UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

INCREMENTAL ALGORITHMS FOR LARGE PROBLEMS

POL FORNER GOMEZ

Thesis supervisor
GERARD ESCUDERO BAKX (Department of Computer Science)

Thesis co-supervisor
EDELMIRA PASARELLA SANCHEZ (Department of Computer Science)

Degree
Bachelor's Degree in Informatics Engineering (Computing)

Bachelor's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

01/07/2024

Contents

List of Figures

List of Tables

Abstract

This work is a final project for a Bachelor's degree in Computer Engineering, specializing in Computing. The project is completed at the Universitat Politècnica de Catalunya (UPC), specifically at the Facultat d'Informàtica de Barcelona (FIB). Gerard Escudero Bakx directs the project, with supervision from Edelmira Pasarella Sanchez. The project builds upon the master's thesis completed by Royo-Sales et al. [1] in 2021,supervised by Edelmira Pasarella Sanchez. Royo-Sales et al. [1] thesis focused on developing a Haskell library for the dynamic pipeline paradigm and its application to an algorithm for incremental enumerating bitriangles (IEBT). This work involves understanding Royo-Sales et al. [1] work, including the Haskell library, its potential application to other algorithms, and the implementation of the IEBT algorithm. The project aims to provide a guide for utilizing the library in implementing any algorithm that leverages the dynamic pipeline paradigm. Additionally, it proposes a set of improvements to the library itself. Furthermore, the project explores potential enhancements to the IEBT algorithm, accompanied by performance tests to validate these improvements.

Este trabajo es un proyecto final del Grado en Ingeniería Informática, con especialidad en Computación. El proyecto se realiza en la Universitat Politècnica de Catalunya (UPC), específicamente en la Facultat d'Informàtica de Barcelona (FIB). Gerard Escudero Bakx dirige el proyecto, con la supervisión de Edelmira Pasarella Sanchez. El proyecto se basa en la tesis de máster realizada por Royo-Sales et al. [1] en 2021, supervisada por Edelmira Pasarella Sanchez. La tesis de Royo-Sales et al. [1] se centró en desarrollar una librería de Haskell para el paradigma dynamic pipeline y su aplicación a un algoritmo para la enumeración incremental de bitriángulos (IEBT). Este trabajo implica comprender el trabajo de Royo-Sales et al. [1], incluyendo la librería de Haskell, su potencial aplicación a otros algoritmos y la implementación del algoritmo IEBT. El proyecto tiene como objetivo proporcionar una guía para utilizar la librería en la implementación de cualquier algoritmo que se base en el paradigma de canalizaciones dinámicas. Además, propone un conjunto de mejoras a la propia librería. Finalmente, el proyecto explora posibles mejoras del algoritmo IEBT, acompañadas de pruebas de rendimiento para validar dichas mejoras.

Aquest treball és un projecte final per a un Grau en Enginyeria Informàtica, amb especialitat en Computació. El projecte es realitza a la Universitat Politècnica de Catalunya (UPC), específicament a la Facultat d'Informàtica de Barcelona (FIB). Gerard Escudero Bakx dirigeix el projecte, sota la supervisió d'Edelmira Pasarella Sanchez. El projecte es basa en la tesi de màster realitzada per Royo-Sales et al. [1] en 2021, supervisada per Edelmira Pasarella Sanchez. La tesi de Royo-Sales et al. [1] es va centrar en desenvolupar una biblioteca Haskell per al paradigma dynamic pipeline i la seva aplicació a un algorisme per a l'enumeració incremental de bitriangles (IEBT). Aquest treball implica comprendre el treball de Royo-Sales et al. [1], incloent-hi la biblioteca Haskell, la seva potencial aplicació a altres algorismes i la implementació de l'algorisme IEBT. El projecte té com a objectiu proporcionar una guia per a utilitzar la biblioteca en la implementació de qualsevol algorisme que es basi en el paradigma de canalitzacions dinàmiques. A més, proposa un conjunt de millores a la pròpia biblioteca. Finalment, el projecte explora possibles millores de l'algorisme IEBT, acompanyades de proves de rendiment per a validar aquestes millores.

Context and Scope

To initiate this work, this chapter will address all the necessary concepts to understand the work. It will also detail the objectives and requirements, as well as a justification for the work.

1.1 Context

1.1.1 Introduction of concepts

Nowadays, the volume of data generated is immense and, more importantly, it grows continuously. Sensors, social networks, and other sources contribute to this ever-increasing data that necessitates processing and analysis to extract valuable information. A critical challenge arises when processing this data: traditional methods employed for smaller datasets become inapplicable due to the excessive time and resources required. To address this issue, the development of novel algorithms and techniques specifically designed to handle such large data volumes is essential.

Streaming

In addition to having all that data, the amount of data normally it can not be stored and it comes in what is called a data stream. A data stream is a sequence of data that made available over time, meaning that we can not store all the data and we need to compute it in time. For example, a sensor of temperature that sends the temperature every second, a traffic camera that registers all car plates that pass in front of it or just a social network that generates a huge amount of data every second.
This kind of data mentioned before needs to be processed and sometimes the data never ends, so we can not wait to finish to give a result and we must give results along the way.

Incremental algorithms

Here is where incremental algorithms come in. Incremental algorithms give us the ability to obtain results from subsets of data and then update the results before finishing the whole data. This is very useful, because some problems do not need to be solved with all the data or maybe we are not interested in the final result. Recovering a previous example, if we are interested in which models of cars drive in certain road, we can use the camera that registers the car plates to get the answer. It is stupid to wait until the end of data to give the answer (also because it never ends), so we can give a result when we check it. In conclusion, incremental algorithms could be a good approach to solve some problems.

Paralelism

One of the most important techniques for dealing with time problem is parallel computing or parallelism. Parallelism allows us to divide the work and process it in different machines concurrently, reducing the time needed to process the data. When we try to fight against this huge data problems, we must find a solution that can be parallelized given that modern machines have multiple cores and we can use them to process the data.
If we put together streaming and parallelism, it can be distinguished two computational models:

- **Data Parallelism**
  This model splits the data and processes it in parallel. All the computations that perform some action over a subset of data, do not have any dependency with other parallel computations. This model has the advantage that it can implement stateless algorithms, allowing to split and process the data into different machines without contextual information. Nonetheless, this model has the disadvantage that when we need to be aware of the context, it is penalized.

- **Pipeline Parallelism:**
  This model splits the computation in different stages and each stage takes the result of the previous stage to make the computation. The parallelization is done by parallelizing the stages. The main advantage is that stages are non-blocking, meaning that we do not need to process all data to execute the next stage. This allows us to make incremental algorithms. In spite of that, the main disadvantage is that one stage could be the bottleneck of the pipeline and delaying all the process.

Dynamic Pipeline Paradigm

Now that we talked about these 3 concepts: incremental algorithms, streaming and parallelism, it can be introduced the next concept that will be the main focus of this project. The Dynamic Pipeline Paradigm [2] is a Pipeline Parallelism model "*based on a one-dimensional and unidirectional chain of stages connected by means of channels synchronized by data availability*". Royo-Sales et al. [1, Page 9, 2.2] This chain is called Dynamic Pipeline and it can grow and shrink with the continuous arrival of data. So we can use a data stream to feed the pipeline and it will process the data growing and shrinking the as needed.
With this paradigm we can implement incremental algorithms and process the data in parallel easily.

A more in-depth exploration of this concept will be provided later in this work, accompanied by a detailed explanation. 6

1.1.2 Problem to be solved

In his work, Royo-Sales et al. [1] implemented an incremental algorithm using dynamic Pipeline Paradigm to resolve a graph problem: finding bitriangles in bipartite graphs. He decided to implement the algorithm using the functional programming language Haskell, and because of the no existence of one, he created a framework to implement the Dynamic Pipeline paradigm. This marks the commencement of the present work. The primary objective is to build upon and enhance Royo-Sales et al. [1] contributions. Initially, a thorough study of Royo-Sales et al. [1] library implementation will be undertaken to gain proficiency in its usage. Subsequently, a practical application of the problem will be conducted using a toy problem (a simplified problem) to master the library's functionalities and identify potential improvements and extensions. Finally, the implementation of the IEBT algorithm will be refined leveraging the acquired knowledge.

1.1.3 Stakeholders

This project has the potential to attract a wide range of stakeholders. Firstly, it is of interest to the Haskell user community, as this project introduces enhancements to a Haskell library. While the official download page for the library does not reflect a substantial number of downloads, the programming world highly values the availability of examples and diverse developments to streamline the programming process. Currently, it is challenging to find sample code for this library beyond Royo-Sales et al. [1] work. Secondly, the research community utilizing the Dynamic Pipeline paradigm stands to benefit. When developing an algorithm to address a problem using this paradigm, implementation is often necessary for testing and evaluation purposes. Therefore, this project aims to facilitate this task by providing a guide for implementing such problems.

1.1.4 Justification

Having established the existence of stakeholders for this project, the question arises as to whether it is feasible to enhance the library and refine the IEBT algorithm. Several factors support this assertion.

Firstly, and most significantly, Royo-Sales et al. [1] himself acknowledged the potential for improvements in both the Haskell library and the algorithm within his own work. Secondly, discussions were held with the co-director of this project, Edelmira. She possessed the most profound knowledge of the work, having supervised Royo-Sales et al. [1] research and, along with her team, actively working within the Dynamic Pipeline paradigm. She affirmed the algorithm's merit for further development and the potential for refinements. These enhancements revolve around optimizing vertex storage and set operations within the algorithm.

In light of these considerations, the justification for undertaking this project is unequivocal.

1.2 Scope

This section will delve into the objectives and scope of the work. The various requirements, potential obstacles, and risks that may arise will be identified. The work methodology that will be followed throughout the project's development will also be detailed.

1.2.1 Objectives

This project has the following main objectives:

- **Toy problem:** The first goal is to learn how to use the library and develop my own algorithm and implementation for a problem.
- **Improving DP Haskell Library:** The second goal of this project is to improve the Haskell library made by Royo-Sales et al. [1].
- **Improve IEBT algorithm:** The third goal is to improve the implementation of the algorithm mentoined before.

Apart from these main objectives, I have some sub-objectives that I would like to achieve at the end of the project:

- Learn about the dynamic pipeline paradigm.
- Improve my Haskell skills, from my actual basic level to a competent level.
- Learn about the inner workings of the Haskell programming language.
- Acquire knowledge of how to do a bachelor degree project for future projects like master thesis.

1.2.2 Requirements

For the correct development and validity of my final result, it is necessary that my final implementation solves all cases correctly and more efficiently on average than the implementation from which we started. It is also important to follow a correct and modular structure to facilitate possible modification. My optimization has to be in line with known improvements and must be correctly validated. My solution also has to be understandable and adaptable, in order to improve its reuse and sharing. As for the code, I have to make sure it is well documented. Finally, it is necessary that all Haskell functionalities used are updated and supported

1.2.3 Potential obstacles and risks

Its needed to identify the potential obstacles and risks that may arise during the project. This is important to be able to anticipate and mitigate them. Here are the principal ones that I have identified:

Time Limit

I will say that this is one of the most important risks, because as there is a deadline to finish the project, an inconvenience can make to not finish the project. Poor time management could necessitate a reassessment of the project's scope and a potential reduction in its deliverables. Nevertheless, this potential risk is acknowledged, and a corresponding plan will be developed, breaking down tasks into smaller, more manageable units to accommodate changes and avoid an empty-handed conclusion.

Dynamic pipeline framework

I did not use and examined the framework and a problem that worries me is that the framework has some bugs or is not well implemented. This can potentially make me lose a lot of time but as I check, Pablo did a good work documenting it so I trust that I will not have a lot of problems.

1.3 Methodology and rigor

1.3.1 Methodology

For this project, I will following a methodology based on the Scrum methodology, as I have always work with it and I have had good results. Originaly, Scrum was designed for team work, but I will going to take the idea and adapt it to my project, that only I do it individually.

Scrum is a methodology that is based on the iterative and incremental development of the project, where the project is divided into small tasks that are done in a short period of time called sprint. This sprints have 3 main phases: planning, development and review. Taking this concepts into acount, I will divide the project into tasks and I will be following 1 week sprints. It will help me keep good control of the pace of the project and not fall asleep or fall behind. Apart from the experience using this methodology, another of the main reasons for using it is the potential to redirect the project in case of possible problems, as I consider crucial.

1.3.2 Project monitoring and validation

As good programming practices, I will be using git to control the versions of the code. This will allow me to have a backup of the code and a practical way to a acces to previous versions. Also, I will be using Trello as a task manager, to keep track of the tasks and the progress of the project. Trello is very useful and combines very well with the methodology that I have chosen, since I will be able to have a record of the tasks to be done and the tasks completed.

To validate all about the Haskell work, I will be asking and following Gerard advice. Since Gerard has a great level of Haskell language, he will be able to help me validate that I follow a good structure and use of Haskell. Finally, facing the end of the project, Edelmira will help me with the correction and validation of the improvement and implementation of the algorithm.

Planning

2.1 Description of tasks

Before starting, I would like to thank my dear friend Alex Herrero to share with me his Latex project that he did for GEP months ago [3]. I will use his Gantt chart as template for mine, so I will not need to spend time in the creation and investigation of pgfgantt package.

This project officially started the second week of February 2024 although some mails and meets were done before with supervisor and co-supervisor. I will not include this previous work in the planning, as I consider not necessary. This work will be done during the following months, until the third week of June 2024, approximately one week before the oral defense. This is a total of 17 weeks and TFG is an 18 ECTS project, so it will need 450 hours of work, 26 hours per week approximately. Here I will define the principal sections of my project, with the possible resources needed and the minimum hours needed:

2.1.1 Project definition (G)

This section includes everything done in GEP, since its completion is mandatory and has set deadlines. So, all the following task are documentation and only requires my personal laptop since I have the entire environment required to develop in LaTeX. I also may need to contact my supervisor to check some details and ask for some doubts and also check for my GEP tutor feedback of every deliverable. This task has a linear dependency (**see Figure 2.1**) because of GEP structure (in reality the only dependence is that G4 needs to be done the last one).

- **G1** Context and Scope
  This corresponds to the first deliverable of GEP, where is defined the context and scope of the project.
  Role: Project Manajer
  Expected dedication: 20 hours

- **G2** Planning
  This corresponds to the second deliverable of GEP, where is defined the planning of the project.
  Role: Project Manajer
  Expected dedication: 12 hours

- **G3** Budget and Sustainability
  This corresponds to the third deliverable of GEP, where is defined the budged and sustainability of the project.
  Role: Project Manager
  Expected dedication: 18 hours

- **G4** Final Document
  This corresponds to the last deliverable of GEP, where contains all the previous deliverables together with the necessary improvements using the feedback.
  Role: Project Manajer
  Expected dedication: 25 hours

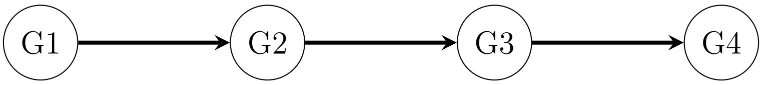


Figure 2.1: This graph represents the task dependencies for project definition, self elaborated

2.1.2 Research and Learning (RL)

In this section is included all the research and learning that I need for the correct development of the project. Here I will only need access to internet to search for information and access to the different pages and documents that I will be using. The principal internet resources that I will use are: Royo-Sales et al. [1], Haskell notes made by Jordi Petit from the subject LP and the Dynamic Pipeline Framework repository. I may also need to contact my supervisor for Haskell doubts and also Royo-Sales et al. [1] if some critical doubt appears. This section also have some dependence (**see Figure 2.2**) because of the nature of the tasks.

- **RL1** Haskell Refresh
  The refresh of my Haskell knowledge and improvement of it. Check for possible libraries and codes
  Role: Project Manajer
  Expected dedication: 20 hours

- **RL2** TFM assimilation
  The reading and understanding of Royo-Sales et al. [1] work.
  Role: Researcher
  Expected dedication: 25 hours

- **RL3** Dynamic Pipeline Framework
  Intensive review of the framework repository.
  Role: Haskell Developer
  Expected dedication: 30 hours

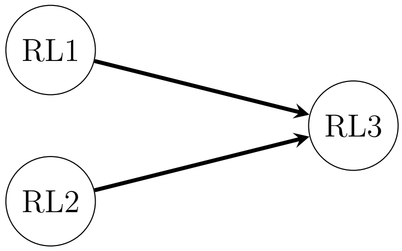


Figure 2.2: This graph represents the task dependencies for research and learning, self elaborated

2.1.3 First Algorithm Development (FA)

This section includes the development of the first algorithm, the one that will be put to test my Haskell knowledge acquired. Here I will need my personal laptop (as I have the entire environment required) and access to the GitHub repository [4]. For all the Haskell doubts I may need to contact my supervisor and for Dynamic pipeline doubts I should contact my co-supervisor as she is the expert. This task has a linear dependency (**see Figure 2.3**), but a critical testing error may need to reimplement some parts of the code, generating a circular dependency.

- **FA1** Algorithm Scaffold
  Here will be set the base of the algorithm and the form of the dynamic pipeline. Check for possible libraries and codes
  Role: Haskell Developer
  Expected dedication: 20 hours

- **FA2** Algorithm Implementation
  This part is the pure implementation of the algorithm.
  Role: Haskell Developer
  Expected dedication: 50 hours

- **FA3** Algorithm Testing
  Testing of the algorithm implementation, littles changes are also contemplated here (no big changes)
  Role: Haskell Developer
  Expected dedication: 10 hours

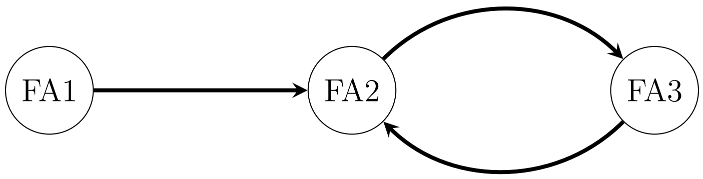


Figure 2.3: This graph represents the task dependencies for first algorithm development, self elaborated

2.1.4 Second Algorithm Development (SA)

This is the final section and the goal of this work, here I will try to improve the original algorithm. Again, here I will only need my personal laptop and access to internet resources. This section will need a lot of contact with my supervisor and co-supervisor, specially my co-supervisor, who is the expert in the algorithm. Here the dependence are similar to the previous section (**see Figure 2.4**), as is also a development of an algorithm. This task is especial, because it is difficult to estimate the exact scope of my project, so maybe I make some improvement and then have more time to repeat this process and make another improvement.

- **SA1** Finding Weak Points
  Here I will be looking for possible improvements in the code.
  Role: Haskell Developer and Researcher
  Expected dedication: 15 hours

- **SA2** Improvement Scaffolds
  Here will be set the base for the improvements of the algorithms find in the previous task Check for possible libraries and codes
  Role: Haskell Developer
  Expected dedication: 15 hours

- **SA3** Algorithm Implementation
  This part is the pure implementation of the algorithm.
  Role: Haskell Developer
  Expected dedication: 80 hours

- **SA4** Algorithm Testing
  Testing of the algorithm implementation, little changes are also contemplated here (no big changes). Here more time is needed in comparation of previous section as the size of the code and algorithm
  Role: Haskell Developer
  Expected dedication: 20 hours

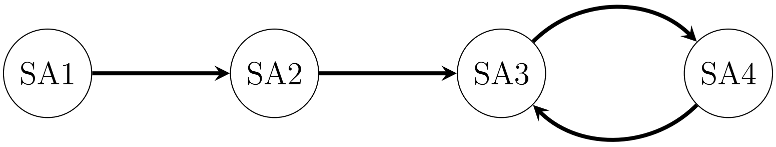


Figure 2.4: This graph represents the task dependencies for second algorithm development, self elaborated

2.1.5 Documentation (D)

Finally, this last section is a bit special because it is not planned to be done at the end of the all the previous sections. It is planned to be done in parallel as the project is developed and the different sections are finished. Here, all I need is my personal laptop to write all the documentation using LaTeX. Here I may need to contact for some help and feedback, but is not mandatory. There are no direct dependence, but we can consider that we need to complete one section or task before starting to write, so we can consider a graph like shown below (**see Figure 2.5**).

- **D1** Documentation of RL section
  This corresponds to the documentation of the research and learning section.
  Role: Project Manager
  Expected dedication: 10 hours

- **D2** Documentation of FA section
  This corresponds to the documentation of the first algorithm section.
  Role: Project Manager
  Expected dedication: 20 hours

- **D3** Documentation of SA section
  This corresponds to the documentation of the second algorithm section.
  Role: Project Manager
  Expected dedication: 20 hours

- **D4** Final Documentation
  This corresponds to the union of all the previous parts, with the additional parts (conclusions, bibliography, etc.) and the revision of the entire document.
  Role: Project Manager
  Expected dedication: 50 hours

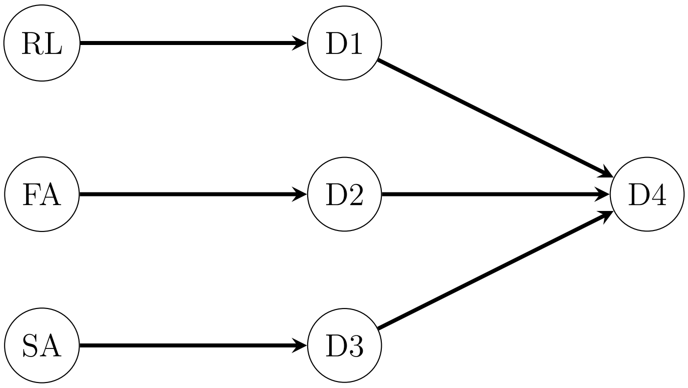


Figure 2.5: This graph represents the task dependencies for documentation, self elaborated

2.2 Estimations and Gantt chart

2.2.1 Summary table

Now that we defined all the tasks, minimum hours and dependence, we can make this following summary table:

Description	TAG	Hours	Previous Tasks	Requirements	Human Resource
Project definition and planning	**G**	**75**	**-**	**-**	**-**
Context and Scope	G1	20	Laptop	None	None
Planning	G2	12	G1	Laptop	GEP Tutor
Budget and Sustainability	G3	18	G2	Laptop	GEP Tutor
Final Document	G4	25	G3	Laptop	GEP Tutor
Research and Learning	**RL**	**75**	**G**	**-**	**-**
Haskell Refresh	RL1	20	None	Laptop, Books	Supervisor
TFM assimilation	RL2	25	None	Laptop	Supervisors
Dynamic Pipeline Framework	RL3	30	RL1, RL2	Laptop	Royo-Sales et al. [1]
First Algorithm Development	**FA**	**80**	**RL**	**-**	**-**
Algortim Scaffold	FA1	20	None	None	Supervisors
Algorith Implementation	FA2	50	FA1	Laptop	Supervisor
Algorithm Testing	FA3	10	FA2	Laptop	None
Second Algorithm Development	**SA**	**130**	**FA**	**-**	**-**
Finding Weak Points	SA1	15	None	Laptop	Co-supervisor
Improvement Scaffolds	SA2	15	SA1	None	None
Algorith Implementation	SA3	80	SA2	Laptop	Supervisor
Algorithm Testing	SA4	20	SA3	Laptop	None
Documentation	**D**	**100**	**RL,FA,SA,**	**-**	**-**
Documentation of RL section	D1	10	RL	Laptop	None
Documentation of FA section	D2	20	FA	Laptop	None
Documentation of SA section	D3	20	SA	Laptop	None
Final Documentation	D4	50	D1, D2, D3	Laptop	Supervisors
Total (G + RL + FA + SA + D): 460 hours					

Table 2.1: This is a table summary of the project planning, self elaborated

2.2.2 Gantt chart

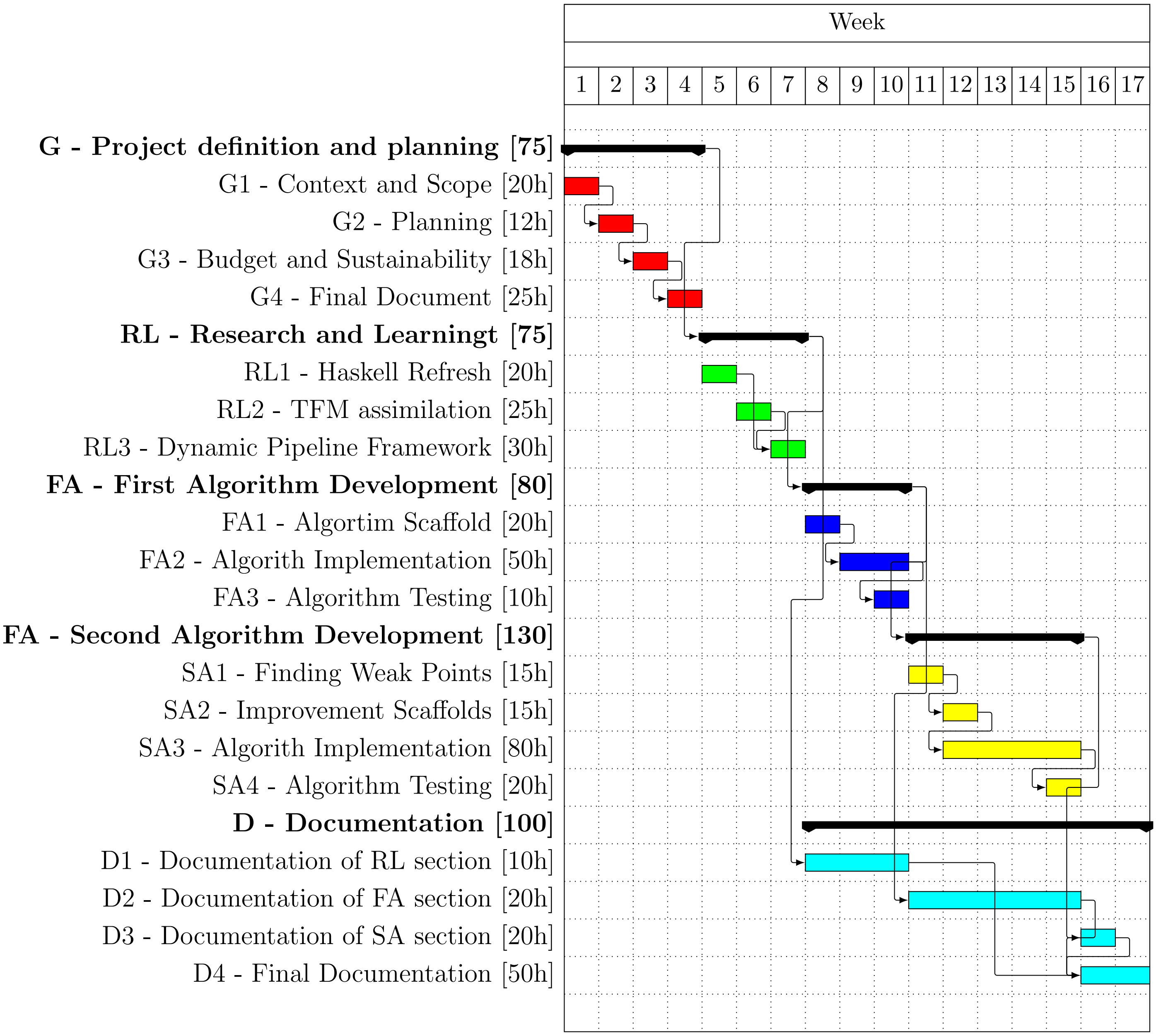


Figure 2.6: This is the Gantt diagram of this project, self elaborated

2.3 Risk management: obstacles and alternatives

As mentioned in previous sections of this work, there are some potential risks that may appear during the development of the project. Here will be explained how I will manage them and the alternatives that could be done in case of critical obstacles.

2.3.1 Obstacles

Time deadline

As said previously, this is one of the most important risks of the project. There is not a lot of time, but I have some tools to deal with this problem. Firstly, the main objective is to obtain more time if needed and my principal activity (and the one that spends more time) is my job as intern. I have flexibility in my schedule and I can reorganize my week to spend more time in the project. Also, I have some holidays that I can use to spend more time in the project.

Dynamic Pipeline Framework

I explained that I never used before this framework and I may have some problems with it or it could have some bugs or limitations. The principal tool to deal with is to contact with his creator, Royo-Sales et al. [1]. Also, I can spend more time studying the framework because I saved some time in case of this obstacle. This problem could potentially add from 10 to 30 hours to the project.

2.3.2 Alternatives

If some of the previous obstacles appear and I can not deal with them, I could be forced to make some changes in the project. So the principal alternative is to reduce the scope of the project, reducing the time spent in the last section: the improvement of the algorithm. This part is the longest of the project and it is planned to be able to be reduced if needed. It is very comfortable, since I do not need any additional resource and I can adapt it to the time that I have left.

Budget and Sustainability

3.1 Budget

In this section I will identify the costs of the project. I will break down the costs into different categories, then I will estimate the costs of each one and finally I will propose a management control system to monitor the costs.

3.1.1 Identification of costs

Here are all the cost group by categories that I will consider in this project.

Hardware

In my project the only hardware that is needed is a computer. We need a computer for all the process: research, programming, writing the documentation, etc. If we needed to test our algorithm with a large dataset, we maybe would need a computer with a high performance or a more specialized hardware, but in this work I will not consider this case. I will consider a generic computer / laptop.

Software

In this project is only planned to use a few software tools. Starting with the programming language, I will use Haskell. Haskell has a compiler called Glasgow Haskell Compiler (GHC) and would be sufficient for the project. I'm also using Visual Studio Code as a text editor, for both writing the code and the documentation. The next software that I will use is LaTeX, for writing the documentation. I also will use GitHub for the version control of the project, that uses Git.

Human resources

Truthy I'm the only human resource that is working in this project, but I will consider me as a person assuming different roles. As mentinoded in the planning section, I will be taking 3 diferents roles:

- Researcher: For tasks RL2, SA1 and some documentation
- Haskell Developer: For tasks RL3, all FA tasks and all SA tasks
- Project Manager: For all G and D tasks

Other costs

I will not consider other costs as the cost of the internet connection, the electricity and the cost of the paper and ink for printing the documentation. As I consider very specific costs and do not have a huge impact in the total cost. Other cost that could be considered is the cost of the transportation, cost of the supervisor and co-supervisor work and other stuff that could be needed for the project. As I consider that these costs are difficult to estimate, I will not consider them in this project.

3.1.2 Cost estimates

Now that all costs are identifies, here I will estimate the cost of each one, considering the duration of the project and the cost of each resource.

Hardware

I will consider a standard price for a good laptop, that is around 1000 euros. Obviously a laptop have a longer life than the project duration, so I will consider the cost of the laptop as a cost that is distributed in time, amortization.

$$\text{Amortization} = \text{Laptop Cost} \cdot \frac{\text{Project Duration}}{\text{Laptop Lifespan}}$$

Normaly, laptops have a lifespan of 3 to 5 years with a avegare of 40 hours per week, so I will consider 4 years, 40 hours per week as the lifespan of the laptop. Also I will consider that the whole project will need the laptop, so there will be a total of 460 hours.

$$\text{Amortization} = 1000€ \cdot \frac{460\text{hours}}{4\text{years} \cdot 52\frac{\text{weeks}}{\text{year}} \cdot 40\frac{\text{hours}}{\text{week}}} = 55,29€$$

Software

All the software that I will use is free. So the total amount of money spend in software will be 0€.

Human resources

I will be tacking the average salary of each role from glassdoor [5] [6][7], a well known website for job search and salary information. We will check Spain salaries and we will consider 2000 hours per year. Here we can see the cost of the human resources, assuming a social security multiplier of 1.3.

Role	**Avg. Salary (€/h)**	**Hours**	**Total (€)**	**Total with Social Security (€)**
Researcher	15	85 hours	1275	1657.5
Haskell Developer	18	200 hours	3600	4680
Project Manager	20	175 hours	3500	4550
Total	18.2	460	8375	10887.5

Table 3.1: This is a table summary of the human resources costs, self elaborated

Risk plan

Some risks may appear during the project, so I will add some extra money to the budget to cover it.

1. Laptop damage: The laptop could be damaged and need to be repaired. I will consider 100€ for this risk.

2. Extra hours: The project could take more time than expected and each hour of the workers is so expensive. Considering 10 % more time, gives us 837 €, so I will add 1000€ to the budget.

3. For other risks and unexpected costs, I will add 10 % of the total budget, to cover it, 1100 €.

Puting all together, the total amount of money for the unexpected costs is 2037 €.

Total budget

Lets sumarize all the costs and calculate the total budget.

Source	**Cost (€)**
Hardware	55,29
Software	0
Human Resources	10887.5
Unexpected	2037
Total	**12979.79**

Table 3.2: This is a table summary of the total budget, self elaborated

3.1.3 Management control

Once we have our budget, I can create a indicator so I can monitor the deviation from the initial plan. This indicator (C) will be the difference between the expected and actual cost:

$$C = C_estimated - C_real$$

I can use this indicator both in a general way and with each subsection of the budget, in order to be able to analyze what may be causing a variation in the budget and be able to fix it, either by modifying the budget itself or by restructuring the project.

3.2 Sustainability report

3.2.1 Economic Dimension

After analized my project and made the budget, we can observe that the main cost of the project is the human resources. Asuming that my plannign is correct, I can confirm that no unnecessary costs are being generated, beacause I'm using free software and the minimun hardware required. Also this project have the goal of improving an implementation of an algorithm, meaning a possible improvement in the efficiency of the algorithm, which could lead to a reduction in time, which means a reduction of the cost.

3.2.2 Environmental Dimension

As mentioned before, the only environmental impact of the project is the energy consumed by the laptop. The energy and impact are minimum, so we can consider this project as environmentally friendly. Also as previous point, the project could lead to a reduction in time, which means a reduction of the energy consumed. A good point about projects like mine is that their useful life can be considered 'unlimited'. This is because it is a research and development project, providing utility at all times.

3.2.3 Social Dimension

In personal terms, I think it will have a great impact on my life and career as a student, since this project will be the first 'big project', which will help me learn from mistakes for future projects. This project may not have a big social impact, but it can do your part in the world of computing and especially in the world of Haskell programming.

3.2.4 Sustainability Matrix

	PPP	**Useful Life**	**Risks**
Economic	12979.79 €	No direct benefits	None
Environmental	9 / 10	More efficient future implementations	None
Social	5 / 10	6 / 10	None

Table 3.3: Sustainability matrix

Laws and regulations

Consulting the BOE, law 14/2011, updated on 01/11/2023, article 15 (page 28) tells us about various obligations of the researcher and here I will comment on those that involve me **empty citation**

- **Avoid plagiarism and misappropriation of authorship of scientific works or third-party technologies**
  On this point, I have at all times cited other people's resources and whenever I have used a third-party technological resource I have made sure that it is open source and that its necessary citations are made.

- **Inform the entities for which it provides services of all the findings, discoveries and results susceptible to legal protection, and collaborate in the processes of protection and transfer of the results of their investigations**
  At all times I have been in contact with my director and have been reporting on my work.

- **Disseminate the results of your investigations, if applicable, as indicated in this law, so that the results are used through communication and transfer to other research, social or technological contexts, and if applicable, for their marketing and valuation. In particular, the research staff must ensure and take the initiative so that its results generate social value**
  This work will be public once finished and everyone will be able to take advantage of the discoveries.

- **Ensure that your work is relevant to society**
  This work can help expand resources on dynamic pipeline and Haskell code.

- **Use the name of the entities for which you provide services in the realization of its scientific activity, in accordance with the internal regulations of said entities and the agreements, pacts and conventions that they subscribe**
  On the cover of this work you will find at all times the name of the entity (UPC) and the faculty (FIB) as well as their logos.

- **Adopt the necessary measures to comply with the applicable regulations in matters of data protection and confidentiality**
  At all times, we are ensuring compliance with data protection, always using data that is free to use and that complies with the laws.

Updates

5.1 Updates respect initial planning

5.1.1 About tasks and time planification

Changes

Regarding the initial planning of the project, there have not been very big changes. The main change has been that I have needed more investment of time in understanding the library and in coding the word counting problem. This is because I have found the need to make several adaptations to the library to allow more options and be able to implement my initial problem. Everything and so I have only needed 2 extra week in the overall time of the **RL** and **FA** sections.

In addition to this, I have been able to specify with Edelmira what improvements I can make to the library. This process was also planned for later, so it also changes the planning a bit.

Consequences

All these changes in planning mean that there is less time left to perform the following tasks (SA, D3, D4). But at the same time I have invested more time getting to know the library and programming, so I foresee that it will take me less time to improvements to the library that Edelmira proposes.

New planning

With this updates, the new planning is as follows:

Description	**TAG**	**Hours**	**Previous Tasks**	**Requirements**	**Human Resource**
Project definition and planning	**G**	**75**	**-**	**-**	**-**
Context and Scope	G1	20	Laptop	None	None
Planning	G2	12	G1	Laptop	GEP Tutor
Budget and Sustainability	G3	18	G2	Laptop	GEP Tutor
Final Document	G4	25	G3	Laptop	GEP Tutor
Research and Learning	**RL**	~~75~~ 95	**G**	**-**	**-**
Haskell Refresh	RL1	20	None	Laptop, Books	Supervisor
TFM assimilation	RL2	~~25~~ 15	None	Laptop	Supervisors
Dynamic Pipeline Framework	RL3	~~30~~ 60	RL1, RL2	Laptop	Juan Pablo
First Algorithm Development	**FA**	~~80~~ **100**	**RL**	**-**	**-**
Algortim Scaffold	FA1	20	None	None	Supervisors
Algorith Implementation	FA2	~~50~~ 70	FA1	Laptop	Supervisor
Algorithm Testing	FA3	10	FA2	Laptop	None
Second Algorithm Development	**SA**	~~130~~ **90**	**FA**	**-**	**-**
Finding Weak Points	SA1	~~15~~ 10	None	Laptop	Co-supervisor
Improvement Scaffolds	SA2	~~15~~ 10	SA1	None	None
Algorith Implementation	SA3	~~80~~ 50	SA2	Laptop	Supervisor
Algorithm Testing	SA4	20	SA3	Laptop	None
Documentation	**D**	**100**	**RL,FA,SA,**	**-**	**-**
Documentation of RL section	D1	10	RL	Laptop	None
Documentation of FA section	D2	20	FA	Laptop	None
Documentation of SA section	D3	20	SA	Laptop	None
Final Documentation	D4	50	D1, D2, D3	Laptop	Supervisors
Total (G + RL + FA + SA + D): 460 hours					

Table 5.1: This is a table summary of the new project planning, self elaborated

5.1.2 About scope

Changes

Since initially planning the scope of my work, I have had to modify it as I have progressed. The reason is that when working with the Haskell library, I have encountered some lack of functionality at some points. I have found that Juan Pablo made a very complete library but a little oriented to his work. That's why when trying to implement the word counting problem, I have had to expand and create new functionalities.

Consequences

No several consequences have been derived from this change. Just that I have had to invest more time in understanding the library, as reflected in the previous section.

5.1.3 About budget

Changes

Because the scope and time planification changes, budged has also changed. The human resource budged needs to be recalculated tacking into account the new time planification. The other sections of the budget like hardware, software and unexpected const remain the same.

Consequences

As we can apreciete in the table below, the new human resource budget is less that 0,5% higher than the initial budget, so the changes are not significant.

New Budget

Role	Avg. Salary (€/h)	Hours	Total (€)	Total with Social Security (€)
Researcher	15	~~85~~ 75 hours	~~1275~~ 1125	~~1657.5~~ 1462.5
Haskell Developer	18	~~200~~ 210 hours	~~3600~~ 3780	~~4680~~ 4914
Project Manager	20	175 hours	3500	4550
Total	~~18.2~~ 18.3	460	~~8375~~ 8405	~~10887.5~~ 10926.5

Table 5.2: This is a table summary of the new human resources costs, self elaborated

Source	Cost (€)
Hardware	55,29
Software	0
Human Resources	~~10887.5~~ 10926.5
Unexpected	2037
Total	~~12979.79~~ **13018.79**

Table 5.3: This is a table summary of the new total budget, self elaborated

Improving the dynamic pipeline library

In this chapter, we will understand and learn how to use the Haskell DynamicPipeline library developed by Juan Pablo Royo Sales. To do this, we will carry out a small analysis and try to implement the small 'toy problem' of counting words. Finally, with all the knowledge acquired, we will try to improve the library, updating it to the new functionalities of the most modern Haskell and adding functionalities to it.

6.1 Introduction

To properly understand the library, we must first understand how the DynamicPipeline paradigm works. This work will not provide an extensive explanation of the library, as it is not within the scope of this thesis and a general understanding is sufficient. For a more detailed explanation, please refer to the work of Juan Pablo Royo Sales.

Dynamic Pipeline Paradigm

As mentioned at the beginning of this work, the dynamic pipeline paradigm is a computational model based on a chain of stages. We can define the structure of a Dynamic Pipeline from four types of stages: the Source (Sr), the Sink (Sk), the Generator (G), and the Filter (F). Defining the behavior of these four stages of the pipeline is sufficient to define the behavior of the entire pipeline.

The different stages are connected to each other by a non-zero number of channels. Initially, the pipeline consists of a Sr, which is connected by channels to the G, which is also connected by channels to the Sk. During execution, the G is responsible for generating Fs, which are placed between the Sr and the G, connecting the channels of the Sr to the first F and the last F to the G. These would be some examples of states of a Dynamic Pipeline:

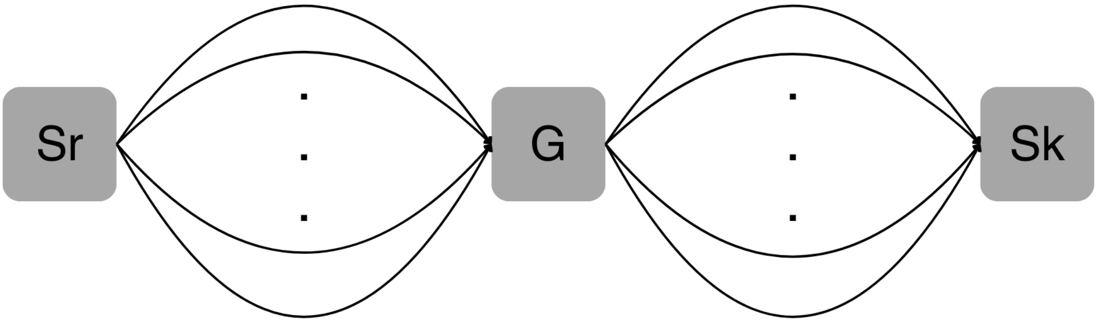


Figure 6.1: This is the initial structure of a Dynamic Pipeline, self-made with Canva

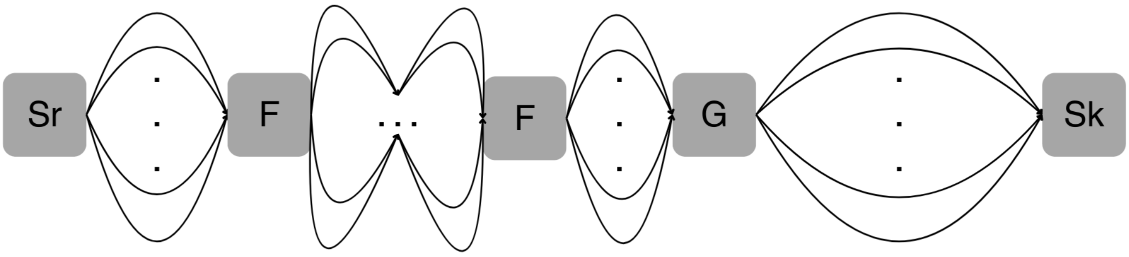


Figure 6.2: This is a posible state of a Dynamic Pipeline, self-made with Canva

Source and Sink

These two stages are the easiest to define and understand. The Source and the Sink are responsible for introducing the input stream into the Pipeline and collecting the results and output them, respectively. The Sr takes as input any form of data stream and is responsible for filling the different channels of the pipeline with data, which may already be processed and adapted by the stage itself. At the end of the pipeline, the Sk is responsible for reading the different channels and returning the output to the outside.

Generator

The Generator is more sophisticated and complex. For each element it processes, it must decide whether to let it continue to the Sk and thus pass to the output or not. In addition, it is responsible for deciding when to generate a Filter and add it to the pipeline. For each element it reads, it can decide whether or not to generate a Filter, which is interposed between the last generated F (or the Sr if it is the first generated F) and itself. It is also responsible for initializing the Filter by giving it a value for its parameter and its state.

Filter

The last stage is the Filter. It is the heart of the pipeline's computation and is responsible for processing and treating the data. Each F has a state, which can be updated, and a parameter. The execution of each filter is made up of a set of actors, which can be understood as the minimum unit of computation. Each actor can consult the filter's parameter and state and update the latter. It can also decide whether to pass on the data it processes and even generate new data to pass through the pipeline. The actors are executed sequentially, one after the other, from the first to the last.

6.2 Dynamic Pipeline Library

Once we understand the basics of the Dynamic Pipeline paradigm, we can move on to review the library developed by Juan Pablo. The first thing we can observe about the Haskell library is that it provides a set of types to work comfortably. In general, there is one type for each concept of the dynamic pipeline (Sr, G, Sk, Channel, Actor, ...):

```
data Sink
data Generator (a :: Type), a ~ Channel
data Source (a :: Type), a ~ Channel
data FeedbackChannel (a :: Type), a ~ Channel
data Channel (a :: Type), a ~ (Type :<+> ... :<+> Eof)
data DynamicPipeline dpDef filtStat filterParam st,
     dpDef ~ Source (Channel ..):=>
                         Generator(Channel ..):=>Sink
data a :=> b, a & b ~ Stage
```

Figure 6.3: These are some data types provided by the library

Based on these initial type definitions, we can see that the library provides us with the mecanism to define the DynamicPipeline type, which will determine the structure of our pipeline. We can observe that it starts with an Sr that has a set of homogeneous type channels connected to a G, which is also connected to the end of the pipeline Sk through homogeneous type channels.

The library also introduces the concept of a Feedback Channel, which is a channel that allows feedback from the Generator (G) back into the pipeline. This means that we can optionally add an additional stage with the sole task of feeding processed data back to the Source (Sr). We will delve deeper into this topic later.

6.2.1 Library combinators

The library also provides a set of combinators (functions) for creating the different stages (Sr, G, Sk, and F). The syntax of these functions can be quite complex, making them difficult to understand. Since the goal of my work is not to fully comprehend the library and the underlying Haskell grammar, I will instead focus on explaining the concept of these functions and how to use them in a more straightforward manner. This will enable anyone with a basic understanding of Haskell to implement their own Dynamic Pipeline for their specific problem.

```
withSource :: Stage (WithSource dpDef (DP st))
withGenerator :: Stage (WithGenerator dpDef filter (DP st))
withSink :: Stage (WithSink dpDef (DP st))
```

Figure 6.4: Some combinators provided by the library

Let's break down each combinator to understand how to use it to achieve the desired result.

Combinator withSource

The withSource combinator is used to create a Source stage. It takes a function as input (we will refere as fillChannels), which is responsible for filling the Source stage's output channels for any tipe of input. This function receives the output channels of the Sr stage that are conected to G (or F) as arguments and should send data to these channels.

```
source :: Stage(    WriteChannel t_1
                 -> WriteChannel t_2
                         .
                         .
                         .
                 -> WriteChannel t_n
                 -> DP st ()
                 )
source = withSource @DPStructure . fillChannels

fillChannels ::WriteChannel t_1
            -> WriteChannel t_2
                     .
                     .
                     .
            -> WriteChannel t_n
            -> DP st ()
fillChannels w_1 ... w_n = ...
```

Figure 6.5: Function to get a Source stage using withSource combinator

Alright, this should be enough to create our Source stage. We simply need to implement the fillChannels function to fill the different channels. For this purpose, the library provides helper functions that simplify filling a channel from an input source (such as an array or a file).

```
--Feed a WriteChannel froma a Monadic Seed
unfoldM :: MonadIO m
        => m a      --Monadic seed
        -> (a -> b) --Map input a to WriteChannel type b
        -> m Bool       --When stop unfolding
        -> WriteChannel b   --WriteChannel to feed
        -> m ()

--Feed a WriteChannel from a file
unfoldFile :: MonadIO m
            => FilePath           --FilePath to read from
            -> WriteChannel b     --WriteChannel to feed
            -> (ByteString -> b)  --Map ByteString to type b
            -> m ()

--Feed a WriteChannel from a foldable (like array)
unfoldT :: (MonadIO m, Foldable t)
            => t a             --Foldable to unfold
            -> WriteChannel b --WriteChannel to feed
            -> (a -> b) --Map input a to WriteChannel type b
            -> m ()
```

Figure 6.6: Functions provided by library to fead channels

Combinator withSink

Similar to withSource, the withSink combinator is used to create a Sink stage. It takes a function as input (we will refere as readChannels), which is responsible for reading the Sink stage's input channels to output it. This function receives the input channels coming from G as arguments and should output the results.

```
sink    :: Stage(    ReadChannel $t_1$
                  -> ReadChannel $t_2$
                             .
                             .
                             .
                  -> ReadChannel $t_n$
                  -> DP st ()
                  )
sink = withSink @DPStructure . readChannels

readChannels ::ReadChannel $t_1$
              -> ReadChannel $t_2$
                         .
                         .
                         .
              -> ReadChannel $t_n$
              -> DP st ()
readChannels $r_1$ ... $r_n$ = ...
```

Figure 6.7: Function to get a Sink stage using withSink combinator

With this steps, similar to withSource, should be enough to create our Sink stage. We simply need to implement the readChannels function to output the results. For this purpose, the library do not provide much help, we have just a function to read a channel and fill a generic output.

```
--Read a ReadChannel and fold it with a monadic function
foldM_ :: MonadIO m
        => ReadChannel a --ReadChannel to read
        -> (a -> m ())   --Computation to do with read element
        -> m ()
```

Figure 6.8: Functions provided by library to read channels

Combinator withGenerator

Finally, we have the withGenerator combinator, which is similar to the previous two but with some additional nuances. As we recall, the G stage has both input and output channels, and it is also responsible for generating F stages. Therefore, it must determine which of the read data should be passed to the output channels and what conditions must be met to generate an F stage.

The withGenerator combinator takes a function as input, which we will refer to as the genAction function. This function is responsible for handling all the tasks mentioned above.

```
generator :: Stage(Filter DPExample filtStat filtPar st
                -> ReadChannel $t_1$
                        .
                        .
                        .
                -> ReadChannel $t_n$
                -> WriteChannel $t_1$
                        .
                        .
                        .
                -> WriteChannel $t_n$
                -> DP st ()
                )
generator = withGenerator @DPStructure . genAction

genAction ::   Filter DPExample filtStat filtPar st
            -> ReadChannel $t_1$
                    .
                    .
                    .
            -> ReadChannel $t_n$
            -> WriteChannel $t_1$
                    .
                    .
                    .
            -> WriteChannel $t_n$
            -> DP st ()
genAction filter $r_1$ ... $r_n$ $w_1$ ... $w_n$ = ...
```

Figure 6.9: Function to get a Generator stage using withGenerator combinator

To create our G stage, we simply need to implement the genAction function. The library provides helper functions that can simplify this task:

```haskell
--Read a ReadChannel and fold it with a monadic function
unfoldF::UnFoldFilter dpDef readElem st filtStat filtPar l
        -> DP st (HList l)

mkUnfoldFilter ::
(readElem -> Bool)  --For each element determine if
                    --interpose a new Filter
-> (readElem -> DP st ()) --For each element that the Filter
                    --is consuming allow to do something
                    --outside the filter with that element.
-> Filter dpDef filtStat filterParam st   --Filter Template
-> (readElem -> filtStat) --How to Initiate Internal
                             --Filter StateT (Memory)
-> ReadChannel readElem --Main ReadChannel to feed filter
-> HList l --Rest of the ReadChannels if there are needed
           --(HNil if it only contians 1)
-> UnFoldFilter dpDef readElem st filtStat filterParam l
```

Figure 6.10: Functions unfolF and mkUnfoldFilter that help to implement genAction

By composing these two functions, we can generate our G stage. The library also provides some variations of the mkUnfoldFilter function for easier use (see functions mkUnfoldFilter', mkUnfoldFilterForAll, mkUnfoldFilterForAll').

6.2.2 Library smart constructors

The next set of functions provided by the library to complete our Dynamic Pipeline implementation are Smart Constructors. In Haskell, a smart constructor is a function that allows us to generate a result in a controlled manner and apply restrictions. Using them ensures that we meet the necessary conditions and invariants for the correct operation of the pipeline.

mkFilter Smart Constructor

The first smart constructor function provided by the library is mkFilter. This function allows us to define a F stage with a single actor (logical unit).

```
filterTemp :: Filter dpDef filtStat filterParam st
filterTemp = mkFilter actor1

actor1 :: filterParam
        -> ReadChannel t_1
                 .
                 .
                 .
        -> ReadChannel t_n
        -> WriteChannel t_1
                 .
                 .
                 .
        -> WriteChannel t_n
        -> StateT filtStat (DP st) ()
actor_1 par r_1 ... r_n w_1 ... w_n = ...
```

Figure 6.11: mkFilter smart constructor with definition of an actor

Similar to the previous functions, we only need to provide the implementation of the actor of F. Within this function, we have to determine what to do with each element it processes and decide whether to update the state of F, which elements to pass to the next filters, ... Here are some functions to perform most of the actions mentioned previously:

```
-- Push element a into WriteChannel
push :: MonadIO m
        => a
        -> WriteChannel a
        -> m ()

-- Pull element Maybe a from ReadChannel
pull :: MonadIO m
        => ReadChannel a
        -> m (Maybe a)

-- Fetch the current value of the state within the monad
get :: Monad m
    => StateT s m s

-- Sets the state within the monad to s
put :: Monad m
    => s
    -> StateT s m ()
```

Figure 6.12: Some functions to implemet the actors

It should also be noted that filters can have more than one actor. The library provides two functions for creating filters with N actors:

```
-- Add a new Actor to an already existing Filter.
(|>>>) :: Actor dpDef filtStat filterParam
                (StateT filtStat (DP st))
        -> Filter dpDef filtStat filterParam st
        -> Filter dpDef filtStat filterParam st

-- Given 2 Actors build a Filter.
(|>>) :: Actor dpDef filtStat filterParam
                (StateT filtStat (DP st))
    -> Actor dpDef filtStat filterParam
                (StateT filtStat (DP st))
    -> Filter dpDef filtStat filterParam st
```

Figure 6.13: Function to create a filter with N actors

mkGenerator Smart Constructor

This second smart constructor allows us to link our definition of G with our definition of F. Simply passing our 2 previously defined functions as arguments to the function, we obtain a stage that encompasses our 2 definitions and combines them:

```
mkGenerator :: Stage (WithGenerator dpDef
             (Filter dpDef filtStat filterParam st) (DP st))
            -> Filter dpDef filtStat filterParam st
            -> GeneratorStage dpDef filtStat filterParam st

generatorStage::GeneratorStage dpDef filtStat filterParam st
generatorStage = mkGenerator generator filterTemp
```

Figure 6.14: mkGenerator combinator

mkDP Smart Constructor

This is the final combiner. It takes the three previously defined definitions (source, sink, generatorStage) and combines them to obtain our dynamic pipeline.

```
mkDP :: Stage (WithSource dpDef (DP st))
    -> GeneratorStage dpDef filtStat filterParam st
    -> Stage (WithSink dpDef (DP st))
    -> DP st ()

DP' :: DP st ()
DP' = mkDP source generatorStage sink
```

Figure 6.15: mkDP combinator

6.2.3 Conclusions

With all of this, we have all the elements to execute our Dynamic Pipeline. The library give us a function that, given a pipeline, run it to final output result.

```
runDP :: DP st ()
         -> IO a

main :: IO ()
main = runDP DP'
```

Figure 6.16: runDP function for runing a DP

In this section, we have reviewed the entire library to extract the key concepts and tools necessary to implement the vast majority of Dynamic Pipelines. We have observed how, using the functions provided by the library, it is only necessary to implement a few basic functions corresponding to the essential functionalities of the pipeline itself. In this way, anyone with a Dynamic Pipeline algorithm can implement it with just some basic Haskell notions.

Here is a summary of the functions to implement for each pipeline:

```
-- Function defining what we put in each channel
fillChannels ::WriteChannel t_1
            -> WriteChannel t_2
                    .
                    .
                    .
            -> WriteChannel t_n
            -> DP st ()

-- Function defining what we do with the final data
readChannels ::ReadChannel t_1
            -> ReadChannel t_2
                    .
                    .
                    .
            -> ReadChannel t_n
            -> DP st ()


genAction ::    Filter DPExample filtStat filtPar st
            -> ReadChannel t_1
                    .
                    .
                    .
            -> ReadChannel t_n
            -> WriteChannel t_1
                    .
                    .
                    .
            -> WriteChannel t_n
            -> DP st ()

-- Function defining an actor of a filter
actorN :: filterParam
        -> ReadChannel t_1
                .
                .
                .
        -> ReadChannel t_n
        -> WriteChannel t_1
                .
                .
                .
        -> WriteChannel t_n
        -> StateT filtStat (DP st) ()
```

Figure 6.17: Functions to implement for a DP

6.3 Implementing the toy problem

At this point, we have understood the two fundamental aspects: how a dynamic pipeline works and how to define and implement one. In this section, we will walk through the entire process of defining and implementing a dynamic pipeline using a toy problem. The goal is to present an example and obtain a template for future implementations of different problems.

6.3.1 Algorithm Definition

The task at hand is to implement a word counting algorithm using a dynamic pipeline. The goal is to incrementally process a stream of characters, counting the occurrences of each word, and printing partial results whenever a period ('.') character is encountered. This approach leverages the incremental nature of dynamic pipelines to provide real-time updates.

I will not delve deeply into the process of designing the algorithm itself, as the idea is to focus on the implementation. Therefore, I will explain my proposed solution for the problem and carry out a small test run to understand the behavior. To keep my design simple, it will only have one channel, and the filters will only have one actor. This will minimize the complexity of understanding. Since we only have one channel, the data type that will flow will be tuples (Word, Int). This will represent a specific word along with its count of how many times it appears. It should be noted that since there is only one channel, the input data and the data generated by the filters themselves (solutions) will go on the same channel, so a rule will have to be created to differentiate them.

Source

Alright, in our case, the Sr will need to obtain the words from the stream and feed the channel with tuples (Word, Int). Since these are the input elements, we will initialize the tuple with (Word, 0). This way, we can differentiate the input data from the solutions (since we know that the solutions will have at least a count of 1).

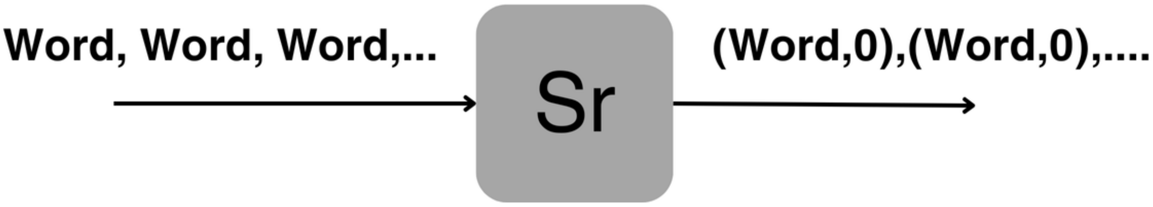


Figure 6.18: Grafic representation of the Source stage

Sink

This stage is the simplest, as it only has to collect the results and display them. The Sk will receive tuples (Word, Int) and should display them (to the console or a file, for example) in the desired format.

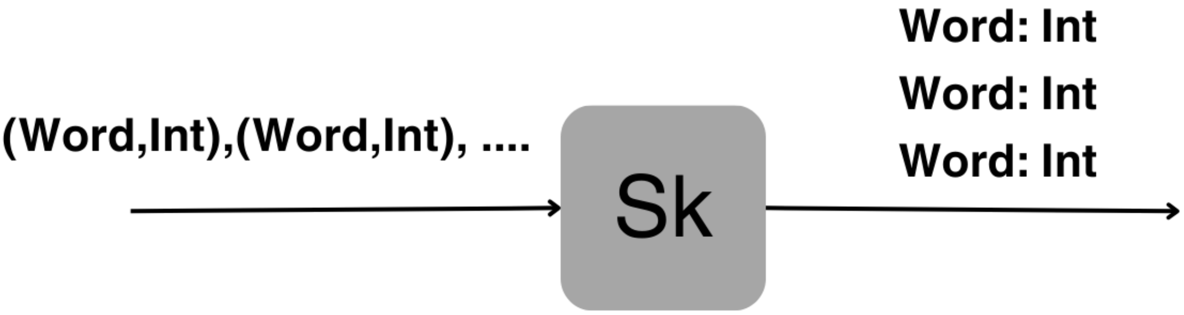


Figure 6.19: Grafic representation of the Sink stage

Generator

Now let's move on to the heart of the pipeline, the Generator. This takes care of several tasks, so let's go through them one by one:

- **Create Filters:** For each element it reads, it must decide whether or not to create a F. For our problem, the most sensible thing would be to have one filter for each unique word, so the G must ensure that it does not create two filters for the same word. To do this, we can take advantage of the fact that filters can also decide not to let words pass. We can leave the responsibility of not letting repeated words pass to the filters and have the G create an F for each word that reaches it, since this will be, in any case, the first appearance.

- **Initialize Filters:** The G must be responsible for initializing the state and parameter of the F. As mentioned previously, each F will handle a different word, so the most logical thing is to initialize the parameter with the word it handles (since it does not change). On the other hand, the state will have the appearance counter, so it can be updated with each new appearance.

- **Pass the elements:** It must decide which elements to pass on to the Sk to be displayed. With the rules we have created, we know that every tuple (Word, 0) is an input element that has not yet been processed, so we must discard it. On the other hand, if the counter is at least 1, we know that it comes from a filter, so it will be a result. Therefore, we will only pass non-initial elements, that is, those with a counter different from 0.

With all this defined, we have our G created and we can move on with the last stage.

Filter

To finish, let's define the F. At this point, we know that its parameter contains the word it handles and that its state contains the word count, so the logic that remains to be defined is quite simple. The filter may receive words equal to its own, in which case it will have to update its counter by adding 1 and discarding the word, or different words, in which case it will simply let them pass. With this logic, we can keep the counter updated at all times and meet the requirements on which the other stages are based (not letting repeated words pass).

The only thing left would be to introduce the incremental component mentioned at the beginning of the section. '.' could arrive, which represent a signal to indicate to the F's that they should pass a partial response. Therefore, before starting the logic, they will have to check if the input is a '.' , so they will have to take the state and join it with the parameter in a tuple with the form (parameter, tuple) followed by the point so that the rest of the filters do the same. In case it is not a '.', the same logic mentioned above will be done.

6.3.2 Algorithm Simulation

With all the stages defined, we can now simulate the algorithm for better understanding.

We start the simulation, this is our initial state of the pipeline. As input we have a string of words 'Dog', 'Cat', 'Dog' ended by a '.'.

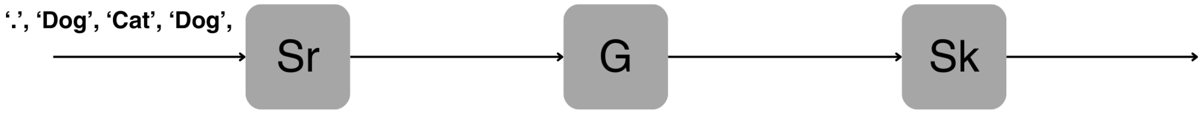


Figure 6.20: Initial state of the Dynamic Pipeline with the input prepared

The first word, 'Dog', will go through the Sr and turn it into a tuple (Dog, 0). This reaches the generator and generates the first filter, since it is an initial data (counter to 0). The Sr continues to process the input while following the process.

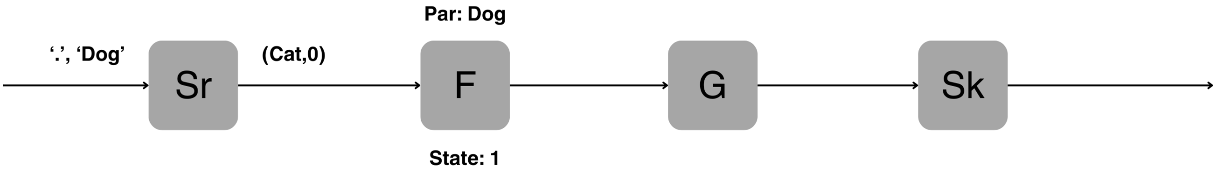


Figure 6.21: State of Dynamic Pipeline after consuming some words

If we continue executing until just after the Sr processes the '.', this is the state of our pipeline:

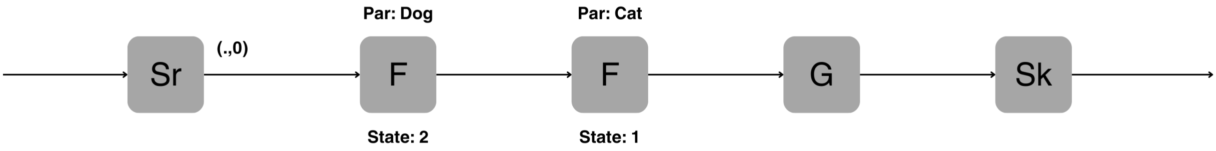


Figure 6.22: State of Dynamic Pipeline before the '.' is processed by filters

When the '.' starts to be processed by F's, they with start to throw partial results. This is the view of the pipeline before the second F starts to process the '.'.

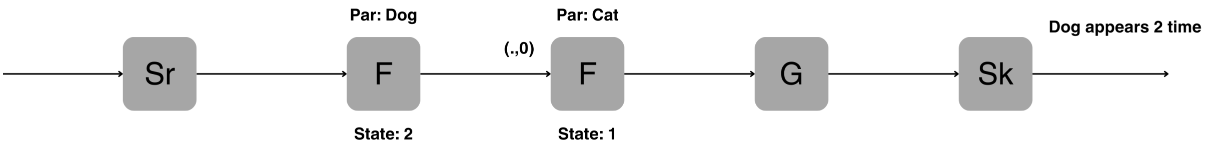


Figure 6.23: State of Dynamic Pipeline before the the second filter processes the '.'

And this is the final state after the whole execution:

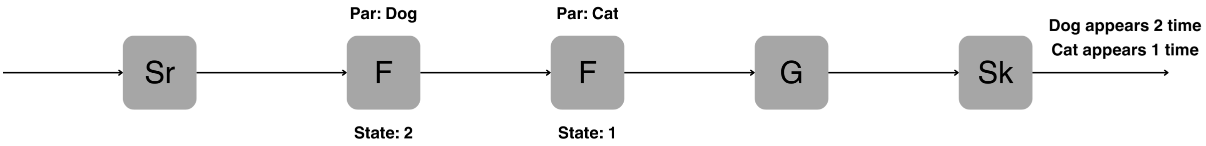


Figure 6.24: Final state of the Dynamic Pipeline

6.3.3 Algorithm Implementation

Now we have all the components necessary to use the library in a real case: We understand how a Dynamic Pipeline works, we know how to implement it using the Haskell library and we have defined an algorithm for a specific problem. Now we just have to apply everything.

Type definition

Let's start by defining our Dynamic Pipeline structure in Haskell:

```
type Word = String
type Pair = (Word,Int)
type filtStat = Int
type filtPar = Pair
type DPExample = Source (Channel (Pair :<+> Eof))
                :=> Generator (Channel (Pair :<+> Eof))
                :=> Sink
```

Figure 6.25: Type definition of the Dynamic Pipeline

I've also defined some additional types for the tuples and for the state and parameters of the F filter, for simplicity.

Source

As we determined earlier, to define the source we only have to implement 1 function (which we have called fillChannels)

```haskell
input :: [Word]
input = ["Dog","Cat","Dog","."]

source' :: Stage (WriteChannel Pair -> DP s ())
source' = withSource @DPExample fillChannels

fillChannels :: WriteChannel Pair
             -> DP st ()
fillChannels wp = unfoldT input wp $ \w -> (w,0)
```

Figure 6.26: Implementation of fillChannels function

We can observe how the fillChannels function uses the unfoldT function provided by the library to read an array and pass it through a write channel. We also use an anonymous function to transform the words into tuples.

Sink

For this stage, we need to implement the readChannels function:

```haskell
sink' :: Stage (ReadChannel Pair -> DP s ())
sink' = withSink @DPExample readChannels

readChannels ::ReadChannel Pair -> DP st ()
readChannels rp = foldM_ rp printResults

printResults :: Pair -> DP st ()
printResults (".",_) = print "**********************"
printResults (w,n) = print $ w ++ "␣" ++ show n
```

Figure 6.27: Implementation of readChannels function

In this case, we use the unique function provided by the library: foldM function. It reads a channel and do a function, in this case, print the results.

Filter

For the F we need to implement all the actors. As we have only one actor, we will only implement one function.

```
filterTemp :: Filter DPExample filtStat filtPar s
filterTemp = mkFilter actor1

actor1 :: filtPar
        -> ReadChannel Pair
        -> WriteChannel Pair
        -> StateT filtStat (DP s) ()
actor1 (par,_) rp wp =
  foldM_ rp $ \(inp,y) ->
    if inp == "."
    then get >>= \x -> push (par,x) wp >> push (".",0) wp
    else    if inp == par
            then modify (+1)
            else push (inp,y) wp
```

Figure 6.28: Implementation of actor

As we defined before, it treats each element with the foldM function following the chain of if's: first if it's a period, where it would pass the result and the period. Otherwise, it would update the state with the modify function or simply let the word pass through.

Generator

Finally, we have to implement the G stage:

```
generator' :: GeneratorStage DPExample filtStat filtPar s
generator' = let gen = withGenerator @DPExample genAction
             in  mkGenerator gen filterTemp

genAction :: Filter DPExample filtStat filtPar s
          -> ReadChannel Pair
          -> WriteChannel Pair
          -> DP s ()
genAction filter rp wp =
  let unfoldFilter =
  mkUnfoldFilter create (moveOn wp) filter iniFilter rp HNil
  in void $ unfoldF unfoldFilter

create :: Pair-> Bool
create (".",_) = False
create (_,0) = True
create _ = False

moveOn :: WriteChannel Pair -> Pair -> DP s ()
moveOn c (".",_) = push (".",0) c
moveOn _ (_,0) = return ()
moveOn c a = push a c

iniFilter :: Pair -> filtStat
iniFilter _ = 1
```

Figure 6.29: Implementation of generator

This stage involves defining the most functions, although the mkUnfoldFilter constructor aids significantly in implementation. The three generator functionalities must be implemented: determining under which conditions to generate a filter, initializing a filter, and deciding which elements to pass through. To achieve this, three functions have been defined: create, iniFilter, and moveOn, which handle the aforementioned functionalities, respectively. The *create* function takes an element of the input channel type and returns a boolean value indicating whether or not to generate a filter.
Similarly, the *iniFilter* function takes an element of the input channel type and returns an element of the filter state type.
Finally, the *moveOn* function receives both the write channel and an element of the channel type and decides whether to push the element or do nothing (effectively discarding the element).

6.3.4 Conclusions

In this section, we have seen the entire process for implementing a simple problem like counting words. To test its effectiveness, some tests have been carried out. This is a tested input:

input = [”dog”, ”cat”,”.”,”dog”,”dog”,”dog”,”.”,”bird”,”cat”,”.”]

And this has been the result obtained:

```
”dog 1”
”cat 1”
”**********************”
”dog 4”
”cat 1”
”**********************”
”dog 4”
”cat 2”
”bird 1”
”**********************”
```

Therefore, we can conclude that the implementation has been developed successfully

6.4 Improving the library

After working with the library, I have been able to identify different shortcomings and lack of functionalities for a more diverse and easier use of the library. In this section, I will detail and comment on different improvements that I have added to the library.

6.4.1 New features

unfoldFilebyChars

When I was working with the library, the first thing I missed was having more tools to work with the input. The library has the functions unfoldM, unfoldFile, and unfoldT to handle general monadic inputs, files, and lists, respectively. The problem is that the unfoldFile function only reads files line by line, since that's how it was useful for Juan Pablo's problem. That's why I decided to create this function to be able to read character by character for letters of the alphabet. In this way, the channel is fed with characters and it can be, for example, the F or the G that are responsible for finding the desired format.

```haskell
unfoldFilebyChars :: FilePath
                    -> WriteChannel a
                    -> (ByteString -> a)
                    -> DP s ()
unfoldFilebyChars file writeChannel fn =
  liftIO $ R.withFile file ReadMode $ \h ->
        unfoldM (hGetchar h) fn (H.hIsEOF h) writeChannel

hGetchar :: Handle -> IO ByteString
hGetchar h = do
    c <- hGet h 1
    if c == "" || c == "." || "a" <= c && c <= "z"
    then return c
    else hGetchar h
```

Figure 6.30: unfoldFilebyChars function

An example usage of the function is as follows:

```haskell
-- Write Channel (wc) Type = String
unfoldFilebyChars "example.txt" wc decodeUtf8
```

Figure 6.31: unfoldFilebyChars function usage

unfoldFilebyWords

Similar to the previous function, this function feeds a channel from a file by reading the words. It separates the input by the blank space character ' ' similar to the getContents function in Haskell.

This improvement is quite significant, offering considerably more flexibility than the existing function. The ability to retrieve content item by item rather than the entire row facilitates simpler input processing.

```
unfoldFilebyWords :: FilePath
            -> WriteChannel a --Write Channel to feed
            -> (ByteString -> a) --Map ByteString to type a
            -> DP s ()
unfoldFilebyWords file writeChannel fn =
  liftIO $ R.withFile file ReadMode $ \h ->
        unfoldM (hGetWord h) fn (H.hIsEOF h) writeChannel

hGetWord :: Handle
        -> IO ByteString
hGetWord h = hGetWordRec h B.empty

hGetWordRec :: Handle
            -> ByteString
            -> IO ByteString
hGetWordRec h r = do
    c <- B.hGet h 1
    if c == "" || c == "␣" || c == "\n" then return r
    else do
        cs <- hGetWordRec h (B.append r c)
        return cs
```

Figure 6.32: unfoldFilebyWords function

An example usage of the function is as follows:

```
-- Write Channel (wc) Type = String
unfoldFilebyWords "example.txt" wc decodeUtf8
```

Figure 6.33: unfoldFilebyWords function usage

pushState

While not essential for the library's core functionality, this function aims to streamline the implementation process. Often, retrieving and passing the filter state through a channel is required. This function simplifies this task, enhancing ease of use. Additionally, an optional function 'f' can be provided to convert the filter state type to the channel type.

```
pushState :: WriteChannel a --Write Channel to feed
            -> (b -> a) --Map state type to channel type
            -> StateT b (DP s) ()
pushState wp f = get >>= flip push wp . f
```

Figure 6.34: pushState function

An example usage of the function is as follows:

```
-- filtStat Type = Int
-- Write Channel (wc) Type = String
pushState wc show
```

Figure 6.35: pushState function usage

foldFile

Another major problem I encountered was that the library also had almost no functions for handling output in a simple way. The library has the foldM function, which is a very generic function for consuming a channel. That's why I decided to add this function, which consumes a channel and writes it to an output file. At the beginning of writing, the file is created if it does not exist, or the file is cleaned if it already existed.

This new functionality is highly desirable, especially when dealing with large inputs or performing analyses. It allows us to gather all results in a convenient format, enabling more efficient processing and reading.

```
foldFile:: MonadIO m
        => FilePath      --Path of file
        -> (a -> Text)   -- Map WriteChannel type a to Text
        -> ReadChannel a --ReadChannel to read
        -> m ()
foldFile file f rc = do
  writeFileText file T.empty
  foldM_ rc (R.appendFileText file . f)
```

Figure 6.36: foldFile function

An example usage of the function is as follows:

```
-- Read Channel (rc) Type = String
foldFile "example.txt" toText rc
```

Figure 6.37: foldFile function usage

6.4.2 Updating the library to a newer version of GHC

In this section, we will update the library to be compatible with newer versions of GHC, the Haskell compiler. Currently, the library supports GHC version 8.10.3 and will be updated to support version 9.0.2. This specific version was chosen as it's the latest supported by all libraries utilized by Dynamic Pipeline. Particularly, the HList library introduces compatibility issues with newer GHC versions. Updating the libraries would be necessary to leverage newer compiler versions.

The library consists of four source files: Flow.hs, Stage.hs, Channel.hs, and DynamicPipeline.hs. DynamicPipeline is the main file and encompasses the other three, hence the need to update all of them.

This is the first error encountered when trying to compile DynamicPipeline.hs with the new GHC version:

```
DynamicPipeline/Stage.hs:515:9: error:
    Couldnt match type: DP st0 a
        with: forall (st :: k). DP st a
    Expected: (forall (st :: k). DP st a) -> IO a
        Actual: DP st0 a -> IO a
    In the expression: runStage
    In an equation for 'runDP': runDP = runStage
    Relevant bindings include
        runDP :: (forall (st :: k). DP st a) -> IO a
            (bound at DynamicPipeline/Stage.hs:515:1)
    |
515 | runDP = runStage
    |
```

Figure 6.38: Console output after compiling DynamicPipeline.hs with GHC 9.0.2

And here is the code for the function that generates the error:

```
runDP :: (forall st. DP st a) -> IO a
runDP = runStage
```

Figure 6.39: Piece of code generating the error

What we can gather from this error is that there's a type inference issue with the runDP function in the Stage.hs file. Based on the observation, it seems that newer GHC versions are stricter with polymorphic types. Delving into the GHC documentation, the first relevant information we find is that the initial GHC 9 release (9.0.1) introduced the following feature: **empty citation**

"Record field selectors are now given type signatures that preserve the user-written order of quantified type variables. Moreover, field selector type signatures no longer make inferred type variables available for explicit type application."

If we apply the new restrictions detailed in the release notes, our code would now look like this:

```
runDP :: forall {k} (st :: k) a. DP st a -> IO a
runDP = runStage
```

Figure 6.40: New version of the code

As we can see now, we need to explicitly state that the type of st is any. With this change, the compilation error no longer appears, so this part is now resolved. However, when compiling again, there are still some errors to address. There is a large output in the console, and here is some of the relevant information:

```
            .
            .
            .
DynamicPipeline/Stage.hs:408:29: error:
Couldnt match type 'b0'
    with MonadState filtStat monadicAction =>
    Stage (WithFilter dpDef filterParam monadicAction)
Expected:
    Actor dpDef filtStat filterParam monadicAction
    -> b0
Actual:
    Actor dpDef filtStat filterParam monadicAction
    -> MonadState filtStat monadicAction =>
    Stage (WithFilter dpDef filterParam monadicAction)
            .
            .
            .
    |
408  | runActor = hUncurry . run . unActor
    |
```

Figure 6.41: Second console output after compiling DynamicPipeline.hs with GHC 9.0.2

Similar to the previous error, here we also have a type inference problem. Once again, the GHC update has made it stricter, causing us to have to modify the code to fix it.

If we see, the funcion unActor is the problem, so here is the code of the definition:

```
newtype Actor dpDef filtStat
                filterParam monadicAction =
  Actor {unActor ::
    MonadState filtStat monadicAction =>
  Stage (WithFilter dpDefinition filterParam monadicAction)}
```

Figure 6.42: Piece of code generating the error

Starting with GHC version 9.0.1, it was added that new type definitions had to be explicit, so the way we have it here can no longer be used. Therefore, we must leave the code like this:

```
newtype Actor dpDef filtStat
                filterParam monadicAction =
Actor {unActor ::
  Stage (WithFilter dpDef filterParam monadicAction)}
```

Figure 6.43: New version of the code

With these changes, we have now been able to compile the library with GHC 9.0.2.

6.5 Chapter summary

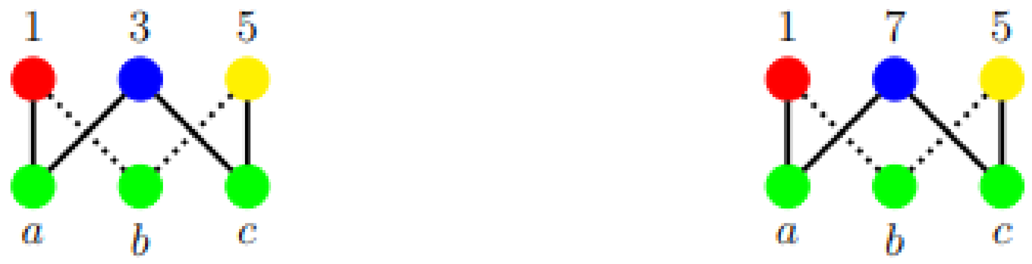
In this section, we have been able to improve the Haskell Dynamic Pipeline library. We have added new features to make it easier to implement new algorithms, and we have also updated the library to support GHC 9.0.2. With these advances, I aim to ensure that the library remains usable and does not become outdated. At the end of the work 8, an experimental test will be carried out to verify if an improvement in the performance of the library has also been obtained.

Improving the IEBT algorithm

This is the second part of my work, where I will try to find and argue for different improvements for the implementation of the algorithm for incremental enumeration of Bitriagles (IEBT). This implementation developed by Juan Pablo uses the Dynamic Pipeline library with which I have been working. The entire first part of my work has been necessary to fully understand the operation of both the dynamic pipeline paradigm itself and the operation of the Haskell library.

7.1 Introduction

A brief introduction to understand the basic concepts is provided here. For a more detailed explanation, refer to Royo-Sales et al. [1, See chapter 6]
The IEBT algorithm involves finding bitriangles in a bipartite graph.

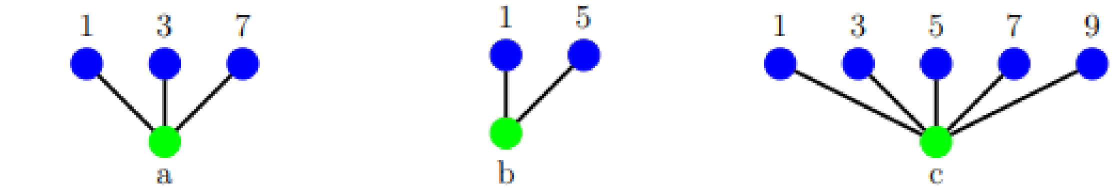


(A) BT $(1, a, 3, c, 5, b, 1)$

(B) BT $(1, a, 7, c, 5, b, 1)$

Figure 7.1: Bitriangle structure, taken from [1]

To achieve this, it groups vertices into different structures until it reaches a structure that forms many bitriangles.
The process is as follows: initially, all the edges of the graph are processed, and the generator creates a filter for each edge it receives, initializing it with that edge. The filters process the edges and check the vertices to find a match with their parameter. If a match is found, it consumes the edge and adds it to its state, thus forming the Aggregated Wedges.



(A) This is the AW $\langle a, \{1, 3, 7\}\rangle$, the compact representation of wedges

(B) This is the AW $\langle b, \{1, 5\}\rangle$, the compact representation of wedges

(C) This is the AW $\langle c, \{1, 3, 5, 7, 9\}\rangle$, the compact representation of wedges

Figure 7.2: Aggregated Wedges strucute, taken from [1]

Once all the edges have been processed, the filters send the AWs generated by the pipeline so that they can be combined to form the Aggregated Double-Wedges.

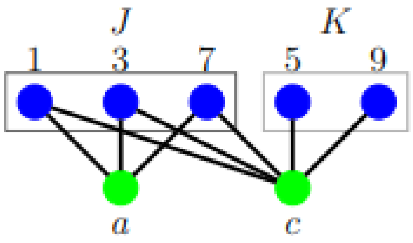


Figure 7.3: Aggregated Doble Wedges structure, taken from [1]

Once the filters have these structures, queries can be sent through the pipeline to obtain the bitriangles.

7.2 Improving structures

The first topic I wanted to address was the data types used to store the different sets. The algorithm needs to store different vertex structures for later matching. These are the different definitions found in the code:

```haskell
type LowerVertex = Int
type UpperVertex = Int
type Edge = (UpperVertex, LowerVertex)

data Command = ByVertex IntSet
            | ByEdge (Set Edge)
            | Count
            | AllBT
            | NoCommand
            | End
        deriving (Show, Read)

data W = W
  { _wLowerVertex :: LowerVertex
  , _wWedges      :: IntSet
  }
  deriving Show

data Triplet = Triplet Int Int Int
data Pair = Pair Int Int

type UT = (IntSet, IntSet, IntSet)

data DW = DW
  { _dwLower :: Pair
  , _dwUpper :: UT
  }

newtype DWTT = DWTT [DW]
  deriving newtype (Semigroup, Monoid)

data BT = BT
  { _btLower :: Triplet
  , _btUpper :: UT
  }

newtype BTTT = BTTT [BT]
  deriving newtype (Semigroup, Monoid)

data BTResult = RBT Q (Int, Int, Int, Int, Int, Int, Int)
              | RC  Q Int

data FilterState = Adj W
                 | DoubleWedges DWTT
                 | BiTriangles BTTT
```

Figure 7.4: Type and data definitions of IEBT implementation

We can see how it represents vertices with integers Int and edges as tuples of integers (Int, Int). The most important thing to analyze are the set representations, as their impact on the performance of the algorithm is crucial. It uses the IntSet structure to represent sets of vertices and also a tuple of 3 IntSets (IntSet, IntSet, IntSet) to represent the set of upper vertex of the Aggregated Doble Wedge and Aggregated Bitriangles. [1, Page52] Also taking into account the use of lists to represent the set of Aggregated Double Wedges and Aggregated Bitriangles.

We will be analizing structure by structure trying to find if the structure used is the most efficient.

7.2.1 Set of vertices representation

The IntSet structure belongs to the Haskell IntSet library [8], which represents an ordered set of Ints. This structure is very efficient because it takes advantage of the range limit of Ints to limit many operations to a cost of O(min(n,W)), where n is the number of elements and W is the number of bits in the Int representation (32 or 64). Additionally, this structure is based on big-endian patricia trees, which have very good performance for set intersection and union operations.

Wedges representation

If we review the first structure that use IntSet, we can see that filters can have three types of states, one of which is the Wedge, which uses this structure. To understand how this structure is used (and therefore whether or not it can be improved), we need to know what computations are performed with it. Observing the code, we can see that only two of the four actors that each filter has deal with W: actor1 and actor2.
On the part of actor1, for each input edge, a condition is checked and if it is affirmative, a vertex is attached to the IntSet set. Therefore, in the worst case, insertions will have to be made and each insertion has a cost of O(n,W), and assuming a sufficiently large n, we are left with O(W) = O(1).
On the other hand, actor2 performs difference and intersection operations with the IntSets. These operations with this library have a linear cost with respect to the size of the sets O(n + m).

Looking at these costs, we can conclude that a very efficient structure seems to be used. We could also use a Hash-type structure, such as a HashSet, to improve the performance of insertions. Reviewing the Haskell libraries, we find the HashSet structure. Despite being useful, the documentation itself warns us that for Int sets, the IntSet library is more efficient. Therefore, for Wedges, IntSets are the best structure.

Doble Wedges and Agregated Bitriangles representation

Next, we should focus on the definition of the UT type, which is formed by a tuple of 3 IntSets. If we review the entire code for functions that use it, we find a similar scenario to the previous one. In general, insertion operations are performed on the sets, and the choice of IntSet is correct.

On the other hand, the other important operation that is performed is to check if a vertex belongs to any of the 3 sets. To do this, it checks each of the 3 sets, so it might be possible to optimize this process a bit. Despite this, the best option that could be done is to try to join the 3 sets so that you only have to search in a larger one, asymptotically reducing the cost by 3. The problem is that we would have to find another way to represent the 3 sets, such as with indexes using a dictionary. For this reason, I have considered that the cost of carrying this structure and other operations that are done on it increases greatly for the improvement we obtain, so I do not see any possible improvement in this regard.

Therefore, the choice of this structure is the correct one to represent the UTs.

7.2.2 Set of edges representation

The next thing to observe is the representation of edge sets. As such, the entire algorithm does not store any edge sets, as it plays with the structure of the different types to infer the edges. There is only one thing for which edges are used: the final Queries.

At this stage, a set of edges can be passed to find bitriangles that contain it, and for this, it uses the Set to store them. In this case, the only operation that is done is to check whether an edge is inside or not. Therefore, I have determined that a possible improvement could be to change the Set, which has a search cost of O(log n), to one that has better performance, such as the HashSet. The HashSet has a cost, in the general case, of constant O(1).

Therefore, I have changed the Set structure to HashSet to look for a performance improvement. At the end of the work 8, an experimental test will be carried out to verify whether there is an improvement or not.

7.3 Chapter Summary

In this chapter, we have been analyzing the implementation of the IEBT algorithm to try to find an improvement.
Despite the fact that the beginning of this work arose with the belief that it could be improved, I have not been able to find a possible significant improvement for the implementation of the IEBT algorithm.

Experiments

In this chapter, we will carry out some experiments to test the improvements made in chapters 6 and 7. For the execution of the programs, I will be using an Asus VivoBook 15 X540UB laptop. These are the technical specifications of the laptop:

- **Processor:** Intel Core i5-8250U CPU @ 1.60GHz
- **Memory** 8GB RAM
- **Graphics adapter:** Intel UHD Graphics 620
- **Storage:** 256 SSD
- **Operating System:** Windows 10 Home 64-bit

To ensure better results, they will all be executed with the laptop running only that process. In all experiments, only the execution time will be evaluated. Memory usage would remain to be checked to complete the evaluation.
All the experiments and results can be found in the 'Experiments' folder on Github. [4]

8.1 Testing Library

The main improvement that can affect the performance of the library is the update of GHC. To test this, we will use the toy problem developed during this work. We will modify it a bit by counting characters instead of words, in order to be able to test larger inputs more efficiently. We will limit the characters to those of the alphabet (plus the dot for the incremental character) thus ensuring that there will be at most 26 filters.

Input

The text of Don Quixote has been used as input, which has been modified to convert or eliminate any unwanted characters. To do this function, a python script has been used that also allows generating inputs of a specific size. Inputs of powers of 10 will be used, from 10 to 100000.

Results

These are the results of running 10 times for each size and taking the mean and standard deviation.

size	mean (s)	std (s)
10	0.029994	0.012285
100	0.100318	0.008902
1000	0.971935	0.097618
10000	11.619426	0.835211
100000	231.629199	1.330213

Table 8.1: Results 10 executions using GHC 8.10.3

size	mean (s)	std (s)
10	0.005374	0.001071
100	0.091677	0.004923
1000	0.881183	0.067268
10000	9.786606	0.606498
100000	159.005325	1.553883

Table 8.2: Results 10 executions using GHC 9.0.2

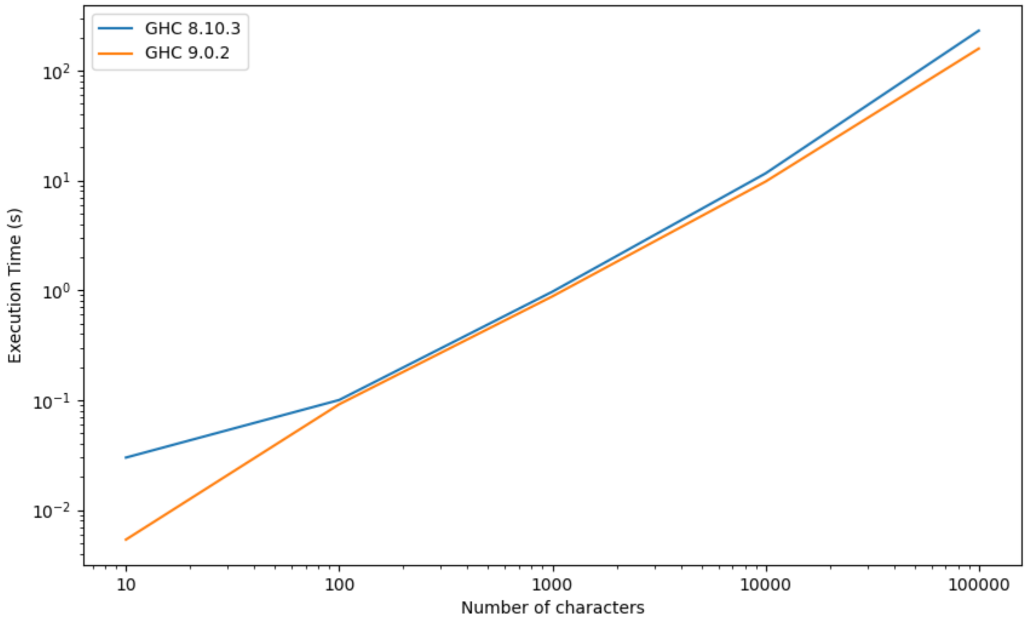


Figure 8.1: Plot of execution mean time vs input size for GHC 8.10.3 and GHC 9.0.2

Discussion

As we can observe, there seems to be a slight improvement with GHC 9.02 compared to GHC 8.10.3, even in some more noticeable cases.

We can see how for much smaller inputs the new version tends to obtain better results, which could be due to better management when creating the filters. In smaller inputs, the task that is done the most is generating filters, because with larger inputs there comes a point where the 26 filters have already been generated (one for each letter) and no more are generated. Perhaps better memory management by the compiler is causing this improvement.
On the other hand, we can observe how in the central part the two have a similar performance, although the new version is still slightly better.
Finally, we can see how with larger inputs the new version again has better performance, this time more noticeable.

In summary, although it seems that with the update to GHC 9.0.2 there is an improvement, we do not obtain very clear results. Other factors such as memory would have to be checked to conclude with a result.

8.2 Testing IEBT

In this experiment, we will test the change of structure from Set to HashSet implemented in the IEBT algorithm.

Input

For this experiment, we will be using a bipartite graph of 3000 edges and we will make queries from size 1000 edges to 3000.

Results

This is the result of running each size 5 times and obtaining the mean and standard deviation.

size	mean (s)	std (s)
1000	5.570314	0.560374
1500	7.731554	1.032222
2000	8.778615	1.534666
2500	9.506494	0.579319
3000	12.134800	2.795664

Table 8.3: Results 5 executions using Set structure

size	mean (s)	std (s)
1000	7.677365	0.378990
1500	8.242099	1.010196
2000	8.417874	0.367513
2500	9.013060	0.483033
3000	9.586805	1.195353

Table 8.4: Results 5 executions using HashSet structure

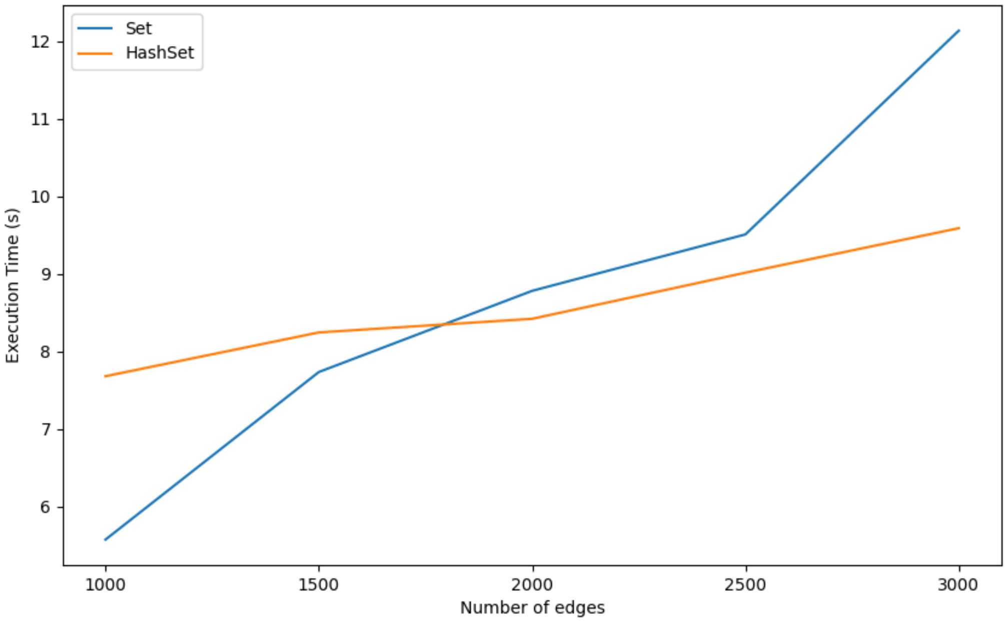


Figure 8.2: Plot of execution mean time vs input size for Set and HashSet

Discussion

We can see that it seems that we have obtained a performance improvement when the input starts to grow.
For smaller inputs, Set obtains better performance, surely because the overhead of making HashSet calculations is high and Set has better performance. On the other hand, when the input grows, we see how HashSet remains more stable and grows in a more linear way, while Set grows faster due to the overhead of doing the searches.

In short, we can conclude that this change has been positive, although it is only for large input types of edges, not for the general case of the algorithm.

Conclusions and future work

In this chapter, we will discuss the conclusions and results of the work. We will also comment on possible future work that could continue this work.

9.1 Conclusions

This project has gone through many phases and has generally achieved positive results.

Firstly, regarding the library, we have been able to fully understand its operation and have created a guide to follow in order to use it more easily. We have followed the process of defining and implementing an algorithm that uses a dynamic pipeline: the word counting problem. We have added new functionalities and updated it to be usable with a newer version of GHC. We have also been able to carry out a small experiment to check if there was a performance improvement with the code update, obtaining a positive result.

On the other hand, there have been attempts to improve the implementation of the IEBT algorithm, with less positive results. Based on the initial assumption that the implementation of the used structures could be improved, all structures have been analyzed for optimization opportunities. We have confirmed that the structures used are generally correct and that no significant improvements are expected in this area. Despite this, we have managed to identify one structure that could be improved and have conducted experiments. The results indicate some improvement, although further testing with larger inputs would be necessary to draw more conclusive conclusions.

Overall, we can draw a positive assessment of this work. The three initially proposed objectives and the overall planning were successfully accomplished. Ultimately, understanding the library and Haskell code required more time than anticipated, which, combined with the limited time frame of this project, prevented further improvements to the library and algorithm.

9.2 Future work

Once the entire project is completed, there are a few concepts that could be further explored for future expansion:

- **Continue updating the library:** It would be necessary to continue updating the library to make it compatible with new GHC versions as they are released. This would involve reviewing some dependencies and considering newer versions to ensure ongoing compatibility. Additionally, any new functionalities added to the library would enhance its usability.

- **Investigate the IEBT algorithm implementation further:** It would be worthwhile to delve deeper into the possibility of improving the data structures used to store the various sets. This would involve exploring more sophisticated matching techniques or alternative methods of representing these structures more efficiently.
- **Conduct more experiments:** Additional resources would be required to conduct experiments with larger and more diverse inputs. It would also be necessary to assess performance using other metrics, such as memory usage, to obtain more consistent results.

Bibliography

[1] J. P. Royo Sales, "An algorithm for incrementally enumerating bitriangles in large bipartite networks," Ph.D. dissertation, UPC, Facultat d'Informàtica de Barcelona, Departament de Ciències de la Computació, Oct. 2021. [Online]. Available: http://hdl.handle.net/2117/361615.

[2] E. Pasarella, M.-E. Vidal, C. Zoltan, and J. P. R. Sales, "A computational framework based on the dynamic pipeline approach," *Journal of Logical and Algebraic Methods in Programming*, vol. 139, p. 100 966, 2024.

[3] A. Herrero Bravo, "An experimental guided approach to the metric dimension on different graph families," Ph.D. dissertation, UPC, Facultat d'Informàtica de Barcelona, Departament de Ciències de la Computació, Jan. 2024. [Online]. Available: http://hdl.handle.net/2117/407759.

[4] P. Forner Gomez, *Source code repository.* [Online]. Available: https://github.com/polforner/TFG.

[5] "Salary: Researcher in spain 2024," Glassdoor. (Mar. 10, 2024), [Online]. Available: https://www.glassdoor.com/Salaries/spain-researcher-salary-SRCH_IL.0,5_IN219_KO6,16.htm (visited on 03/11/2024).

[6] "Salary: Project manager in madrid, spain 2024," Glassdoor. (Mar. 11, 2024), [Online]. Available: https://www.glassdoor.com/Salaries/madrid-project-manager-salary-SRCH_IL.0,6_IM1030_KO7,22.htm (visited on 03/11/2024).

[7] "Salary: Software developer in spain 2024," Glassdoor. (Mar. 1, 2024), [Online]. Available: https://www.glassdoor.com/Salaries/spain-software-developer-salary-SRCH_IL.0,5_IN219_KO6,24.htm (visited on 03/11/2024).

[8] "Haskell intset." (), [Online]. Available: https://hackage.haskell.org/package/containers-0.7/docs/Data-IntSet.html (visited on 06/19/2024).

[9] J. P. R. Sales, *Jproyo/dynamic-pipeline*, original-date: 2021-03-27T11:12:27Z, Jul. 21, 2021. [Online]. Available: https://github.com/jproyo/dynamic-pipeline (visited on 03/04/2024).

[10] "What is the average lifespan of a computer?" (), [Online]. Available: https://www.hp.com/in-en/shop/tech-takes/post/average-computer-lifespan (visited on 03/11/2024).

[11] "Glasgow haskell compiler 9.0.1 user's guide." (), [Online]. Available: https://downloads.haskell.org/ghc/9.0.1/docs/html/users_guide/9.0.1-notes.html (visited on 06/19/2024).

[12] "BOE-a-2011-9617 ley 14/2011, de 1 de junio, de la ciencia, la tecnología y la innovación." (), [Online]. Available: `https://www.boe.es/buscar/act.php?id=BOE-A-2011-9617` (visited on 06/19/2024).