TAP

ISEP – Master's Degree in Software Engineering

Project Report – Viva Scheduling Algorithms

Elaborated by:
Luís Serapicos (1230196)

Telo Gaspar (1231304)

Index

Index 2

1. Introduction........ 3
2. Algorithm First Come First Served – Milestone 1........ 3
3. Algorithm Prefence Maximization – Milestone 3........ 4
4. Domain Properties 5

4.1. Generators........ 5

4.1.1. Generate Three Digit Number 5

4.1.2. Generate Resource Id 5

4.1.3. Generate Roles (President, Advisor, Coadvisor, Supervisor) 5

4.1.4. Generate Unique Resource........ 5

4.1.5. Generate Agenda Duration........ 6

4.1.6. Generate Availability Start........ 6

4.1.7. Generate Availability End 6

4.1.8. Generate Viva Roles........ 6

4.1.9. Generate Viva 6

4.1.10. Generate Intersected Availabilities and Scheduled Viva 6

4.2. Properties 7

4.2.1. All vivas must be scheduled in the intervals in which its resources are available property........ 7

4.2.2. One resource cannot be overlapped in two scheduled vivas property........ 7

4.2.3. The generated availability intervals must contain at least one interval equal to or greater than the viva duration property 7

4.2.4. The generated resources must have unique identifiers property........ 7

4.2.5. The total preference of all scheduled vivas must match the total preference of the agenda property........ 7

4.2.6. The generated availability intervals must not overlap each other property........ 8

5. Functional Design........ 8
6. Testing 9
7. Functional Programming Learning 9
8. Conclusion........ 9

1. Introduction

This project revolves around the scheduling process involving multiple entities and must adhere to various constraints to ensure optimal timing for each viva. At its essence, the project entails developing two algorithms, using functional programming, that iteratively enhance the scheduling process, aiming to achieve an optimal solution.

The scheduling problem addressed is NP-hard, requiring the consideration of constraints, such as the availability of resources, where their roles in the viva should be common, the vivas scheduled should not have overlapped times and after a viva is scheduled, it's resources availabilities should be updated with the new times excluding the viva time.

2. Algorithm First Come First Served – Milestone 1

The first algorithm developed follows a First Come First Served approach. It schedules each viva based on the order they appear in the input data. Each viva is assigned to the latest time slot where all required resources are available.

The FCFS algorithm serves as a first iteration to solve the problema of scheduling vivas, providing a straightforward method to handle it with minimal complexity. This approach ensures that every viva is scheduled as soon as possible, given the constraints of resource availability, but it gives us some limitations, like not giving a schedule that fully satisfies all the resources and it doesn't consider the preference maximization.

The workflow of this algorithm starts with intersecting the availabilities between the required resources with a function that takes a resource from the list of roles that are present in the viva, and checks the availabilities of that resource and the remaining resources if they all have one time slot in common, where that time slot has a range of, at least, the agenda duration time. Given the result of the viva and it's intersected availabilities, we will now find the latest start time from those availabilities, because we are aiming to find the first availability that actually interesects between the suitable availabilities. The process is repeated for each viva till it finishes the schedule, but after each scheduled viva we need to update the resources with new availabilities, because after a viva is scheduled, their resources are no longer available at that time. So we implemented an utility function that does that job, named "fixAvailabilities" that given an availability and the time slot (start and end times) it will either and empty list if the time slot is the same as the availability, return a new availability if the time slot finishes or starts are in the end or start of the availability, or return two availabilities if the time slot is in the middle of the availability. The final step is to generate the agenda with the total preference and the vivas scheduled into a xml file.

3. Algorithm Prefence Maximization – Milestone 3

The second algorithm aims to maximize the total preferences of all scheduled vivas. It evaluates multiple scheduling possibilities and selects the one that provides the highest overall preference score, taking into account the availability and preferences of each resource. Maximizing preferences is crucial for achieving a schedule that is more favorable for all participants.

The implemented algorithm comes with a new approach, where we call a new function, findAllTimeSlots, that returns all the combinations of schedules, with it's preferences, that are possible for a given viva, and with the returned time slots the new function findGlobalPreference will make all the schedule combinations and calculate the totalPreference for each one, and finally return the best one.

This is a brute force approach, so it will obviously come with an increased computational complexity compared with the FCFS algorithm.

4. Domain Properties

In the second milestone, our job was to test edge cases of the first algorithm. We created property-based tests for the viva scheduling problem domain. Property-based testing is a powerful testing methodology that allows us to specify the logical properties our program should always satisfy and then uses these properties to generate test cases.

4.1. Generators

Firstly, we defined some generators. Generators are used to create random data for property-based tests. The defined generators for the domain classes Viva, Availability, and Resource, as well as for the opaque types used in the domain model. These generators are designed to produce valid instances of the domain classes, ensuring that the generated data adheres to the constraints and patterns defined in the domain model.

4.1.1. Generate Three Digit Number

The threeDigitNumber generator is used to generate random three-digit numbers, which are commonly used as identifiers for resources and other entities in the domain model. The generator ensures that the generated numbers are within the valid range of 001 to 999, providing a diverse set of identifiers for testing purposes.

4.1.2. Generate Resource Id

The `resourceId` generator creates random resource identifiers composed of a prefix ('T' for teachers and 'E' for external persons) followed by a three-digit number. This pattern ensures that each identifier is unique and adheres to the specified format, making it suitable for creating valid resource instances for testing purposes. The generated identifiers enable the clear distinction between different types of resources within the scheduling system.

4.1.3. Generate Roles (President, Advisor, Coadvisor, Supervisor)

The `role` generator randomly selects roles from a predefined set (President, Advisor, Coadvisor, Supervisor) to assign to resources involved in vivas. This generator ensures that the roles are representative of the actual roles assigned in the viva scheduling context, providing realistic and valid role assignments for resources during testing.

4.1.4. Generate Unique Resource

The `uniqueResource` generator combines the `resourceId` and `resourceName` generators to create unique resource instances. Each generated resource has a distinct identifier and name, ensuring that the resources used in testing are unique and reflective of real-world data. This generator is essential for creating a diverse set of resource instances for thorough and comprehensive testing.

4.1.5. Generate Agenda Duration

The `agendaDuration` generator creates random agenda durations representing the time allocated for each viva. The generated durations fall within the valid range of 1 hour, 1.5 hours, and 2 hours. This ensures that the generated durations are diverse and representative of the actual timeframes used in viva scheduling scenarios.

4.1.6. Generate Availability Start

The `availabilityStart` generator produces random start times for resource availabilities within the valid range of 08:00 to 18:00. This generator ensures that the start times are representative of typical availability periods for resources, providing realistic data for testing the scheduling algorithms.

4.1.7. Generate Availability End

The `availabilityEnd` generator generates random end times for resource availabilities, ranging from 09:00 to 20:00. This ensures that the generated end times reflect realistic availability periods, providing valid data for testing the scheduling constraints and ensuring that resources are available within the expected timeframes.

4.1.8. Generate Viva Roles

The `vivaRoles` generator combines the `resourceId` and `role` generators to create random role assignments for resources involved in vivas. Each generated role assignment includes a resource identifier and a role, ensuring that the assignments are random yet valid, reflecting the actual role distribution in viva scheduling.

4.1.9. Generate Viva

The `viva` generator creates random viva instances by combining the `vivaStudent`, `vivaTitle`, and `vivaRoles` generators. Each generated viva includes a random student name, dissertation title, and role assignments, ensuring that the generated instances are diverse and reflective of real-world scenarios. This generator is crucial for testing the scheduling algorithms with a variety of viva configurations.

4.1.10. Generate Intersected Availabilities and Scheduled Viva

The `intersectedAvailabilities` generator creates random intersected availabilities for resources, which is used in the early phase of the algorithm to check for overlaps between availabilities. The `scheduledViva` generator produces random scheduled vivas by generating a duration, unique resources (teachers and externals), combining these resources, and selecting a subset. It then generates a list of vivas using these resources and applies the algorithm to schedule them, resulting in a comprehensive list of scheduled vivas along with the associated teachers, externals, and durations. This generator is essential for testing the scheduling process and ensuring that the algorithm correctly handles resource availability and scheduling constraints.

4.2. Properties

The properties we have considered relevant for our domain are:

4.2.1. All vivas must be scheduled in the intervals in which its resources are available property

This property ensures that a viva is only scheduled when all its required resources are available. It helps to maintain the integrity of the scheduling process by ensuring that no viva is scheduled at a time when any of its required resources is unavailable.

4.2.2. One resource cannot be overlapped in two scheduled vivas property

This property ensures that a resource is not double-booked. It helps to prevent scheduling conflicts by ensuring that a resource is not assigned to more than one viva at the same time.

4.2.3. The generated availability intervals must contain at least one interval equal to or greater than the viva duration property

This property ensures that the availability intervals generated for resources contain at least one interval that is equal to or greater than the duration of the viva. It helps to ensure that the scheduling algorithm can find a suitable time slot for the viva within the availability intervals of the resources.

4.2.4. The generated resources must have unique identifiers property

It helps to prevent conflicts and inconsistencies in the scheduling process by ensuring that each resource is uniquely identified and do not appear more than once in the same viva.

4.2.5. The total preference of all scheduled vivas must match the total preference of the agenda property

This property ensures that the total preference of all scheduled vivas matches the total preference of the agenda.

4.2.6. The generated availability intervals must not overlap each other property

This property ensures that the generated availability intervals for resources do not overlap with each other. It helps to prevent scheduling conflicts by ensuring that each resource is available at distinct time intervals.

We consider these properties relevant to our domain as they help to check the scheduling of vivas. They were implemented to catch potential scheduling conflicts and inconsistencies, improving the reliability of the scheduling algorithm.

5. Functional Design

The functional programming techniques were improved in the milestone 3, after the milestone 1 were lacking, like more reusable functions because the implemented algorithm FCFS was quite complex, so that even testing was not that easy because of the need to build large test cases, but within this milestone we can test each function and know earlier the results that it gives and if those are the desired ones.

As of other functional designs, pattern matching was still highly used in both algorithms since it is powerful dealing with different cases or conditions like case classes and enums, helping the code to behave as expected because it is type safe, replacing the common if statements.
Immutable data types were also used, since immutability ensures that data structures are not modified after creation, leading to more predictable and easier to debug code.

6. Testing

Several unit tests for different components of the scheduling system were written to cover domain errors, XML processing, file I/O operations, simple types validation, and XML to domain conversion and algorithms functions in both FCFS and Preference Maximization.
For the milestone 3, the tests were improved in terms of organization. Different files were created to implement tests for the different components in each.

7. Functional Programming Learning

The use of higher-order functions, like foldLeft, is a clear example of the difference between functional and imperative programming in Scala. These functions enable a declarative style of programming that focuses on what to compute rather than how to compute it.

The foldLeft function allows us to aggregate or reduce a collection of elements into a single result in a very expressive and declarative way. This function takes an initial accumulator value and a binary operator function, then it applies this function to the elements of the collection from left to right, carrying the result forward.
For example, in the scheduleAllVivas function we use a foldLeft function to schedule each viva, as it iterates through the list of vivas to carry a list of updated resources and a scheduled viva, so that after the final iteration it returns a full schedule of vivas.

Overall, the use of functional programming in Scala was difficult but it really makes us look at another way to solve a complex problem in a easier way, by writing simpler and more straightforward code.

8. Conclusion

Even though we implemented a brute force algorithm, we made researches about other ways to solve the global maximization preference problem. We reunited studies from brute force, greedy, Hopcroft–Karp, Hungarian and Weighted Interval Scheduling algorithms to see which was the best fit between complexity and problem solving. We decided to use the brute force approach since: a greedy approach could not give the optimal global preference in every case; the Hopcroft–Karp algorithm was graph based but didn't have weights so we couldn't really use all the constraints; the Hungarian algorithm is one-to-one matching, and because we have resources with multiple availabilities, this was not possible; the Weighted Interval Scheduling algorithm was almost the best choice, but the need to create multiple tables to make the calculations preferences was almost the same as a brute force approach.