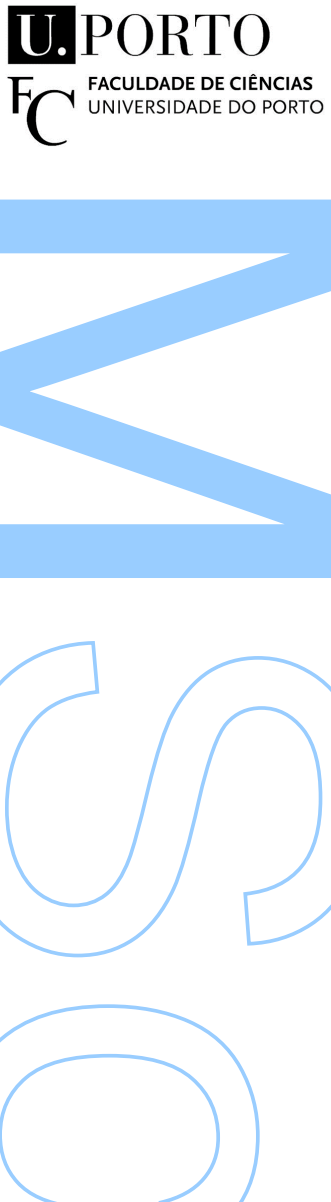

Algorithms for Examination Timetabling

Diogo Pinheiro de Almeida
Master in Network and Information Systems Engineering
Departament of Computer Science
Faculty of Sciences of the University of Porto
2025

Algorithms for Examination Timetabling

Diogo Pinheiro de Almeida

Dissertation carried out as part of the Master in Network and Information Systems Engineering

Departament of Computer Science

2025

Supervisor

Prof. João Pedro Pedroso, Associate Professor, Faculty of Sciences of the University of Porto

Co-supervisor

Prof. Ana Paula Tomás, Associate Professor, Faculty of Sciences of the University of Porto

Acknowledgements

I would like to express my gratitude to my supervisors Prof. João Pedroso and Prof. Ana Tomás, during the entire thesis. Their assistance and motivation helped in times of need, and if not for them, the development of this thesis would not be possible.

I would like to thank my best friends Guilherme, Xavier, Andrea, my parents Adérito and Eduarda and my grandparents Rosa and Vitor, for always helping when needed and being there to support me through the hardest moments of my life.

Resumo

A construção de horários é uma tarefa desafiante para as instituições de ensino, particularmente quando estão envolvidas instâncias de grande dimensão e restrições complexas. No entanto, muitas instituições ainda utilizam métodos de agendamento manuais que frequentemente resultam em horários ineficientes e são vulneráveis a erros humanos.

O objetivo principal desta tese é desenvolver e avaliar uma solução com Monte Carlo Tree Search para problemas de horários de exames. A abordagem proposta está concebida para lidar com múltiplos objetivos hierárquicos e é avaliada utilizando tanto as instâncias de benchmark da International Timetabling Competition Examination Track como dados reais provenientes da Faculdade de Ciências da Universidade do Porto. Como objetivo secundário, desenvolveu-se um modelo matemático baseado em AMPL resolvido com Gurobi para fornecer uma análise comparativa com a abordagem Monte Carlo Tree Search.

Os resultados finais mostram que cada abordagem destaca-se em diferentes condições. Nas instâncias de benchmark, o método Monte Carlo Tree Search provou ser mais robusto, encontrando soluções admissíveis na maioria dos casos onde o modelo de otimização baseado em AMPL frequentemente falhava em escalar. No entanto, a qualidade da sua solução permaneceu substancialmente abaixo da dos métodos do estado de arte. Em contraste, quando aplicados às instâncias FCUP, que são menores e mais estruturadas, a formulação AMPL+Gurobi superou claramente o Monte Carlo Tree Search, alcançando pontuações de violação de restrições ”soft” significativamente mais baixas.

Palavras-chave: Monte Carlo Tree Search (MCTS), ITC2007, AMPL, Horários de Exames

Abstract

The construction of timetables is a challenging task for educational institutions, particularly when large instances and complex constraints are involved. However, many institutions still use manual scheduling methods that often lead to inefficient timetables and are vulnerable to human error.

The main objective of this thesis is to develop and evaluate a Monte Carlo Tree Search solution for examination timetabling problems. The proposed approach is designed to handle multiple hierarchical objectives and is evaluated using both the International Timetabling Competition Examination Track benchmark instances and real-world data from the Faculdade de Ciências da Universidade do Porto. As a secondary objective, an AMPL-based mathematical model solved with Gurobi is developed to provide a comparative analysis with the Monte Carlo Tree Search approach.

The final results show that each approach excels under different conditions. On the benchmark instances, the Monte Carlo Tree Search method proved more robust, finding feasible solutions in most cases where the AMPL-based optimization model frequently failed to scale. However, its solution quality remained substantially below that of state-of-the-art methods. In contrast, when applied to the smaller and more structured real-world FCUP datasets, the AMPL+Gurobi formulation clearly outperformed Monte Carlo Tree Search, achieving significantly lower soft-constraint violation scores while maintaining feasibility.

Keywords: Monte Carlo Tree Search (MCTS), ITC2007, AMPL, Examination Timetabling

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AFDA Adaptive Flex-Deluge Algorithm. 12

CP Constraint Programming. 5

CSP Constraint Satisfaction Problem. 11

DSatur Degree of Saturation algorithm. 28, 31

FCUP Faculdade de Ciências da Universidade do Porto. vi, 2, 3, 38– 42, 44, 45

GD Great Deluge. 8, 11– 13

GP Goal Programming. 5, 6

GRASP Greedy Randomized Adaptive Search Procedure. 11

HC Hill Climbing. 7, 11– 13

IFS Iterative Forward Search. 11

ILP Integer Linear Problem. 5, 6, 18

ILS Iterated Local Search. 11, 12

IP Integer Programming. 6, 11

ITC2007 International Timetabling Competition 2007. vi, 2– 4, 9– 13, 18, 21, 26, 28, 33– 38, 40, 41, 44, 45

LAHC Late Acceptance Hill Climbing. 13

LD Largest Degree. 12

LNS Largest Number of Students. 12

LP Linear Programming. 5, 6

MCTS Monte Carlo Tree Search. vi, vii, 2– 4, 13, 14, 16, 26, 27, 29, 33, 35, 36, 40– 42, 44, 45

MDP Markov decision process. 13, 16

MIP Mixed-Integer Problem. 5, 6, 18, 21, 36, 37

NLP Nonlinear Programming. 5, 6

SA Simulated Annealing. 8, 11, 13, 44

SCHC Step Counting Hill Climbing. 12, 13

TS Tabu Search. 8, 11, 12, 44

UCT Upper Confidence Bound 1 applied to trees. 14, 30

1. Introduction

In this thesis, we investigate examination timetabling problems in academic institutions. In general, as noted by Burke et al. [1], a timetabling problem involves four parameters: T, a finite set of times; R, a finite set of resources; M, a finite set of meetings; and C, a finite set of constraints. The goal is to assign times and resources to the meetings so as to satisfy the constraints as far as possible. This definition can be used for many timetabling problems, such as nurse scheduling, sports timetabling, transportation timetabling, and educational timetabling.

The educational timetabling problem can be separated into three main categories, depending on the type of institution and the kind of event we are scheduling, either classes, lectures, or exams [2]:

- High-School Timetabling - The weekly scheduling for all classes of a high school, avoiding teachers meeting two classes or more at the same time, and vice versa.
- University Course Timetabling - The weekly scheduling for all lectures of a set of university courses, avoiding as much as possible the overlap of lectures of courses having common students.
- University Examination Timetabling - The scheduling for the exams of a set of university courses, avoiding overlap of exams of courses having common students, and spreading the exams for the students as much as possible.

Timetabling constraints are context-dependent and classified as hard or soft, where hard constraints are a set of rules that must be followed to create a feasible timetable solution and soft constraints are constraints that are not mandatory. Solutions are feasible if all hard constraints are met, otherwise infeasible. According to Ceschia et al. [2], in a timetabling problem, the goal is to minimize a cost function defined by a weighted combination of soft constraint violations while satisfying hard constraints. Among feasible solutions, quality is determined by how well the cost is minimized: an optimal solution minimizes the cost function, while suboptimal solutions meet hard constraints but yield higher costs. The sketch in Fig. 1.1 illustrates this classification.

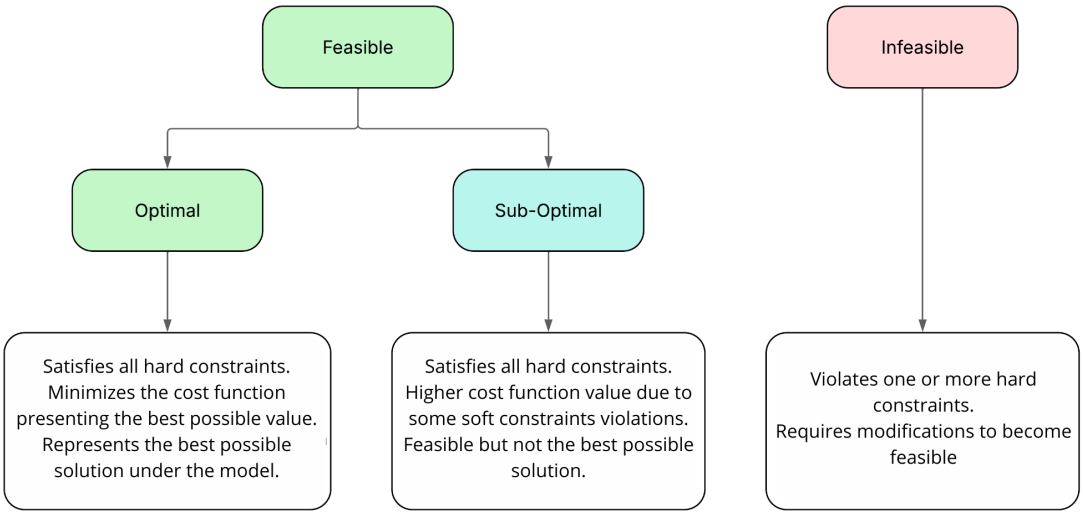

Figure 1.1: Possible solution classifications.

1.1. Objectives

Examination timetabling is a particularly hard variant that has been widely studied [1, 3].

The main goal of this thesis is to develop and evaluate a Monte Carlo Tree Search (MCTS)-based method for solving examination timetabling problems. The tool can function independently or assist in timetable construction by suggesting improvements, detecting conflicts, and assessing quality. It supports multiple hierarchically structured objectives and is benchmarked using instances from the International Timetabling Competition 2007 (ITC2007). Additionally, real-world data from Faculdade de Ciências da Universidade do Porto (FCUP) is used to test the approach. A secondary goal is to model the problem in AMPL [4] and solve it with Gurobi [5], enabling a performance comparison between the optimization-based and heuristic MCTS approaches.

1.2. Overview of the thesis

The remainder of this thesis is organized as follows.

Chapter 2 explores examination timetabling problems and the methodologies used, followed by the formal definition of the ITC2007 benchmark problem, the top performing approaches, and more recent state-of-the-art approaches, concluding with an overview of MCTS.

Chapter 3 presents a mathematical model influenced from ITC2007 examination track, implemented using the AMPL modeling language and solved with an external optimization solver `Gurobi`.

Chapter 4 details the proposed MCTS-based solution method.

Chapter 5 reports empirical results from both approaches on ITC2007 and real FCUP data, including performance comparisons.

Chapter 6 presents the conclusions from the development and evaluation of both approaches and outlines directions for future work.

1.3. Contributions of the thesis

This thesis makes several contributions to the study of examination timetabling, focusing on both metaheuristic and optimization-based solution methodologies. The main contributions are as follows.

- A complete MCTS-based algorithm was designed and implemented to construct feasible timetables under the ITC2007 examination timetabling problem framework. The algorithm incorporates various strategies for node selection, expansion, simulation, and backpropagation, tailored to the specific constraints and objectives of examination timetabling.
- A comprehensive mixed-integer programming formulation was developed, incorporating period, room, and institutional constraints, as well as a student-grouping strategy to reduce model size. The AMPL implementation also supports multi-room exam assignments, extending the flexibility of the original ITC2007 model.
- Both approaches were empirically evaluated using benchmark instances from the ITC2007 examination track and real-world data from FCUP. The results highlight complementary strengths. While MCTS excels in finding feasible solutions in large benchmark instances, AMPL achieved better solution quality on the smaller, more structured FCUP datasets.

2. Background

This chapter reviews examination timetabling problems and solution methodologies. It introduces the problem formally, briefly surveys various approaches, presents ITC2007 benchmark and its top performing methods, and concludes with an overview of the MCTS algorithm and its four key steps.

2.1. Examination timetabling problem

Examination timetabling problem was defined by Qu et al. [1] as ”assigning a set of exams into a limited number of timeslots (time periods) and rooms (of certain capacity), subject to a set of constraints.”

Manually solving timetabling problems is time-consuming and demands expert knowledge and familiarity with institutional constraints and scheduling logic, often resulting in suboptimal schedules. That has driven research into automated timetabling for over 60 years [6].

Examination timetabling problems are classified into two main types: uncapacitated, which ignores room size constraints, and capacitated, which accounts for limited room capacity per time slot [1–3].

The uncapacitated variant represents the traditional version of the examination timetabling problem [3]. Carter et al. [7] laid the foundation for modern research in examination timetabling. One of their key contributions was the creation of the first widely recognized benchmark for evaluating and comparing timetabling algorithms. Known as the Toronto dataset, it consists of 13 benchmark examination timetabling problems collected from various universities, primarily in Canada. This dataset has become a widely used standard for testing and comparing automated timetabling algorithms [1]. Originally uncapacitated, the dataset was later extended to include room capacity limits and three exam slots per day to better model real-world constraints [3].

2.2. Examination timetabling solution methodologies

Most surveys on examination timetabling problems categorize algorithmic techniques into several major classes [1, 3, 6, 8]. While many successful methods in the literature involve a hybridization of multiple techniques [1], the primary categories include mathematical

optimization, heuristics, metaheuristics, matheuristics, hyper-heuristics, and hybrid approaches.

Given that this thesis implements both a mathematical optimization approach (using AMPL with the `Gurobi` solver) and a metaheuristic approach (MCTS), a more detailed description of these two categories is provided in the following sections. All categories can be briefly described as follows [3]:

- Mathematical optimization - formulates the problem as a mathematical model with an objective function to optimize that is subject to constraints, using exact methods described in Section 2.2.1.
- Heuristics - employ problem-specific rules and domain knowledge to construct solutions within an acceptable computational cost.
- Metaheuristics - employ general algorithmic frameworks that guide the search process through the solution space using strategies that balance exploration and exploitation.
- Matheuristics - combine the exact methods of mathematical programming with metaheuristics, utilizing mathematical models in a heuristic framework.
- Hyper-heuristics - depending on how they explore the search space, they can be delineated in two major types: generation hyper-heuristics create new heuristics using heuristic components and operators; selection hyper-heuristics enhance the solutions by selecting low-level heuristics from a set.
- Hybrid approaches - integrate multiple methodologies, either by embedding a metaheuristic algorithm into a local search tool or by incorporating various metaheuristic algorithms into a unified framework.

2.2.1 Mathematical optimization

Mathematical optimization, commonly referred to as mathematical programming is categorized into Linear Programming (LP) and Nonlinear Programming (NLP) [3]. This category includes Integer Linear Problem (ILP), Mixed-Integer Problem (MIP), Constraint Programming (CP) and Goal Programming (GP) [3].

Winston [9] defined a LP problem as an optimization problem for which we attempt to maximize (or minimize) a linear function of decision variables named objective function.

The values of the decision variables must satisfy a set of constraints, where each constraint must be a linear equation or inequality. A sign restriction is associated with each variable x_i, indicating whether it must be non-negative ($x_i \geq 0$) or unrestricted in sign. Alternatively, in NLP the objective function or some constraints may not be linear [9].

Integer Programming (IP) [9] is the subset of mathematical optimization problems in which some or all decision variables are restricted to integer values. This allows modeling discrete decisions, such as binary choices, which cannot be accurately represented using continuous variables. When all decision variables are integers and both the objective function and constraints are linear, the problem is known as an ILP [9]. If the model contains both integer and continuous variables, and the objective may be nonlinear, it is referred as a MIP [9]. MIPs are generally more difficult to solve than linear programs due to their combinatorial nature. Most are classified as NP-hard, meaning that no polynomial-time algorithms are known, and do not exist unless P=NP. GP is one of the oldest multi criteria decision making techniques used in optimization of multiple objective goals [10], thus being considered as a branch of multi-objective optimization. The basic principle of GP is independent of the attainability of the goals, an objective will be defined such that optimization gives a result as close to the desired goals as possible [10]. If it is not possible to achieve all goals, then it finds a solution that represents the best possible compromise. For a single goal problem, the formulation and solution are similar to LP [10].

2.2.2 Metaheuristics

As noted by Lewis [8], drawing from the Metaheuristics Network definition [11], a metaheuristic ”is a set of concepts that can be used to define heuristic methods that can be applied to a wide set of different problems. In other words, a metaheuristic can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem”.

The dominance of metaheuristics as the preferred methodology has been documented across multiple time periods, with various surveys identifying metaheuristics as the most frequent approach [1–3, 8]. However, when applying metaheuristics to timetabling problems specifically, Lewis [8] observes that these problems present unique challenges due to the presence of both hard and soft constraints. Due to this, Lewis [8] indicates that most metaheuristic algorithms for timetabling fall into one of three categories: one-stage algorithms, two-stage algorithms, and algorithms that allow relaxations.

The one-stage algorithm is used to obtain a solution that simultaneously satisfies hard and soft constraints. The most common form of approaches are the two-stage algorithms. This category is separated into two stages: the first stage involves ignoring soft constraints and concentrating on solving hard constraints in order to arrive at a feasible solution; the second stage involves attempting to identify the optimal solution by trying to lower the values of the soft constraints as much as possible given the first phase's solution. Algorithms that allow relaxation can weaken some constraints to attempt to solve the relaxed problem. At a later step of the algorithm, the satisfaction of the initial weakened constraints will be taken into account.

According to Talbi [12] while designing a metaheuristic, two opposite criteria must be taken into account: exploration of the search space and exploitation of the best solutions found. In exploitation, the promising regions are explored more thoroughly in hope to find better solutions, while exploration ensures that unexplored regions are visited to ensure that all regions of the search space are evenly explored and not confined to a reduced number of regions [12]. Regions are determined promising by the obtained ”good” solutions.

Although metaheuristics can be classified using various criteria [12], we organize our review around the single-solution based search versus population-based search. This distinction captures the fundamental algorithmic differences between exploitation-oriented methods that refine individual solutions and exploration-oriented approaches that evolve entire solution populations.

Single-solution metaheuristics

Single-solution metaheuristics iteratively apply the generation and replacement procedures from the current single solution [12]. In the generation phase, a set of candidate solutions are generated from the current s solution through local transformation of the solution to generate a set $C_(s)$. In the replacement phase, a solution $s' \in C_(s)$ is selected to replace the current solution, and this process iterates until a given stopping criteria [12].

Since all approaches discussed in the following sections use one or more of these metaheuristics algorithms, we will be discussing the most used below.

The Hill Climbing (HC) algorithm [13] iteratively moves to neighboring solutions that improve on the current one, selecting the best available move at each step. By focusing

solely on improving steps, it can efficiently ascend toward local optima but lacks mechanisms for escaping them, often requiring enhancements or restarts to explore the broader solution space.

Tabu Search (TS) [14] explores the solution space by evaluating neighboring solutions and allowing moves to both improving and non-improving candidates, guided by adaptive memory structures (the ”tabu list”). This list prevents the search from revisiting recently explored solutions, enabling the algorithm to escape local optima and explore more diverse regions of the solution space.

The Great Deluge (GD) algorithm [15] is characterized by accepting new solutions if they are better than a gradually decreasing threshold (the ”water level”), allowing the search to escape local minima by controlling which worsening moves are accepted as the threshold drops over time.

The Simulated Annealing (SA) algorithm [16] explores the solution space by occasionally accepting worse solutions according to a probability function that decreases over time (the ”temperature”). This probabilistic acceptance of non-improving moves allows the algorithm to escape local optima, with the likelihood of such moves diminishing as the temperature cools, thereby balancing exploration and exploitation throughout the search.

Population-based metaheuristics

In population-based algorithms a whole population of solutions is evolved [12]. Common examples include evolutionary algorithms, swarm intelligence algorithms, and nature-inspired algorithms like Genetic Algorithms (GA), Particle Swarm Optimization (PSO), and Ant Colony Optimization (ACO). In GA [12], the search process begins with an initial population of solutions, often generated randomly. The algorithm then proceeds iteratively through generations, applying variation operators such as crossover and mutation to generate new candidate solutions. After the generation phase, a selection or replacement mechanism determines which individuals will survive to the next generation. This may involve replacing the entire population or only a subset, depending on the specific variant of the algorithm. The selection process is typically guided by a fitness function, which evaluates how well each individual solves the problem. This iterative process continues until a predefined stopping criterion is met, and the final output is usually the best-performing individual(s) from the population, representing an approximate solution to the optimization problem.

2.3. ITC2007 examination timetabling track

The ITC2007 was created to advance research in capacitated examination timetabling by providing a standardized benchmark for evaluating solution techniques. Prior to the competition, timetabling problems were widely studied, but comparisons between different approaches were challenging since there were no common datasets and problem formulations.

Although subsequent competitions, such as ITC2011, ITC2019, and ITC2021 were organized to further promote research in the timetabling area, they shifted focus towards other types of timetabling problems. ITC2011 addressed high-school timetabling problem that involves scheduling lectures across a week while considering room assignments [17]. ITC2019 focused on the university course timetabling problem with an emphasis on real-world multi-room constraints [18]. ITC2021 changed entirely from educational timetabling problems to sports timetabling whose goal is to construct a compact double round-robin double tournament with 16 to 20 teams [19]. As such, despite their relevance, these later competitions did not revisit the examination timetabling problem, making ITC2007 the most prominent benchmark for research in this specific domain.

The examination track consists of 12 benchmark instances designed to capture a wide range of examination timetabling scenarios encountered in universities. These datasets vary substantially in size and structure: the number of exams ranges from small instances with fewer than 100 exams to large instances exceeding 800 exams, while the number of students and enrolment patterns differ significantly across cases. Some instances feature dense conflict matrices with many students enrolled in multiple overlapping courses, whereas others display sparser conflict patterns. The number of available periods also varies, with some instances offering tight scheduling windows and others allowing more flexibility. This diversity means that each instance poses a distinct challenge, ranging from highly constrained feasibility-focused cases to more flexible but quality-driven ones, making the ITC2007 suite a comprehensive and demanding benchmark for algorithmic evaluation. [20].

The set of hard constraints are the following:

- It is not permitted for any student to take more than one exam at once;
- A room cannot have more students allocated to it than it can hold;
- The duration of the exam cannot exceed the allotted time period;

- Period hard constraints must be followed; e.g., *Exam1* must be scheduled after *Exam2*;
- Room allocations hard constraints must be followed; e.g., if *Exam1* is considered exclusive and is scheduled for *Room1* and *Period1*, then no other exam can be scheduled for that same combination;

The set of soft constraints is calculated using the following parameters:

- The number of occurrences when students have to sit two exams in a row on the same day;
- The number of occurrences when students have to sit two exams on the same day. This constraint only becomes important when there are more than two examination periods on one day;
- The number of occurrences when exams for the same student are scheduled too close, violating the minimum number of periods that should separate them;
- The number of occurrences of exams timetabled in rooms along with other exams of differing time duration;
- The number of large exams scheduled for later periods;
- The number of times a period is used which has an associated penalty specified by the institution;
- The number of times a room is used which has an associated penalty specified by the institution;

With these constraints defined, the solutions were then evaluated as follows. First, it is checked if the solution is feasible, if all hard constraints are satisfied, and a distance to feasibility is computed. If the solution is feasible, it is then evaluated based on a cost function that measures the soft constraints total penalty. Then, competitors' solutions were ranked based on the distance to feasibility and their cost value.

2.3.1 Winning approaches

In this section, we review the ITC2007's top performing approaches.

Tomáš Müller's method won two of the three ITC2007 tracks, examination track and curriculum based course timetabling, and finished fifth in post enrollment based course timetabling [21]. To tackle the tracks, the method uses a hybrid approach, structured as a two-phase algorithm [22]. The method employed the Iterative Forward Search (IFS) method in the first stage with conflict-based statistics [23] to prevent IFS from looping, to obtain feasible solutions [22]. The IFS algorithm [23] incrementally constructs a complete solution by assigning variables one at a time, starting from an empty assignment. In each iteration, a variable is selected and a value is assigned with the aim of minimizing the number of violated hard constraints. If the assignment causes conflicts, the conflicting variables are unassigned, allowing the search to dynamically recover from constraint violations. The process continues until all variables are assigned. By prioritizing variables that are harder to place and selecting values that minimize disruption, IFS effectively guides the construction toward feasible solutions. In the second stage, several optimization algorithms are used to improve the cost of the feasible solution found, using them in the following order: HC, GD, and SA [22].

Christos Gogos method was developed only for the examination track and achieved second place [21]. Gogos method follows a multi-phase hybrid strategy, beginning with a preprocessing step that checks for hidden dependencies between exams [24]. Following the preprocessing stage, a construction stage is conducted multiple times utilizing the metaheuristic Greedy Randomized Adaptive Search Procedure (GRASP) [25]. GRASP iteratively builds feasible solutions by combining greedy randomized choices with local search. In this implementation, GRASP incorporates enhancements such as multiple ordering criteria and elements TS to improve the solution [24]. After construction, the method applies local search combining HC and SA to further refine the solution. In the final optimization phase, an IP formulation is used to solve room assignment subproblems optimally for each period using a Branch and Bound strategy with the GLPK solver [24].

Atsuta et al. used the same approach for all three tracks and got second place in post enrollment based course timetabling while also achieving third place on examination track and curriculum based course timetabling [21]. Their approach formulates the problem as a Constraint Satisfaction Problem (CSP), a modeling framework where variables must be assigned values from given domains such that a set of constraints is satisfied, and solves it with a hybrid metaheuristic composed of TS and Iterated Local Search (ILS) [26]. To begin, a general purpose CSP formulation encodes variables for exam-period-room assignments, with domains determined by feasible scheduling combinations. A dummy variable

is used to allow partial assignments, ensuring the solver can proceed even with initially infeasible configurations [26]. The algorithm then employs TS to explore the solution space, using short-term memory to avoid cycles and a strategic oscillation mechanism to shift emphasis between feasibility and optimization. Once a feasible or promising solution is obtained, ILS – a metaheuristic that iteratively perturbs a locally optimal solution and applies local search to refine it – is used to escape local optima [26]. This phase repeatedly perturbs the solution by modifying single assignments and re-optimizes using local search, restarting from high-quality solutions [26].

2.3.2 Newer Approaches

In this section, we describe other approaches to the ITC2007 examination track, which were proposed after the contest and were highlighted as the most promising according to [2].

In 2016, Edmund Burke and Yuriy Bykov developed the Adaptive Flex-Deluge Algorithm (AFDA) [27]. Their approach improves search performance by adding a more flexible acceptance criterion to the GD algorithm. AFDA is designed around two key enhancements: slowing down uphill moves to maintain solution quality and adapting the search process based on specific problem properties [27]. AFDA/ Largest Degree (LD) and AFDA/ LD+ Largest Number of Students (LNS) are two specialized variations of the algorithm. AFDA/ LD utilizes the LD heuristic for replacement moves, while AFDA/ LD+ LNS combines LD with the LNS heuristic for room moves, effectively guiding the search towards better solutions [27]. According to experimental results, AFDA performed better than several existing algorithms, such as GD and greedy HC achieving nine of the top-ranked solutions in the ITC2007 benchmark results [2, 27].

Yuriy Bykov and Sanja Petrovic introduced the Step Counting Hill Climbing (SCHC) algorithm [28]. This algorithm enhances the standard HC approach by introducing a step-counting acceptance criterion, which provides a flexible balance between exploitation and exploration during the search process. The SCHC algorithm operates by allowing non-improving moves for a fixed number of steps, controlled by a single parameter known as the step counting period. Specifically, the algorithm maintains an acceptance threshold equal to the current solution's cost for a set number of iterations [28]. During this period, any candidate solution whose cost does not exceed the threshold is accepted, even if it does not improve the current solution. This approach enables SCHC to escape

shallow local optima more effectively than standard HC which only accepts strictly improving moves [28]. Three variations of SCHC were developed: SCHC-all, which counts all moves; SCHC-acp, which only counts moves that were accepted; and SCHC-imp, whose count is just of improving moves [28]. Among these, SCHC-acp outperformed SA, GD, and other SCHC variants in many instances of the ITC2007 benchmark results [2, 28].

Then, in 2017, Edmund Burke and Yuriy Bykov introduced the Late Acceptance Hill Climbing (LAHC) heuristic [29]. LAHC modifies the traditional HC approach by incorporating a delayed move acceptance criterion, which helps the search escape local optima and explore the solution space more thoroughly [29]. The core idea of LAHC is to accept a candidate solution not only if it is better than the current solution, but also if it is better than the solution found a fixed number of iterations earlier. This is achieved by maintaining a list of previous solution costs of length L [29]. At each iteration, the algorithm compares the cost of the candidate solution to the cost stored L steps back in the list. If the candidate's cost is less than or equal to this delayed cost, the move is accepted and the list is updated accordingly. This mechanism allows the algorithm to accept worsening moves under controlled circumstances, promoting diversification and enabling the search to escape shallow local minima [29]. Experimental results on the ITC2007 benchmark instances showed that LAHC performed better in comparison to the podium approaches and also the SA and GD algorithm, in the majority of instances of ITC2007, especially when applied to bigger problem sets [2, 29].

2.4. Monte Carlo Tree Search background

MCTS is a heuristic search algorithm used for decision making in problems with large or infinite search spaces. The method has become a state-of-the-art technique in the domain of game playing (e.g., Go), but has also found applications in practical domains (transportation, scheduling or security). This algorithm is directly applicable to problems that can be modeled by a Markov decision process (MDP) [30]. MDP is a process modeled as a tuple $(\mathcal{S}, A, T, \mathcal{R})$ where:

- $\mathcal{S}$ - is a set of states that are possible in a state space;
- A - denotes a set of actions available to perform in state s;
- $T(s, s', a)$ - is the transition function modeled by a probability that action a performed in state s will lead to state s';

- $\mathcal{R}(s)$- is the immediate reward for reaching state s;

In non-trivial problems, the state space cannot be fully searched [30, 31], thus MCTS is allotted some computational budget which can be either a specified number of iterations or a specified time limit. MCTS is an anytime algorithm. This property means that it can be stopped at any time and offer the current best decision. Naturally, with more iterations, it is more likely that the recommended decision is the optimal one.

A MCTS iteration proceeds through four steps as depicted in figure 2.1.

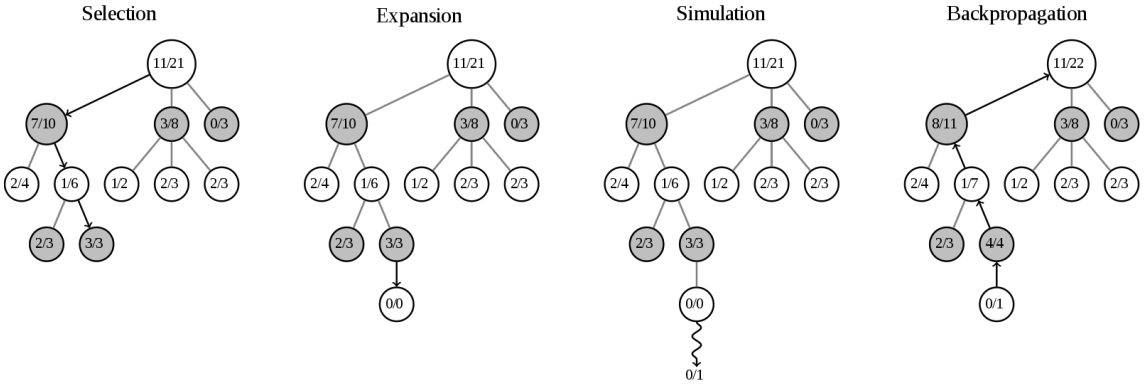

Figure 2.1: The four steps of a single iteration of Monte Carlo tree search. The numbers within the nodes represent the total reward (left) and number of visits to the node (right). The figure depicts a search tree for a two-player game.

The four steps of each MCTS iteration are described in the following sections, based on the methodologies presented in [30, 31].

2.4.1 Selection

Starting from the root node, the child that currently looks most ”promising” is selected. This is done recursively until a node n which has not yet been fully expanded is reached. The definition of promising is one of the key aspects determining the performance of MCTS, and can be represented by a utility function $U(n)$ that we wish to maximize at each level of the tree. The main difficulty in selecting child nodes is maintaining a balance between the exploitation and exploration of moves. The Upper Confidence Bound 1 applied to trees (UCT) algorithm is capable of balancing the exploitation and exploration and is used as the utility function $U(n)$ in the selection step with the following formula:

$$U(i) = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N}{n_i}} \tag{2.1}$$

In this formula w_i stands for the number of wins for child node i, n_i stands for the number of times node i has been visited, N is the total number of visits to the parent node of i, and C is the exploration parameter that can be chosen empirically.

The selection procedure is presented in pseudo-code form in algorithm 1.

Algorithm 1 Selection step in Monte Carlo tree search

```
function Selection(n)
    if n has unexplored actions then
        return n
    else
        C_s ← set of children of n
        n' ← arg max_{n∈C_s} U(n)
        return Selection(n')
    end if
end function
```

2.4.2 Expansion

One or more children of the selected node n, from the selection function, are created by applying unexplored actions to the state associated with n. These new nodes are then attached to the tree. The two main schemes for expansion are:

- Single expansion - a single child node is created using an unexplored action from node n. The remaining unexplored actions are recorded for a later time when node n is selected again for expansion. The action to explore can be selected randomly or based on some heuristic evaluation;
- Full expansion - all children of node n are immediately created by generating all possible actions. Using this scheme, the order of creation is irrelevant, since all are created in the same iteration;

2.4.3 Simulation

Starting from the state of the node (or nodes) created in the expansion step, available actions are iteratively applied until a terminal state is reached. Various approaches can be taken in simulation, ranging from uniform random decisions to heuristic construction algorithms that incorporate domain-specific knowledge.

The general simulation process is presented in pseudo-code form in algorithm 2.

Algorithm 2 Simulation step in Monte Carlo tree search

```
1: function Simulation(n)
2:     while n is not terminal do
3:         A ← set of available actions in n
4:         a ← select action from A
5:         n ← copy of n with a applied
6:     end while
7:     return f(n)                                  ▷ reward
8: end function
```

The steps in lines 3 and 5 require a generative model of the problem domain. Although this is evidently problem-dependent, it is not what is referred to when talking about using domain-specific knowledge. When incorporating this knowledge in MCTS, the foremost place this can be done is in the action selection of line 4.

2.4.4 Backpropagation

Once a simulation has reached a terminal state, its reward Δ is evaluated. This reward may represent the cumulative return over a sequence of states traversed during the simulation, and can be derived from the MDP's reward function $\mathcal{R}(s)$ – for instance, by summing or averaging rewards collected along the trajectory.

Starting with the simulation's final node n, and going up the tree all the way until the root node is reached, node statistics are updated to include the newly found Δ. Depending on the application domain, the set of node statistics being tracked can vary. In game-playing, the total reward obtained from a node is tracked in addition to its number of visits. Applying MCTS to different problems can change the statistics needed, and accordingly, the backpropagation step must be adapted.

Algorithm 3 displays a simple backpropagation procedure.

Algorithm 3 Backpropagation step in Monte Carlo tree search

```
1: function Backpropagation(Δ, n)
2:     while n is not null do
3:         increment visit count for n
4:         add reward Δ to node n
5:         n ← parent of node n                    ▷ move up the tree
6:     end while
7: end function
```

3. Mathematical model

This section presents the mathematical model developed for solving the exam timetabling problem. The model defined in this thesis is primarily inspired by the formulation described in [20], which was developed for ITC2007 examination track, but unlike the formulation in [20], which is an ILP, our mathematical model is formulated as a MIP. The model is structured into several components:

- Sets and parameters, which define the input data structure;
- Decision variables, which capture exam assignments and soft constraint penalties;
- Objective function, which balances competing interests from students, examiners, and university administration.
- Hard constraints, which ensure the feasibility of the timetable;
- Soft constraints, which encode desirable features of a good schedule;

The formulation introduces both binary and continuous variables to support partial relaxations where needed, and aggregates students with identical exam enrollments to reduce redundant computations.

Following the formal specification of the model, a complete implementation in the AMPL modeling language is provided. The AMPL version is a direct translation of the mathematical model, with figures showing the encoding of sets, parameters, variables, objective function, and constraints. While AMPL defines the model structure, the actual optimization is performed by an external solver. For this purpose, `Gurobi` was chosen due to its proven efficiency in solving MIP problems.

3.1. Sets and parameters

All sets and parameters discussed are either directly present in the input file, or can be derived from it.

E is the set of all exams and contains three parameters:

- s_i^E - Size of exam $i \in E$ i.e., the number of students attending it;

- d_i^E - Duration of exam $i \in E$;
- f_i^E - Boolean that is 1 only if exam i is subject to frontload penalties;

In the input file format, the ”FRONTLOAD” entry specifies 3 parameters. The first parameter is *number of largest exams*. Largest exams are specified by class size, which is used to select which exams are subject to frontload penalty, that is, the exams for which $f_i^E = 1$.

D is the set of all unique exam durations used across all exams. u_{id}^D is a boolean that is 1 only of exam i has *duration type* d, where d is an index for set D. We call them *types* because the duration values do not matter, only whether they are equal.

S is the set of students. Student enrollments are encoded by t_{is} that is 1 only if student s is enrolled in exam i, 0 otherwise. To minimize redundant evaluations of soft constraints for students with identical exam enrollments, a grouping approach is introduced, where only a student s from a group of students, who share the same set of enrolled exams, is selected as the representative of that group. The parameter n_s is the number of students being represented by s.

Available rooms are grouped in set R, which is used in the following parameters:

- s_r^R - Size of room $r \in R$;
- w_r^R - Weight that specifies the penalty for room r;

Set P contains all scheduling periods and has four parameters:

- d_p^P - Duration of period $p \in P$;
- f_p^P - Boolean that is 1 only if period p is subject to frontload penalties.
- w_p^P - Weight that specifies penalty for using period p;
- y_{pq} - Boolean that is 1 only if periods p and q are in the same day;

3.2. Period related hard constraints

H^{aft}, H^{coin}, and H^{excl} are sets of pairs of exams that represent the three period hard constraints, where each represents:

- H^{aft} - For every pair $(e1, e2) \in H^{\text{aft}}$ exam $e1$ must occur strictly after exam $e2$;

- H^{coin} - For every pair $(e1, e2) \in H^{\text{coin}}$ exams $e1$ and $e2$ must occur in the same period, though not necessarily in the same room;
- H^{excl} - For every pair $(e1, e2) \in H^{\text{excl}}$ exam $e1$ and $e2$ must not occur in the same period;

3.3. Room related hard constraints

H^{sole} is a set of exams. For every exam $e \in H^{\text{sole}}$, if exam e is assigned to period p and room r then e must be the sole occupier, i.e. no other exam can be assigned to both p and r.

3.4. Institutional weights and parameters

To fine-tune soft constraints' evaluation, several global penalty parameters are defined within the input files. These parameters allow the institution to express its preferences regarding various aspects of the timetable. The weights are as follows:

- w^{2R} - Weight for having two exams in consecutive periods;
- w^{2D} - Weight for having two exams in the same day;
- w^{PS} - Weight for having multiple exams within a limited number of periods. This weight defaults to 1 as it is not currently specified in the input format, but included here for completeness;
- w^{NMD} - Weight for having mixed exam durations in the same room and period;
- w^{FL} - Weight for scheduling large exams in the final periods;
- g^{PS} - The preferred minimal *gap* between exams for a student;

3.5. Variables

The decision variables are separated into primary and secondary. The primary decision variables are used to represent the assignments of exams to periods and rooms, while the secondary variables receive their values given the assignment of the primary variables.

3.5.1 Primary variables

Exam assignment to periods and rooms is tracked by three primary variables:

- $X_{ip}^{P} = 1$ if exam i is in period p, 0 otherwise;
- $X_{ir}^{R} = 1$ if exam i is in room r, 0 otherwise;
- $Y_{ir}^{R} \in [0, 1]$ indicates the proportion of exam i in room r;

The variable Y_{ir}^{R}, that is not part of the original ITC2007 but added by us, allows for partial room allocation, enabling a relaxation of the problem when needed and allowing the division of large exams into several rooms to be dealt with in the MIP formulation.

3.5.2 Secondary variables

A number of auxiliary non-negative numeric variables are defined to measure the quality of solutions and support penalty calculations for soft constraints.

- C_s^{2R} - Two exams in consecutive periods penalty for student representative s;
- C_s^{2D} - Two exams in the same day penalty for student representative s;
- C_s^{PS} - Multiple exams within a limited number of periods penalty for student representative s;
- C^{NMD} - Mixed exams durations penalty, i.e., two or more exams with different durations scheduled together;
- C^{FL} - Frontload penalty, i.e., large exams in the final periods;
- C^{P} - Room usage penalty, combines the associated weight of every room used for exams;
- C^{R} - Period usage penalty, combines the associated weight of every period used for exams;

The combination of these variables serves as the basis for minimizing the objective function based on penalties.

We also use U_{dpr}^{D}, that is, 1, if *duration type* D is used in period p and room r, 0 otherwise.

3.6. Objective

The objective function was constructed with a compromise between various interested parties:

- C_s^{2R}, C_s^{2D} and C_s^{PS} - The desire each student representative has for a good individual timetable;
- C^{NMD} - Interest of exam invigilators to not have exams with different durations on the same period and room;
- C^{FL} - Represents the desire of the exam markers to first receive the largest exams in order to have more time for marking.
- C^{P} and C^{R} - Interest of the university in avoiding the use of certain periods and rooms.

The objective function is then defined as follows:

$$\sum_{s \in S} (w^{2R} C_s^{2R} + w^{2D} C_s^{2D} + w^{PS} C_s^{PS}) \times n_s + w^{NMD} C^{NMD} + w^{FL} C^{FL} + C^{P} + C^{R} \quad (1)$$

3.7. Constraints

3.7.1 Hard constraints

The model incorporates a set of hard constraints to ensure the feasibility of any timetable generated. It should be noted that constraint (4) introduces non-linearity through the product of decision variables X_{ip}^P and Y_{ir}^R. However, since X_{ir}^R is binary and Y_{ir}^R is bounded between 0 and 1 by constraints (11) and (12), this bilinear term is linearized.

Each exam must be assigned to at least one room (2) and one period (3).

$$\forall i \in E: \quad \sum_{r \in R} X_{ir}^R \geq 1 \quad (2)$$

$$\forall i \in E: \quad \sum_{p \in P} X_{ip}^P \geq 1 \quad (3)$$

The total number of students assigned to a room in any given period must not exceed the capacity of the room.

$$\forall p \in P, \forall r \in R: \quad \sum_{i \in E} s_i^E X_{ip}^P Y_{ir}^R \leq s_r^R \tag{4}$$

The duration of any exam does not last longer than the assigned period.

$$\forall p \in P, \forall i \in E: \quad d_i^E X_{ip}^P \leq d_p^P \tag{5}$$

No student representative is scheduled for more than one exam in the same period.

$$\forall p \in P, \forall s \in S: \quad \sum_{i \in E} t_{is} X_{ip}^P \leq 1 \tag{6}$$

Additional constraints enforce precedence, or simultaneous or mutual exclusion between exam's period assignment.

$$\forall (i,j) \in H^{\text{aft}}, \ \forall p,q \in P, \text{ with } p \leq q: \quad X_{ip}^P + X_{jq}^P \leq 1 \tag{7}$$

$$\forall (i,j) \in H^{\text{coin}}, \ \forall p \in P: \quad X_{ip}^P = X_{jp}^P \tag{8}$$

$$\forall (i,j) \in H^{\text{excl}}, \ \forall p \in P: \quad X_{ip}^P + X_{jp}^P \leq 1 \tag{9}$$

The room exclusivity constraint ensures that exams that require exclusive access to a room do not share it with other exams.

$$\forall i \in H^{\text{sole}}, \ \forall j \in E, \ j \neq i, \ \forall p \in P, \ \forall r \in R: \quad X_{ip}^P + X_{ir}^R + X_{jp}^P + X_{jr}^R \leq 3 \tag{10}$$

To support the relaxation of the room assignment, additional constraints are introduced to ensure consistency in the relaxed model. The first constraint ensures that the proportion of students of exam i in room r is zero if that exam is not scheduled for that room.

$$\forall i \in E, \forall r \in R: \quad Y_{ir} \leq X_{ir}^R \tag{11}$$

The second constraint ensures that each exam is fully assigned across rooms, summing to one unit of room allocation:

$$\forall i \in E: \quad \sum_{r \in R} Y_{ir} = 1 \tag{12}$$

3.7.2 Soft constraints

The model includes a number of soft constraints to guide the optimization towards high-quality solutions besides mere feasibility. The two-in-a-row (13) and two-in-a-day (14) constraints track when a student has exams scheduled in adjacent periods or on the same day.

$$C_s^{2R} = \sum_{\substack{i,j \in E \\ j \neq i}} \sum_{\substack{p,q \in P \\ q=p+1 \& y_{pq}=1}} t_{is} t_{js} X_{ip}^P X_{jq}^P \tag{13}$$

$$C_s^{2D} = \sum_{\substack{i,j \in E \\ j \neq i}} \sum_{\substack{p,q \in P \\ q>p+1 \& y_{pq}=1}} t_{is} t_{js} X_{ip}^P X_{jq}^P \tag{14}$$

Period Spread constraint checks if exams are spread across a wider range, of at least g^{PS} number of periods.

$$C_s^{PS} = \sum_{\substack{i,j \in E \\ j \neq i}} \sum_{\substack{p,q \in P \\ p<q \leq p+g^{\mathrm{PS}}}} t_{is} t_{js} X_{ip}^P X_{jq}^P \tag{15}$$

Non-Mixed Duration looks for mixed exam durations in the same room and period to encourage uniform structure.

$$\forall d \in D,\ \forall i \in E \text{ with } u_{id} = 1,\ \forall p \in P,\ \forall r \in R: \quad U_{dpr}^D \geq X_{ip}^P + X_{ir}^R - 1 \tag{16}$$

$$\forall p \in P,\ \forall r \in R: \quad 1 + C_{pr}^{NMD} \geq \sum_{d \in D} U_{dpr}^D \tag{17}$$

$$C_{pr}^{NMD} \geq 0 \tag{18}$$

$$C^{NMD} = \sum_{p \in P} \sum_{r \in R} C_{pr}^{NMD} \tag{19}$$

Frontload models the penalty for having large exams scheduled after the early periods, i.e., in periods for which f_p^P is 1.

$$C^{FL} = \sum_{i \in E} \sum_{p \in P} f_i^E f_p^P X_{ip}^P \tag{20}$$

Soft Period (21) and Soft Room (22) introduce penalties associated with less desirable periods or rooms, based on predefined weights.

$$C^P = \sum_{p \in P} \sum_{i \in E} w_p^P X_{ip}^P \tag{21}$$

$$C^R = \sum_{r \in R} \sum_{i \in E} w_r^R X_{ir}^R \tag{22}$$

4. Monte Carlo Tree Search

Although MCTS has gained significant attention in domains such as game playing and scheduling, to the best of our knowledge, it has not been applied to the ITC2007 examination track. Due to its strengths in balancing exploration and exploitation in large search spaces, MCTS presents an interesting approach to the examination timetabling problem.

Having introduced the fundamental concepts of MCTS in Section 2.4, the following sections present its concrete implementation for the examination timetabling problem. Figure 4.1 provides an overview of the main classes and their relationships in the implementation. The following sections begin with an explanation of the data structures that represent the state of a partial solution and the node structure used in the MCTS tree, as these components are essential to maintaining the feasibility and integrity of candidate timetables throughout the search process. Additionally, each of the four stages of a MCTS iteration is examined in detail with a focus on the domain-specific heuristics to guide the search effectively, ensuring both the satisfaction of hard constraints and the minimization of soft constraint violations.

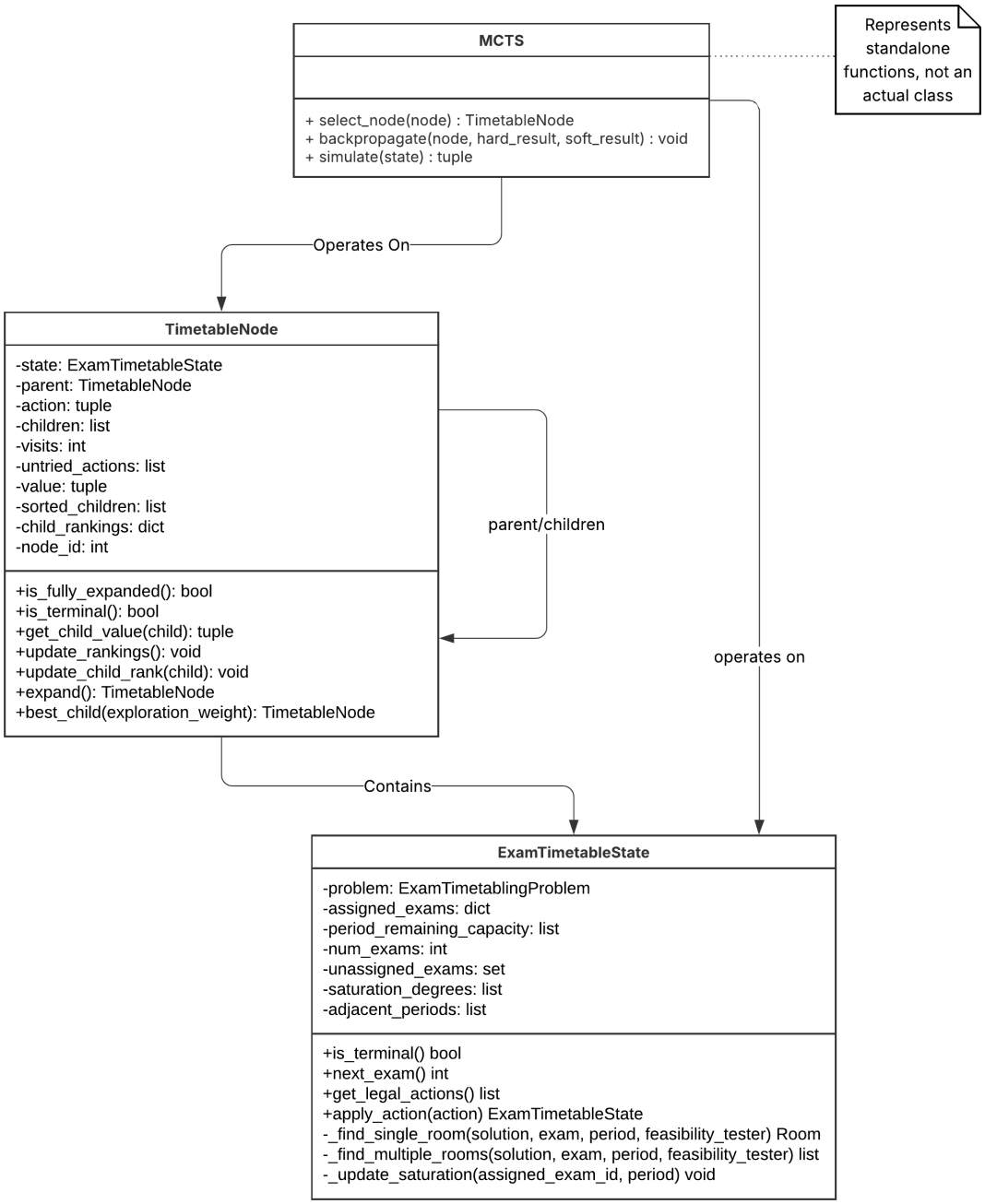

Figure 4.1: Overview of main classes and their relationships from MCTS approach

4.1. State

The `ExamTimetableState` class encapsulates the state of a partial or complete exam timetable, representing a snapshot of the solution process, keeping track of which exams were assigned to each period and room or combination of rooms, while maintaining feasibility by only creating and applying new states that do not break any constraints.

The class constructor initializes with a problem parameter that is an instance of the input model based on the ITC2007 instance defined in `ExamTimetabingProblem`, explained in Appendix A.2, containing all the information from the input file already processed for handling, and also an optional dictionary of already assigned exams to maintain continuity in the construction of the timetable. This constructor sets up several data structures, such as a dictionary to map exams to their period and room assignments, a set of exams that are yet to be scheduled, and lists that are used in a heuristic inspired by the Degree of Saturation algorithm (DSatur) to prioritize exams with more conflicts. If the state is initialized with preassigned exams, the class updates the data structures to reflect the already defined decisions.

To determine which exam should be scheduled next, as mentioned above, a heuristic based on the DSatur algorithm is used. This graph coloring heuristic is designed so that no two adjacent vertices of the graph share the same color. Within the context of our problem, each exam is treated as a node where an edge exists between two exams if any students are enrolled in both, with periods being treated as “colors”. This heuristic prioritizes scheduling exams that have fewer feasible periods due to conflicts with already assigned exams. For each unassigned exam, the heuristic computes a saturation degree that is comprised of the number of distinct periods already assigned to its conflicting neighbors. After the exam, the highest saturation degree is selected for scheduling. If there is a tie in this selection, a second criterion is used where the exam with the highest number of conflicts with unassigned exams is selected.

By selecting the next exam to be scheduled, the method `get_legal_actions` generates all feasible assignments, considering both period and room constraints. Firstly, it checks if the exam has a coincidence constraint that requires the exams within the constraint to be scheduled in the same period. If any of the linked exams are already assigned, then the method attempts to create assignments for that same period with either a single-room or multi-room allocation. If there is no coincidence constraint or no feasible assignment was found, the method tests the feasibility of all available periods, that are sorted in ascending

order of their remaining room capacity. Then, for each feasible period, as in the previous case, it checks the feasibility of a single room that perfectly or efficiently fits the exam. If this is not possible, it then looks for a combination of multiple rooms that can satisfy the exam's capacity requirements. A class called `FeasibilityTester` was developed to test the feasibility of periods and rooms, ensuring that only valid assignments are generated. This class, which is explained in Appendix A.2, checks for all constraints mentioned in Section 3.7.

Once a feasible assignment has been selected, the `apply_action` method creates a new state with the additional assignment. It also updates the list of unassigned exams, reduces the available capacity of the selected period and updates the saturation degrees of neighboring exams to reflect the new constraints. This approach maintains the integrity of the MCTS tree and ensures that branching results in independent and consistent states, which is essential for correct exploration during the search process.

4.2. Node

The `TimetableNode` class defines the structure of a node within the MCTS tree. Each node instance encapsulates a state. The `ExamTimetableState` class, described above, represents a partial or complete feasible timetable. The constructor can accept an optional reference to a parent node and the assignment that led to the current state. The node's key features include the ability to track unvisited branches and statistical information.

The node stores several attributes:

- `untried_actions` - a list of all feasible assignments that can be applied to the current state, obtained by calling the state's `get_legal_actions` method.
- A list of children nodes.
- A counter for the number of visits.
- A cumulative value - this is a tuple that contains the hard constraint and soft constraint violations for multi-objective evaluation.

To ensure a node has feasible assignments remaining, the `is_fully_expanded` method checks if the `untried_actions` attribute is empty. Additionally, `is_terminal` method checks if the current state's timetable is complete.

4.3. Iteration steps

4.3.1 Selection

As previously mentioned, the selection phase is responsible for traversing the search tree to identify a promising node for expansion. Starting at the root, the process repeatedly selects the most promising child node, traversing downwards until it reaches a partially expanded or terminal node.

The `select_node` function implements this traversal, which operates in a few steps:

- It starts at a given node and uses the `is_terminal` method to check if the node is a terminal state.
- If the node is not a terminal state and is not fully expanded (as determined by the `is_fully_expanded`), it is returned for the next step: expansion.
- Otherwise, the function calls the `best_child` method to move to the child with the lowest value.

The `best_child` method implements a modified UCT strategy for minimization problems. It assigns a selection score to each child node based on the exploitation and exploration components detailed in Section 2.4, with a specific alteration to the exploitation term.

The exploitation term is derived from a ranking system maintained by the parent node. This system assigns each child a score that is inversely proportional to its rank in a sorted list of children, called `sorted_children`. The children are sorted based on an average value of hard and soft constraint violations, which are added during backpropagation. The sorting prioritizes children with fewer violations. Once sorted, the best-performing child is assigned a ranking score of -1, the second-best a score of -1/2, and so on. These scores are stored in the `child_rankings` dictionary. This approach ensures a normalized reward that favors high-quality solutions while enabling a consistent comparison between children, regardless of the magnitude of their constraint violations.

The total selection score for each child is the sum of its exploitation and exploration terms, and the child with the lowest combined score is selected, as this implementation is framed as a minimization problem.

4.3.2 Expansion

Expansion explores new branches of the solution space by applying untried assignments to the current state. When the method is called, it removes one assignment from `untried_actions` and applies it to the timetable state using the `apply_action` method. This results in a new state with a feasible extension of the timetable. A new `TimetableNode` is then instantiated with this new state. It is linked to the current node as its parent and associated with the assignment that produced it. This new node is then added to the parent's list of children. Afterwards, the child is evaluated through a value function that returns a normalized version of its current value. This process promotes early exploration of newly created nodes by assigning high values to unvisited nodes. The resulting value is then used to insert the child into the `sorted_children` list.

4.3.3 Simulation

The simulation step, as said above, attempts to complete a partial timetable starting from the current node and state. This step does not expand the tree further. Instead, it generates a full solution to estimate how promising the current node's state is.

The simulation begins by making a copy of the current node, to ensure that any modifications do not affect the node in the tree. The simulation then enters a loop that continues as long as the copied state is non-terminal. In each iteration, it selects the next exam to schedule using the DSatur heuristic, as mentioned in Section 4.1.

After an exam is selected, periods with enough capacity are identified and considered feasible if they pass a constraint check using the `FeasibilityTester` class. If no feasible period with enough capacity is found, the algorithm relaxes this requirement and looks for any feasible period regardless of capacity. The simulation then assigns scores to periods, preferring those that minimize conflicts with already assigned exams while also having a soft preference for periods with more spare capacity. If no feasible periods are found, the simulation selects a random period as a fallback.

Once a period is selected, the simulation attempts to find a single room with the exact or best-fit capacity. If this is not possible, it then considers combinations of rooms that can collectively satisfy the exam's seating needs. As with the period selection, if no suitable room or combination of rooms is found, a random room is selected.

The resulting assignment (of the selected exam, period, and room or combination of rooms) is applied to the state, which updates the internal structure and progresses the simulation. This process repeats, exam by exam, until all exams are scheduled and the terminal state is reached. At that point, the resulting timetable is evaluated by the `Solution` class for the number of hard and soft constraints violated. Those two values are then passed to the backpropagation step.

4.3.4 Backpropagation

In this step, the results of a simulated solution are propagated up the tree from the expanded node to the root. This step is crucial to update the statistics used during the selection phase since it allows the search tree to gradually learn which parts of its parts are more promising.

The function takes as input the leaf node and the simulation results. For each node, it increments the visit count and updates the cumulative values. If the solution is feasible, the node's soft score is updated; otherwise, the hard score is updated. This distinction helps prioritize feasible solutions during selection while still considering soft constraints for fitness evaluation. Additionally, each node's parent updates the rank of its child to maintain the ranking-based exploitation metric for the selection phase.

5. Experimental results

In this section, we report the results obtained from both approaches explained in the previous chapters. We will compare our results from both approaches with values from some of the five finalists of the ITC2007, as well as with those more up-to-date approaches.

It is important to note that the results presented in this section are not directly comparable to the original ITC2007 competition results or other published approaches. As described in Section B.1, our formulation includes modifications to accommodate the possibility of assigning multiple rooms to a single exam, a feature not included in the original ITC2007 problem specification. This modification changes the problem structure and constraint satisfaction requirements, making direct numerical comparisons with existing approaches potentially misleading. The comparisons provided should, therefore, be interpreted as an indication of relative performance rather than definitive benchmarking results. Due to this modification, it was not possible to use the benchmark program from the official ITC2007 competition website. For this reason, a time limit of two hours was chosen to allow both approaches sufficient time to generate several timetables, which is valuable for these algorithmic strategies.

All computational experiments presented in this thesis were performed on a machine with the following specifications: an **Intel I7 processor at 3.20GHz**, **64 GB of RAM**, and running **macOS version 13.2.1**.

5.1. Analysis of benchmark instances

Table 5.1 presents the specifications for the 12 instances of the ITC2007 examination track. Although the instances follow a common structure, they differ widely in scale and constraint tightness, which strongly affects their difficulty. These variations help explain why the AMPL+`Gurobi` model finds feasible solutions only for a subset of instances, while the MCTS approach succeeds on all but instances 3 and 11.

The main factors influencing complexity are the numbers of **exams** and **students**, which determine the density of conflicts and the size of the search space. Large instances, such as 2, 3, 5, 7, and 11, with more than 800 exams and over 12000 students, tend to be significantly harder, whereas smaller instances like 4, 9, and 12 are comparatively easier.

Room availability also impacts difficulty: instances with very few rooms, 4 and 5, restrict the space of feasible assignments, while those with many rooms, 2, 3, 10, and 11, increase combinatorial complexity for exact solvers. Similarly, the number of **periods** influences scheduling flexibility; instances with long timetabling horizons (7 and 8) are generally easier to place, whereas those with fewer periods (6 and 12) are more constrained.

Finally, instances with large numbers of **period** and **room hard constraints**, particularly 3 and 11, impose rigid scheduling requirements that make them difficult for both heuristic and optimization-based approaches.

Overall, the ITC2007 instances form a heterogeneous benchmark, with differences in scale and constraint density leading to distinct feasibility challenges and solution behaviors across algorithms.

Instance	# students	# exams	# rooms	# periods	# Period HC	# Room HC
1	7891	607	7	54	12	0
2	12 743	870	49	40	8	2
3	16 439	934	48	36	82	15
4	5045	273	1	21	20	0
5	9253	1018	3	42	27	0
6	7909	242	8	16	22	0
7	14 676	1096	15	80	28	0
8	7718	598	8	80	20	1
9	655	169	3	25	10	0
10	1577	214	48	32	58	0
11	16 439	934	40	26	83	15
12	1653	78	50	12	9	7

Table 5.1: Overview of instances with student, exam, room, period, period hard constraints, and room hard constraints details.

5.2. Monte Carlo Tree Search results

To achieve results for the instances, the algorithm performs multiple iterations to create and explore the tree in search of an optimal solution. We conducted 5 independent runs for each instance. As seen in Table 5.2, there is a variation in the number of iterations for the 12 instances, since larger instances require more computational time for each iteration. The values in the table represent the average number of iterations over the 5 runs. The values presented in the subsequent tables are the best values obtained from these 5 runs.

Instance	# iterations
1	333
2	171
3	132
4	4017
5	108
6	7351
7	69
8	295
9	13 655
10	5293
11	141
12	127 451

Table 5.2: Average number of iterations for each benchmark instance of ITC2007 from 5 independent runs.

In larger instances, the number of iterations is smaller due to the large number of variables, meaning the search space remains only partially explored. As a result, the balance between exploration and exploitation is not fully realized, and with fewer iterations, the algorithm can rely disproportionately on early simulations that can lead to suboptimal solutions.

In Table 5.3, the results obtained through the MCTS approach are presented, comparing our values to those of the finalists of the ITC2007 examination track. In Table 5.4 we compare our results to some of the newer state-of-the-art approaches.

Instance	Müller 2009 [22]	Gogos et al. 2012 [24]	Atsuta et al. 2007 [26]	De Smet 2008 [32]	Pillay 2008 [33]	Proposed MCTS Approach
1	**4370**	5905	8006	6670	12 035	28 031
2	**400**	1008	3470	623	3074	33 500
3	**10 049**	13 862	18 622	–	15 917	–
4	**18 141**	18 674	22 559	–	23 582	31 518
5	**2988**	4139	4714	3847	6860	108 876
6	**26 950**	27 640	29 155	27 815	32 250	53 457
7	**4213**	6683	10 473	5420	17 666	75 256
8	**7861**	10 521	14 317	–	16 184	123 155
9	**1047**	1159	1737	1288	2055	5227
10	16 682	–	15 085	**14 778**	17 724	64 360
11	**34 129**	43 888	–	–	40 535	–
12	5535	–	**5264**	–	6310	9549

Table 5.3: Comparison of the proposed MCTS approach with the ITC2007 finalists. The table reports the objective function (1) values for each method on 12 benchmark instances. The comparison is made between the best values obtained from 5 independent runs of our proposed method and the best values reported by the other approaches. The best solution of any instance are in boldface. "–" indicates that a feasible solution could not be obtained.

Instance	Burke & Bykov 2016 [27]	Bykov & Petrovic 2016 [28]	Burke & Bykov 2017 [29]	Proposed MCTS Approach
1	3691	**3647**	3787	28 031
2	**385**	**385**	402	33 500
3	**7359**	7487	7378	–
4	**11 329**	11 779	13 278	31 518
5	2482	**2447**	2491	108 876
6	25 265	**25 210**	25 461	53 457
7	3608	**3563**	3589	75 256
8	6818	**6614**	6701	123 155
9	**902**	924	997	5227
10	**12 900**	12 931	13 013	64 360
11	**22 875**	23 784	22 959	–
12	5107	**5097**	5234	9549

Table 5.4: Comparison of the proposed MCTS approach with state-of-the-art approaches. The table reports the objective function (1) values for each method on 12 benchmark instances. The comparison is made between the best values obtained from 5 independent runs of our proposed method and the best values reported by the other approaches. The best solution of any instance is in boldface. "–" indicates that a feasible solution could not be obtained.

The poor results compared to the other approaches can be attributed to several limitations. First, unlike in games where MCTS benefits from clear feedback of a win or loss, feedback in timetabling is more complex. This is because it involves multiple soft constraints that affect the cost of the solution, which can vary and change during intermediate decisions, thereby affecting later decisions. The branching factor in timetabling is very high due to the large number of exams, periods, and rooms, which makes it computationally expensive to explore. These factors collectively limit the advantages of the algorithm and can help explain the weaker performance values.

5.3. AMPL+`Gurobi` results

AMPL has a key strength in its flexibility for problem solving with a wide range of solvers, including `CPLEX`, `Gurobi`, `MINOS`, among others. For this examination timetabling problem that involves many hard and soft constraints, `Gurobi` stands out as one of the best solvers due to its state-of-the-art performance in MIP solving. Because of this, `Gurobi` was selected as the solver for our mathematical model to create timetables.

Despite the computational power of `Gurobi`, many ITC2007 instances did not reach optimality within the allotted time. In particular, `Gurobi` returned two model status codes for the different instances: status code `402` and status code `472`.

Instances 2, 4, 5, 6, 10, and 12 triggered code 402, which indicates Gurobi was unable to distinguish between infeasibility and unboundedness during its presolve phase. This usually occurs when a model contains variables with loose bounds or when constraints do not restrict the feasible region. However, this warning did not prevent the solver from producing valid feasible solutions, suggesting the issue was likely due to conservative presolver behavior rather than a problem with model formulation.

Instances 1, 3, 7, 8, 9, and 11 had the code 472, which indicates that the solver was still working but had to halt due to the imposed time limit.

In Table 5.5, the results obtained through AMPL+Gurobi approach are presented, comparing our values with those of the ITC2007 finalists of the examination track. In Table 5.6, we compare our results to some of the newer state-of-the-art approaches.

Instance	**Müller** 2009 [22]	**Gogos et al.** 2012 [24]	**Atsuta et al.** 2007 [26]	**De Smet** 2008 [32]	**Pillay** 2008 [33]	**Proposed** AMPL+Gurobi Approach
1	**4370**	5905	8006	6670	12 035	–
2	400	1008	3470	623	3074	**17**
3	**10 049**	13 862	18 622	–	15 917	–
4	**18 141**	18 674	22 559	–	23 582	21 570
5	**2988**	4139	4714	3847	6860	55 577
6	**26 950**	27 640	29 155	27 815	32 250	56 570
7	**4213**	6683	10 473	5420	17 666	–
8	**7861**	10 521	14 317	–	16 184	–
9	**1047**	1159	1737	1288	2055	–
10	16 682	–	15 085	**14 778**	17 724	53 249
11	**34 129**	43 888	–	–	40 535	–
12	5535	–	**5264**	–	6310	5923

Table 5.5: Comparison of the proposed AMPL+Gurobi approach with ITC2007 finalists. The table reports the objective function (1) values for each method on 12 benchmark instances. The comparison is made between the best values of each approach. The best solution of any instance are in boldface. "–" indicates that a feasible solution could not be obtained.

Despite the powerful capabilities of Gurobi, obtaining an optimal solution for the ITC2007 instances remains challenging, confirming the difficulty of this problem. This is combined with the large-scale nature of the instances and the presence of both hard and soft constraints, which must be carefully modeled. These factors are detrimental to a MIP approach that requires significant computational time and can help explain the difficulties in finding feasible timetables.

The results obtained demonstrate the challenges of applying exact optimization methods to large-scale examination timetabling problems. Out of the 12 instances, feasible solutions were only obtained for 5, with the remaining 7 instances failing to produce any feasible solution within the time constraints. Among the instances where solutions were

Instance	Burke & Bykov 2016 [27]	Bykov & Petrovic 2016 [28]	Burke & Bykov 2017 [29]	Proposed AMPL+Gurobi Approach
1	3691	**3647**	3787	–
2	385	**385**	402	**17**
3	**7359**	7487	7378	–
4	**11 329**	11 779	13 278	21 570
5	2482	**2447**	2491	55 577
6	25 265	**25 210**	25 461	56 570
7	3608	**3563**	3589	–
8	6818	**6614**	6701	–
9	**902**	924	997	–
10	**12 900**	12 931	13 013	53 249
11	**22 875**	23 784	22 959	–
12	5107	**5097**	5234	5923

Table 5.6: Comparison of the proposed AMPL+Gurobi approach with state-of-the-art approaches. The table reports the objective function (1) values for each method on 12 benchmark instances. The comparison is made between the best values of each approach. The best solution of any instance is in boldface. "–" indicates that a feasible solution could not be obtained.

found, the performance varies considerably, with one particularly anomalous result. Instance 2 produced a value of 17, which is extraordinarily low compared to all benchmark approaches. While this result was initially met with suspicion, a comprehensive review of the model and its output confirmed that the solution is valid and adheres to all constraints. This dramatic difference is probably due to having few rooms available for large exams, but large capacity in small rooms and a low number of conflicts between exams. The Gurobi solver, known for its robustness, found a solution that, while mathematically sound, is dramatically different from all others. This outcome highlights a unique and currently unexplained property of instance 2 that warrants deeper analysis. Instance 12 demonstrated more credible competitive performance with a value of 5923, which, while not optimal, remains within a reasonable range compared to state-of-the-art methods.

5.4. FCUP instances

5.4.1 Data preparation

The raw FCUP institutional data required preprocessing to conform to the ITC2007 instance format, which serves as the standard benchmark for timetable evaluation in this thesis. This subsection describes the data transformations required to transform the university's enrollment records into the required format.

Initially, enrollment records were organized in a student-centric format, where each row represented a student in a single course. To comply with the ITC2007 format, this information was restructured to group students by course. Each course was then associated

with a list of enrolled students and is now treated as an examination.

The definition of periods was guided by the official academic calendar of FCUP. Specifically, the periods created were extracted from the designated Normal Exam Period listed in the institutional calendar. Each valid weekday within this range was divided into two distinct time slots–one in the morning and the other in the afternoon–thereby creating a two-period daily structure. For weekends, Sundays were entirely excluded, while Saturdays were limited to a single morning time slot. This constraint reflects the institutional preference to minimize weekend usage. To further reinforce this preference, Saturday periods were assigned a specific SoftPeriod weight to discourage selection unless necessary.

The extraction of room information from institutional spreadsheets was less complicated, with the only caution coming from handling cases where rooms had different capacities for regular classes and for examinations. When multiple capacity values were available, the standard examination capacity was prioritized to ensure realistic room usage scenarios.

The weights associated with the soft constraints were defined by the author and do not reflect explicit institutional guidelines. The specific weights applied are shown in Table 5.7.

Constraint	**Weight**
TwoInARow	9
TwoInADay	5
NonMixedDurations	10
PeriodSpread	4
FrontLoad	5
SoftPeriod	50
SoftRoom	0

Table 5.7: Overview of FCUP soft constraints defined weights.

As such, some of these values may not yet align with the preferences of the institution. For example, the penalty associated with the spread of exams between periods may be overestimated relative to the institution's tolerance.

At this stage of the thesis, the institution did not impose hard constraints on rooms or periods. However, as mentioned in 6.1, in future work we suggest to incorporate other constraints to more accurately reflect real-world administrative requirements.

The final counts of students, exams, available periods, and rooms are presented in Table 5.8.

Instance	# students	# exams	# rooms	# periods
1	2696	228	63	23
2	2684	236	63	23

Table 5.8: Overview of instances with student, exam, room and period details.

Since FCUP's requirements include the possibility of assigning multiple rooms to a single exam, the existing approaches developed for ITC2007 mentioned in Sections 2.3.1 and 2.3.2 could not be applied to these instances. However, the actual timetable implemented by FCUP for these instances provides a real-world benchmark for comparison.

5.4.2 Monte Carlo Tree Search results

We applied the MCTS approach to the FCUP instances to evaluate its performance on real-world institutional data. Each instance had the same amount of time as an ITC2007 instance to find an optimal solution. We have in Table 5.9 the average number of iterations from 5 independent runs. In Table 5.10, we have all soft constraint values, as explained in Section 3.7.2, from the best timetable obtained.

Instance	# iterations
1	5123
2	5314

Table 5.9: Average number of iterations for each FCUP instance from 5 independent runs.

Constraint	Instance 1	Instance 2
# TwoInARow	2313	2412
# TwoInADay	0	0
# NonMixedDurations	0	0
# PeriodSpread	2083	2720
# FrontLoad	395	400
# SoftPeriod	1700	1400
# SoftRoom	0	0

Table 5.10: Overview of each individual soft constraint from two FCUP instances, solved with the MCTS approach.

5.4.3 AMPL+`Gurobi` results

As shown in Table 5.11, the AMPL+`Gurobi` approach significantly outperformed MCTS across all soft constraints, constructing timetables with fewer violations. It showed considerable improvements in all constraints, with a special emphasis on the near-optimal

values of the FrontLoad penalty, while also reducing the use of the Saturdays slots to just 4% in instance 1 and 3% in instance 2.

Constraint	Instance 1	Instance 2
# TwoInARow	648	803
# TwoInADay	0	0
# NonMixedDurations	0	0
# PeriodSpread	832	908
# FrontLoad	0	5
# SoftPeriod	400	300
# SoftRoom	0	0

Table 5.11: Overview of constraint violation values of each individual soft constraint from two FCUP instances, solved with `Gurobi`.

The superior performance of AMPL+`Gurobi` can be attributed to the smaller scale and more structured nature of the FCUP instances when compared to the ITC2007 benchmark instances. The solver was stopped due to the time limit, and the best solution found within that limit is reported. Additionally, the linear progression of the academic curriculum results in many students sharing identical exam enrollments, which allows the student grouping optimization described in the mathematical model to significantly reduce instance sizes and improve solution quality.

5.4.4 Comparison with FCUP manual timetables

To provide a comprehensive evaluation of our algorithmic approaches, we obtained the actual timetables manually created by FCUP administrative staff for the same examination periods covered in our instances. The manual timetables were evaluated using the same penalty weights described in Section 5.4.1.

From this evaluation, we found that the manually created timetables contain hard constraint violations, which render them infeasible according to our problem specifications. Specifically, both manual timetables exhibit conflicting exam assignments, where students are scheduled for multiple exams within the same time period. Instance 1 contains 20 such conflicts, while Instance 2 has 6 conflicting assignments. In contrast, both MCTS and AMPL+`Gurobi` approaches successfully generated feasible timetables with zero hard constraint violations.

Despite the feasibility issues, analyzing the soft constraint performance of manual timetables provides insight into human scheduling priorities. The manual timetables were evaluated using the penalty weights described in Section 5.4.1. Table 5.12 presents the breakdown of soft constraint violations for manually created timetables.

Constraint	Instance 1	Instance 2
# TwoInARow	945	864
# TwoInADay	55	30
# NonMixedDurations	0	0
# PeriodSpread	1258	1623
# FrontLoad	290	300
# SoftPeriod	500	1000
# SoftRoom	0	0

Table 5.12: Overview of constraint violation values of each individual soft constraint from two FCUP instances, manually created by institutional staff.

Table 5.13 provides a direct comparison of the total constraint violation values across all three approaches applied to the FCUP instances.

Approach	Instance 1	Instance 2
Manual (FCUP Staff)	3048	3817
MCTS	6491	6932
AMPL+`Gurobi`	1880	2016

Table 5.13: Comparison of total constraint violation values between manual and algorithmic approaches for FCUP instances.

The results from Table 5.13 present a trade-off between feasibility and soft constraint optimization. While the manual timetables achieve better soft constraint performance than MCTS, they come at the cost of violating hard constraints, which represents an unacceptable compromise in formal optimization terms. AMPL+`Gurobi` demonstrates the clear advantage of mathematical optimization, achieving both feasibility and superior soft constraint performance. With constraint violation values of 1880 and 2016, it outperforms both the manual approach and MCTS while maintaining zero hard constraint violations. These results suggest that human schedulers may prioritize certain soft constraints at the expense of feasibility. Institutional staff likely focused on practical considerations such as student convenience and resource utilization, thus breaking some hard constraints and creating scheduling conflicts. The superior performance of algorithmic approaches demonstrates their value not only in maintaining feasibility but also in achieving better overall optimization outcomes. This analysis suggests that institutions could benefit from

adopting algorithmic scheduling tools, both to ensure feasible timetables and to improve student experience through better soft constraint satisfaction.

6. Conclusion

In this thesis, a MCTS-based algorithmic approach was developed to solve an examination timetabling problem, with the main objective of creating a tool capable of producing high-quality timetables using the ITC2007 problem instances as benchmarks for evaluation. As a secondary objective, a mathematical formulation was developed and implemented in AMPL to provide a comparative analysis with the heuristic MCTS method.

Due to the reasons described in Chapter 5, MCTS did not achieve strong results on the ITC2007 instances. This outcome was expected, given the limitations of the algorithm when applied to problems characterized by an extremely high branching factor and a vast, unstructured search space. Similarly, the approach based on mathematical optimization encountered significant scalability issues as the problem size increased.

When comparing our results with those of Müller [22] or state-of-the-art approaches, it becomes clear that the timetabling community has favored combining constructive heuristics, such as graph coloring, with local search techniques like SA and TS.

However, the application to real-world institutional data from FCUP revealed that the mathematical programming approach with AMPL+`Gurobi` proved highly effective for this smaller exam timetabling problem, significantly outperforming MCTS. This finding shows that exact optimization methods can be particularly suitable for academic environments with organized curriculum progressions and predictable enrollment patterns, where student grouping can effectively reduce the computational burden.

Overall, this thesis highlights the critical role of choosing algorithms that can effectively handle the combinatorial complexity inherent in large-scale timetabling problems. AMPL proved to be an unsuitable choice for ITC2007 instances, as it was better suited for the smaller FCUP instances. On the other hand, MCTS offers interesting research directions due to its flexibility and generality. However, practical success continues to rely on algorithms capable of balancing between exploitation and exploration, enabling efficient navigation of the vast and highly constrained search space.

6.1. Future work

Future work involves testing different approaches to improve the results of the implemented MCTS approach. Some of these approaches may include the use of different

domain-specific heuristics to improve the use of computational resources, which would allow for a greater number of iterations within the same time frame, thereby increasing the chances of discovering good solutions. Specifically, these new heuristics would aim to improve the initial search phase of the algorithm, guiding the exploration more effectively from the root node.

Regarding the secondary objective of this thesis–the adaptation of the developed approaches to help FCUP automate their exam scheduling process–the current model has limitations. Despite the transformation of FCUP's internal data into the ITC2007 format, the current model does not accommodate several institution-specific constraints, such as:

- The ability to specify for any exam whether it must be scheduled in a computer-equipped room or if any standard room suffices.
- The introduction of more flexible soft constraints, particularly in cases where students are repeating exams, so that first-time examinees are given scheduling priority in terms of fairness.

Incorporating these and more requirements would improve the applicability of the MCTS approach for FCUP and real-world settings.

As discussed in Section 5.3 future work could focus on creating several new instances that are nearly identical to instance 2 but with minor, controlled modifications. By systematically observing how these small changes affect the final objective value, we could test the hypothesis that the low value is due to the unique combination of few large rooms and a low number of conflicts.

References

[1] R. Qu, E. Burke, B. Mccollum, L. Merlot, and S. Lee, "A survey of search methodologies and automated system development for examination timetabling," *J. Scheduling*, vol. 12, pp. 55–89, 02 2009. [Cited on pages 1, 2, 4, and 6.]

[2] S. Ceschia, L. Di Gaspero, and A. Schaerf, "Educational timetabling: Problems, benchmarks, and state-of-the-art results," *European Journal of Operational Research*, vol. 308, pp. 1–18, 07 2022. [Cited on pages 1, 12, and 13.]

[3] E. Siew, S. Sze, S. L. Goh, G. Kendall, N. Sabar, and S. Abdullah, "A survey of solution methodologies for exam timetabling problems," *IEEE Access*, vol. PP, pp. 1–1, 01 2024. [Cited on pages 2, 4, 5, and 6.]

[4] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed. Thomson Brooks/Cole, 2003. [Cited on page 2.]

[5] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024, accessed: 27-03-2025. [Online]. Available: https://www.gurobi.com [Cited on page 2.]

[6] A. Schaerf, "A survey of automated timetabling," *Artificial Intelligence Review*, vol. 13, pp. 87–127, 01 1996. [Cited on page 4.]

[7] M. W. Carter, G. Laporte, and S. Y. Lee, "Examination timetabling: Algorithmic strategies and applications," *The Journal of the Operational Research Society*, vol. 47, no. 3, pp. 373–383, 1996. [Cited on page 4.]

[8] R. Lewis, "A survey of metaheuristic-based techniques for university timetabling problems," *OR Spectrum*, vol. 30, pp. 167–190, 01 2008. [Cited on pages 4 and 6.]

[9] W. Winston and J. Goldberg, *Operations research: applications and algorithms*, 01 2004. [Cited on pages 5 and 6.]

[10] U. Orumie and D. Ebong, "A glorious literature on linear goal programming algorithms," *American Journal of Operations Research*, vol. 04, pp. 59–71, 01 2014. [Cited on page 6.]

[11] M. M. Christian Blum, "Welcome to the metaheuristics network web site," accessed: 07-04-2025. [Online]. Available: https://www.metaheuristics.org/index.html [Cited on page 6.]

[12] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, 06 2009, vol. 74. [Cited on pages 7 and 8.]

[13] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, Third Edition.*, 04 2011, vol. 175. [Cited on page 7.]

[14] F. Glover and M. Laguna, *Tabu search I*, 01 1999, vol. 1. [Cited on page 8.]

[15] D. Gunter, "New optimization heuristics: The great deluge algorithm and the record-to-record travel," *Journal of Computational Physics*, vol. 104, no. 1, pp. 86–92, 1993. [Cited on page 8.]

[16] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science (New York, N.Y.)*, vol. 220, pp. 671–80, 06 1983. [Cited on page 8.]

[17] G. Post, L. Di Gaspero, J. Kingston, B. Mccollum, and A. Schaerf, "The third international timetabling competition," *Annals of Operations Research*, vol. 239, 02 2013. [Cited on page 9.]

[18] T. Müller, H. Rudová, and Z. Müllerová, "Real-world university course timetabling at the international timetabling competition 2019," *Journal of Scheduling*, vol. 28, pp. 247–267, 04 2024. [Cited on page 9.]

[19] D. Van Bulck and D. Goossens, "The international timetabling competition on sports timetabling (itc2021)," *European Journal of Operational Research*, 11 2022. [Cited on page 9.]

[20] B. Mccollum, P. Mcmullan, E. Burke, A. Parkes, and R. Qu, "The second international timetabling competition: Examination timetabling track," 10 2007. [Cited on pages 9 and 18.]

[21] B. McCollum, "Welcome to the home page of itc2007, the second international timetabling competition," 2007, accessed: 25-10-2024. [Online]. Available: https://www.eeecs.qub.ac.uk/itc2007/index.htm [Cited on page 11.]

[22] T. Müller, "Itc2007 solver description: A hybrid approach," *Annals of Operations Research*, vol. 172, pp. 429–446, 01 2008. [Cited on pages 11, 35, 37, and 44.]

[23] T. Müller and H. Rudová, "Conflict-based statistics," 01 2004. [Cited on page 11.]

[24] C. Gogos, P. Alefragis, and E. Housos, “An improved multi-staged algorithmic process for the solution of the examination timetabling problem,” *Annals of Operations Research*, pp. 1–19, 01 2010. [Cited on pages 11, 35, and 37.]

[25] T. A. Feo and M. G. Resende, “Greedy randomized adaptive search procedures,” *Journal of Global Optimization*, vol. 6, pp. 109–133, 03 1995. [Cited on page 11.]

[26] M. Atsuta, K. Nonobe, and T. Ibaraki, “Itc2007 track 2: An approach using general csp solver,” accessed: 05-04-2025. [Online]. Available: https://www.eeecs.qub.ac.uk/itc2007/winner/bestcoursesolutions/Atsuta_et_al.pdf [Cited on pages 11, 12, 35, and 37.]

[27] E. Burke and Y. Bykov, “An adaptive flex-deluge approach to university exam timetabling,” *INFORMS Journal on Computing*, vol. 28, pp. 781–794, 11 2016. [Cited on pages 12, 36, and 38.]

[28] Y. Bykov and S. Petrovic, “A step counting hill climbing algorithm applied to university examination timetabling,” *Journal of Scheduling*, vol. 19, pp. 479–492, 08 2016. [Cited on pages 12, 13, 36, and 38.]

[29] E. Burke and Y. Bykov, “The late acceptance hill-climbing heuristic,” *European Journal of Operational Research*, vol. 258, pp. 70–78, 07 2016. [Cited on pages 13, 36, and 38.]

[30] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. [Cited on pages 13 and 14.]

[31] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, “Monte carlo tree search: a review of recent modifications and applications,” *Artificial Intelligence Review*, vol. 56, pp. 2497–2562, 07 2022. [Cited on page 14.]

[32] G. De Smet, “Itc2007 - examination track,” 01 2008. [Online]. Available: http://www.cs.qub.ac.uk/itc2007/winner/bestexamsolutions/Geoffrey_De_smet_examination_description.pdf [Cited on pages 35 and 37.]

[33] N. Pillay. (2007) A developmental approach to the examination timetabling problem. Accessed: 27-04-2025. [Online]. Available: http://www.cs.qub.ac.uk/itc2007/winner/bestexamsolutions/pillay.pdf [Cited on pages 35 and 37.]

A. ITC2007 framework

This appendix provides an overview of the `itc2007_framework` developed to tackle the Examination Timetabling Track from the Second International Timetabling Competition. The framework is designed to follow a layered architecture that promotes modularity and scalability, with each layer encapsulating specific responsibilities to facilitate a clear separation and clarity within the system. The assortment, dependencies, and main classes of each layer can be seen in Figure A.1.

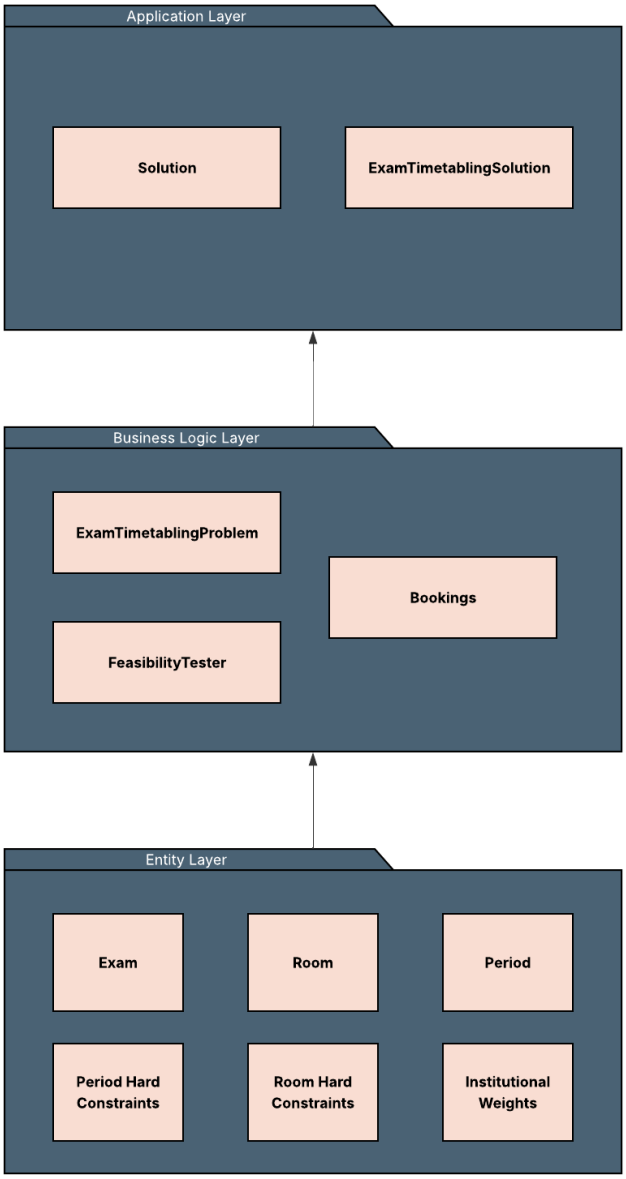

Figure A.1: Overview of the layers that compose the framework.

A.1. Entity layer

The Entity Layer serves as the foundation of the framework by describing the fundamental components and data structures used throughout the system. Each class models an object relevant to the exam timetabling track, as defined in the datasets. The classes are:

- exam.py

  This module models individual exams to be scheduled. Each `Exam` object has an identifier, the duration of the exam, a list of students enrolled in the exam, and a boolean to identify if the exam has an exclusive constraint.

```
class Exam:
    number: int
    duration: int
    students: List[str]
    exclusive: bool  = False
```

- room.py

  Defines the `Room` class, which represents the physical spaces where exams are to be conducted. Its key properties include the room capacity and its penalty.

```
class Room:
    number: int
    capacity: int
    penalty: int
```

- period.py

  The `Period` class models time slots in which exams can be scheduled. Each period has information on the date and time, as well as its duration and the associated penalty.

```
class Period:
    number: int
    date: date
    time: time
    duration: int
    penalty: int
```

- period_hard_constraint.py

  This module encapsulates constraints related to periods. For each constraint, it stores both the exams affected by it and the type of constraint.

```
class PeriodHardConstraint:
    exam_one: int
    constraint_type: str
    exam_two: int
```

- room_hard_constraint.py

  Same as the module above, `RoomHardConstraint` class defines the constraints related to room usage.

```
class RoomHardConstraint:
    exam_number: int
    constraint_type: str
```

- institutional_weighting.py

  This module captures the soft constraint weights imposed by the institution. Unlike hard constraints, that must always be satisfied, these weights affect the quality of the solution.

```
class InstitutionalWeighting:
    weightingType: str
    paramOne: int
    paramTwo: int = -1
    paramThree: int = -1
```

A.2. Business layer

The Business Layer encapsulates the core logic responsible for defining, managing, and validating. As the framework's operational core, it interprets the data structures from the Entity layer and enforces the rules and constraints that permit the creation of legitimate timetables. This layer ensures that solutions not only exist but also follow the hard and soft constraints defined by the institution.

- booking.py

The `Booking` class represents an assignment of an exam to a specific period and room. It functions as the atomic unit of scheduling and is used throughout the solution-building process.

```
class Booking:
    exam: Exam
    period: Period
    rooms: List[Room]
```

- exam_timetabling_problem.py

  The `ExamTimetablingProblem` class serves as the central data model within the Business Layer. It consolidates all the structural components defined in the Entity Layer. This class encapsulates the entire problem instance and provides utility methods to extract, relate, and validate information essential to the problem-solving process.

  When the framework receives a file path to an instance of the problem, the class method `from_file` begins by parsing structural metadata from the file. It first calls `read_information`, which reads the file line by line to determine how many entries exist for each component. With this metadata, the method calculates the line indices of each section. It then uses a set of static methods to extract the actual data and construct the corresponding objects. Once all lists are built, the method instantiates and returns an `ExamTimetablingProblem` object populated.

```
def from_file(cls, file_path):
    lines_size = cls._read_information(file_path)
    with open(file_path, 'r') as file:
        lines = file.readlines()

    exam_lines = int(lines_size.get("Exams", 0))
    starting_exam_line = 1
    period_lines = int(lines_size.get("Periods", 0))
    starting_period_line = 2 + exam_lines
    room_lines = int(lines_size.get("Rooms", 0))
    starting_room_line = 3 + exam_lines + period_lines
    periodHard_lines = int(lines_size.get("PeriodHardConstraints", 0))
    starting_periodHard_line = 4 + exam_lines + period_lines + room_lines
    roomHard_lines = int(lines_size.get("RoomHardConstraints", 0))
```

```
    starting_roomHard_line = 5 + exam_lines + period_lines + room_lines +
    periodHard_lines
    weight_lines = int(lines_size.get("InstitutionalWeightings", 0))
    starting_weight_line = 6 + exam_lines + period_lines + room_lines +
    periodHard_lines + roomHard_lines

    exams = cls._read_exams(lines, starting_exam_line, exam_lines)
    periods = cls._read_periods(lines, starting_period_line, period_lines)
    rooms = cls._read_rooms(lines, starting_room_line, room_lines)
    period_hard_constraints = cls._read_period_hard_constraints(lines,
    starting_periodHard_line, periodHard_lines)
    room_hard_constraints = cls._read_room_hard_constraints(lines,
    starting_roomHard_line, roomHard_lines)
    institutional_weightings = cls._read_institutional_weightings(lines,
    starting_weight_line, weight_lines)

    return cls(exams, periods, rooms, period_hard_constraints,
    room_hard_constraints, institutional_weightings)
```

Once the data is loaded, the constructor begins by storing these inputs directly in the corresponding instance attribute, then it constructs several secondary structures to support the scheduling logic:

- Room-Period Availability

  A dictionary is initialized to represent all possible room-period combinations, with each pair initially marked as available by being set to `False`. This structure enables constant-time lookup when checking whether a room is already occupied in a given period.

```
def dictionary_room_period(self):
    return {(room, period): False for room in self.rooms for period in
    self.periods}
```

- Period Capacities

  The dictionary `period_capacity` maps each `Period` object to the total available room capacity across all rooms. This structure is used to validate whether a given period can physically accommodate an exam or not.

```
def calculate_period_capacities(self) -> Dict[Period, int]:
    period_capacities = {}
    for period in self.periods:
        total_capacity = sum(room.capacity for room in self.rooms)
        period_capacities[period] = total_capacity
    return period_capacities
```

– Clash Matrix

The `clash_matrix` is a two-dimensional NumPy array with size n, where n is the number of exams. Each element in the matrix stores the number of students enrolled in both exams. Initially, the matrix is filled with zeros, and then a nested loop iterates over all exam pairs to populate it. This matrix serves as the core structure to detect conflicts.

```
    num_exams = len(exams)
    self.clash_matrix = np.zeros((num_exams, num_exams), dtype=int)

    for i, exam_one in enumerate(exams):
        for j, exam_two in enumerate(exams):
            if i != j:
                self.clash_matrix[i, j] = len(set(exam_one.students) &
    set(exam_two.students))
```

– Exclusion Constraint

This method updates the `clash_matrix` by incrementing the relevant cells for all pairs of exams that are explicitly forbidden from being scheduled at the same time.

```
def type_has_exams(self, period_constraint: str) -> List[
    PeriodHardConstraint]:
    return [constraint for constraint in self.period_hard_constraints
    if constraint.constraint_type == period_constraint]

def exclusion_in_matrix(self):
    constraints = self.type_has_exams("EXCLUSION")

    for constraint in constraints:
```

```
            self.clash_matrix[constraint.exam_one, constraint.exam_two] +=
        1
9           self.clash_matrix[constraint.exam_two, constraint.exam_one] +=
        1
```

- Exclusive Room Constraint

  The method `exams_exclusive` flags any exam that is subject to exclusive room usage by scanning the room hard constraint and marking the corresponding exam object by setting their `exclusive` attribute to `True`.

```
1   def exams_exclusive(self):
        for constraint in self.room_hard_constraints:
3           self.exams[constraint.exam_number].set_exclusive()
```

- feasibility_tester.py

  The `FeasibilityTester` class acts as a validation component that is responsible for determining whether a given assignment of an exam to a period and room respects all hard constraints. Upon initialization, it receives an instance of `ExamTimetabling-Problem` to access the clash matrix, constraints and structural definitions directly.

  - Period Feasibility

    This method determines whether a given exam can be assigned to a given period. First, the method retrieves all exams that are linked to the given exam via the Coincidence constraint and ensures that all such exams have durations less than or equal to the period's length, and if any exam exceeds it, then the assignment is rejected. Next, it verifies that if any of the coincident exams are already scheduled, then they must be assigned to the same period, if there is a conflicting period, then feasibility fails. The method then checks the `clash_matrix` to ensure that the given exam does not share students with any exam already scheduled in the same period, while also implicitly checking for the Exclusion constraint since they are encoded in the matrix. Lastly, it verifies that any defined ordering constraints are respected. If the exam is required to occur after another that is already scheduled, the candidate period must be strictly later, or vice-versa, with any violation leading to infeasibility. This method only return `True` if it passes all these checks.

```
def feasible_period(self, solution: Solution, assign_exam: Exam, period
    : Period) -> bool:
    exams = self.problem.exams_with_coincidence(assign_exam)
    if not all(exam.duration <= period.duration for exam in exams):
        return False

    for exam in exams:
        if exam == assign_exam:
            continue
        if solution.period_from(exam) != period and solution.room_from(
    exam) != None:
            return False

    for exam in solution.exams_from_period(period):
        if(self.problem.clash_matrix[exam.number][assign_exam.number] >
     0):
            return False

    for period_constraint in self.problem.exams_with_type("AFTER",
    assign_exam.number):
        if period_constraint.exam_two == assign_exam.number and
    solution.period_from(self.problem.exams[period_constraint.exam_one
    ]) != None and solution.period_from(self.problem.exams[
    period_constraint.exam_one]).number <= period.number:
            return False
        if period_constraint.exam_one == assign_exam.number and
    solution.period_from(self.problem.exams[period_constraint.exam_two
    ]) != None and solution.period_from(self.problem.exams[
    period_constraint.exam_two]).number >= period.number:
            return False

    return True
```

– Room Feasibility

This method tests whether a particular room can feasibly host a given exam during a specific period. First, it computes the remaining room capacity after subtracting the total number of students already scheduled on the same period and room, and if the number of students in the given exam exceeds this

value, then the room is deemed unfit. If the exam has an Exclusive constraint, it must be allocated the entire room, for this, the method ensures that the remaining capacity is exactly equal to the room's full capacity, if not, the room is left unused. The method further checks whether any other exams already scheduled in the room at the given period are subject to the exclusivity constraint, if yes, the room cannot be shared, and feasibility is denied. Similar to the period feasibility method, it returns `True` only if all these checks are passed.

```
def feasible_room(self, solution: Solution, assign_exam: Exam, period:
    Period, room: Room) -> bool:
    capacity = self.current_room_capacity(solution, period, room)
    if len(assign_exam.students) > capacity:
        return False

    if self.problem.room_exclusivity(assign_exam) and capacity != room.
    capacity:
        return False

    for constraint in self.problem.room_hard_constraints:
        if(solution.is_exam_set_to(period, room, self.problem.exams[
    constraint.exam_number])):
            return False

    return True
```

- Rooms Feasibility

  This variant of the room feasibility method performs a similar check but omits the direct student capacity comparison, assuming the exam's size has already been verified in the inclusion of the multiple rooms and focuses only on exclusivity rules and conflicting constraints.

```
def feasible_rooms(self, solution: Solution, assign_exam: Exam, period:
     Period, room: Room) -> bool:
    capacity = self.current_room_capacity(solution, period, room)
    if self.problem.room_exclusivity(assign_exam) and capacity != room.
    capacity:
        return False

```

```
    for constraint in self.problem.room_hard_constraints:
7       if(solution.is_exam_set_to(period, room, self.problem.exams[
    constraint.exam_number])):
            return False
9
    return True
```

A.3. Application layer

The Application Layer is responsible for representing and manipulating solutions. It utilizes the structures and logic provided by the Entity and Business Layers to generate, modify, and evaluate timetables. This layer acts as the interface between the problem definition and the solving process, offering tools to construct and validate timetables.

- solution.py

  The `Solution` class manages the mapping between exams, periods, and rooms, while also providing tools to evaluate the feasibility and quality of the timetable. Upon construction, the `Solution` object prepares two primary data structures, `bookings` is a dictionary that links each exam to a tuple containing its assigned period and either a single room or a list of rooms, and `pre_association` is a nested dictionary that organizes the exams assigned to each period room pair.

```
def __init__(self, problem: ExamTimetablingProblem):
2    self.problem = problem
    self.bookings: Dict[Exam, tuple] = {}
4    self.pre_association: Dict[Period, Dict[Room, List[Exam]]] = {
        period: {room: [] for room in problem.rooms}
6       for period in problem.periods
    }
```

  Exams are assigned through the `set_exam` method, which updates both data structures to maintain synchronization. This method is robust in that it supports both single-room and multi-room assignments by detecting iterability in the `rooms` parameter.

```
1 def set_exam(self, period: Period, rooms, exam: Exam):
    self.bookings[exam] = (period, rooms)
```

```
    if hasattr(rooms, '__iter__'):
        for room in rooms:
            self.pre_association[period][room].append(exam)
    else:
        self.pre_association[period][rooms].append(exam)
```

The class provides several ways to retrieve information on existing assignments, to know what period is assigned to a scheduled exam, the `period_from` method is called, and to know which room or combination of rooms is assigned to an exam, the method `rooms_from` is used.

```
def period_from(self, exam: Exam) -> Period:
    return self.bookings[exam][0] if exam in self.bookings else None

def rooms_from(self, exam: Exam) -> List[Room]:
    return self.bookings[exam][1] if exam in self.bookings else None
```

To examine allocations from the perspective of periods or rooms, the methods `exams_from_period` and `exams_from_period_room`. These methods are built atop the `pre_association` dictionary and allow efficient queries when checking for conflicts or capacity violations, since it is particularly useful when paired with the `FeasibilityTester` that relies on these methods to test the feasibility of an assignment.

```
def exams_from_period_room(self, period: Period, room: Room) -> List[Exam]:
    return self.pre_association[period][room] if period in self.
    pre_association and room in self.pre_association[period] else []

def exams_from_period(self, period: Period) -> List[Exam]:
    exams = []

    if period in self.pre_association:
        for room in self.pre_association[period]:
            exams.extend(self.pre_association[period][room])

    return exams
```

The class offers several scoring interfaces:

- calculate_score - return a global feasibility penalty based on hard constraint violations;
- calculate_score_periods - returns a feasibility penalty based only on hard constraint violations involving periods;
- calculate _softs - returns a fitness value based on the soft constraint violations;

To prepare the solutions for evaluation, the method `dictionary_to_list` transforms the `bookings` dictionary into a list of `Booking` objects which are then passed to an instance of `ExamTimetablingSolution` that performs the actual scoring computations.

```
def dictionary_to_list(self) -> List[Booking]:
    return [Booking(exam, period, room) for exam, (period, room) in self.
    bookings.items()]

def calculate_score(self) -> int:
    bookings = self.dictionary_to_list()
    solution = ExamTimetablingSolution(self.problem, bookings)
    return solution.distance_to_feasibility()

def calculate_score_periods(self) -> int:
    bookings = self.dictionary_to_list()
    solution = ExamTimetablingSolution(self.problem, bookings)
    return solution.distance_to_feasibility_period()

def calculate_softs(self) -> int:
    bookings = self.dictionary_to_list()
    solution = ExamTimetablingSolution(self.problem, bookings)
    return solution.soft_constraint_violations()
```

- exam_timetabling_solution.py

  The `ExamTimetablingSolution` class is responsible for evaluating a timetable; it encapsulates both the current set of `Booking` instances and provides methods for calculating hard and soft constraint violations.

  The class offers two methods for computing the number of hard constraint violations the `distance_to_feasibility` and `distance_to_feasibility_period`. The

former includes room constraints, while the latter focuses only on period-based violations.

```
def distance_to_feasibility(self) -> int:
    return (
        self.conflicting_exams() +
        self.overbooked_periods() +
        self.too_short_periods() +
        self.period_constraint_violations() +
        self.room_constraint_violations()
    )

def distance_to_feasibility_period(self) -> int:
    return (
        self.conflicting_exams() +
        self.too_short_periods() +
        self.period_constraint_violations()
    )
```

- conflicting_exams - Checks for scheduling clashes where two exams that share students are placed in the same period, with each clash increasing the violation count.
- overbooked_periods - Counts the number of bookings where the assigned room or rooms do not have enough capacity to accommodate all the students enrolled in an exam.
- too_short_periods - Identifies assignments where an exam's duration exceeds the available time in the assigned period.
- period_constraint_violations - Checks for the three types of period constraint, with each violation increasing the total.
- room_constraint_violations - Checks exams that are marked with the exclusive constraint if they do not share any room with other exams in the assigned period.

```
def conflicting_exams(self) -> int:
    conflits = 0
    for booking_a in self.bookings:
```

```
        for booking_b in self.bookings:
            if booking_a.exam.number == booking_b.exam.number:
                continue

            period_clash = booking_a.period.number == booking_b.period.
    number
            students_share = self.problem.clash_matrix[booking_a.exam.number
    ][booking_b.exam.number] > 0
            if period_clash and students_share:
                conflits += 1

    return conflits

def overbooked_periods(self) -> int:
    overbooked = 0
    for booking in self.bookings:
        if hasattr(booking.rooms, '__iter__') and not isinstance(booking.
    rooms, str):
            total_capacity = sum(room.capacity for room in booking.rooms)
        else:
            total_capacity = booking.rooms.capacity
        if len(booking.exam.students) > total_capacity:
            overbooked += 1
    return overbooked

def too_short_periods(self) -> int:
    too_short = 0
    for booking in self.bookings:
        if booking.exam.duration > booking.period.duration:
            too_short += 1

    return too_short

def period_constraint_violations(self) -> int:
    period_violations = 0
    for constraint in self.problem.period_hard_constraints:
        booking_one = next((b for b in self.bookings if b.exam.number ==
    constraint.exam_one), None)
```

```
        booking_two = next((b for b in self.bookings if b.exam.number ==
    constraint.exam_two), None)

        if booking_one is None or booking_two is None:
            continue

        if constraint.constraint_type == "EXAM_COINCIDENCE":
            if booking_one.period.number != booking_two.period.number:
                period_violations += 1

        elif constraint.constraint_type == "EXCLUSION":
            if booking_one.period.number == booking_two.period.number:
                period_violations += 1

        elif constraint.constraint_type == "AFTER":
            if booking_one.period.get_datetime() < booking_two.period.
    get_datetime():
                period_violations += 1

    return period_violations

def room_constraint_violations(self) -> int:
    room_violations = 0
    for constraint in self.problem.room_hard_constraints:
        if constraint.constraint_type == "ROOM_EXCLUSIVE":
            booking = next((b for b in self.bookings if b.exam.number ==
    constraint.exam_number), None)
            if booking is None:
                continue

            if hasattr(booking.rooms, '__iter__') and not isinstance(booking
    .rooms, str):
                rooms_to_check = booking.rooms
            else:
                rooms_to_check = [booking.rooms]

            not_alone = False
            for room in rooms_to_check:
                for b in self.bookings:
```

```
                    if b.exam.number != booking.exam.number and b.period.
    number == booking.period.number:
                        if hasattr(b.rooms, '__iter__') and not isinstance(b
    .rooms, str):
                            if any(room.number == other_room.number for
    other_room in b.rooms):
                                not_alone = True
                                break
                        else:
                            if room.number == b.rooms.number:
                                not_alone = True
                                break
                if not_alone:
                    break
            if not_alone:
                room_violations += 1

    return room_violations
```

The `soft_constraint_violations` method computes the total soft constraint penalty of the timetable, returning the cumulative value from all the individual soft constraint checks.

```
def soft_constraint_violations(self) -> int:
    return (
        self.two_in_a_row_penalty() +
        self.two_in_a_day_penalty() +
        self.period_spread_penalty() +
        self.mixed_durations_penalty() +
        self.frontload_penalty() +
        self.period_penalty() +
        self.room_penalty()
    )
```

- two_in_a_row_penalty - Calculates a penalty when students are scheduled for two exams in consecutive periods of the same day, with each shared student between exams adding a weighted penalty.

- two_in_a_day_penalty - Similar case to two_in_a_row_penalty but instead of exams being scheduled to consecutive periods, they only have to be scheduled to the same day.
- frontload_penalty - Applies a penalty if large exams, as defined by the institution, are scheduled too late in the timetable.
- mixed_durations_penalty - Identifies and penalizes cases where exams of varying durations are scheduled to the same room and period.
- period_spread_penalty - Penalizes tight clustering of exams for the same student within a short number of periods.
- room_penalty - Sums up the room-specific soft penalties associated with each exam's assigned room or combination of rooms.
- period_penalty - Adds penalties based on the period-specific weights associated with each exam's assigned period.

```
def two_in_a_row_penalty(self) -> int:
    row_penalty = 0
    weighting = next((w for w in self.problem.institutional_weightings if w.weightingType == "TWOINAROW"), None)
    if weighting is None:
        return 0

    for i, booking_a in enumerate(self.bookings):
        for booking_b in self.bookings[i+1:]:

            in_a_row = abs(booking_a.period.number - booking_b.period.number) == 1
            same_day = booking_a.period.date == booking_b.period.date

            if in_a_row and same_day:
                row_penalty += weighting.paramOne * self.problem.clash_matrix[booking_a.exam.number][booking_b.exam.number]

    return row_penalty

def two_in_a_day_penalty(self) -> int:
    day_penalty = 0
```

```
    weighting = next((w for w in self.problem.institutional_weightings if w.weightingType == "TWOINADAY"), None)
    if weighting is None:
        return 0

    for i, booking_a in enumerate(self.bookings):
        for booking_b in self.bookings[i+1:]:

            not_in_a_row = abs(booking_a.period.number - booking_b.period.number) != 1
            same_day = booking_a.period.date == booking_b.period.date
            if not_in_a_row and same_day:
                day_penalty += weighting.paramOne * self.problem.clash_matrix[booking_a.exam.number][booking_b.exam.number]

    return day_penalty

def frontload_penalty(self) -> int:
    load_penalty = 0
    weighting = next((w for w in self.problem.institutional_weightings if w.weightingType == "FRONTLOAD"), None)
    if weighting is None:
        return 0

    largest_exams = sorted(self.problem.exams, key=lambda e: len(e.students), reverse=True)[:weighting.paramOne]

    last_periods_index = max(0, len(self.problem.periods) - weighting.paramTwo)
    last_periods = self.problem.periods[last_periods_index:]

    for exam in largest_exams:
        exam_booked = next((b for b in self.bookings if b.exam.number == exam.number), None)
        if exam_booked is None:
            continue

        if any(p.number == exam_booked.period.number for p in last_periods):
            load_penalty += weighting.paramThree
```

```
52
        return load_penalty
54
    def mixed_durations_penalty(self) -> int:
56      mixed_penalty = 0
        weighting = next((w for w in self.problem.institutional_weightings if w.
        weightingType == "NONMIXEDDURATIONS"), None)
58      if weighting is None:
            return 0
60
        for period in self.problem.periods:
62          for room in self.problem.rooms:
                room_period_bookings = []
64              for booking in self.bookings:
                    if booking.period.number == period.number:
66                      if hasattr(booking.rooms, '__iter__') and not isinstance
        (booking.rooms, str):
                            if any(r.number == room.number for r in booking.
        rooms):
68                              room_period_bookings.append(booking)
                        else:
70                          if booking.rooms.number == room.number:
                                room_period_bookings.append(booking)
72              if not room_period_bookings:
                    continue
74
                unique_durations = {b.exam.duration for b in
        room_period_bookings}
76              num_unique_durations = len(unique_durations)
                mixed_penalty += (num_unique_durations - 1) * weighting.paramOne
78      return mixed_penalty

80  def period_spread_penalty(self) -> int:
        spread_penalty = 0
82      weighting = next((w for w in self.problem.institutional_weightings if w.
        weightingType == "PERIODSPREAD"), None)
        if weighting is None:
84          return 0
```

```
86      for booking_a in self.bookings:
            for booking_b in self.bookings:
88              spread = booking_b.period.number - booking_a.period.number
                if spread <= 0:
90                  continue

92              if spread <= weighting.paramOne:
                    spread_penalty += self.problem.clash_matrix[booking_a.exam.
    number][booking_b.exam.number]
94      return spread_penalty

96  def room_penalty(self) -> int:
        r_penalty = 0
98      for booking in self.bookings:
            if hasattr(booking.rooms, '__iter__') and not isinstance(booking.
    rooms, str):
100             r_penalty += sum(room.penalty for room in booking.rooms)
            else:
102             r_penalty += booking.rooms.penalty
        return r_penalty
104
    def period_penalty(self) -> int:
106     p_penalty = 0
        for booking in self.bookings:
108         p_penalty += booking.period.penalty
        return p_penalty
```

B. AMPL implementation

B.1. AMPL model

In this section, the complete AMPL implementation of the mathematical model is presented. For a clearer understanding of the variables used, a detailed mapping is provided in Table B.1.

The AMPL model begins by defining the set of entities necessary, their parameters and the institutional weights discussed above.

```
set EXAM;

param sizeExam {EXAM} >= 0;
param durationExam {EXAM} > 0;
param examFL {EXAM} binary;

set STUDENT;

param sizeStudent {STUDENT} > 0;

set STUDENT_EXAMS {STUDENT} within EXAM;

set DURATION;

param duration {EXAM, DURATION} binary;

set PERIOD;

param durationPeriod {PERIOD} > 0;
param periodFL {PERIOD} binary;
param weightPeriod {PERIOD} >= 0;

param same_day {PERIOD, PERIOD} binary;

set ROOM;

param sizeRoom {ROOM} > 0;
param weightRoom {ROOM} >= 0;
```

```
29

    set AFTER within {EXAM, EXAM};
31  set COINCIDENCE within {EXAM, EXAM};
    set EXCLUSION within {EXAM, EXAM};
33

    set EXCLUSIVE within EXAM;
35

    # Weighting parameters
37  param TWOINAROW;
    param TWOINADAY;
39  param PERIODSPREAD := 1;
    param NONMIXEDDURATIONS;
41  param FRONTLOAD;
    param gPERIODSPREAD;
```

Afterwards the primary and secondary variables are defined to mirror the decision variables in Section 3.5.

```
    # Primary Variables
2   var X_P {EXAM, PERIOD} binary;
    var X_R {EXAM, ROOM} binary;
4   var MI_R {EXAM, ROOM} >= 0, <= 1;

6   # Secondary Variables for Penalties
    var C_2R {STUDENT} >= 0;
8   var C_2D {STUDENT} >= 0;
    var C_PS {STUDENT} >= 0;
10  var C_NMD >= 0;
    var C_FL >= 0;
12  var C_P >= 0;
    var C_R >= 0;
```

The objective statement is a direct transcription of Equation 1.

```
1   minimize Total_Penalty:
    sum {s in STUDENT} (TWOINAROW * C_2R[s] + TWOINADAY * C_2D[s] + PERIODSPREAD
    * C_PS[s]) * sizeStudent[s]
3   + NONMIXEDDURATIONS * C_NMD
```

```
    + FRONTLOAD * C_FL
5   + C_P + C_R;
```

The constraint block contains a translation of Equations 2-22. Hard constraints are grouped first.

```
1   # (2)
    s.t. AssignToOneRoom {i in EXAM}:
3       sum {r in ROOM} X_R[i, r] >= 1;
    # (3)
5   s.t. AssignToOnePeriod {i in EXAM}:
        sum {p in PERIOD} X_P[i, p] >= 1;
7   # (4)
    s.t. RoomCapacity {p in PERIOD, r in ROOM}:
9       sum {i in EXAM} sizeExam[i] * X_P[i, p] * MI_R[i, r] <= sizeRoom[r];
    # (5)
11  s.t. PeriodDuration {p in PERIOD, i in EXAM}:
        durationExam[i] * X_P[i, p] <= durationPeriod[p];
13  # (6)
    s.t. NoStudentConflict {p in PERIOD, s in STUDENT}:
15      sum {i in STUDENT_EXAMS[s]} X_P[i, p] <= 1;
    # (7)
17  s.t. Precedence {(i, j) in AFTER, p in PERIOD, q in PERIOD: q >= p}:
        X_P[i, p] + X_P[j, q] <= 1;
19  # (8)
    s.t. Coincidence {(i, j) in COINCIDENCE, p in PERIOD}:
21      X_P[i, p] = X_P[j, p];
    # (9)
23  s.t. Exclusion {(i, j) in EXCLUSION, p in PERIOD}:
        X_P[i, p] + X_P[j, p] <= 1;
25  # (10)
    s.t. HardRoomConstraints {i in EXCLUSIVE, j in EXAM, p in PERIOD, r in ROOM:
    j != i}:
27      (X_P[i, p] + X_R[i, r] + X_P[j, p] + X_R[j, r]) <= 3;
    # (11)
29  s.t. MI_RUsage {i in EXAM, r in ROOM}:
        MI_R[i, r] <= X_R[i, r];
31  # (12)
    s.t. MI_count {i in EXAM}:
```

```
33          sum {r in ROOM} MI_R[i, r] == 1;
```

Followed by the soft constraints.

```
1       # (13)
        s.t. TwoInARowPenalty {s in STUDENT}:
3           C_2R[s] = sum {i in STUDENT_EXAMS[s], j in STUDENT_EXAMS[s], p in PERIOD,
         q in PERIOD: j != i && q = p + 1 && same_day[p, q] = 1}
                        X_P[i, p] * X_P[j, q];
5       # (14)
        s.t. TwoInADayPenalty {s in STUDENT}:
7           C_2D[s] = sum {i in STUDENT_EXAMS[s], j in STUDENT_EXAMS[s], p in PERIOD,
         q in PERIOD: j != i && q > p + 1 && same_day[p, q] = 1}
                        X_P[i, p] * X_P[j, q];
9       # (15)
        s.t. PeriodSpreadPenalty {s in STUDENT}:
11          C_PS[s] = sum {i in STUDENT_EXAMS[s], j in STUDENT_EXAMS[s], p in PERIOD,
         q in PERIOD: j != i && p < q && q <= p+gPERIODSPREAD}
                        X_P[i, p] * X_P[j, q];
13
        # Auxiliary variables for C_NMD calculations
15      var U_D {DURATION, PERIOD, ROOM} binary;
        # (18)
17      var C_NMD_pr {PERIOD, ROOM} >= 0;

19      # (16)
        s.t. Define_U_D {d in DURATION, i in EXAM, p in PERIOD, r in ROOM: duration[i
        , d] = 1}:
21          U_D[d, p, r] >= X_P[i, p] + X_R[i, r] - 1;
        # (17)
23      s.t. NonMixedDurationsPenalty {p in PERIOD, r in ROOM}:
            1 + C_NMD_pr[p, r] >= sum {d in DURATION} U_D[d, p, r];
25      # (19)
        s.t. Total_NonMixedDurations_Penalty:
27          C_NMD = sum {p in PERIOD, r in ROOM} C_NMD_pr[p, r];
        # (20)
29      s.t. FrontLoadPenalty:
            C_FL = sum {i in EXAM, p in PERIOD} examFL[i] * periodFL[p] * X_P[i, p];
31      # (21)
```

```
    s.t. SoftPeriodPenalty:
33      C_P = sum {p in PERIOD, i in EXAM} weightPeriod[p] * X_P[i, p];
    # (22)
35  s.t. SoftRoomPenalty:
        C_R = sum {r in ROOM, i in EXAM} weightRoom[r] * X_R[i, r];
```

Table B.1: Mapping Between Mathematical Model and AMPL Implementation.

Mathematical Notation	**Code Variable**	**Description**
E	`EXAMS`	Set of all exams.
s_i^E	`sizeExam[i]`	Number of students in exam i.
d_i^E	`durationExam[i]`	Duration of exam i.
f_i^E	`examFL[i]`	Boolean to indicate if exam i is subject to frontload penalties.
D	`DURATION`	Set of all durations used in exams.
u_{id}^D	`U_D`	Boolean to indicate if exam i has *duration type* d.
S	`STUDENT_EXAMS`	Set of student groups with identical exam sets.
n_s	`sizeStudent[s]`	Number of students being represented.
R	`ROOMS`	Set of rooms.
s_r^R	`sizeRoom[r]`	Size of room r.
w_r^R	`weightRoom[r]`	Weight that specifies the penalty for room r.
P	`PERIODS`	Set of periods.
d_p^P	`durationPeriod[p]`	Duration of period p.
f_p^P	`periodFL[p]`	Boolean to indicate if period p is subject to frontload penalties.
w_p^P	`weightPeriod[p]`	Weight that specifies the penalty for period p.
y_{pq}	`same_day[p][q]`	Boolean to indicate if periods p and q are in the same day.
H^{aft}	`AFTER`	Set of pairs of exams that must occur one after the other.

Continued on next page

Table B.1 – continued from previous page

Mathematical Notation	Code Variable	Description
H^{coin}	COINCIDENCE	Set of pairs of exams that must occur in the same period.
H^{excl}	EXCLUSION	Set of pairs of exams that must not occur in the same period.
H^{sole}	EXCLUSIVE	Set of exams that must be the sole occupiers of the rooms allocated.
w^{2R}	TWOINAROW	Weight for having two exams in consecutive periods.
w^{2D}	TWOINADAY	Weight for having two exams in the same day.
w^{PS}	PERIODSPREAD	Weight for having multiple exams within a limited number of periods.
w^{NMD}	NONMIXEDDURATIONS	Weight for having mixed exam durations in the same room and period.
w^{FL}	FRONTLOAD	Weight for scheduling large exams in the final periods.
g^{PS}	gPERIODSPREAD	The preferred minimal gap between exams for a student.
X_{ip}^{P}	X_P	Boolean to indicate if exam i is in period p.
X_{ir}^{R}	X_R	Boolean to indicate if exam i is in room r.
Y_{ir}^{R}	MI_R	Indicates the proportion of exam i in room r.
C_s^{2R}	C_2R	Two exams in consecutive periods penalty for student representative s.
C_s^{2D}	C_2D	Two exams in the same day penalty for student representative s.
C_s^{PS}	C_PS	Multiple exams within a limited number of periods penalty for student representative s.
C^{NMD}	C_NMD	Mixed exams durations penalty.

Continued on next page

Table B.1 – continued from previous page

Mathematical Notation	Code Variable	Description
C^{FL}	`C_FL`	Large exams in the final periods penalty.
C^{P}	`C_P`	Room usage penalty for using rooms with an associated weight.
C^{R}	`C_R`	Period usage penalty for using periods with an associated weight.