An interactive tool for supporting university timetabling

Daniela Tomás
Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2025

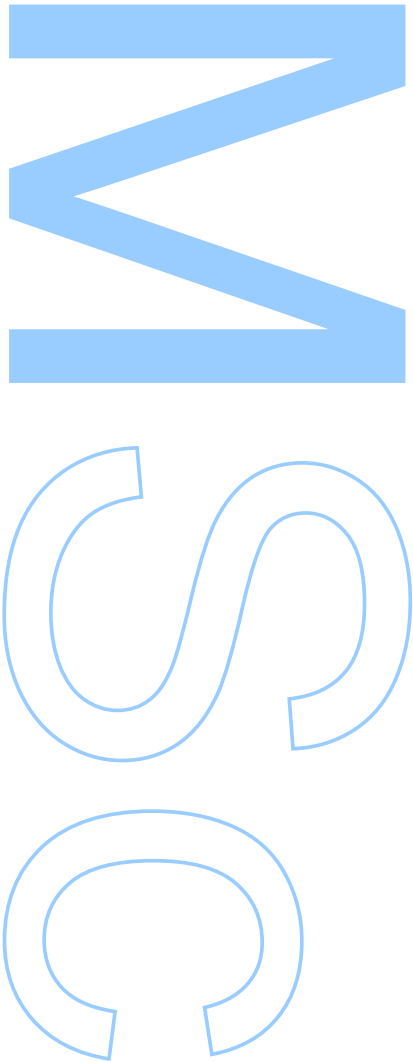
Orientador
Prof. Dr. João Pedro Pedroso, Faculdade de Ciências

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

Acknowledge ALL the people!

Resumo

A geração de horários para escolas e universidades é um problema de otimização combinatória NP-difícil, caracterizado por um espaço de pesquisa vasto e por restrições. Na Faculdade de Ciências da Universidade do Porto (FCUP), a elaboração dos horários a cada semestre para milhares de estudantes, dezenas de currículos e centenas de docentes é, atualmente, um processo manual, demorado e sujeito a erros.

Esta dissertação aborda a variante Curriculum-Based Course Timetabling (CB-CTT) do University Course Timetabling Problem, apresentando uma abordagem híbrida inovadora que combina Monte Carlo Tree Search (MCTS) com Hill Climbing (HC). O MCTS conduz a pesquisa global, simulando e avaliando múltiplas atribuições de evento, período e sala, priorizando os eventos mais restritos e expandindo os ramos mais promissores. Sempre que o MCTS encontra, durante a fase de simulação, um horário viável melhorado, o HC assume o controlo e aplica seis movimentos de vizinhança para refinar iterativamente a solução, enquanto preserva a sua viabilidade.

Testes com as instâncias do terceiro track da ITC-2007 demonstram que o híbrido MCTS-HC consegue encontrar consistentemente horários viáveis, apesar da sua performance permanecer abaixo dos melhores resultados publicados…

Apesar de algumas melhorias e mudanças introduzidas no algoritmo MCTS padrão, continuam a existir desafios...

Palavras-chave: University Course Timetabling Problem, Curriculum-based Course Timetabling, Monte Carlo Tree Search, Hill Climbing, ITC-2007

Abstract

Timetable generation for schools and universities is an NP-hard combinatorial optimization problem with a vast search space and constraints. At the Faculty of Sciences of the University of Porto (FCUP), preparing semester schedules for thousands of students, dozens of curricula, and hundreds of lecturers is currently manual, time-consuming, and prone to errors.

This dissertation addresses the Curriculum-Based Course Timetabling (CB-CTT) variant of the University Course Timetabling Problem by presenting a novel hybrid approach that combines Monte Carlo Tree Search (MCTS) with Hill Climbing (HC). MCTS drives the global search, simulating and evaluating multiple event, period, and room assignments, prioritizing the most constrained events and expanding the most promising branches. Whenever MCTS obtains an improved feasible timetable during simulation, HC takes over and applies six neighborhood moves to iteratively improve the solution while preserving the feasibility.

Tests on the ITC-2007 track 3 instances show that the MCTS-HC hybrid consistently finds feasible timetables, but its performance remains below the best published results...

Despite some improvements and changes to the standard MCTS algorithm, challenges remain...

Keywords: University Course Timetabling Problem, Curriculum-based Course Timetabling, Monte Carlo Tree Search, Hill Climbing, ITC-2007

Table of Contents

List of Figures vi
List of Abbreviations vii

1. Introduction 1
   1.1. Motivation 1
   1.2. Objectives 2
   1.3. Methodology and Contributions 2
   1.4. Dissertation layout 2
2. Background 4
   2.1. Combinatorial Optimization Problems 4
   2.2. University Course Timetabling Problem 4
      2.2.1. Curriculum-Based and Post-Enrollment Course Timetabling 5
      2.2.2. Constraints 5
   2.3. International Timetabling Competition (ITC) 6
   2.4. Monte Carlo Tree Search 6
      2.4.1. Phases 7
      2.4.2. Properties 8
   2.5. Local Search 8
      2.5.1. Hill Climbing 8
   2.6. Previous Work 9
3. Related Work 11
   3.1. Operational Research (OR) 11
   3.2. Metaheuristics 11
      3.2.1. Single Solution-Based 12
      3.2.2. Population-Based 12
   3.3. Hyperheuristics 12
   3.4. Multi-Objective 13
   3.5. Hybrid 13
   3.6. Monte Carlo Tree Search 14
4. Development 15
   4.1. Problem Formulation 15
      4.1.1. Constraints 16
   4.2. Monte Carlo Tree Search 16
      4.2.1. Search Space 16

4.2.2. MCTS Tree 17
4.2.2.1. Tree Nodes 17
4.2.3. Events Allocation 18
4.2.4. Periods and Rooms Allocation 19
4.2.4.1. Initial Periods and Rooms Allocation Approach 19
4.2.4.2. Final Periods and Rooms Allocation Approach 20
4.2.5. Methodology 21
4.2.5.1. Selection 21
4.2.5.2. Expansion 22
4.2.5.3. Simulation 24
4.2.5.4. Backpropagation 26
4.2.6. Normalization 26
4.3. Hill climbing 27
4.3.1. Neighborhoods 27
4.3.2. Methodology 28
4.4. Diving 28
4.5. Front-end 29
5. Tests 30
5.1. Test Instances 30
5.2. Test Setup 30
5.3. Performance Metrics 31
5.4. Testing Procedure 32
6. Results 33
6.1. Python vs PyPy Performance 33
6.2. Feasibility 33
6.3. C Parameter Behaviour 33
6.4. Diving strategy 33
6.5. Competition Results Comparison 33
7. Conclusion 34

Bibliography 34

List of Figures

2.1. Hard and soft constraints example 6
2.2. MCTS approach 7
2.3. Previous work interface 10
2.4. Previous work UML 10

4.1. MCTS tree 17
4.2. MCTS steps 21
4.3. Diving approach 29

List of Abbreviations

CB-CTT Curriculum-based Course Timetabling. 1, 4

COP Combinatorial Optimization Problem. 3, 13

FCUP Faculty of Sciences of the University of Porto. 1, 4, 15

HC Hill Climbing. 2, 3, 6, 7, 11, 12, 14, 22, 24, 25

ITC International Timetabling Competition. 5, 11

ITC-2007 Second International Timetabling Competition. 5, 12, 15, 24, 27, 28

MCTS Monte Carlo Tree Search. iv, 1– 3, 5, 13, 14, 16, 24, 27, 28

PATAT International Conference on the Practice and Theory of Automated Timetabling. 5

PE-CTT Post-Enrollment Course Timetabling. 4, 5, 13

SA Simulated Annealing. 6, 11, 12

TS Tabu Search. 6, 11, 13

UCT Upper Confidence Bounds for Trees. 6, 19

UCTTP University Course Timetabling Problem. 1– 4, 10, 11, 13

1. Introduction

University Course Timetabling Problem (UCTTP) is a complex combinatorial optimization problem that consists of allocating events, rooms, lecturers, and students to weekly schedules while meeting certain constraints. A particular focus of this research is on Curriculum-based Course Timetabling (CB-CTT), a variant of UCTTP where the scheduling process is centered around courses and their associated curricula.

To illustrate the complexity of the problem, consider two courses within the same curriculum: *Introduction to Programming* and *Calculus I*. Each course comes with distinct scheduling demands: the former might require three lectures per week (two theoretical and one lab), while *Calculus I* may demand two lectures (a theoretical and a practical). Each lecture requires specific resources and teaching environments: practical programming lectures typically require a computer lab, while large auditoriums are better suited for theoretical lectures to accommodate more students. Since these lectures cannot overlap, variations in lecture frequency and format add an additional layer of complexity to the scheduling process. Moreover, other specific requirements may arise depending on the institution's policies and the particular needs of courses or lecturers.

The challenge intensifies when considering a complete bachelor's or master's degree, which spans several years and involves numerous courses, each with multiple lectures and unique requirements. Given the scale and complexity of the problem, obtaining an optimal solution in usable time is typically infeasible. Nevertheless, heuristic algorithms have proved capable of producing approximate and high-quality solutions effectively.

1.1. Motivation

The process of building timetables at Faculty of Sciences of the University of Porto (FCUP) each semester is currently time-consuming, not automated, and the final results are not always the most satisfactory. In general, existing tools focus primarily on visualizing timetables or on basic conflict detection. This situation not only results in suboptimal schedules but also increases the workload for administrative staff and impacts the satisfaction of both lecturers and students.

In the academic year 2023/2024, FCUP enrolled more than 5000 students, offered 117 different curricula, and coordinated 646 collaborators [1]. This scale underscores the need

for an effective timetabling system that can manage such scale while minimizing errors and conflicts.

1.2. Objectives

The primary objective of this dissertation is to enhance the efficiency and quality of FCUP's weekly timetable development by providing step-by-step interactive recommendations and detecting potential conflicts during its construction. This functionality must be integrated into a timetable visualization interface that was previously developed using reactive programming.

1.3. Methodology and Contributions

To address the problem challenges, this dissertation proposes a novel hybrid approach combining two heuristic algorithms: Monte Carlo Tree Search (MCTS) and Hill Climbing (HC). The MCTS heuristic search algorithm was chosen, as it has been applied to various optimization problems and has proven to be particularly effective in games. HC was also chosen to be used in conjunction with MCTS for local optimization.

Although MCTS has shown positive results in relevant areas, its application to UCTTP remains largely unexplored. Therefore, the proposed system presents an opportunity to advance the state of the art and help in further studies, by leveraging the global search capabilities of MCTS to explore diverse scheduling possibilities with HC to refine solutions locally. This hybrid approach aims to deliver both feasible and better-quality timetables while addressing the practical challenges faced by FCUP in managing large-scale scheduling.

1.4. Dissertation layout

After the introductory chapter stating the motivation, objectives, methodology, and contributions, the remainder of the dissertation is organized as follows:

- **Chapter 2 - Background:** Introduces the fundamental concepts needed to contextualize and understand the UCTTP, along with an overview of the algorithms used.
- **Chapter 3 - Related Work:** Reviews existing research on UCTTP, discussing various approaches, including operational research, metaheuristics, hyperheuristics, multi-objective, and hybrid methods.

- **Chapter 4 - Development:** Describes the design, implementation, and integration of the proposed hybrid approach.
- **Chapter 5 - Tests:** Details the experimental setup and evaluation metrics used to assess the system's performance.
- **Chapter 6 - Results:** Presents and analyzes the results obtained from the experiments.
- **Chapter 7 - Conclusion:** Summarizes the findings, discusses the contributions, and outlines potential future enhancements.

2. Background

This chapter will cover the concepts needed to contextualize and understand the UCTTP and the algorithms used to obtain the final result, namely the MCTS and HC algorithms.

2.1. Combinatorial Optimization Problems

Finding a solution for maximizing or minimizing a value is common in several real world problems. These problems often require searching for an optimal solution from a finite set of possibilities. They are characterized by their discrete nature and the challenge of finding optimal solutions in large search spaces.

Combinatorial Optimization Problems (COPs) arise in various fields, such as artificial intelligence, machine learning, auction theory, applied mathematics, and so on. Some well-known COPs include Knapsack Problem, Traveling Salesman Problem, and Graph Coloring.

To solve these problems, various algorithmic techniques are used, including:

- **Exact algorithms:** Guarantee optimal solutions by exhaustively exploring the solution space. However, they can be computationally expensive, especially for large and complex problems, because their complexity often grows exponentially.
- **Approximation algorithms:** Produce provably near-optimal solutions, providing a quantifiable measure of how far the solution might diverge from the best possible one, making them valuable for problems where exact solutions are infeasible.
- **Heuristic algorithms:** Focus on practicality and efficiency by finding good solutions within a reasonable time frame. While they do not ensure optimality, they are often effective for solving large-scale problems. This type of algorithms will be the primary focus of this dissertation.

2.2. University Course Timetabling Problem

UCTTP is a NP-hard COP that involves allocating events, rooms, lecturers, and students to weekly schedules while meeting certain predefined constraints. It falls under the broader category of Educational Timetabling, which also includes other challenging problems such as Examination Timetabling.

Due to the size and complexity of the problem, obtaining an optimal solution in usable time is typically infeasible. Therefore, it is necessary to employ robust optimization techniques.

2.2.1 Curriculum-Based and Post-Enrollment Course Timetabling

UCTTP can be categorized into two main types:

- **Curriculum-Based Course Timetabling (CB-CTT):** Courses are grouped into predefined curricula, ensuring that no student or lecturer is allocated to overlapping courses within their curriculum.
- **Post-Enrollment Course Timetabling (PE-CTT):** Events are scheduled after students enroll in their courses, taking into account their preferred event combinations while minimizing conflicts.

These two problems differ significantly in terms of timing, flexibility, and constraints. PE-CTT is performed after the student has enrolled in their courses, while CB-CTT is performed first. PE-CTT adjusts to the student choices, providing greater flexibility, while CB-CTT follows a more rigid and predefined structure. Furthermore, PE-CTT often involves more complex constraints due to diverse student preferences, whereas CB-CTT deals with more predictable patterns. As a result, PE-CTT is appropriate for institutions with a flexible course enrollment, while CB-CTT is ideal for institutions with fixed curricula. FCUP's timetables generation fits into CB-CTT.

2.2.2 Constraints

In the context of UCTTP, constraints are often divided into two different types (Figure 2.1):

- **Hard constraints:** Ensure the feasibility of the timetable and must be strictly satisfied. Typically, a hard constraint includes avoiding overlapping events for the same student or a lecturer.
- **Soft constraints:** Represent preferences to improve the quality of the solution without being mandatory. An example of a soft constraint is minimizing gaps in students' timetables to ensure a more compact timetable.

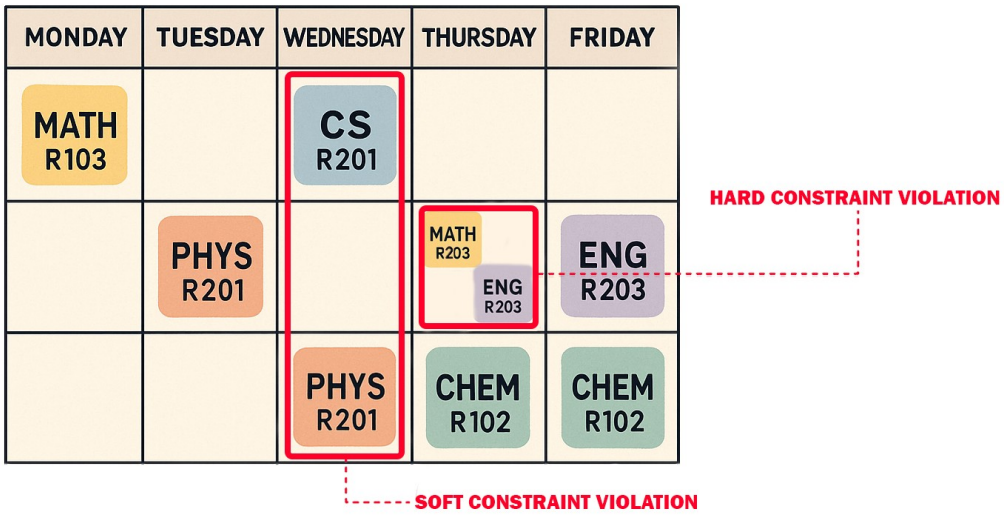


FIGURE 2.1: Hard and soft constraints example

2.3. International Timetabling Competition (ITC)

Since 2002, the International Conference on the Practice and Theory of Automated Timetabling (PATAT)* has supported timetabling competitions to encourage research in this field. There have already been five editions of the International Timetabling Competition (ITC), with three focusing on university timetabling. The third edition, held in 2011, centered on high school timetabling.

The first ITC in 2002 focused on the PE-CTT problem and utilized artificially generated instances. Over successive editions, the competition introduced problem instances that increasingly reflected real-world constraints and complexities.

The Second International Timetabling Competition (ITC-2007)† was structured into three distinct tracks: Examination Timetabling Problem, Post-enrollment Course Timetabling, and Curriculum-based Course Timetabling. The second one is an extended version of the previous competition. This dissertation focuses on the Curriculum-Based Course Timetabling track (track 3) of the ITC-2007, as it aligns with the problem under investigation. Even after 18 years, these benchmark datasets remain widely used in academic research and optimization studies, demonstrating their relevance in the field.

2.4. Monte Carlo Tree Search

MCTS is a decision-making algorithm that has been successfully applied in a variety of optimization problems with a huge search space. The algorithm has proven to be

*https://patatconference.org/communityService.html
†https://www.eeecs.qub.ac.uk/itc2007/

particularly effective in games such as Go and Chess, where the algorithm can even outperform the best human players.

2.4.1 Phases

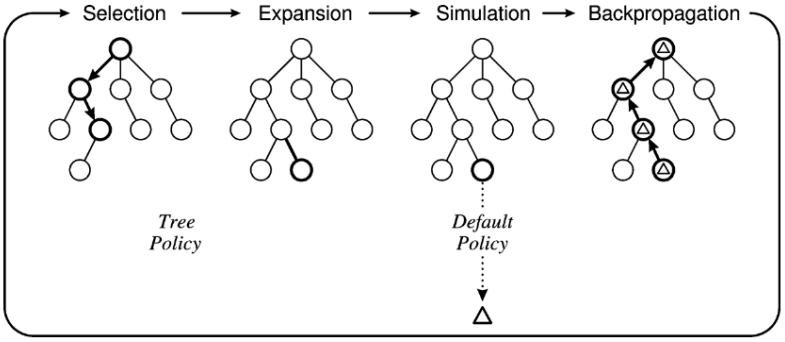


FIGURE 2.2: One MCTS iteration [2]

The algorithm consists of four phases, as illustrated in Figure 2.2 [2]. These phases are repeated for a predefined number of iterations or until a computational budget (such as time or memory) is reached. The phases are as follows:

1. **Selection:** The tree is traversed from the root node until it finds a node that is not completely expanded, i.e., a non-terminal node with unvisited children. The selection is guided by a policy that balances exploration and exploitation. Typically, the policy used is Upper Confidence Bounds for Trees (UCT), which selects nodes that maximize formula 2.1.

$$UCT = \frac{w_i}{n_i} + 2C\sqrt{\frac{2\ln n}{n_i}}, \tag{2.1}$$

   where w_i is the total reward of all playouts through this state, n_i is the number of visits of child node i, C is a constant greater than zero (typically $\sqrt{2}$), and n is the number of visits of the parent node.

2. **Expansion:** One or more child nodes are added to the node previously reached in the selection phase.

3. **Simulation:** From the newly added node(s), a simulation is performed according to a default policy, which may include random moves until a terminal node is reached.

4. **Backpropagation:** The simulation result is then propagated back through the traversed nodes, where the number of visits and the average reward for each node are updated until it reaches the root.

2.4.2 Properties

MCTS has several characteristics that make it especially suitable for solving complex problems [2]:

- **Aheuristic:** Does not require domain-specific heuristics to be effective, which is specially useful in complex problems. While adding some domain knowledge can improve performance, MCTS can still yield good results with simple random simulations.
- **Anytime:** Can return a valid solution at any point during its execution. Since each iteration immediately updates the search tree, the current best move is always available.
- **Asymmetric:** Builds its tree asymmetrically, focusing more on promising areas of the search space. This leads to more efficient exploration and better use of computational resources.

2.5. Local Search

Local search is an optimization technique that iteratively improves a solution within a given search space by making small changes and retaining those that yield better results. It is effective for large-scale problems where the search is computationally expensive or unfeasible. Popular local search algorithms include HC, Simulated Annealing (SA), Tabu Search (TS), and Genetic Algorithms (GA). While local search algorithms are powerful, they can sometimes get stuck and fail to find the global best solution. Techniques such as simulated annealing help escape local optima and explore a broader range of solutions.

2.5.1 Hill Climbing

HC is a local search optimization algorithm that begins with an initial solution, which can be randomly generated or specifically chosen depending on the problem. From this starting point, HC evaluates neighbouring solutions, which are generated by making small modifications to the current solution. If a neighbouring solution is found to be better than the current one, the algorithm moves to that new solution. The algorithm finishes when there is no better neighbouring solution that offers an improvement over the current one, indicating that a local optimum has been attained.

While HC is effective and easy to implement, it has limitations.The algorithm can get stuck in:

- **Local optimum:** A solution state that is better than its neighbours but not necessarily the global optimum, i.e., the best possible solution in the entire search space, restricting further improvement.
- **Plateaus:** Mostly flat regions in the search space where neighbouring solutions have the same value, making it challenging for the algorithm to determine a direction of improvement.
- **Ridges:** A region in the search space where moving in all directions appears to lead downhill, and reaching a better solution often requires a sequence of non-improving moves, which HC does not explore.

To address these issues, variations of the algorithm have been developed:

- **Simple Hill-Climbing:** Evaluates one neighbour at a time and moves to the first improvement found, making it efficient but susceptible to local maxima.
- **Steepest-Ascent Hill-Climbing:** Evaluates all neighbours of the current state and chooses the best among them. This algorithm is a variation of the simple hill-climbing algorithm but takes more time.
- **Stochastic Hill-Climbing**: Randomly selects a neighbour, without evaluation, and moves to it if it improves the solution. This approach is less commonly used than the other two.

2.6. Previous Work

A previous project* developed a timetabling visualization tool using Flask and reactive programming principles with the Elm language. Flask facilitated data management and communication between the front-end and the database (Figure 2.4), ensuring that timetable updates were efficiently processed and displayed.

This tool allow users to manually construct and modify schedules while providing a responsive interface (Figure 2.3). However, despite its usability, the tool has some limitations that this dissertation attempts to address:

*__Front-end:__ `https://github.com/luismdsleite/schedule` __Back-end:__ `https://github.com/luismdsleite/schedule-backend`

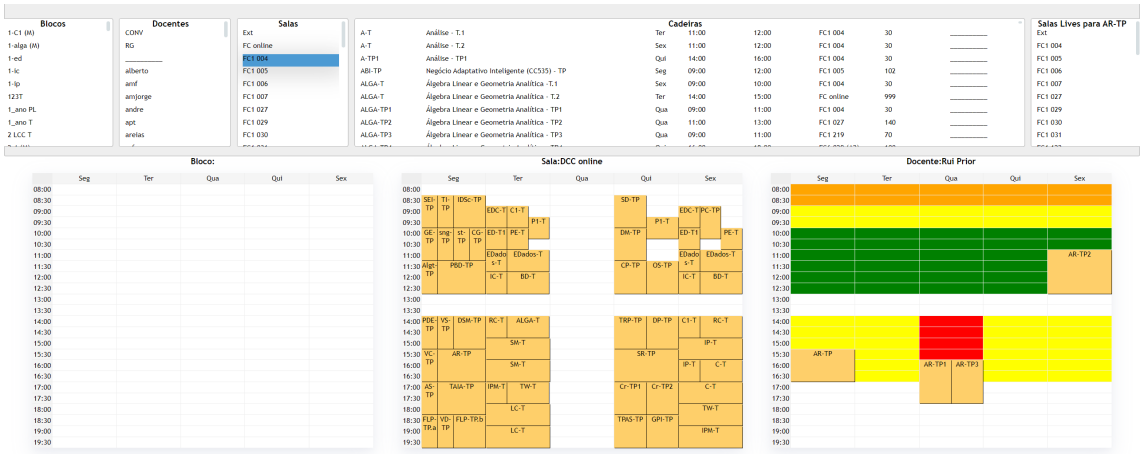


FIGURE 2.3: Previous work interface

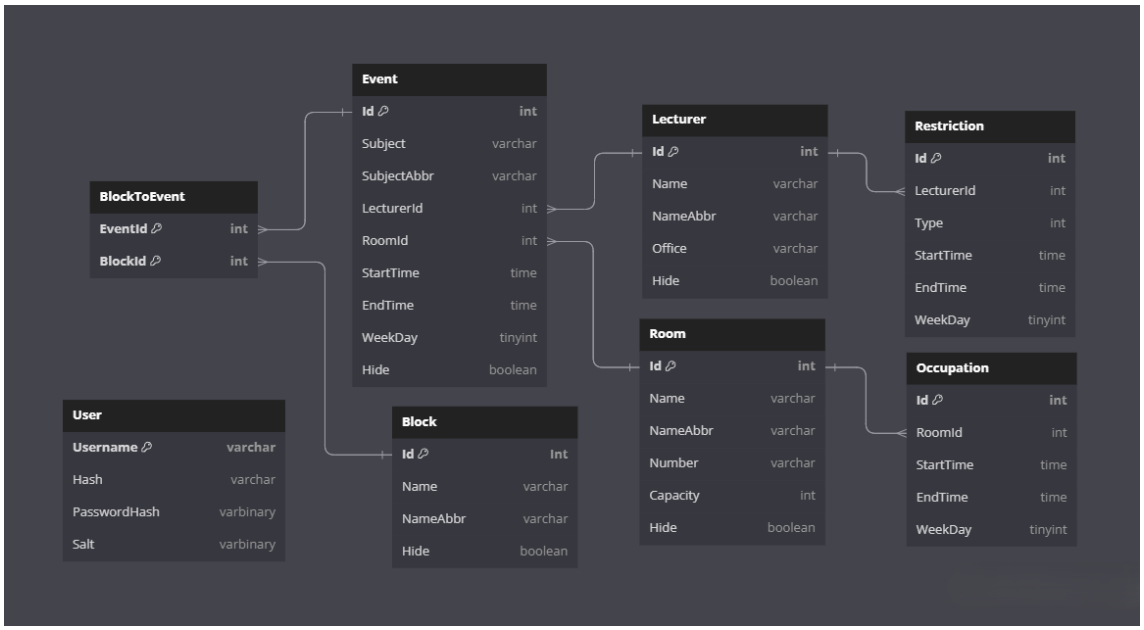


FIGURE 2.4: Previous work UML

- **No conflict detection support:** Users had to manually check for conflicts, increasing the risk of errors.
- **Lack of automated guidance:** It did not provide recommendations or suggestions to help users make optimal scheduling decisions.
- **No quality assessment:** The system lacked mechanisms to evaluate the quality of a given timetable.

3. Related Work

The literature on the UCTTP is vast, so this chapter will cover various approaches to address this problem, mostly based on surveys.

According to Lewis' survey [3], algorithms can be divided into three categories:

- **One-Stage** algorithms use a weighted sum function of constraints to identify solutions that satisfy both hard and soft constraints at the same time;
- **Multi-Stage** algorithms aim to optimize soft constraints while ensuring feasibility;
- **Multi-stage with relaxation** is divided into multiple stages, with some constraints being relaxed while satisfying others.

Other surveys, such as Abdipoor et al. [4], categorized UCTTP into five groups: operational research (or), metaheuristics, hyperheuristics, multi-objective, and hybrid approaches.

3.1. Operational Research (OR)

Operational Research (OR) techniques provide mathematically rigorous solutions. Despite their complexity, these methods are simple to implement and can be considered if time and memory constraints are not a concern [5]. However, the other types of approaches provide better quality results.

As Chen et al. and Babaei et. al detailed in their surveys [5, 6], OR includes Graph Coloring (GC), Integer and Linear Programming (IP/LP) and Constraint Satisfaction(s) Programming (CSPs) based techniques.

3.2. Metaheuristics

Metaheuristics *provide "acceptable" solutions in a reasonable time for solving hard and complex problems* [7].

Metaheuristics are similarly defined by Du et al. as *a class of intelligent self-learning algorithms for finding near-optimum solutions to hard optimization problems* [8].

Metaheuristics, while not guaranteeing global optimality, have become one of the most used solution strategy for UCTTP and can be classified into two categories: Single solution-based and Population-based.

3.2.1 Single Solution-Based

Single solution-based metaheuristics, also known as local search algorithms, focus on modifying a single solution throughout the search in order to improve that solution.

This metaheuristic approach includes algorithms such as HC, SA, TS, and Iterated Local Search (ILS). SA is perhaps the most effective single solution-based metaheuristic, particularly in benchmark datasets, but TS has also been shown to be successful in minimizing hard constraints [4].

3.2.2 Population-Based

Population-based meta-heuristics iteratively improve a population of solutions by generating a new population in the same neighborhood as the existing ones. This method can be subdivided into Evolutionary Algorithms, such as Genetic Algorithms, and Swarm Intelligence, such as Ant Colony Optimization [4, 8].

Population-based metaheuristics, as opposed to single-solution-based metaheuristics, are more focused on exploration rather than exploitation [7, 8]. Despite promising results on real-world datasets, population-based approaches were rarely applied to benchmark datasets and performed poorly in ITC competitions compared to single solution-based methods [4].

3.3. Hyperheuristics

Heuristic methods have been highly effective in solving a wide range of problems. However, their application to new problems is often challenging due to the vast number of parameter tuning and the lack of clear guidance [9]. To address this problem, hyperheuristics aim to generalize methods by selecting or combining the most suitable heuristic(s) for a specific problem, rather than explicitly solving the problem.

Hyper-heuristic and multi-objective approaches are less common, possibly due to their performance [6].

3.4. Multi-Objective

Multi-objective or multi-criteria approaches aim to optimize multiple conflicting objectives simultaneously. These methods are frequently used to find the optimal Pareto front, which is a set of compromise optimal solutions. However, the disadvantage lies in the execution effort [6]. Several multi-criteria algorithms can also be included in other categories, such as metaheuristics.

3.5. Hybrid

Hybrid approaches mix algorithms from two or more of the previously mentioned types of approaches.

Hybrid approaches can indeed be divided into two main categories [4]:

- **Collaborative hybrids:** Involve running different algorithms sequentially, intertwined, or in parallel, sharing information without structural integration. This collaboration enables each component to focus on its area of the problem, with occasional communication guiding the overall search process.
- **Integrative hybrids:** Integrate components of different algorithms into a single framework, creating a cohesive and interdependent method.

In particular, Tomáš Müller, winner of the ITC-2007 tracks 1 and 3, developed a collaborative hybrid approach* for all three tracks [10], finding feasible solutions for all instances of tracks 1 and 3 and most instances of track 2.

The algorithms used included Iterative Forward Search (IFS), HC, Great Deluge (GD), and optionally SA. During the construction phase, IFS is employed to find a complete solution, followed by HC to find the local optimum. When a solution cannot be improved further using HC, GD is used to iteratively decrease, based on a cooling rate, a bound that is imposed on the value of the current solution. Optionally, SA can be used when the bound reaches its lower limit.

The solver's techniques have been integrated into UniTime† system, which is widely used for scheduling in academic institutions. Furthermore, the principles behind Müller's hybrid methodology are still relevant in modern optimization problems.

*https://github.com/tomas-muller/cpsolver-itc2007
†https://www.unitime.org/

While Müller's approach was highly successful in 2007, subsequent research has revealed that other methods may outperform it in some instances. However, it remains a benchmark against which new approaches are often compared.

3.6. Monte Carlo Tree Search

MCTS application to COPs remains relatively unexplored. In the context of UCTTP, Goh's investigation [11] is the only known study that explores the potential of MCTS in solving the PE-CTT.

Goh's investigation employs a two-stage approach, first utilizing MCTS to find initial feasible solutions and then applying local search techniques to enhance the quality of these solutions.

The research introduces several enhancements to the standard MCTS algorithm, including heuristic-based simulations and tree pruning techniques, which improved the MCTS performance and effectiveness. The study also provides a comprehensive comparison of MCTS against traditional methods such as Graph Coloring heuristic and TS, with TS emerging as the most effective method.

One major limitation of MCTS for timetabling is, as highlighted by Goh, its rigorous decision-making process. Events are assigned sequentially and cannot be reassigned, limiting the algorithm's flexibility. In contrast, local search methods, such as TS, allow dynamic reassignment, enabling a more efficient exploration of the search space. Furthermore, the tree-based structure of MCTS restricts its hybridization with local search techniques, which have proven essential in achieving significant results in other learning-based algorithms. Another drawback of MCTS is its high computational cost, making it less effective under the strict time constraints imposed by timetabling problems.

This dissertation aims to demonstrate that, despite its limitations, MCTS still holds potential for timetabling problems with the appropriate modifications and hybridization.

4. Development

This chapter describes the development of the timetable system, formulating the problem, discussing the implementation of the MCTS and HC hybrid approach, detailing the design choices, algorithmic improvements and integration efforts made throughout the project.

4.1. Problem Formulation

The entities in the problem are listed below:

- **Periods,** $P = \{P_0, P_1, ..., P_{|P|-1}\}$**:** Days are divided into fixed timeslots, with periods consisting of a day and a timeslot.
- **Rooms,** $R = \{R_0, R_1, ..., R_{|R|-1}\}$**:** Each room has a capacity and a type.
  - Capacity (number of seats)
- **Lecturers,** $L = \{L_0, L_1, ..., L_{|L|-1}\}$**:**
  - Availability
- **Events,** $E = \{E_0, E_1, ..., E_{|E|-1}\}$**:** Represent the events to be scheduled (usually a lecture). Each event has associated attributes:
  - Capacity (number of students)
  - Number of lectures
  - Minimum working days (days over which the events of the same course should be spread)
  - Available periods
  - Priority
  - Lecturer
  - Period (day, timeslot)
  - Room
- **Blocks,** $B = \{B_0, B_1, ..., B_{|B|-1}\}$**:** A block may represent a group of related events. Usually represents all events from the same curriculum, which is a set of courses in a study program.

4.1.1 Constraints

There are several hard and soft constraints that affect the creation of FCUP timetables. The following hard and soft constraints were selected, drawing inspiration from those of ITC-2007:

Hard constraints:

- **H1:** Events belonging to the same course must be scheduled and must be assigned to distinct periods.
- **H2:** Two events can be scheduled in the same room, but only if they are in different periods.
- **H3:** Events of the same curricula, or taught by the same lecturer, must be scheduled in different periods.
- **H4:** If a lecturer is unavailable in a given period, then no events can be taught by this lecturer in that period.

Soft constraints:

- **S1:** Events of the same course should be spread into the given minimum number of days.
- **S2:** Events belonging to the same curriculum should be in consecutive periods.
- **S3:** The capacity of the room should be higher or equal than the capacity of the event.
- **S4:** All events of a course should be given in the same room.

The problem is defined as a maximization problem, with the algorithm attempting to minimize soft constraint violations while strictly adhering to hard constraints.

4.2. Monte Carlo Tree Search

4.2.1 Search Space

The search space, S, for this problem is large as it involves a product of all possible event-period-room combinations:

$$S = E \cdot P \cdot R,$$

where E is the set of events, P is the set of periods (day, timeslot), and R is the set of rooms.

To reduce the search space, unavailable periods and unavailable rooms for a chosen period are discarded immediately in the MCTS $expansion$ phase.

4.2.2 MCTS Tree

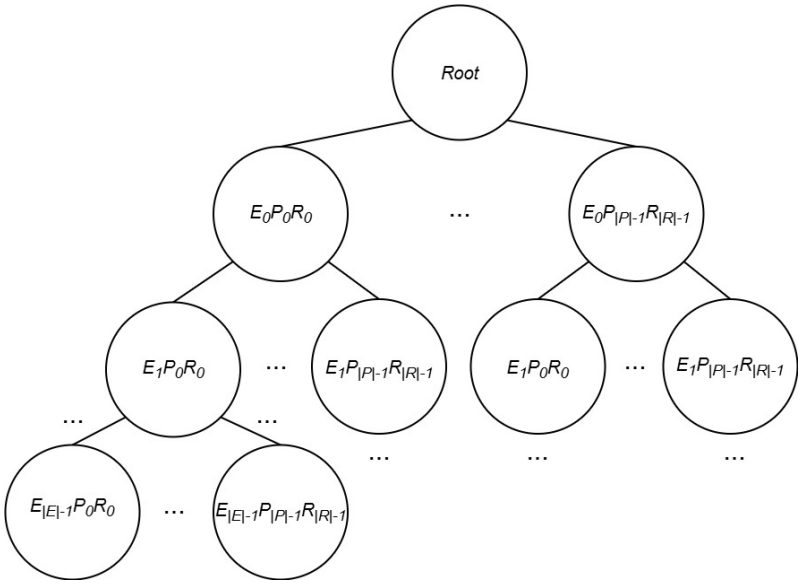


FIGURE 4.1: MCTS tree

Figure 4.1 illustrates the basic structure of the MCTS tree. The root represents the initial state, where no events have been assigned. The first level corresponds to assigning the first event, E_0, to various P and R. Each child node represents a specific assignment $((E_0, P_0, R_0), ..., (E_0, P_{|P|-1}, R_{|R|-1}))$. Subsequent levels correspond to the sequential assignment of events $E_1, E_2, ..., E_{|E|-1}$.

4.2.2.1 Tree Nodes

A node is composed of the following attributes:

- A **path** is a dictionary that represents the sequence of actions leading to a node. The root is the only node that has an empty path.
- A **parent** is a reference to the parent node, allowing the tree structure to be navigated upwards. The root is the only node that does not have a parent.
- **Children** is initialized as an empty list at first but will later hold references to the node's child nodes.

- The **expansion limit** specifies the maximum number of children that a node can have, which helps control the expansion process. It is estimated based on the number of available rooms per period for the next event. A value of zero indicates that the node cannot be expanded further.
- **Visits** is initialized to 0, tracking the number of times a node has been visited.
- The **hard and soft scores** are both initialized to 0, representing the cumulative hard and soft scores for a node, which are used to evaluate the quality of the node's state.
- The **best hard and soft penalty results** are initialized to negative infinity, representing the best hard and soft penalty results encountered for a node. These attributes will be updated during the algorithm as better results are found.

4.2.3 Events Allocation

Events are sorted in advance so that the most difficult event to place is placed first in the tree. The priority of an event (Formula 4.1) is calculated based on:

- The difference between the number of lectures and minimum working days to prioritize events with more lectures spread over fewer days;
- The number of available periods, as having more available periods indicates fewer scheduling restrictions;
- The capacity of an event, which implies that events with greater capacity have higher priority;
- The number of curricula in which the event is, as events that appear in more curricula are more difficult to allocate.

If two or more events have the same priority score, they are ordered randomly.

$$\begin{aligned} Priority = (\#lectures - \#min\_working\_days) \cdot 4 \\ -\#available\_periods \cdot 3 \\ +capacity \cdot 2 \\ +\#blocks \end{aligned} \tag{4.1}$$

4.2.4 Periods and Rooms Allocation

Initially, our approach to allocating periods and rooms in the MCTS tree followed a structured sequence. This method aimed to explore all possible (period, room) combinations based on the number of children of the current node being expanded. However, as we progressed, this approach started to fail when the availability of rooms varied across different periods. A refined approach was then developed to address these limitations more robustly.

4.2.4.1 Initial Periods and Rooms Allocation Approach

In the initial approach, each event assignment generated in the MCTS tree follows a structured sequence for period and room allocation. Let:

- n be the number of children of the current node being expanded;
- AP the number of available periods;
- AR the number of available rooms.

The assignment is determined as follows:

- The period index was selected using:

$$n \mod AP \tag{4.2}$$

- The room index was selected using:

$$\frac{n}{AP} \mod AR \tag{4.3}$$

This resulted in a deterministic exploration order. For instance, if $AP = 3$ and $AR = 2$, the assignment would be:

$$n = 0 \Rightarrow P_0, R_0$$
$$n = 1 \Rightarrow P_1, R_0$$
$$n = 2 \Rightarrow P_2, R_0$$
$$n = 3 \Rightarrow P_0, R_1$$
$$n = 4 \Rightarrow P_1, R_1$$
$$n = 5 \Rightarrow P_2, R_1$$

This approach assumes that the number of children will always perfectly expand all (period, room) combinations, but it fails when the search space is reduced, resulting in different periods having different available rooms.

Consider the following scenario:

- Available Periods = $[(0,1),(3,2)]$
- Rooms at (0,1) = $[{}'C',{}' D',{}' E',{}' F',{}' G']$ (5 rooms)
- Rooms at (3,2) = $[{}'B']$ (1 room)

Here, $AP = 2$, but the number of available rooms (AR) depends on the period. The expansion limit is 6, thus we compute assignments from $n = 0$ to $n = 5$:

$$
\begin{aligned}
n = 0: &\quad P_0 = (0,1), \quad R_0 =' C' && \text{(Valid)} \\
n = 1: &\quad P_1 = (3,2), \quad R_0 =' B' && \text{(Valid)} \\
n = 2: &\quad P_0 = (0,1), \quad R_1 =' D' && \text{(Valid)} \\
n = 3: &\quad P_1 = (3,2), \quad R_0 =' B' && \text{(Invalid: Duplicate)} \\
n = 4: &\quad P_0 = (0,1), \quad R_2 =' E' && \text{(Valid)} \\
n = 5: &\quad P_1 = (3,2), \quad R_1 =' B' && \text{(Invalid: Duplicate)}
\end{aligned}
$$

Because room availability is period-specific, the formula produces duplicate assignments when AR is not consistent across all periods.

4.2.4.2 Final Periods and Rooms Allocation Approach

To overcome these limitations, instead of computing the (period, room) combinations via index math, we pre-compute all valid combinations once. This method produces a flat list of unique, valid assignments, and expansion always proceeds without duplication.

Moreover, periods that are less frequently available across all events are given higher priority. In this way, the algorithm reduces the risk of future conflicts, guiding the tree toward more constrained decisions early. For each period, available rooms are also sorted by how closely their capacity matches the event's requirements, minimizing wasted space and increasing flexibility in room usage.

4.2.5 Methodology

The MCTS algorithm consists of four phases: *selection*, *expansion*, *simulation*, and *backpropagation*. Each phase plays a critical role in exploring the search space and updating the tree structure with the simulation outcomes. Figure 4.2 provides a high-level overview of these four phases in the MCTS process.

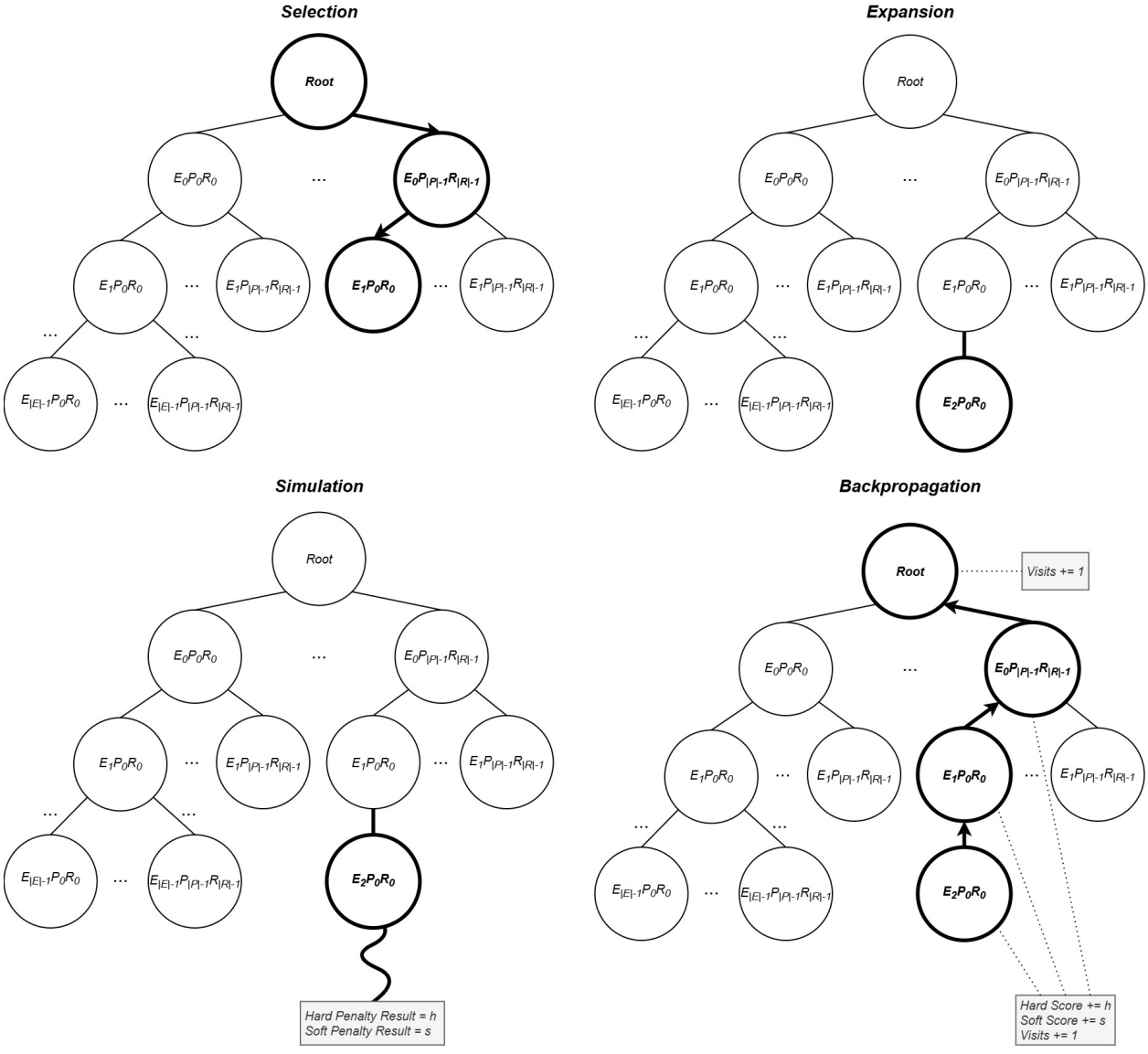


FIGURE 4.2: MCTS steps

4.2.5.1 Selection

The *selection* phase (Algorithm 1) begins at the root and traverses the tree by iteratively selecting the best child nodes until a terminal (leaf) node is reached. A node is considered

terminal if either it cannot be expanded further (i.e., its expansion limit is zero) or if it has reached the maximum depth (i.e., all the events have been assigned). Conversely, a node is deemed fully expanded if either its expansion limit is zero or if the number of its children equals its expansion limit. When the traversal encounters a node that is not fully expanded, the process stops, and that node is selected for further expansion.

The best child node (Algorithm 2) is determined by applying the UCT formula (2.1). However, as the problem has two goals, maximizing the hard constraints first and then the soft constraints, the UCT formula was applied differently. Initially, the algorithm computes the weights for each child node using the hard score. Among the children with the highest hard score, the weights are recalculated based on the soft score. The child node with the highest weight in this second calculation is selected as the best child. However, if all of a node's children have reached their expansion limit, the algorithm backtracks: if the current node has a parent, the current node expansion limit is set to zero, and it moves up to the parent. Reaching the root indicates that the entire tree is fully expanded.

Algorithm 1 Selection

```
1:  procedure SELECTION
2:      current_node ← root
3:      while not current_node.is_terminal_node(len(events)) do
4:          if not current_node.is_fully_expanded() then
5:              break
6:          end if
7:          unflagged_children ← [child for each child in current_node.children if child
    and child.expansion_limit ≠ 0]
8:          if unflagged_children is empty then
9:              if current_node.parent exists then
10:                 current_node.expansion_limit ← 0
11:                 current_node ← current_node.parent
12:             else
13:                 return False
14:             end if
15:         else
16:             current_node ← current_node.best_child(unflagged_children)
17:         end if
18:     end while
19:     self.current_node ← current_node
20:     return True
21: end procedure
```

4.2.5.2 Expansion

The $expansion$ phase (Algorithm 3) is responsible for expanding the node previously selected by assigning the next event to a valid (period, room) combination and creating a

Algorithm 2 Best Child

```
function BEST_CHILD(unflagged_children, c_param = 1.4)
    choices_weights ← [child.score_hard/child.visits + c_param · sqrt(2 ln(self.visits)/child.visits) for each child in unflagged_children]

    max_weight ← max(choices_weights)
    best_children ← [unflagged_children[i] for each i, weight in enumerate(choices_weights) if weight = max_weight]

    choices_weights ← [child.score_soft/child.visits + c_param · sqrt(2 ln(self.visits)/child.visits) for each child in best_children]

    return best_children[index_of(max(choices_weights))]
end function
```

corresponding child node.

The next event to schedule is determined by the depth of the current node. For the selected event, the algorithm gathers the available (weekday, timeslot) combinations and filters suitable rooms for each period (Algorithm 4), based on the event's capacity requirements and current room occupation.

A list of valid (period, room) combinations is generated. If no such combinations exist or the node has already been expanded for all possibilities, expansion is terminated, and the node is marked as fully expanded.

A combination is selected based on the number of already-expanded children. A new path is constructed by assigning the selected period and room to the event.

Before proceeding, the algorithm evaluates the partial timetable (i.e., the new path) against a global hard penalty threshold. If the new assignment violates hard constraints resulting in a penalty below the threshold, a null child is added to indicate that the expansion stopped for that branch. This pruning condition reduces unnecessary exploration.

If the expansion continues, the algorithm estimates the number of valid (period, room) combinations for the next event. This value is stored as the child's expansion_limit to guide future branching.

Finally, a new child node is created with the updated path and expansion limit. This new node is added to the tree, and the current node is updated to this newly created node.

Algorithm 3 Expansion

```
function EXPANSION
    event ← events[current_node.depth()]
    available_periods ← event["Available_Periods"]

    if available_periods is empty then
        current_node.expansion_limit ← 0
        return False
    end if

    rooms_by_period ← find_available_rooms()
    period_room_combinations ← [(weekday, timeslot, room) for (weekday, timeslot), rooms in room_by_period.items()]

    if period_room_combinations is empty or current_node is fully expanded then
        current_node.expansion_limit ← 0
        return False
    end if

    new_weekday, new_timeslot, new_room ← period_room_combinations[len(current_node.children)]
    new_path ← copy(current_node.path)
    new_path[event["Id"]] ← {**event, "RoomId": new_room, "WeekDay": new_weekday, "Timeslot": new_timeslot}

    new_expansion_limit ← = calculate_expansion_limit(new_path)
    if new_expansion_limit is None: return False

    child_node ← MCTSNode(expansion_limit = expansion_limit, parent = current_node, path = new_path)
    current_node.children.append(child_node)
    current_node ← child_node
    return True
end function
```

4.2.5.3 Simulation

The *simulation* phase estimates the value of multiple actions, in this case, distinct event allocations, which will guide the selection and expansion steps in future iterations.

A random event allocation approach was initially explored but produced suboptimal results. Therefore, a more structured method was implemented.

The *simulation* function (Appendix 6 Algorithm 6) starts by creating a copy of the current node's path, which contains the events already scheduled. It then identifies the remaining events, i.e., events that have not been assigned yet. These remaining events are sorted based on two criteria:

Algorithm 4 Find Available Rooms

```
function FIND_AVAILABLE_ROOMS(event_capacity, rooms, events, available_periods)
    period_room_availability ← {period: set(rooms.keys()) for each period in available_periods}
    for each other_event in events do
        occupied_period ← (other_event["WeekDay"], other_event["Timeslot"])
        if occupied_period in period_room_availability then
            period_room_availability[occupied_period].discard(other_event["RoomId"])
        end if
    end for

    suitable_rooms ← {room_id for each room_id, room in rooms.items() if room["Capacity"] ≥ event_capacity}

    for each period in available_periods do
        if period_room_availability[period] is not empty then
            intersected ← period_room_availability[period] ∩ suitable_rooms
            if intersected is not empty then
                sorted_rooms ← sort(intersected, by (room["Capacity"] - event_capacity))
            else
                sorted_rooms ← sort(period_room_availability[period], by |room["Capacity"] - event_capacity|)
            end if
            period_room_availability[period] ← sorted_rooms
        end if
    end for

    return period_room_availability
end function
```

1. Whether they were previously unassigned in prior simulations, to prioritize harder-to-schedule events.

2. Their priority, ensuring that more critical events are allocated first.

For each unvisited event, the best available period and room combination is sought. The approach iterates through all the available periods and rooms, calculating the hard and soft penalties for each combination. During the calculation of the soft penalty, the importance of compactness (i.e., minimizing the time between related events) is adjusted, with a higher weight applied as the algorithm progresses.The combination with no hard penalties and the lowest soft penalty is selected as the best option. If multiple optimal choices exist, one is randomly selected. The event is then updated with the selected room, weekday, and timeslot. If no valid allocation is found, the event will have higher priority in the next simulation.

After scheduling all events, the simulation result is calculated, and the best and worst penalty scores are then updated based on these results. If a new best result is found, the timetable is saved to a file. If a new best result is found and the hard penalty is zero, indicating a feasible solution, the HC algorithm (section 4.3) is used to further optimize the timetable.

4.2.5.4 Backpropagation

The $backpropagation$ phase (Algorithm 5) updates the tree based on the simulation results.

It starts from the current node, which is the leaf node where the simulation occurred, and moves up until reaching the root node. During this process, it performs updates for each node in the path, including incrementing the number of visits and updating the hard and soft scores with the simulation results.

Algorithm 5 Backpropagation

```
1: function BACKPROPAGATION(simulation_result_hard, simulation_result_soft)
2:     node ← current_node
3:     while node ≠ None do
4:         node.visits += 1
5:         node.score_hard += simulation_result_hard
6:         node.score_soft += simulation_result_soft
7:         node ← node.parent
8:     end while
9: end function
```

4.2.6 Normalization

The normalization formula 4.4 is applied to both hard and soft constraints to standardize the simulation results [12]:

$$N(n) = \frac{e^a - 1}{e - 1}, \quad \text{with } a = \frac{best\_penalty_n - worst\_penalty}{best\_penalty - worst\_penalty}, \tag{4.4}$$

where $best\_penalty$ and $worst\_penalty$ represent the best and the worst simulation results in the entire tree, and $best\_penalty_n$ is the best simulation result under node n.

This formulation ensures that the best simulation result is mapped to a value close to one, while the worst is mapped to zero.

4.3. Hill climbing

The HC algorithm (Appendix 6 Algorithm 7) is applied as a local search method to further optimize the timetable obtained from the MCTS simulation phase. This approach refines an initially feasible solution by iteratively exploring small modifications to enhance overall quality.

To maintain the structure established during the tree search, HC only modifies events that were not allocated in the MCTS phase. This ensures that the global search strategy remains intact while improving suboptimally placed events by the simulation phase.

4.3.1 Neighborhoods

After finding a feasible solution with MCTS, HC refines the solution by exploring neighboring states. The algorithm explores the solution space using six different types of neighbors inspired by the ITC-2007 track 3 winner [10]:

- **Period move:** Change the timeslot and weekday of an event while keeping the same room.
- **Room move:** Change the room of an event while keeping the same timeslot and weekday.
- **Event move:** Change both the room and timeslot of an event.
- **Room stability move:** Assign all events of the same course to the same room if possible.
- **Compactness move:** Move an event to a timeslot adjacent to another event of the same curriculum.
- **Minimum working days move:** Spread a course's events across more days to avoid teaching all lectures in a short period.

Each move has a weight, which affects how often it is chosen. Period, room and event moves appear to be the most effective, consequently they have a higher weight than the others.

4.3.2 Methodology

The HC approach follows the following steps:

1. Starts with an initial complete timetable with no hard constraints violated.
2. Repeatedly selects a random neighborhood move, based on the assigned weights.
3. Applies the move and evaluates the new timetable.
4. If the new timetable is better, it keeps the changes; otherwise, it reverts them.
5. Stops when:
    - A timetable satisfying all hard and soft constraints is found.
    - No improvements are identified after a predefined number of idle iterations (HC_IDLE = 5000).
    - The time limit is reached.

4.4. Diving

To improve effectiveness in timetable generation, we also implemented a diving strategy in MCTS.

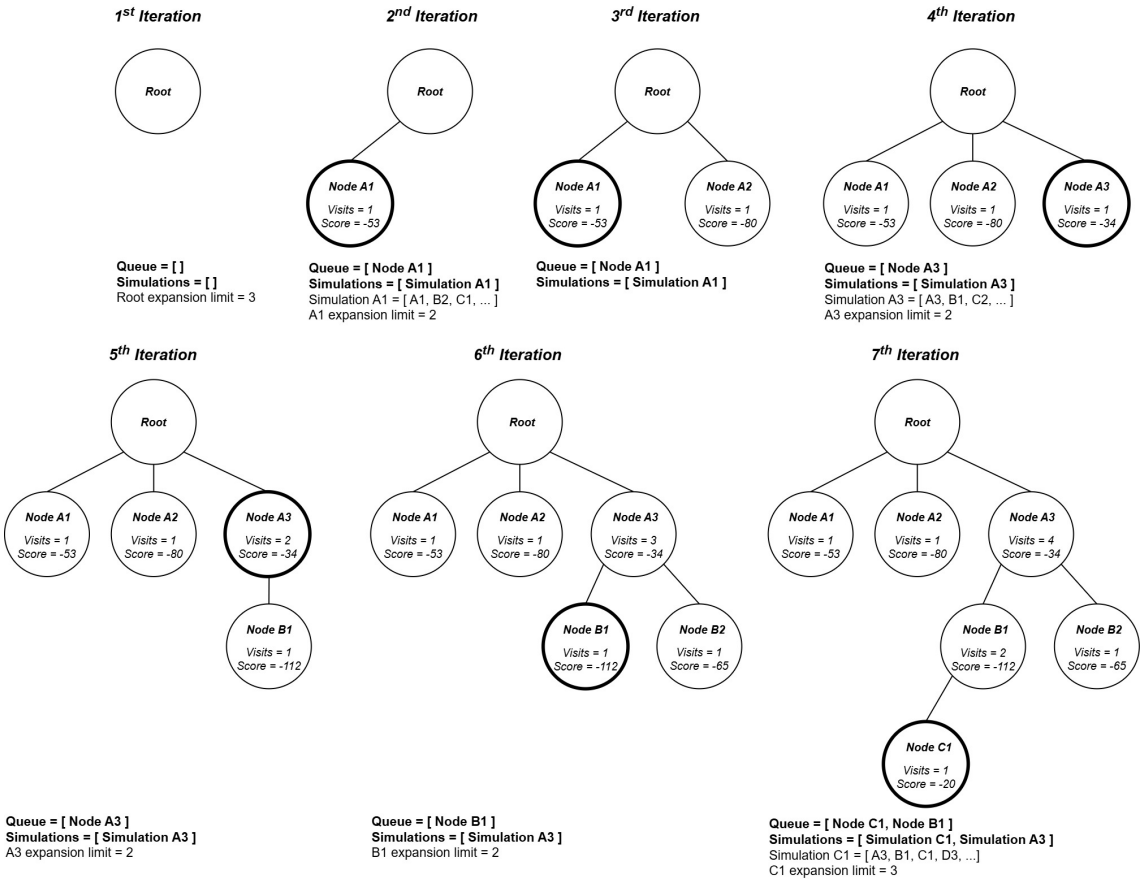


FIGURE 4.3: Diving approach

4.5. Front-end

The development process involved several adaptations and refinements. Initially, the previous work was analyzed and made functional, serving as a baseline for the new system.

The MCTS algorithm was being developed while also being integrated into the front-end interface intended to showcase the project. However, as the focus shifted towards adapting it to the ITC-2007 track 3 benchmark for performance evaluation and fine-tuning the algorithm, the front-end development gradually lost priority. As the algorithm evolved and diverged significantly from its original form, aligning it with the prior work proved difficult. Consequently, continuing its development would be time-consuming and ultimately not worthwhile, as it was primarily intended for demonstration purposes. Instead, the system should be directly integrated into the sigarra* website.

*https://sigarra.up.pt/up/pt/web$_{page.inicial}$

5. Tests

This chapter outlines the numerous tests used to evaluate the performance of the final algorithm on instances from the ITC-2007 benchmark*.

5.1. Test Instances

The evaluation used all 21 instances from the ITC-2007 track 3 dataset ($comp01.ctt$ to $comp21.ctt$). Each instance includes detailed information about about curricula, courses, rooms, unavailability constraints, lectures, periods and lecturers. The key features of each instance are summarized in Table 5.1.

Instance	Curricula	Courses	Rooms	Constraints	Lectures	Periods	Lecturers
comp01	14	30	6	53	160	30	24
comp02	70	82	16	513	283	25	71
comp03	68	72	16	382	251	25	61
comp04	57	79	18	396	286	25	70
comp05	139	54	9	771	152	36	47
comp06	70	108	18	632	361	25	87
comp07	77	131	20	667	434	25	99
comp08	61	86	18	478	324	25	76
comp09	75	76	18	405	279	25	68
comp10	67	115	18	694	370	25	88
comp11	13	30	5	94	162	45	24
comp12	150	88	11	1368	218	36	74
comp13	66	82	19	468	308	25	77
comp14	60	85	17	486	275	25	68
comp15	68	72	16	382	251	25	61
comp16	71	108	20	518	366	25	89
comp17	70	99	17	548	339	25	80
comp18	52	47	9	594	138	36	47
comp19	66	74	16	475	277	25	66
comp20	78	121	19	691	390	25	95
comp21	78	94	18	463	327	25	76

TABLE 5.1: Key features of the ITC-2007 track 3 instances.

All instances guarantee at least one feasible solution, which means that there are no violations of hard constraints. Additionally, to validate the correctness of a solution, the competition organizers provided a C++ source code that identifies the type of violation and calculates the corresponding penalty.

5.2. Test Setup

The tests were conducted for different values of the parameter C (Formula 2.1), which affects the exploration-exploitation trade-off in the MCTS algorithm. A higher C value

*https://www.eeecs.qub.ac.uk/itc2007/Login/SecretPage.php

increases the weight of the second term, allowing the algorithm explore less-visited nodes with more intensity. On the other hand, a lower C value favors nodes with a higher average reward, resulting in increased exploitation of known favorable choices. Specifically, we tested the algorithm with some values of C ranging from 0.1 to 1000.

Additionally, an alternative UCT formula was evaluated, modifying the exploitation term to use the accumulated reward instead of the average (Formula 5.1), giving an advantage to the nodes that were exploited first.

$$UCT = w_i + 2C\sqrt{\frac{2\ln n}{n_i}}, \tag{5.1}$$

where w_i is the total reward of all playouts through this state, n_i is the number of visits of child node i, C is a constant greater than zero (typically $\sqrt{2}$), and n is the number of visits of the parent node.

5.3. Performance Metrics

Each test corresponds, therefore, to a different C value, and the performance measurements are recorded for 21 problem instances, denoted as comp01 to comp21. For each instance, the method iterates numerous times until a stopping condition is met, which is usually a time constraint.

For each test, the following key performance metrics were considered:

- **Best hard penalty:** The lowest number of hard constraint violations found during the execution of the algorithm for a given instance. A value of zero indicates a feasible solution that satisfies all hard constraints.
- **Worst hard penalty:** The highest number of hard constraint violations found during the execution of the algorithm for a given instance.
- **Best soft penalty:** The lowest soft constraint penalty achieved during the search process for a given instance.
- **Worst soft penalty:** The highest soft constraint penalty achieved during the search process for a given instance.
- **Iteration of best solution:** The specific iteration at which the best solution (in terms of hard or soft penalties) was found.

- **Total number of iterations:** The overall count of iterations performed during the algorithm's execution for each problem instance.
- **Time to best solution:** The elapsed time required to reach the best solution for a given instance.

5.4. Testing Procedure

To thoroughly evaluate the algorithm's robustness and consistency, two different testing approaches were employed:

1. **1-hour runs:** The algorithm was executed for 1 hour to assess its ability to converge towards high-quality solutions given sufficient runtime.
2. **10-minute runs:** The algorithm was run for 10 minutes using ten different seeds (1-10). Using fixed seeds ensured that the results could be reproduced, allowing for a fair comparison of different parameter settings and configurations. This approach also helped to evaluate the consistency and variability of results across different initializations.

6. Results

This chapter presents the main outcomes of the proposed MCTS-based system for CB-CTT, including its performance, feasibility, and evaluations of enhancements such as the diving strategy.

6.1. Python vs PyPy Performance

During the development and testing of the system, both standard Python and PyPy* were evaluated for performance. While both interpreters correctly executed the MCTS algorithm, the performance difference was substantial. PyPy called over 3 times more functions compared to Python.

Given the compute-intensive nature of MCTS, PyPy was ultimately chosen for running all the experiments and benchmarks.

6.2. Feasibility

The proposed system consistently produced feasible solutions across all tested configurations and datasets.

6.3. C Parameter Behaviour

The exploration constant C in the UCT formula (Formula 2.1) was varied across several orders of magnitude (from 0.1 to 1000), including the modified variant incorporating accumulated rewards (Formula 5.1). Surprisingly, the results showed that varying C had minimal impact on solution quality, which indicates a lower-than-expected sensitivity to node selection.

6.4. Diving strategy

6.5. Competition Results Comparison

*A Just-In-Time (JIT) compiling alternative. Link: https://pypy.org/

7. Conclusion

TODO

The proposed system presents a novel application of MCTS to UCTTP, combined with HC for local improvements. By leveraging interactive recommendations and conflict detection, the tool provides a more effective and adaptive scheduling process for FCUP and can be extended to other institutions and help in other studies.

Future work will focus on the diving strategy, refining the heuristic functions and improving computational performance.

References

[1] U. D. Porto, “Fcup em números.” [Online]. Available: https://www.up.pt/fcup/pt/a-fcup/institucional/fcup-em-numeros/ [Cited on page 1.]

[2] 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, Mar. 2012, conference Name: IEEE Transactions on Computational Intelligence and AI in Games. [Online]. Available: https://ieeexplore.ieee.org/document/6145622/?arnumber=6145622 [Cited on pages 7 and 8.]

[3] R. Lewis, “A survey of metaheuristic-based techniques for university timetabling problems,” *OR Spectrum*, vol. 30, no. 1, pp. 167–190, Jan. 2008. [Online]. Available: https://doi.org/10.1007/s00291-007-0097-0 [Cited on page 11.]

[4] S. Abdipoor, R. Yaakob, S. L. Goh, and S. Abdullah, “Meta-heuristic approaches for the university course timetabling problem,” *Intelligent Systems with Applications*, vol. 19, p. 200253, Sep. 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2667305323000789 [Cited on pages 11, 12, and 13.]

[5] H. Babaei, J. Karimpour, and A. Hadidi, “A survey of approaches for university course timetabling problem,” *Computers & Industrial Engineering*, vol. 86, pp. 43–59, Aug. 2015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0360835214003714 [Cited on page 11.]

[6] M. C. Chen, S. N. Sze, S. L. Goh, N. R. Sabar, and G. Kendall, “A survey of university course timetabling problem: Perspectives, trends and opportunities,” *IEEE Access*, vol. 9, pp. 106 515–106 529, 2021, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/document/9499056/?arnumber=9499056 [Cited on pages 11, 12, and 13.]

[7] E. Talbi, “Metaheuristics: From design to implementation,” *John Wiley & Sons google schola*, vol. 2, pp. 268–308, 2009. [Cited on pages 11 and 12.]

[8] K.-L. Du, M. Swamy *et al.*, *Search and optimization by metaheuristics*. Springer, 2016, vol. 1. [Cited on pages 11 and 12.]

[9] E. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, pp. 1695–1724, 07 2013. [Online]. Available: https://www.researchgate.net/publication/256442073_Hyper-heuristics_A_survey_of_the_state_of_the_art [Cited on page 12.]

[10] T. Müller, "Itc2007 solver description: a hybrid approach," *Annals of Operations Research*, vol. 172, no. 1, pp. 429–446, Nov. 2009. [Online]. Available: http://link.springer.com/10.1007/s10479-009-0644-y [Cited on pages 13 and 27.]

[11] S. L. Goh, "An investigation of monte carlo tree search and local search for course timetabling problems," *University of Nottingham, Malaysia*, pp. 76–105, Jul. 2017. [Online]. Available: https://eprints.nottingham.ac.uk/43558/ [Cited on page 14.]

[12] J. P. Pedroso and R. Rei, "Tree search and simulation," in *Applied Simulation and Optimization: In Logistics, Industrial and Aeronautical Practice*, M. Mujica Mota, I. F. De La Mota, and D. Guimarans Serrano, Eds. Cham: Springer International Publishing, 2015, pp. 109–131. [Online]. Available: https://doi.org/10.1007/978-3-319-15033-8_4 [Cited on page 26.]

Appendix A

Algorithm 6 Simulation

```
1: function SIMULATION(start_time, time_limit)
2:     assigned_events ← current_node.path.copy()
3:     unassigned_events ← set()
4:     remaining_events ← sorted(events[current_node.depth():],
                        key = λ event: (event["Id"] in previous_unassigned_events,
event["Priority"], random.random()),
                        reverse = True)
5:
6:     for each i,event in enumerate(remaining_events) do
7:         best_room_and_period ← find_best_room_and_period()
8:         if best_room_and_period exists then
9:             event["RoomId"], event["WeekDay"], event["Timeslot"] ←
best_room_and_period
10:            assigned_events[event["Id"]] ← event
11:        else
12:            previous_unassigned_events.add(event["Id"])
13:            unassigned_events.add(event["Id"])
14:        end if
15:    end for
16:
17:    hard_penalty_result, soft_penalty_result ← evaluate_timetable(assigned_events,
unassigned_events)
18:
19:    update_penalties(soft_penalty_result, hard_penalty_result)
20:
21:    if (hard_penalty_result > global_best_hard_penalty) or (hard_penalty_result =
global_best_hard_penalty and soft_penalty_result > global_best_soft_penalty) then
22:        global_best_hard_penalty ← hard_penalty_result
23:        global_best_soft_penalty ← soft_penalty_result
24:        with open(output_filename, 'w') as file:
25:                write_best_simulation_result_to_file(list(assigned_events.values()),
file)
26:        if len(unassigned_events) == 0 and hard_penalty_result == 0 and
soft_penalty_result ≠ 0 then
27:            global_best_soft_penalty ← hill_climber.run_hill_climbing(assigned_events,
                    events[current_node.depth()]["Id"],
                    global_best_soft_penalty, start_time, time_limit)
28:            update_penalties(global_best_soft_penalty)
29:        end if
30:    end if
31:
32:    simulation_result_hard ← normalize_hard(current_node.best_hard_penalty)
33:    simulation_result_soft ← normalize_soft(current_node.best_soft_penalty)
34:
35:    return simulation_result_hard, simulation_result_soft
36: end function
```

Algorithm 7 Run Hill Climbing

```
1: function RUN_HILL_CLIMBING(best_timetable, start_key, best_result_soft, start_time, time_limit)
2:     best_result_soft ← best_result_soft
3:     neighborhoods ← [(period_move, 1), (room_move, 1), (event_move, 1), (room_stability_move, 0.7), (min_working_days_move, 0.5), (curriculum_compactness_move, 0.7)]
4:
5:     idle_iterations ← 0
6:
7:     while idle_iterations < HC_IDLE and (time.time() - start_time ≤ time_limit) do
8:         (current_neighborhood, _) ← weighted random choice from neighborhoods
9:
10:        unscheduled_events ← dict_slice(best_timetable, start_key)
11:
12:        modified_timetable ← current_neighborhood(best_timetable, unscheduled_events)
13:
14:        if modified_timetable is None then
15:            idle_iterations ← idle_iterations + 1
16:            continue
17:        end if
18:
19:        result ← evaluate_timetable(modified_timetable)
20:
21:        if result is not None then
22:            if result > best_result_soft then
23:                best_result_soft ← result
24:                best_timetable ← modified_timetable
25:                write_simulation_results(output_filename, best_timetable.values(), start_time, 0, result)
26:                if result = 0 then
27:                    return 0
28:                end if
29:                idle_iterations ← 0
30:            else
31:                revert_changes(best_timetable)
32:                idle_iterations ← idle_iterations + 1
33:            end if
34:        else
35:            revert_changes(best_timetable)
36:            idle_iterations ← idle_iterations + 1
37:        end if
38:    end while
39:
40:    return best_result_soft
41: end function
```