Journal of Scheduling (2023) 26:497–517
https://doi.org/10.1007/s10951-022-00747-5

Simulated annealing with penalization for university course timetabling

Kadri Sylejmani[1] · **Edon Gashi**[1] · **Adrian Ymeri**[1]

Accepted: 9 June 2022 / Published online: 20 July 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

In this paper, we present our solver for the new variant of the University Timetabling Problem, which was introduced in the framework of Fourth International Timetabling Competition (ITC2019). This problem is defined on top of previous course timetabling problems in the literature, but introduces several new elements, both in terms of new features like student sectioning and new required and optional elements like distribution constraints. Our approach for solving this problem is based on the simulated annealing metaheuristic and consists of two phases. The first phase focuses on finding a feasible solution, and the second phase attempts to optimize the final score while keeping the solution feasible. Our solver detects local optima and applies gradual penalization to force solutions to new neighborhoods. The solver also detects required constraints which are difficult to satisfy and performs a specialized search on them. These adaptively applied mechanisms allow the solver to find feasible solutions for all problem instances of the competition. Results show that our solver gives good overall results and is competitive against other approaches presented in ITC2019.

Keywords ITC 2019 · University class timetabling · Simulated Annealing · Constraint penalization

1 Introduction

University timetabling is a well studied-problem in the literature that aims to model the scenario for automated scheduling of university courses by satisfying constraints like availability of classrooms, student preferences, hierarchical relationships between classes, and other complex constraints. Many commercial and non-commercial tools (e.g., Prime Timetable,[1] UniTime,[2] FET,[3] etc.) exist with the intention of automating the process of generation of course schedules, yet the problem still remains interesting and relevant. Such systems offer a more complete user experience, including rich presentation layers (Mall, 2018) which allow users to tweak parameters and give a more diverse input. While the presentation and data storage are important for any software system, in this paper we focus only on solving problems given a strictly defined input shape and features.

The International Series of Conferences on the Practice and Theory of Automated Timetabling (PATAT)[4] is a biyearly forum that supports many timetabling competitions. One of these competitions is The International Timetabling Competition which is devoted to university course timetabling. This paper proposes a solution to the problem presented in the fourth edition of the competition. The fourth edition (ITC2019[5]) presents the most complex university timetabling problem discussed so far in the

✉ Adrian Ymeri
adrian.ymeri@uni-pr.edu

Kadri Sylejmani
kadri.sylejmani@uni-pr.edu

Edon Gashi
edon.gashi@uni-pr.edu

1 University of Prishtina, Prishtina, Kosovo

1 https://primetimetable.com/.
2 https://www.unitime.org/.
3 https://lalescu.ro/liviu/fet/.
4 http://www.patatconference.org/.
5 https://www.itc2019.org/.

literature, including time, room, student and many class distribution demands.

Our solution is based on the simulated annealing metaheuristic (Kirkpatrick, 1984) which we transform to a simpler hill climbing (Lim et al., 2006) with random walks in specific search phases. Our solution method has been applied on the ITC2019 test set and managed to solve all of them, placing us first on the first and second milestones of the competition. Our solver also won third place on the final phase and the first prize in the open source category.

This paper is organized into six sections: Sect. 2 describes the University Timetabling Problem as proposed in the ITC2019; Sect. 3 discusses similar works in the field; Sect. 4 shows the solution approach and discusses the solver; Sect. 5 presents some experimental measurements and the comparisons to best known results for all data sets; and, in Sect. 6 we provide conclusions and comments on the future work for the University Timetabling Problem.

2 Problem formulation

With the aim of making the paper self-contained, in this section, we give a brief definition of the problem entities and accompanying constraints. We refer the reader to Müller et al. (2018) for the complete details of the formulation.

The specific entities of the problem include the following:

Times The course timetabling period has a certain number of weeks, a number of days in each week, and a number of time slots per day. A single time slot has a certain duration (usually 5 min), and it is used to model travel times from one room to the other, as well as model classes that meet at irregular times. A given class can meet several times a week; however, it may take place only in certain weeks of the semester (e.g., in odd weeks). Two particular classes are assumed to overlap if they share at least one week, at least one day of the week and they overlap within this day in at least one of their corresponding time slots.

Rooms Each room has a capacity, which denotes the number of available seats. A room may not be available at certain time slots during one or more days in the week over the course of the semester. If two given rooms are not closely collocated, a travel time between them is expressed in the form of number of time slots needed to travel from one room to the other. A class cannot be scheduled in a room at a time slot when the room is not available or when there is some other class placed in the room at an overlapping time slot. A given room may only be suitable for a specific set of classes.

Courses Each course consists of one or more configurations such that each student attends (some) classes in a given configuration only. Each course configuration consists of one or more teaching subparts/units (e.g., lecture, recitation or laboratory). Each student that enrolls for a given course must attend one class from each teaching subpart of a single course configuration. A given course subpart may consist of several classes, whereas a given class may have a parent class defined, which means that a student who attends a class must also attend its parent class. Each class has a set of possible time slots when it can meet. Each eligible time slot has a specified penalty when it is selected for a given class. Each class consists of set of rooms where it can meet, although the model allows assigning classes only to time slots and without rooms. Each eligible room has a given penalty when selected for a certain class.

Students Each student is associated with a list of courses that she or he needs to attend. Each student, for each of her/his courses, needs to be assigned in one class of every teaching subpart of a single course configuration. If there is a parent–child relation between classes of a course, the student must attend both classes in the relation. The number of students that can attend a given class is usually constrained to an upper limit. A student conflict happens when a given student is assigned in two classes that overlap in at least one time slot or they are scheduled one after the other in rooms that are too far apart (i.e., the travel time between the rooms is longer than the number of time slots between the time when the first class ends and the second class begins). Each student conflict is associated with a specified penalty, which does not depend on the number of actual class meetings (over the weeks of the semester) that are in the conflict or the number of the overlapping time slots.

In addition to the above constraints, that are applicable for entities such as times, rooms, courses and students, there are additional, so-called distribution constraints, that can be employed between any two or more classes. These constraints can be either required, which we refer to as hard constraints, or optional, which we refer to as soft constraints. These constraints can enforce specific rules for either time slots, days, weeks, rooms, or special relations between those. In the following, we present individual distribution constraints:

SameStart Any set of given classes must start at the same time slots of the day, regardless of the specific day of the week, or specific week of the semester.

SameTime/DifferentTime Any set of given classes must be assigned at the same/different time slots of the day, regardless of the specific day of the week, or specific week of the semester. In case of the SameTime constraint, where one of the classes is shorter, then the shorter class can start later, but it must not end after the longer class.

SameDays/DifferentDays Any set of given classes must be assigned in the same/different days of the week, regardless of their specific time slots of the day, or specific weeks of the semester. In case of the SameDays constraint, where one of the classes has fewer meetings during the week, then the

class with fewer meetings must be assigned to be taught on the subset of days used by the class with more meetings.

SameWeeks/DifferentWeeks Any set of given classes must be assigned in the same/different weeks of the semester, regardless of their specific time slots of the day, or specific days of the week. In case of the SameWeeks constraint, where one of the classes has fewer weeks during the semester, then the class with fewer weeks must be assigned to be taught on the subset of weeks used by the class with more weeks.

SameRoom/DifferentRoom Any set of given classes must be taught in the same/different room/s.

Overlap/NotOverlap Any set of given classes must be assigned to overlap (or not to overlap) in all corresponding time slots of a given day, in all corresponding days of the week and in all corresponding weeks of the semester.

SameAttendees Any set of given classes with same attendees cannot overlap in all corresponding time slots of a given day, in all corresponding days of the week and in all corresponding weeks of the semester. In case, such a set of given classes are assigned in the same days, and then they must be assigned such that the attendees have enough time to travel between consecutive classes that are taught in different rooms.

Precedence Any set of given classes must be assigned one after the other during the days of a given week. For classes that have multiple meetings in a week or that are on different weeks, this constraint is only applied for the first meeting of the corresponding classes.

WorkDay(S) On any given day, the number of time slots from first time slot of the first assigned class to the last time slot of the last assigned class should not be more than S time slots.

MinGap(G) Any two classes that are assigned on the same days (i.e., that overlap on days of the week and weeks of the semester) should have a gap of at least G time slots.

MaxDays(D) Any set of given classes cannot spread more than D days during the week, regardless of whether they are taught in the same week of the semester or not.

MaxDayLoad(S) Any set of given classes must be spread over the days of each and every week such that the sum of time slots of classes on every day does not exceed number S.

MaxBreaks(R, S) For any set of given classes, the maximal number of breaks during a day should not be more than the value of parameter R. A single break appears (is formed) when there are at least S free time slots between any two consecutive assignments of the set of given classes.

MaxBlock(M, S) For any set of given classes, the maximal number of time slots of classes assigned consecutively (denoted as a block of classes) should be less or equal to the value of parameter M. A block of classes appears (is formed) when there are no more than S free time slots in-between any two consecutive assignments of the set of given classes.

3 Related work

The literature on University Course Timetabling is very vast, which is why we decided to focus only on the work related to the series of timetabling competitions as discussed below.

The topic of the First International Timetabling Competition ITC2002 (Paechter et al., 2002) is related to the University Timetabling Problem, which consists of a set of events (classes) that have to be scheduled in a period of 45 time slots (5 days of 9 h each), a set of rooms where events take place, a set of students who attend the events, and some features that rooms need to possess in order for the events to take place there. There are three hard (required) constraints: no student can participate in multiple events simultaneously, the room size and features comply with the assigned event, and no multiple events can take place in a room simultaneously. There are also three soft (optional) constraints: a student has a class in the last slot of the day, there are multiple consecutive classes for a student during a day, and the student has only a single class in the day. These constraints determine the solution score.

Kostuch (2003) solves this problem by using an approach of two stages, where at the first stage a feasible timetable is constructed, and in the second stage a Simulated Annealing metaheuristic is utilized to optimize the solution based on the objective function. Further, Cordeau1 et al. (2003) tackles the problem of university timetabling by using an efficient Tabu Search metaheuristic, which is divided into two phases—in the first phase the goal is to find a feasible solution, whereas the second phase proceeds with the goal of solution improvement.

Bykov (2003) proposes a Great Deluge local search algorithm with a move operator that randomly moves one event from one time slot to another one. Gaspero and Schaerf (2003) employ a local search technique that consists of three sequential stages, namely Hill Climbing, Tabu Search and Multi-swap shake. An exhaustive list of approaches that solve this variant of University Timetabling Problem can be found in Eckersley (2004).

The second International Timetabling Competition ITC2007 (Di Gaspero et al., 2007) is organized in three tracks, where the second and third are about course timetabling. The Post Enrollment-based Course Timetabling (second track in ITC2007) models the situation where the timetable is constructed after the student enrollment, in such a way that all of them can attend the events in which they are enrolled. Cambazard et al. (2012) present an approach that is a hybridization of local search and constraint programming techniques, where a list coloring relaxation function is also employed. Further, the constraint programming approach uses a problem decomposition technique, which is incorporated into a large neighborhood search scheme. Méndez-Díaz et al. (2016) present a ILP-based heuristic for a generaliza-

tion of the post-enrollment course timetabling problem. The proposed approach is a two-stage heuristic procedure, where in the first stage the assignments of classes to time slots are made by considering a subset of students and a relaxation of the ILP model, whereas in the second phase, the ILP model is solved by fixing some of the variables to the values of the solution obtained in the first stage. Nagata (2018) propose a local search-based algorithm that has a mechanism for adapting the neighborhood size with the aim of constructing a random partial neighborhood that is used for controlling the trade-off between search space exploration and exploitation. Ceschia et al. (2012) design a metaheuristic approach based on Simulated Annealing, which consist of a neighborhood relations made of two moves, namely move events and swap events. The former move an event from its currently assigned time slot to another, whereas the latter swap the time slots of two distinct events. Goh et al. (2019) present a two-stage approach for solving the post-enrollment course timetabling problem. In the first stage, a feasible solution is constructed, whereas in the second stage, the solution is further improved by considering soft constraint violations. The presented approach is based on the Simulated Annealing metaheuristic that is empowered with a Reheating and Learning algorithm, which is a reinforcement learning-based methodology used to obtain a suitable neighborhood structure.

The Curriculum-based Course Timetabling (third track in ITC2007) is about weekly scheduling of lectures for a number of university courses in a given number of rooms and time periods, where clashes between courses are resolved according to the teaching curricula published by the university and not on the terms of student enrollment data. The problem's test set consists of real curriculum data from the university of Udine in Italy. Müller (2009) uses a constraint-based solver to tackle this variant of university timetabling. The solver's algorithm consists of three phases, namely: (1) generation of feasible solution by using an Iterative Forward Search algorithm that uses a Conflict-based Statistics approach to escape cycles, (2) Hill Climbing to find the local optimum, and (3) a Great Deluge technique (and optionally a Simulated Annealing metaheuristic) to escape from the local optimum.

Furthermore, Lü and Hao (2010) present an approach that consists of three main phases, namely initialization, intensification and diversification. The initialization phase uses a fast greedy algorithm to construct a feasible solution, while the second and the third phase use an adaptive intensification and diversification strategy, respectively. The earlier is based on Tabu Search by using a neighborhood structure of double Kempe chains, whereas the latter uses the perturbation mechanism of Iterated Local Search that is penalty based.

Geiger (2012) presents a local search strategy that is based on the principles of Threshold Accepting, which aims to escape local optima over the course of algorithm execution. This approach implements a stochastic neighborhood that randomly removes and reassigns events in the running solution. For a detailed overview of further approaches that solve the curriculum-based course timetabling problem, the reader is referred to Bettinelli et al. (2015). Bellio et al. (2016) propose a single-stage simulated annealing technique for the curriculum-based course timetabling problem. The neighborhood of the proposed method consists of two operators, which either move or swap lectures from a time slot or room. Moreover, authors employ a non-uniform probability strategy for selecting one of the moves of the neighborhood, which consists of two stages. Kalender et al. (2012) present an improvement oriented heuristic selection strategy, which is combined with a simulated annealing approach as a hyper-heuristic. Furthermore, the proposed approach uses a set of low level constraint-oriented neighborhood moves that help in construction of a method that is able to solve state-of-the-art instances, including a dataset generated at Yeditepe University.

The Third International Timetabling Competition was devoted to High School Timetabling; therefore, it is out of the scope of this work.

The Fourth International Timetabling Competition (ITC2019) has the topic of the University Timetabling Problem, which introduces several new hard and soft constraints when compared to the previous problems of similar nature. The novelty of this problem is that it enables student sectioning along with other standard time and room assignments for courses. This problem is modeled based on the real-world data sets that have been obtained from UniTime,[6] which is a widely used platform by the universities around the world when it comes to lecture timetabling. In the following, we briefly describe all of the five approaches that have been finalists in this competition, where the first is the winner.

Holm et al. (2021) presents a Mixed Integer Programming (MIP)-based algorithm that incorporates several different parts. The first part makes a preprocessing of the input instance, where several unnecessary information is removed. Further, this approach employs two algorithms for generation of initial solutions, and a Fix-and-Optimize metaheuristic, which are all based on a MIP formulation. Efstratios et al. (2021) present a mathematical programming approach, which is a mixed integer linear programming model that is solved using the branch-and-cut algorithm. This method employs several elements to boost the algorithm performance, such as preprocession of several characteristics of the instances, the aggregation of constraints and the efficient use of auxiliary variables in the problem formulation. Er-rhaimini (2021) presents an approach where the initial solution is created by stochastically assigning classes at random, until all of them are assigned, or removing them

[6] https://www.unitime.org/.

when no more choices exist for new assignments. In the iterative phase, the solution is improved by trying to insert the prioritized/unassigned classes into the timetable, while ensuring that the algorithm does not get stuck in infinite loops. Lemos et al. (2021) presents a MaxSAT approach, which uses the state-of-the-art solver TT-Open-WBO-Inc (Nadel, 2019). The performance of the algorithm is improved by preprocesing the input, whereas a neighborhood search procedure is employed to further optimize the solution.

On our side, we have solved this problem by using a customized simulated annealing approach (Gashi & Sylejmani, 2020), where penalization strategies have been employed in order to make the algorithm find a feasible solution first, and then be more flexible in escaping local optima. Our algorithm is able to solve all the instances, and in general it also finds good solutions when compared to other algorithms of metaheuristic nature. In the following sections, we first describe our approach in detail and then visualize the solution process with experiments.

Furthermore, there are several applications of the simulated annealing metaheuristic for the course timetabling problems in general. Gunawan et al. (2012) propose a solver for the university timetabling problem, which integrates both teacher assignment and course scheduling. The proposed solution consists of two stages, where in the first stage, an initial solution is obtained by a mathematical programming approach based on Lagrangian relaxation, whereas, in the second stage, the solution is further improved by a simulated annealing metaheuristic. The proposed algorithm has been tested on a real-life dataset from a university in Indonesia, as well as on several randomly generated datasets. Zheng et al. (2015) propose a simulated annealing metaheuristic for the curriculum-based course timetabling problem with extra traveling distance constraints. The proposed approach consists of a two-stage procedure, where the first one aims at satisfying the hard constraints violations, whereas the second one minimizes the soft constraints violations. Furthermore, the proposed simulated annealing technique uses some search functions/memories to incorporate a variety of space and time adjustment strategies within the neighborhood exploration mechanism.

4 Solution approach

4.1 Problem model and preprocessing

Before starting the search, some preprocessing is performed on the problem instances. All classes and students are normalized to have sequential IDs beginning from 0. This allows efficient storage and access in arrays. We hold a mapping to real IDs, which is needed when outputting the solution.

4.1.1 Time and room variables

Time and room variables are collected from classes. We consider a time/room assignment to be a variable if it has more than one option. We refer to the list of possible assignments of a variable as its domain. For each variable, we sort its domain by smallest penalty in an ascending order. If a class has only one room, then we remove any time assignments which overlap with the room's unavailable times from that class' time domain.

These variables are stored in their respective arrays: *TimeVariables* for time, *RoomVariables* for rooms, and *AllClassVariables* for all of them combined. We pick a random index from these arrays when performing a mutation. This simplifies the process of picking candidates for mutation purposes since we can never pick fixed variables that have a domain size of 1.

4.1.2 Course configuration variables

Because of parent–child relationships between classes in course configurations, the problem of assigning infeasible combinations arises if we mutate values randomly. For this reason, we create a simpler model where each course configuration acts as a variable with a list of feasible assignments. We find the possible assignments using a simple tree search. We only do this once at the preprocessing stage.

In Fig. 1, we show a course configuration with 6 classes. Parent–child relationships are described using edges which connect parents to children. We see that by picking class B we must unconditionally have class A in our solution, but the option for the next class is either C or F.

By performing a tree search, we discover 4 possible paths to connect all configuration subparts in this tree: ABC, ABF, DEC, DEF. We will refer to these assignment combinations as class chains. In practice, there aren't too many such configurations because a typical course has few classes which could also be further constrained in parent–child relationships.

This logic applies only for one course configuration. A class may have more than one configuration, where each configuration may have a different number of subparts. In

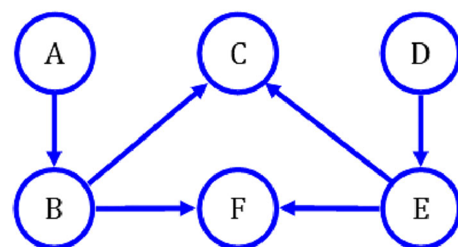

Fig. 1 A sample course configuration

such cases, we compute all paths for each configuration. In the end, we merge the paths to a single list.

Each of these chains is assigned to an index, and moving forward we consider a course enrollment as a variable with a domain of n elements, where n represents the number of class chains in the course configuration. Each class chain holds the ID of the configuration it belongs to and also the indexes of classes inside the subparts of that configuration. With this approach, we can refer to any feasible combination of a course by a simple integer value in interval $0...n-1$.

Like time and room variables, if a course happens to have only one class chain in total, then we do not consider it as a variable.

4.1.3 Worst penalty calculation

When evaluating a solution, we measure a penalty relative to the worst possible case it could have. We precompute the following worst cases.

Worst room penalty is calculated by adding up the worst (most penalized) room assignments of all classes.

Worst time penalty is calculated by adding up the worst (most penalized) time assignments of all classes.

Worst student penalty is calculated by assuming that all classes of a student are in conflict with each other. We get the number of classes c_n by adding up the maximal number of classes of each course the student is enrolled to. The number of maximal conflicts is calculated as $c_n * (c_n - 1)/2$.

Worst distribution penalty is calculated by adding up all the non-required distribution constraints assuming their worst possible configuration. Distribution evaluation is done using the rules as described in the problem statement. For some constraints it's applicable to use a mathematical formula with the highest input value, while for other conflict-related constraints we assume a conflict between all pairs of classes in the constraint $c_n * (c_n - 1)/2$.

Worst solution penalty is the sum of worst room penalty, worst time penalty, worst student penalty, and worst distribution penalty.

4.2 Solution model

A solution is the list of all variables and their values. We model solutions to always be complete. This means that variables are always assigned to some value, even if their configuration results to an invalid solution. We consider a solution to be infeasible (or invalid) when it has conflicts between classes, unavailable rooms being used, or unsatisfied required constraints.

A variable assignment in a solution is one of the following:

1. **Class time**: *Class ID* paired with a *Time Index*
2. **Class room**: *Class ID* paired with a *Room Index*
3. **Student enrollment**: The tuple *Student ID* & *Course Index* paired with *Class Chain Index*

Their internal data structure modeling is discussed in the following sections.

4.3 Solution representation

4.3.1 Representation of class time and room

A state in the search space is represented by three vectors, where, for each class (i.e., Class ID), the first vector indicates the room (i.e., Room Index) where the class takes place, whereas the second vector stores the time schedule (i.e., Class Index) when the class takes place, which points to the zero-based index of an element in the class's domain. If the domain has only one element (meaning there is only possible time schedule for that class), then its value is always 0. For example, if a class has its time schedule assignment set to 2, it means that the class's schedule is set to the third option in the list of possible schedules. There are cases where a class does not require any room space. In that situation, in the vector of room assignments, we store the value -1 in the corresponding index. Finally, in the third vector, we store the total number of enrolled students in each class. We need this to evaluate the constraint of capacity overflows.

4.3.2 Student variables

We store the state of students in a vector called *StudentStates*. For each element in this vector, we keep a list of *EnrollmentStates*. Elements in enrollment states vector match in order with the courses the student is enrolled in.

An element in the*EnrollmentStates* vector has a *ConfigIndex* which represents the configuration of the course that the student is enrolled in. Further, an element in *EnrollmentStates* also has an array of integers, where the index in the array represents the position of the subpart, whereas the value represents the position of the class in that subpart. Although this model allows for combinations which violate parent–child relationships of classes, we always assign these states from the precomputed chains *(i.e., Class Chain Index)* of their respective courses.

4.3.3 Distribution constraints

When evaluating a given distribution constraint, we return a tuple that contains both penalties (*hardPenalty*, *softPenalty*). Soft constraints return values of (0, p) and required constraints return values of (q, 0), where p and q are calculated according to the problem rules. For required constraints, we use a constraint penalty factor of 1, whereas the weight of soft constraints depends on the constants of the problem instance.

4.4 Initial solution

Initial solutions are deterministic. For each class, we set the time and room assignment to 0 (as long as the class has a room in its domain). We pick courses for each student i in interval $0 \ldots students_count$ by choosing a chain with ID $i \mod number_of_chains$. In other words, we attempt to distribute students to different chains in a pseudo-random order.

We have experimented with various greedy algorithms by assigning low-cost pairs of rooms and times, but due to the sheer number of constraints and combinations our attempts were fruitless, and we did not pursue such approaches further. The initial solution generation is a remnant of those experiments, and we never got to revisit it, mostly because we have found that the choice of initial solution does not make any significant difference in results. Overall, a purely random solution would be less complex and more intuitive. One benefit of starting from the same solution every time was that it helped us better understand the effects of various parameters and mechanisms of the algorithm during development.

4.5 Evaluation

We maintain three separate penalties to determine the quality and state of the solution: soft penalty, hard penalty, and student overflows. Although student overflow penalty affects the feasibility of the solution, we keep it separate because it is easier to satisfy compared to class conflicts and distribution constraints. The evaluation function used in the local search depends on the phase and feasibility of the solution.

4.5.1 Hard and soft penalty

Soft penalty is calculated using the rules as specified in the problem definition (Müller et al., 2018). Hard penalty is calculated based on the following rules:

1. A conflict between a pair of classes gives 1 hard penalty point.
2. A time assignment conflicting with a room's unavailable schedule gives 1 hard penalty point.
3. An unsatisfied required constraint gives hard penalty points equal to the soft penalty the constraint would give if it were not required (with weight 1).

4.5.2 Student overflow penalty

Student overflow penalty is calculated by adding up all over enrollments of all classes in the solution. For example, if we have a class with a capacity of 20 and we enroll 23 students, another class with a capacity of 25 and we enroll 26 students, the total overflow in this case is 4 (3 for the first class and 1 for the latter).

When evaluating (discussed later), we multiply the counted over-enrollments with the coefficient c_2 in Eq. 1

$$c_2 = \frac{overflow_factor \cdot student_penalty}{worst_soft_penalty} \tag{1}$$

In Eq. 1 *student_penalty* is the optimization constant that is specified in the input problem instance, *worst_soft_penalty* is the penalty calculated in problem preprocessing as discussed earlier, and *overflow_factor* is calculated based on Eq. 2. Further, *max_classes*, in Eq. 2, represents the theoretical maximum of classes a student can attend, or in other words, the number of classes if the student would have picked course configurations with most subparts for all their courses. Multiplying by 1.01 causes this penalty to always surpass any equivalent soft penalty points.

$$overflow_factor = 1.01 \cdot max_classes \tag{2}$$

4.5.3 Search penalty

Stern (1992) proposes a simulated annealing approach for solving the problem of matrix permutation, where a heuristic temperature-dependent penalty function is employed. This has shown to constantly outperform the standard approach, and as result, several other problems have been solved using similar strategies as describe by Henderson et al. (2003). In this regard, our solver applies three penalty components that are discussed so far, which are combined to evaluate the quality of a solution. Here we will discuss how these penalties are combined when evaluating a solution.

We define the search penalty in Eq. 3. When a solution is infeasible by having nonzero hard penalty, the search penalty is the sum of its hard penalty and the 2 digit (after decimal point) rounded sum of its class overflows and the normalized soft penalty. In Eq. 3 p_h stands for hard penalty, p_s for soft penalty, and p_c for class overflows penalty. Coefficient c_1 is an empirically defined constant with value $c_1 = 0.01$, whereas c_2 is calculated based on previously discussed Eq. 1.

The normalized soft penalty is the actual soft penalty divided by the worst possible penalty of the problem as shown in Eq. 4. We showed how the worst soft penalty is calculated in the problem preprocessing section.

Rounding the soft penalty helps create a "rolling effect" so as not to waste time in local optima, where micro-optimization processes are not helpful in making the solution feasible.

When a solution reaches zero hard penalty, the goal shifts to optimize class overflows and soft penalty, as shown in the second case of Eq. 3. As discussed earlier, the constant c_2

is computed to be sufficiently large in order to ensure class overflow elimination is prioritized.

$$\text{SearchPenalty}(s) = \begin{cases} c_1 p_h + \text{round}_2(c_2 p_c + \text{normalize}(p_s)), & p_h > 0 \\ c_2 p_c + \text{normalize}(p_s), & p_h = 0 \end{cases} \tag{3}$$

$$\text{Normalize}(p_s) = \frac{p_s}{\text{worst soft penalty of problem}} \tag{4}$$

4.5.4 Stochastic tunneling–Fstun

We further normalize the evaluation function based on the technique of stochastic tunneling (Wenzel & Hamacher, 1999). The function f_{stun} is given in Eq. 5. This function normalizes the evaluation function to a maximum of 1, where 0 is the best solution. Using this approach, extremely high values of the evaluation function become flattened.

$$f_{\text{stun}}(x) = 1 - \exp\left[-\gamma(f(x) - f_0)\right] \tag{5}$$

During the phase of the simulated annealing search (discussed later), the energy difference between a current solution and a candidate solution is the difference between normalized values of the search penalty, as defined by Eq. 6. The function f_{search} is the generalized case for any evaluation function. As discussed later in Sect. 4.9, we use the normalized value defined in Eq. 3 and also add situational penalties (Eq. 9).

We have found that normalizing the evaluation function to known bounds made it easier for us to define problem agnostic constants and temperatures.

$$\Delta E(s', s) = f_{\text{stun}}\left(f_{\text{search}}(s')\right) - f_{\text{stun}}\left(f_{\text{search}}(s)\right) \tag{6}$$

4.5.5 Constraint-focused evaluation

During the random walk phase of the search process (discussed in Sect. 4.10), we define the focused search evaluation function. This modified evaluation function, as outlined by Eq. 7, is given as the sum of the regular normalized search penalty and the sum of the focused constraints' hard penalty. Parameter *focus* is a subset of unsatisfied hard constraints. For a given constraint x in the set *focus*, $\text{constraintPenalty}(x)$ is the integer value calculated according to the third rule in Sect. 4.5.1.

$$\text{focusedPenalty}(s, \mathit{focus}) = f_{\text{stun}}(\text{searchPenalty}(s)) + \sum_{x}^{\mathit{focus}} \text{constraintPenalty}(x) \tag{7}$$

The goal of this search strategy is to find similarly scored solutions that have persistently failing constraints satisfied.

4.6 Neighborhood operators

A mutation is a single operation which changes the schedule of a class, the room of a class, or the enrollment configuration of a student attending a particular course. In concrete terms, it is one of the following operations:

- Change the value of a time variable to a new random index.
- Change the value of a room variable to a new random index.
- Change the configuration of a student's course to a new random index.

We do not mutate student enrollments if the solution has nonzero hard penalty. We have found that attempting to optimize non-critical components before having a feasible solution only slows down the search. These micro-optimization elements are lost during search restarts (discussed later) anyway.

A room or time mutation follows these steps:

1. Pick a random variable from the list of variables.
2. Assign the time/room of the variable's class to a new random value $0 \ldots n-1$, where n is the domain size of the variable.

An enrollment mutation follows these steps:

1. Pick a random variable from the list of student-course variables.
2. Get a random class chain from the variable's course. Enroll the variable's student to the classes corresponding to that chain.

In both cases, the new solution is evaluated by calculating only the difference caused by the mutation. Using this strategy of delta evaluation helps achieve considerable performance improvements.

During simulated annealing search (discussed later), mutations are picked randomly and uniformly from a pool (list) of possible operators. The list of operators is different when the solution is feasible compared to when it is infeasible. Values of distributions are shown in Table 2 under "MutationOccurrences" parameters, where for example FeasibleTimeMutationOccurrences represents the number of time mutations present in the operator pool when the current solution is feasible. If a problem has no type of variable, for example student enrollments, then no such operators are added to the pool. As an example, suppose we have 2 time

mutation occurrences, 2 room mutation occurrences, and a random variable occurrence. The chance for a time mutation being picked is 2/5=40%, the chance for a room mutation being picked is 2/5=40%, and the chance for a randomly picked variable (room or time) mutation is 1/5=20%. It does not necessarily mean that the chance for a time mutation is 50%, because picking a random variable uniformly does not mean that it has an equal chance of getting either. Some classes have fixed time variables, and some classes have fixed room variables or no rooms at all. Given this fact, the distribution of variables can vary, and the random variable mutation attempts to reduce biases if either variable type dominates the problem.

Because of the possible imbalance discussed above, for infeasible solutions our operator pool consists of: 2 occurrences of random variable mutations, 1 occurrence of random time mutation, and 1 occurrence of random room mutation. This means that at all times (if such variable exists) we have at least a 25% chance of mutating a time and likewise a room variable, and 50% of mutating whatever variable is most statistically likely to occur when picked uniformly. We do not include enrollment mutation operators at this stage because they do not affect solution feasibility.

When the solution becomes feasible, we include student enrollment mutations in the operator pool. Table 2 summarizes the distributions used at this stage. A double enrollment mutation means changing 2 random enrollments to new randomly picked chains as a single operation. This is done in effort to emulate "swapping" between classes. We have not done extensive testing to whether this provides a noticeable benefit in algorithm performance.

4.7 Cooling schedule

In our simulated annealing search, we have used the Lundy and Mees (1986) cooling schedule, shown in Eq. 8, where t refers to the current temperature, t' refers to the next temperature, and β is an empirically defined constant.

$$t' = \frac{t}{1 + \beta t} \tag{8}$$

We have used $\beta = 6 * 10^{-3}$ when the solution has zero hard penalty, otherwise we use $\beta = 3 * 10^{-3}$. In the first iteration, the initial temperature is set to $t = 10^{-3}$.

The search often reaches a stagnation point, where the solver does not move forward in the direction of finding a feasible solution. Hence, on time-outs (i.e., solver does certain number of iterations without improvement), we penalize variables and restart from a slightly increased temperature to allow for some perturbation. These mechanisms are described in the following sections.

4.8 Restart strategy

We define a search cycle as the period between the start or restart of the search up until there are no more improvements in the local solution. We refer to the *local best* solution as the best score found so far in the cycle, starting from infinity. We conclude that a cycle has timed out when there have passed $max_{timeout}$ consecutive iterations where neighborhood operators have been applied and the solution does not have a lower search penalty (as defined in Eq. 9) compared to the local best solution of that cycle.

After a cycle has timed out, we penalize features of the current solution and restart the search from a restart temperature of $t_{restart} = 10^{-4}$. However, during cycle restarts, if the solution is infeasible and we detect that some constraints are unsatisfied for a repeat number of cycles, we instead enter focused search (described in Sect. 4.10) and skip penalization. We discuss how penalization is done next.

4.9 Penalization

When the restart condition is fulfilled, we penalize combinations with the most hard penalty and gradually force the current solution to change. The penalization combined with the higher temperature after a restart pressures variables to new configurations in hopes of escaping from the stagnation point.

The modified evaluation function in Eq. 9 considers the penalization of assignments (features) in the current solution. An assignment is a class-time or class-room combination present in the solution. For example, if class 3 is assigned to room 8, we have the assignment (3, 8). The sum in Eq. 9 means we look up the penalty table for all assignments in the solution and add them together. The *modifiedPenalty* function is what is actually used during the main search. Initially all penalties are set to 0 (line 3 in Algorithm 2). When there are no penalties, then *modifiedPenalty* = *searchPenalty* because the second component of Eq. 9 is 0.

$$\text{modifiedPenalty}(s, \textit{penalties}) = \text{searchPenalty}(s) + \sum_{x}^{\textit{assignments}_s} \textit{penalties}_x \tag{9}$$

For each class we store 2 arrays of decimal numbers, one for room assignments and one for time assignments. These are what we call the penalty table or simply *penalties*. The index in these arrays represents the element in the variable's domain, and the value represents the penalty assigned to that variable if it is currently assigned to that element of the domain. This table (initially all zeroes) persists during the search and is updated according to the following description.

For each restart cycle we do the following. If the solution is not feasible (i.e., it has $p_h > 0$), active assignments (that are part of the solution) that are in a conflict are penalized according to Eq. 10. A time/room assignment (hereinafter variable) is in conflict if:

- The class of the variable conflicts with another class
- The variable is a time variable, and its class is part of an unfulfilled hard constraint which has to do with time distribution of classes
- The variable is a room variable, and its class is part of an unfulfilled hard constraint which has to do with room distribution of classes
- The class of the variable is part of an unfulfilled hard constraint which has to do with either time or room distribution of classes

The following are considered time constraints: DifferentDays, DifferentTime, DifferentWeeks, MaxBlock, MaxBreaks, MaxDayLoad, MaxDays, MinGap, NotOverlap, Overlap, Precedence.

The following are considered room constraints: SameRoom, DifferentRoom.

The following are considered as common (time or room) constraints: SameAttendees.

In Eq. 10, p is the current penalty, $rate_p$ is a constant with value of 1.1, *conflicts* is the number of conflicts the variable is part of, and $flat_p$ is a constant with value of 0.004. Regardless of the feasibility of the solution, we decay the penalties of variables in non active indexes (values that are currently not assigned) according to the formula $p' = penalty_{decay} * p$, where we have used $penalty_{decay} = 0.9$. This allows the variable to be used again after some time by assuming that by then the solver has found a way out of the local optimum caused by that variable. In the case of a new best solution being found, all variable penalties are reset to 0.

$$p' = p * rate_p + conflicts * flat_p \tag{10}$$

For a summarized description of the parameters and their values, the reader is referred to Table 2.

4.10 Random walks

The penalization mechanism has worked well for conflicts between a small number of variables, such as unavailable rooms or conflicts between classes. Some problems have constraints that span over dozens of classes. For such constraints, this style of penalization has not been able to guide the solution to a state where those constraints are satisfied.

During restarts, we keep count of unsatisfied required constraints. From this list we take up to $max_{constraints} = 3$ required constraints that have not been satisfied for 3 or more consecutive restart cycles. If such constraints exist, we enter a random walk search phase which is focused on those constraints.

This search has the goal of pivoting the solution to a new neighborhood where those constraints are satisfied. During this phase, Eq. 7 in Sect. 4.5.5 is used for evaluation.

The second component represents the sum of penalties of the focused required constraints. With the second component being an integer value and f_{stun} being scaled to a maximum of 1, it means that the former dominates the evaluation function and solutions which have satisfied the focused constraints are always given priority.

The idea behind this is to solve the difficult and large constraints. To keep the solution from changing drastically and being damaged, we use a simple hill climbing algorithm. The neighborhood operator for this hill climbing is a random walk that begins from the current solution. During the random walk, up to *distance* (random uniform number between 1..*distance*) random time or room variable mutations are generated and accumulated in a hill climbing fashion, meaning a mutation in the chain is accepted only if the new solution is evaluated to be equal to or better than the current solution according to the function in Eq. 7. Based on some preliminary experimentation, we have picked the value *distance=50*. A higher value can satisfy the focused constraints more quickly, but can also cause more damage to the overall solution quality. A lower value risks not exploring enough combinations to satisfy some of the larger constraints.

Two conditions may stop the search: the solution becoming feasible ($hardPenalty = 0$), or $n_{timeout} = 500{,}000$ cycles having passed without improvement of the modified evaluated function. A summary of this search is given as pseudocode in Algorithm 1.

Algorithm 1 Focused search on particular constraints

```
1: procedure ConstraintSearch(solution,
       focused constraints)
2:   timeout ← 0
3:   while timeout < timeout limit do
4:     candidate ← RandomWalk(solution, distance)
5:     if focusedPenalty(candidate) <
         focusedPenalty(solution) then
6:       solution ← candidate
7:       timeout ← 0
8:     else
9:       timeout ← timeout + 1
10:    end if
11:  end while
12:  return solution
13: end procedure
```

The returned solution will usually have a hard penalty that is similar to the solution before the search, but their structural nature will be different. We have found that when creating

such disturbances in the structure of a solution, new ways to get out of the stagnation point are opened.

After exiting the constraint search, regular search continues from the restart temperature. Penalization and constraint search are mutually exclusive, meaning that after a cycle only one of these can occur.

4.11 Simulated annealing

The simulated annealing (SA) algorithm relies in a search strategy where a current solution c is mutated (denoted as c'), based on the predefined operators, and accepted if:

- It has a better score (or lower penalty) according to Eq. 9 compared to the current solution.
- It has a worse score, but the acceptance condition in Eq. 11 is fulfilled.

In our search, we also accept a candidate solution if it has an equal score with the current one. As we'll see later, this allows for more exploration when the solution is infeasible.

The idea behind the random acceptance condition is to allow for some acceptance of worse solutions during earlier phases of the search process. In Eq. 11, $\Delta E(s', s)$ is defined in Eq. 6, where f_{search} is the modifiedPenalty defined in Eq. 9. The variable t represents the temperature value, which changes after every cycle based on the cooling function/schedule proposed by Lundy and Mees (1986) defined using Eq. 8. A cooling function slowly lowers the temperature value based on a decreasing curve. As the value of t gets lower, the chance of accepting worse solutions decreases.

$$\text{rand}(0, 1) \leq e^{\frac{-\Delta E(s', s)}{t}} \tag{11}$$

In Algorithm 2, we show a summary of the simulated annealing search in pseudocode form.

Algorithm 2 Simulated annealing with penalization

```
1: procedure SOLVE(initial solution)
2:   t ← initial temperature
3:   penalties ← initial penalties
4:   best ← initial solution
5:   local best ← ∞
6:   local timeout ← 0
7:   current ← best
8:   while stopping criteria not met do
9:     t ← cool(t)
10:    candidate ← mutate(current)
11:    if candidate better than best then
12:      best ← candidate
13:    end if
14:    if searchPenalty(candidate) < local best then
15:      local best ← searchPenalty(candidate)
16:      local timeout ← 0
17:    else
18:      local timeout ← local timeout + 1
19:    end if
20:    if modifiedPenalty(candidate) <
       modifiedPenalty(current) then
21:      current ← candidate
22:    else if accept(current, candidate, t) then
23:      current ← candidate
24:    end if
25:    if local timeout > max timeout then
26:      local best ← ∞
27:      local timeout ← 0
28:      t ← restart temperature
29:      persistent constraints ←
       (constraints with age > age limit)
30:      if infeasible(current)∧
                         persistent constraints ≠ ∅ then
31:        focused constraints ←
                         oldest 3 persistent constraints
32:        current ←
               ConstraintSearch(current, focused constraints)
33:      else
34:        penalties ← scale(penalties)
35:      end if
36:    end if
37:  end while
38:  return best
39: end procedure
```

5 Experimental analysis

In this section, we report the computational experiments of the algorithm by presenting parameter settings, search method analysis and comparison to the existing approaches for the University Timetabling Problem. The code was written in the F# functional programming language and it was compiled based on the.NET Core code execution process by using the F# 4.1 compiler. All the experiments have been conducted in a machine with 1x Server System Building Block PROnex 1029GQ-TRT, 2x Intel Xeon SKL-EP Gold 6130 2.10GHz 16C 22MB Cache, 12x 8GB DDR4-2666MHz ECC RAM, 2x 2TB SATA 6Gbp/s 2.5" HDD and 1x AMD Radeon Instinct MI25. Further, all the experiments have been done by running the instances ten times per each configuration in 24-hour duration. The presented results have been validated against the web validator of ITC2019 competition. The source code of the solver can be found in GitHub[7]

5.1 Test set

The test set consists of thirty instances that are derived from real-word scenarios in ten different universities around the world that include Purdue University, Masaryk University, AGH University of Science and Technology, Lahore University of Management Sciences, Istanbul Kültür Uni-

[7] https://github.com/edongashi/itc-2019.

Table 1 Test set characteristics

#	Instance name	Courses	Classes	Rooms	Hard Dist	Soft Dist	Hard Class Pairs	Soft Class Pairs	Students	Weeks
1	agh-fis-spr17	340	1239	80	820	400	3521	769	1641	16
2	agh-ggis-spr17	272	1852	44	2202	488	29,475	2046	2116	16
3	bet-fal17	353	983	62	861	390	5773	6564	3018	16
4	iku-fal17	1206	2641	214	2237	665	26,768	1117	0	14
5	mary-spr17	544	882	90	3151	796	40,242	14,903	3666	16
6	muni-fi-spr16	228	575	35	645	95	2113	1038	1543	15
7	muni-fsps-spr17	226	561	44	331	69	2284	851	865	19
8	muni-pdf-spr16c	1089	2526	70	1456	570	12,574	3214	2938	13
9	pu-llr-spr17	687	1001	75	416	218	891	622	27,018	16
10	tg-fal17	36	711	15	459	42	79,299	17,090	0	14
11	agh-ggos-spr17	406	1144	84	1181	507	6745	828	2254	16
12	agh-h-spr17	234	460	39	288	111	2366	1247	1988	16
13	lums-spr18	313	487	73	449	69	2942	651	0	20
14	muni-fi-spr17	186	516	35	639	60	1614	957	1469	14
15	muni-fsps-spr17c	116	650	29	562	147	6952	397	395	14
16	muni-pdf-spr16	881	1515	83	579	433	6836	2723	3443	13
17	nbi-spr18	404	782	67	585	11	2131	29	2293	15
18	pu-d5-spr17	212	1061	84	1262	273	2734	6268	13,497	15
19	pu-proj-fal19	2839	8813	768	6399	1398	18,171	32,031	38,437	17
20	yach-fal17	91	417	28	529	116	1977	545	821	16
21	agh-fal17	1363	5081	327	4836	2318	45,193	13,596	6925	18
22	bet-spr18	357	1083	63	1004	414	7652	8111	2921	16
23	iku-spr18	1290	2782	208	2833	655	32,505	779	0	13
24	lums-fal17	328	502	73	521	76	3938	670	0	20
25	mary-fal18	540	951	93	349	164	922	1418	5051	16
26	muni-fi-fal17	188	535	36	635	152	1805	1019	1685	13
27	muni-fspsx-fal17	515	1623	33	1070	289	17,542	2186	1152	21
28	muni-pdfx-fal17	1635	3717	86	2433	1068	42,650	7446	5651	13
29	pu-d9-fal19	1154	2798	224	2039	707	4368	5966	35,213	15
30	tg-spr18	44	676	18	376	50	72,444	1986	0	16

versity, Bethlehem University, Universidad Yachay Tech, Turkish-German University, University of Nairobi and Maryville University. The instances have been taken from the UniTime[8] timetabling module. The size and complexity of instances is quite diverse, with classes from 500 up to 8800, with student from 2000 to 38,000 and with rooms from 50 to 770. In some instances, it is the problem complexity that is more critical than its size. In addition, the instances based on European universities are simpler in terms of classes, since they take place once a week, whereas the instances from USA universities possess classes that are taught several times a week. During the competition, the data set was divided into three chunks that were called early, middle and late instances, where the last chunk of instances is more difficult to solve, both in terms of feasibility and soft constraint optimization. Table 1 presents a summary of the features of individual instances in the test set, where the first ten belong to the early set, the second ten to the middle set and the last ten to the late set. For a more detailed test set description about the complexity and the diversity of individual test instances, the reader is referred to the official web set of ITC2019[9] competition.

5.2 Search method analysis

All parameters are determined empirically by running all instances and verifying that the solver found a valid solution for all of them. The competition rules state that parameters/constants are not allowed to be tuned for specific instances and the solver is not allowed to recognize a partic-

[8] https://www.unitime.org/.

[9] https://www.itc2019.org/.

Table 2 Parameter settings

Parameter name	Value	Comment
MaxTimeout	10^6	Value of $max_{timeout}$ in Section 4.8 and in Algorithm 2
RollingEffect	True	Whether to accept solutions with equal quality during local search
FStunGamma	0.95	Parameter γ in Eq. 5
FocusedSearchEnabled	True	Whether to enable the random walk phase in Section 4.10
FocusedSearchMaxConstraints	3	Maximum number of unsatisfied constraints to focus on. Parameter $max_{constraints}$ in Section 4.10
FocusedSearchMinWeight	4	Number of consecutive restart cycles that a constraint needs to be unsatisfied to be included in focused search
FocusedSearchTimeoutMax	$5 * 10^5$	Number of cycles without improvement to exit focused search. Parameter n_timeout in section 4.10
FocusedSearchDistance	50	Maximum number of consequent mutations during random walk. Parameter distance in Algorithm 1
InfeasibleTimeMutationOccurrences	1	Distribution of probabilities for mutations when solution is infeasible. Possible mutations are added to a list n times (parameter value). A mutation is picked by choosing a random item from the whole list. For example, if we have 2 time mutation occurrences and 3 room mutation occurrences, the chance for a time mutation being picked is 2/5=40%. A "variable" means either room or time picked at random from the list of all variables.
InfeasibleRoomMutationOccurrences	1	
InfeasibleVariableMutationOccurrences	2	
InfeasibleEnrollmentMutationOccurrences	0	
InfeasibleDoubleEnrollmentMutationOccurrences	2	
FeasibleTimeMutationOccurrences	1	Distribution of probabilities for mutations when solution is feasible. Operations are the same as in the infeasible set.
FeasibleRoomMutationOccurrences	1	
FeasibleVariableMutationOccurrences	2	
FeasibleEnrollmentMutationOccurrences	2	
FeasibleDoubleEnrollmentMutationOccurrences	1	
TemperatureInitial	10^{-2}	Initial temperature in Algorithm 2
TemperatureRestart	10^{-4}	Restart temperature in Algorithm 2
TemperatureBeta	$6 * 10^{-3}$	Value of β in Section 4.7 when solution is feasible
TemperatureBetaUnfeasible	$3 * 10^{-3}$	Value of β in Section 4.7 when solution is infeasible
InfeasibleSolutionHardPenaltyFactor	10^{-2}	Parameter c_1 in Eq. 3
HardPenalizationRate	1.1	Determines the factor by which existing penalties are increased. Parameter $flat_p$ in Eq. 10
HardPenalizationFlat	$4 * 10^{-3}$	Parameter *flat_p* in Eq. 10
HardPenalizationDecay	0.9	Rate at which penalties of non-active assignments are reduced. Discussed in Section 4.9
SoftPenalizationRate	1.1	Same as HardPenalizationFlat but used when the solution is feasible
SoftPenalizationFlat	10^{-3}	Same as HardPenalizationFlat but used when the solution is feasible
SoftPenalizationDecayRate	0.9	Same as HardPenalizationDecay but used when the solution is feasible
SoftPenalizationDecayFlat	10^{-3}	Fixed value subtracted to penalties of non-active assignment. Only applicable to feasible solutions

ular problem instance. To adhere to this requirement, if the solver failed to solve some particular instance, we refined the algorithm to make it more general. After we made the algorithm general enough to be able to solve all instances, we further tuned parameters related to soft penalty optimization and setting values which gave best overall results across all instances.

Table 2 shows a summary of all parameters, their values used for the competition (all instances), and the explanation behind them. The names used in this table correspond to the parameter names in code. In the comment column, we reference the section or equation where they are described within the paper.

Further, we examine solution evolution for problem instances of varying complexity. The final outcome is represented by a graphic of each of the instances.

Figure 2 shows 3 runs of the muni-fsps-spr17 instance. The graphics on the left show hard penalty evolution, whereas the graphics on the right show the soft penalty for that particular run (one run per row) against the total runtime. Note that the time scale is different for these graphics because feasible solutions are often obtained rapidly and the vast majority of the time is invested optimizing the soft penalty of the problem. For runs 1 and 3, the hard penalty line is barely visible owing to the fact that it drops to zero within a few seconds of starting the solver. For run 2 it took around 10 min to reach a feasible solution. The infeasible search phase is clearly visible in the soft penalty graphic of run 2, because soft penalty starts dropping only after hard penalty reaches 0. While the solution is infeasible, soft penalty varies wildly because it is ignored and the solver rolls over on equal hard penalty (accepts a solution with less or equal hard penalty). As we discussed in earlier sections, attempting to optimize soft penalty while searching for feasible solutions slowed the solver dramatically without bringing any notable improvements.

Figure 3 shows 3 runs of the pu-d5-spr17 instance, which is much more difficult to solve. For the time we sampled the solver (around 3 h), only run 2 managed to find a feasible solution at roughly 1.5 h. In the soft penalty graphics of run 1 and run 3 we can see a large variation between soft penalty.

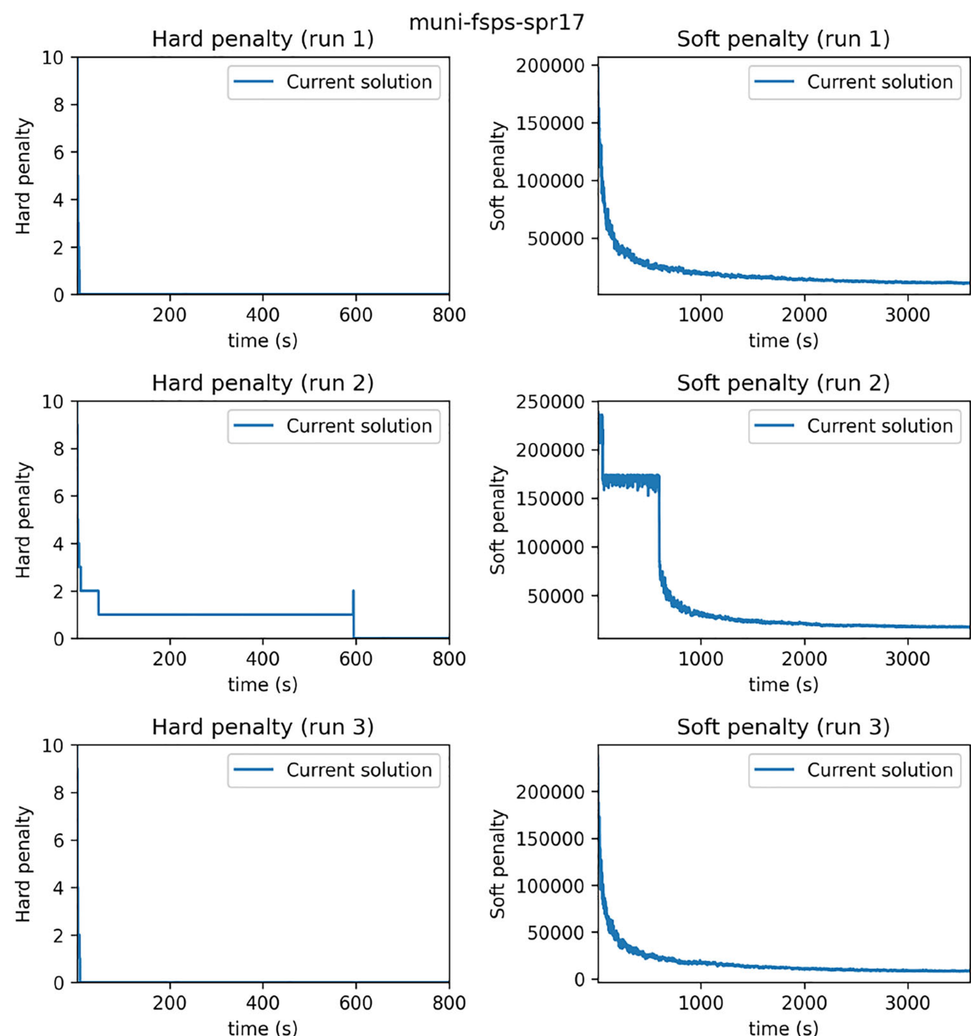

Fig. 2 Algorithm performance for instance ’muni-fsps-spr17’

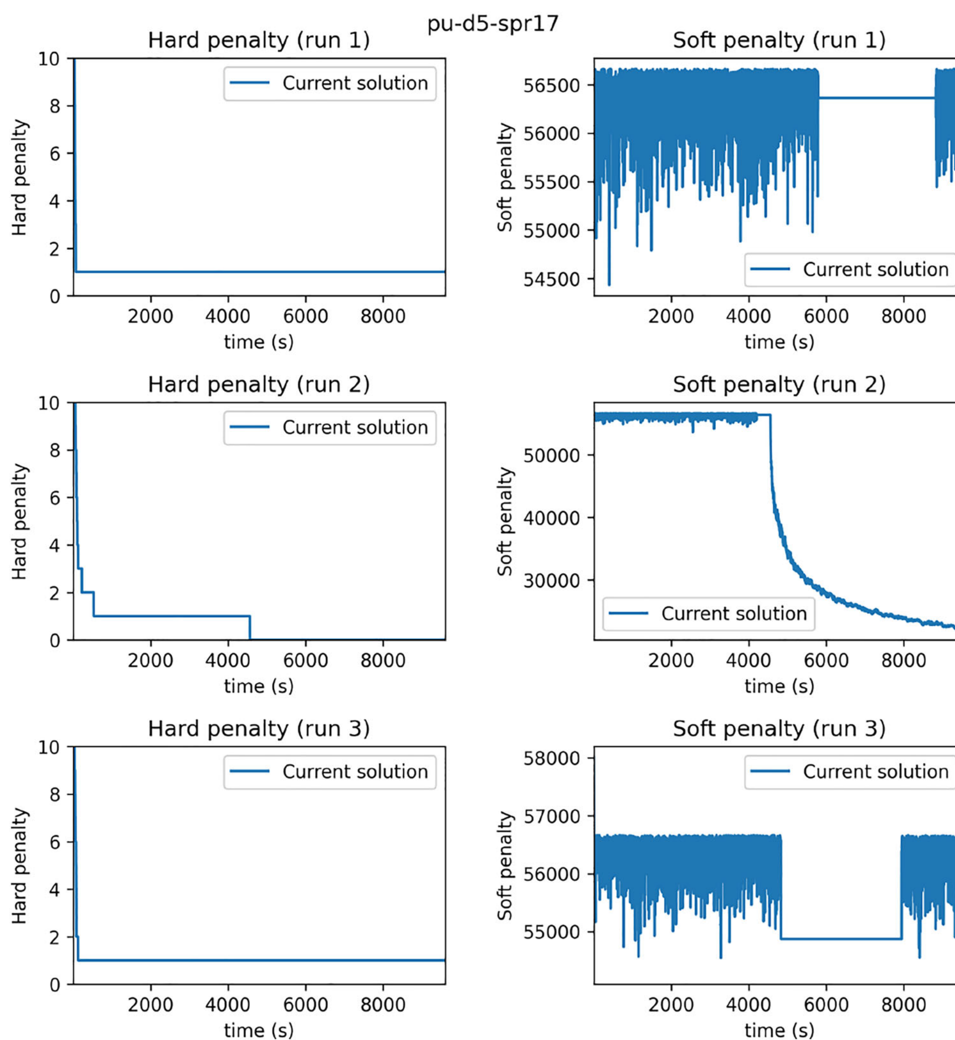

Fig. 3 Algorithm performance for instance 'pu-d5-spr17'

The flat line represents the random walk phase, for which we did not trace soft penalty. After the feasible solution is found in run 2, the soft penalty decreases in a curve similar to that of Fig. 2. The short flattening of soft penalty in run 2 before a feasible solution is found indicates that the feasible solution was found during the random walk phase.

In Fig. 4 we show 3 runs of agh-h-spr17 instance, which is moderately difficult. Runs 2 and 3 are similar in runtime and score. In all three cases a feasible solution was found during random walks. Run 1 solved it during the second cycle of random walks, whereas runs 2 and 3 found a feasible solutions on the first cycle at around 20 min. Nevertheless, final soft penalty is consistent for all runs and for most instances where feasible solutions are found.

In Tables 3 and 4, we present computational experiments that we have obtained in ITC2019 competition when running the solver with all algorithm components (in both tables tagged as 'Best SAP') and within 24 h limit (third and fourth column in Table 3 tagged as 'Default configuration'). Further, in the remaining columns in both tables, we show the results of the algorithm when running it without utilizing specific components. The solver has been executed 10 times, each with a duration of 24 h, for each configuration (i.e., without using a specific algorithm component) and for each instance it the dataset. Table 3 shows that the FStun function (i.e., stochastic tunneling) and the penalization strategy are both crucial for the proposed approach, since if either of them is not applied than none of the instances can be solved to feasibility ('x' indicates no feasible solution can be generated). Further, Table 3 shows that the rolling effect is also essential for the algorithm, especially for the instances belonging to the late test set of the ICT2019 competition, because no feasible solution can be generated if the rolling effect is not employed. In general, the 'focused search' component seems to be important in terms of solution quality when it comes to soft constraint optimization, because, if this component is not used, the quality degrades quite a lot for most of the instances, even though for some specific instances, like pu-d5-spr17, bet-spr18 and ums-fal17, new better solutions are found when this exact component is not applied.

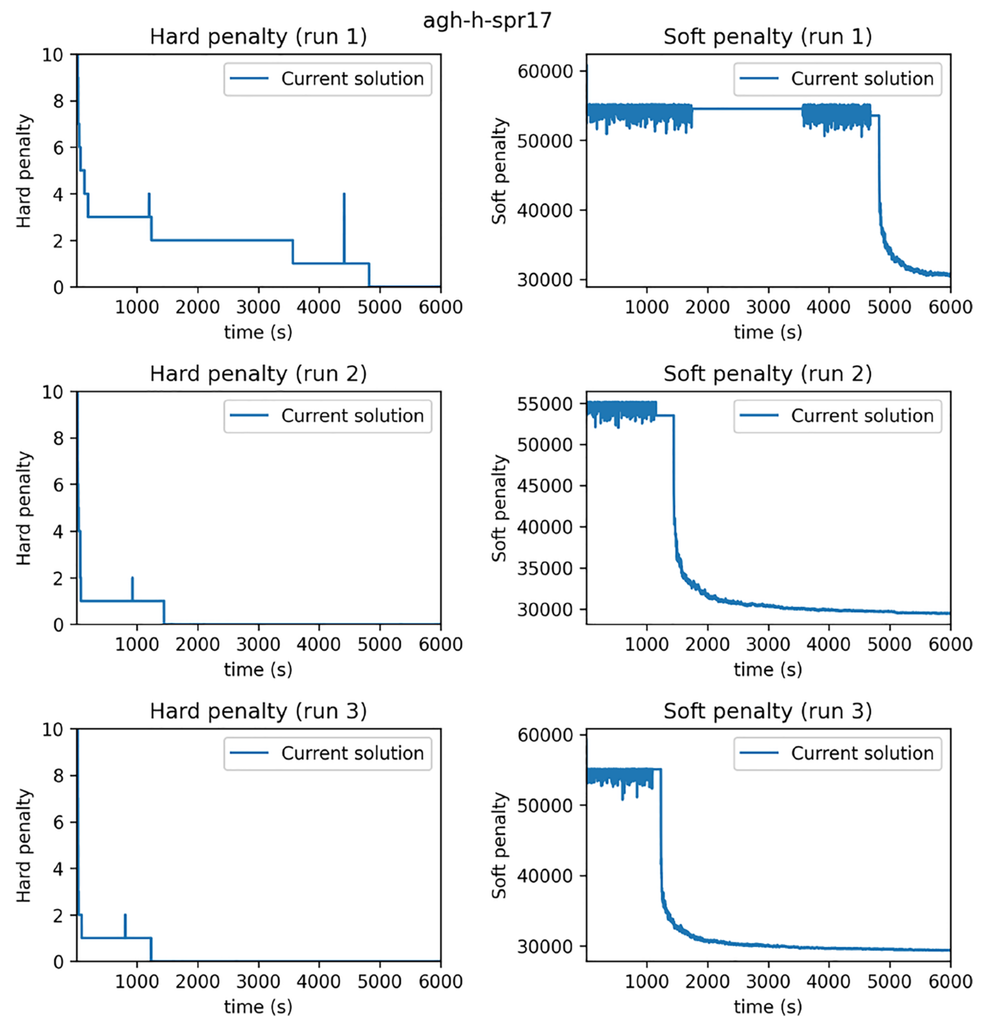

Fig. 4 Algorithm performance for instance 'agh-h-spr17'

We have also experimented with running the algorithm without applying specific neighborhood operators, namely room mutation, time slot mutation and variable mutation, for the phases when the algorithm is trying to generate a feasible solution (tagged as infeasible search period) and when the algorithm is looking to optimize the soft constraints (tagged as feasible search period). These results are presented in the last two columns of Table 3 and all columns, except the first two, of Table 4. When the results are averaged over all instances having been generated a feasible solution for a specific algorithm configuration, the degradation is the lowest, standing at 17.2%, if the operator for time slot mutation is not applied in the feasible search period, whereas the degradation is the largest, specifically 37.8%, if the room mutation operator is not applied in the infeasible search period.

Finally, it is important to note down that for four specific instances, namely agh-ggis-spr17, pu-proj-fal19, muni-pdfx-fal17 and tg-spr18, if at least one of the above-mentioned components of the algorithm is not applied, then no feasible solution can be generated.

5.3 Comparison results

In Table 5, we present the comparison results of our approach against the best known results (as of September 2020) of four of the other finalists, namely Holm et al. (2020), Efstratios et al. (2021), Er-rhaimini (2021) and Lemos et al. (2021). In addition, we have also included the results presented by Müller (one of the organizers of the competition), who, after the end of the competition, has published the results obtained by the solver that is used by UniTime timetabling system. The presented results of our approach (tagged as Sylejmani et al.) are the best results that have been achieved when running the solver for each instance.

The results in Table 5 show that our approach is outperformed, in all of the instances, by the the approaches of Holm et al. (2020) and Müller, whereas Efstratios et al. (2021) performs better in 23 (out of 30) instances. Our approach performs better than the approach of Efstratios et al. (2021) in seven instances, better than the approach Er-rahaimini Er-rhaimini (2021) in 19 instances, and better than the approach

Table 3 Analysis of algorithm components (Part 1)

Instance name	Best SAP	Default configuration (%)		No focused search (%)		No FStun Gamma (%)		No penalization (%)		No rolling effect (%)		No room mutation (feasible search period) (%)	
		Average	Best	Average	Best	Average	Best	Average	Best	Average	Best	Average	Best
agh-fis-spr17	6799	5.29	5.9	20.19	19.74	x	x	x	x	13.43	13.38	4.33	4.15
agh-ggis-spr17	77,932	x	x	x	x	x	x	x	x	x	x	x	x
bet-fal17	299,205	x	x	4.98	3.17	x	x	x	x	2.02	1.33	1.32	1.25
iku-fal17	50,613	22.7	26.25	23.5	23.45	x	x	x	x	26.12	25.53	19.91	19.84
mary-spr17	15,894	10.39	25.3	28.45	26.26	x	x	x	x	14.08	13.63	24.22	23.76
muni-fi-spr16	5006	14.34	21.06	11.03	6.03	x	x	x	x	16.11	14.7	10.11	8.03
muni-fsps-spr17	1938	134.57	365.82	486.86	384.98	x	x	x	x	95.63	95.36	29.67	29.67
muni-pdf-spr16c	58,206	35.9	37.08	32.49	23.12	x	x	x	x	27.84	27.82	16.47	16.46
pu-llr-spr17	16,874	30.66	32.74	25.85	23.28	x	x	x	x	30.49	30.42	26.79	26.46
tg-fal17	8044	x	x	x	x	x	x	x	x	x	x	x	x
agh-ggos-spr17	9328	58.75	74.27	x	72.32	x	x	x	x	64.77	64.74	42.84	42.52
agh-h-spr17	25,081	59.56	22.63	17.67	12.14	x	x	x	x	8.51	8.07	9.77	9.69
lums-spr18	107	9.35	11.64	10.9	9.35	x	x	x	x	26.17	12.15	27.1	12.15
muni-fi-spr17	4692	2.17	3.84	7.21	1.68	x	x	x	x	6.09	5.88	4.25	3.96
muni-fsps-spr17c	9222	69.56	74.94	84.01	63	x	x	x	x	69.54	67.47	60.7	60.47
muni-pdf-spr16	40,074	3.62	7.83	4.04	0.54	x	x	x	x	5.9	5.62	−0.21	−0.25
nbi-spr18	26,517	16.60	17.24	17.61	15.75	x	x	x	x	12.05	10.29	15.56	15.51
pu-d5-spr17	19,440	2.89	4.82	0.21	−4.5	x	x	x	x	8.22	0.59	2.45	2.26
pu-proj-fal19	237,909	x	x	x	x	x	x	x	x	x	x	x	x
yach-fal17	1727	6.2	10.36	21.28	16.27	x	x	x	x	20.63	19.63	31.57	29.36
agh-fal17	184,030	x	x	x	x	x	x	x	x	x	x	40.84	40.38
bet-spr18	360,437	18.42	6.57	0	−0.01	x	x	x	x	x	x	0.94	0.93
iku-spr18	85,969	x	x	x	6.71	x	x	x	x	x	x	x	x
lums-fal17	486	6.58	7.77	4.73	−0.41	x	x	x	x	x	x	20.06	5.35
mary-fal18	7199	31.77	33.31	36.05	32.88	x	x	x	x	x	x	18.38	15.47
muni-fi-fal17	4712	17.57	28.93	15.19	9.02	x	x	x	x	x	x	12.81	12.37
muni-fspsx-fal17	41,933	17.53	35.78	13.33	3.94	x	x	x	x	x	x	30.19	30.02
muni-pdfx-fal17	159,203	x	x	x	x	x	x	x	x	x	x	x	x
pu-d9-fal19	82,757	67.81	91.99	x	72.56	x	x	x	x	x	x	98.43	98.2
tg-spr18	15,992	x	x	x	x	x	x	x	x	x	x	x	x

Table 4 Analysis of algorithm components (Part 2)

Instance name	Best SAP	No room mutation (infeasible search period) (%)		No time slot mutation (feasible search period) (%)		No time slot mutation (infeasible search period) (%)		No variable mutation (feasible search period) (%)		No variable mutation (infeasible search period) (%)	
		Average	Best	Average	Best	Average	Best	Average	Best	Average	Best
agh-fis-spr17	6799	4.62	1.72	1.68	0.65	10.52	8.9	17.36	15.87	28.48	28.28
agh-ggis-spr17	77,932	x	x	x	x	x	x	x	x	x	x
bet-fal17	299,205	2.4	2.4	7.43	7.33	1.27	1.2	1.44	1.42	1.24	1.23
iku-fal17	50,613	28.12	28.01	27.08	26.89	26.68	26.49	28.38	28.27	28.02	27.93
mary-spr17	15,894	14.76	14.76	27.81	27.55	32.05	31.81	26.56	26.21	24.04	23.85
muni-fi-spr16	5006	15.77	15.08	25.5	24.91	9.71	8.47	16.37	14.98	17.41	17.18
muni-fsps-spr17	1938	392.17	391.54	50.16	20.38	88.12	87.15	90.17	88.39	240.31	238.44
muni-pdf-spr16c	58,206	23.04	23.01	20.46	20.32	34.4	33.97	22.65	22.47	22.52	22.47
pu-llr-spr17	16,874	26.33	25.84	34.02	33.4	19.14	18.61	22.19	21.51	22.15	21.51
tg-fal17	8044	31.18	30.31	33.01	32.74	x	x	x	x	x	x
agh-ggos-spr17	9328	54.54	54.05	51.29	51.1	67.46	66.18	30.11	28.4	29.75	28.4
agh-h-spr17	25,081	8.91	8.7	10.19	9.87	18.44	18.1	14.89	14.57	14.88	14.57
lums-spr18	107	14.25	7.48	17.57	7.48	31.92	14.95	24.81	9.35	13.29	9.35
muni-fi-spr17	4692	3.28	0.64	12.01	9.53	2.99	1.49	11.32	9.1	13.49	9.1
muni-fsps-spr17c	9222	51.54	50.6	51.34	50.62	58.53	58.2	23.83	23.31	23.87	23.31
muni-pdf-spr16	40,074	8.6	8.47	4.25	4.02	−2.7	−3.1	−89.07	7.75	8.01	7.75
nbi-spr18	26,517	10.02	9.85	16.23	16.07	8.11	7.52	19.83	18.98	19.45	18.98
pu-d5-spr17	19,440	2.06	0.63	9.82	1.17	20.99	3.87	0.26	−1.99	15.25	−1.99
pu-proj-fal19	237,909	x	x	x	x	x	x	x	x	x	x
yach-fal17	1727	23.58	15.69	15.47	14.24	25.98	17.83	28.21	24.15	29.34	24.15
agh-fal17	184,030	x	x	x	x	x	x	x	x	x	x
bet-spr18	360,437	0.42	0.35	−0.7	−0.74	x	x	−0.14	−0.2	0.5	−0.2
iku-spr18	85,969	10.67	10.28	10.43	8.24	x	x	x	x	x	x
lums-fal17	486	13.63	−0.21	12.08	3.7	10.11	3.91	13.63	7	10.42	7
mary-fal18	7199	33.81	29.92	−92.61	-93	34.94	34.52	15.53	14.97	15.48	14.97
muni-fi-fal17	4712	12.8	10.04	14.07	9.97	2.2	−3.27	14.08	11.38	15.14	11.38
muni-fspsx-fal17	41,933	7.88	7.33	2.41	1.5	30.04	27.6	11.5	5.69	7.58	5.69
muni-pdfx-fal17	159,203	x	x	x	x	x	x	x	x	x	x
pu-d9-fal19	82,757	150.79	150.51	69.09	68.47	49.2	48.64	43.53	38.14	38.84	38.14
tg-spr18	15,992	x	x	x	x	x	x	x	x	x	x

Table 5 Presentation of the gap (in percentage) from the best known results

Instance name	Best	Holm et al. (%)	Müller (%)	Rappos et al. (%)	Sylejmani et al. (%)	Er-rhaimini (%)	Lemos et al. (%)
agh-fis-spr17	3039	0	12.2	49.9	123.7	87.8	N/A
agh-ggis-spr17	34,285	0	6.2	6.7	127.3	65.5	N/A
bet-fal17	289,965	0	0.2	1.8	3.1	N/A	2.1
iku-fal17	18,968	0	24.5	41.5	166.8	134.5	57.7
mary-spr17	14,910	0	1.4	0.7	6.5	11.9	N/A
muni-fi-spr16	3756	0	7.5	2.3	33.2	38.6	N/A
muni-fsps-spr17	868	0	1.2	1.7	123.2	376.3	N/A
muni-pdf-spr16c	33,724	0	18.3	11.1	72.5	130	59.5
pu-llr-spr17	10,038	0	7.5	33.3	68.1	91.5	N/A
tg-fal17	4215	0	0	0	90.8	74.5	60.7
agh-ggos-spr17	2864	0	19.3	120.6	225.6	169.7	2684.3
agh-h-spr17	22,175	0.05	0	17.9	13.1	16.1	N/A
lums-spr18	95	0	3.1	20	12.6	87.3	475.7
muni-fi-spr17	3825	0	3.6	12.1	22.6	42	372.6
muni-fsps-spr17c	2596	0	16	27.2	255	806	23714.2
muni-pdf-spr16	17,208	0	16.8	41.3	132.8	125.6	1707.2
nbi-spr18	18,014	0	3.7	5.7	47.2	68.2	177.1
pu-d5-spr17	15,204	4.6	0	23.7	27.8	33.1	3.4
pu-proj-fal19	117,425	25.7	0	377.9	102.6	49.9	9.32
yach-fal17	1074	15.3	0	71.6	60.8	196.1	N/A
agh-fal17	118,038	24.5	0	N/A	20.8	55.9	29.8
bet-spr18	348,524	0	0.2	3.3	1.5	3.4	7
iku-spr18	25,868	0	39.1	41.9	76	232.3	174.2
lums-fal17	349	0	5.7	10.6	132.9	39.2	59.8
mary-fal18	4422	0	8.8	27.5	897.2	62.8	57
muni-fi-fal17	2999	0	8.2	26.5	38.7	57.1	60.7
muni-fspsx-fal17	10,123	68.3	0	226	900.8	314.2	933.5
muni-pdfx-fal17	98,373	13.7	0	53.9	53.9	61.8	95.1
pu-d9-fal19	39,942	0	11.9	235.5	19	107.2	76.3
tg-spr18	12,704	0	14.5	1.2	151.1	25.8	55

of Lemos et al. (2021) in 21 instances. In addition, the solutions of our solver have a gap of less than 15% from the best-known solutions in 5 (out of 30) instances. Furthermore, the average gap from the best known solution for early, middle and late chunks of instances is about 80%, 90% and 230%, respectively.

6 Conclusions and future work

In this paper, we have presented a Simulated Annealing approach based on several penalization components for the problem of University Timetabling as presented in ITC2019 competition. The presented penalization mechanisms have a noticeable impact in making it possible for our approach to produce feasible solutions and, in overall, make a better university schedule optimization.

It is evident that our algorithm performs well in comparison to the techniques that do not relay on commercial solvers, such as the approach presented by Er-rahaimini Er-rhaimini (2021). Nonetheless, the techniques based on MIP formulation, such as Holm et al. (2020) and Efstratios et al. (2021), are superior against our approach, although the latter not in all of the instances. Our approach performs better than the MaxSat approach of Lemos et al. (2021) in more than two thirds of instances. However, the UniTime solver, which is also open source, produces better results than our proposed technique for all the competition instances.

As part of our future work, we plan to develop additional local search operators for neighborhood exploration and hybridize our SA approach with population-based approaches such the Ant Colony Optimization (ACO). We aim to use our SA approach as an improving techniques within local search phase of ACO. In order to achieve this, we will further investigate the University Timetabling Problem with the intention of defining several alternative components/features that can be used within the ACO algorithm.

Acknowledgements The work on this paper was partially supported by the HERAS program within the project entitled "Automated Curriculum-based Course Timetabling in the University of Prishtina." Furthermore, we would like to thank the organizers of the ITC2019, Tomáš Müller, Hana Rudová, and Zuzana Müllerová for their collaboration with us in this matter. Finally, we thank Brikena Avdyli for her valuable proof-reading comments.

References

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

Bettinelli, A., Cacchiani, V., Roberti, R., & Toth, P. (2015). An overview of curriculum-based course timetabling. *Top, 23*(2), 313–349.

Bykov Y. (2003). The description of the algorithm for international timetabling competition. *International Timetable Competition*,

Cambazard, H., Hebrard, E., O'Sullivan, B., & Papadopoulos, A. (2012). Local search and constraint programming for the post enrolment-based course timetabling problem. *Annals of Operations Research, 194*(1), 111–135.

Ceschia, S., Di Gaspero, L., & Schaerf, A. (2012). Design, engineering, and experimental analysis of a simulated annealing approach to the post-enrolment course timetabling problem. *Computers & Operations Research, 39*(7), 1615–1624.

Cordeau1, R. M., Cordeau, J. -F., Jaumard, B., & Morales, R. (2003). Efficient timetabling solution with tabu search.

Di Gaspero, L. & Schaerf, A. (2003). Timetabling competition ttcomp 2002: solver description. *International Timetabling Competition*.

Di Gaspero, L., McCollum, B., & Schaerf, A. (2007). *The second international timetabling competition (itc-2007): Curriculum-based course timetabling (track 3)*. Citeseer: Technical report.

Eckersley, A. (2004). An investigation of case-based heuristic selection for university timetabling.

Efstratios, R., Eric, i., Robert, S., & Heche, J.-F. (2021). International timetabling competition 2019: A mixed integer programming approach for solving university timetabling problems.

Er-rhaimini, K. (2021). Forest growth optimization for solving timetabling problems.

Gashi, E., & Sylejmani, K. (2020). Simulated annealing with penalization for university course timetabling.

Geiger, M. J. (2012). Applying the threshold accepting metaheuristic to curriculum based course timetabling. *Annals of Operations Research, 194*(1), 189–202.

Goh, S. L., Kendall, G., & Sabar, N. R. (2019). Simulated annealing with improved reheating and learning for the post enrolment course timetabling problem. *Journal of the Operational Research Society, 70*(6), 873–888.

Gunawan, A., Ng, K. M., & Poh, K. L. (2012). A hybridized Lagrangian relaxation and simulated annealing method for the course timetabling problem. *Computers & Operations Research, 39*(12), 3074–3088.

Henderson, D., Jacobson, S.H., & Johnson, A.W. (2003). The theory and practice of simulated annealing. In *Handbook of metaheuristics*, (pp. 287–319). Springer.

Holm, D. S., Mikkelsen, R. Ø., Sørensen, M., & Stidsen, T. R. (2021). A mip based approach for international timetabling competation 2019.

Holm, D. Sø., Mikkelsen, R. Ø., Sørensen, M., & Stidsen, T. J. R. (2020). A mip formulation of the international timetabling competition 2019 problem.

Kalender, M., Kheiri, A., Özcan, E., & Burke, E. K. (2012). A greedy gradient-simulated annealing hyper-heuristic for a curriculum-based course timetabling problem. In *2012 12th UK workshop on computational intelligence (UKCI)*, (pp. 1–8). IEEE

Kirkpatrick, S. (1984). Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics, 34*(5–6), 975–986.

Kostuch, P. (2003). Timetabling competition-sa-based heuristic. *International Timetabling Competition*.

Lemos, A., Monteiro, P. T., & Lynce, I. (2021). Itc-2019: A maxsat approach to solve university timetabling problems.

Lim, A., Rodrigues, B., & Zhang, X. (2006). A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operational Research, 174*(3), 1459–1478.

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

Lundy, M., & Mees, A. (1986). Convergence of an annealing algorithm. *Mathematical Programming, 34*(1), 111–124.

Mall, R. (2018). *Fundamentals of software engineering*. Delhi: PHI Learning Pvt. Ltd.

Méndez-Díaz, I., Zabala, P., & Miranda-Bront, J. J. (2016). An ilp based heuristic for a generalization of the post-enrollment course timetabling problem. *Computers & Operations Research, 76,* 195–207.

Müller, T., Rudová, H., & Müllerová, Z. (2018). University course timetabling and international timetabling competition 2019. In *Proceedings of 12th international conference on the practice and theory of automated timetabling (PATAT)*, (p. 27).

Müller, T. (2009). Itc 2007 solver description: a hybrid approach. *Annals of Operations Research, 172*(1), 429.

Nadel, A. (2019). Anytime weighted maxsat with improved polarity selection and bit-vector optimization. In *2019 Formal methods in computer aided design (FMCAD)*, (pp. 193–202). IEEE.

Nagata, Y. (2018). Random partial neighborhood search for the post-enrollment course timetabling problem. *Computers & Operations Research,90,* 84–96.

Paechter, B., Gambardella, L. M., & Rossi-Doria, O. (2002). The first international timetabling competition. http://www.idsia.ch/Files/ttcomp2002,

Stern, J. M. (1992). Simulated annealing with a temperature dependent penalty function. *ORSA Journal on Computing, 4*(3), 311–319.

Wenzel, W., & Hamacher, K. (1999). Stochastic tunneling approach for global minimization of complex potential energy landscapes. *Physical Review Letters, 82*(15), 3003.

Zheng, S., Wang, L., Liu, Y., & Zhang, R. (2015). A simulated annealing algorithm for university course timetabling considering travelling distances. *International Journal of Computing Science and Mathematics, 6*(2), 139–151.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.