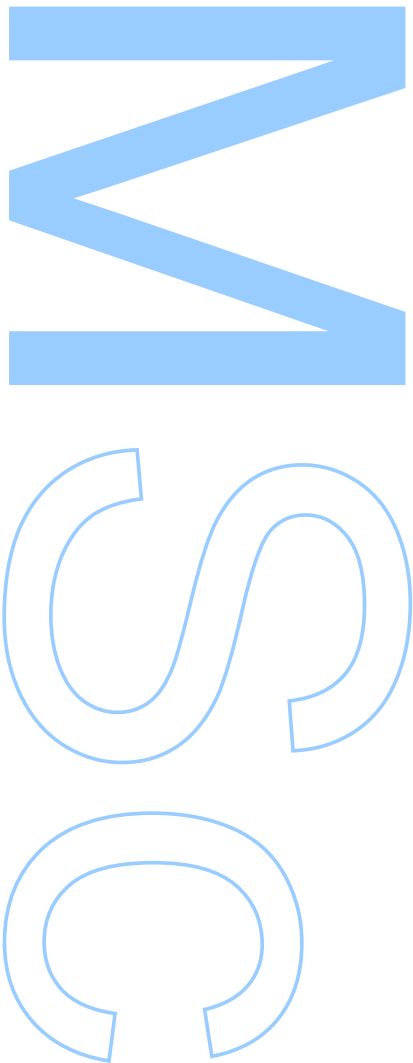
An interactive tool for supporting university timetabling

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

Coorientador
Prof. Dr. Pedro Vasconcelos, Faculdade de Ciências

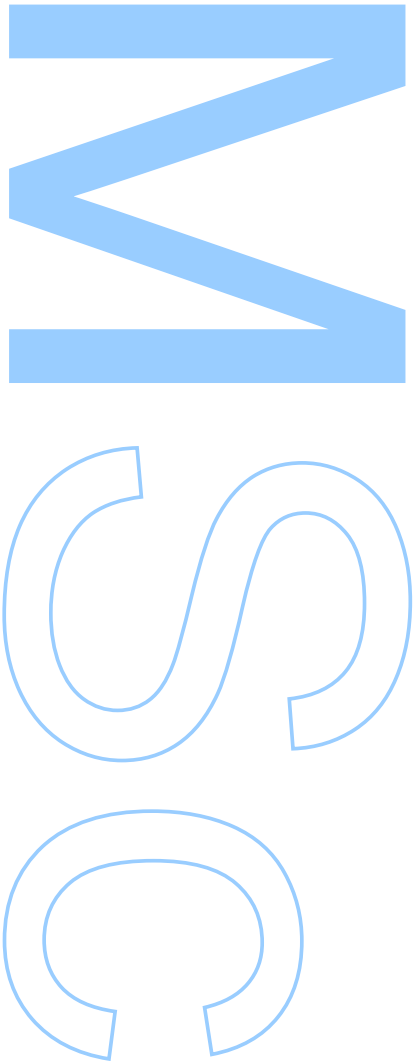


U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

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

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

Acknowledge ALL the people!

Resumo

Este tese é sobre alguma coisa

Palavras-chave: informática (keywords em português)

Abstract

This thesis is about something, I guess.

Keywords: informatics

Table of Contents

List of Figures

List of Abbreviations

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

COP Combinatorial Optimization Problem. 3

FCUP Faculty of Sciences of the University of Porto. 1, 4, 11

HC Hill Climbing. 2, 3, 10, 18, 20

ITC International Timetabling Competition. 5, 8

ITC-2007 Second International Timetabling Competition. 8, 11, 21

MCTS Monte Carlo Tree Search. iv, 1– 3, 5, 10, 12, 20

PATAT International Conference on the Practice and Theory of Automated Timetabling. 4

PE-CTT Post-Enrollment Course Timetabling. 4

UCT Upper Confidence Bounds for Trees. 5, 15

UCTTP University Course Timetabling Problem. 1– 4, 7

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 scheduling is centered around courses.

Moreover, when considering a complete bachelor's or master's degree, which spans several years and involves numerous courses, each with multiple classes and unique requirements, the challenge intensifies. Due to the size and complexity of the problem, obtaining an optimal solution in usable time is typically not feasible. Nevertheless, heuristic algorithms have proved capable of producing approximate and good-quality solutions effectively.

1.1. Objective

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. Therefore, 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.

1.2. Contributions

In the academic year 2021/2022, FCUP enrolled more than 4300 students, offered 127 different curricula, and coordinated 589 collaborators [1], underscoring the need for an efficient timetabling system that can manage such scale. The approach in use is not only inefficient but also prone to errors and conflicts. To address the problem, the Monte Carlo Tree Search (MCTS) heuristic search algorithm was chosen, as it has been applied to various optimization problems and has proven to be particularly effective in games.

Hill Climbing (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, presenting an opportunity to advance the state of the art and help in further studies. Therefore, this dissertation proposes a novel hybrid approach that combines the global exploration capabilities of MCTS with the local optimization offered by HC.

1.3. Dissertation layout

The remainder of the dissertation is organized as follows:

- **Chapter 2 - Background:**
- **Chapter 3 - Related Work:**
- **Chapter 4 - Development:**
- **Chapter 5 - Tests:**
- **Chapter 6 - Results:**
- **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:

- **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.

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

2.4.1 Phases

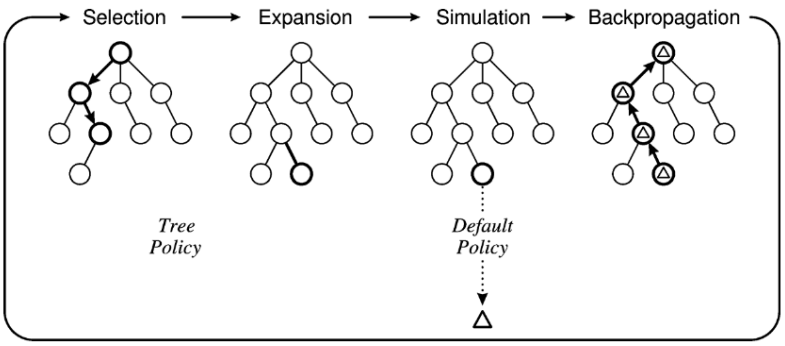


FIGURE 2.1: One MCTS iteration [2]

*https://patatconference.org/communityService.html

The algorithm consists of four phases, as illustrated in Figure 2.1 [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.

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

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

2.5. 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 Hill Climbing, Simulated Annealing, Tabu Search, and Genetic Algorithms. 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.

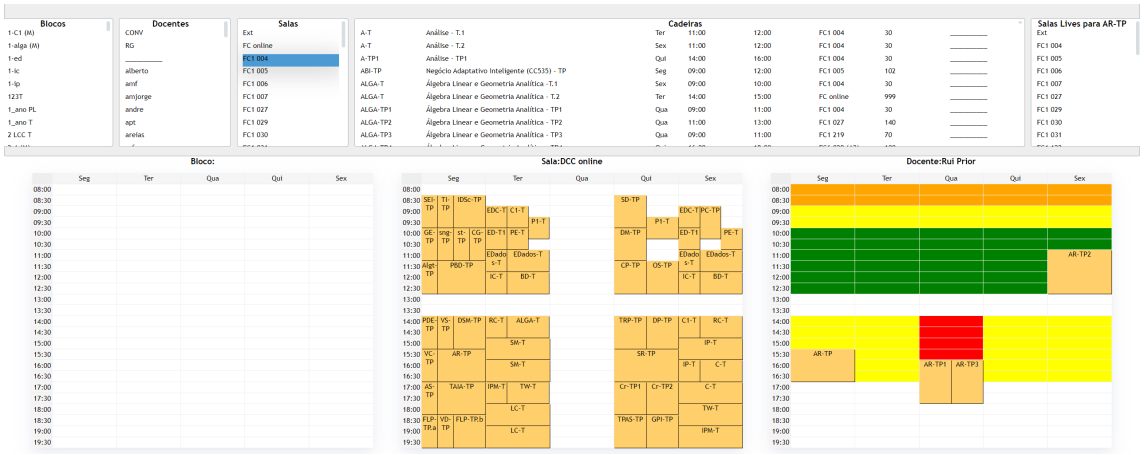


FIGURE 2.2: Previous work interface

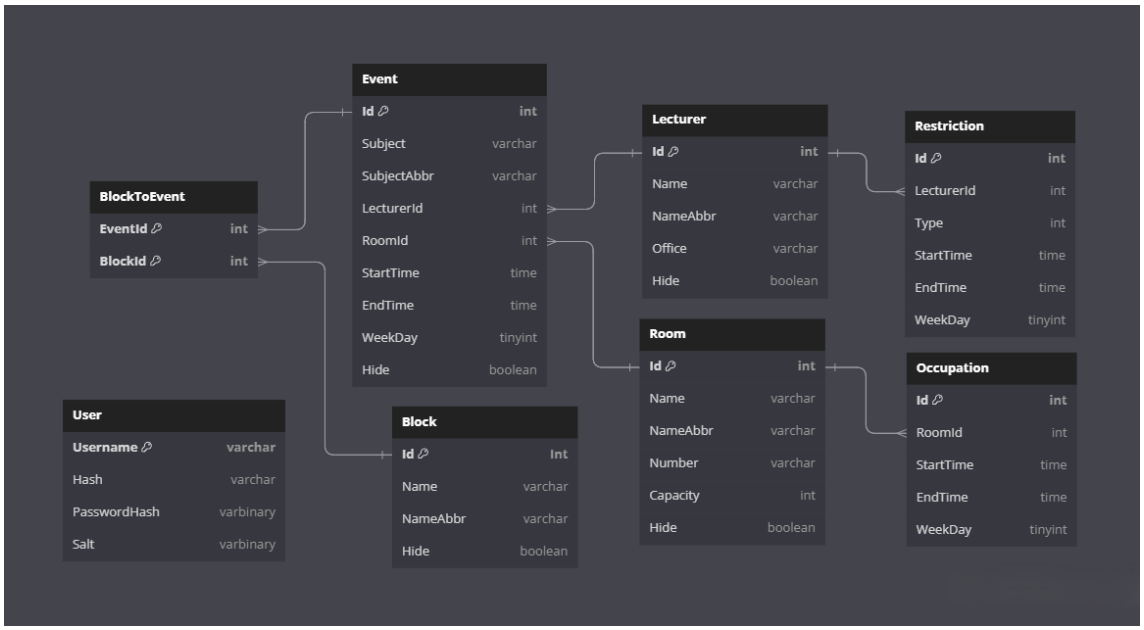


FIGURE 2.3: Previous work UML

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 2.3), 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 2.2). 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/tree/main`

- **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 addressing 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 different categories, including operational research, metaheuristics, hyperheuristics, multi-objective, and hybrid approaches.

3.1. Operational Research (OR)

Operational research approaches, despite their complexity, are simple to implement and can be considered if time and space constraints are not a concern. However, the other types of approaches provide better results [5]. 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].

In recent years, metaheuristics have become 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 Hill Climbing (HC), Simulated Annealing (SA), Tabu Search (TS), and Iterated Local Search (ILS). Simulated Annealing is perhaps the most effective single solution-based metaheuristic, particularly in benchmark datasets, but Tabu Search 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 Generic 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 choices and the lack of clear guidance [9]. To address this problem, hyper-heuristics aim to generalize methods by selecting the most suitable heuristic or sequence of heuristics 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.

In particular, Tomáš Müller, winner of the ITC-2007* tracks 1 and 3, developed a 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, Hill Climbing, Great Deluge, and optionally Simulated Annealing. During the construction phase, Iterative Forward Search is employed to find a complete solution, followed by Hill climbing to find the local optimum. When a solution cannot be improved further using Hill Climbing, Great Deluge is used to iteratively decrease, based on a cooling rate, a bound that is imposed on the value of the current solution. Optionally, Simulated Annealing can be used when the bound reaches its lower limit.

*https://www.eeecs.qub.ac.uk/itc2007/

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. 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.

4.2. Problem Formulation

(...) The entities in the problem are listed below:

- **Periods,** $P = \{P_1, P_2, ..., P_{|P|}\}$**:** Days are divided into fixed timeslots, with periods consisting of a day and a timeslot.

  In the previous work, there is no periods. There are weekdays, a start time and an end time. For now, in the current approach, an event is only associated with a period of one hour and without any link to previous work. If I implement this concept to previous work, an event should be assigned to one or more half-hour periods.

- **Rooms,** $R = \{R_1, R_2, ..., R_{|R|}\}$**:** Each room has a capacity and a type.
  - Occupation (the room may be utilized for non-teaching reasons, such as exams) (TODO)
  - Capacity (number of seats)
  - Type (TODO)
- **Lecturers,** $L = \{L_1, L_2, ..., L_{|L|}\}$**:**
  - Office
  - Restriction (availability and TODO: preferences for periods)
- **Events,** $E = \{E_1, E_2, ..., E_{|E|}\}$**:** Represent the events to be scheduled. Each event has associated attributes:

Previous work:

- Start and end time
- Weekday
- Duration
- Lecturer
- Room

TODO: adicionar ao trabalho anterior?

- Period
- Capacity (number of students)
- Number of lectures
- Minimum working days (days over which the events of the same block should be spread)
- Available periods
- Priority

- **Blocks,** $B = \{B_1, B_2, ..., B_{|B|}\}$**:** A block may represent a group of related events, such as all events of a subject, or events from the same course and curricular year.

4.2.1 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 subject 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 course and curricular year, or taught by the same lecturer, must be scheduled in different periods.
- **TODO H4:** If a room is unavailable in a given period, then no events can be scheduled in that period.
- **H5:** If a lecturer is unavailable in a given period, then no events can be taught by this lecturer in that period.
- **TODO H6:** Lecturers and students must have a free lunch period.

Soft constraints:

- **S1:** Events of the same subject should be spread into the given minimum number of days.
- **S2:** Events belonging to the same course and curricular year 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 subject should be given in the same room.

4.3. Monte Carlo Tree Search

(...)

4.3.1 MCTS Tree

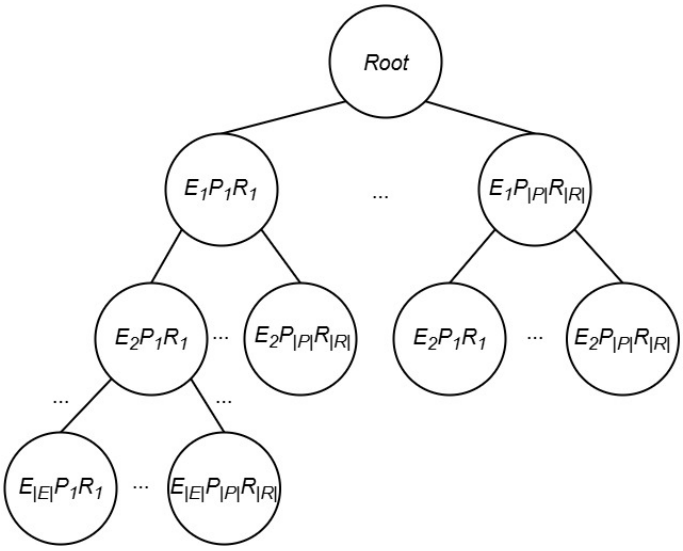


FIGURE 4.1: MCTS tree

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, $E1$, to various P and R. Each child node represents a specific assignment $((E_1, P_1, R_1), ..., (E_1, P_{|P|}, R_{|R|}))$. Subsequent levels correspond to the sequential assignment of events $E_2, E_3, ..., E_{|E|}$. Unavailable periods are discarded immediately to reduce the search space.

The 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 blocks in which the event is, as events that appear in more blocks are more difficult to allocate.

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

4.3.1.1 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.

4.3.2 Methodology

Figure 4.2(...)

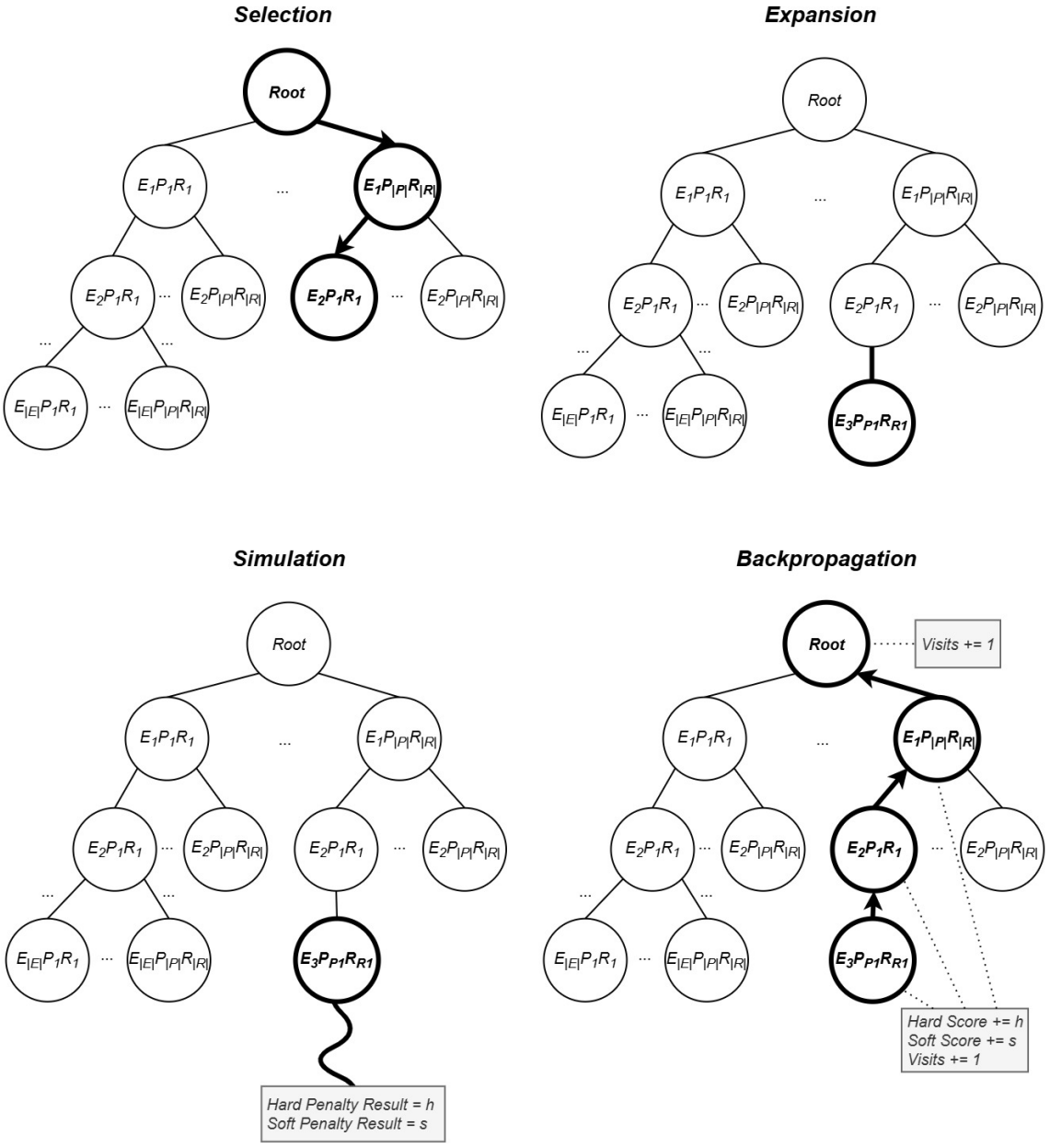


FIGURE 4.2: MCTS steps

4.3.2.1 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.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:             best_child ← current_node.best_child(unflagged_children)
17:             current_node ← best_child
18:         end if
19:     end while
20:     self.current_node ← current_node
21:     return True
22: end procedure
```

Algorithm 2 Best Child

```
function BEST_CHILD(unflagged_children, c_param = 1.4)
    choices_weights ← [child.score_hard + 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 + 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
```

4.3.2.2 Expansion

The $expansion$ phase (Algorithm 3) is responsible for expanding the node previously selected by adding a new child node. It starts by retrieving the next unvisited event based on the current node's depth, which is the length of its path.

The algorithm also retrieves the corresponding weekday and timeslot and finds the available rooms per period for the event (Algorithm 4), removing occupied rooms and filtering rooms based on capacity. If rooms are available, it selects the corresponding one and creates a new event with the selected room, weekday, and timeslot. After this selection, the expansion limit for the next event in the timetable is estimated, i.e., the number of children that the new current node child will probably have. If no more events remain, the limit is set to zero. Otherwise, it sums the number of available rooms per period for the next event.

Finally, a new child node is created and added to the tree. The current node is updated to this newly created node.

4.3.2.3 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.

Algorithm 3 Expansion

```
1: procedure EXPANSION
2:     event ← events[current_node.depth()]
3:     available_periods ← event["Available_Periods"]
4:
5:     period ← len(current_node.children)
6:     period_index ← period % len(available_periods)
7:     new_weekday, new_timeslot ← available_periods[period_index]
8:
9:     available_rooms ← find_available_rooms()
10:
11:    new_room_index ← period // len(available_periods) % len(available_rooms)
12:    new_room ← available_rooms[new_room_index]
13:
14:    new_path ← copy(current_node.path)
15:    new_path[event["Id"]] ← {**event, "RoomId": new_room, "WeekDay": new_weekday, "Timeslot": new_timeslot}
16:
17:    next_event ← events[current_node.depth()+1] if current_node.depth() + 1 < len(events) else None
18:
19:    if next_event is None then
20:        expansion_limit ← 0
21:    else
22:        expansion_limit ← sum( len(rooms) for each rooms in find_available_rooms() )
23:    end if
24:
25:    child_node ← MCTSNode(expansion_limit = expansion_limit, parent = current_node, path = new_path)
26:    current_node.children.append(child_node)
27:    current_node ← child_node
28: end procedure
```

The $simulation$ phase 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:

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

Algorithm 4 Find Available Rooms

```
1: function FIND_AVAILABLE_ROOMS(event_capacity, rooms, events, available_periods)
2:     period_room_availability ← {period: set(rooms.keys()) for each period in available_periods}
3:
4:     for each other_event in events do
5:         occupied_period ← (other_event["WeekDay"], other_event["Timeslot"])
6:         if occupied_period in period_room_availability then
7:             period_room_availability[occupied_period].discard(other_event["RoomId"])
8:         end if
9:     end for
10:
11:    suitable_rooms ← {room_id for each room_id, room in rooms.items() if room["Capacity"] ≥ event_capacity}
12:
13:    for each period in available_periods do
14:        if period_room_availability[period] is not empty then
15:            period_room_availability[period] ← period_room_availability[period] ∩ suitable_rooms
16:            if empty: keep original period_room_availability[period]
17:        end if
18:    end for
19:
20:    return period_room_availability
21: end function
```

approach is similar to constraint programming as it iterates through all the available periods and rooms, calculating the hard and soft penalties for each combination. 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 4.4) is used to further optimize the timetable.

4.3.2.4 Backpropagation

The $backpropagation$ phase (Algorithm 6) updates the tree based on the simulation results.

Algorithm 5 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:    current_hard_values.append(hard_penalty_result)
20:    current_soft_values.append(soft_penalty_result)
21:
22:    update_penalties(soft_penalty_result, hard_penalty_result)
23:
24:    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
25:        global_best_hard_penalty ← hard_penalty_result
26:        global_best_soft_penalty ← soft_penalty_result
27:        with open(output_filename, 'w') as file:
28:                write_best_simulation_result_to_file(list(assigned_events.values()),
file)
29:        if len(unassigned_events) == 0 and hard_penalty_result == 0 and
soft_penalty_result ≠ 0 then
30:            global_best_soft_penalty ← hill_climber.run_hill_climbing(assigned_events,
                    events[current_node.depth()]["Id"],
                    global_best_soft_penalty, start_time, time_limit)
31:            update_penalties(global_best_soft_penalty)
32:        end if
33:    end if
34:
35:    simulation_result_hard ← normalize_hard(current_node.best_hard_penalty)
36:    simulation_result_soft ← normalize_soft(current_node.best_soft_penalty)
37:
38:    return simulation_result_hard, simulation_result_soft
39: end function
```

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 6 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
```

4.3.3 Normalization

The normalization formula 4.2 is applied to both hard and soft constraints to standardize the simulation results [11]:

$$N(n) = \frac{e^a - 1}{e - 1}, \quad \text{with } a = \frac{best\_penalty_n - worst\_penalty}{best\_penalty - worst\_penalty}, \tag{4.2}$$

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.

4.4. Hill climbing

The HC algorithm is applied as a local search method to further optimize the timetable obtained from the MCTS simulation phase. After finding a feasible solution with MCTS, refines the solution by exploring neighboring states.

(...)

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 frontend.

However, as the focus shifted towards adapting it to the ITC-2007 track 3 benchmark for performance evaluation and fine-tuning the algorithm, the frontend development gradually took a backseat. Over time, the algorithm evolved significantly, diverging from its original form, which made it challenging to align with the prior work.

5. Tests

6. Results

7. Conclusion

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 5 and 6.]

[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 10.]

[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 10 and 11.]

[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 10.]

[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 10, 11, and 12.]

[7] E. Talbi, “Metaheuristics: From design to implementation,” *John Wiley & Sons google schola*, vol. 2, pp. 268–308, 2009. [Cited on pages 10 and 11.]

[8] K.-L. Du, M. Swamy *et al.*, *Search and optimization by metaheuristics*. Springer, 2016, vol. 1. [Cited on pages 10 and 11.]

[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 11.]

[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 page 12.]

[11] 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 23.]