being optimized at one step.

6.2.2 Experiment 2: Improving starting solutions

While Experiment 1 was mainly to test if we can find acceptable solutions when running with no starting solution, our main application goal is to have starting solutions and improve those, so we focus on the second task, running with starting solutions, next. We cannot necessarily directly transfer our observations from Experiment 1, because in

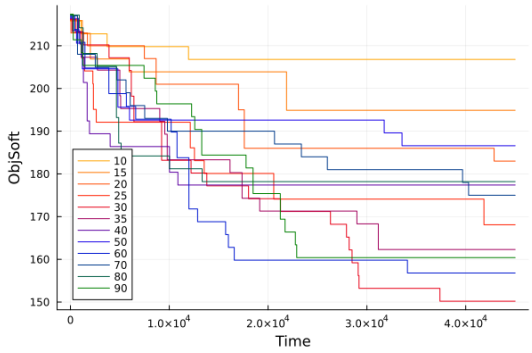
Experiment 1, we are dealing with solutions way farther away from an optimal solution. We initially start with running some calculations for all the BrazilInstances. For each instance, as starting solution, we use the first (and worst) submission on the University of Twente High School Timetabling Project [XHSb], where we also obtained the problem instances from. This comes with the drawback that these solutions are, in most cases, already somewhat good, notably they all have already reached $objHard = 0$, so we immediately swap to focusing only on $objSoft$ as objective value while fixing $objHard = 0$. It is also more likely that these solutions are already in or close to a local optimum which is potentially hard to get out of.
For BrazilInstance 1-4 and BrazilInstance 7, and within two runs, we are only able to shed a couple of violated constraints, which is not very useful to compare the performance of different *unassignSize*s. In BrazilInstance 5 and 6, however, we achieve around 50-60 points of improvement (with most constraints costing 3 each). For that reason, to examine if and by how much higher *unassignSizes* perform better, we do three runs on both BrazilInstance5 and BrazilInstance6 for several different values for *unassignSize*.

In the graphs of Figure 5, we can see for each unassignSize the best/worst of these run, measured by how much time they spend on the best/worst solution among the three runs with same *unassignSize* at the same point in time. What we observe there is that, with these starting conditions, the tendency of large *unassignSize*s falling off significantly compared to small ones is reversed. Small *unassignSize*s, like 10 to 20, perform generally poorer than the larger ones. However, among the large *unassignSize*s, starting with around 25 and above, it is hard to say which ones are best for finding the best solution, as those values result in mixed performance quality across the runs. What seems to be the case, is that the bad cases seem to be somewhat consistently stabilizing in a more narrow corridor when looking at different *unassignSize*s. In other words, a large portion of runs with *unassignSize*s above 25 end up with rather similar results, but a few stand out positively with objective values below the average results by a two-figure margin.
A possible explanation for these phenomena is that when we get closer to the optimal solution, it is more relevant and also harder to deal with local optima. It could be that the small *unassignSize*s get stuck in one of these after a small number of improvements. Because the neighborhood that is created by unassigning a region of a certain size in the MIP is small in this case, it could be more likely or even certain that it is chosen so that an escape from the local optimum cannot happen. With large *unassignSize*s, on the other hand, these neighborhoods are way larger and therefore local optima are less likely to be stuck in for a long time.

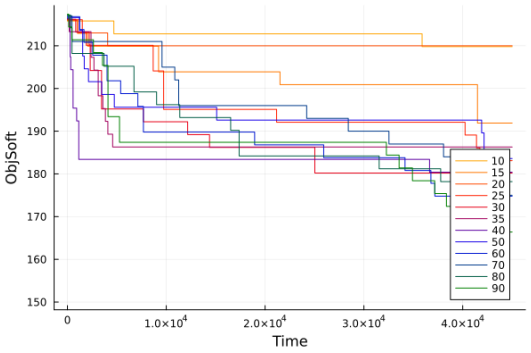
If we take a look at Table 3, the most significant difference we can see compared to the results of experiment 1 is that the "Outlier Malus", i.e. the ratio of the mean with and without outliers is always very close to 1. That means that outliers are less frequent and less extreme, or, in other words, the time spent per iteration behaves relatively stable. A reasonable implication of that is that, due to iteration duration and *unassignSize* correlating inversely (i.e. they balance each other out), for *unassignSize*s large enough

Tab. 3: Iteration Durations on BrazilInstance5, experiment 2

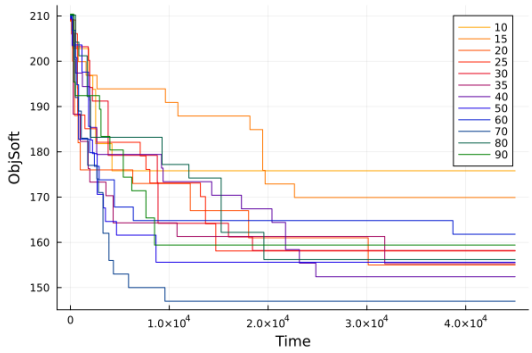
UnassignSize	# Iterations	Median	Mean	Mean with IQR	"Outlier malus"
10	1321	34.0	34.07	33.97	1.00
10	1072	42.0	41.98	41.80	1.00
10	1062	43.0	42.37	42.25	1.00
15	1206	37.0	37.32	37.07	1.01
15	1198	38.0	37.57	37.32	1.01
15	1637	27.0	27.49	27.35	1.01
20	1583	28.0	28.43	27.85	1.02
20	1647	27.0	27.32	27.04	1.01
20	1670	26.0	26.93	26.44	1.02
25	1592	27.0	28.27	27.69	1.02
25	656	67.0	68.58	66.33	1.03
25	646	67.0	69.60	67.75	1.03
30	559	71.0	80.50	74.55	1.08
30	581	69.0	77.08	71.04	1.09
30	637	65.0	70.67	66.39	1.06
35	531	76.0	84.68	78.48	1.08
35	573	72.0	78.44	74.37	1.05
35	524	80.0	85.85	81.58	1.05
40	311	112.0	127.86	116.94	1.09
40	361	136.0	124.63	110.85	1.12
40	394	109.0	114.12	113.55	1.01
50	204	216.5	220.05	214.52	1.03
50	238	172.0	188.82	175.06	1.08
50	234	178.5	192.33	185.34	1.04
60	235	169.0	191.35	173.22	1.10
60	222	184.0	202.34	187.92	1.08
60	228	181.0	197.14	191.77	1.03
70	220	181.0	203.91	188.79	1.08
70	230	186.5	195.24	192.42	1.01
70	267	161.0	167.97	163.18	1.03
80	192	194.5	233.53	219.18	1.07
80	241	169.0	185.85	173.71	1.07
80	283	147.0	159.02	148.93	1.07
90	265	156.0	169.43	161.93	1.05
90	290	147.5	154.86	148.99	1.04
90	243	161.0	185.22	170.22	1.09



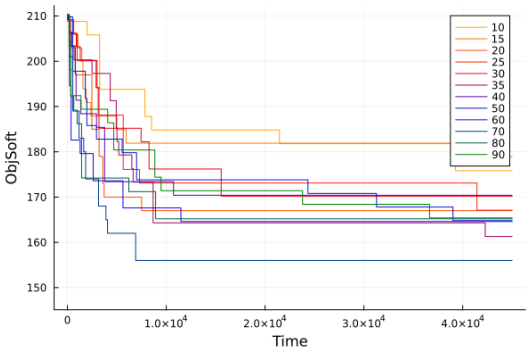
(a) Best out of 3 runs on BrazilInstance5 for different *unassignSize*s



(b) Worst out of 3 runs on BrazilInstance5 for different *unassignSize*s



(c) Best out of 3 runs on BrazilInstance6 for different *unassignSize*s



(d) Worst out of 3 runs on BrazilInstance6 for different *unassignSize*s

Fig. 5: Best and Worst runs out of three on BrazilInstances 5 and 6 when running with starting solution. The same graphs can be found in the appendix **??** to **??** in full scale, we leave them here in small size to allow for easy side by side comparison between best and worst run graphs.

to theoretically avoid most local optima, it comes down to mostly random chance of how long it takes until a new best solution can be found. Most runs are fine, some runs are lucky and perform better, and those with the smallest *unassignSize*s are too likely to get stuck in local optima to be competitive with larger ones.
On a side note, we want to mention that the significant drop in iteration count between the first and the second and third runs at *unassignSize* 25 is unexpected even inside an assumption of randomness; We believe the most likely explanation is performance discrepancies of different computing units and scheduling circumstances on the computing cluster we used. A quick close comparison of the runs shows that, while the run with more iterations did end up on a slightly better objective value, it is not a significant improvement. Also, most likely, the smaller *unassignSize*s were computed on the same cluster unit, and those did not generally perform better than larger *unassignSize*s despite possibly having an advantage in CPU performance.

7 Conclusion

Even though our experiments were conducted on a rather small scale, which is primarily due to the limited availability of realistic data, through the observations from the experiment, we can come to some careful conclusions. First of all, we showed that the method that we developed is capable of doing what we originally designed it for: Improving the solution for a given timetabling problem within reasonable amounts of time. This is especially true when comparing it to monolithic MIP approaches, which generally take not only unrealistically long to solve once problems are no longer very small, but also it is very intransparent, how long it will take to find its solution. While we can hardly circumvent this inherent disadvantage of MIP approaches with our MIP-based method, our step-wise improvement makes sure that improvements happen regularly. Furthermore, our method, specifically the algorithm that selects the set of variables to reconsider, comes with a certain flexibility when defining the neighborhood structures to search in. We also showed that our algorithm is capable of improving solutions for both good and bad starting solutions.
A very interesting observation to pick up again in that regard is that different solution qualities make different parameter choices, namely $unassigSize$ more performant. It would require larger-scale examination for high confidence about it, but at least in our circumstances, it appears as if starting with a small $unassignSize$ of 20 or 25 for bad starting solutions and later swapping to larger values like 50 or more was a good practice. We imagine that this parameter is best to be kept so that the probability of a better solution being in the generated neighborhood and the time of an iteration balances each other out. There are different strategies that could be tried out in that regard, like e.g. increasing $unassignSize$ once too many iterations are unable to improve the current solution.
Another potential improvement that we definitely want to mention is designing a strategy which attempts to reduce the impact of the very extreme outliers in iteration duration shown in Table 2. For example, one could define a detection system that cancels an iteration once it takes too long beyond a certain value that is estimated based on the durations of previous iterations. Another approach could be to adapt a beam search strategy where other beams are informed and synchronized once one beam hits a new best solution. Especially with the highly diverse neighborhood structure generated by the selection algorithm with random elements, such a strategy employing multi-threaded beam search could lead to promising performance increases.
Lastly, it would be an interesting experiment to port the method to other optimization problems. The algorithm used for our experiments is specifically tuned to fit HSTT problem instances, so we highly recommend adapting the method for other types of problems as the target domain requires, but since the method combine an MIP-based approach with local search in an attempt to untilize the advantages of both as good as possible, this can be promising for other domains. In the end, as we mentioned in the report, the approach has lots of similarity with the metaheuristic of Large Neighborhood Searches, which has been proven to be a capable problem solving method.

8 Appendix

8.1 All graphs of run 5 of the initial runs

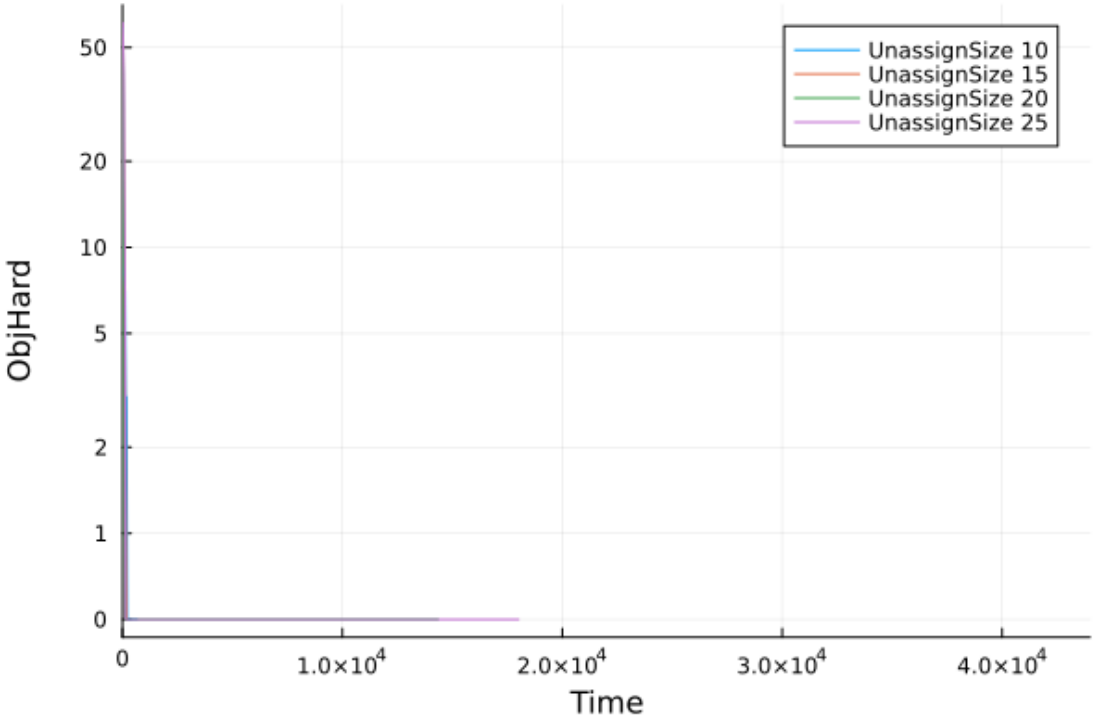


Fig. 6: Run 5 for initial runs on BrazilInstance1 for different *unassignSizes*

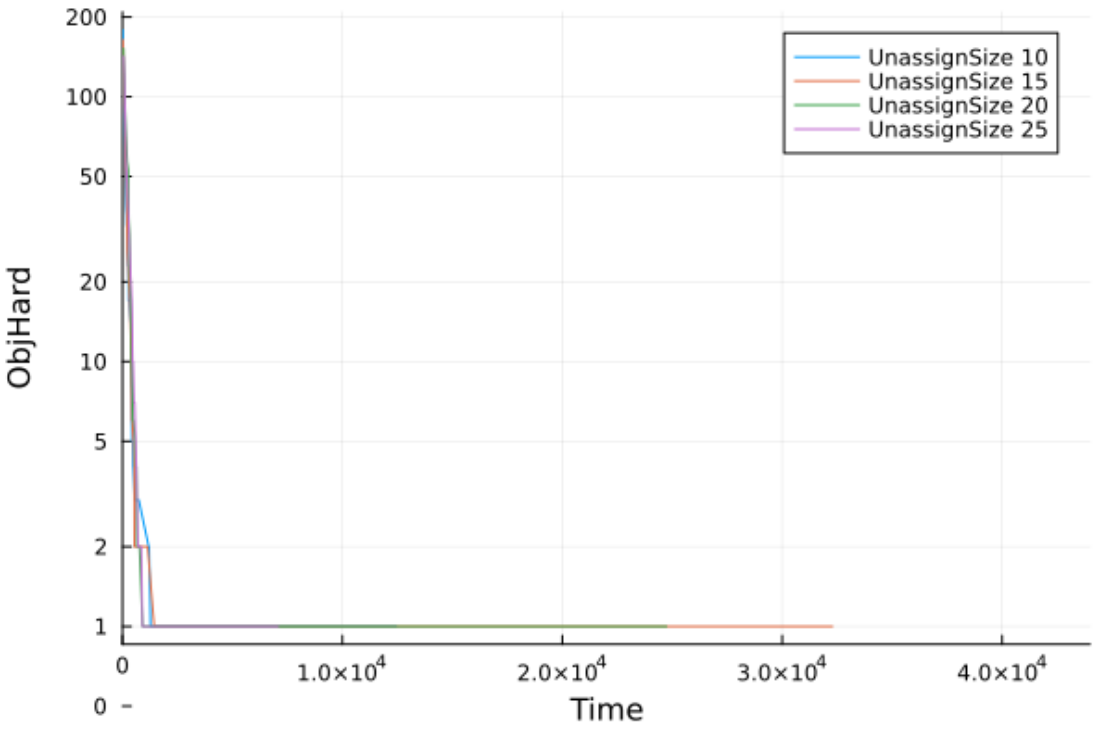


Fig. 7: Run 5 for initial runs on BrazilInstance2 for different *unassignSizes*

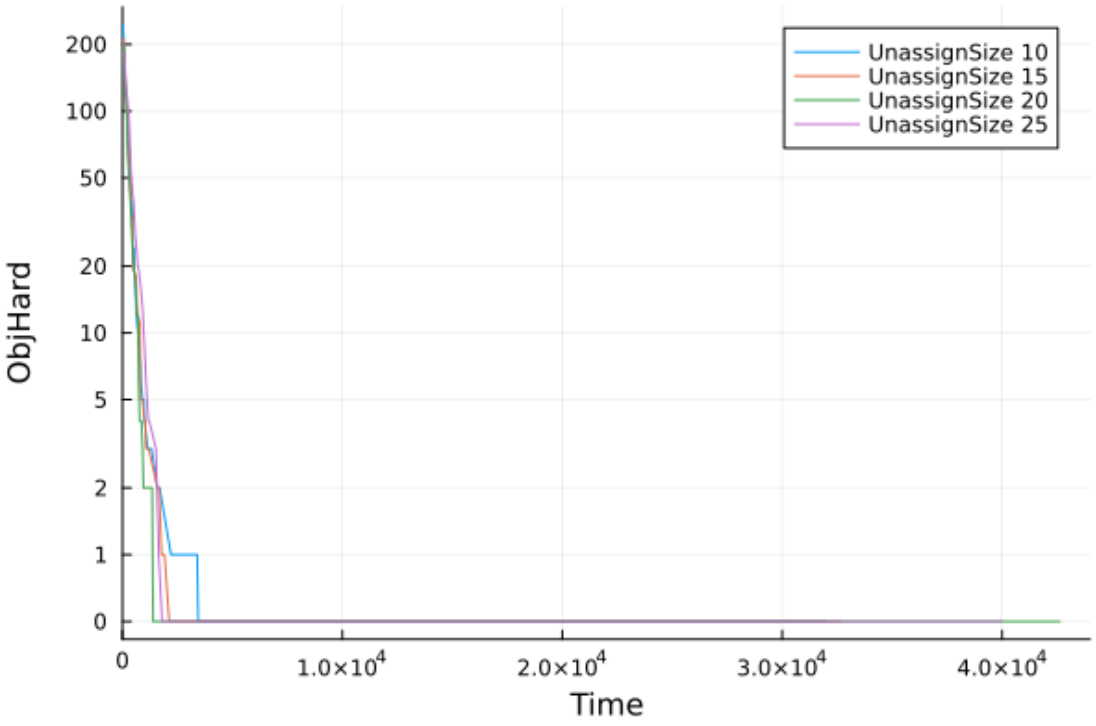


Fig. 8: Run 5 for initial runs on BrazilInstance3 for different *unassignSizes*

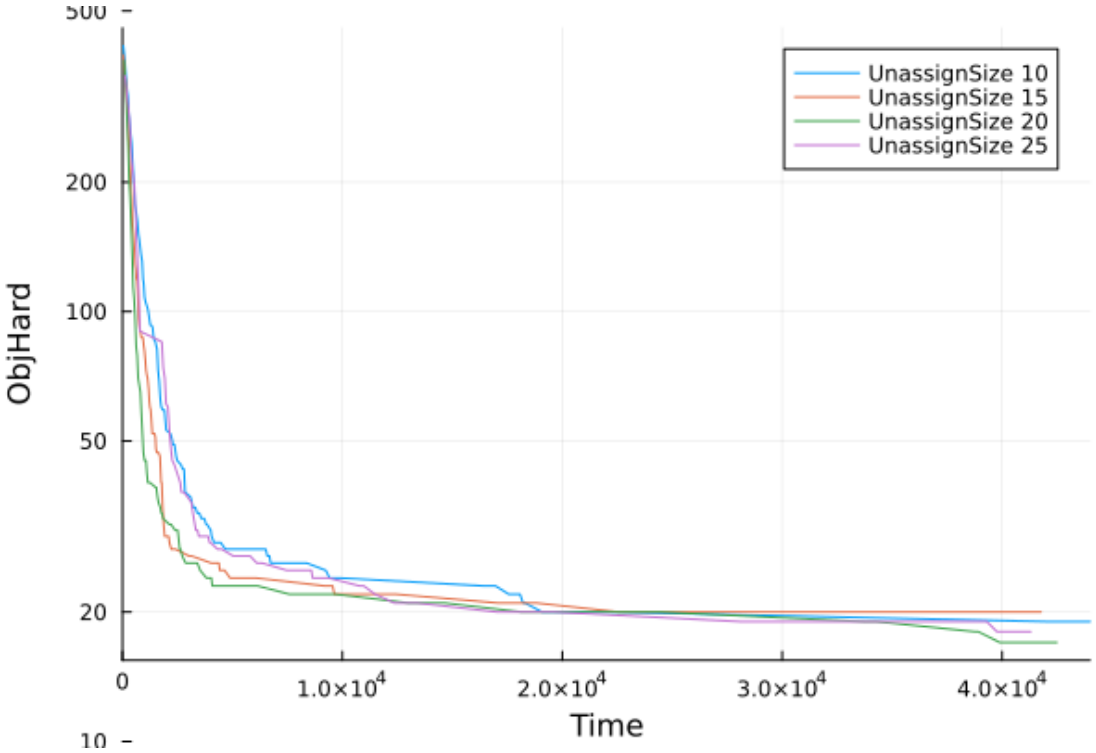


Fig. 9: Run 5 for initial runs on BrazilInstance4 for different *unassignSizes*

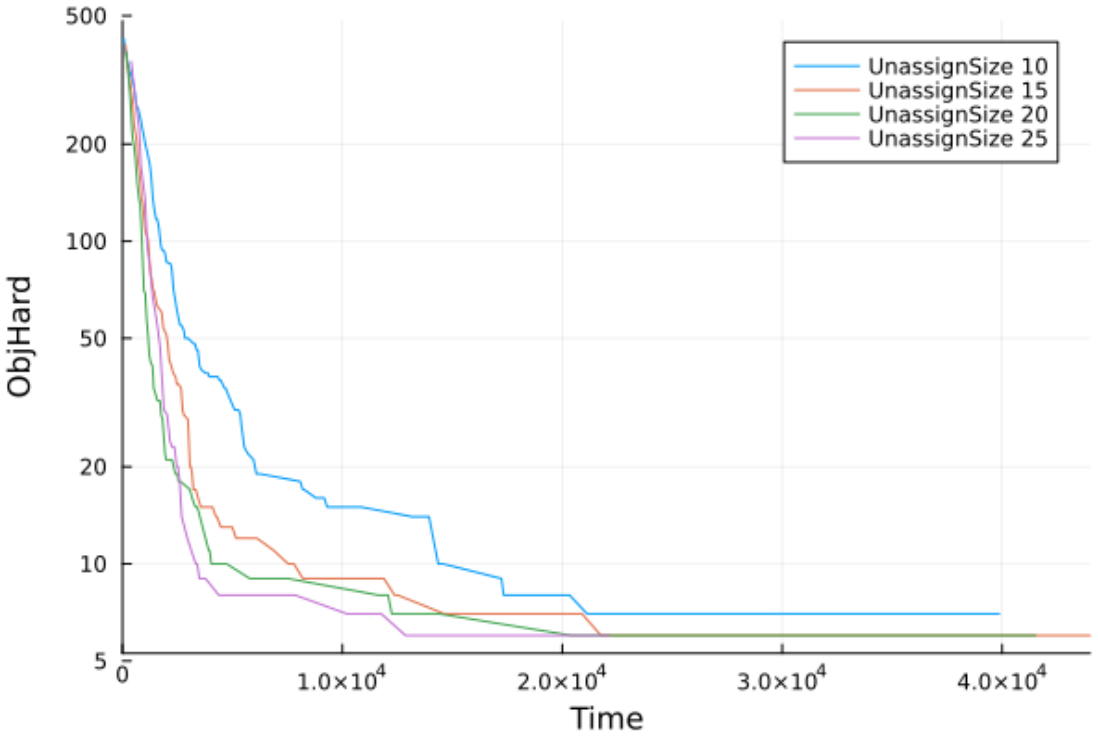


Fig. 10: Run 5 for initial runs on BrazilInstance5 for different *unassignSize*s

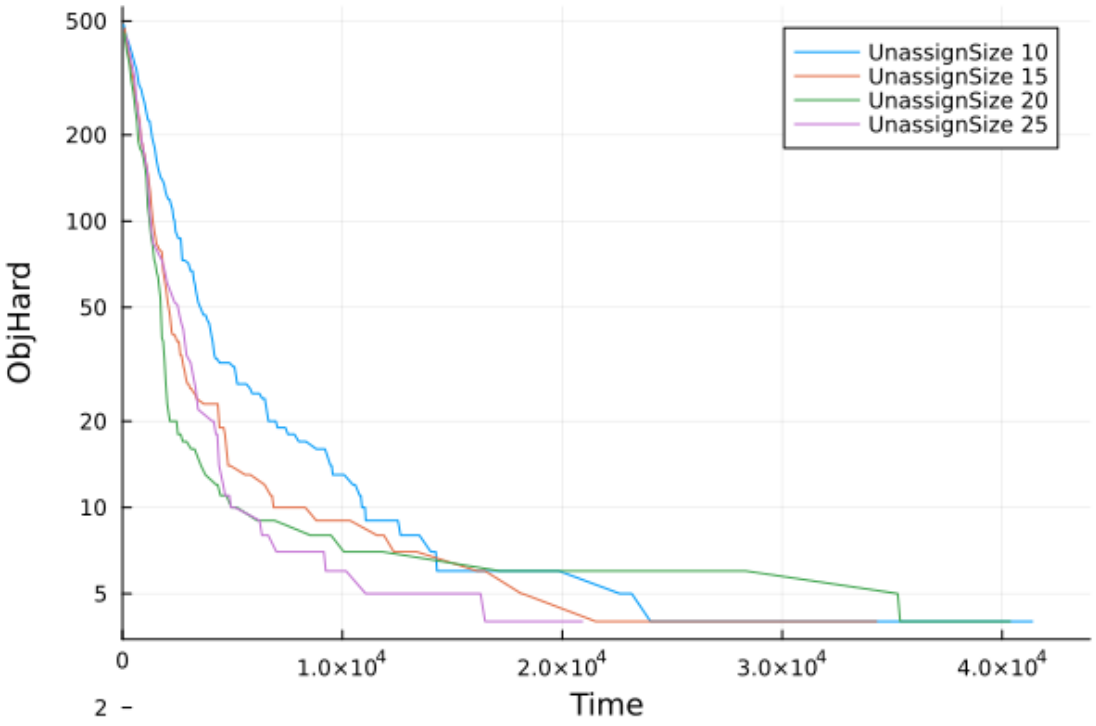


Fig. 11: Run 5 for initial runs on BrazilInstance6 for different *unassignSizes*

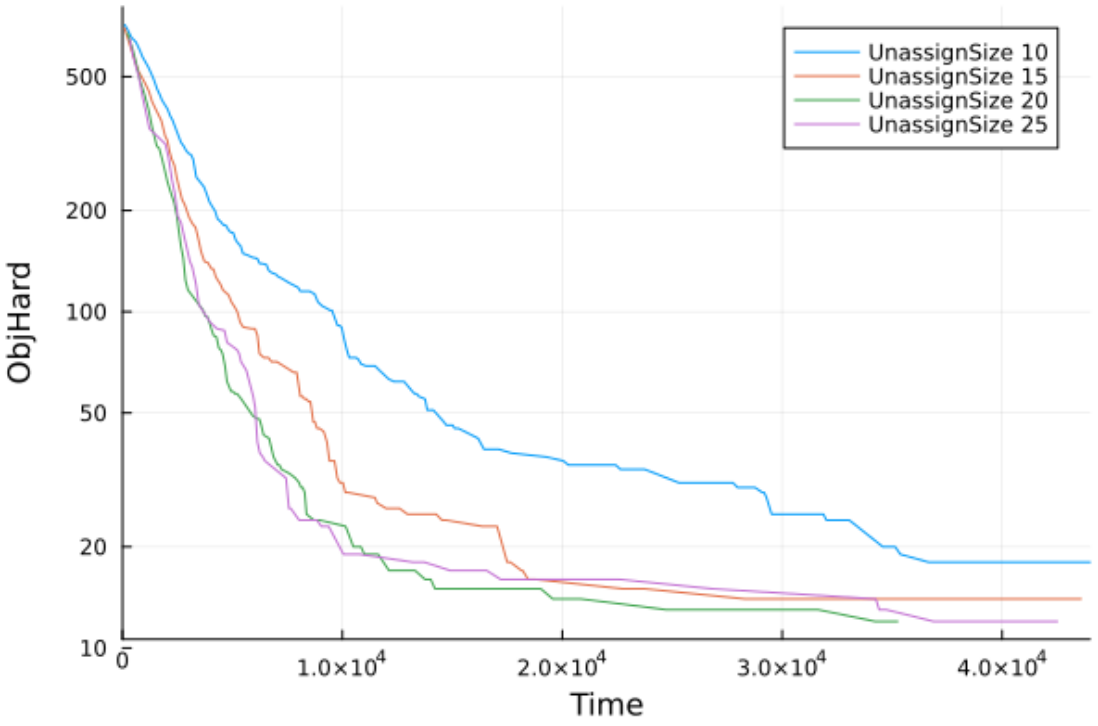


Fig. 12: Run 5 for initial runs on BrazilInstance7 for different *unassignSize*s

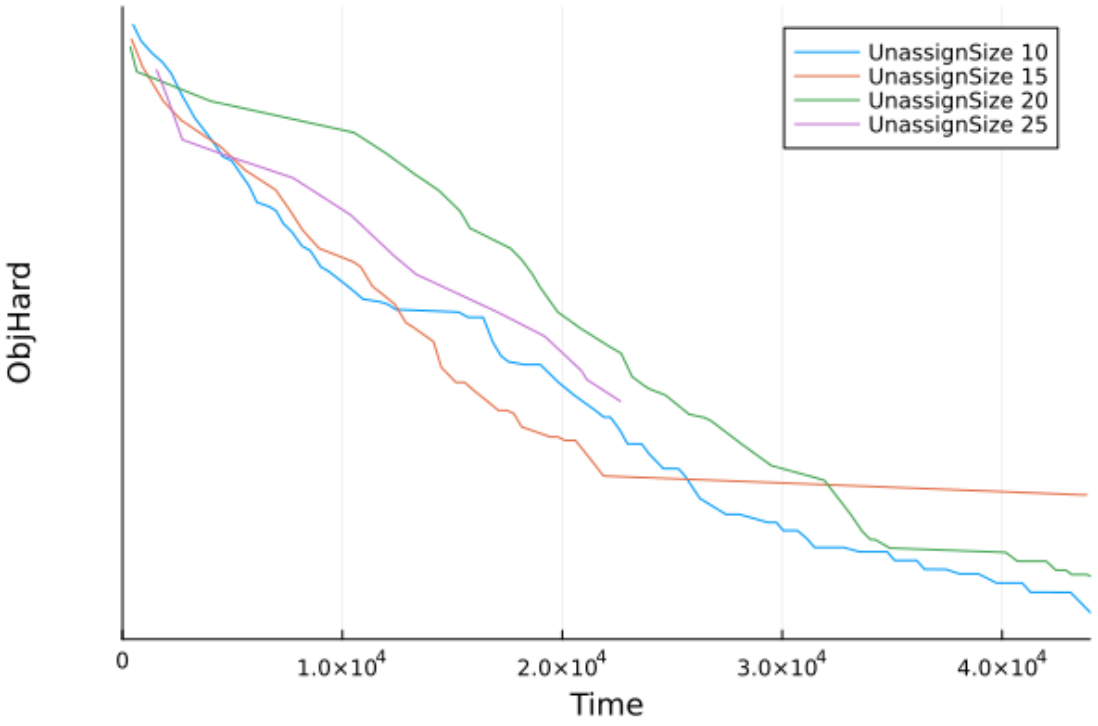


Fig. 13: Run 5 for initial runs on AustraliaTES99 for different *unassignSizes*

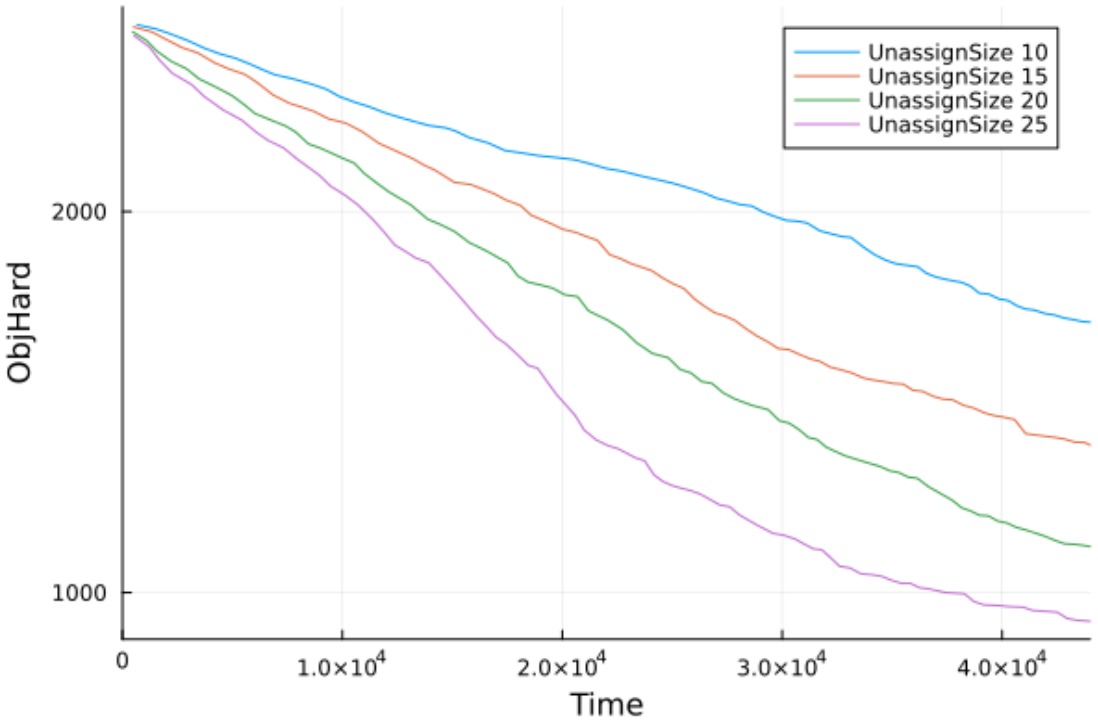


Fig. 14: Run 5 for initial runs on ItalyInstance4 for different *unassignSizes*

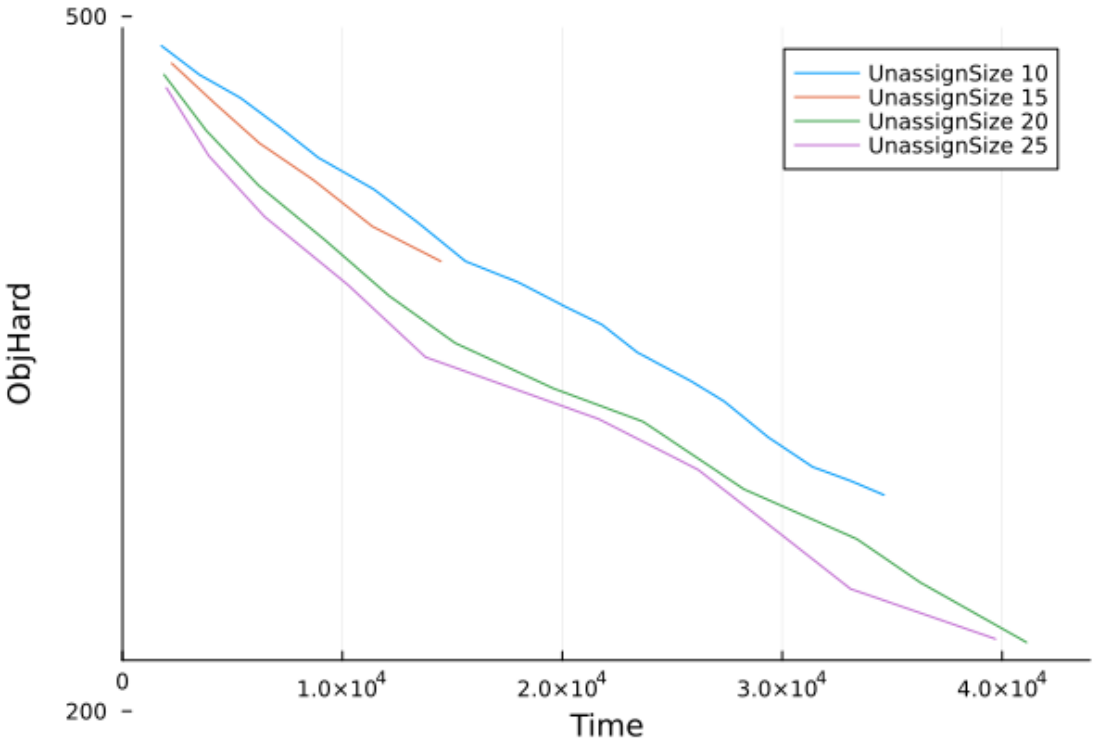


Fig. 15: Run 5 for initial runs on SouthAfricaWoodlands2009 for different *unassignSize*s

8.2 Experiment 2 best and worst runs (full scale)

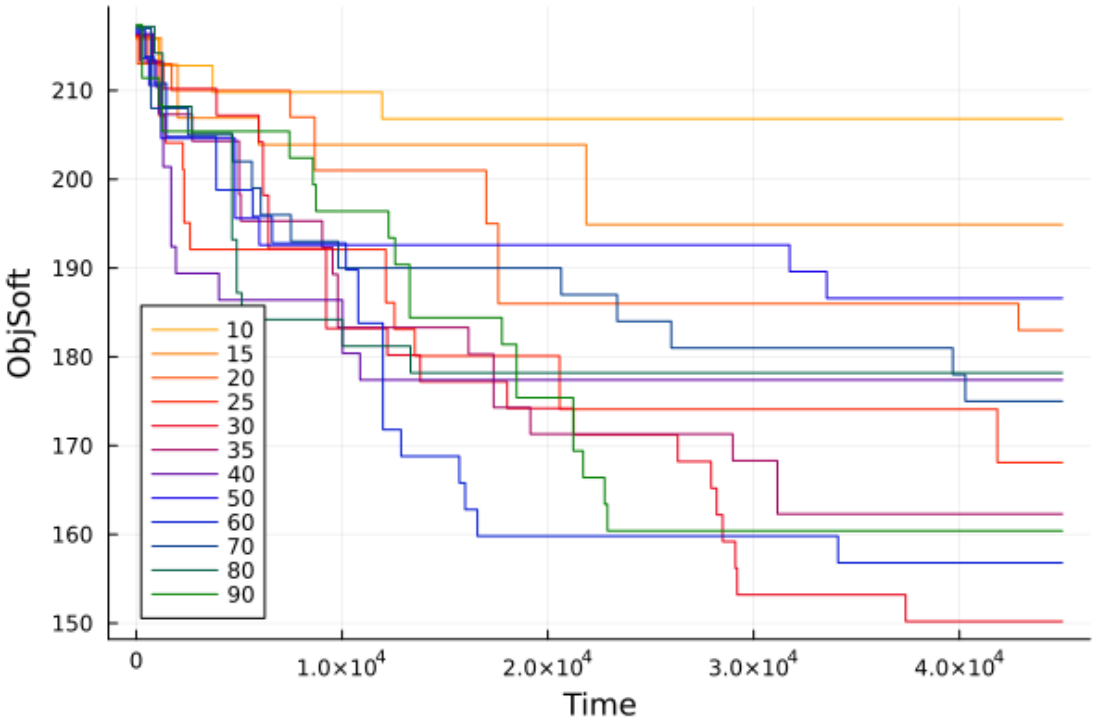


Fig. 16: Best out of 3 runs on BrazilInstance5 for different *unassignSizes*

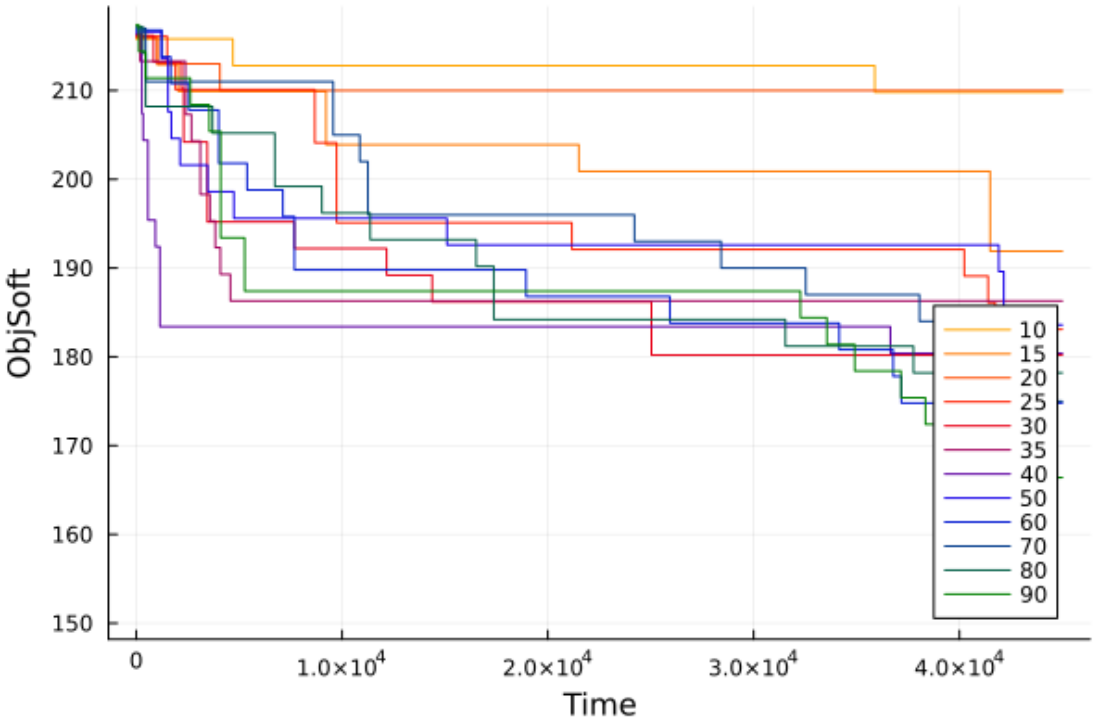


Fig. 17: Worst out of 3 runs on BrazilInstance5 for different *unassignSizes*

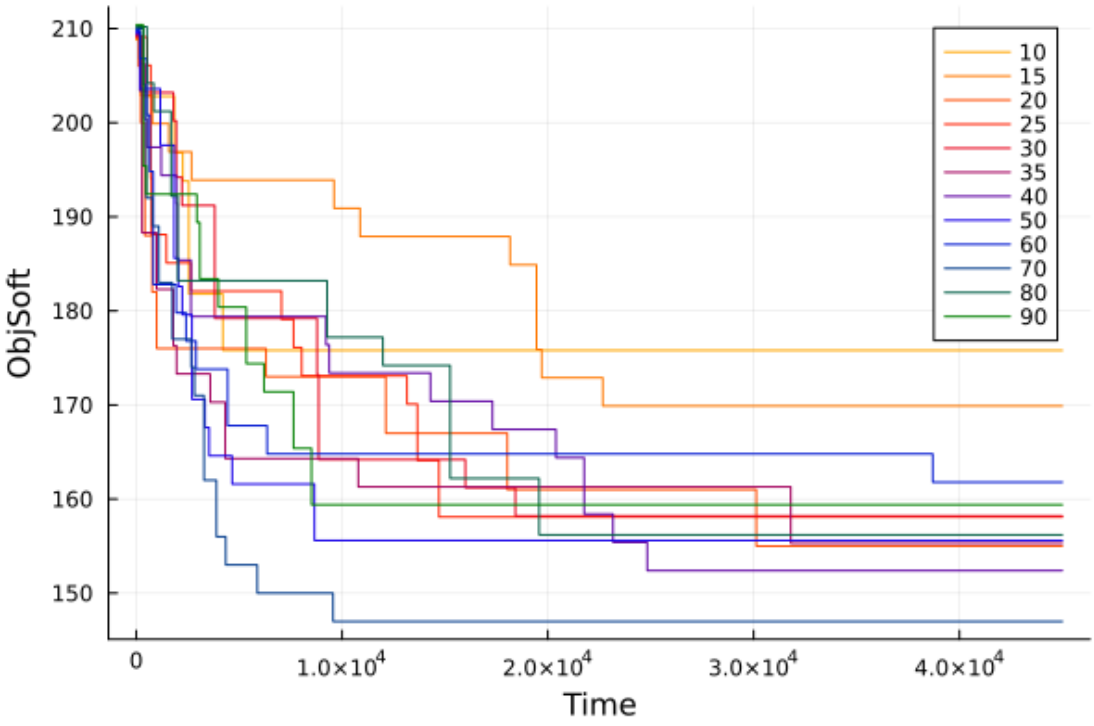


Fig. 18: Best out of 3 runs on BrazilInstance6 for different *unassignSize*s

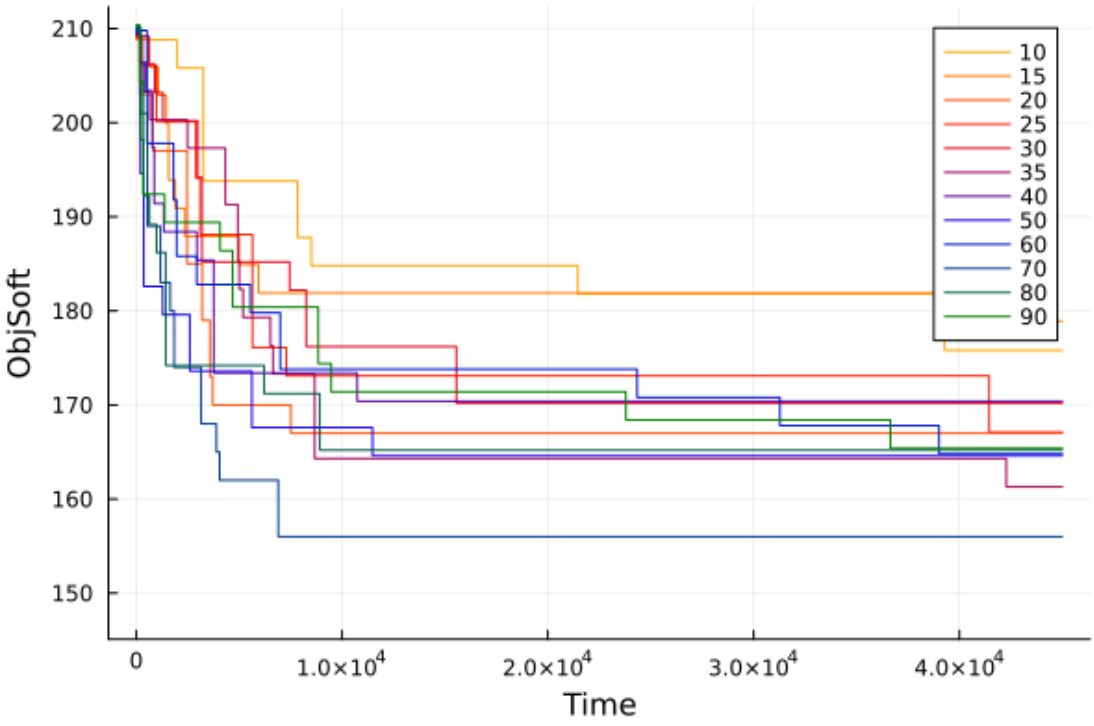


Fig. 19: Worst out of 3 runs on BrazilInstance6 for different *unassignSizes*

References

[CK96] Tim B. Cooper and Jeffrey H. Kingston: The complexity of timetable construction problems. In Edmund Burke and Peter Ross (editors): *Practice and Theory of Automated Timetabling*, pages 281–295, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg, ISBN 978-3-540-70682-3.

[Cod] Code repository. `https://gitlab2.informatik.uni-wuerzburg.de/s410714/masterpraktikum-stundenplan`.

[DdB14] Arton P. Dorneles, Olinto C.B. de Araujo, and Luciana S. Buriol: A fix-and-optimize heuristic for the high school timetabling problem. *Computers & Operations Research*, 52:29–38, 2014, https://doi.org/10.1016/j.cor.2014.06.023, ISSN 0305-0548. `https://www.sciencedirect.com/science/article/pii/S0305054814001816`.

[FS14] George HG Fonseca and Haroldo G Santos: Variable neighborhood search based algorithms for high school timetabling. *Computers & Operations Research*, 52:203–208, 2014.

[FSC16] George H.G. Fonseca, Haroldo G. Santos, and Eduardo G. Carrano: Integrating matheuristics and metaheuristics for timetabling. *Computers & Operations Research*, 74:108–117, 2016, https://doi.org/10.1016/j.cor.2016.04.016, ISSN 0305-0548. `https://www.sciencedirect.com/science/article/pii/S0305054816300879`.

[IQR] Iqrmethoddescription. `https://www.geeksforgeeks.org/machine-learning/interquartile-range-to-detect-outliers-in-data/`.

[KSS15] Simon Kristiansen, Matias Sørensen, and Thomas R Stidsen: Integer programming for the generalized high school timetabling problem. *Journal of Scheduling*, 18(4):377–392, 2015.

[OOOO20] O. A. Odeniyi, E. O. Omidiora, S. O. Olabiyisi, and C. A. Oyeleye: A mathematical programming model and enhanced simulated annealing algorithm for the school timetabling problem. *Asian Journal of Research in Computer Science*, 5(3):21–38, May 2020, 10.9734/ajrcos/2020/v5i330136. `https://journalajrcos.com/index.php/AJRCOS/article/view/94`.

[PDGK+16] Gerhard Post, Luca Di Gaspero, Jeffrey H Kingston, Barry McCollum, and Andrea Schaerf: The third international timetabling competition. *Annals of Operations Research*, 239(1):69–75, 2016.

[SCR13] Landir Saviniec, Ademir Aparecido Constantino, and Wesley Romao: Vns based algorithms to the high school timetabling problem. *Anais do XLV simpósio brasileiro de pesquisa operacional*, pages 845–856, 2013.

[SKS+12] Matias Sørensen, Simon Kristiansen, Thomas R Stidsen, *et al.*: International timetabling competition 2011: An adaptive large neighborhood search algorithm. In *Proceedings of the ninth international conference on the practice and theory of automated timetabling (PATAT 2012)*, page 489, 2012.

[TGKS21] Joo Siang Tan, Say Leng Goh, Graham Kendall, and Nasser R. Sabar: A survey of the state-of-the-art of optimisation methodologies in school timetabling problems. *Expert Systems with Applications*, 165:113943, 2021, https://doi.org/10.1016/j.eswa.2020.113943, ISSN 0957-4174. `https://www.sciencedirect.com/science/article/pii/S0957417420307314`.

[TSdSC19] Ulisses Rezende Teixeira, Marcone Jamilson Freitas Souza, Sérgio Ricardo de Souza, and Vitor Nazário Coelho: An adaptive vns and skewed gvns approaches for school timetabling problems. In Angelo Sifaleras, Said Salhi, and Jack Brimberg (editors): *Variable Neighborhood Search*, pages 101–113, Cham, 2019. Springer International Publishing, ISBN 978-3-030-15843-9.

[XHSa] Website with hseval and xhstt specification. `http://jeffreykingston.id.au/cgi-bin/hseval.cgi?op=spec`.

[XHSb] Website with xhstt datasets. `https://www.utwente.nl/en/eemcs/dmmp/hstt/`.

[ZLML10] Defu Zhang, Yongkai Liu, Rym M'Hallah, and Stephen C.H. 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, https://doi.org/10.1016/j.ejor.2009.09.014, ISSN 0377-2217. `https://www.sciencedirect.com/science/article/pii/S0377221709006055`.