Journal of Scheduling (2022) 25:429–452
https://doi.org/10.1007/s10951-022-00728-8

A parallelized matheuristic for the International Timetabling Competition 2019

Rasmus Ø. Mikkelsen[1] · Dennis S. Holm[1]

Accepted: 15 February 2022 / Published online: 3 May 2022

Abstract
The International Timetabling Competition 2019 (ITC 2019) presents a novel and generalized university timetabling problem composed of traditional class time and room assignment and student sectioning. In this paper, we present a parallelized matheuristic tailored to the ITC 2019 problem. The matheuristic is composed of multiple methods using the graph-based mixed-integer programming (MIP) model defined for the ITC 2019 problem by Holm et al. (A graph-based MIP formulation of the International Timetabling Competition 2019. J Sched, 2022. https://doi.org/10.1007/s10951-022-00724-y). We detail all methods included in the parallelized matheuristic and the collaboration between them. The parallelized matheuristic includes two methods for producing initial solutions and uses a fix-and-optimize matheuristic to improve solutions. Additionally, the method uses the full MIP model to calculate lower bounds. We describe how the methods perform collaboratively through solution sharing, and a diversification scheme invoked when the search stagnates. Furthermore, we explain how we decompose the problem for instances with a large number of students. We evaluate components of the parallelized matheuristic using the 30 benchmark instances of the ITC 2019. The complete parallelized matheuristic performs well, even solving some instances to proven optimality. The presented method is the winning algorithm of the competition, further demonstrating its quality.

Keywords Mixed-integer programming · Parallelized matheuristic · Fix-and-optimize · University timetabling · International Timetabling Competition 2019 · ITC 2019

1 Introduction

University timetabling is a complex scheduling problem that all universities must regularly solve in practice. The classical university course timetabling problem consists of developing a semester timetable such that all course events (lectures, exercises, etc.) are assigned a room and time. The goal is to make a feasible high-quality timetable. A timetable is feasible if it satisfies all hard constraints, and its quality is measured by the violations of soft constraints. Timetables of higher quality have fewer undesirable features and lead to better utilization of resources and more preferable schedules for teachers and students.

The International Timetabling Competition 2019 (ITC 2019) presents a university course timetabling problem general enough that it can encompass many practical university timetabling problems, as evident by the competition using real-world data from ten different universities in eight countries on five continents. The problem definition is also novel as it combines student sectioning and classical course time and room assignment (Müller et al. 2018a). Thus, solution approaches that work well on this problem definition should be usable in many practical cases.

This paper describes a parallelized matheuristic for the university timetabling problem presented at the ITC 2019. Our solution approach combines multiple methods based on mixed-integer programming (MIP). Most notably, we use

The authors have contributed equally to this work. The authors would like to thank Bernd Dammann and Sebastian Borchert for providing valuable support for working with high-performance computing. Dennis S. Holm's PhD project is part of the Data Science for University Management project (dsumsoftware.com) funded by MaCom A/S and Innovation Fund Denmark (IFD). Rasmus Ø. Mikkelsen's industrial PhD project is funded by IFD. IFD has supported the work solely financially and has not participated in any research-related activities.

✉ Rasmus Ø. Mikkelsen
rasmi@dtu.dk ; rasmusomikkelsen@gmail.com

Dennis S. Holm
dsho@dtu.dk

1 Technical University of Denmark, DTU Management, Akademivej, Building 358, 2800 Kgs. Lyngby, Denmark

simultaneous searches using a fix-and-optimize matheuristic combined with solving the full MIP model using a black-box solver. We use the matheuristics to find high-quality solutions and primarily use the full MIP model to provide lower bound information. The parallelized matheuristic shares solutions between search methods to accelerate the combined search and implements a diversification scheme to escape local optima. Additionally, the parallelized matheuristic includes a special setup for data instances with a large number of students.

The proposed solution approach presents a general framework that can be applied to any MIP model. One simply needs to define neighborhoods for the fix-and-optimize matheuristics and methods for constructing initial solutions. However, since many MIP models need to be solved, the framework's performance is directly tied to the strength of the MIP formulation. Here, we use the MIP model presented by Holm et al. (2020), which uses data reductions and different graph structures to provide a strong MIP formulation.

The paper is organized as follows. In Sect. 2, we discuss related work. Section 3 presents the ITC 2019 problem. In Sect. 4, we give a short introduction to the MIP model. Section 5 thoroughly details the fix-and-optimize matheuristic. In Sect. 6, we describe the complete parallelized matheuristic. Section 7 evaluates the method through computational results, and Sect. 8 concludes.

2 Related work

The university course timetabling problem is a long-studied, practical timetabling problem. The course timetabling requirements can vary significantly for different universities, resulting in many problem variants (Tripathy 1992; Schaerf 1999). The International Timetabling Competitions 2007 (ITC2007) introduced two university course timetabling benchmark problems: post-enrollment-based (track 2) and curriculum-based (track 3) (McCollum et al. 2010; Lewis et al. 2007; Di Gaspero et al. 2007). Common benchmark problems are essential for driving the research field forward, and the curriculum-based problem of the ITC2007 in particular has received much attention. Researchers have applied many different solution techniques to the problem, but metaheuristic approaches have been especially popular (Kristiansen & Stidsen 2013; Bettinelli et al. 2015).

The hybridization of metaheuristics and exact methods results in matheuristics. Matheuristics presents a powerful solution approach in which the goal typically is to gain the speed of heuristics while retaining the desirable properties of exact methods, such as bounding information. In recent years, mathematical programming-based approaches have become increasingly popular and have attained state-of-the-art results for high school timetabling (Fonseca et al. 2016; Tan et al. 2021). Mathematical programming-based approaches for course timetabling have also achieved good results, see, e.g., Burke et al. (2010); Lach and Lübbecke (2012); Burke et al. (2012). However, much research using exact methods has focused on improving lower bounds (Bettinelli et al. 2015).

Lindahl et al. (2018) introduced the fix-and-optimize matheuristic applied to the curriculum-based course timetabling problem of the ITC2007, producing better results than the competition-winning algorithm. The authors also describe an approach for adapting the neighborhood size using the relative gap of the incumbent solution and the lower bound of each subproblem. The fix-and-optimize matheuristic has also achieved state-of-the-art results on other scheduling problems, such as high school timetabling (Dorneles et al. 2014) and capacitated lot-sizing problems (Lang and Shen 2011; Helber and Sahling 2010).

3 Problem definition

The ITC 2019 problem defines a novel university timetabling problem combining classical course time and room assignment and student sectioning. Courses consist of one or more classes that must all be assigned one of their predefined times and available rooms (if requested), each defined with a non-negative integer penalty. The course classes are separated into configurations, which are further divided into subparts, and this hierarchical structure is important for student sectioning. Students request courses which they must attend. A valid student assignment observes that each student attends precisely one class of each subpart of a single configuration for each requested course and that any class does not exceed its attendance limit. Configurations and subparts are designed so that students enrolled in a course are guaranteed to attend a valid combination of the course's classes.

Distribution constraints place restrictions on the assignment between two or more classes and can be either hard or soft. There are 19 different types of distribution constraints, e.g., forbid/penalize temporal overlap between classes, enforce/prefer classes to be scheduled on the same day. Most distribution constraint types are evaluated pairwise for the affected classes, but four are evaluated using all classes for which they are defined. Soft distribution constraints have a nonnegative penalty value.

A feasible solution satisfies all hard constraints concerning class, time, and room assignment and student sectioning, and all hard distribution constraints. We measure the quality of a solution by the weighted sum of soft distribution constraint penalties, class time and room assignment penalties, and the number of student conflicts. A student conflict occurs when a student attends two classes that overlap in time or are scheduled such that it is impossible to arrive in time for the second class due to travel distance. The university defines weights for time assignment, room assignment, student sectioning, and distribution constraints according to their priorities and preferences.

The competition consists of 30 instances (see Table 1) made publicly available in three separate stages: Early, Middle, and Late. The competition's goal is to find the best possible solutions for each instance before the competition deadline. The ranking of competitors is determined using a scoring scheme where instances released later in the competition are given a higher score. The Late instances are released 10 days before the final deadline.

The instances vary significantly in terms of size, characteristics, and complexity. For example, one of the Early instances, *tg-fal17*, consists of only 36 courses with 711 classes, 15 rooms, and no student sectioning. This is contrasted by the Middle instance *pu-proj-fal19*, which includes a staggering 2839 courses with 8813 classes, 768 rooms, and 38,437 students. However, the complexity of an instance is not only determined by the number of classes, rooms, students, etc. Other aspects such as the types and number of distribution constraints, course structures, and class/room/student utilization have a significant effect.

The problem presented by the ITC 2019 is fascinating as it provides a unified problem definition that can encompass the practical timetabling problem arising at many different universities. This enables the competition to use real-world data, which the organizers collected using the timetabling system UniTime (Müller et al. 2018b). The ITC 2019 problem is based on the model used in UniTime, with some simplifications to reduce modeling complexity while retaining hardness. For a complete problem description, we refer the reader to Müller et al. (2018a).

Table 1 Some characteristics of all 30 competition instances

Instance	Courses	Classes	Rooms	Students
agh-fis-spr17	340	1239	80	1641
agh-ggis-spr17	272	1852	44	2116
bet-fal17	353	983	62	3018
iku-fal17	1206	2641	214	–
mary-spr17	544	882	90	3666
muni-fi-spr16	228	575	35	1543
muni-fsps-spr17	226	561	44	865
muni-pdf-spr16c	1089	2526	70	2938
pu-llr-spr17	687	1001	75	27,018
tg-fal17	36	711	15	–
agh-ggos-spr17	406	1144	84	2254
agh-h-spr17	234	460	39	1988
lums-spr18	313	487	73	–
muni-fi-spr17	186	516	35	1469
muni-fsps-spr17c	116	650	29	395
muni-pdf-spr16	881	1515	83	3443
nbi-spr18	404	782	67	2293
pu-d5-spr17	212	1061	84	13,497
pu-proj-fal19	2839	8813	770	38,437
yach-fal17	91	417	33	821
agh-fal17	1363	5081	327	6925
bet-spr18	357	1083	63	2921
iku-spr18	1290	2782	208	–
lums-fal17	328	502	97	–
mary-fal18	540	951	93	5051
muni-fi-fal17	188	535	36	1685
muni-fspsx-fal17	515	1623	33	1152
muni-pdfx-fal17	1635	3717	86	5651
pu-d9-fal19	1154	2798	224	35,213
tg-spr18	44	676	24	–

4 Mixed-integer programming model

The core part of our solution approach is the graph-based MIP model (referred to as the *full MIP* model) defined by Holm et al. (2020). The full MIP model is very comprehensive and beyond the scope of this paper. Thus, we settle for introducing the main decision variables required for this work and provide a minimal overview of the MIP model.

The ITC 2019 problem is defined by a set of classes $\mathcal{C}$ where each class $c \in \mathcal{C}$ must be assigned one of its available times $t \in \mathcal{T}_c$ and rooms $r \in \mathcal{R}_c$. If the class does not need a room, then $\mathcal{R}_c = \{\tilde{r}\}$, where $\tilde{r}$ is a dummy room. Additionally, each student $s \in \mathcal{S}$ must be assigned classes such that they attend their required courses. The problem includes some basic feasibility constraints, e.g., student sectioning constraints to observe specific course/class structures, no room double-booking, and we must schedule all classes. In addition, distribution constraints restrict/penalize the assignment between two or more classes. We define $\mathcal{C}_\delta$ as the set of classes that are part of distribution constraint $\delta \in \Delta$, where Δ is the set of all distribution constraints.

Assignment of classes to times and rooms is a major part of the problem, which naturally leads to the main decision variable $x_{c,t,r} \in \{0, 1\}$ defined as

$$x_{c,t,r} = \begin{cases} 1 & \text{if class } c \text{ is scheduled in time } t \text{ in room } r \\ 0 & \text{otherwise} \end{cases}$$

Additionally, we define auxiliary variable $y_{c,t} \in \{0, 1\}$ as

$$y_{c,t} = \begin{cases} 1 & \text{if class } c \text{ is scheduled in time } t \\ 0 & \text{otherwise} \end{cases}$$

The $y_{c,t}$ variable makes defining some constraints much more straightforward and helps reduce the number of nonzero values in the model. Additionally, we focus on these two sets of variables in the implemented fix-and-optimize matheuristic.

Student sectioning is the other major part of the problem, for which we have defined $e_{s,c} \in \{0, 1\}$ to be 1 if student s attends class c and 0 otherwise. For most instances, the student sectioning part is not difficult compared to the class assignment. However, for instances like *pu-proj-fal19*, which has 38,437 students, it becomes quite complex and computer memory-intensive.

Therefore, we define a version of the MIP model that applies the student sectioning to a known, feasible timetable. For a fixed timetable, we know which class pairs overlap in time or have a large enough travel distance such that a student conflict is possible. Thereby, we can avoid generating numerous redundant student sectioning variables and constraints. Additionally, we can ignore the class assignment part of the problem, including distribution constraints, resulting in a much more manageable model. We refer to this model as *the student sectioning MIP* model. The student sectioning MIP model is thus similar to the full MIP model, but where the full model has variables related to the timetable, these are fixed and presented as parameters to the student sectioning MIP model. Thus, the student sectioning MIP model has only one main decision variable, the student-class assignment variable $e_{s,c}$.

5 Fix-and-optimize matheuristic

The fix-and-optimize matheuristic can be considered to be a large neighborhood search heuristic. The algorithm takes an initial solution and iteratively improves it by searching a large neighborhood around the current solution. The neighborhood is defined and explored using MIP by fixing a subset of variables before solving the model using a MIP solver. Therefore, the matheuristic is very applicable to problems where small instances can be solved to optimality, but large instances cannot. Fixing variables results in a subproblem with a smaller solution neighborhood, effectively reducing the more difficult MIP model to be more easily solved. The fix-and-optimize matheuristic uses this idea by iteratively fixing a subset of variables and solving the resulting MIP model. Any improving solution is used to warm-start the MIP model in the next iteration.

This approach has a few benefits. Since the algorithm is MIP-based and only fixes/unfixes variables, no direct deliberation is given to the constraints or the structure of the problem. Therefore, constraints can be added, removed, or changed, and the algorithm still works. This is in direct contrast to many move-based metaheuristics, in which some moves may rely greatly on specific constraints of the problem and can easily become obsolete as a result of a model change. The fix-and-optimize matheuristic always works on the same model, and therefore it is only necessary to build the MIP model once. Variable fixing is done so that a solution found in one iteration is always feasible regarding the variable fixing in the next iteration. Thus, the MIP solver is warm-started in each iteration, providing great performance benefits. Also, since a MIP model is solved in each iteration, we know the subproblem's lower bound, which can be used to guide the search on a higher level. Lastly, an implemented fix-and-optimize will automatically have performance increases as MIP solvers are themselves improved.

As fix-and-optimize can only improve known solutions, it must be given an initial solution. Therefore, it is dependent on other methods for generating an initial solution. We discuss the heuristics used for generating initial solutions in Sect. 6.1.

An important aspect of fix and optimize is the searched neighborhood, i.e., how and which variables we choose to fix. In Sect. 5.1, we discuss our implemented neighborhoods and a strategy for updating the neighborhood size dynamically throughout the search.

Algorithm 1 shows the implemented fix-and-optimize. The input is the problem instance, an initial solution, a neighborhood specification, and an initial neighborhood size. First, in line 1, the algorithm builds the MIP model for the given instance and sets the warm-start values of the initial solution. In the loop, using the given neighborhood and neighborhood size, the algorithm chooses a subset of variables to allow to remain free, and fixes all other variables of the neighborhood to their given values (lines 3 and 4). In line 5, the algorithm solves the resulting subproblem and updates the best solution if it finds an improving solution. Finally, in lines 6 and 7, the algorithm updates the neighborhood size based on the performance of the optimization process and unfixes all previously fixed variables. Note that we only update the size during the search, and the algorithm continues to use the same neighborhood.

Algorithm 1 Fix-and-optimize

```
Input: Instance $I$, Initial solution $S$, Neighborhood $N$ and initial neighborhood size $P$
Output: Solution $S$
1: $M \leftarrow$ Build MIP for $I$ and set warm-start for $S$
2: while termination condition not met do
3:     $V$ = GetVariablesToRemainFree($N$, $P$)
4:     Fix all variables of $N \setminus V$
5:     $S \leftarrow$ Solve $M$
6:     $P \leftarrow$ UpdateNeighborhoodSize()
7:     Unfix all fixed variables
8: end while
```

5.1 Neighborhoods

As stated in Sect. 4, the implemented fix-and-optimize focuses on the classes' time and room assignments, which the developed neighborhoods reflect. Each neighborhood is defined by a heuristic for choosing classes to allow rescheduling and the variables considered for fixing. Common to all class-choosing heuristics is that classes are considered course-wise; a list of all courses is randomized, and classes are picked by going through this list and extracting all classes associated with the given courses. This continues until the list includes enough classes to satisfy the size parameter (P in Algorithm 1). This parameter defines the percent of classes that should remain unfixed. Picking all classes associated with a course ensures some connection in the unfixed variables, as these classes are closely related and greatly influence each other.

Each class-choosing heuristic separates itself from the others by how it includes extra classes when considering a given course. Three heuristics for picking classes to remain unfixed are defined: standard (S), common (C), and adjacent (A).

The standard heuristic goes through the randomized course list, extracting all course classes. No additional classes are chosen. The common heuristic additionally includes classes that have common distribution constraints with any course classes. Thus, for every class of the course, we examine all distribution constraints containing that class and extract the other classes of those constraints. Note that we only extract the additional individual classes and do not consider the rest of their associated courses.

The adjacent heuristic uses a class-time conflict graph for including additional classes to unfix. We use the class-time conflict graph described by Holm et al. (2020), in which we define vertices to represent $y_{c,t}$ variables and have an edge between two vertices if the associated $y_{c,t}$ variables conflict with each other. We use information from almost all hard distribution constraints to define the class-time conflict graph, and therefore an edge may cover multiple different hard conflicts. In the adjacent heuristic, when considering a course, we also extract all classes adjacent to any vertex related to that course. Thus, using a conflict graph is similar to considering common distribution constraints, but only includes hard constraints and some additional information of classes with fixed times. As such, this heuristic focuses on unfixing class assignment variables that interdependently affect feasibility.

Our neighborhoods use one of the described heuristics, and they consider either the class-time assignment variables ($y_{c,t}$) or the class-time-room variables ($x_{c,t,r}$) for fixing/unfixing. Using the $y_{c,t}$ variable allows for room changes of all classes (even those with fixed time) in every iteration, while this is not possible when using the $x_{c,t,r}$ variables. We never fix the student-class assignment variable ($e_{s,c}$), and the student sectioning part of the problem is always left free. The combination of the class-choosing heuristic and assignment variables defines our neighborhoods, resulting in the six neighborhoods shown in Table 2.

Table 2 Six defined neighborhoods using three heuristics and two variable sets

Neighborhood	Heuristic	Variable set
S_X	Standard (S)	$x_{c,t,r}$
S_Y	Standard (S)	$y_{c,t}$
C_X	Common (C)	$x_{c,t,r}$
C_Y	Common (C)	$y_{c,t}$
A_X	Adjacent (A)	$x_{c,t,r}$
A_Y	Adjacent (A)	$y_{c,t}$

5.1.1 Updating neighborhood size

It is vital to continually control the neighborhood's size, as this has a significant effect on the difficulty of the subproblems. The goal is to have problems that are neither too hard nor too easy to solve; the neighborhood should be large enough to include new (hopefully improving) solutions but not so large that they are exceedingly time-consuming to find. Hence, we dynamically update the neighborhood size throughout the search to strike such a balance. Dynamically updating the neighborhood size is especially important for problems such as the ITC 2019 problem since the MIP model complexity varies greatly from instance to instance and may benefit from vastly different neighborhood size parameters.

We focus the implementation on solving the ITC 2019 problem in the competitive setting defined for the release of the 10 *Late* instances, which is 10 days before the competition deadline. Therefore, we have tuned the implementation toward a 10-day time frame to find the best possible solutions for each instance. Some instances lead to MIP models that are very difficult to solve, resulting in iterations where the solver spends much time on pre-solving the model and solving the root node linear program (LP) relaxation. We are more willing to spend some time solving easier subproblems than wasting much time trying and failing to solve difficult subproblems. Therefore, we implement a conservative neighborhood size updating scheme, such that we are more willing to decrease the neighborhood size than increase it. Additionally, the neighborhood size is drastically reduced if the solution process has not completed the pre-solve and solved the root node LP after an hour. The solving process is given a time limit of 1 h to reach the branch-and-bound tree's root node and 30 min to explore the branch-and-bound tree.

Algorithm 2a details how the neighborhood size is updated based on the solver performance. We update the neighborhood size by considering (a) how quickly the solver progresses through the solving process, (b) whether any improving solutions are found, and (c) the optimality gap. We check the decision conditions in a prioritized manner (lines 1, 4, and 6), such that when any condition is satisfied, we make the corresponding decision immediately. We return to the conditions shortly.

The initial size is $P = 25\%$, meaning that 25% of the variable set is left free, and we adjust the neighborhood size in steps of d (set to 5%). We settled on an initial size of 25% through preliminary experiments. The update algorithm has a Boolean list H to save previous neighborhood update decisions. In each iteration, we add *true* to the H list if the size should increase and *false* otherwise. The neighborhood size is updated conservatively, favoring decreasing by doing so immediately upon such a decision (Algorithm 2b). Alternatively, if the neighborhood size should increase, we check the H list and increase the size if three of the last five inputs are *true*, i.e., if the last few iterations heavily favor increasing (Algorithm 2c).

Regarding the decision conditions, in line 1 of Algorithm 2a, we consider a special case where the solving process does not reach the root node of the branch-and-bound tree within an hour. In such cases, the model is far too difficult to solve, and we drastically reduce the neighborhood by dividing the current size by two and rounding down to the nearest divisor of d. Additionally, we reset the H list since the information from previous iterations is of little value following such a substantial change. In lines 4–5, we decide to decrease the neighborhood size if it takes longer than 30 min to reach the root node when solving the subproblem. In lines 6–11, we consider the case where no solutions are found and decide to increase the neighborhood size if the relative optimality gap is less than 5% and decrease otherwise.

If none of the checked conditions are satisfied, i.e., the solver quickly reached the root node and found improving solutions, we do not change the neighborhood size and add *false* to H (line 13). Finally, we return the potentially updated neighborhood size in line 15.

Algorithm 2a UpdateNeighborhoodSize()

```
Given: Neighborhood size $P$ and boolean list $H$
1: if Not reached root node in one hour then
2:     $P \leftarrow P/2$ rounded down to nearest divisor of $d$ without remainder
3:     Reset $H$
4: else if Time to root node > 30 minutes then
5:     Decrease($P$, $H$)
6: else if Found no improving solutions then
7:     if Gap < 0.05 then
8:         Increase($P$, $H$)
9:     else
10:        Decrease($P$, $H$)
11:    end if
12: else
13:     Add false to $H$
14: end if
15: return $P$
```

Algorithm 2b Decrease(P, H)

```
1: Add false to $H$
2: $P \leftarrow P - d$
```

Algorithm 2c Increase(P, H)

```
1: Add true to $H$
2: if at least 3 of last 5 inputs in $H$ are true then
3:     $P \leftarrow P + d$
4: end if
```

6 Parallelized matheuristic

In this section, we describe the parallelized matheuristic, in which we run multiple search methods in parallel. First, we provide a brief overview to demonstrate the basics of the algorithm. In the subsequent subsections, we thoroughly describe individual aspects of the complete algorithm.

Figure 1 shows the general flow of the parallelized matheuristic. Rounded boxes and rectangles represent methods/algorithms and types of solutions, respectively. Given a new instance, we first reduce it as described by Holm et al. (2020) to remove redundancies and tighten the instance data. Afterward, we start all methods in parallel:

- The 2SCA and the 3SCA methods (described in Sect. 6.1) produce initial solutions.
- A MIP solver starts solving the full MIP model.
- A number of fix-and-optimize processes prepare for an initial solution by building the MIP model.

As discussed in Sect. 5, the fix-and-optimize processes require an initial solution to begin the search. Thus, once a fix-and-optimize method is initialized, it looks for and uses the current best-known solution as its initial solution. Early in the search, the current best-known solution is typically an initial solution produced by the initial solution heuristics, but the full MIP solver could also produce it.

While the search is moving forward, i.e., the best-known solution is regularly updated, the fix-and-optimize processes collectively remain in a state of intensification. In this state, the fix-and-optimize processes and the full MIP solver collaborate through solution sharing, as detailed in Sect. 6.2. If the search stagnates, the parallelized algorithm enters the diversification state described in Sect. 6.3. Here, the fix-and-optimize searches abandon solution sharing, and each process separately starts a new search using individual initial solutions. When the best-known solution is improved (by any method), the fix-and-optimize processes reset to this solution and resume solution sharing (intensification).

For instances with many students, the algorithm includes changes and additions shown as the Large Student Sectioning

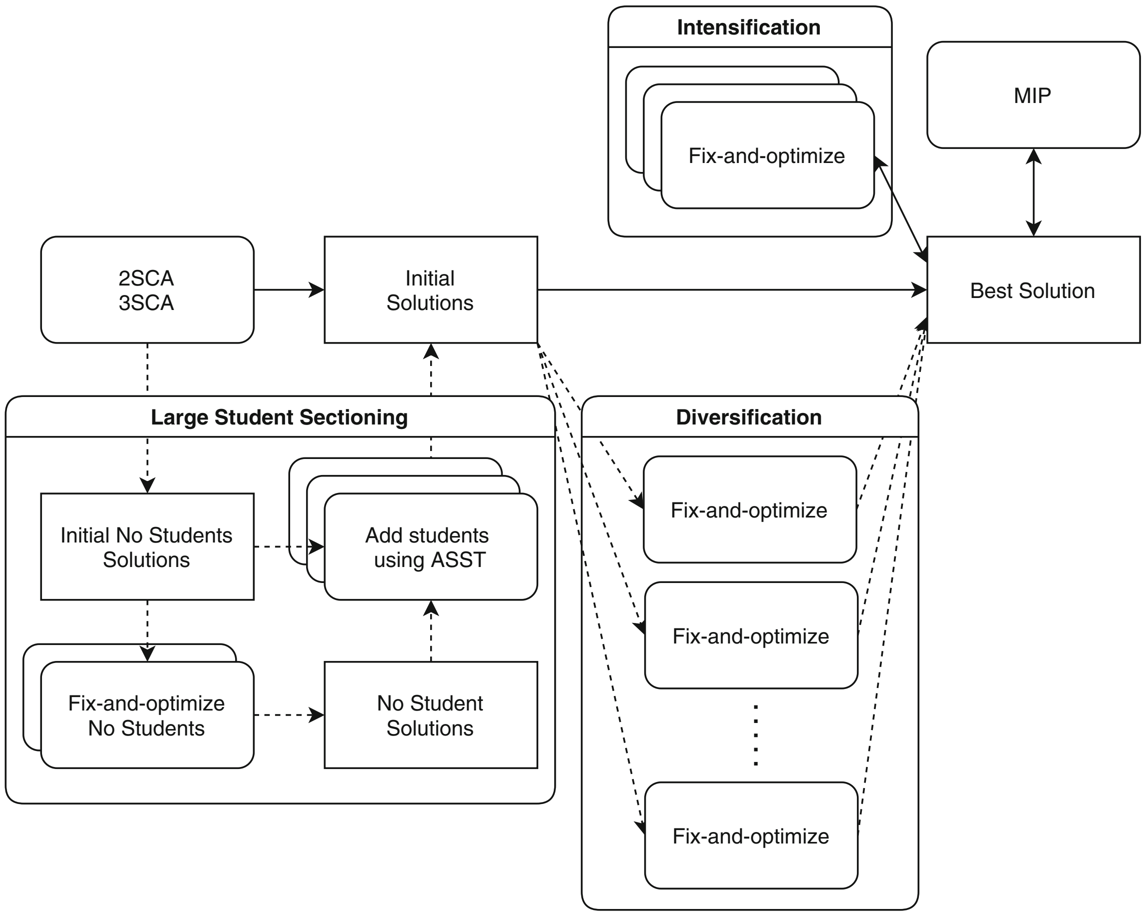

Fig. 1 Flowchart illustrating the parallelized matheuristic, showing the solution flow, method collaboration, and the changes imposed by the "Large Student Sectioning" and "Diversification" modes

part of Fig. 1. In essence, the parallelized algorithm decouples producing timetables and solving student sectioning when generating initial solutions. Section 6.4 discusses the Large Student Sectioning setup.

6.1 Initial solutions

We use two MIP-based constructive heuristics to generate initial solutions: the two-stage constructive algorithm (2SCA) and the three-stage constructive algorithm (3SCA). In these methods, we assume that there always exists a feasible student sectioning, since the problem defines student conflicts as soft violations. For valid data, the hard constraints associated with student sectioning, i.e., class limits and valid student-course-class assignments, can always be satisfied. Therefore, given any feasible timetable, students can always be added afterward without losing feasibility, although it may cause a significant number of student conflicts.

The 2SCA (shown in Algorithm 3) constructs a feasible solution in two stages by first generating a feasible timetable (lines 1–2) and then adding student sectioning (lines 3–4).

Algorithm 3 Two-Stage Constructive Algorithm

```
   Input: Instance $I$
   Output: Solution $S$
1: $M \leftarrow$ Build MIP for $I$ with only unassigned classes objective and no student sectioning
2: $S_M \leftarrow$ Solve $M$ to optimality
3: $M_s \leftarrow$ Build student sectioning MIP for $I$ from $S_M$
4: $S \leftarrow$ Solve $M_s$ with time limit of five minutes and relative gap of 0.01, or until a solution is found
```

The 2SCA uses a modified MIP model to generate a feasible timetable. The modified MIP model ignores students and soft constraints, allows unscheduled classes, and the objective function only includes the number of unscheduled classes. Thus, a solution with an objective value of 0 is a feasible timetable for the full MIP model. When the algorithm finds such a solution, it uses the student sectioning MIP model to create a student sectioning for the given timetable. The result is a feasible timetable with student sectioning.

The 3SCA (shown in Algorithm 4) constructs a feasible solution in three stages. The first two stages use a modified MIP model that ignores soft distribution constraints,

Algorithm 4 Three-Stage Constructive Algorithm

```
    Input: Instance $I$
    Output: Solution $S$
 1: $M \leftarrow$ Build MIP for $I$ with no soft distribution constraints, no
    student sectioning, and only time and room assignment penalties as
    objectives
 2: do
 3:     Amend $M$ to allow for unassigned rooms
 4:     $S_1 \leftarrow$ Solve $M$
 5:     Fix time-assignment of $M$ from $S_1$ using $y_{c,t}$
 6:     Amend $M$ to not allow for unassigned rooms
 7:     $S_2 \leftarrow$ Solve $M$
 8:     if $M$ is infeasible then
 9:         Unfix and cut off current time-assignment in $M$
10:     end if
11: while $M$ is infeasible
12: $M_s \leftarrow$ Build student sectioning MIP for $I$ from $S_2$
13: $S \leftarrow$ Solve $M_s$ with time limit of five minutes and relative gap of
    0.01, or until a solution is found
```

students, and only has class time and room assignment penalties in the objective function. The first stage assigns classes to times by using a version of the MIP model that allows for not assigning classes to a room (lines 3–4). Stage 2 fixes the class-time assignment of stage 1 and forbids classes with unassigned rooms (lines 5–7). Solving the MIP model of stage 2 results in either a feasible timetable or an infeasible model. In the case of an infeasible model, we cut off the class-time assignment before going back to stage 1 (lines 8–10). When the algorithm finds a feasible timetable, it proceeds to stage 3, where it uses the student sectioning MIP model to add student sectioning to the found timetable (lines 12–13).

When solving the first MIP model in the 2SCA, we allow the MIP solver to run until a solution is found with an objective value of 0 (solved to optimality). When solving the models in stages 1 and 2 of the 3SCA, we give the MIP solver a time limit of 5 min and a relative optimality gap of 0.05. However, since we need a solution to proceed to the next stage, we force the search to continue if the solver does not produce a solution within the time limit. Specifically, we use callbacks to override the solver time limit and only allow the search to terminate if it has found a solution. In cases where we extend the search, we end the search immediately upon the first solution found. Both the 2SCA and the 3SCA add students to the found timetables by solving the student sectioning MIP with a time limit of 5 min and a relative optimality gap of 0.01. Similarly, as described above, we force the search to continue until it finds a feasible solution.

Comparatively, the 2SCA randomly chooses a feasible timetable, and the 3SCA considers quality by using a greedy approach for assigning times and then rooms. Thus, the 3SCA might prove to be slower, but it should have a higher probability of providing higher-quality initial solutions than the 2SCA.

We enable the 2SCA and the 3SCA to produce additional initial solutions by allowing them to run multiple times. In such cases, they add a cut after finding a feasible timetable, forbidding the previous solution, forcing the algorithm to find a new feasible timetable. We show the cut used in the 2SCA below. S_n is the set of $x_{c,t,r}$ variables set to 1 in stage 1 in the nth iteration of the 2SCA. The cut for the 3SCA uses $y_{c,t}$ instead of $x_{c,t,r}$.

$$\sum_{S_n} x_{c,t,r} \leq |S_n| - 1$$

Furthermore, the 2SCA and the 3SCA can skip the last stage and only generate timetables without assigning students. As mentioned, the produced timetables are feasible in the full MIP model, and student sectioning cannot make the complete solution infeasible (can only add penalties through student conflicts). This fact is an important feature that we use in Sect. 6.4.

6.2 Solution sharing

For the fix-and-optimize matheuristic, we use several neighborhoods that may have varied performance on different instances. We have no clear way to determine a priori which neighborhood is better suited for a given instance and therefore opt to use them all, with a single fix-and-optimize process using each neighborhood. Naturally, it would be beneficial to share good solutions between each search, to ensure that none falls significantly behind. This also enables us to use each neighborhood to continually improve the best-found solution with their different focuses and strengths.

In such a scheme, it is crucial to strike a balance between allowing each fix-and-optimize process to search its own neighborhood and simultaneously not waste time where no improvements are possible. Therefore, each fix-and-optimize search checks whether there is a better-known solution every five iterations. If such a solution exists, the fix-and-optimize uses it going forward. Each fix-and-optimize iteration uses a maximum of 1.5 h, and therefore the process performs such a check at most every 7.5 h.

The parallelized algorithm also solves the full MIP model in parallel to the fix-and-optimize searches. As the MIP model is likely only competitive with fix-and-optimize processes on smaller instances, we run the MIP solver tuned to focus on improving the lower bound, as this information is very valuable for evaluating the state of the search. The MIP solver continually watches for and sets any new best-known solution (using callbacks), as this can be useful for managing the size of the branch-and-bound tree and speeds up the search. However, the MIP solver requires some time for reading and setting new solutions, which does take time away from improving the bound.

6.3 Diversification

Even with solution sharing, there is a risk of the fix-and-optimize searches getting stuck in a local optimum. Therefore, we include a diversification scheme to attempt to escape from such a local optimum and continue the search. However, in cases where the collaborative search stagnates, and the method begins the diversification search, we find it unlikely that we can find considerably better solutions. Thus, we include this diversification scheme in an attempt to find the best solutions possible but do not expect it to have a significant impact on the average performance of our method.

The implemented parallelized matheuristic focuses on the final 10 days of the competition, where the goal is to find the best possible solutions on the Late instances. With this in mind, we have chosen that the fix-and-optimize processes enter diversification mode after 12 h with no improving solution. When diversifying, the fix-and-optimize processes "scramble." They each reset to a random new initial solution (generated by the 2SCA or the 3SCA) and use that as a starting point for a new search. They stop solution sharing, and each fix-and-optimize process uses all three heuristics for choosing classes to unfix, chosen at random in each iteration. The fix-and-optimize processes use all heuristics to counter the effects of turning solution sharing off by not limiting a search using a heuristic that may not be well suited for a given instance. We make no change to the variable set used for fixing, so we only allow the class-choosing heuristic part of the neighborhood to change.

Each fix-and-optimize process needs to determine when to abandon the current diversification search and reset to a new initial solution. Once more, it is essential to strike a balance between giving the search a chance and not wasting too much time where success is unlikely. Algorithm 5a details the heuristic used for determining when to abandon the current diversification search and try again from another initial solution. If no new initial solution is available when the algorithm determines to restart, the search simply continues and resets to a new solution when one becomes available, unless we have found an improving solution in the meantime. This is the role of input parameter F, which is used in line 3 and updated in line 7. Lines 3–5 check whether it was previously decided to abandon the search ($F = true$), and the current solution is non-improving. Lines 6–9 are relevant when the new solution improves, and F is updated to allow the search to continue normally and returns not to abandon the search. The diversifying search is always allowed to continue for at least five iterations, which is ensured by lines 10–12. Finally, we impose a limit on the number of consecutive iterations where the search does not find improving solutions. This limit is dependent on the gap between the best solution of the current diversifying search and the best-known solution. Algorithm 5b shows the limits used (ranging from two to five), which are defined to allow for more leniency as the search gets closer to the best-known solution value. Whenever the search resets to a new initial solution, the algorithm resets all parameters to their default values and clears the diversification-solution value list (B).

Algorithm 5a AbandonDiversifySolution(S, S^*, B, F)

```
Input: New solution $S$, best-known solution $S^*$, list of previous solution values $B$, boolean $F$ stating if failed to get new initial solution
Output: Boolean value stating whether to get a new solution for diversification
1: $B^* \leftarrow \min(B)$                ▷ Best solution found in this search
2: Add $S$ to $B$
3: if $F$ and $S \geq B^*$ then
4:     return true
5: end if
6: if $S < B^*$ then
7:     $F \leftarrow$ false
8:     return false
9: end if
10: if length of $B \leq 5$ then
11:     return false
12: end if
13: $L \leftarrow$GetConsecutiveNonImprovementLimit($B^*$, $S^*$)
14: return If last $L$ solutions in $B$ have equal values
```

Algorithm 5b GetConsecutiveNonImprovementLimit(B^*, S^*)

```
Input: Current best solution $B^*$, best known solution $S^*$
Output: Integer number of consecutive non-improvement limit
1: $G \leftarrow$ gap from $B^*$ to $S^*$
2: if $G \geq 20\%$ then
3:     return 2
4: else if $G \geq 10\%$ then
5:     return 3
6: else if $G \geq 5\%$ then
7:     return 4
8: else
9:     return 5
10: end if
```

When the parallelized matheuristic finds a new best-known solution, all fix-and-optimize processes jump to this solution and reset all parameters. The search continues normally with solution sharing turned on and each fix-and-optimize process using its designated heuristic for choosing classes to unfix.

6.4 Large student sectioning

Some competition instances include a very large number of students, which causes problems with regard to computational memory and time when using the full MIP model. This is at least the case with *pu-proj-fal19*, which has the largest number of courses, classes, and students of all competition instances. Holm et al. (2020) noted that the full MIP model for *pu-proj-fal19* requires more than 256 GB of RAM to store in memory. Therefore, we include some additional methods and a special setup for such instances, allowing for decoupling timetable development and student sectioning.

By separating the class assignment and student sectioning problems, the respective MIP models become much more manageable. However, solving the class assignment problem without considering students comes at the cost of no look-ahead, which may result in poor solutions. It may be the case that high-quality timetables disregarding students yield many student conflicts, which is especially unfortunate if student conflicts have a relatively large weight. Therefore, the aim of the extension is not to pursue the best possible solutions in general, but to improve the chances of finding feasible solutions for large and difficult instances.

For instances with 30,000 students or more, we invoke a unique setup with a few changes/additions. We denote this setup Large Student Sectioning (LSS). One addition is the Add Student Sectioning to Timetables (ASST) method, which takes a timetable as input and adds students using the student sectioning MIP described in Sect. 4. We generate valid timetables without students using the 2SCA and the 3SCA heuristics by each heuristic skipping their last stage. Furthermore, we run two additional fix-and-optimize processes which both ignore student sectioning. One uses $y_{c,t}$ and the other uses $x_{c,t,r}$ for variable fixing. We include these two searches to produce timetables of high quality (disregarding student sectioning). In order to produce many feasible timetables, these two fix-and-optimize processes are always in diversification mode, meaning that there is no solution sharing, and they use all heuristics for getting classes to unfix. Additionally, we make these two searches more willing to abandon a diversification search, doing so immediately when they find a non-improving solution after the first five iterations.

The ASST method prioritizes using timetables produced by the two special fix-and-optimize processes. If none are available, it chooses a timetable produced by the 2SCA or the 3SCA. We solve the student sectioning MIP model with a time limit of 10 min, but if it finds no solution in that time, we force the solver to continue until it produces a feasible solution. We run five ASST in parallel, and they share a list of timetables that have had students added, such that they collectively only consider the same timetable once. We run multiple ASST processes because we have observed cases where the ASST method has difficulty keeping up with the influx of input solutions. The parallelized matheuristic considers the solutions produced by the ASST methods as initial solutions.

In summary, for instances with 30,000 students or more, we run the default setup with the addition of LSS. The result is that the constructive heuristics skip their last stage of adding students and only produce feasible timetables. Instead, five ASST processes add students to the timetables in a prioritized manner. To improve the relatively random timetables produced by the 2SCA and the 3SCA, we include two fix-and-optimize searches that ignore student sectioning. Hence, we run the default setup with some additional methods to increase the likelihood of finding feasible solutions.

6.5 Competition setup

For completeness, we describe the computational setup we used to find solutions for the Late instances during the final 10 days of the competition. As soon as the competition organizers made the data publicly available, we reduced each instance using the preprocessing techniques discussed by Holm et al. (2020). We then ran the complete parallelized matheuristic simultaneously on each Late instance for the remaining time of the competition, or until the full MIP solver had proven optimality.

We ran all algorithms in a cluster setting on 64 bit computers running Scientific Linux 7.7 equipped with 256 GB of RAM and two Intel Xeon E5-2650 v4 CPUs clocked at 2.20 GHz. We used Gurobi 8.1.1 as the MIP solver and had the following setup:

- One 2SCA and one 3SCA both producing at most 200 initial solutions and both using four threads
- One full MIP model solve with focus on bound using 16 threads
- Six fix-and-optimize processes, one using each combination of class-choosing heuristic (standard, common, adjacent) and variable set ($x_{c,t,r}$, $y_{c,t}$), all using four threads.

For instances with 30,000 or more students, we had the following changes/additions:

- The solution limit for the 2SCA and the 3SCA increased to 1000 initial solutions (without students)
- Two fix-and-optimize processes ignoring students, both using four threads
- Five ASST processes adding students to timetables, each using a single thread.

7 Computational results

This section evaluates the parallelized matheuristic through computational tests on all 30 instances used for ranking in the ITC 2019. Similarly as during the competition, we perform all computational tests in a cluster setting. However, here we use computers equipped with 756 GB of RAM and two Intel

Xeon Gold 6226R CPUs clocked at 2.90 GHz. Additionally, we use Gurobi 9.0 as the MIP solver using four threads (unless stated otherwise).

Although we have more RAM available here than during the competition, we are still unable to build the full MIP for *pu-proj-fal19*. We have implemented the algorithm in the .NET Framework 4.8 and, for these tests, run it through the Mono 6.8 runtime, which unfortunately consistently crashes while building the full MIP model after using approximately 350 GB of RAM. Therefore, in the following tests where we use the full MIP model, we have not gathered any data for *pu-proj-fal19*.

In the following, we investigate different aspects of the complete parallelized algorithm. Section 7.1 examines the initial solution heuristics. In Sect. 7.2, we evaluate the performance of the fix-and-optimize matheuristic. Section 7.3 considers the effects of solution sharing, and Sect. 7.4 examines the Large Student Sectioning setup. Finally, in Sect. 7.5, we examine the effects of the implemented diversification scheme.

7.1 Initial solutions

In the parallelized matheuristic, we use both the 2SCA and the 3SCA to generate initial solutions. We run the 2SCA and the 3SCA five times on each instance, and Table 3 shows the average objective cost and time to find the first solution. As expected, the 2SCA finds solutions more quickly than the 3SCA, but they are generally of lesser quality. The 3SCA is only quicker to find the first solution for one instance (*muni-fsps-spr17*), but in this case, both methods require little time, with only 39 and 157 s for the 3SCA and the 2SCA, respectively. Conversely, the 2SCA is many hours faster than the 3SCA for some instances. Instances *iku-spr18* and *muni-pdfx-fal17* represent extreme cases, where the 3SCA is not able to find a single feasible solution within 5 days, and the 2SCA does so in 21,564 and 5828 s, respectively. For most other instances, the 3SCA produces a solution of higher quality. The notable exception is the three *muni-fsps* instances, where the 2SCA produced better solutions. These three instances give much greater weight to student conflicts than all other timetable penalties. Thus, the extra time spent in the 3SCA on improving the timetable's quality without considering students is counterproductive as it increases the number of student conflicts, which is much more costly. Additionally, we note that the time-consuming and challenging part is to find a feasible timetable. For example, on *bet-fal17*, the last stage of building and solving the student sectioning MIP model for both heuristics requires less than a minute.

Figure 2 shows the solution values with and without students for the 2SCA and the 3SCA for instances *agh-ggis-spr17*, *pu-llr-spr17*, *muni-fsps-spr17c*, and *agh-fis-spr17*. We run both the 2SCA and the 3SCA using a time limit of 24 h and a solution limit of 100 solutions. All plots show that the 2SCA is faster at generating solutions but varies more in the objective value, which we expected since the first stage of the 2SCA produces timetables without considering any normal timetable-related objectives. Since the 3SCA does consider timetable quality, the solutions it produces are of more similar quality, which we see in particular in the plots for *agh-ggis-spr17* and *pu-llr-spr17*. These two plots also show a great difference in solution quality between the two algorithms, especially on *pu-llr-spr17*, where the 3SCA finds much better solutions. The plot for *muni-fsps-spr17c* shows the special case where the 2SCA consistently finds better solutions. For this instance, the 3SCA finds timetables (excluding students)

Table 3 An objective and time comparison of the first solution found by the 2SCA and the 3SCA on all 30 instances. Bold results are the best objective/time for that instance

Instance	2SCA		3SCA	
	Objective	Time (s)	Objective	Time (s)
agh-fis-spr17	39,210	**6768**	**27,339**	21,315
agh-ggis-spr17	183,854	**85**	**174,288**	604
bet-fal17	376,666	**42,032**	**331,940**	275,946
iku-fal17	161,312	**8208**	**32,078**	69,807
mary-spr17	70,194	**53**	**37,024**	210
muni-fi-spr16	16,519	**19**	**9402**	116
muni-fsps-spr17	**115,797**	157	142,878	**39**
muni-pdf-spr16c	542,507	**2209**	**503,734**	51,682
pu-llr-spr17	92,625	**83**	**24,859**	138
tg-fal17	30,036	**20**	**9921**	22
agh-ggos-spr17	95,185	**1305**	**71,515**	3075
agh-h-spr17	64,317	**11,304**	**49,885**	13,401
lums-spr18	1892	**196**	**394**	804
muni-fi-spr17	16,792	**29**	**10,056**	168
muni-fsps-spr17c	**506,576**	**264**	751,055	1297
muni-pdf-spr16	288,154	**335**	**132,251**	11,688
nbi-spr18	130,939	**35**	**43,151**	81
pu-d5-spr17	44,064	**139**	**28,216**	321
pu-proj-fal19	**564,671**	**21,398**	–	–
yach-fal17	**18,210**	**351**	18,529	654
agh-fal17	538,236	**5807**	**489,711**	25,024
bet-spr18	448,581	**7749**	**391,124**	95,225
iku-spr18	**188,769**	**21,564**	–	–
lums-fal17	2969	**233**	**1128**	1151
mary-fal18	29,505	**352**	**12,588**	700
muni-fi-fal17	21,462	**33**	**9761**	96
muni-fspsx-fal17	**845,095**	**1411**	1,015,658	8466
muni-pdfx-fal17	**804,202**	**5828**	–	–
pu-d9-fal19	327,100	**837**	**107,983**	4420
tg-spr18	101,058	**14**	**73,586**	19

with an objective value between 5000 and 10,000 less than the 2SCA. However, the cost of adding students to these timetables outweighs these differences. The plot for *agh-fis-spr17* shows how the 3SCA is only able to find three solutions within the 24-h time limit, as opposed to the 2SCA, which finds 75. However, these few solutions are of much greater quality than any produced by the 2SCA.

Although the 2SCA and the 3SCA have varying performance for different instances, the general observation is that the 2SCA is quicker to find solutions but of lower quality than the 3SCA. These two methods aim to quickly find an initial solution that the fix-and-optimize search can improve. However, it is perhaps more beneficial to wait longer for the better 3SCA solution than proceeding with the first 2SCA solution. To investigate, we run an experiment where we allow fix-and-optimize to run for 24 h, minus the time required to produce the initial solution.

Table 4 shows the average best solution found, running five separate fix-and-optimize searches using the initial solutions of the 2SCA and the 3SCA and the altered running times. We use the S_Y and S_X neighborhoods to obtain fair results for the default fix-and-optimize search setting. The 2SCA initial solution results in the best average solution eight and ten times for S_Y and S_X, respectively. Similarly, the 3SCA initial solution yields the best average solution eight and nine times for S_Y and S_X, respectively. Thus, the results show no clear superiority between the two initial solution heuristics. For only producing an initial solution, the 2SCA does so more robustly and quickly, producing an initial solution for all instances. However, the 3SCA is not without merit, as there are cases where it provides an initial solution of much better quality only shortly after the 2SCA, resulting in better performance for the following fix-and-optimize search. Additionally, we use the initial solutions generated as new initial solutions for fix-and-optimize during diversification, where it is typically better to use higher-quality solutions.

7.2 Fix-and-optimize processes

Using different heuristics for choosing classes to unfix is only relevant if these heuristics achieve different results. We compare the chosen classes to remain unfixed from each heuristic pair, using the same course list as input and extracting 25% of all instance classes. Table 5 shows the average percentage overlap in chosen classes for five different course list inputs. We expect some overlap because we give the same input to each heuristic, and all heuristics include all classes when considering a course. The overlap between the standard heuristic and the others is approximately 50%. As mentioned in Sect. 5.1, the adjacent heuristic is somewhat similar to the common heuristic, as it focuses mainly on hard distribution constraints. This similarity leads to an increased overlap, averaging 72.3% for all instances. Three instances have more

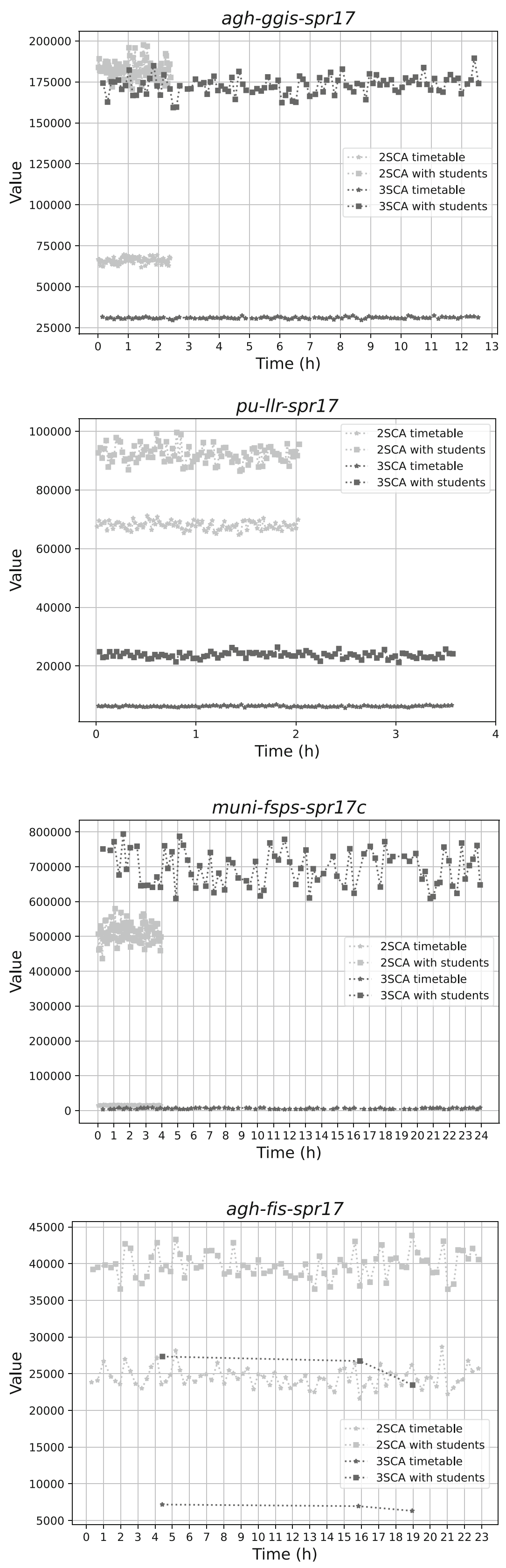

Fig. 2 Solution values per time for solutions generated by the 2SCA and the 3SCA on *agh-ggis-spr17*, *mary-fal18*, *muni-fspsx-fal17* and *agh-fis-spr17*

Table 4 Average best solution found by fix-and-optimize processes using 2SCA and 3SCA initial solutions. We run each fix-and-optimize search for 24 h minus the time required to find the initial solution (shown in Table 3)

Instance	2SCA		3SCA	
	S_Y	S_X	S_Y	S_X
agh-fis-spr17	5,156.0	4,518.0	5,536.6	**4,337.6**
agh-ggis-spr17	**46,204.2**	46,487.6	49,098.6	46,294.6
bet-fal17	345,482.4	**327,404.0**	–	–
iku-fal17	30,805.2	24,483.8	23,176.0	**21,307.4**
mary-spr17	16,358.0	15,762.2	**15,425.8**	15,444.8
muni-fi-spr16	3940.8	3988.4	**3,900.8**	3937.8
muni-fsps-spr17	870.6	870.2	870.8	**868.6**
muni-pdf-spr16c	121,942.2	**86,645.6**	225,511.4	146,454.4
pu-llr-spr17	10,143.0	10,184.6	**10,112.2**	10,157.0
tg-fal17	**4215.0**	**4215.0**	**4215.0**	**4215.0**
agh-ggos-spr17	11,381.4	**6754.6**	14,330.2	7386.6
agh-h-spr17	27,361.6	27,173.6	26,974.8	**26,614.2**
lums-spr18	99.0	98.2	99.6	**95.8**
muni-fi-spr17	4227.0	4369.6	4241.4	**4,192.8**
muni-fsps-spr17c	21,370.2	10,102.8	31,753.0	**7,061.0**
muni-pdf-spr16	34,173.8	28,103.6	34,709.4	**22,231.4**
nbi-spr18	**18,032.0**	18,171.0	18,097.4	18,064.6
pu-d5-spr17	20,366.0	22,321.8	**18,108.8**	20,009.4
pu-proj-fal19	–	–	–	–
yach-fal17	**4353.6**	5251.8	4882.0	5260.2
agh-fal17	333,869.4	**285,111.6**	348,112.6	311,511.6
bet-spr18	390,160.0	**389,610.6**	–	–
iku-spr18	**39,325.6**	51,181.8	–	–
lums-fal17	358.4	**358.0**	**358.0**	360.4
mary-fal18	**5291.0**	5684.8	5,446.4	5313.2
muni-fi-fal17	**3531.2**	3602.6	3595.0	3575.8
muni-fspsx-fal17	172,442.2	**40,678.0**	213,343.6	57,734.8
muni-pdfx-fal17	421,507.2	**162,142.4**	–	–
pu-d9-fal19	143,942.6	141,219.0	**56,230.2**	65,756.0
tg-spr18	**12,704.0**	**12,704.0**	**12,704.0**	12,772.0

Bold results shows the best for that instance

than 90% overlap due to the instances having a large percentage of hard constraints used in the conflict graph. For example, in *lums-fal17*, 91% of its distribution constraints are hard and are of types which contribute to the conflict graph.

To compare neighborhood and algorithm performance, we run the fix-and-optimize search using each neighborhood on all instances. We use the first initial solution available as shown in Table 3, i.e., the 2SCA solution for all but *muni-fsps-spr17*, which uses the 3SCA solution. When solving subproblems in the fix-and-optimize algorithm, we set the MIP focus parameter to "feasibility" (`MIPFocus=1` for Gurobi), prioritizing finding feasible solutions quickly. We compare the fix-and-optimize searches to the full MIP model, solved using default settings. To have a fair comparison, we warm-start the MIP solver with the same initial solution. We run all experiments for 24 h and using four threads.

Table 6 shows the average best solutions obtained for each neighborhood and the full MIP model solver. We run each test five times and highlight the best results using boldface. As expected, when comparing fix-and-optimize search and solving the full MIP model, the matheuristic generally yields much better results. For many instances, the full MIP model is simply too comprehensive for the solver to handle. All values marked with an asterisk in the UB column show cases where the MIP solver found only one or two solutions, where the first is the initial warm-started solution, and the second is a heuristic solution found during the pre-solve phase. In some cases, the solver did not even finish solving the root node relaxation. However, we also see cases where the MIP solver produces the best average results, most notably for

Table 5 Percentage overlap of classes for each neighborhood pair using the same course list input and extracting 25% of all classes

Instance	*S*–*C*	*S*–*A*	*C*–*A*
agh-fis-spr17	40.6	45.0	72.6
agh-ggis-spr17	45.3	65.3	52.0
bet-fal17	50.6	50.9	86.7
iku-fal17	34.4	33.1	69.3
mary-spr17	48.5	59.5	63.7
muni-fi-spr16	46.0	57.3	62.1
muni-fsps-spr17	45.4	45.8	79.4
muni-pdf-spr16c	37.3	42.3	73.4
pu-llr-spr17	61.9	56.4	55.7
tg-fal17	39.4	27.2	81.1
agh-ggos-spr17	44.2	44.3	83.2
agh-h-spr17	40.1	43.0	85.8
lums-spr18	35.8	37.3	94.0
muni-fi-spr17	59.5	60.5	78.2
muni-fsps-spr17c	57.6	58.7	90.9
muni-pdf-spr16	41.9	42.7	80.2
nbi-spr18	50.7	50.5	85.1
pu-d5-spr17	81.8	84.9	78.9
pu-proj-fal19	68.5	57.9	51.8
yach-fal17	59.6	67.2	71.4
agh-fal17	41.3	54.1	68.1
bet-spr18	47.1	48.3	89.0
iku-spr18	33.7	32.3	61.8
lums-fal17	37.3	38.0	99.1
mary-fal18	55.4	69.9	62.0
muni-fi-fal17	55.1	58.3	69.7
muni-fspsx-fal17	33.7	47.8	44.4
muni-pdfx-fal17	32.9	39.1	58.6
pu-d9-fal19	71.8	72.8	69.6
tg-spr18	27.0	30.8	50.8
Average	47.5	50.7	72.3

instance *iku-fal17*, where it has substantially better average performance than the fix-and-optimize searches. Additionally, the lower bound information provided by the MIP solver is valuable, indicating that we solve at least one instance to optimality and others with a small optimality gap.

The table also shows that the neighborhoods based on the standard heuristic generally have better performance than the others, finding a majority of the best average solutions. In fact, S-based neighborhoods resulted in the best average solution on 20 instances, C-based neighborhoods on five instance, and A-based neighborhoods on nine instances. In particular, C_Y and A_X have poor performance, finding the best average solution only on *tg-fal17*, which all methods individually solve to optimality. In most cases where either C or A has the best performance, the solutions of S are not far behind. The most notable exceptions are *pu-d5-spr17*, *iku-spr18*, and *pu-d9-fal19*, with gaps of 9.9%, 11.5%, and 13.3%, respectively, between the best-performing S-based neighborhood and the best-performing neighborhood. For *pu-d5-spr17*, the results are relatively stable, as the neighborhood with the largest relative standard deviation (RSD) is S_X, with a value of 3.81%, which is very stable. The results are less stable for *iku-spr18* and *pu-d9-fal19*, where S_X and A_Y, result in the largest RSD values of 28.97 and 13.63%, respectively.

In general, it seems that S-based neighborhoods result in the best performance of the fix-and-optimize search. The success of the S heuristic compared to the others may be because we give all methods an initial solution of low quality and a relatively short running time, considering that we have tuned the heuristic to a 10-day run. Perhaps the standard heuristic is especially good for an initial dive, while the others are better for thoroughly searching the solution space, once improving solutions are more difficult to find. Conversely, C and A may be well suited only for specific cases. For example, there certainly seems to be a pattern of A_Y resulting in good performance on *muni-fi-** and *pu-d** instances. In the future, we could investigate the connection between well-performing neighborhoods and instance characteristics.

We also see that for a given instance, fix-and-optimize will generally have better performance using either $x_{c,t,r}$ or $y_{c,t}$ for fixing, regardless of the class-choosing heuristic; i.e., if $S_X \geq S_Y$, then we likely have that $C_X \geq C_Y$ and $A_X \geq A_Y$. It appears that the favorable variable set is mostly related to the size of the instance and MIP model, such that large instances achieve better performance using $x_{c,t,r}$ and smaller instances using $y_{c,t}$. This separation makes sense, as using $x_{c,t,r}$ on difficult instances results in more constrained and more easily solved subproblems. Conversely, using $y_{c,t}$ on more easily handled problems results in subproblems with a larger solution space and allows for greater flexibility in the search.

Finally, for the fix-and-optimize matheuristic, we investigate the effects of dynamically updating the neighborhood size throughout the search. We do so by running fix-and-optimize using different fixed neighborhood sizes (5, 15, 25, 35, and 45%). To limit the number of runs, we settle for using only the best-performing neighborhood as shown in Table 6. Table 7 shows the average results of five 24-h runs compared to the default setting where the size starts at 25% and is dynamically updated (values taken from Table 6).

The results show that dynamic neighborhood sizes result in the best performance on nine instances. On the remaining 20 tested instances, a fixed neighborhood size results in the best performance, with nine instances favoring 15% and the other 11 instances distributed almost evenly among the other fixed sizes. However, dynamically updating the size is almost as good as the best fixed neighborhood size in all

Table 6 Best average solution and bound values found in 24 h

Instance	Fix-and-optimize						MIP	
	S_Y	S_X	C_Y	C_X	A_Y	A_X	UB	LB
agh-fis-spr17	5380.8	**4512.0**	5955.0	5173.6	5204.6	5133.6	17,129.0	875.0
agh-ggis-spr17	46,793.6	43,564.2	43,704.0	**40,861.4**	48,784.4	46,748.6	161,115.0	12,241.6
bet-fal17	323,545.2	**309,112.2**	348,339.2	327,760.0	345,421.4	327,480.0	375,833.0*	21,288.0
iku-fal17	28,848.6	24,355.4	28,594.6	24,134.4	23,164.6	28,287.2	**20,256.0**	16,151.2
mary-spr17	15,658.2	17,364.2	20,899.2	21,581.8	**15,293.6**	16,106.2	15,367.6	14,064.0
muni-fi-spr16	3957.0	3942.0	4196.0	4294.4	**3852.8**	4194.8	14,566.0	3301.0
muni-fsps-spr17	870.8	868.6	2391.2	4268.8	869.2	869.6	**868.0**	867.0
muni-pdf-spr16c	113,079.2	**82,564.2**	146,395.4	98,361.8	146,361.2	100,031.8	519,123.0*	0.0
pu-llr-spr17	10,183.2	10,221.4	24,925.2	35,823.2	10,119.4	17,643.8	**10,057.2**	9864.8
tg-fal17	**4215.0**	**4215.0**	**4215.0**	**4215.0**	**4215.0**	**4215.0**	**4215.0**	**4215.0**
agh-ggos-spr17	10,122.0	**6888.6**	13,889.8	9552.4	13,110.4	9695.8	59,472.0	851.0
agh-h-spr17	27,145.6	**26,255.8**	32,263.4	30,895.6	28,447.0	28,761.0	43,076.0	7222.0
lums-spr18	98.0	98.4	189.4	199.6	122.4	125.6	**95.0**	24.0
muni-fi-spr17	4371.8	4336.8	5080.8	5338.6	**4238.8**	4554.4	15,232.0	2249.0
muni-fsps-spr17c	22,234.0	**10,531.2**	31,002.8	15,592.4	29,360.2	19,485.0	469,457.0	246.0
muni-pdf-spr16	44,524.8	**29,486.0**	51,321.8	38,663.4	53,810.4	38,900.8	288,154.0*	0.0
nbi-spr18	**18,073.8**	18,096.8	18,284.6	18,323.0	18,105.4	18,337.2	18,181.0	17,646.6
pu-d5-spr17	21,201.6	23,774.6	22,014.0	24,021.2	**19,288.0**	22,194.4	27,007.0	4474.0
pu-proj-fal19	–	–	–	–	–	–	–	–
yach-fal17	**3795.0**	4444.0	4223.2	5816.0	4011.6	5949.2	16,217.0	514.0
agh-fal17	334,852.6	**286,555.2**	355,222.6	317,845.4	376,150.4	319,616.4	538,236.0*	1125.0
bet-spr18	392,999.2	**383,667.6**	408,694.4	394,488.6	410,585.0	395,778.6	448,581.0*	19,326.0
iku-spr18	38,532.4	39,503.6	37,572.4	**34,553.2**	36,984.6	39,968.2	35,126.2	20,607.2
lums-fal17	**357.6**	**357.6**	464.6	505.6	371.8	371.8	460.6	250.0
mary-fal18	**5355.0**	6161.8	8763.2	9516.6	5459.8	6605.8	13,783.0	2926.4
muni-fi-fal17	3764.4	3546.8	4346.4	4277.8	**3541.4**	3582.4	21,462.0*	1610.8
muni-fspsx-fal17	170,151.8	**44,470.0**	177,999.8	104,173.6	165,552.8	142,950.6	804,607.0*	1,406.4
muni-pdfx-fal17	402,218.0	**170,695.8**	461,588.0	219,444.2	490,537.8	248,263.8	778,492.0*	0.0
pu-d9-fal19	137,127.2	139,684.2	164,506.8	184,878.0	**121,022.4**	127,100.8	316,313.0*	0.0
tg-spr18	**12,704.0**	**12,704.0**	12,908.0	**12,704.0**	**12,704.0**	17,900.0	**12,704.0**	12,389.6

Bold results shows the best for that instance
*where the MIP solver found no solutions or a single heuristic solution during the presolve phase

cases, achieving the second-best performance on 16 out of the 20 instances.

Figure 3 shows the solution value progression of a single comparison run of the dynamic and fixed sizes searches and the dynamic neighborhood size for instances *yach-fal17*, *tg-fal17*, and *agh-ggos-spr17*.

The plots for *yach-fal17* and *tg-fal19* show opposite situations where fix-and-optimize obtains better results from a small and large neighborhood size, respectively. On *yach-fal17*, 5 and 15% yield the best results, and we see the dynamic neighborhood gradually reducing its size to a minimum of 5%, which allows it to lag less behind than the other runs. Conversely, the plot for *tg-fal17* shows the dynamic neighborhood increasing its size to 85%, allowing the fix-and-optimize search to produce an (optimal) solution costing 4215 in 32 iterations. The searches limited to a fixed neighborhood size have much worse performance. Unsurprisingly, the largest fixed neighborhood of 45% returns the second-best result, finding a solution costing 4357 in 7799 iterations.

The plot for *agh-ggos-spr17* shows a more balanced situation. For this instance, a fixed size of 15% results in the best average performance, followed closely by 25% and dynamic setting. The figure supports this, showing a quick dive in solution value for most settings but with the 15%, 25%, and dynamic runs the most persistent. The figure also shows the dynamic neighborhood size decreasing from the initial 25 to 10%, with a 5% decrease at iterations 7, 15, and 26. Thus, the dynamic neighborhood size does not quickly decrease but

Table 7 Best average solution values in 24 h for different fixed and dynamic neighborhood sizes

Instance	Neighborhood	No update					Dynamic
		5%	15%	25%	35%	45%	
agh-fis-spr17	S_X	8287.4	5217.6	4559.0	5280.4	6831.0	**4512.0**
agh-ggis-spr17	C_X	51,991.8	42,155.0	42,030.4	45,238.0	57,988.2	**40,861.4**
bet-fal17	S_X	322,693.0	**308,225.4**	329,497.2	356,191.6	365,520.6	309,112.2
iku-fal17	A_Y	36,018.6	24,568.8	23,275.6	23,446.0	23,807.4	**23,164.6**
mary-spr17	A_Y	18,889.4	15,521.8	15,426.0	15,442.6	15,425.4	**15,293.6**
muni-fi-spr16	A_Y	3968.2	3849.8	**3819.8**	3975.0	7461.8	3852.8
muni-fsps-spr17	S_X	1506.4	876.0	872.6	873.0	871.0	**868.6**
muni-pdf-spr16c	S_X	134,328.4	**75,917.0**	85,483.8	107,368.0	141,031.0	82,564.2
pu-llr-spr17	A_Y	10,345.4	10,237.6	10,168.6	**10,108.2**	10,176.6	10,119.4
tg-fal17	S_Y	8506.4	5449.0	4978.8	4707.4	4375.0	**4215.0**
agh-ggos-spr17	S_X	15,577.0	**6,381.6**	7,338.6	16,328.4	28,663.2	6888.6
agh-h-spr17	S_X	29,578.2	26,525.4	26,481.4	**24,465.2**	26,283.4	26,255.8
lums-spr18	S_Y	123.6	103.8	101.0	99.6	**97.6**	98.0
muni-fi-spr17	A_Y	5060.4	4260.4	**4206.8**	6421.4	8749.6	4238.8
muni-fsps-spr17c	S_X	19,699.8	11,174.0	**7,136.2**	49,897.0	225,230.0	10,531.2
muni-pdf-spr16	S_X	65,957.0	34,477.4	**28,038.8**	53,604.8	93,544.2	29,486.0
nbi-spr18	S_Y	21,755.8	19,403.8	18,436.4	18,311.2	18,174.2	**18,073.8**
pu-d5-spr17	A_Y	20,299.6	**19,230.0**	21,595.4	24,844.4	34,175.2	19,288.0
pu-proj-fal19	–	–	–	–	–	–	–
yach-fal17	S_Y	**2810.4**	2929.2	6684.4	7755.8	7954.8	3795.0
agh-fal17	S_X	295,353.4	316,191.4	340,419.0	464,210.4	482,725.0	**286,555.2**
bet-spr18	S_X	386,302.8	**381,474.8**	403,388.4	422,996.2	441,730.4	383,667.6
iku-spr18	C_X	44,922.2	37,993.0	33,326.6	**32,575.2**	34,089.6	34,553.2
lums-fal17	S_Y	458.4	384.8	381.6	363.4	**357.0**	357.6
mary-fal18	S_Y	7472.8	**5130.8**	6748.8	9253.2	13,952.2	5355.0
muni-fi-fal17	A_Y	3474.2	**3461.0**	3888.0	6232.4	11,294.0	3541.4
muni-fspsx-fal17	S_X	124,667.4	**39,755.2**	57,901.0	298,199.8	491,193.4	44,470.0
muni-pdfx-fal17	S_X	254,471.0	**147,203.8**	201,668.4	279,862.6	715,373.2	170,695.8
pu-d9-fal19	A_Y	**120,455.4**	203,953.2	296,574.6	323,173.2	327,100.0	121,022.4
tg-spr18	S_Y	20,498.0	17,952.8	17,732.4	16,724.8	12,874.0	**12,704.0**

Bold results shows the best for that instance

strikes some balance, consistent with the 15 and 25% fixed size having the best performance. In this particular comparison, the searches using the dynamic and 25% fixed sizes actually end up finding better solutions than the 15% run.

7.3 Solution sharing

In the parallelized matheuristic, we share solutions between the fix-and-optimize searches and the full MIP model solver such that most computational time can be spent where there is a chance of moving the search forward. Doing so is especially important for fix-and-optimize, where an individual search may fall behind due to a poor neighborhood fit or misfortune. By periodically moving the fix-and-optimize search to the best-known solution, we increase its chance of contributing to the overall search. While we also continually feed the best-known solution to the full MIP model solver, this is primarily to help prune nodes in the branch-and-bound tree.

To investigate the effects of solution sharing, we repeat the experiments of Table 6, with the same settings except that we include solution sharing. That is, we have five runs with six fix-and-optimize searches (one for each neighborhood) and one full MIP model solve for 24 h and have them share solutions as described in Sect. 6.2. Table 8 shows the average results, demonstrating that solution sharing yields better or similar results compared to no sharing in all cases except one. The exception is *pu-llr-spr17*, for which solution sharing in this experiment resulted in an average best solution value that is 0.6 worse than no sharing. However, this is a minute difference, and it is dramatically outweighed by the considerable

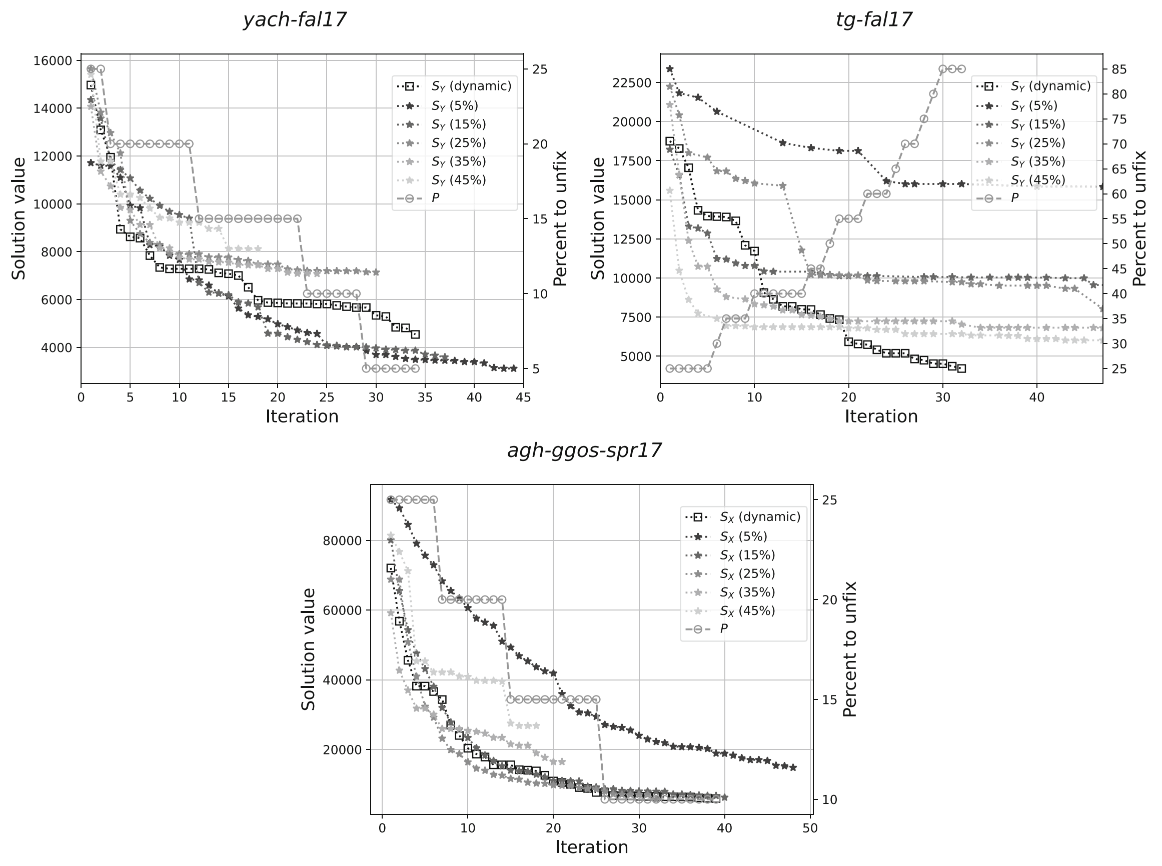

Fig. 3 Solution values and neighborhood size (P) per iteration for the dynamic and fixed neighborhood size fix-and-optimize searches on *yach-fal17*, *tg-fal17*, and *agh-ggos-spr17*

improvements seen for many other instances. The average improvement for all instances in the best-found solution value is 12.2%. This improvement rate is quite significant, especially considering that we have solved some instances to (or close to) optimality, and the room for improvement is thus relatively small. Of the 29 instances considered, 13 saw an improvement in the best-found solution value of more than 10%.

Table 8 also includes the average best-found lower bounds of the full MIP solver, for which we see no major change, with an average improvement of only 0.1%. However, this is not surprising considering the short running time. Solution sharing is beneficial for improving the bound when new solutions can help prune nodes in the branch-and-bound tree. The MIP solver begins branching for only nine of the instances. All others are working in the root node of the tree, some even still solving the root node linear relaxation. Some instances achieve a worse lower bound when sharing solutions, which may be a consequence of the MIP solver spending time parsing and handling past solutions, resulting in the solver changing its search strategy. However, given the short running time and the fact that we do not tune the MIP solver to focus on improving the lower bound, these results regarding the change in the lower bound produced are nonessential.

In all, solution sharing improves the overall search speed and consistently achieves better results within the first 24 h. Furthermore, although we see no improvement in the lower bound, this would be expected for longer running times.

7.4 Large student sectioning

The MIP model for *pu-proj-fal19* is massive, requiring a significant amount of RAM and making it intractable to solve both the full MIP model and the significantly constrained MIP model used in fix-and-optimize. To counter this problem, we introduced some additions that decouple timetable development and student sectioning. By using these methods, we can find feasible solutions for *pu-proj-fal19*. The additions are invoked for instances with 30,000 students or more, meaning that *pu-proj-fal19* and *pu-d9-fal19* are the only affected instances.

Table 9 shows the best solution values found in 24 h using three different setups on *pu-proj-fal19* and *pu-d9-fal19* five times. The tested setups are as follows. The default parallelized matheuristic where we run the 2SCA, the 3SCA, six

Table 8 Best solution and bound values found in 24 h with and without solution sharing

Instance	No sharing		Sharing		Improvement (%)	
	UB	LB	UB	LB	UB	LB
agh-fis-spr17	4512.0	875.0	3631.2	875.0	19.5	0.0
agh-ggis-spr17	40,861.4	12,241.6	38,772.6	12,146.0	5.1	−0.8
bet-fal17	309,112.2	21,288.0	305,422.0	21,288.0	1.2	0.0
iku-fal17	20,256.0	16,151.2	19,227.0	16,166.2	5.1	0.1
mary-spr17	15,293.6	14,064.0	14,927.2	14,026.4	2.4	−0.3
muni-fi-spr16	3852.8	3301.0	3831.6	3279.8	0.6	−0.6
muni-fsps-spr17	868.0	867.0	868.0	867.2	0.0	0.0
muni-pdf-spr16c	82,564.2	0.0	62,797.2	0.0	23.9	0.0
pu-llr-spr17	10,057.2	9864.8	10,057.8	9905.4	0.0	0.4
tg-fal17	4215.0	4215.0	4215.0	4215.0	0.0	0.0
agh-ggos-spr17	6888.6	851.0	4,854.2	851.0	29.5	0.0
agh-h-spr17	26,255.8	7222.0	22,601.0	7222.0	13.9	0.0
lums-spr18	95.0	24.0	95.0	19.8	0.0	−17.5
muni-fi-spr17	4238.8	2249.0	3972.4	2255.0	6.3	0.3
muni-fsps-spr17c	10,531.2	246.0	4,340.6	246.0	58.8	0.0
muni-pdf-spr16	29,486.0	0.0	23,496.6	0.0	20.3	0.0
nbi-spr18	18,073.8	17,646.6	18,015.2	17,663.0	0.3	0.1
pu-d5-spr17	19,288.0	4474.0	18,214.4	4474.0	5.6	0.0
pu-proj-fal19	–	–	–	–	–	–
yach-fal17	3795.0	514.0	2213.4	514.6	41.7	0.1
agh-fal17	286,555.2	1125.0	269,101.4	1125.0	6.1	0.0
bet-spr18	383,667.6	19,326.0	373,840.2	19,326.0	2.6	0.0
iku-spr18	34,553.2	20,607.2	27,684.0	20,360.0	19.9	−1.2
lums-fal17	357.6	250.0	349.8	250.6	2.2	0.2
mary-fal18	5355.0	2926.4	4640.8	2901.0	13.3	−0.9
muni-fi-fal17	3541.4	1610.8	3173.4	1591.0	10.4	−1.2
muni-fspsx-fal17	44,470.0	1406.4	33,306.6	1,758.0	25.1	25.0
muni-pdfx-fal17	170,695.8	0.0	143,734.2	0.0	15.8	0.0
pu-d9-fal19	121,022.4	0.0	90,635.2	0.0	25.1	0.0
tg-spr18	12,704.0	12,389.6	12,704.0	12,299.2	0.0	−0.7

fix-and-optimize, and one full MIP solve and include solution sharing. LSS has the 2SCA and the 3SCA skipping student sectioning, two fix-and-optimize improving those solutions (ignoring student sectioning), and five ASST adding students to the timetables produced. The last setup is “Default + LSS,” in which we combine the two, such that we also include the six fix-and-optimize searches and the full MIP solver.

The best solutions produced for *pu-proj-fal19* using the LSS setup vary somewhat in solution value; the average value is 204,011.2, but the greatest difference is 30,963. The solutions found for *pu-d9-fal19* using LSS are more consistent (the largest difference is 6017) but of much worse quality than those produced by the other setups. For *pu-d9-fal19*, we observe a slight benefit to using the default setup without the addition of LSS, with average best-found solution values of 48,333.4 and 52,748.6, respectively. We explain the difference due to the high quality of the first initial solution found by the 3SCA, which costs on average 107,983 and is found after 4420 s. In the default setup, this initial solution is significantly better than any solution found so far, and due to solution sharing, the fix-and-optimize searches soon afterward “jump” to this solution and continue the search from a much better solution. When using the LSS methods, the 3SCA skips the last stage of adding students, which the ASST methods instead handle. However, since the ASST methods prioritize investigating solutions produced by the specialized fix-and-optimize searches (that ignore students), the excellent 3SCA solution never has students added. Thus, the collective search does not have a chance for the same initial significant dive and instead has a more uniform search progression. However, it is interesting that the differences are almost negated within the 24-h time limit.

Table 9 Best solutions found for five runs of 24 h on the *pu-proj-fal19* and *pu-d9-fal19* instances using different setups

Run	*pu-proj-fal19*	*pu-d9-fal19*		
	LSS	Default	LSS	Default + LSS
1	188,892	48,348	88,598	53,147
2	219,855	46,147	94,615	53,540
3	199,040	49,657	92,027	50,449
4	198,095	48,449	88,799	54,781
5	214,174	49,066	89,305	51,821
Average	204,011.2	48,333.4	90,668.8	52,747.6

Finally, we note that the three tested setups for *pu-d9-fal19* achieved comparable or better results than those shown in Table 8, which is because those tests use the much worse 2SCA initial solution. It is noteworthy that given a time frame of 24 h, the LSS setup alone, i.e., methods designed to produce initial solutions, achieve the same solution quality as the default setup limited to using the weaker 2SCA initial solution.

7.5 Full setup with diversification

Here, we run the complete setup on all instances for a total of 10 days, closely mimicking the setup used during the final part of the competition. That is, we run the 2SCA and the 3SCA producing up to 200 initial solutions (both using four threads), a single MIP solver focusing on improving bound (using four threads instead of the 16 used in the competition), and six fix-and-optimize processes (each using a different neighborhood and four threads). Additionally, for *pu-proj-fal19* and *pu-d9-fal19*, we include the LLS setup. However, compared to the process during the competition, we use different computers and MIP model solver as described at the beginning of Sect. 7.

Table 10 shows the results of a single 10-day run of the complete parallelized matheuristic on all competition instances. We show both the best-known solution after the first 24 h and 10 days and the lower bound provided by the MIP solver. The algorithm has solved three instances to proven optimality and six other instances to an optimality gap of less than 5%. There are also instances where there is little to no difference between the best-known solution after 24 h and 10 days. However, there are a few difficult instances for which there is considerable improvement after the initial 24 h.

To investigate the effects of diversification, we examine the progression of solution values per time. Figure 4 shows such plots for instances *agh-h-spr17*, *iku-spr18*, and *agh-fal17*, respectively. The search stagnates and begins diversification for the first two instances. Note that since fix-and-optimize uses all heuristics for choosing classes while diversifying, the plots do not reflect the actual heuristic used during diversification but keep their original format for clarity.

For *agh-h-spr17*, the parallelized matheuristic enters diversification twice during the search. The best solution of the initial dive is found after approximately 78 h, with an objective value of 22,372. Diversification begins 12 h later, and at approximately 126 h (36 h into diversification), the search finds a new best-known solution with an objective value of 22,350. As seen in the figure, all fix-and-optimize searches reset to this solution and continue a collaborated search. The fix-and-optimize searches collaboratively improve the solution 84 times during the next 48 h, before once again stagnating (at a solution costing 21,559), resulting in a new diversification search that does not find any improving solutions for the remaining 54 h. However, the first diversification search indeed helped to move the search forward.

The search on *iku-spr18* shows close collaboration between the full MIP solver and the fix-and-optimize searches and that the full MIP solver can be extremely effective for pushing the search forward once improving solutions become challenging to find. The fix-and-optimize searches collectively enable rapid improvement of the initial solution, and after approximately 19 h, the full MIP solver processes its first best-known solution. From then on, the MIP solver and the fix-and-optimize searches all slowly improve upon the best-known solution. The fix-and-optimize searches find their last improving solution after 122 h, but the full MIP solver produces two additional improving solutions. The best-known solution of the initial dive was found after 129 h with a value of 25,875, and at that time, the best-known lower bound was 25,621, resulting in a relative optimality gap of 0.98%. The fix-and-optimize searches then enter the diversification mode but do not produce any new best-known solution. Instead, at approximately 215 h into the run, the full MIP solver alone pushes the search forward, finding a new best-known solution (costing 25,865). Since the best-known solution is improved, the fix-and-optimize searches stop diversification and resume the search from the solution. Shortly afterward, the full MIP solver again improves the best-known solution (to 25,864), providing a new solution for the fix-and-optimize searches to use. Still, they cannot produce improving solutions and begin diversifying again after approximately 232 h. However, the full MIP solver alone improves the best-known solution once more, and the fix-and-optimize searches begin intensification from that solution for the remaining time. In summary, this 10-day run of the parallelized matheuristic on *iku-spr18* resulted in a solution costing 25,863 and a lower bound of 25,752, yielding a relative optimality gap of 0.43%.

The search on *agh-fal17* does not begin diversification within the 10-day running time. Instead, the search pro-

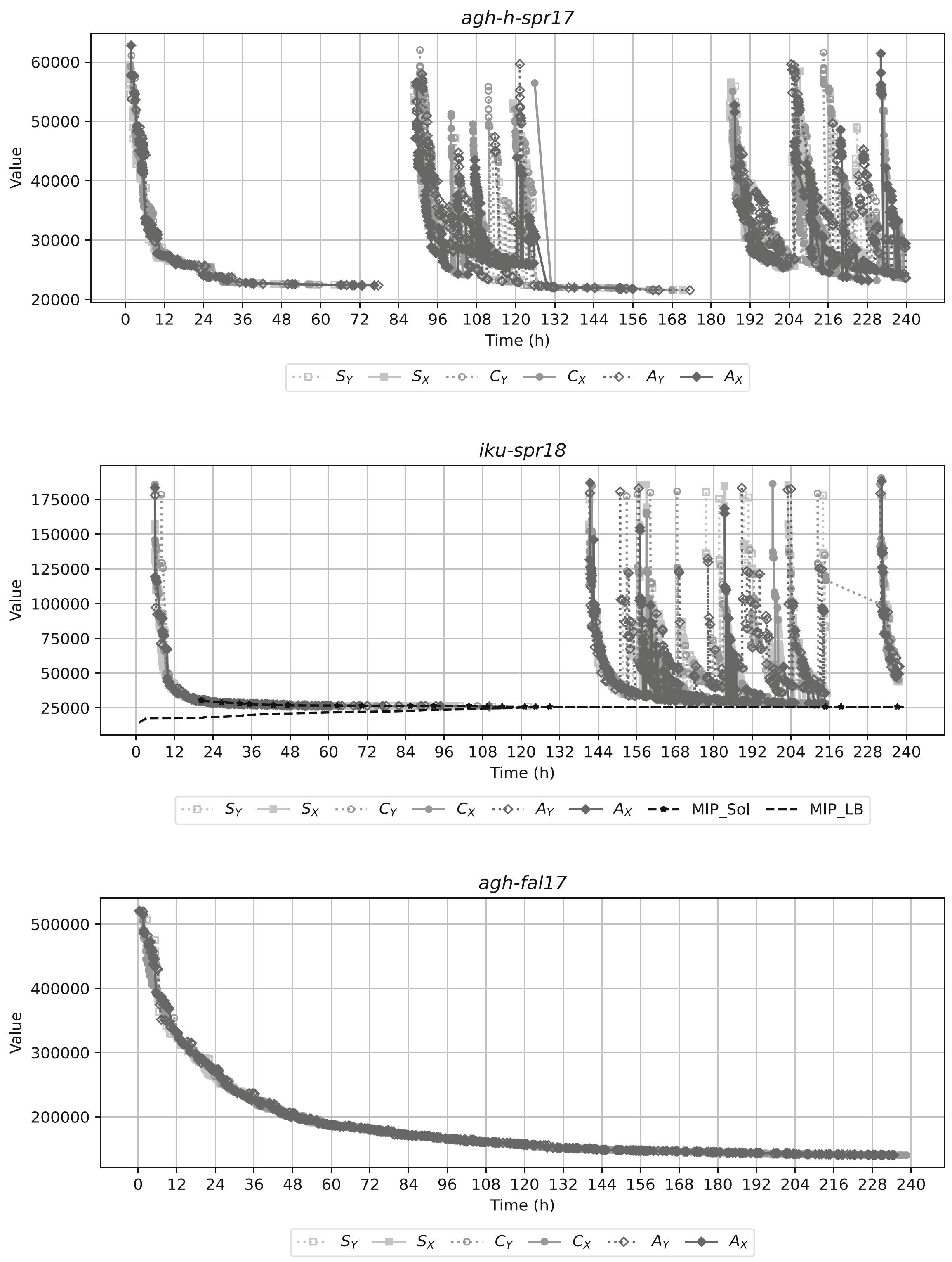

Fig. 4 Solution values per time using the complete parallelized matheuristic with diversification for 10 days on *agh-h-spr17*, *iku-spr18*, and *agh-fal17*

Table 10 Results of a single 10-day run of the complete parallelized matheuristic on all competition instances

Instance	Best solution		Lower bound	Gap (%)
	24 h	10 days		
agh-fis-spr17	3463	3094	1336	56.8
agh-ggis-spr17	38,026	35,147	16,556	52.9
bet-fal17	319,059	290,127	22,248	92.3
iku-fal17	19,498	18,989	18,069	4.8
mary-spr17	14,924	14,922	14,289	4.2
muni-fi-spr16	3766	3764	3602	4.3
muni-fsps-spr17	868	868	868	0.0
muni-pdf-spr16c	66,812	35,731	0	100.0
pu-llr-spr17	10,055	10,038	10,010	0.3
tg-fal17	4215	4215	4215	0.0
agh-ggos-spr17	4652	3237	1577	51.3
agh-h-spr17	23,883	21,559	7705	64.3
lums-spr18	95	95	24	74.7
muni-fi-spr17	3845	3796	2478	34.7
muni-fsps-spr17c	3777	2780	1360	51.1
muni-pdf-spr16	22,533	19,320	10,402	46.2
nbi-spr18	18,014	18,014	17,924	0.5
pu-d5-spr17	17,731	15,842	5923	62.6
pu-proj-fal19	219,832	178,135	0	100.0
yach-fal17	1717	1410	516	63.4
agh-fal17	261,826	140,194	1125	99.2
bet-spr18	375,677	350,410	61,821	82.4
iku-spr18	28,436	25,863	25,752	0.4
lums-fal17	349	349	252	27.8
mary-fal18	4546	4331	3385	21.8
muni-fi-fal17	3199	3129	1786	42.9
muni-fspsx-fal17	36,461	12,390	6328	48.9
muni-pdfx-fal17	138,916	84,703	0	100.0
pu-d9-fal19	47,938	39,251	0	100.0
tg-spr18	12,704	12,704	12,704	0.0

gresses slowly, and progressive improvements are visible in the plot until the end. No bound data are available, as the MIP did not finish solving the root node linear relaxation. Thus, on large instances like *agh-fal17*, the collaborative fix-and-optimize searches are essential for the complete parallelized matheuristic to produce high-quality solutions. In this run, the fix-and-optimize searches start from an initial solution costing 538,236 and produce a solution costing 140,194.

8 Conclusion

We have proposed a parallelized matheuristic for the ITC 2019 problem. The combined approach uses multiple different methods, all using the graph-based MIP model detailed by Holm et al. (2020). We use two different constructive heuristics to quickly find an initial solution and find additional solutions for future diversification. We run multiple fix-and-optimize searches in parallel and solve the full MIP model to obtain bounding information, continually sharing the best-known solution between each search to push the search forward more quickly. We have also proposed a diversification scheme which the parallelized matheuristic invokes when the search stagnates. Additionally, we have implemented a unique setup for instances with a considerable number of students, which decouples class assignments and student sectioning to find initial solutions.

Computational results show that the implemented fix-and-optimize matheuristic performs well in advancing the search. We investigated several aspects of fix-and-optimize, including the effects of the initial solution, the neighborhood, and dynamically updating the neighborhood size. We could not determine any definitive best initial solution heuristic but found that the choice of neighborhood and dynamically updating its size is essential for excellent performance.

Experiments also show that collaboration between multiple fix-and-optimize searches and the full MIP model solver is especially helpful in moving the search forward. Sharing solutions resulted in better performance on all but one instance compared to no solution sharing. The inclusion of a unique setup for instances with a large student sectioning was also beneficial. This addition enables the parallelized matheuristic to find feasible solutions on instances like *pu-proj-fal19*, which proved too memory-intensive for the other methods used. However, for the other affected instance (*pu-d9-fal19*), it provided no benefit. We also ran the complete parallelized matheuristic for 10 days, showcasing situations where the diversification scheme helped move an otherwise stuck search forward. Additionally, on the *iku-spr18* instance, this experiment showed a situation where the fix-and-optimize searches were likely incapable of moving the search forward but where the full MIP model solver did, further validating the solver's inclusion.

The proposed parallelized matheuristic has proved to be a competitive solution approach to the problem posed in the ITC 2019. It can find solutions for all competition instances, with some even to proven optimality. The merit of the approach is further attested to by the fact that it is the winning algorithm of the ITC 2019. In Appendix 1, we show the objective values of our submitted solutions and bounds at the time of writing.[1]

8.1 Future research

Although the parallelized matheuristic was proven to perform well, we have identified some immediate avenues for improvement, which we should address in future research.

We can improve the performance of an individual fix-and-optimize search by refining how it dynamically updates its neighborhood size. We should allow it to update the neighborhood size more fluidly and with smaller/larger changes than the implemented fixed increase/decrease of 5%. Furthermore, the algorithm should try to set an appropriate initial neighborhood size based on the instance instead of using a fixed initial size of 25%. In particular for very hard instances, this leads to wasted time, as the algorithm has to spend several iterations decreasing the neighborhood size to an appropriate level. Additionally, it might be a good idea to dynamically update other algorithm parameters during the search, especially the subproblem's time limit. For example, in cases where the algorithm converges and it becomes more challenging to find improving solutions, the algorithm should be given more time to thoroughly explore each subproblem. In such cases, the neighborhood size updating heuristic could also use other criteria for making decisions. For example, when the search converges toward the optimal solution, an initial low gap is unavoidable and should not be used in the decision-making process.

We can improve the complete parallelized matheuristic by further developing the diversification scheme. In the current implementation, the fix-and-optimize processes simply search individually and hope to find a new best-known solution, an approach that is somewhat dependent on randomness. Perhaps it would be beneficial to divide diversification into two phases. First, each fix-and-optimize would search individually and produce some high-quality solutions, and in phase 2, the fix-and-optimize processes would start sharing solutions again, using the potential solutions from phase 1 as starting points. Another interesting option is applying methods like path relinking (Glover et al. 2000) to combine high-quality solutions and using the results for further searching.

However, another option is to dedicate a single fix-and-optimize process to generating high-quality diversification solutions from the beginning. Then, once diversification is necessary, the other fix-and-optimize processes use these solutions instead of random initial solutions. Saviniec et al. (2018) describes such an approach for the high school timetabling problem, and it achieves state-of-the-art performance.

Each fix-and-optimize search uses only a single neighborhood during the default intensification search in our implementation. Solution sharing is one approach for countering problems resulting from such a rigid setting. Another option is to look into each search using multiple or all neighborhoods, and then to adaptively update how often it uses each neighborhood, similar to an adaptive large neighborhood search (Røpke and Pisinger 2006). That way, the fix-and-optimize search uses neighborhoods that consistently move the search forward more often than the underperforming neighborhoods. Such a design should help find better solutions more quickly.

We tried decomposing the problem into class assignments and student sectioning for instances with 30,000 or more students. This affected two instances: *pu-proj-fal19* and *pu-d9-fal19*. For *pu-proj-fal19*, the decomposition proved to be invaluable, but it provided no benefit for *pu-d9-fal19*. Perhaps the number of students should not be the only deciding factor for determining when to use this decoupling. For example, the search on *agh-fal17* showed moderate progression after many days of running the full parallelized matheuristic, indicating that it is an especially challenging instance to

[1] An updated table is available at https://dsumsoftware.com/itc2019/.

solve. This instance is also substantial (has the largest number of classes besides *pu-proj-fal19*) and could perhaps benefit from the same decoupling, even though it has "only" 6925 students. Additionally, we would like to examine other forms of problem decomposition.

Finally, we should also investigate how to improve the collaboration between search methods. Experiments show that solution sharing on average improves performance given a 24-h time frame. However, we could investigate more balanced collaboration. For example, in the experiment where we run the algorithm for 10 days on *agh-h-spr17* (see Fig. 4), we see a case where no collaboration yielded a better result more quickly. The initial dive of the search took about 78 h, with very slow progression after the first 36 h, meaning that the algorithm spends about 42 h on only very slight improvements. However, when the search begins diversifying and each fix-and-optimize searches individually, the algorithm finds a better solution in only 36 h. Therefore, we could investigate obtaining a better balance between intensification and diversification instead of simply turning solution sharing (collaboration) on or off.

Appendix A: Our submitted solutions and bounds

Table 11 shows the results submitted during the competition, the 10-day results from Sect. 7.5 (Table 10), and our best-known lower bounds. During the competition, we produced the results for the Late instances using the described parallelized matheuristic with the precise setup detailed in Sect. 6.5. Since the competition organizers published the Late instances 10 days before the deadline, we know that we produced our submitted results in less than that. However, we cannot say anything for sure regarding the time to produce our other results. The competition organizers released the Early and Middle instances earlier during the competition, and we have developed the parallelized matheuristic and its components using those instances. Consequently, we have often used previously found solutions for warm-starting MIP solvers and as initial solutions for fix-and-optimize searches and different parameter settings for our methods, including the number of cores.

Therefore, for comparison, Table 11 also shows the results of the 10-day runs produced for this paper. Here, we use hardware and software slightly different from what we used during the competition. Comparing the 10-day and competition results, we see that the presented parallelized algorithm can produce solutions comparable to those submitted in the competition on all instances.

Finally, the table also reports our best-known lower bounds, some of which we have produced with methods not discussed in this paper. Using these lower bounds, we see that we solved five instances to optimality both during the competition and in the 10-day runs.

Table 11 An overview of the best solution we submitted during the competition, the results of the single 10-day run (Table 10), and our best-known lower bounds

Instance	Best solution		Lower bound
	Competition	10-day run	
agh-fis-spr17	3081	3094	1336
agh-ggis-spr17	35,808	35,147	23,164
bet-fal17	290,086	290,127	89,278
iku-fal17	18,968	18,989	18,021
mary-spr17	14,910	14,922	14,359
muni-fi-spr16	3756	3764	3602
muni-fsps-spr17	868	868	868
muni-pdf-spr16c	36,487	35,731	14,279
pu-llr-spr17	10,038	10,038	10,038
tg-fal17	4215	4215	4215
agh-ggos-spr17	3055	3237	1982
agh-h-spr17	23,502	21,559	8,945
lums-spr18	95	95	24
muni-fi-spr17	3825	3796	2500
muni-fsps-spr17c	2596	2780	1361
muni-pdf-spr16	18,151	19,320	13,008
nbi-spr18	18,014	18,014	18,014
pu-d5-spr17	15,910	15,842	6981
pu-proj-fal19	148,016	178,135	54,872
yach-fal17	1239	1410	516
agh-fal17	186,200	140,194	5,728
bet-spr18	348,589	350,410	63,444
iku-spr18	25,878	25,863	25,781
lums-fal17	349	349	254
mary-fal18	4423	4331	3496
muni-fi-fal17	2999	3129	1890
muni-fspsx-fal17	17,074	12,390	7,747
muni-pdfx-fal17	117,412	84,703	26,711
pu-d9-fal19	43,006	39,251	28,000
tg-spr18	12,704	12,704	12,704

References

Bettinelli, A., Cacchiani, V., Roberti, R., & Toth, P. (2015). An overview of curriculum-based course timetabling. *Top, 23*(2), 313–349. https://doi.org/10.1007/s11750-015-0363-2

Burke, E. K., Mareček, J., Parkes, A. J., & Rudová, H. (2010). Decomposition, reformulation, and diving in university course timetabling. *Computers & Operations Research, 37*(3), 582–597. https://doi.org/10.1016/j.cor.2009.02.023 Hybrid Metaheuristics.

Burke, E. K., Mareček, J., Parkes, A. J., & Rudová, H. (2012). A branch-and-cut procedure for the Udine Course Timetabling problem.

Annals of Operations Research, 194(1), 71–87. https://doi.org/10.1007/s10479-010-0828-5

Di Gaspero, L., Mccollum, B., & Schaerf, A. (2007). The second International Timetabling Competition (ITC-2007): Curriculum-based course timetabling (Track 3). Technical report. Technical Report QUB/IEEE/Tech/ITC2007/CurriculumCTT/v1.0, Queen's University, Belfast.

Dorneles, Á. P., de Araújo, O. C., & Buriol, L. S. (2014). A fix-and-optimize heuristic for the high school timetabling problem. *Computers & Operations Research, 52*, 29–38.

Fonseca, G. H., Santos, H. G., & Carrano, E. G. (2016). Integrating matheuristics and metaheuristics for timetabling. *Computers & Operations Research, 74*, 108–117. https://doi.org/10.1016/j.cor.2016.04.016

Glover, F., Laguna, M., & Martí, R. (2000). Fundamentals of scatter search and path relinking. *Control and Cybernetics, 29*(3), 653–684.

Helber, S., & Sahling, F. (2010). A fix-and-optimize approach for the multi-level capacitated lot sizing problem. *International Journal of Production Economics, 123*(2), 247–256.

Holm, D., Mikkelsen, R., Sørensen, M., & Stidsen, T. (2020). A graph-based MIP formulation of the international timetabling competition 2019. *Journal of Scheduling*. https://doi.org/10.1007/s10951-022-00724-y

Kristiansen, S. & Stidsen, T. (2013). *A Comprehensive Study of Educational Timetabling—a Survey*. Number 8.2013 in DTU Management Engineering Report. DTU Management Engineering.

Lach, G., & Lübbecke, M. E. (2012). Curriculum based course timetabling: New solutions to Udine benchmark instances. *Annals of Operations Research, 194*(1), 255–272. https://doi.org/10.1007/s10479-010-0700-7

Lang, J. C., & Shen, Z.-J.M. (2011). Fix-and-optimize heuristics for capacitated lot-sizing with sequence-dependent setups and substitutions. *European Journal of Operational Research, 214*(3), 595–605.

Lewis, R., Paechter, B., & Mccollum, B. (2007). Post enrolment based course timetabling: A description of the problem model used for track two of the second International Timetabling Competition. In *Cardiff Working Papers in Accounting and Finance A2007-3, Cardiff Business School, Cardiff University*.

Lindahl, M., Sørensen, M., & Stidsen, T. R. (2018). A fix-and-optimize matheuristic for university timetabling. *Journal of Heuristics, 24*(4), 645-665.

McCollum, B., Schaerf, A., Paechter, B., McMullan, P., Lewis, R., Parkes, A. J., Di Gaspero, L., Qu, R., & Burke, E. K. (2010). Setting the research agenda in automated timetabling: The second International Timetabling Competition. *INFORMS Journal on Computing, 22*(1), 120–130. https://doi.org/10.1287/ijoc.1090.0320

Müller, T., Rudová, H., & Müllerová, Z. (2018a). University course timetabling and International Timetabling Competition 2019. In Burke, E. K., Di Gaspero, L., McCollum, B., Musliu, N., & Özcan, E., (Eds.), *Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling (PATAT 2018), Vienna, Austria* (pp. 5–31).

Müller, T., Rudová, H., & Müllerová, Z. (2018b). University course timetabling and International Timetabling Competition 2019. https://www.unitime.org/present/patat18-slides.pdf. Accessed 12 Apr 2021.

Røpke, S., & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science, 40*(4), 455–472. https://doi.org/10.1287/trsc.1050.0135

Saviniec, L., Santos, M. O., & Costa, A. M. (2018). Parallel local search algorithms for high school timetabling problems. *European Journal of Operational Research, 265*(1), 81–98.

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

Tan, J. S., Goh, S. L., Kendall, G., & Sabar, N. R. (2021). A survey of the state-of-the-art of optimisation methodologies in school timetabling problems. *Expert Systems with Applications, 165*, 113943.

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