An algorithm for incrementally enumerating bitriangles in large bipartite networks

Master Thesis

Master in Innovation and Research in Informatics
Advance Computing

Juan Pablo Royo Sales

October, 2021

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB) UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

Advisors: Edelmira Pasarella, Computer Science Department

Maria-Esther Vidal, Leibniz Information Centre for Science and Technology-TIB, and L3S Centre at the Leibniz University of Hannover

Cristina Zoltan, Computer Science Department

".. when a Mathematical Reasoning can be had it's as great a folly to make use of any other, as to grope for a thing in the dark, when you have a Candle standing by you."

Of the Laws of Chance, John Arbuthnot (1662)

Contents

List of Figures **vi**

List of Tables **viii**

List of Algorithm **ix**

List of Source Code **xi**

1 Introduction **2**

1.1 Problem Statement 5

1.2 Proposed Solution 5

1.3 Contribution 6

1.4 Document Overview 7

1.5 Chapter Summary 7

2 Preliminaries **8**

2.1 Streaming Processing 8

2.2 Dynamic Pipeline Paradigm 9

2.3 Streaming in Haskell Language 10

2.4 Diefficiency Metrics 11

2.5 Chapter Summary 12

3 Related Work **13**

3.1 Subgraph Enumeration 13

3.2 Bitriangle Counting in Bipartite Graphs 15

3.3 Pay-as-you-go Model 17

3.4 Chapter Summary 17

4 Proof of concept: Weakly Connected Components of a Graph 18

4.1 $\mathsf{DP}_{\mathsf{WCC}}$ Algorithm 19

4.2 Empirical Evaluation 24

4.3 Chapter Summary 30

5 Dynamic Pipeline Framework in Haskell **31**

5.1 Framework Design 31

5.1.1 Background 31

5.1.2 Architectural Design 32

5.2 Implementation 34

5.2.1 DSL Grammar 34

5.2.2 DSL Validation 37

5.2.3 Interpreter of DSL (IDL) 39

5.2.4 Runtime System (RS) 44

5.3 Libraries and Tools 47

5.3.1 Parallelization 47

5.3.2 Channels 47

5.4 Chapter Summary 48

6 An Algorithm for Incrementally Enumerating Bitriangles 49

6.1 Preliminaries Definitions 49

6.2 Algorithm Sketch 54

6.3 Dynamic Pipeline for Enumerating Bitriangles 55

6.4 Correctness of the Algorithm 66

6.5 DP-BT-Haskell implementation 67

6.6 Chapter Summary 72

7 Empirical Evaluation 73

7.1 Experiments Configuration 73

7.1.1 Benchmark 74

7.1.2 Metrics 75

7.1.3 Scenarios 76

7.1.4 Implementation 77

7.2 Experimental Results 78

7.2.1 E1: Continuous behavior Analysis 78

7.2.2 E2: Benchmark Analysis 85

7.2.3 E3: Performance Analysis 89

7.3 Discussion 93

7.4 Chapter Summary 95

8 Conclusions and Future Work 97

8.1 Conclusions 97

8.2 Observed Limitations 99

8.3 Future Work 99

A Appendix **102**

A.1 Source Code 102

A.2 Running Experiments 102

A.3 Diefficiency Metrics - Traces 104

Bibliography **105**

List of Figures

1.1 [Int] Example of BT in a BG 5

2.1 [Pre] Inital DP Configuration 10

2.2 [Pre] Evolution of DP 10

4.1 [PoC] Graph WCC Example 19

4.2 [PoC] $\mathsf{DP}_{\mathsf{WCC}}$ Initial Setup 20

4.3 [PoC] $\mathsf{DP}_{\mathsf{WCC}}$ Evolving first state 21

4.4 [PoC] $\mathsf{DP}_{\mathsf{WCC}}$ Evolving second state 21

4.5 [PoC] $\mathsf{DP}_{\mathsf{WCC}}$ Evolving third state 22

4.6 [PoC] $\mathsf{DP}_{\mathsf{WCC}}$ Evolving fourth state 22

4.7 [PoC] $\mathsf{DP}_{\mathsf{WCC}}$ Evolving last state 23

4.8 [PoC] `dief@t` Metrics 26

4.9 [PoC] `dief@t` Metrics (Radial) 27

4.10 [PoC] Thread Metrics: Fraction of Time 28

4.11 [PoC] Memory Metrics: Allocation by Data Type 29

5.1 [DPF-Haskell] Architectural design of DPF-Haskell 33

5.2 [DPF-Haskell] DSL Grammar definition 35

6.1 [IEBT] Example of Bipartite Graph 50

6.2 [IEBT] Example of bitriangles 50

6.3 [IEBT] Definitions Examples Aggregated Wedge 51

6.4 [IEBT] Example of connector wedge 51

6.5 [IEBT] Example Aggregated double-wedge 52

6.6 [IEBT] Example Aggregated bitriangle 53

6.7 [IEBT] $\mathsf{DP}_{\mathsf{BT}}$ Initial setup 56

6.8 [IEBT]]$\mathsf{DP}_{\mathsf{BT}}$ With Filter instances 58

7.1 [EE] $\mathsf{DP}_{\mathsf{BT}}$ Finding Optimal Memory Setup 79

7.2 [EE] `dief@t` General Results 81

7.3 [EE] `dief@t` General Results (Radial) 84

7.4 [EE] E2 Benchmark Results: Criterion Plot 87

7.5 [EE] E2 Total Execution Time Comparison 89

7.6 [EE] Thread Metrics: General overview 91

7.7 [EE] Thread Metrics: Partitioned 91

7.8 [EE] Memory Metrics: Allocation by Data Type 92

List of Tables

4.1 [PoC] Execution times 24

4.2 [PoC] Mean Execution times 25

6.1 [IEBT] Summary of notations and their meanings 53

6.2 [IEBT] Summary of Channels used in DP-BT-Haskell 55

6.3 [IEBT] Summary of auxiliary functions for handling DP-BT-Haskell internals 56

7.1 [EE] Selected Networks of Bipartite Graph 74

7.2 [EE] Experiment Data Setup for experiments 77

7.3 [EE] E1 Procedure 80

7.4 [EE] E1 `dief@t` values 83

7.5 [EE] E2 Procedure 86

7.6 [EE] E2 Average Execution Time 88

7.7 [EE] E3 Procedure 90

List of Algorithms

[A1] Source ($\mathsf{Sr}_{\mathsf{BT}}$) 57

[A2] Generator (G_{BT}) 57

[A3] Sink ($\mathsf{Sk}_{\mathsf{BT}}$) 58

[A4] Filter (F_{BT}) 59

[A5] Actor1 ($actor_1$) 59

[A6] Actor2 ($actor_2$) 61

[A7] Actor3 ($actor_3$) 62

[A8] Actor4 ($actor_4$) 63

[A9] Function `buildBtVertex` 64

[A10] Function `buildBtEdge` 65

List of Source Code

5.1 [Flow.hs] Σ enconding of G_{dsl} 35

5.2 [Flow.hs] Σ enconding of G_{dsl} - Especial non-terminals 36

5.3 [Repeated.hs Example of DPP encoded in G_{dsl} 36

5.4 [Stage.hs] Validating encoded in G_{dsl} - FCF 38

5.5 [Stage.hs] Using validation of DPP encoded in G_{dsl} 39

5.6 [Stage.hs] Using with Interpreters of DPP encoded in G_{dsl} . . 40

5.7 [Stage.hs] WithSource Associate Type Details 41

5.8 [Stage.hs] Stage Data Type 42

5.9 [Stage.hs] Filter / Actor Data Type 42

5.10 [Stage.hs] Filter / Actor smart constructors and combinators 43

5.11 [Stage.hs] Generator 44

5.12 [Stage.hs] unfoldF 45

5.13 [Stage.hs] UnfoldFilter combinators 46

6.14 [BTriangle.hs] Enconding of DP-BT-Haskell 67

6.15 [BTriangle.hs] actor_1 68

6.16 [BTriangle.hs] actor_2 69

6.17 [BTriangle.hs] actor_3 70

6.18 [BTriangle.hs] actor_4 71

6.19 [`Edges.hs`] `filterBTByVertex` and `filterBTByEdge` 72

A.20 [`metrics.csv`] Metrics CSV Dbpedia 104

A.21 [`results.csv`] Results CSV Dbpedia 104

Acknowledgements

I would like to express my gratitude first to the main supervisor of this work, Edelmira Pasarella, who has taken the chance to accept me as her student without knowing me beforehand and supporting me during the whole process. I would like to thank as well to the other supervisors: Cristina Zoltan who helped me, among other things, with the formalization of the most challenging part, the algorithm, and Maria-Esther Vidal for teaching and guiding me in the empirical evaluation part. I wish to thank to my partner, Erica, who has supported me for the last two years and a half during this journey. I would like to make a special thanks to my 6-years old daughter, Ludmila, who despite her young age, she has perfectly understood when his father needed to spend more time studying than playing with her. I hope this has also played an important lesson in your life Ludmila: sometimes you need to work hard to achieve something you want. Finally, grateful to myself because I could manage to deal with all the commitments: an eight-hour daily work, the classes, my parenthood, and the rest of my personal life.

Abstract

In this work, we tackle the problem of providing an algorithm for incrementally enumerating bitriangles in large bipartite Networks. A bipartite network is a graph with two disjoint vertex sets where edges only connect vertices from different sets. In bipartite graphs, the bitriangle is considered the smallest unit of cohesion. For this kind of networks, counting bitriangles is extremely useful for conducting social analysis and computing metrics such as clustering coefficient. This is why lately there has been efforts in developing efficient algorithms for counting bitriangles. However, there are cases in which knowing structural information of bitriangles in a bipartite network is required; e.g. in disorder/disease gene association networks where it is important to know the gene causing the disease. In general, specific information about bitriangles is useful for any problem requiring linking data in bipartite networks. Depending on the size of the graph, enumerating bitriangles could be a high-resource consuming task and an "all-or-nothing" computation model might not deliver any result at all. To be concrete, in this work we propose, implement and empirically evaluate an algorithm for enumerating bitriangles incrementally according to some specific criteria. Additionally, we provide a correctness proof of the algorithm. Our proposal is based on the Dynamic Pipeline computational model that nicely supports the *pay-as-you-go* approach. In this way users are able to receive results and conduct analysis as long as they have computational resources at their disposal. This solution could be extremely useful and powerful for real networks and scenarios like expressed before. Apart from this main contribution, we develop a Dynamic Pipeline Framework in (parallel) Haskell. In fact, the implementation of our algorithm relies upon this framework. Finally, we conduct, analyze and report experiments against large bipartite networks having up to 350 millions of bitriangles. Obtained results satisfy our expectations. To assess the incremental delivery of results we measure the *Diefficiency metrics*, i.e. the continuous efficiency of the implementation of the algorithm for generating incremental results.

Chapter 1

Introduction

In many real-world applications, the relationships among two different types of entities can be modeled by means of bipartite graphs. This is a graph in which the set of vertices is formed by the union of two disjoint sets of vertices corresponding to the two different types of entities in the relationship. In this kind of bipartite graph or network, edges only connect vertices from each one of the different entities or vertices. In the literature, we can find many examples of bipartite networks (also called affiliation networks or two-mode networks) in different domains. Just for mentioning some few examples we have phenotype-disease gene associations network (*diseasome* bipartite network) [1], drugs-side effects network [2], customer–product network, author-paper network, the Netflix subscribers-TV shows etc.

In general, the majority of metrics used for analyzing unipartite graphs, for instance, clustering coefficient, social analysis, or triangle-based community computation [3–5] are based on computing the number of triangles in the network. One of the most common techniques to analyze bipartite networks, i.e. to compute graph parameters as clustering coefficient, etc., is to transform them into classical unipartite graphs by means of a method called projection. However, the projection of the bipartite graph distorts the relationships represented in the original networks. Among other problems caused by transforming bipartite networks to unipartite ones, there is the upgrowth of the number of links and hence the distortion of some properties of the original graph such as the number of triangles, the density, etc. (see [6] for more details). In particular, this problem impacts the in-memory manipulation of the graph and distorts the link prediction for bipartite networks.

Discarding the transformation of bipartite graphs into unipartite graphs, in a bipartite graph, there are no triangles as classically defined for graphs. Thus, since computing and using triangles do not fit well for the case of bipartite graphs, Opsahl [7] proposes to use another locality graph pattern or motif, the bitriangle or 6-cycle, i.e. a cycle with three vertices from one type of vertices and three vertices from the other type of vertices, as the smallest unit of cohesion of a bipartite graph. In his work, Opsahl argues that bitriangles in bipartite graphs capture the idea of the triadic closure in unipartite graphs. Yang et al. [8] study the problem of counting bitriangles and, propose and analyze, different algorithms to do it. In particular, they propose an algorithm for local counting bitriangles. This is, counting bitriangles in a bipartite graph in which a given vertex/edge occurs.

For example, in [9] the link prediction problem in a social network is stated as, given a snapshot of a social graph, inferring which new interactions among its members are likely to occur in the near future. In that work, authors develop approaches to link prediction based on measures of the "proximity" of nodes in a network. In [10] the local link prediction problem is addressed for the specific case of bipartite graph. In this work, according to the way in which the network is analyzed, the authors described the link prediction problem twofold, the local link prediction problem and the latent link prediction problem. The local link prediction problem only considers the immediate neighborhood –in particular, the triangle model – of vertices. On the contrary, in the latent link prediction problem, the whole model of the network is used. Local link prediction in bipartite networks can be revisited using a bitriangle model approach. Link prediction problem considers evolving networks. However, depending on the nature of the addressed problem, counting bitriangles in a (possibly persistent) bipartite graph is not enough to establish underlying relationships among its vertices. This happens because establishing these relationships could require knowing specific structural details. We mean that not only the number of bitriangles is important but which are these bitriangles. This is, problems that require the enumeration of bitriangles in a bipartite graph. For example, in the *diseasome* bipartite network presented [1] two disorders are connected if there is a gene that is implicated in both. In order to make decisions, a scientist not only could need to know how many bitriangles are in that graph but –an even more specific question– given a disorder which genes are involved (connected) to it. Another example could be when considering the Netflix subscribers -TV Shows bipartite network where two vertices

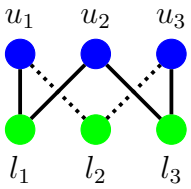
are connected if a subscriber watches or follows a TV show. Knowing which bitriangles containing a specific TV Show could help to make decisions about finishing or creating new seasons of that TV Show.

Depending on the size of the considered bipartite graph, enumerating bitriangles can be a high resource-consuming process and an "all-or-nothing" computation approach can lead to a timeout or memory stuck. This is why, for overcoming this problem, instead of considering a classical enumerating algorithm where computation produces "all-or-nothing" results, an iterative "pay-as-you-go" approach [11] can be followed. From our point of view, this means that bitriangles can be incrementally emitted as results. Users continuously receive answers from the algorithm as long as the provided resources support the computation.

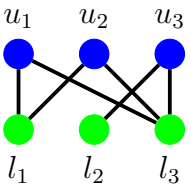
In this work we tackled the problem of providing and implementing an Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network (IEBT). The algorithm must emit results in an *incremental* way because we want the user to be able to obtain bitriangles as they are computed. Effective streaming processing of large amounts of data has been studied for several years [12–14] as a key factor providing fast and incremental results in big data algorithmic problems. One of the most explored techniques, regardless of the approach, is the exploitation of parallel techniques to take advantage of the available computational power as much as possible. In this regard, the Dynamic Pipeline Paradigm (DPP) [15] has lately emerged as one of the models that exploit data streaming processing using a dynamic pipeline parallelism approach [14]. This computational model relies on a functional approach, where the building blocks are functional stages to construct pipelines that dynamically enlarge and shrink depending on incoming data. Besides, the implementation of an algorithm according to the DPP is suitable to generate incremental results. We believe that DPP is a proper computational model for solving the IEBT. Moreover, since DPP is focused on dynamic functional units, it encourages us to use Haskell Programming Language, a purely functional programming language where functions are first class citizens, for implementing IEBT under the DPP.

1.1 Problem Statement

A bitriangle in a bipartite graph is defined as a 6-cycle with three vertices from one type of vertex set and three vertices from the other type of vertex set. We can also say that it is formed by two connected wedges closed by an additional wedge [8]. An example of what it is a Bitriangle (BT) and what it is not, can be seen in Figure 1.1a and Figure 1.1b.



(A) Example of BT in a BG



(B) Not a BT in a BG

FIGURE 1.1: Example of BT in BG. In the left figure we see a bitriangle well formed by 3 wedges. In the right image the figure does not form a bitriangle because l_2 is only connected with u_3 breaking the cycle

Enumerating all the possible BT is computationally hard. However, as we have stated before, most frequently only partial results are needed. In that sense, we can provide a query-oriented algorithm to search all the BT that matches some query criteria and that incrementally deliver results to the user. In this work, we propose and implement an Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network.

1.2 Proposed Solution

The solution proposed in this work is to implement an Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network using Dynamic Pipeline Paradigm implemented in Haskell Programming Language (Haskell). In order to achieve that goal, we first conduct a proof of concept to assess the feasibility of using Haskell for implementing an algorithm with the DPP. In that assessment, we work on solving the problem of Weak Connected Components (WCC) of a graph. Then, we develop a Dynamic Pipeline Framework (DPF) written in Haskell Programming Language that could help to implement any algorithm using DPP. Following that, we provide the formal definition of IEBT using DPP, in a pseudo-code format an its correctness

proof. Finally, we provide the implementation of the algorithm, called DP_{BT} in Haskell (DP-BT-Haskell), using the DPF.

1.3 Contribution

The main contribution of this work is to implement, formalize and empirically evaluate a Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network using Dynamic Pipeline Paradigm written in Haskell. We believe that this implementation is a step forward in the field, and we think our approach opens new research lines and improvements to be addressed in the future.

In order to assess the feasibility of DPP implemented in Haskell, we made a proof of concept solving Weak Connected Components problem of a graph using DPP with Haskell. We have also empirically evaluated this implementation with interesting results that we are going to cover in chapter 4.

Finally, and as a result of the proof of concept work, is the development and publication of a Haskell framework called `dynamic-pipeline` [16]. The framework was published on The Haskell Package Repository (Hackage) [17] on 2021 June 17th in its first version, providing to Haskell community the ability to build algorithms using DPP. This is a novel contribution since it is the first library published on Haskell that implements DPP.

To summarize, our contributions are:

- We introduce, implement and empirically evaluate an Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network under the Dynamic Pipeline Paradigm.
- We conduct a proof of concept to assess the use of Haskell Programming Language as an implementation language for Dynamic Pipeline Paradigm solving the Weak Connected Components problem.
- We develop a Dynamic Pipeline Framework (DPF) in (parallel) Haskell to implement any algorithm using Dynamic Pipeline Paradigm.

1.4 Document Overview

The document is organized as follows. In chapter 2 we present and describe the preliminary concepts needed to support this work such as Streaming Processing, Dynamic Pipeline Paradigm, and Streaming processing related to Haskell and Diefficiency Metrics. Following that, in chapter 3 we describe some work done in related fields like subgraph enumeration problems, using other streaming models to compute subgraphs, counting bitriangles in bipartite graphs, and *pay-as-you-go* model. Additionally, in chapter 4 we present and describe a proof of concept that we conducted for verifying the suitability of Haskell as an implementation language for DPP. Continuing with that, in chapter 5 we deeply describe the Dynamic Pipeline Framework written in Haskell, its architectural design, and the techniques used for conducting this implementation. After that chapter, we finally arrive at chapter 6 where we focus on the main problem of this work, where we provide the pseudo-code of the algorithm for incrementally enumerating bitriangles in large bipartite networks We also present the correctness proof of that algorithm as well as the Haskell implementation using the framework defined in the previous chapter. In chapter 7 we describe the experimental analysis conducted to assert our assumptions and answer our research questions. Finally, at the end of the document in chapter 8, we present the conclusions obtained after conducting this work, as well as the future work and limitations that we observe.

1.5 Chapter Summary

In this chapter, we have presented the motivation of this work, as well as the problem statement related to that motivation and the proposed solution to that problem. Additionally, we provide a document overview to facilitate the reader's review.

Chapter 2

Preliminaries

Before moving forward with the core of the research, we describe the fundamental concepts that support the different parts of our study. That is Stream Processing in general, Dynamic Pipeline Paradigm and, Streaming Processing in the context of Haskell.

2.1 Streaming Processing

The development of Streaming Processing techniques have potentiated areas as massive data processing for data mining algorithms, big data analysis, IoT applications, etc. Data Streaming (DS) has been studied using different approaches [12–14] allowing to process a large amount of data efficiently with an intensive level of parallelization. We can distinguished two different parallelization streaming computational models: Data Parallelism (DAP) and Pipeline Parallelism (PP).

Data Parallelism (DAP) The data is split and processed in parallel and, the computations that perform some action over that subset of data do not have any dependency with other parallel computation. A common model that has been proved successful over the last decade is MapReduce (MR) [18]. Different frameworks or tools like Hadoop [19], Spark [20], etc., support this computational model efficiently. One of the main advantages of this kind of model is the ability to implement stateless algorithms. Data can be split

and treated in different threads or processors without the need for contextual information. On the other hand, when there is a need to be aware of the context, parallelization is penalized, each computational step should be fully calculated before proceeding with the others. For example, this is the case of `reduce` operation on many of the above-mentioned frameworks or tools.

Pipeline Parallelism (PP) It break the computation in a series of sequential stage, where each stage takes the result of the previous stage as an input and downstream its results to the next. Each *pipeline Stage* is parallelized and, it could potentially exist one stage per data item of the stream. The communication between stages takes place through some means, typically channels. One of the main advantages of this model is that the stages are non-blocking, meaning that there is no need to wait to process all data to run the next stage. This kind of paradigm enables computational algorithms that can generate incremental results, preventing the user waits until the end of the whole data stream processing to get a result. On the other hand, the disadvantage over DAP is that although pipeline stages are parallelized, some intensive computation in one stage might delay processing the next stage because of its sequential dependency nature. Therefore, the user must be sure each stage runs extremely fast computations on it.

The nature of our problem requieres that results are output incrementally, i.e. Bitriangles are emitted incrementally. Additionally, data need to be aware of the context to compute the BTs. Considering the stream processing models presented above, we have chosen PP. We think it is the model that better fits the requirements of the enumerating incrementally bitriangles problem. We are going to see in the next section what is the specific PP computation model used for that purpose.

2.2 Dynamic Pipeline Paradigm

The *Dynamic Pipeline Paradigm* (DPP) [15] is a PP computational model based on a one-dimensional and unidirectional chain of stages connected by means of channels synchronized by data availability. This chain of stages is a computational structure called *Dynamic Pipeline* (DP). A DP stretches and shrinks depending on the spawning and the lifetime of its stages, respectively.

Modeling an algorithmic solution as a DP corresponds to define a dynamic computational structure in terms of four kinds of stages: *Source* (Sr), *Generator* (G), *Sink* (Sk) and *Filter* (F) stages. In particular, the specific behavior of each stage to solve a particular problem must be defined as well as the number and the type of channels connecting them. Channels are unidirectional according to the flow of the data. The *Generator* stage is in charge of spawning *Filter* stage instances. This particular behavior of the *Generator* gives the elastic capacity to DPs. *Filter* stage instances are stateful operators in the sense described in [21]. This is, *Filter* instances have a state. The deployment of a DP consists in setting up the initial configuration depicted in Figure 2.1. The activation of a DP starts when a stream of data items arrives at the initial configuration of the DP. In particular, when a data stream arrives to the *Source* stage. During the execution, the *Generator* stage spawns *Filter* stage instances according to incoming data and the *Generator* defined behavior. This evolution is illustrated in Figure 2.2. If the data stream is bounded, the computation finishes when the lifetime of all the stages of DP has finished. Otherwise, if the stream data is unbounded, the DP remains active and incremental results are output.

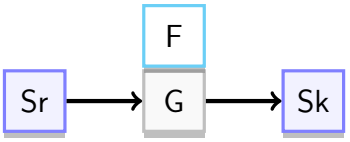


FIGURE 2.1: Initial configuration of a Dynamic Pipeline. An initial DP consists of three stages: Sr, G together its filter parameter F, and Sk. These stages are connected through its channels –represented by right arrows– as shown in this figure.

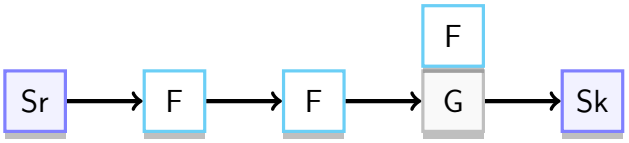


FIGURE 2.2: Evolution of a DP. After creating some filter instances (shadow Filter squares) of the filter parameter (light Filter square) in the Generator, the DP has stretched.

2.3 Streaming in Haskell Language

Streaming computational models have been implemented in Haskell Programming Language during the last 10 years. One of the first libraries in the ecosystem was `conduit` [22] in 2011. After that, several efforts on improving

streaming processing on the language has been made not only at abstraction level for the user but as well as performance execution improvements like `pipes` [23] and `streamly` [24] lately. Moreover, there is an empirical comparison between those three, where a benchmark analysis has been conducted [25].

Although most of those libraries offer the ability to implement DAP and PP, none of them provide clear abstractions to create DPP models because the setup of the stages should be provided beforehand. In the context of this work, we have done a proof of concept at the beginning, but it was not possible to adapt any of those libraries to implement properly DPP. The closest we have been to implement DPP with some of those libraries was when we explored `streamly`. In this case, there is a `foldrS` combinator that could have been proper to the purpose of generating a dynamic pipeline of stages based on the data flow, but it was not possible to manipulate the channels between the stages to control the flow of the data. It is important to remark that, even though, the library `streamly` implements channels, they are hidden from the end-user and, there is not a clear way to manipulate them.

To the best of our knowledge no similar library under the DPP approach has been written in Haskell Programming Language. One important motivation to develop our own framework is that we not only wanted to satisfy our research needs but, as a novel contribution, we wanted to deliver a DPF to the Haskell community as well. We hope this contribution encourages and helps writing algorithms under the Dynamic Pipeline Paradigm.

2.4 Diefficiency Metrics

In proof of concept in chapter 4 and in this work on the empirical analysis chapter 7, we use two important metrics to measure the diefficiency, i.e. continuous efficiency of a program to generate incremental results. The metrics to measure diefficiency are Diefficiency Metric `dief@t` (`dief@t`) and Diefficiency Metric `dief@k` (`dief@k`) [26]. The metric `dief@t` measures the continuous efficiency during the first t time units of execution regarding the results generated by the program. The higher value of the `dief@t` metric, the better the continuous behavior. The metric `dief@k` measures the continuous efficiency while producing the first k answers regarding the results generated by DP-BT-Haskell. The lower the value of the `dief@k` metric, the better the continuous

behavior. Both metrics have been measured using `diefpy` Tool (`diefpy`) [27] and traces obtained by the execution of the experiment scenarios (Traces examples are provided in section A.3). Apart from this, `diefpy` generates two different kind of plots. On the one hand, a bi-dimensional plot containing all the (x, y) points taken from traces like Source Code A.21, where each x is the t time where the answer was generated and the y is the generated answer number. This plot is useful to have a visual view of the continuous behavior. On the other hand, a radial plot contains the visual comparison of `dief@t` metric with respects to other non-continuos metrics such as **i)** Completeness (Comp) which is the total number of answers produced by the scenario, **ii)** Time for the first tuple (TFFT) which measure the elapsed time spent by the scenario to produce the first answer, **iii)** Execution Time (ET) which measures the elapsed time spent by the scenario to complete the execution of a query and, **iv)** Throughput (T) which measure the number of total answers produced by the scenario after evaluating a query divided by its execution time ET .

2.5 Chapter Summary

In this chapter, we have presented the *preliminary work* that has been done on the related areas that affect our research objective. First, we have shown the different stream processing computational models. Then, we have described the DPP. After that, we have presented the available tools and libraries for stream processing in Haskell. Finally, we have presented a brief description of the diefficiency metrics, which are the principal metrics that we use to conduct experimental analysis in both, proof of concept and the algorithm of this work itself.

Chapter 3

Related Work

Having the fact that a BT is a subgraph, at the beginning of this chapter, we explore some works related to the subgraph enumeration problem. In the case of BT in BG, we describe a recent work, that has been conducted in BT counting problem. Finally, we describe some related research that has been done lately regarding *pay-as-you-go* model.

3.1 Subgraph Enumeration

Enumerating subgraphs using Map-Reduce. This work has been presented by Afrati et al. [12] for solving the subgraph enumeration problem using a single round map-reduce. The problem presented in this work is to enumerate all instances of a given subgraph (sample graph) in a large graph using a single map-reduce round. In that work, all the examples are conducted with the smallest subgraph known for unipartite graphs, this is the triangle. The solution proposed is presented as a special case of computing a multiway join but improving complexity reducing the communication cost and computational cost. To achieving this, the authors present an improvement over the *Partition Algorithm of Suri and Vassilvitskii* [28] replicating all edges the same number of times reducing the communication cost to conciliate duplicated triangles. Regarding the computation cost, an improvement over the *multiway-join* algorithm is proposed using an ordering of the buckets node. The main advantage of this work is the use of a stream parallelization model like Map-Reduce, bringing the choice to exploit parallel and distributed

computation to gain efficiency. The use of Map-Reduce combined with the improvements proposed by the authors of this work [12] reduce even more the processing time for the enumeration. One of the limitations is the use of a streaming model like Map-Reduce. Although it is a parallel and distributed model, incremental results are not possible until all the reducers are calculated. Computation can be done in parallel, however the production of subgraphs is restricted by the reducers limiting the capability of delivering results incrementally. The other limitation is the adjustment of computational resources, which in this work is done statically beforehand. Partitioning is done by the number of nodes and edges which is known in advance. As we have stated in chapter 1, DPP overcome both of these limitations by providing a Dynamic Pipeline parallelization model, where the resources can be adjusted dynamically, and results are generated incrementally as soon as they are computed.

Distributed subgraph matching on timely dataflow This work has been presented by Lai et al. [29] for solving the subgraph matching problem in large graphs using a distributed computational model. The main contribution of this work is the optimization of four strategies algorithms use on `Timely` dataflow system [30]. The underlying idea of the proposed algorithm is performing a random partition of the vertices using hashing, where the vertices hashed neighbors are placed on the same partition. Query vertices are attributes, and results are relational tables, enabling the subgraph matching problem to be expressed with natural joins, where the solution is to find the optimal distributed join plan. The join algorithms improved are `BinJoin`, `WOptJoin` and `ShrCube` using the following optimizations techniques: `Batching`,`TrIndexing` and `Compression`. In the case of `Batching`, the optimization relies on processing in batch mode the partial results that match a subset of vertices in a way that each partial result can be batched in a single task to process against the whole result. `TrIndexing` or Triangle Indexing precomputes triangles of a data graph and indices to prune unfeasible results beforehand. Finally, `Compression` maintains intermediate results, matched vertices, in a compressed form. The compression form is an array without unfolding each element with its respective matching pair, reducing communication and maintenance. The results exposed on the empirical analysis in this work show a suitable level of efficiency but, it depends on the machine

characteristics and the topologies of the analyzed graphs. One of the most interesting contributions of this work that we have adapted in our solution is the use of `Compression` technique. In our description of the bitriangle enumeration algorithm, described in chapter 6, we define intermediate objects such as aggregated wedges, aggregated double-wedges, and aggregated bitriangles. Those intermediate structures help us to build and store bitriangles in a compressed representation in order to use less memory footprint and computation time.

motif-paths. Xiaodong Li et al. [31], present an algorithm for calculating the shortest *motif-paths*. A *motif-path* is a concatenation –a path– of two or more *motifs*, where a *motif* is a small graph with few nodes, considered as a fundamental unit of a graph. In their work, authors bring light to the computation of the shortest *motif-paths* between two nodes is a very useful tool to link prediction. They propose an incremental search in an algorithm called *Incremental Motif-path Search (IMS Search)*. The idea of this algorithm starts similarly to other shortest path search algorithms. Giving a source vertex s find the shortest path to a target vertex t. The sketch of the algorithm proceeds in the following manner: *motif-instances* are discovered around some seed. Then, *motif-paths* can be constructed based on those instances. We use this approach of seeds –vertices and edges in our case– for optimizing the representation of the building blocks and the discovery of bitriangles. Additionally, we think this kind of problem is suitable to be implemented with Dynamic Pipeline Paradigm. In effect, using DPP it is possibly to build a dynamic pipeline where each *motif-instance* corresponds to one filter. Therefore, we could eliminate repeated *motif-instances* and wrong path detection by the disposition of the filters and the communication between them.

3.2 Bitriangle Counting in Bipartite Graphs

The problem of Counting BT has been addressed in the work of Yang et al. [8] where they present several *polly-time* algorithms to solve it. Authors present different approaches to calculate using combinatorial algorithmics the number of BT in BG. To be concrete, authors present three algorithms to count all the BT in the graph *(Global Counting)*, and two algorithms to count

only locally BT *(Local Counting)*: this means that given a vertex or an edge, the algorithms count only the number of BT in which that vertex or edge is participating.

In this work, authors based all the algorithms, *Global* and *Local*, on using intermediate small units or graph patterns that could form part of bitriangles. These small graph patterns are: wedge, wj-unit, super-wedge and swj-unit. In the case of *Global Counting*, the best algorithm in terms of time complexity, is the Ranked Super-wedge based algorithm (RSWJ-Count). It is based on ranking the vertices by their degree. Exploring higher degree vertices first allows to detect the ones with more super-wedges, and therefore, count *swj-unit* faster by sharing computations.

In regards to *Local Counting*, the work presents two algorithms. One algorithm locally counts BT given a particular vertex, and the other locally counts BT given a particular edge. In both cases, the algorithms use the same *swj-unit* small unit for counting the bitriangles that are presented in *Global Counting*. The difference with the *swj-unit* based algorithm for counting globally, is that instead of traversing all the vertices or edges to count the *swj-unit*, it explores *swj-unit* locally based on the given vertex or edge. The main advantage of doing *Local Counting* is the ability to reduce processing time when the user does not need to analyze the whole network but just a part of it. The principal limitation of *Local Counting*, apart from sharing the same limitation of *Global Counting* regarding incremental results, is that it does not have the possibility to reuse previous calculations if several vertices or edges searches are requested. In this case, the algorithm needs to be recalculated locally for every edge or vertex request.

In this work, we have adapted the idea of using small units or structures of the graph to find a bigger structure like bitriangle. However, although we use the concept of *wedge* as well, the rest of the small units or structures we have defined are different. Additionally, as said before, following the Lai et al. approach, these structures are represented in a compressed way.

3.3 Pay-as-you-go Model

As we have described in chapter 1, the *pay-as-you-go* computational model focuses on incrementally deliver results on high resource-consumer processes. *pay-as-you-go* model [32] plays a fundamental role in applications that needs to process large amount of data, where the user does not need to obtain all the results but a fraction of them to start making decisions and doing analysis. At the same time, the user is able to administrate the resources he/she can afford. The more resources the user can afford, the more data or faster results he/she will obtain.

Fact-Checking Nguyen et al. [32] have conducted a study with primary focus on *pay-as-you-go* model, where they use incremental quality estimation to provide fact-checking over World Wide Web documents. The work continuously improves the credibility assessment of documents in the database and, users may then examine that information to decide whether to stop or resume validation. Having into consideration the results on this work, we think that a *pay-as-you-go* model can be described properly using Dynamic Pipeline Paradigm. In effect, one of the intrinsic features of DPP is the capability of adjusting the computational resources to the incoming data.

3.4 Chapter Summary

In this chapter, we have presented all the work that is related to our research. First, we have shown all the details of the most recent work regarding subgraph enumeration problem. After that, we analyze the latest and most important work on counting Bitriangle in Bipartite Graph. Then, we have also explored, what is the latest research and explorations in the use of *pay-as-you-go* models.

Chapter 4

Proof of concept: Weakly Connected Components of a Graph

One of the biggest challenges of implementing a Dynamic Pipeline is to find a programming language with a proper set of tools supporting both of the primary features of the DPP: i) *parallel* processing and ii) *strong theoretical* foundations to manage computations as first-class citizens. Haskell is a statically typed pure functional language designed on strong theoretical foundations where computations are primary entities. This pure functional language has evolved from its birth in 1987 and nowadays provides a powerful set of tools for writing multithreading and parallel programs with optimal performance [33, 34].

In the context of this research, we first assessed the suitability of Haskell Programming Language to implement a Dynamic Pipeline. To be concrete, we conducted a proof of concept implementing a Dynamic Pipeline in Haskell for solving a particular and very relevant problem as the computation/enumeration of the Weakly Connected Components of a graph. In particular, the main objective of our proof of concept was to study the critical features required in Haskell for a DPF implementation, the real possibilities of emitting incrementally results, and the performance of such kinds of implementations. Indeed, we explored the basis of an implementation of a DPF in a pure (parallel) functional language as Haskell. This is, we determined the particular features (i.e., versions and libraries) that will allow for an efficient implementation of a

DPF. Moreover, we conducted an empirical evaluation to analyze the performance of the Dynamic Pipeline implemented in Haskell for enumerating WCC with special attention to the emission of results, i.e. WCC, incrementally. In this chapter, we focus on the presentation of our algorithm for enumerating WCC using Haskell as well as the results obtained in the empirical evaluation of its implementation. Almost all the content of this chapter is published in [35]. The details of the specific requirements of the Haskell system according to the results of the proof of concept will be presented in chapter 5.

4.1 $\mathsf{DP}_{\mathsf{WCC}}$ Algorithm

Let us consider the problem of computing/enumerating the (weak) connected components of a graph G using DPP. A connected component of a graph is a subgraph in which any two vertices are connected by paths. Thus, finding connected components of an undirected graph implies obtaining the minimal partition of the set of nodes induced by the relationship *connected*, i.e., there is a path between each pair of nodes. An example of that graph can be seen in Figure 4.1. The input of the Dynamic Pipeline for computing the WCC of a graph, $\mathsf{DP}_{\mathsf{WCC}}$, is a sequence of edges ending with eof[1]. The connected components are output as soon as they are computed, i.e., they are produced incrementally. Roughly speaking the idea of the algorithm is that the weakly connected components are built in two phases. In the first phase filter instance stages receive the edges of the input graph and create sets of connected vertices. During the second phase, these filter instances construct maximal subsets of connected vertices, i.e. the vertices corresponding to (weakly) connected components. $\mathsf{DP}_{\mathsf{WCC}}$ is defined in terms of the behavior of its four kinds stages: *Source* ($\mathsf{Sr}_{\mathsf{WCC}}$), *Generator* ($\mathsf{G}_{\mathsf{WCC}}$), *Sink* ($\mathsf{Sk}_{\mathsf{WCC}}$), and *Filter*($\mathsf{F}_{\mathsf{WCC}}$) stages. Additionally, the channels connecting these stages must be defined. In $\mathsf{DP}_{\mathsf{WCC}}$, stages are connected linearly and unidirectionally through the channels IC_{E} and $\mathsf{IC}_{\mathsf{set(V)}}$. Channel IC_{E} carries edges while channel $\mathsf{IC}_{\mathsf{set(V)}}$ conveys sets of connected vertices. Both channels end by the eof mark. The behavior

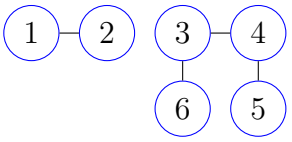


FIGURE 4.1: Example of a graph with two weakly connected components: $\{1, 2\}$ and $\{3, 4, 5, 6\}$

[1]Note that there are neither isolated vertices nor loops in the source graph G.

of $\mathsf{F_{WCC}}$ is given by a sequence of two actors (scripts). Each actor corresponds to a phase of the algorithm. In what follows, we denote these actors by $\mathsf{actor_1}$ and $\mathsf{actor_2}$, respectively. The script $\mathsf{actor_1}$ keeps a set of connected vertices (CV) in the state of the $\mathsf{F_{WCC}}$ instance. When an edge e arrives, if an endpoint of e is present in the state, then the other endpoint of e is added to CV. Edges without incident endpoints are passed to the next stage. When eof arrives at channel $\mathsf{IC_E}$, it is passed to the next stage, and the script $\mathsf{actor_2}$ starts its execution. If script $\mathsf{actor_2}$ receives a set of connected vertices CV in $\mathsf{IC_{set(V)}}$, it determines if the intersection between CV and the nodes in its state is not empty. If so, it adds the nodes in CV to its state. Otherwise, the CV is passed to the next stage. Whenever eof is received, $\mathsf{actor_2}$ passes–through $\mathsf{IC_{set(V)}}$– the set of vertices in its state and the eof mark to the next stage; then, it dies. The behavior of $\mathsf{Sr_{WCC}}$ corresponds to the identity transformation over the data stream of edges. As edges arrive, they are passed through $\mathsf{IC_E}$ to the next stage. When receiving eof on $\mathsf{IC_E}$, this mark is put on both channels. Then, $\mathsf{Sr_{WCC}}$ dies.

Let us describe this behavior with the example of the graph shown in Figure 4.1.

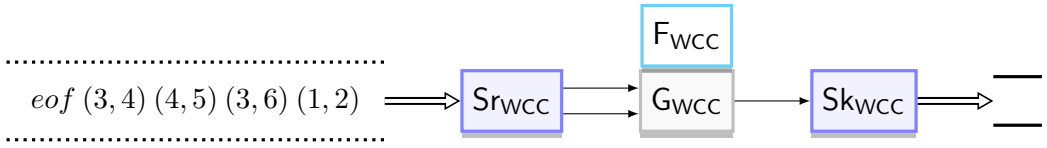
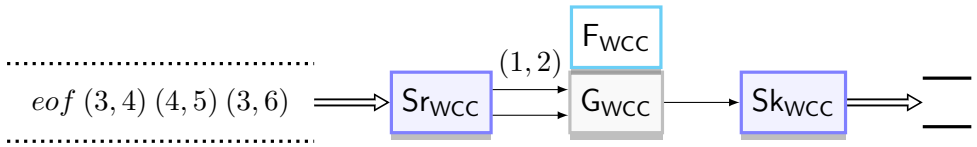
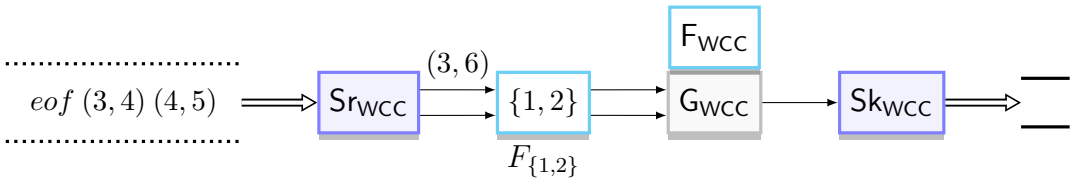


FIGURE 4.2: $\mathsf{DP_{WCC}}$ Initial setup. Stages Source, Generator, and Sink are represented by the squares labeled by $\mathsf{Sr_{WCC}}$, $\mathsf{G_{WCC}}$ and $\mathsf{Sk_{WCC}}$, respectively. The square $\mathsf{F_{WCC}}$ corresponding to the Filter stage template is the parameter of $\mathsf{G_{WCC}}$. Arrows $\rightrightarrows$ between represents the connection of stages through two channels, $\mathsf{IC_E}$, and $\mathsf{IC_{set(V)}}$. The arrow $\rightarrow$ represents the channel $\mathsf{IC_{set(V)}}$ connecting the stages $\mathsf{G_{WCC}}$ and $\mathsf{Sk_{WCC}}$. The arrow $\Longrightarrow$ stands for I/O data flow. Finally, the input stream comes between the dotted lines on the left and the WCC computed incrementally will be placed between the solid lines on the right.

Figure 4.2 depicts the initial configuration of $\mathsf{DP_{WCC}}$. The interaction of $\mathsf{DP_{WCC}}$ with the "external" world is done through the stages $\mathsf{Sr_{WCC}}$ and $\mathsf{Sk_{WCC}}$. Indeed, once activated the initial $\mathsf{DP_{WCC}}$, the input stream – consisting of a sequence containing all the edges in the graph in Figure 4.1 – feeds $\mathsf{Sr_{WCC}}$ while $\mathsf{Sk_{WCC}}$ emits incrementally the resulting weakly connected components. In what follows Figure 4.3, Figure 4.4, Figure 4.5, Figure 4.6 and Figure 4.7 depict the evolution of the $\mathsf{DP_{WCC}}$.

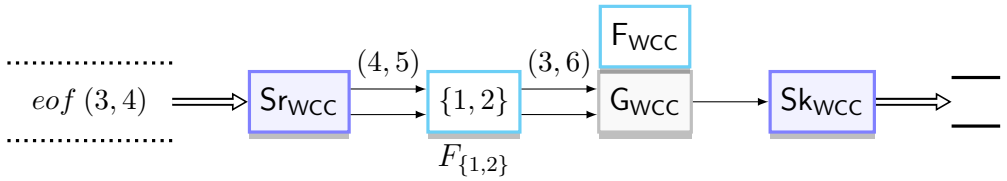


(A) The edge $(1, 2)$ is arriving to $\mathsf{G_{WCC}}$.

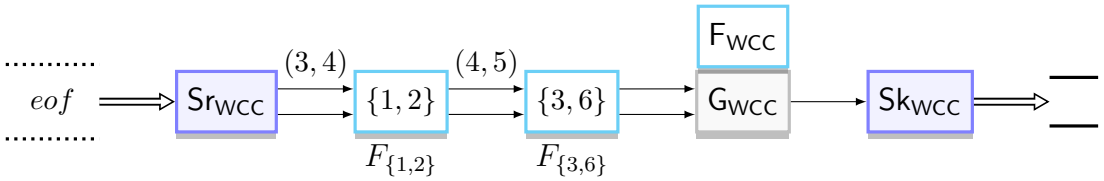


(B) When the edge $(1, 2)$ arrives to $\mathsf{G_{WCC}}$, it spawns a new instance of $\mathsf{F_{WCC}}$ before $\mathsf{G_{WCC}}$. Filter instance $F_{\{1,2\}}$ is connected to $\mathsf{G_{WCC}}$ through channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. The state of the new filter instance $F_{\{1,2\}}$ is initialized with the set of vertices $\{1, 2\}$. The edge $(3, 6)$ arrives to the new filter instance $F_{\{1,2\}}$.

FIGURE 4.3: Evolution of the $\mathsf{DP_{WCC}}$: First state



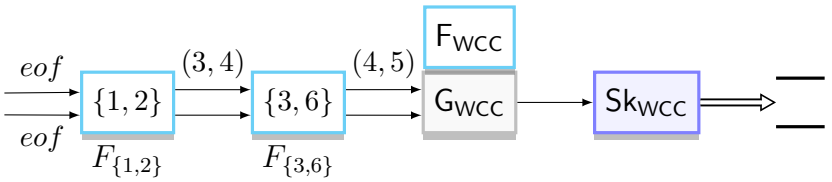
(A) None of the vertices in the edge $(3, 6)$ is in the set of vertices $\{1, 2\}$ in the state of $F_{\{1,2\}}$, hence it is passed through $\mathsf{IC_E}$ to $\mathsf{G_{WCC}}$.



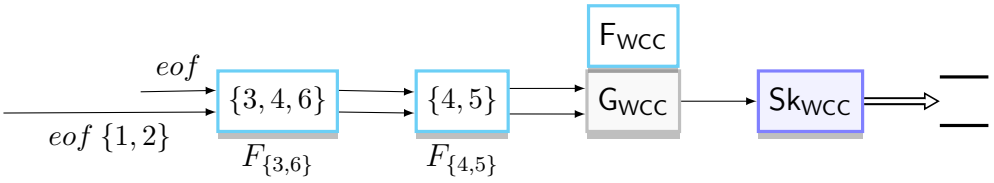
(B) When the edge $(3, 6)$ arrives to $\mathsf{G_{WCC}}$, it spawns the filter instance $F_{\{3,6\}}$ between $F_{\{1,2\}}$ and $\mathsf{G_{WCC}}$. Filter instance $F_{\{1,2\}}$ is connected to the new filter instance $F_{\{3,6\}}$ and this one is connected to $\mathsf{G_{WCC}}$ through channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. The state of the new filter instance $F_{\{3,6\}}$ is initialized with the set of vertices $\{3, 6\}$. The edge $(3, 4)$ arrives to $F_{\{1,2\}}$ and $\mathsf{Sr_{WCC}}$ is fed with the mark eof. Edges $(3, 4)$ and $(4, 5)$ remain passing through $\mathsf{IC_E}$.

FIGURE 4.4: Evolution of the $\mathsf{DP_{WCC}}$: Second state

It is importat to highlight that during the states shown in Figure 4.3a, Figure 4.3b, Figure 4.4a, Figure 4.4b and Figure 4.5a the only actor executed in

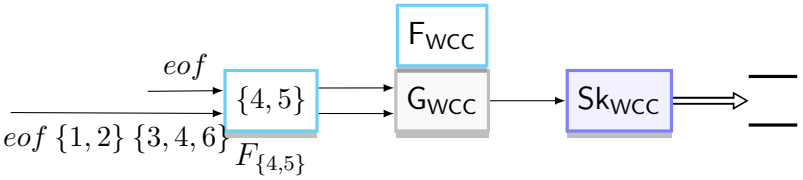


(A) $\mathsf{Sr_{WCC}}$ fed both, $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$, channels with the mark eof received from the input stream in previous state and then, it died. The edge $(4, 5)$ is arriving to $\mathsf{G_{WCC}}$ and the edge $(3, 4)$ is arriving to $F_{\{3,6\}}$.

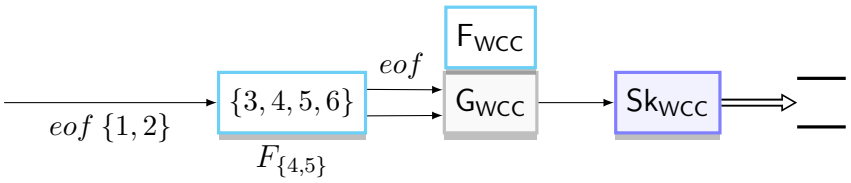


(B) When the edge $(4, 5)$ arrives to $\mathsf{G_{WCC}}$, it spawns the filter instance $F_{\{4,5\}}$ between $F_{\{3,6\}}$ and $\mathsf{G_{WCC}}$. Filter instance $F_{\{3,6\}}$ is connected to the new filter instance $F_{\{4,5\}}$ and this one is connected to $\mathsf{G_{WCC}}$ through channels $\mathsf{IC_E}$ and $\mathsf{IC_{set(V)}}$. Since the edge $(3, 4)$ arrived to $F_{\{3,6\}}$ at the same time and vertex 3 belongs to the set of connected vertices of the filter $F_{\{3,6\}}$, the vertex 4 is added to the state of $F_{\{3,6\}}$. Now, the state of $F_{\{3,6\}}$ is the connected set of vertices $\{3, 4, 6\}$. When the mark eof arrives to the first filter instance, $F_{\{1,2\}}$, through $\mathsf{IC_{set(V)}}$, this stage passes its partial set of connected vertices, $\{1, 2\}$, through $\mathsf{IC_{set(V)}}$ and dies. This action will activate $\mathsf{actor_2}$ in next filter instances to start building maximal connected components. In this example, the state in $F_{\{3,6\}}$, $\{3, 4, 6\}$, and the arriving set $\{1, 2\}$ do not intersect and, hence, both sets of vertices, $\{1, 2\}$ and $\{3, 4, 6\}$ will be passed to the next filter instance through $\mathsf{IC_{set(V)}}$.

FIGURE 4.5: Evolution of the $\mathsf{DP_{WCC}}$: Third state

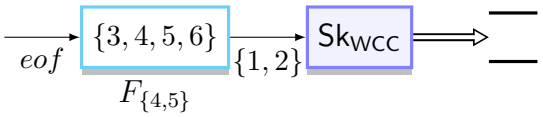


(A) The set of connected vertices $\{3, 4, 6\}$ is arriving to $F_{\{4,5\}}$. The mark eof continues passing to next stages through the channel $\mathsf{IC_E}$.



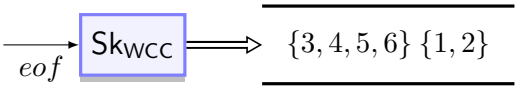
(B) Since the intersection of the set of connected vertices $\{3, 4, 6\}$ arrived to $F_{\{4,5\}}$ and its state is not empty, this state is enlarged to be $\{3, 4, 5, 6\}$. The set of connected vertices $\{1, 2\}$ is arriving to $F_{\{4,5\}}$

FIGURE 4.6: Evolution of the $\mathsf{DP_{WCC}}$: Fourth state

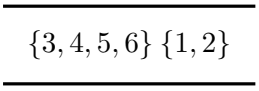


(A) $F_{\{4,5\}}$ has passed the set of connected vertices $\{1, 2\}$ and it is arriving to $\mathsf{Sk}_{\mathsf{WCC}}$. The mark eof is arriving to $F_{\{4,5\}}$ through $\mathsf{IC}_{\mathsf{set(V)}}$.

(B) Since the mark eof arrived to $F_{\{4,5\}}$ through $\mathsf{IC}_{\mathsf{set(V)}}$, it passes its state, the set $\{3, 4, 5, 6\}$ through $\mathsf{IC}_{\mathsf{set(V)}}$ to next stages and died. The set of connected vertices $\{1, 2\}$ arrived to $\mathsf{Sk}_{\mathsf{WCC}}$ and this implies that $\{1, 2\}$ is a maximal set of connected vertices, i.e. a connected component of the input graph. Hence, $\mathsf{Sk}_{\mathsf{WCC}}$ output this first weakly connected component.



(C) Finally, the set of connected vertices $\{3, 4, 5, 6\}$ arrived to $\mathsf{Sk}_{\mathsf{WCC}}$ and was output as a new weakly connected component. Besides, the mark eof also arrived to $\mathsf{Sk}_{\mathsf{WCC}}$ through $\mathsf{IC}_{\mathsf{set(V)}}$ and thus, it dies.



(D) The weakly connected component of in the graph Figure 4.1 such as they have been emitted by $\mathsf{DP}_{\mathsf{WCC}}$.

FIGURE 4.7: Last states in the evolution of the $\mathsf{DP}_{\mathsf{WCC}}$

any filter instance is actor_1 (constructing sets of connected vertices). Afterwards, although actor_1 can continue being executed in some filter instances, there are some instances that start executing actor_2 (constructing sets of maximal connected vertices). This is shown from Figure 4.5a to Figure 4.7a.

4.2 Empirical Evaluation

For the empirical evaluation we consider the following research questions: **RQ1)** Does $\mathsf{DP}_{\mathsf{WCC}}$ in Haskell support the dynamic parallelization level that $\mathsf{DP}_{\mathsf{WCC}}$ requires? **RQ2)** Is $\mathsf{DP}_{\mathsf{WCC}}$ in Haskell competitive compared with default implementations on base libraries for the same problem? **RQ3)** Does $\mathsf{DP}_{\mathsf{WCC}}$ in Haskell handle memory efficiently?

We have conducted different kinds of experiments to test our assumptions and verify the correctness of the implementation. First, we have performed an *Implementation Analysis* in which we have selected some graphs from Stanford Network Data Set Collection (SNAP) [36] and analyze how the implementation behaves under real-world graphs if it timeouts or not and if it is producing correct results in terms of the amount of WCC that we know beforehand. We have also tested the implementation doing a *Benchmark Analysis* where we focus on two different types of benchmarks. On the one hand, using `criterion` library [37], we have evaluated a benchmark between our solution and WCC algorithm implemented in `containers` Haskell library [38] using `Data.Graph`. On the other hand, we have compared if the results are being generated incrementally in both cases and how that is done during the pipeline execution time. This last analysis has been conducted using `diefpy` tool [26, 27]. Finally, we have executed a *Performance Analysis* in which we have to gather profiling data from Glasgow Haskell Compiler (GHC) for one of the real-world graphs to measure how the program performs regarding multithreading and memory allocation.

Implementation analysis The following represents the execution for running these graphs on our DPP implementation.

Network	Exec Param	MUT Time	GC Time	Total Time
Enron Emails	`+RTS -N4 -s`	2.797s	0.942s	3.746s
Astro Physics Coll Net	`+RTS -N4 -s`	2.607s	1.392s	4.014s
Google Web Graph	`+RTS -N8 -s`	137.127s	218.913s	356.058s

TABLE 4.1: This table shows the GHC execution time measurement of selected networks. Column `Exec Param` describe the runtime flags provided to the running program. `MUT Time` is the time in seconds the program was executing computations (a.k.a. program time). `GC` Time is garbage collector time. Total time is the sum of `MUT` + `GC` time.

It is important to point out that since the first two networks are smaller in the number of edges compared with *web-Google*, executing those with 8 cores as the `-N` parameters indicates does not affect the final speed-up since GHC is not distributing threads on extra cores because it handles the load with 4 cores only. As we can see in Table 4.1, we are obtaining remarkable execution times for the first two graphs, and it seems not to be the case for *web-Google* due to the topology of the graph; it is denser in terms of connected components than the others.

Benchmark Analysis Regarding mean execution times for each implementation on each case measure by `criterion` library [37], we can display the following results:

Network	**DP-WCC-Haskell**	**Haskell `containers`**	**Speed-up**
Enron Emails	4.68s	6.46s	1.38
Astro Physics Coll Net	4.98s	6.95s	1.39
Google Web Graph	386s	106s	0.27

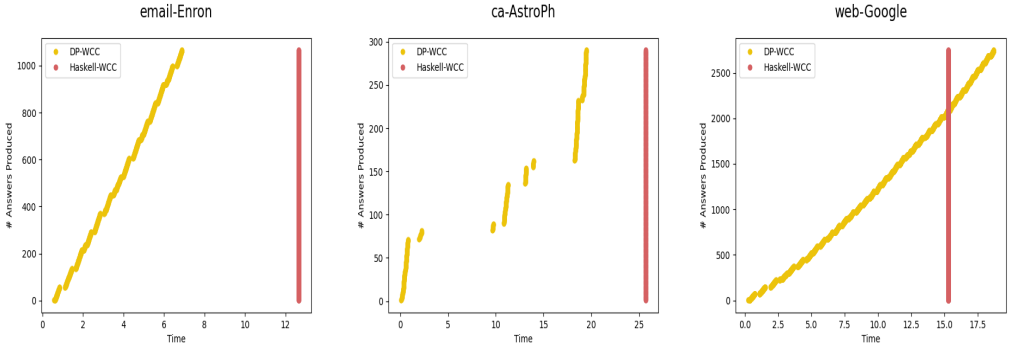
TABLE 4.2: Mean execution time of each network running under `criterion` library comparing both implementations in Haskell: DP-WCC-Haskell and `containers` lib. `criterion` runs 1000 times each implementations and takes mean execution times of each. `Speed-up` column shows the ratio between `Haskell containers` and `DP-WCC-Haskell`

These results allow for answering Question [Q2], where we have seen that the graph topology is affecting the performance and the parallelization, penalizing DP_{WCC} in Haskell (DP-WCC-Haskell) for this particular case. In this benchmark, the solution against a non-parallel `containers` `Data.Graph` confirms the hypothesis.

Diefficency metrics The definition of this metric has been discussed on section 2.4. Some considerations are needed before starting to analyze the data gathered with `diefpy` Tool (`diefpy`) tool. Firstly, the tool is plotting the results according to the traces generated by the implementation, both DP-WCC-Haskell and Haskell *containers*. By the nature of DPP model, we can gather or register that timestamps as long as the model is generating results. In the case of Haskell `containers`, this is not possible since it calculates WCC at once. This is not an issue and we still can check at what point in time all

WCC in Haskell `containers` are generated. In those cases, we are going to see a straight vertical line.

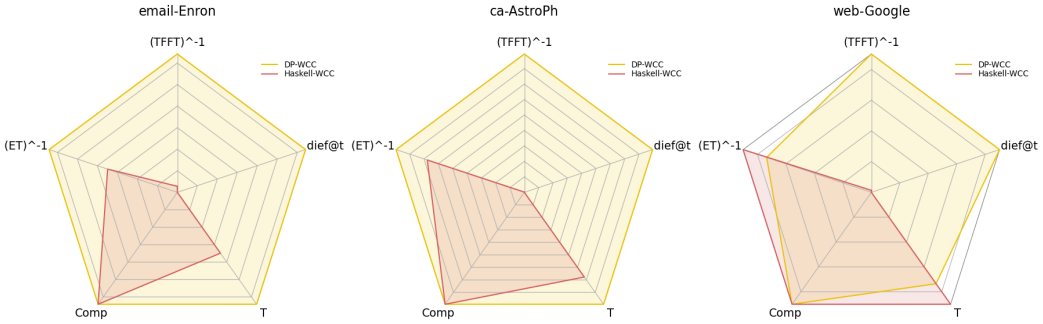
It is important to remark that we needed to scale the timestamps because we have taken the time in nanoseconds. After all, the incremental generation between one WCC and the other is very small but significant enough to be taken into consideration. Thus, if we left the time scale in integer milliseconds, microseconds, or nanoseconds integer part, it cannot be appreciated. In case of escalation, we are discounting the nanosecond integer of the first generated results resulting in a time scale that starts close to 0. This does not mean that the first result is generated at 0 time, but we are discarding the previous time to focus on how the results are incrementally generated.



(A) email-Enron `dief@t` (B) ca-AstroPh `dief@t` (C) web-Google `dief@t`

FIGURE 4.8: This plots are showing the `dief@t` on the three networks comparing both Haskell implementations DP-WCC-Haskell and `containers` lib. Red lines indicates `containers` Haskell `dief@t` metric. Yellow points indicates DP-WCC-Haskell `dief@t` metric

Based on the results shown in Figure 4.8 and Figure 4.9 above, all the solutions in DP-WCC-Haskell indicates continuous behavior, but there is some difference that we would like to remark. In the case of *email-Enron* and *ca-AstroPh* graphs as we can see in Figure 4.8a and Figure 4.8b, there seems to be a more incremental generation of results. This behavior is measured with the values of Diefficiency Metric `dief@t`. *ca-AstroPh* as it can be seen in Figure 4.8b, is even more incremental, and it is showing a clear separation between some results and others. The *web-Google* network, which is shown in Figure 4.8c, is a little more linear, and that is because all the results are being generated with very little difference in time between them. Having the biggest WCC at the end of *web-Google* DPP algorithm it is retaining results until the biggest WCC can be solved, which takes longer.



(A) email-Enron `dief@t` (B) ca-AstroPh `dief@t` (C) web-Google `dief@t`

FIGURE 4.9: Radial plot shows how the different metrics provided by `diefpy` tool such as T, TFFT, `dief@t`, ET and Comp are related each other for each Haskell implementation: DP-WCC-Haskell and `containers`. Red area indicates `containers` Haskell `dief@t` metric. Yellow area indicates DP-WCC-Haskell `dief@t` metric

Multithreading For analyzing parallelization and multithreading we have used *ThreadScope* [39] which allows us to see how the parallelization is taking place on GHC at a fine grained level and how the threads are distributed throughout the different cores requested with the `-N` execution `ghc-option` flag. The distribution of the load is more intensive at the end of the execution, where `actor2` filter stage of the algorithm is taking place and different filters are reaching execution of that second actor. We can appreciate how many threads are being spawned and by the tool and if they are evenly distributed among cores.

Figure 4.10 zooms in on *ThreadScope* output in a particular moment, approximately in the middle of the execution. The numbers inside green bars represent the number of threads that are being executed on that particular core (horizontal line) at that execution slot. Thus, the number of threads varies among slot execution times, because as it is already known, GHC implements *Preemptive Scheduling* [40]. It can be appreciated in Figure 4.10 our first assumption that the load is evenly distributed because the mean number of executing threads per core is 571.

Memory allocation Another important aspect in our case is how the memory is being managed to avoid memory leaks or other non-desired behavior that increases memory allocation during the execution time. This is even more important in the particular implementation of WCC using DPP model

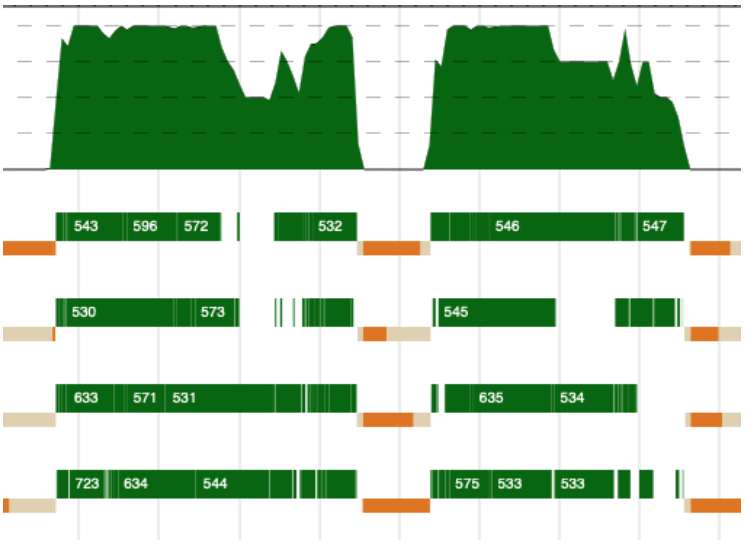


FIGURE 4.10: Threadscope Image of Zoomed Fraction of 10 nanoseconds. Upper green area shows the amount of core used during that fraction of time. The lower are where it shows four separated green bars describe the behavior on each core. The number inside the green bar show the amount of threads running on that core at that moment. Finally orange bars are GC time.

because it requires to maintain the set of connected components in memory throughout the execution of the program or at least until we can output the calculated WCC if we reach to the last *Filter* and we know that this WCC cannot be enlarged anymore. In order to verify this, we measure memory allocation with *eventlog2html* [41] which converts generated profiling memory eventlog files into graphical HTML representation.

As we can see in Figure 4.11, DP-WCC-Haskell does efficient work on allocating memory since we are not using more than 57 MB of memory during the whole execution of the program. On the other hand, if we analyze how the memory is allocated during the execution of the program, it can also be appreciated that most of the memory is allocated at the beginning of the program and steadily decrease over time, with a small peak at the end that does not overpass even half of the initial peak of 57 MB. The explanation for this behavior is quite straightforward because, in the beginning, we are reading from the file and transforming a `ByteString` buffer to `(Int, Int)` edges. This is seen in the image in which the dark blue that is on top of the area is `ByteString` allocation. Light blue is allocation of `Maybe a` type which is the type that is returned by the *Channels* because it can contain a value or not. Data value `Nothing` is indicating end of the *Channel*. Another important

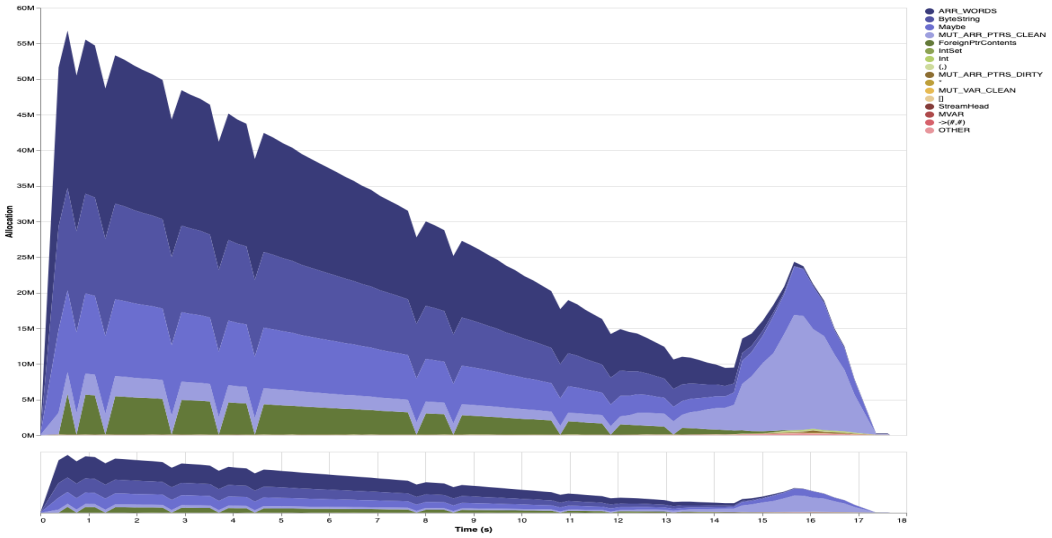


FIGURE 4.11: This plot is showing the accumulated memory allocation size of each Haskell Data Type throughout the execution of the program. The dark blue area shows the `ARR_WORDS` data type which is `String` values. There are many of them because all that it comes from a file is in `String` format and need to be converted to the proper Data type. Rest of the light blue areas belong to `ByteString` which is the format treated in the input file as well, and `Maybe` type which is the type of data transfer between channels.

aspect is the green area which represents `IntSet` allocation, which in the case of our program is the data structure that we use to gather the set of vertices that represents a WCC. This means that the amount of memory used for gathering the WCC itself is minimum, and it is decreasing over time, which is another empirical indication that we are incrementally releasing results to the user. It can be seen as well that as long the green area reduces the lighter blue (`MUT_ARR_PTRS_CLEAN` [42]) increases at the same time indicating that the computations for the output (releasing results) is taking place. Finally, according to what we have stated above, we can answer the question [Q3], showing that not only the memory management was efficient, but at the same time, the memory was not leaking or increasing across the running execution program.

The empirical evaluation of the DP-WCC-Haskell implementation to compute weakly connected components of a graph, evidence suitability, and robustness to provide a Dynamic Pipeline Framework in that language. Measuring using

metrics reveals some advantageous capability of $\mathsf{DP}_{\mathsf{WCC}}$ implementation to deliver incremental results compared with default containers library implementation. Regarding the main aspects where DPP is strong, i.e., pipeline parallelism and time processing, the $\mathsf{DP}_{\mathsf{WCC}}$ performance shows that Haskell can deal with the requirements for the WCC problem without penalizing neither execution time nor memory allocation. In particular, the $\mathsf{DP}_{\mathsf{WCC}}$ implementation outperforms in those cases where the topology of the graph is sparse and where the number of vertices in the largest WCC is not big enough. To conclude, the proof of concept has gathered enough evidence to show that the implementation of Dynamic Pipeline in Haskell Programming Language is feasible. This fact opens a wide range of algorithms to be explored using the Dynamic Pipeline Paradigm, supported by purely functional programming language.

4.3 Chapter Summary

In this chapter, we have presented a proof of concept that allows us to assess the feasibility of implementing DPP using Haskell. The obtained results gave us insights about how to proceed for implementing a first version of a DPF using (parallel) Haskell and, afterward, to implement an algorithm for enumerating incrementally the bitriangles of a bipartite graph based on the DPP.

Chapter 5

Dynamic Pipeline Framework in Haskell

The design and implementation of Haskell Dynamic Pipeline Framework (DPF-Haskell) is a fundamental piece of the present work. A Dynamic Pipeline Framework written in Haskell Programming Language which allow Haskell users to implement any suitable algorithm for Dynamic Pipeline Paradigm. During the process of conducting this research, we have implemented DPF-Haskell [16] and publishes it into The Haskell Package Repository [17]. In this chapter, we describe the design and implementation details of DPF-Haskell.

5.1 Framework Design

5.1.1 Background

A suitable framework should provide the user the right level of abstraction that removes and hides underlying complexity, allowing the developer to focus on the problem that needs to be solved. There are several design approximations to implement a framework: i) *Configuration Based* where the user only focuses on completing a specific configuration either on a file or a database or both. Once this configuration is completed, the user provides it to the framework's runtime system in order to execute the program. An example of this could be WordPress [43], ii) *Convention over Configuration (CoC)* where the user writes his code and definition following certain patterns in naming or source

code location. Using the source code and content-specific information, the framework interprets the execution flow that needs to be executed. This technique has been deeply explored in the last 10 years. One of the first framework that introduce this design paradigm was Ruby on Rails [44]. Other examples are Spring Boot and Cake PHP for example [45, 46], iii) *Application Programing Interface (API)* where the framework or library provides a certain amount of functionality implemented in terms of functions or interfaces, and the user needs to compose those functions or implement those abstractions to achieve the desired results. This has been the traditional design paradigm for building any library or framework, and finally iv) *Domain-specific Language (DSL)* [47] where the framework or library provides a new language that represents the domain problem, encoding the solution in terms of that DSL language. An example of this type of design is Hibernate Query Language [48] .

There exists two types of Domain-specific Languages [49]: External Domain-specific Language (DSL) and Embedded Domain-specific Language (EDSL). The purpose of DSL, is to create a completely new language with its own semantic, syntax, and interpreter. DSLs are not general-purpose languages, because as their name indicates, they are domain-specific. EDSL are syntactically embedded in the host language of the library, and the user writes in that host language, but restricted by the EDSL abstractions. DPF-Haskell follows a EDSL approach taking advantage of the strong type Haskell system providing correctness at type-level [50].

5.1.2 Architectural Design

In this section we focus on the architectural design of the DPF-Haskell using a EDSL approach. We have built a framework that contains three important components: DSL, IDL and RS.

In Figure 5.1 we can appreciate the different components mentioned before that are the grey boxes.

DSL The user interacts with the DSL component where defines how the DPP flow should be. Defining the flow consists on to provide a type-level specification about the channels that communicate each stage of the pipeline,

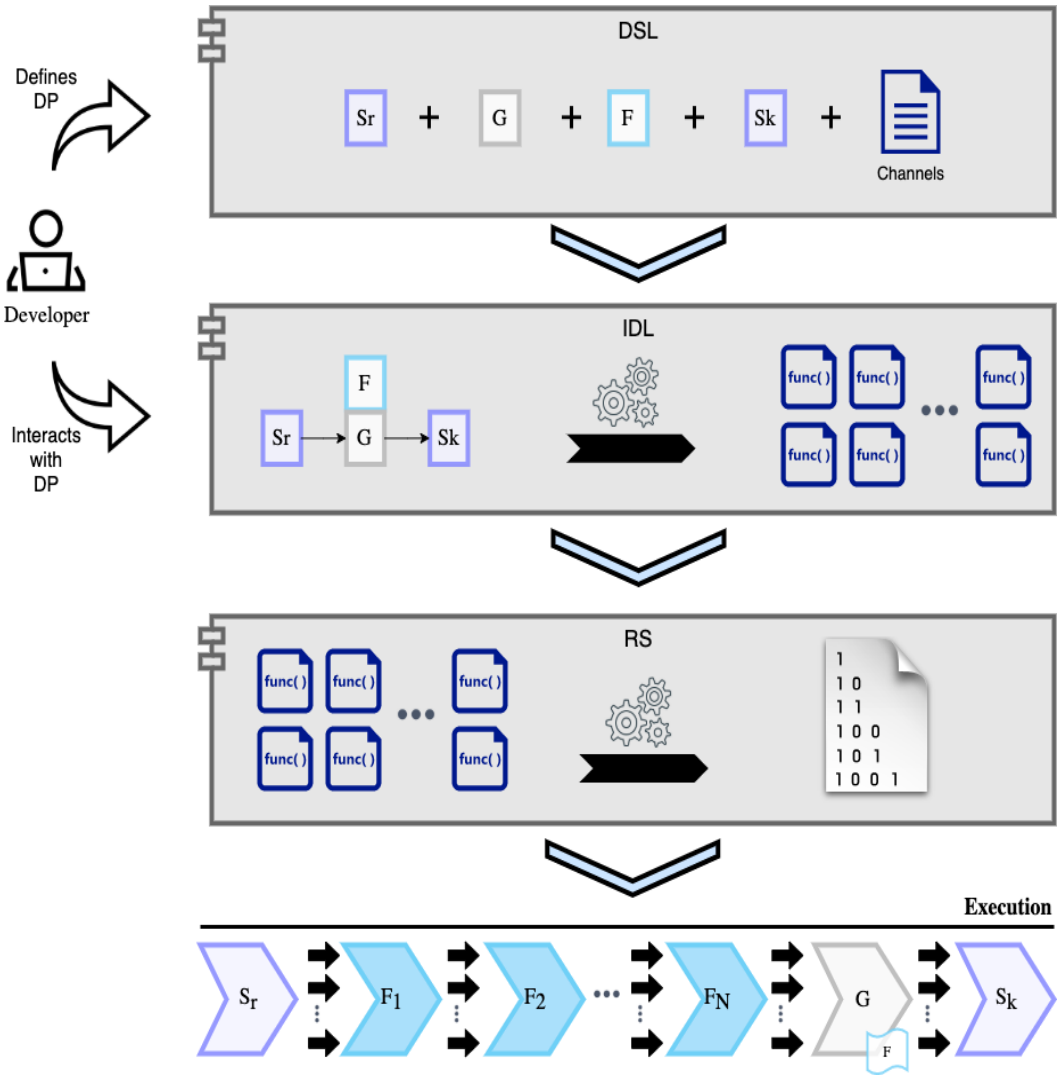


FIGURE 5.1: This diagram shows the architectural design of DPF-Haskell. DPF-Haskell is a DSL which is built on three main components: DSL, IDL and RS. In the DSL we can see how the user can compose the main stages of the DPP. IDL is showing how the frameworks is helping the user to transform that definition into real function or computations. Finally RS execute all that definition plus functions. Execution layer indicates an example of a DPP running after being executed.

as well as the data types those channels carry. For example, in the case of chapter 4 that we develop the WCC algorithm, the user knows stages Sr, G, and Sk need to be connected with two channels. One of those channels is carrying the edges – `Edge` data type – and the other the accumulated connected components – `ConnectedComp` data type –.

IDL Based on the definition provided in the DSL, the user interacts with the IDL to build the functions with the algorithms needed for each stage: Sr, G, Sk, F, and actors.

RS RS is fed with the DPP definition and the functions implementations to finally execute the program.

5.2 Implementation

In this section, we describe the implementation details of each architectural layer: DSL, IDL and RS. As we have explained in section 1.3, this library was published on Hackage [16], the source code is open and can be found on this Github Repository [51].

5.2.1 DSL Grammar

In order to provide correctness verification at compilation level, we define a Context-Free Grammar (CFG) that generates a DPP DSL language. CFG enables the user to define a DPP at type-level.

Definition 5.1 (DSL CFG)**.** Lets $G_{dsl} = (N, \Sigma, DB, P)$ be a Context-Free Grammar, such that N is the set of non-terminal symbols, Σ the set of terminal symbols, $DP \in N$ is the start symbol and P are the generation rules. Figure 5.2 shows the formal definition of the grammar.

$$N = \{DP, S_r, S_k, G, F_b, CH, CH_s\},$$
$$\Sigma = \{\texttt{Source}, \texttt{Generator}, \texttt{Sink}, \texttt{FeedbackChannel}, \texttt{Type}, \texttt{Eof}, \texttt{:=>}, \texttt{:<+>}\},$$

$$\begin{aligned}
P = \{ & \\
DP &\rightarrow S_r \texttt{ :=> } G \texttt{ :=> } S_k \mid S_r \texttt{ :=> } G \texttt{ :=> } F_b \texttt{ :=> } S_k, \\
S_r &\rightarrow \texttt{Source } CH_s, \\
G &\rightarrow \texttt{Generator } CH_s, \\
S_k &\rightarrow \texttt{Sink}, \\
F_b &\rightarrow \texttt{FeedbackChannel}CH, \\
CH_s &\rightarrow \texttt{Channel } CH, \\
CH &\rightarrow \texttt{Type } \texttt{:<+>} \ CH \mid \texttt{Eof}\}
\end{aligned}$$

FIGURE 5.2: This is the Context-Free Grammar defined for the DSL. In the first box we can see N which is the set of non-terminals symbols of the Grammar. Σ which is the set of the terminal symbols and P the production rules of the grammar.

For encoding G_{dsl} on the Haskell, we use an *Index type* [52] to keep track, at type-level, of the extra information required by the DPP definition such as channels and data types the channels carry.

```
1 data Source (a :: Type)
2 data Generator (a :: Type)
3 data Sink
4 data Eof
5 data Channel (a :: Type)
6 data FeedbackChannel (a :: Type)
```

SOURCE CODE 5.1: This code is showing most of the data types that represent the same terminal symbols $\Sigma \in G_{dsl}$. Those types that are indexed by another kind `Type`, allows to store information at type-level needed for interpret the DSL

In Source Code 5.1, there is an *Index type* for each element of Σ encoded in Haskell Types. The highlighted lines in Source Code 5.1 shows the terminal symbols Σ that are not indexed, because neither `Sink` nor `Eof` are carrying

extra type-level information. In the case of `Sink`, since it is the last stage that does not connect further with any other stage, we do not need to indicate any channel information. `Eof` it is just a terminal type to disambiguate the `Channel (a :: Type)` subtree for the full parser tree. `Channel` can carry any type because it needs to be polymorphic to support a different number of channels and data types.

```
1    data chann1 :<+> chann2 = chann1 :<+> chann2
2    deriving (Typeable, Eq, Show, Functor, Traversable, Foldable, Bounded)
3    infixr 5 :<+>
4
5    data a :=> b = a :=> b
6    deriving (Typeable, Eq, Show, Functor, Traversable, Foldable, Bounded)
7    infixr 5 :=>
```

SOURCE CODE 5.2: Special terminal symbols $\{$`:<+>`, `:=>`$\} \in \Sigma$. This terminal symbols allows to index two types in order to combine several of them and build a chain of stages (`:=>`) and a set of channels (`:<+>`).

There are two important terminal symbols in Σ: `:=>` and `:<+>`. In Source Code 5.2, the definition shows how `:=>` and `:<+>` can combine 2 (two) types. The propose of writing `:=>` and `:<+>` as types is to have a syntactic sugar type combinator for writing the DSL according to the CFG. Apart from that, they are different because two distinguishable terminal symbols Σ are needed to separate the encoding of the pipeline stage (Sr, G, Sk) from the encoding of channel composition in the same stage, as we can appreciate in Definition 5.1.

Now, we can start defining our pipelines at type-level. For example, if we want to generate a DPP that eliminates duplicated elements in a stream, we know that we only need one channel connecting the stages that carries out the type of the element, in this case, `Int` (see Source Code 5.3).

```
1  type DPExample = Source (Channel (Int :<+> Eof))
2                 :=> Generator (Channel (Int :<+> Eof))
3                 :=> Sink
```

SOURCE CODE 5.3: This example shows the DSL encoding in DPP of repeated elements problems

5.2.2 DSL Validation

The language generated by the grammar needs to be validated to avoid errors or provide an incorrect DPP definition. Fortunately, Haskell provides several Type-level techniques [53] which allows to verify properties of programs before running them, preventing the users to introduce bugs, reducing errors. This verification done by the compiler establish a Curry-Howard Isomorphism [50], i.e. *Propositions as Types - Programs as Proof.* It is important to remark here that Haskell is not a theorem prover System like Coq [54], but some verifications, as we present in this work, can be done with GHC to ensure correctness on programs. Although Haskell provides tools to build advanced type-level verifications, all these techniques require the addition of *Haskell Language Extensions.*

Once we have the encoded DPP problem in the DSL grammar – see subsection 5.2.1 –, we can proceed on validating that encoded grammar. The implementation of the validation of the DSL CFG at type-level, has been done using *Associated Type Families* [55].

```haskell
type family And (a :: Bool) (b :: Bool) :: Bool where
    And 'True 'True = 'True
    And a b         = 'False


type family IsDP (dpDefinition :: k) :: Bool where
    IsDP (Source (Channel inToGen) :=> Generator (Channel genToOut) :=>
    ↪ Sink)
        = And (IsDP (Source (Channel inToGen))) (IsDP (Generator (Channel
        ↪ genToOut)))
    IsDP ( Source (Channel inToGen) :=> Generator (Channel genToOut) :=>
    ↪ FeedbackChannel toSource :=> Sink)
        = And (IsDP (Source (Channel inToGen))) (IsDP (Generator (Channel
        ↪ genToOut)))
    IsDP (Source (Channel (a :<+> more)))
        = IsDP (Source (Channel more))
    IsDP (Source (Channel Eof))                = 'True
    IsDP (Generator (Channel (a :<+> more)))  = IsDP (Generator (Channel
    ↪ more))
    IsDP (Generator (Channel a))               = 'True
    IsDP x                                     = 'False

type family ValidDP (a :: Bool) :: Constraint where
  ValidDP 'True = ()
  ValidDP 'False = TypeError
                   ( 'Text "Invalid Semantic for Building DP Program"
                     ':$$: 'Text "Language Grammar:"
                     ':$$: 'Text "DP       -> Source CHANS :=> Generator
                     ↪ CHANS :=> Sink"
                     ':$$: 'Text "DP       -> Source CHANS :=> Generator
                     ↪ CHANS :=> FEEDBACK :=> Sink"
                     ':$$: 'Text "CHANS    -> Channel CH"
                     ':$$: 'Text "FEEDBACK -> FeedbackChannel CH"
                     ':$$: 'Text "CH       -> Type :<+> CH | Eof"
                     ':$$: 'Text "Example: 'Source (Channel (Int :<+>
                     ↪ Int)) :=> Generator (Channel (Int :<+> Int)) :=>
                     ↪ Sink'"
                   )
```

SOURCE CODE 5.4: Type Families `And`, `IsDP` and `ValidDP` which allows to perform a type-level validation over a DSL CFG definition.

In Source Code 5.4, there are 3(three) Type families that helps to validate the DSL CFG. `IsDP` associated type family is checking the production rules P of the grammar defined in Figure 5.2, returning a promoted data type [56] (not a boolean value) `'True` in case the production rule matches all the generated

language, or `'False` otherwise. `ValidDP` is taking the result of `IsDP` type application, associating `'True` promoted boolean type to empty `()` constraint. An empty constraint is an indication of nothing to be restricted, meaning that if `ValidDP` is used as a constraint, and it is fully applied to `()`, it will give the compiler the evidence that there is no error at type-level. `ValidDP` is also associating `'False` to a custom `TypeError` which will appear at compilation time if the DPP DSL definition fully applies to that.

```
1  mkDP :: forall dpDefinition filterState filterParam st.
2      ( ValidDP (IsDP dpDefinition)
3      , DPConstraint dpDefinition filterState st filterParam)
4   => Stage (WithSource dpDefinition (DP st))
5   -> GeneratorStage dpDefinition filterState filterParam st
6   -> Stage (WithSink dpDefinition (DP st))
7   -> DP st ()
8  mkDP = ...
9
10 someFunc = mkDP @DPExample ...
```

Source Code 5.5: Definition of `mkDP` function of the Framework which uses type-level validation of the grammar `ValidDP (IsValid Type)`. Last line of the code is showing that using that function will compile-time check the definition of `DPExample` type.

5.2.3 Interpreter of DSL (IDL)

IDL component takes the DPP definition made on with DSL component to interpret and generate the function definitions that the user needs to fill in for solving a specific problem. In section 2.2, we have described what the user needs to provide in a DPP algorithm: Sr, G, Sk, and the F with the non-empty set of Actors. The IDL generates the function definitions with an empty implementation to be completed by the user, ensuring that those functions will give "Proof" – in terms of Curry-Howard Correspondence [50] – of the "Propositions" defined on the DSL.

Similar techniques that we used on subsection 5.2.2 are also used here. On the first hand, we use *Type-level Defunctionalization* [57, 58] to let the compiler generates the signatures of the required functions. On the other hand, we use *Term-level Defunctionalization* to interpret those functions. Moreover,

Indexed Types [52] and *Heterogeneous List* [59] are used to keep track of the dynamic number and polymorphic types of the functions parameters.

```
1  withSource :: forall (dpDefinition :: Type) st. WithSource dpDefinition (DP
   ↪  st)
2              -> Stage (WithSource dpDefinition (DP st))
3  withSource = mkStage' @(WithSource dpDefinition (DP st))
4
5  withGenerator :: forall (dpDefinition :: Type) (filter :: Type) st.
   ↪  WithGenerator dpDefinition filter (DP st)
6                -> Stage (WithGenerator dpDefinition filter (DP st))
7  withGenerator = mkStage' @(WithGenerator dpDefinition filter (DP st))
8
9  withSink :: forall (dpDefinition :: Type) st. WithSink dpDefinition (DP st)
10             -> Stage (WithSink dpDefinition (DP st))
11 withSink = mkStage' @(WithSink dpDefinition (DP st))
```

SOURCE CODE 5.6: This code is showing the different interpreters combinators to help the user to generate the functions of the principal stages of DPP

In Source Code 5.6 we can appreciate the different combinators of the IDL that helps the user of the framework to interpret the DSL to generate the function definitions. `Stage` data type will be cover in Source Code 5.8, but it is a wrapper type of a pipeline stage – minimal unit of execution –, containing the function to be executed – here is the use *Term-level Defunctionalization* –. `withSource`, `withGenerator`, and `withSink` are aliases of the function `mkStage'` which is the combinator that is applying the *Associated Type* related to that stage. For example `withSource`, is equivalent to `mkStage' @(WithSource dpDefinition (DP st))`. For each *Associated Type Family* defintion, there exist an equivalent term-level definition: `WithSource` type with `withSource` term , `WithGenerator` type with `withGenerator` term, and `WithSink` type with `withSink` term – notice the capital case letter "W" indicating the type and not the term –.

```haskell
type family WithSource (dpDefinition :: Type) (monadicAction :: Type ->
↪  Type) :: Type where
  WithSource (Source (Channel inToGen) :=> Generator (Channel genToOut) :=>
  ↪  Sink) monadicAction
      = WithSource (ChanIn inToGen) monadicAction
  WithSource (Source (Channel inToGen) :=> Generator (Channel genToOut) :=>
  ↪  FeedbackChannel toSource :=> Sink) monadicAction
      = WithSource (ChanOutIn toSource inToGen) monadicAction
  WithSource (ChanIn (dpDefinition :<+> more)) monadicAction
      = WriteChannel dpDefinition -> WithSource (ChanIn more) monadicAction
  WithSource (ChanIn Eof) monadicAction
      = monadicAction ()
  WithSource (ChanOutIn (dpDefinition :<+> more) ins) monadicAction
      = ReadChannel dpDefinition -> WithSource (ChanOutIn more ins)
      ↪  monadicAction
  WithSource (ChanOutIn Eof ins) monadicAction
      = WithSource (ChanIn ins) monadicAction
  WithSource dpDefinition _
      = TypeError
          ( 'Text "Invalid Semantic for Source Stage"
            ':$$: 'Text "in the DP Definition '"
            ':<>: 'ShowType dpDefinition
            ':<>: 'Text "'"
            ':$$: 'Text "Language Grammar:"
            ':$$: 'Text "DP       -> Source CHANS :=> Generator CHANS :=>
            ↪  Sink"
            ':$$: 'Text "DP       -> Source CHANS :=> Generator CHANS :=>
            ↪  FEEDBACK :=> Sink"
            ':$$: 'Text "CHANS    -> Channel CH"
            ':$$: 'Text "FEEDBACK -> FeedbackChannel CH"
            ':$$: 'Text "CH       -> Type :<+> CH | Eof"
            ':$$: 'Text "Example: 'Source (Channel (Int :<+> Int)) :=>
            ↪  Generator (Channel (Int :<+> Int)) :=> Sink'"
          )
```

SOURCE CODE 5.7: An example of the Associated Type Family `WithSource` that allows to implement *Type-level Defunctionalization* technique that will be the Type-level verification of the term `withSource`

In Source Code 5.7, in the highlighted lines, it can be seen how *Type-level Defunctionalization* is being expanded in a signature function definition with the form `WriteChannel a -> ReadChannel b -> ... -> monadicAction ()` depending on DPP language definition.

```haskell
data Stage a where
  Stage :: Proxy a -> a -> Stage a

mkStage' :: forall a. a -> Stage a
mkStage' = Stage (Proxy @a)
```

SOURCE CODE 5.8: `Stage` data type for implementing *Term-level Defunctionalization* providing evidence to the Type-Level Associated types

In Source Code 5.8, `Stage` data type uses a `Proxy` phantom type. This phantom type allow `Stage` to index the type definition generated by `a`. For example, in Source Code 5.6, when `withSource` interpreter is applied to `WithSource dpDefinition`, the compiler is provided with `dpDefinition` DSL type, it expands the function signature belonging to that DPP definition inside the `Stage`.

Generator and Filter According to DPP definition in section 2.2, G has a F template in order to know how to dynamically interpose a new F during the runtime execution of the program. Let's first study F Data Type in the context of the framework.

```haskell
newtype Actor dpDefinition filterState filterParam monadicAction =
    Actor {  unActor :: MonadState filterState monadicAction => Stage
    ↪  (WithFilter dpDefinition filterParam monadicAction) }

newtype Filter dpDefinition filterState filterParam st =
    Filter { unFilter :: NonEmpty (Actor dpDefinition filterState
    ↪  filterParam (StateT filterState (DP st))) }
    deriving Generic
```

SOURCE CODE 5.9: This code shows the definition of the `Filter` data type which contains a non-empty set of `Actor`. The `Actor` data type is an `Stage` in the Context of the `MonadState` to allow keeping a local memory in the execution context of the filter.

In Source Code 5.9 the definition of the `Filter` data type contains a non-empty set of `Actor`. An `Actor` is a `Stage`, because an actor is the minimal unit of execution of a filter. A `Filter` has a `NonEmpty Actor` – Non-empty List – because a filter is built by a sequence of actors calls. Moreover, `Actor` Stage is

defunctionalized with `WithFilter` *Associated Type Family*. `Filter` runs in an explicit `StateT` monadic context. This is because the F instance should have an state, according to DPP definition in section 2.2. For example, in the case of $\mathsf{DP}_{\mathsf{WCC}}$, as we have seen in chapter 4, F keeps an updated list of connected components that updates as long as it receives more edges that are connected with the current list of vertices. `Actor` data type – see Source Code 5.9 –, is constrained by `MonadState` which is in the same execution context of the whole `NonEmpty Actor` list of the `Filter`. This means the `StateT` is executed for each `Actor` of that filter, sharing the same state between them.

```
1  mkFilter :: forall dpDefinition filterState filterParam st. WithFilter
   ↪  dpDefinition filterParam (StateT filterState (DP st))
2          -> Filter dpDefinition filterState filterParam st
3  mkFilter = Filter . single
4
5  single :: forall dpDefinition filterState filterParam st. WithFilter
   ↪  dpDefinition filterParam (StateT filterState (DP st))
6         -> NonEmpty (Actor dpDefinition filterState filterParam (StateT
          ↪  filterState (DP st)))
7  single = one . actor
8
9  actor :: forall dpDefinition filterState filterParam st. WithFilter
   ↪  dpDefinition filterParam (StateT filterState (DP st))
10       -> Actor dpDefinition filterState filterParam (StateT filterState (DP
         ↪  st))
11 actor = Actor . mkStage' @(WithFilter dpDefinition filterParam (StateT
   ↪  filterState (DP st)))
12
13 (|>>>) :: forall dpDefinition filterState filterParam st. Actor
   ↪  dpDefinition filterState filterParam (StateT filterState (DP st))
14        -> Filter dpDefinition filterState filterParam st
15        -> Filter dpDefinition filterState filterParam st
16 (|>>>) a f = f & _Wrapped' %~ (a <|)
17 infixr 5 |>>>
18
19 (|>>) :: forall dpDefinition filterState filterParam st. Actor dpDefinition
   ↪  filterState filterParam (StateT filterState (DP st))
20       -> Actor dpDefinition filterState filterParam (StateT filterState (DP
         ↪  st))
21       -> Filter dpDefinition filterState filterParam st
22 (|>>) a1 a2 = Filter (a1 <|one a2)
23 infixr 5 |>>
```

SOURCE CODE 5.10: Combinators and small constructor to enable building actors and filter.

Finally, in Source Code 5.10, some combinators and smart constructors are provided in the framework to enable the construction of `Filter` and `Actor`. `mkFilter` is a smart constructor for `Filter` Data Constructor. `single` wraps one actor inside a `Filter`. `actor` is a smart constructor for `Actor` Data Constructor. `(|>>>)` is an appending combinator of an `Actor` to a `Filter`. `(|>>>)` also ensures actor execution order, i.e. the latest actor added is the latest to be executed.

```
1    data GeneratorStage dpDefinition filterState filterParam st =
     ↪  GeneratorStage
2    { _gsGenerator      :: Stage (WithGenerator dpDefinition (Filter
     ↪  dpDefinition filterState filterParam st) (DP st))
3    , _gsFilterTemplate :: Filter dpDefinition filterState filterParam st
4    }
```

SOURCE CODE 5.11: `Generator` Data type which contains the `Stage` code of the generator itself, and the `Filter` template that it can be spawned by the `Generator`.

In Source Code 5.11, G contains a F template and its own stage behavior. `Generator` data type has a field with the `Filter` template that could be spawned by the algorithm defined by the user according to the data received from its input channels. `Generator` has also another field with the behavior of the G – a `Stage` –.

5.2.4 Runtime System (RS)

The RS can be divided into two parts: the mechanism to generate stages dynamically in runtime, and the execution entry point of the DPP. Regarding execution entry point, all the stages that we have seen in previous sections are the pieces needed to build an executable `DP st a` monad. This executable monad has an existential type similar to `ST` monad to not escape out from the context on different stages. Once the dynamic pipeline starts to execute, the core of the framework dynamically generates stages between G and previous stages, according to the user definition, i.e. an *anamorphism* [60] that creates F instances until some condition is met.

```
1  unfoldF :: forall dpDefinition readElem st filterState filterParam l.
   ↪  SpawnFilterConstraint dpDefinition readElem st filterState filterParam
   ↪  l
2          => UnFoldFilter dpDefinition readElem st filterState filterParam l
3          -> DP st (HList l)
4  unfoldF = loopSpawn
5
6  where
7    loopSpawn uf@UnFoldFilter{..} =
8      maybe (pure _ufRsChannels) (loopSpawn <=< doOnElem uf) =<< DP (pull
       ↪  _ufReadChannel)
9
10   doOnElem uf@UnFoldFilter{..} elem' = do
11     _ufOnElem elem'
12     if _ufSpawnIf elem'
13      then do
14        (reads', writes' :: HList l3) <- getFilterChannels <$> DP
          ↪  (makeChansF @(ChansFilter dpDefinition))
15        let hlist = elem' .*. _ufReadChannel .*. (_ufRsChannels
          ↪  `hAppendList` writes')
16        void $ runFilter _ufFilter (_ufInitState elem') hlist
          ↪  (_ufReadChannel .*. (_ufRsChannels `hAppendList` writes'))
17        return $ uf { _ufReadChannel = hHead reads', _ufRsChannels = hTail
          ↪  reads' }
18      else return uf
```

Source Code 5.12: `unfolF` is the *anamorphism* combinator to spawn new `Filter` types between the `Generator` and previous stages.

In Source Code 5.12, it is presented how is the *anamorphism* mechanims that generates dynamic stages between G and the previous stages. That *anamorphism* is implemented with the function `unfoldF`. That function receives an `UnFoldFilter` Data type, which contains the recipe for controlling that unfold recursive call. In line 12, `_ufSpawnIf` field of `UnFoldFilter`, indicates when to stop the recursion. Inside the conditional, in line 14, new channels are created for the new filter to be spawned. Those new channels connect the new filter with the previous stages and with `Generator`. After that, in line 16 `runFilter` starts the monadic computation, spawning the filter stage with its actors. Finally, the new list of channels are returned for the next recursive step to allow further channel connections.

```haskell
mkUnfoldFilter :: (readElem -> Bool)
    -> (readElem -> DP st ())
    -> Filter dpDefinition filterState filterParam st
    -> (readElem -> filterState)
    -> ReadChannel readElem
    -> HList l
    -> UnFoldFilter dpDefinition readElem st filterState filterParam l


mkUnfoldFilterForAll' :: (readElem -> DP st ())
                      -> Filter dpDefinition filterState filterParam st
                      -> (readElem -> filterState)
                      -> ReadChannel readElem
                      -> HList l
                      -> UnFoldFilter dpDefinition readElem st filterState
                      ↪  filterParam l

mkUnfoldFilterForAll :: Filter dpDefinition filterState filterParam st
                      -> (readElem -> filterState)
                      -> ReadChannel readElem
                      -> HList l
                      -> UnFoldFilter dpDefinition readElem st filterState
                      ↪  filterParam l
```

SOURCE CODE 5.13: Combinators for building `UnfoldFilter` types indicating the type of the `unfold` that the user want to achieve.

Several smart constructors are also provided for building `UnfoldFilter` Data Type. In Source Code 5.13 the first combinator is the default smart constructor. i) First field `(readElem -> Bool)` indicate if the a new filter should be spawn or not. ii) Second field `(readElem -> DP st ())` is a monadic optional computation to do when received a new element, for example logging. iii) Third field `Filter` data type to be spawned. iv) Fourth field `(readElem -> filterState)` is initialization of the `Filter` State. v) Fifth field `(ReadChannel readElem)` that feeds the filter instance. vi) Last field is the *Heterogeneous List* with the rest of the channels to connect with other stages. . The combinator `mkUnfoldFilterForAll` is an smart constructor of `UnfoldFilter` that allows to spawn a new filter for each element received in the G.

5.3 Libraries and Tools

5.3.1 Parallelization

One of the most important components of the implementation is the selection of concurrency libraries to support an intensive parallelization workload. Parallelization techniques and tools have been intensively studied and implemented in Haskell [34]. Indeed, it is well known that green threads and sparks allow spawning thousands to millions of parallel computations. These parallel computations do not penalize performance when compare with Operative System (OS) level threading [33]. A straightforward assumption to achieve here, is to use `monad-par` library [34, 61]. Nevertheless, in this work, we have discarded the use of sparks [62] because we can achieve the level of required parallelism spawning green threads only. The next obvious choice is to use `forkIO :: IO () -> IO ThreadId` from `base` library [63]. However, that would imply handling all the threads lifecycles and errors programmatically without any abstraction to facilitate that complex task. Therefore, we choose `async` library [64] which enables to spawn asynchronous computations [33] on Haskell using green threads, and at the same time, it provides useful combinators to managing thread terminations and errors.

5.3.2 Channels

We have several techniques to our disposal to communicate between threads or sparks in Haskell like `MVar` or concurrent safe mechanisms like Software Transactional Memory (STM) [65]. At the same time, in Haskell library ecosystem, we dispose of `Channels` abstractions based on both mentioned communication techniques. In that sense, for conducting the communication between dynamic stages and data flowing in a DPP, we have selected `unagi-chan` library [66] which provides the following advantages to our solution: Firstly, `MVar` channel without using STM reducing overhead. STM is not required in a DPP because one specific stage which is running in a separated thread, can only access to its `I/O` channels for reading/writing accordingly, and those operations are not concurrently shared by other threads (stages) for the same channels. Second, non-blocking channels. `unagi-chan` library contains blocking and non-blocking channels for reading. This aspect is key to gain speed

up on the implementation. Third, the library is optimized for $x86$ architectures with use of low-level `fetch-and-add` instructions. Finally, `unagi-chan` is $100x$ faster [67] on Benchmarking compare with STM and default base `Chan` implementations.

5.4 Chapter Summary

In this chapter, we have described with a great level of detail how Haskell Dynamic Pipeline Framework has been conceived from a Design point of view, as well as all the Haskell data types and language techniques that we use for that implementation. At the end of the chapter, we have mentioned external libraries used for the runtime system and the reason for their choice.

Chapter 6

An algorithm for incrementally enumerating Bitriangles using DP

In this chapter, we first give the foundation to define and implement an algorithm for incrementally enumerating bitriangles in a large bipartite graph. Then, we introduce an algorithm based on the DPP. In particular, we present the pseudo-code of the stages of the proposed $\mathsf{DP}_{\mathsf{BT}}$. Finally, we provide a proof of correctness of our proposal and the main details of the implementation of the $\mathsf{DP}_{\mathsf{BT}}$ using the DP-BT-Haskell.

6.1 Preliminaries Definitions

In order to understand how the algorithm works, we need to provide some basic definitions, and define other small structural units that we use for giving a solution to the problem. Let's enumerate all those definitions in the following paragraphs.

Definition 6.1 (Bipartite Graph)**.** A Bipartite Graph (BG) is an undirected graph $G = (V, E)$ such that $V = (U \cup L)$, $U \cap L = \emptyset$ and $E \subseteq U \times L$.

Additionally, without loss of generality, we assume that $U \subseteq \mathbb{N}$ and $L \subseteq \mathbb{N}$. Consequently, $(U, <)$ and $(L, <)$ are strict total orders. We can see an example of a BG in Figure 6.1.

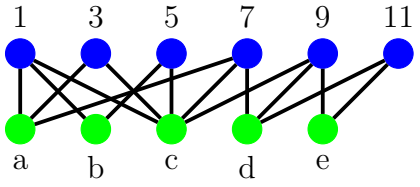
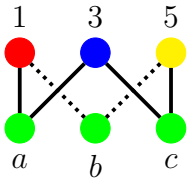


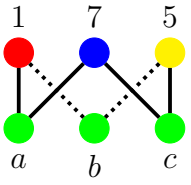
FIGURE 6.1: A bipartite graph in which five bitriangles can be enumerated

Identifying the different bitriangles in the graph in Figure 6.1 can be done manually. In what follows, we use this graph to illustrate the new definitions. This small example allows us to realize that identifying and listing bitriangles in large graphs is a challenging task.

Definition 6.2 (Bitriangle)**.** Let the triples $\mu = (u_1, u_2, u_3)$ and $\ell = (l_1, l_2, l_3)$ on U and L, respectively, i.e. $\{u_1, u_2, u_3\} \subseteq U$, $\{l_1, l_2, l_3\} \subseteq L$. The 6-cycle $(u_1, l_1, u_2, l_3, u_3, l_2, u_1)$ is a *Bitriangle (BT)* in G, denoted by $BT^{\mu}_{\ell} = BT^{(u_1,u_2,u_3)}_{(l_1,l_2,l_3)}$.



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



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

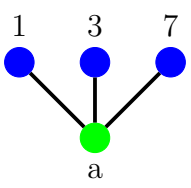
FIGURE 6.2: Bitriangles of the triple (a, b, c) combined with vertices from U $\{1, 3, 5, 7\}$ of Figure 6.1. In this case two bitriangles have been built from the Figure 6.6

A bitriangle $(u_1, l_1, u_2, l_3, u_3, l_2, u_1)$ admits different ways of traversing it depending on the vertex where the traversal starts. This fact gives rise to different feasible permutations of its representation as a 6-cycle. For example in the Figure 6.2 the bitriangle $(1, a, 3, c, 5, b, 1)$, can also be traverse using the permutations $(a, 1, b, 5, c, 3, a)$, $(5, c, 3, a, 1, b, 5)$, ..., and so on. Notice that all these feasible permutations guarantee the bitriangle condition of vertices intercalation from U and L.

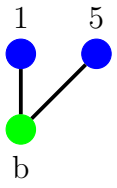
Definition 6.3 (Wedge (WG))**.** A *Wedge (WG)* in G is a triple (u_1, l, u_2), $\{u_1, u_2\} \subseteq U$, $l \in L$ and $\{(u_1, l), (u_2, l)\} \subseteq E$. The vertex l is the middle vertex of the wedge.

With this definition in place, we can now define an aggregated wedge, which is a compressed form of all the wedges having $l \in L$ as a middle vertex.

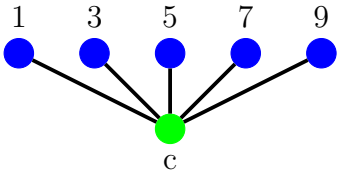
Definition 6.4 (Aggregated Wedge (AW))**.** An *Aggregated Wedge (AW)* is a pair $\langle l, W_l \rangle$, where $l \in L$, $W_l \subseteq U$ and for all $u \in W_l$, the edge $(u, l) \in E$.



(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

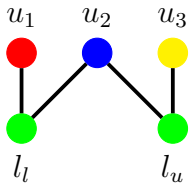


(C) This is the AW $\langle c, \{1, 3, 5, 7, 9\} \rangle$, the compact representation of wedges

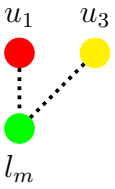
FIGURE 6.3: Aggregated Wedge of nodes a, b, c of Figure 6.1

In Figure 6.3, we are only enumerating the first three lower layer vertices AW. Next, we define a more refined structure that will allow us to enrich the AW in order to lead us to a BT. That intermediate structure is an *aggregated double-wedge*. For defining ADW we first need another structure called *double-wedge*. Intuitively, a *double-wedge*, is similar to a WG and AW but relating two vertices in the lower layer L.

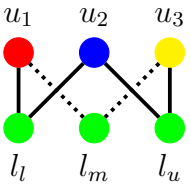
Definition 6.5 (Double-wedge (DW))**.** A *Double-wedge (DW)* in G is a path of length 4 $(u_1, l_1, u_2, l_2, u_3)$ where $\{u_1, u_2, u_3\} \subseteq U$ and $\{l_1, l_2\} \subseteq L$. Vertices l_1 and l_2 are the middle vertices of DW.



(A) A *double-wedge* $(u_1, l_l, u_2, l_u, u_3)$



(B) A connector *wedge* (u_1, l_m, u_3) with middle vertex l_m



(C) A *bitriangle* formed by *double-wedge* $(u_1, l_l, u_2, l_u, u_3)$ and connector *wedge* (u_1, l_m, u_3)

FIGURE 6.4: Example on how to build a bitriangle from a *double-wege* plus a *wedge* connector

In Figure 6.4 the *wedge* (u_1, l_m, u_3), is called the *connector wedge*. This connector wedge together with the *double-wedge* $(u_1, l_l, u_2, l_u, u_3)$ form a a bitriangle.

Definition 6.6 (Aggregated double-wedge (ADW))**.** Let $U_l = \langle I, J, K\rangle$ be a triplet such that $I \subseteq U, J \subseteq U$ and $K \subseteq U$, where I, J and K are disjoint sets. An *Aggregated double-wedge (ADW)* is a pair $\langle (l_1, l_2), U_l\rangle$, where $\{l_1, l_2\} \subseteq L$ and for all $u_i \in I, u_j \in J$ and $u_k \in K$, $\{(u_i, l_1), (u_j, l_1), (u_j, l_2), (u_k, l_2)\} \in E$.

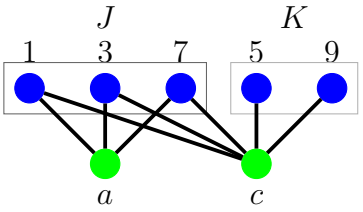


FIGURE 6.5: Aggregated double-wedge $\langle (a, c), \langle \emptyset, \{1, 3, 7\}, \{5, 9\}\rangle\rangle$ of Figure 6.1. Upper layer nodes $1, 3, 7$ and $5, 9$ are enclosed in a square indicating the set that they belong to. Remember in Definition 6.6 we are forming three sets $U_l = \langle I, J, K\rangle$. $I = \emptyset$ in this case because they should be disjoint sets and a and c share the same upper layer nodes in J.

In Figure 6.5 we can see the ADW built by a and c where, according to Definition 6.6, $a = l_1$ and $c = l_2$. The last aggregated structure that enable the algorithm to build a BT is the *Aggregated Bitriangle*. Intuitively, this intermediate structure is an aggregation of a ADW with another new AW related with the former structure.

Definition 6.7 (Aggregated wedge BT-Connector (AW-BT))**.** Let $\langle (l_1, l_2), U_l\rangle$ be an ADW, where $\{l_1, l_2\} \subseteq L$ and $U_l = \langle I, J, K\rangle$. An AW $\langle l, W_l\rangle$ is an *Aggregated wedge BT-Connector (AW-BT) if and only if* $l > l_1$ and $l < l_2$ and $W_l \cap (I \cup J) \neq \emptyset$ and $W_l \cap (K \cup J) \neq \emptyset$

Definition 6.8 (Aggregated Bitriangle (ABT))**.** Let $\hat{U}_l = \langle I, J, K\rangle$, such that $I \subseteq U, J \subseteq U, K \subseteq U$. An *Aggregated Bitriangle (ABT)* is a pair $\langle \ell, \hat{U}_l\rangle$, where $\ell = (l_1, l_2, l_3)$ is a triple on L, $l_1 < l_2 < l_3$ and for all $\langle I, J, K\rangle$ and for all $\mu = (u_i, u_j, u_k)$ such that $u_i \in I, u_j \in J, u_k \in K$, $BT_\ell^\mu \in \mathbb{BT}$.

As we can see in Figure 6.6, we have all the structures at our disposal to build a BT. In Figure 6.2, the two possible BT can be extracted from the ABT presented in Figure 6.6 are $(1, a, 3, c, 5, b, 1)$ and $(1, a, 7, c, 5, b, 1)$.

IEBT will allow to enumerate BT according to some locality criteria. In this regards, we next define the *query operators* to be used for that purpose.

Definition 6.9 (Query Operator (QO))**.** a *Query Operator* Q is a value from the *sum type* $\mathcal{P}(U + L) + \mathcal{P}(E)$ producing as a result a possible set of BT that include any of the vertices or edges given in Q.

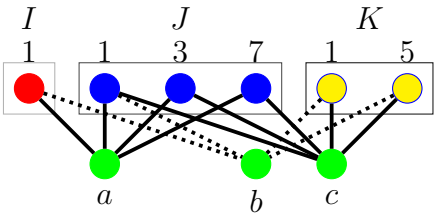


FIGURE 6.6: Aggregated wedge BT-Connector $\langle b, \{1,5\}\rangle$ connects to Aggregated double-wedge $\langle (a,c), \langle \emptyset, \{1,3,7\}, \{5,9\}\rangle\rangle$ of Figure 6.1, constructing the Aggregated Bitriangle $\langle (a,b,c), \langle \{1\}, \{1,3,7\}, \{1,5\}\rangle\rangle$. Note that $\hat{U}_{(a,b,c)} = \{I, J, K\}$ does not contains disjoint sets according to Definition 6.8.

In Table 6.1, we present a summary of the notation used in this chapter.

Notation	**Meaning**
$G = ((U \cup L), E)$	a bipartite graph
n, m	the number of vertices and edges in G, resp.
(u, l)	an edge between vertices u and l
(u_1, l, u_2)	a wedge with middle vertex l
$\langle l, W_l\rangle$	an AW
$\mathbb{AW}$	the set of all the possible AW in G
$(u_1, l_1, u_2, l_2, u_3)$	a DW with middle vertices l_1 and l_2
$\langle (l_1, l_2), U_l\rangle$	an ADW
$\mathbb{DW}$	the set of all the possible ADW in G
dw	Subset of $\mathbb{DW}$, such that $\mathsf{dw} \subseteq \mathbb{DW}$
$\left\langle (l_1, l_2, l_3), \hat{U}_l \right\rangle$	an ABT
$\mathbb{AT}$	the set of all the possible ABT in G
at	Subset of $\mathbb{AT}$, such that $\mathsf{at} \subseteq \mathbb{AT}$
$BT^{(u_1,u_2,u_3)}_{(l_1,l_2,l_3)}$	the BT $u_1, l_1, u_2, l_3, u_3, l_2, u_1$
$\mathbb{BT}$	the set of all the possible BT in G
bt	Subset of $\mathbb{BT}$, such that $bt \subseteq \mathbb{BT}$
$\mathcal{P}$	Parts of set

TABLE 6.1: This table summarizes all the different object definitions that we have detailed in section 6.1. The first column describe the formal term used in the definitions on section 6.1. The second column summarize the meaning of each term

6.2 Algorithm Sketch

The algorithm for enumerating incrementally bitriangles in a large bipartite graph consists in two main phases. During the first phase, the bipartite network is received by the $\mathsf{DP_{BT}}$ as a stream of edges and a graph index structure is created. This graph index is represented by the different structures stored along the filters stages of the $\mathsf{DP_{BT}}$. This first phase is in charge of constructing the compressed structures *double wedge*, *aggregated double-wedge*, and finally *aggregated bitriangles*. The second phase is a querying phase. During the querying phase local queries can be submitted to the $\mathsf{DP_{BT}}$. When a query arrives to the $\mathsf{DP_{BT}}$, the enumeration incremental of bitriangles process –according to the criteria in the submitted queries– is launched. In this phase, enumerated bitriangles are extracted from the different aggregated bitriangles occurring in the graph index, i.e. the aggregated bitriangles stored in filter stages. More concretely, the Dynamic Pipeline Algorithm for Enumerating Bitriangles ($\mathsf{DP_{BT}}$) is defined in terms of the behavior of its four kinds stages: *Source* ($\mathsf{Sr_{BT}}$), *Generator* ($\mathsf{G_{BT}}$), *Sink* ($\mathsf{Sk_{BT}}$), and *Filter*($\mathsf{F_{BT}}$) stages. The algorithm considers a Bitriangle as a convenient composition of three wedges as we can see the example in Figure 1.1a. In order to reduce memory footprint, the algorithm aggregates results, i.e. the set of wedges having the same middle vertex is represented as a pair $\langle l, W_l \rangle$ where l is the middle vertex and W_l is the set of adjacent vertices of l called Aggregated Wedge (AW) (see Definition 6.4). The algorithm first collects AW for every vertex in the L set of the graph. Afterwards it constructs Aggregated double-wedge (ADW) for every pair of distinct vertices, Finally constructs ABT for selected triples of vertices. The following Table 6.2 describes the different channels that are connecting the stages in $\mathsf{DP_{BT}}$.

The setup of $\mathsf{DP_{BT}}$ can be appreciated in Figure 6.7. $\mathsf{Sr_{BT}}$ reads from input stream all $(u, l) \in E$ and transfers to the following stage each (u, l) using C_E. For every (u, l) that arrives to $\mathsf{G_{BT}}$, a new $\mathsf{F_{BT}}$ instance with parameter $l \in L$ is spawn. $\mathsf{F_{BT}}$ contains four actors. First, $\mathsf{actor_1}$ receives from IC_E the edges and builds aggregated wedges, when there are no more edges it downstreams it to it's neighbour $\mathsf{F_{BT}}$ using OC_{W1}. Then $\mathsf{actor_2}$ receives from IC_{W1} aggregated wedges from previous $\mathsf{F_{BT}}$, downstream to next $\mathsf{F_{BT}}$ and $\mathsf{G_{BT}}$ using OC_{W1} and at the same time use its information to see if it can build an aggregated double-wedge. If an aggregated double-wedge could be constructed, it will be stored

Channel	**Meaning**
$C = \langle IC, OC \rangle$	A Channel pair that connects input and output channel
OC	Set of Output Channels
C_E	Channel of $e \in E$
IC_E	Input Channels carrying $e \in E$
OC_E	Output Channels carrying $e \in E$
C_{W_l1}	Channel of AW
IC_{W_l1}	Input Channels carrying AW
OC_{W_l1}	Output Channels carrying AW
C_{W_l2}	Channel of AW
IC_{W_l2}	Input Channels carrying AW
OC_{W_l2}	Output Channels carrying AW
C_Q	Channel of QO
IC_Q	Input Channels carrying QO
OC_Q	Output Channels carrying QO
C_{BT}	Channel of $\mathbb{BT}$
IC_{BT}	Input Channels carrying $\mathbb{BT}$
OC_{BT}	Output Channels carrying $\mathbb{BT}$

TABLE 6.2: Summary of Channels used in DP-BT-Haskell. The subindex on the Channel name indicates the element type this channel is carrying, either producing or consuming. Channels prefixed with C are a generic form of denominating a channel independently of it is a producing or consuming. Channels prefixed with IC are Input Channels or Consumers. Channels prefixed with OC are Output Channels or Producers.

in the filter state. The need of bypassing all the aggregated wedges up to $\mathsf{G_{BT}}$ is for retro feeding all the pipeline with all the aggregated wedges a second time, in order to find an Aggregated wedge BT-Connector (AW-BT) as we defined in Definition 6.7. From retro feeding channel IC_{W2}, $\mathsf{actor_3}$ receives all the AW and builds ABT storing them in ST. Q Commands are downstream from $\mathsf{Sr_{BT}}$ to $\mathsf{actor_4}$ using channel IC_Q. $\mathsf{actor_4}$ receives all the commands, and for each of them, if there is match according to Definition 6.9, it enumerates those BT extracted from ST and downstream to $\mathsf{Sk_{BT}}$ through OC_{BT}.

6.3 Dynamic Pipeline for Enumerating Bitriangles

In this section, we present all the pseudo-code definitions of each of the stages described in section 6.2. In spite of this work has been implemented using

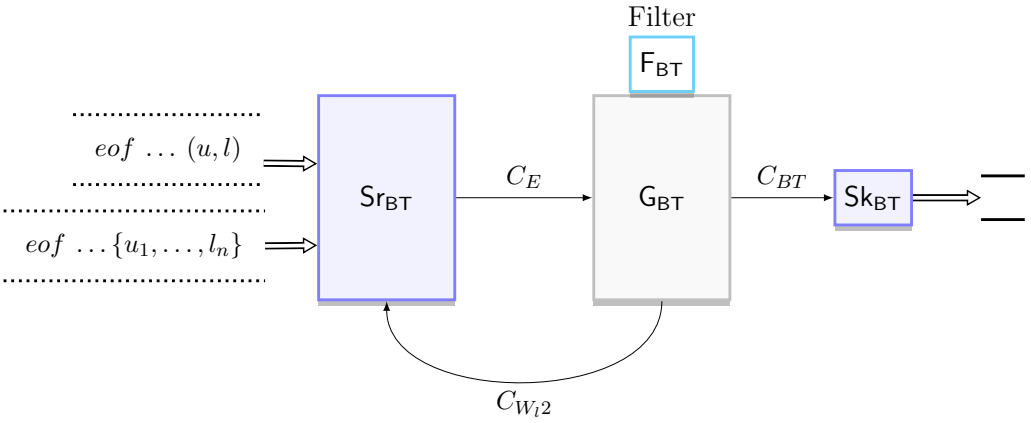


FIGURE 6.7: Example of an initial setup of $\mathsf{DP_{BT}}$. The text between dotted lines indicates the incoming data from an external source such as a file, socket, or any other. There are two incoming sources because one of them is carrying the edges and the other the commands QO. Thick black double lines at the most right indicate the output targe that can be file, socket, screen, or any other. There is no data there because it is the initial step of DP-BT-Haskell. At this initial step only $\mathsf{Sr_{BT}}$, $\mathsf{G_{BT}}$, and $\mathsf{Sk_{BT}}$ are set up with initial channels.

Haskell, and in particular DPF-Haskell, the pseudo-code algorithms presented here are language independent. Before starting with the details, we introduce in Table 6.3 auxiliary functions that we use in the pseudo-code to help understanding better the desired behavior of the algorithm.

Function	**Meaning**
`spawn`(F, l, ST)	Spawn new filter instance with parameters F, $l \in L$ and ST as the State of the Filter
`killFilter`	Kill this filter instance because PostCondition is not satisfied
`filterIsDead`	State after calling `killFilter` on filter. Indicates if Filter is die or not. If it is dead, this filter instance does not participate anymore in the pipeline streaming processor
`getState`	Get Current State ST for Filter Instance
`updateState`(ST)	Update Current State ST for Filter Instance
`push`$(\mathtt{v}, OC_x)$	push some value `v` to some Output Channel OC_x
`matchQ`(Q, BT)	Check if a QO Q produces $\mathbb{BT}$

TABLE 6.3: This table shows some auxiliary functions that are used to help understanding the pseudo-code general behavior. Depending on the language implementation chosen these functions might not exists at all, but we generalize here in order to describe a pseudo-code language-independent. For example in our Haskell implementation there is not `killFilter` because functions in Haskell clean after execution finish automatically by GHC

[A1]: Source ($\mathsf{Sr_{BT}}$): It process all the edges from the file or input channel IO_E and send to the following stages. It also receives from IC_{W_l2} all the feedback AW that is sending back the $\mathsf{G_{BT}}$. At the end it process also from other File or Input stream the Query Command Q to be sent to th filters

Input Data : IO_E: File or Input Stream with Set of Edges E
Input Commands: IO_Q: File or Input Stream with Query Operator Q
Input Channels : $IC = \langle IC_{W_l2} \rangle$
Output Channels: $OC = \langle OC_E, OC_{W_l1}, OC_{W_l2}, OC_Q, OC_{BT} \rangle$

```
1 forall (u,l) ∈ IO_E do                          // Edges to Generator/Filter
2 |  push((u,l), OC_E)
3 end
4 forall ⟨l,W_l⟩ ∈ IC_Wl2 do      // Feedback from Generator to Filter
5 |  push(⟨l,W_l⟩, OC_Wl2)
6 end
7 forall Q ∈ IO_Q do                                  // Send Query Commands
8 |  push(Q, OC_Q)
9 end
```

Source $\mathsf{Sr_{BT}}$ In algorithm [A1] we can see in line 2 how the edges arriving from the input stream of the graph are downstream to the pipeline. Another important part as well is line 5, the $\mathsf{Sr_{BT}}$ is retro feed with AW stream that is generated during pipeline execution. This is important to finally build the BT as we have describe in section 6.2. Finally, line 8 shows how all the queries are downstream as well.

[A2]: Generator ($\mathsf{G_{BT}}$): For each edge (u, l) it receives from previous stage, it spawn a new filter using l as parameter of the Filter and $\{u\}$ as the state. It also receives all the AW that previous filters built and sends back to $\mathsf{Sr_{BT}}$. Finally it sends to the $\mathsf{Sk_{BT}}$ the BT matched by filters according to Command Query Q

Parameter : F
Input Channels : $IC = \langle IC_E, IC_{W_l1}, IC_{W_l2}, IC_Q, IC_{BT} \rangle$
Output Channels: $OC = \langle OC_{W_l2}, OC_{BT} \rangle$

```
1 forall (u,l) ∈ IC_E do
2 |  spawn(F, l, ⟨l,{u}⟩)
3 end
4 forall ⟨l,W_l⟩ ∈ IC_Wl1 do  // Feedback channel to retrofit Source
5 |  push(⟨l,W_l⟩, OC_Wl2)
6 end
7 forall BT^(u1,u2,u3)_(l1,l2,l3) ∈ IC_BT do
8 |  push(BT^(u1,u2,u3)_(l1,l2,l3), OC_BT)
9 end
```

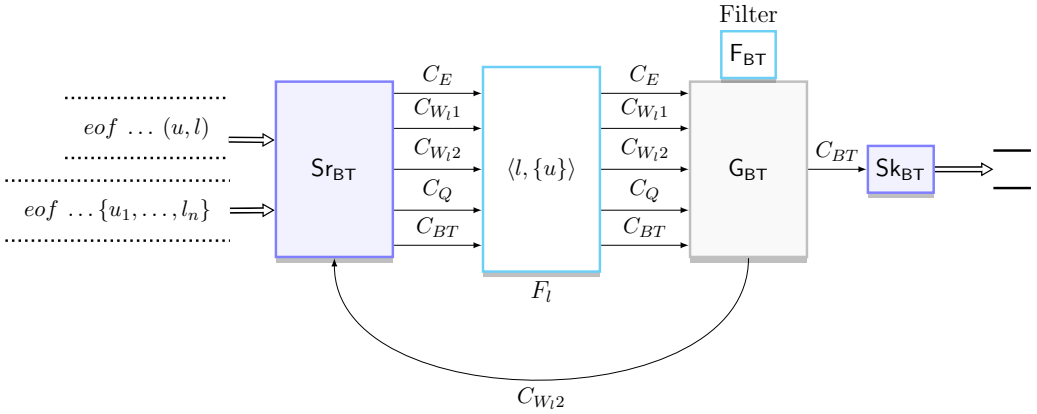


FIGURE 6.8: This is the evolving state of the $\mathsf{DP_{BT}}$ shows in Figure 6.7. This image is showing what happen when a $\mathsf{F_{BT}}$ is spawned. We can see how all the channels are set up in the middle of the spawned $\mathsf{F_{BT}}$ and also between $\mathsf{F_{BT}}$ and $\mathsf{G_{BT}}$.

Generator $\mathsf{G_{BT}}$ $\mathsf{G_{BT}}$ also have three main loops. In line 2 it receives each edge (u,l) not consumed by any already spawn $\mathsf{F_{BT}}$, and spawns a new $\mathsf{F_{BT}}$ using $l \in L$ as filter parameter and initializing $ST = \langle l,\{u\}\rangle$ as it can be seen in Figure 6.8. Spawn assumes that, the implementation will connect the channels to keep the downstream correct. Defines all the input channels of $\mathsf{G_{BT}}$ as input channels of the newly spawn $\mathsf{F_{BT}}$ and set $\mathsf{F_{BT}}$'s output channels as $\mathsf{G_{BT}}$ input channels. In line 5 we can see how the algorithm is receiving and retro feeding $\mathsf{Sr_{BT}}$ with all the AW produced by the $\mathsf{F_{BT}}$. This also assumes that OC_{W2} in $\mathsf{G_{BT}}$ is connected with IC_{W2} in $\mathsf{Sr_{BT}}$. Finally, line 8 sends all the results that Q matches in the different filters to $\mathsf{Sk_{BT}}$.

[A3]: Sink ($\mathsf{Sk_{BT}}$): It receives BT from $\mathsf{G_{BT}}$ and send to the File or Output Stream

Output : IO_{BT}: File or Output Stream with BT
Input Channels: $IC = \langle IC_{BT}\rangle$

1 **forall** $BT^{(u_1,u_2,u_3)}_{(l_1,l_2,l_3)} \in IC_{BT}$ **do** `// Read Results and put in` IO_{BT}
2 $\quad put(BT^{(u_1,u_2,u_3)}_{(l_1,l_2,l_3)}, IO_{BT})$
3 **end**

Sink $\mathsf{Sk_{BT}}$ In algorithm [A3] shows a simple stage that receives all results and sends them to some output handler (file, standard output, etc.).

[A4]: Filter ($\mathsf{F_{BT}}$): Call sequentially to all the actors in this filter

Filter Parameter : $l \in L$
Filter State : $ST = \mathbb{AW} + \mathcal{P}(\mathbb{DW}) + \mathcal{P}(\mathbb{AT})$
Input Channels : $IC = \langle IC_E, IC_{W_l1}, IC_{W_l2}, IC_Q, IC_{BT} \rangle$
Output Channels: $OC = \langle OC_E, OC_{W_l1}, OC_{W_l2}, OC_Q, OC_{BT} \rangle$

```
1 def filter():
2 |   actor1()
3 |   actor2()
4 |   actor3()
5 |   actor4()
6 end
```

Filter $\mathsf{F_{BT}}$ In algorithm [A4] we can see the simple call sequence over all the actor functions of the filter template. The values stored in the *state of the filter* with parameter l are of the sum type $ST = \mathbb{AW} + \mathcal{P}(\mathbb{DW}) + \mathcal{P}(\mathbb{AT})$.

[A5]: Actor1 ($actor_1$): Build a set of aggregated wedges. This is, $W_l \subseteq U$ adjacent to l Filter parameter. For each received edge (u', l') which $l \neq l'$ by pass the edge to next filters. It updates the State of the filter with W_l if it could build a W_l with more than 1 vertex in U.

Parameter : $l \in L$
ST : $\langle l, W_l \rangle$
Input Channels : $IC = \langle IC_E, IC_{W_l1}, IC_{W_l2}, IC_Q, IC_{BT} \rangle$
Output Channels: $OC = \langle OC_E, OC_{W_l1}, OC_{W_l2}, OC_Q, OC_{BT} \rangle$
Post-Cond : $|W_l| > 1 \vee$ `filterIsDead`

```
1  def actor1():
2  |   ⟨l, W_l⟩ ← getState
3  |   forall (u', l') ∈ IC_E do
4  |   |   if l = l' then
5  |   |   |   W_l ← W_l ∪ {u'}
6  |   |   else
7  |   |   |   push((u', l'), OC_E)
8  |   |   end
9  |   end
10 |   if |W_l| > 1 then
11 |   |   updateState(⟨l, W_l⟩)
12 |   |   push(⟨l, W_l⟩, OC_{W_l1})
13 |   else
14 |   |   killFilter
15 |   end
16 end
```

Filter-actor_1 actor_1 receives all edges not consumed by previous $\mathsf{F_{BT}}$. If the received edge (u', l') is incident to l i.e. $l = l'$, then u' will be added to the list of AW. Otherwise the edge is downstream to the next stage. In line 11 the state is updated and AW downstream, if and only if at least 1 wedge was collected. Otherwise, the filter is marked as dead using the `killFilter` function.

Filter-actor_2 In algorithm [A6] ADW is built. Following that idea and according to the Definition 6.6, this algorithm will collect all the AW from previous filters if an only if the condition in line 6 is met. Note that the two AW, W_l and W_l' have different vertices from the set L. W_l intersection check is mandatory since if AW are disjoint, we cannot aggregate them. After this checking from line 13 to line 15, the algorithm builds three disjoint Sets to separate upper edges in three subsets; those which are incident only of l and l' which are I and K and those that are shared by both lower layer vertices. This can be appreciated in Figure 6.5. Once ADW is built $ST = \mathsf{dw}$ updating the state for the next actor_3.

Filter-actor_3 actor_3 focuses on treating elements from feedback channel IC_{W2} which is going to downstream all the AW of all filters. This is because in order to build ABT finding all the possibles AW-BT. This is what is doing algorithm [A7] according to definition Definition 6.7 and Definition 6.8. If that can be achieved, algorithm sets $ST = \mathsf{at}$ and actor_4 can be executed.

Filter-actor_4 Once the execution reaches actor_4, it is ready for processing Q Query Operator. Since we have a compressed representation of BT, which is a similar idea exposed here [29], we need to enumerate incrementally all the BT present in the compressed format and filter the ones that match the command. The matches proceeds in the following manner. Given a ABT with the form $\left\langle \ell, \hat{U}_l \right\rangle$, such that $\ell = (l_1, l_2, l_3)$ and $\hat{U}_l = \langle I, J, K \rangle$, where $I \subseteq U, J \subseteq U, K \subseteq U$, if the QO contains a command with vertices, those vertices are searched in ℓ and $\hat{U}_l$. There are two posibilities. On the first case, if any of those vertices that belongs to Q matches ℓ, then forall $u_i \in I, u_j \in J, u_k \in K$ where $u_i \neq u_j \neq u_k$, a BT is built using a 6-cycle $(u_i, l_1, u_j, l_3, u_k, l_2, u_i)$ path. On the second case, if any of those vertices that belongs to Q matches on some vertex in $\hat{U}_l$, build only the BT in which 6-cycle $(u_i, l_1, u_j, l_3, u_k, l_2, u_i)$ path

[A6]: Actor2 ($actor_2$): Receiving all aggregated wedges from previous filters, build a set of all possible aggregated double-wedges $\mathsf{dw} = \{\langle (l, l'), U_l \rangle\}, \mathsf{dw} \subseteq \mathbb{DW}$, which first component l is smallest between the Parameter of the Filter and the vertices from the incoming wedges. At the end, it updates the State of the filter with dw if $\mathsf{dw} \neq \emptyset$

Parameter : $l \in L$
ST : $\langle l, W_l \rangle$
Input Channels : $IC = \langle IC_E, IC_{W_l 1}, IC_{W_l 2}, IC_Q, IC_{BT} \rangle$
Output Channels: $OC = \langle OC_E, OC_{W_l 1}, OC_{W_l 2}, OC_Q, OC_{BT} \rangle$

Pre-Cond : $W_l \subseteq U, |W_l| > 1$
Post-Cond : $|\mathsf{dw}| \geq 1 \vee$ `filterIsDead`

```
1  def actor2():
2      ⟨l, W_l⟩ ← getState
3      forall ⟨l', W'_l⟩ ∈ IC_{W_l 1} do
4          // Send Wedge from previous filters to next one
           push(⟨l', W_l⟩, OC_{W_l 1})
5          dw ← ∅
6          if W'_l ∩ W_l ≠ ∅ then
7              (l_l, l_u) ← (arg min_{l,l'}, arg max_{l,l'})
8              if l < l' then
9                  (W_{l_l}, W_{l_u}) ← (W_l, W'_l)
10             else
11                 (W_{l_l}, W_{l_u}) ← (W'_l, W_l)
12             end
13             I ← W_{l_l} \ W_{l_u}
14             J ← W_{l_l} ∩ W_{l_u}
15             K ← W_{l_u} \ W_{l_l}
16             U_l ← ⟨I, J, K⟩
17             dw ← dw ∪ {⟨(l_l, l_u), U_l⟩}
18         end
19     end
20     if dw = ∅ then
21         killFilter
22     else
23         updateState(dw)
24     end
25 end
```

contains that vertex in Q. Using the same construction process a BT is built if QO contains a command with edges and any of those edges matches with any of the possible BT 6-cycle path.

[A7]: Actor3 ($actor_3$): Receiving all aggregated wedges that came from feedback channel, build a Set of all possible Aggregated bitriangles $\mathsf{at} = \{\langle (l_l, l_m, l_u), U_l \rangle\}, \mathsf{at} \subseteq \mathbb{AT}$, , such that $l = l_l \vee l = l_u$, where l is the Filter Parameter and l_m is the middle vertex of the incoming wedge. At the end, it updates the State of the filter with at if $\mathsf{at} \neq \emptyset$

Parameter : $l \in L$
ST : $\mathsf{dw} \subseteq \mathbb{DW}$
Input Channels : $IC = \langle IC_E, IC_{W_l 1}, IC_{W_l 2}, IC_Q, IC_{BT} \rangle$
Output Channels: $OC = \langle OC_E, OC_{W_l 1}, OC_{W_l 2}, OC_Q, OC_{BT} \rangle$

Pre-Cond : $|\mathsf{dw}| \geq 1$
Post-Cond : $|\mathsf{at}| \geq 1 \vee$ `filterIsDead`

```
1  def actor3():
2      dw ← getState
3      at ← ∅
4      forall ⟨l′, W_l⟩ ∈ IC_{W_l 2} do
5          // By pass to be used by following Filters
           push(⟨l′, W_l⟩, OC_{W_l 2})
6          // For each double wedge in State
7          foreach ⟨(l_l, l_u), ⟨I, J, K⟩⟩ ∈ dw, l_l < l′ ∧ l_u > l′ do
8              I′ ← I ∪ J
9              K′ ← K ∪ J
10             if W_l ∩ I′ ≠ ∅ ∧ W_l ∩ K′ ≠ ∅ then
11                 I′ ← I′ ∩ W_l
12                 K′ ← K′ ∩ W_l
13                 Û_l ← ⟨I′, J, K′⟩
14                 at ← at ∪ { ⟨(l_l, l′, l_u), Û_l⟩ }
15             end
16         end
17     end
18     if at = ∅ then
19         killFilter
20     else
21         updateState(at)
22     end
23 end
```

The following pseudo-code definitions on algorithm [A9] and algorithm [A10] are auxiliary functions that are called from actor_4 if Q pattern match either with $\mathcal{P}(U + L)$ or $\mathcal{P}(E)$.

[A8]: Actor4 ($actor_4$): Receives all the Query Operator Q that arrives from channel IC_Q. For each QO Q it builds the bitriangles from $\mathsf{at} \subseteq at$ according to the query, and downstream to channel OC_{BT} to be processed by the Sink

Parameter : $l \in L$
ST : $\mathsf{at} \subseteq \mathbb{AT}$
Input Channels : $IC = \langle IC_E, IC_{W_l1}, IC_{W_l2}, IC_Q, IC_{BT} \rangle$
Output Channels: $OC = \langle OC_E, OC_{W_l1}, OC_{W_l2}, OC_Q, OC_{BT} \rangle$
Pre-Cond : $|\mathsf{at}| \geq 1$

```
1  def actor4():
2      at ← getState
3      forall Q ∈ IC_Q do
4          foreach ⟨(l_l, l_m, l_u), ⟨I, J, K⟩⟩ ∈ at do
5              switch Q do
6                  case P(U + L) do
7                      if
                        P(U+L) ∩ {l_l, l_m, l_u} ≠ ∅ ∨ P(U+L) ∩ (I ∪ J ∪ K) ≠ ∅
                        then
8                          bt ←
                            buildBtVertex(⟨(l_l, l_m, l_u), ⟨I, J, K⟩⟩), P(U + L))
9                          forall BT^(u1,u2,u3)_(l1,l2,l3) ∈ bt do
10                             push(BT^(u1,u2,u3)_(l1,l2,l3), OC_BT)
11                         end
12                     end
13                 end
14                 case P(E) do
15                     foreach (u, l) ∈ P(E) do
16                         if (l = l_l ∨ l = l_m ∨ l = l_u) ∧ u ∈ (I ∪ J ∪ K) then
17                             bt ← buildBtEdge(⟨(l_l, l_m, l_u), ⟨I, J, K⟩⟩), (u, l))
18                             forall BT^(u1,u2,u3)_(l1,l2,l3) ∈ bt do
19                                 push(BT^(u1,u2,u3)_(l1,l2,l3), OC_BT)
20                             end
21                         end
22                     end
23                 end
24             end
25         end
26     end
27 end
```

[A9]: Function `buildBtVertex`: Given a Set of Vertex in $\mathcal{P}(U+L)$, if any of those vertices matches with some vertex in the parameter of the function $\langle(l_l, l_m, l_u), \langle I, J, K\rangle\rangle$, build the set of bitriangles that matches that query

Input : $\langle(l_l, l_m, l_u), \langle I, J, K\rangle\rangle \in \mathsf{at}, \mathsf{at} \subseteq \mathbb{AT}$
Input : $\mathcal{P}(U + L)$
Output: $\mathsf{bt} \subseteq \mathbb{BT}$ or $\emptyset$ if cannot build any bt

```
1  def buildBtVertex(⟨(l_l, l_m, l_u), ⟨I, J, K⟩⟩, P(U + L)):
2      bt ← ∅
3      if P(U + L) ∩ {l_l, l_m, l_u} then
4          // If it is in lower i need to build all for this lower
           triplet
5          foreach i ∈ I do
6              foreach j ∈ J, j ≠ i do
7                  foreach k ∈ K, k ≠ i ∧ k ≠ j do
8                      bt ← bt ∪ {BT^(i,j,k)_(l_l,l_m,l_u)}
9                  end
10             end
11         end
12     else
13         // Otherwise just build those that are in the this
           upper v
14         foreach i ∈ I do
15             foreach j ∈ J, j ≠ i do
16                 foreach k ∈ K, k ≠ i ∧ k ≠ j ∧ (P(U + L) ∩ {i, j, k} ≠ ∅ do
17                     bt ← bt ∪ {BT^(i,j,k)_(l_l,l_m,l_u)}
18                 end
19             end
20         end
21     end
22     return bt
23 end
```

[A10]: Function `buildBtEdge`: Given a Set of Edges in $\mathcal{P}(E)$, if any of those edges matches with some edge in the parameter of the function $\langle(l_l, l_m, l_u), \langle I, J, K\rangle\rangle$, build the set of bitriangles that matches that query

Input : $\langle(l_l, l_m, l_u), \langle I, J, K\rangle\rangle \in \mathsf{at}, \mathsf{at} \subseteq \mathbb{AT}$
Input : (u, l)
Output: $\mathsf{bt} \subseteq \mathbb{BT}$ or $\emptyset$ if cannot build any bt

```
1  def buildBtEdge(⟨(l_l, l_m, l_u), ⟨I, J, K⟩⟩, P(U + L)):
2      bt ← ∅
3      foreach i ∈ I do
4          foreach j ∈ J, j ≠ i do
5              foreach
                 k ∈ K, k ≠ i ∧ k ≠ j ∧ hasEdge((u, l), (l_l, l_m, l_u), (i, j, k)) do
6                  bt ← bt ∪ {BT^(i,j,k)_(l_l,l_m,l_u)}
7              end
8          end
9      end
10     return bt
11 end
12 def hasEdge((u, l), (l_l, l_m, l_u), (i, j, k)):
13     if (u, l) = (l_l, i) ∨ (u, l) = (l_l, j) ∨ (u, l) = (l_u, j) ∨ (u, l) =
         (l_u, k) ∨ (u, l) = (l_m, i) ∨ (u, l) = (l_m, k) then
14         return True
15     end
16     else
17         return False
18     end
19 end
```

6.4 Correctness of the Algorithm

Given a bipartite graph G we prove the algorithm enumerates all the bitriangles in G without duplicating them. First, we prove that if $BT^{\{u_1,u_2,u_3\}}_{\{l_1,l_2,l_3\}} = (u_1, l_1, u_2, l_3, u_3, l_2, u_1)$ is a bitriangle occurring in G, then only one of the feasible permutations of this 6-cycle is contained in an aggregated bitriangle of $\mathbb{AT}$ (Theorem 6.10). Second, we prove that all the bitriangles occuring in G are contained in an aggregate bitriangles of $\mathbb{AT}$ (Theorem 6.11).

Theorem 6.10 (Uniqueness)**.** *Given a bipartite graph* $G = ((U \cup L), E)$*,* $\forall \mathsf{bt} \in \mathbb{BT}$ *IEBT stores* bt *in an* $\mathsf{at} \in \mathbb{AT}$ *only once.*

Proof. Let $\mathsf{bt} \in \mathbb{BT}$, $\mathsf{bt} = (u_1, l_1, u_2, l_3, u_3, l_2, u_1)$. Let us suppose that IEBT stores two different feasible permutations of bt, $\mathsf{bt}_1 = (u'_1, l'_1, u'_2, l'_3, u'_3, l'_2, u'_1)$ and $\mathsf{bt}_2 = (u''_1, l''_1, u''_2, l''_3,\ u''_3, l''_2, u''_1)$ in some aggregated bitriangles. By Definition 6.8 and $\mathsf{actor_3}$ line 7, $l'_1 < l'_2 < l'_3$ and $l''_1 < l''_2 < l''_3$. The fact that $\mathsf{bt}_1 \neq \mathsf{bt}_2$ implies that $(l'_1, l'_2, l'_3) \neq (l''_1, l''_2, l''_3)$. This means that (l'_1, l'_2, l'_3) is a permutation of (l''_1, l''_2, l''_3) (and viceversa). Thus, because of the strict order defined on L, just one, either $l'_1 < l'_2 < l'_3$ or $l''_1 < l''_2 < l''_3$ holds. Therefore, only one of the triples (l'_1, l'_2, l'_3) or (l''_1, l''_2, l''_3) is used in lines 7–16 in $\mathsf{actor_3}$ to include $\mathsf{bt} = (u_1, l_1, u_2, l_3, u_3, l_2, u_1)$ in an aggregated bitriangle □

Theorem 6.11. *Given a bipartite graph* G*, if the bitriangle* $\mathsf{bt} = (u_1, l_1, u_2, l_3, u_3, l_2, u_1) \in \mathbb{BT}$*, then* bt *can be enumerated.*

Proof. We proceed in a constructive way. Let $\mathsf{bt} = (u_1, l_1, u_2, l_3, u_3, l_2, u_1) \in \mathbb{BT}$. When $\mathsf{actor_1}$ running in a filter $\mathsf{S_F}$ with parameter l_1, F_{l_1}, ends reading all the edges, the state of $F_{l_1} = \langle l_1, W_{l_1} \rangle$ and $\{u_1, u_2\} \subseteq W_{l_1}$. Similarly, when $\mathsf{actor_1}$ running in filter F_{l_3} ends reading all the edges, $F_{l_3} = \langle l_3, W_{l_3} \rangle$ and $\{u_2, u_3\} \subseteq W_{l_3}$. This is, wedges (u_1, l_1, u_2) and (u_2, l_3, u_3) are stored in the aggregated wedges (states) of F_{l_1} and F_{l_3}, respectively. When running $\mathsf{actor_2}$ in filter F_{l_1} or in filter F_{l_3}, in lines 7-12 the pair $(l_l, l_u) \equiv (\arg\min_{l_1,l_3}, \arg\max_{l_1,l_3})$ is added to dw. This is, the double wedge $(u_1, l_l, u_2, l_u, u_3)$ is stored either, in the state F_{l_1} or the state of F_{l_3}. When running $\mathsf{actor_3}$, in filter F_{l_1} or in filter F_{l_3}, $\mathsf{actor_3}$ receives in $IC_{W_l 2}$, line 4, the connector wedge (l_2, W_{l_2}) from the incoming aggregated BT-connector. In lines 7-16, if the condition in line 10 holds, i.e. non empty intersection, since edges (u_1, l_2) and (u_3, l_2) are in E, $\{u_1, u_3\} \subseteq W_{l_2}$. Therefore, $\mathsf{actor_3}$ adds bt to the aggregated bitriangle of

the filter. This is, the permutation $(u_1, l_l, u_2, l_u, u_3, l_2, u_1)$ of the bitriangle bt is stored in at and therefore it can be listed by actor_4 □

6.5 DP-BT-Haskell implementation

As we have seen in the previous chapter 5, first we need to define the $\mathsf{DP}_{\mathsf{BT}}$ using the DSL.

```
1 type DPBT = Source (Channel (Edge :<+> W :<+> Q :<+> BT :<+>
  ↪  BTResult :<+> W :<+> Eof))
2       :=> Generator (Channel (Edge :<+> W :<+> Q :<+> BT :<+>
        ↪  BTResult :<+> Eof))
3       :=> FeedbackChannel (W :<+> Eof)
4       :=> Sink
```

SOURCE CODE 6.14: [BTriangle.hs] Enconding of DP-BT-Haskell

In Source Code 6.14 can be appreciated the use of feedback channel in the highlighted line. Automatically DP-BT-Haskell is going to connect this with the $\mathsf{Sr}_{\mathsf{BT}}$.

$\mathsf{Sr}_{\mathsf{BT}}, \mathsf{Sk}_{\mathsf{BT}}, \mathsf{G}_{\mathsf{BT}}$ are not going to be covered because they are straightforward to follow from the code. The most sofisticated part of the algorithm, as we have seen in section 6.3, relies on actor filter.

In Source Code 6.15 actor_1 source code can be appreciated. Since all the actors are inside the same filter context, all of them have access to read and write channels of all the filters. That explains the number of parameters which is generated by the IDL. The first highlighted line is the catamorphism (foldM_) of the IC_E channel, which corresponds to line 5. At the last highlighted line the code is downstream the aggregated wedge collected in this filter.

In Source Code 6.16, the interesting part of actor_2 is the construction of dw. That is done by the function buildDW and buildDW' which are building the three subsets required to create aggregated double-wedges dw.

actor_3 in Source Code 6.17, shows the process of collecting aggregated bitriangles at. In highlighted lines on Source Code 6.17 we can see the algorithm to

```haskell
actor1 :: Edge
       -> ReadChannel (UpperVertex, LowerVertex)
       -> ReadChannel W
       -> ReadChannel Q
       -> ReadChannel BT
       -> ReadChannel BTResult
       -> ReadChannel W
       -> WriteChannel (UpperVertex, LowerVertex)
       -> WriteChannel W
       -> WriteChannel Q
       -> WriteChannel BT
       -> WriteChannel BTResult
       -> WriteChannel W
       -> StateT FilterState (DP st) ()
actor1 (_, l) redges _ _ _ _ _ we ww1 _ _ _ _ = do
 foldM_ redges $ \e@(u', l') -> do
   e `seq` if l' == l then modify $ flip modifyWState u' else push e
    ↪  we
 finish we
 state' <- get
 case state' of
   Adj w@(W _ ws) -> when (IS.size ws > 1) $ push w ww1
   _              -> pure ()
```

Source Code 6.15: [BTriangle.hs] actor_1

detect when the AW received in the feedback channel are AW-BT according to Definition 6.7.

actor_4 in Source Code 6.18 shows the pattern match done over the query Q sum type and the construction of the final BT to be downstream to the $\mathsf{Sk}_{\mathsf{BT}}$.

Finally, definition of `filterBTByEdge` and `filterBTByVertex` can be seen in Source Code 6.19 and differs from the pseudo-code presented in algorithm [A10] and algorithm [A9] because it is using Haskell specific combinators and for-comprehension lists to take advantage of the non-strictness of the language as well as some concurrency primitives.

```haskell
actor2 :: Edge
       -> ReadChannel (UpperVertex, LowerVertex)
       -> ReadChannel W
       -> ReadChannel Q
       -> ReadChannel BT
       -> ReadChannel BTResult
       -> ReadChannel W
       -> WriteChannel (UpperVertex, LowerVertex)
       -> WriteChannel W
       -> WriteChannel Q
       -> WriteChannel BT
       -> WriteChannel BTResult
       -> WriteChannel W
       -> StateT FilterState (DP st) ()
actor2 (_, l) _ rw1 _ _ _ _ _ ww1 _ _ _ _ = do
 state' <- get
 case state' of
   Adj (W _ w_t) -> do
     modify $ const $ DoubleWedges mempty
     foldM_ rw1 $ \w@(W l' w_t') -> do
       push w ww1
       buildDW w_t w_t' l l'
     finish ww1
   _ -> pure ()

buildDW :: IntSet -> IntSet -> LowerVertex -> LowerVertex -> StateT
↪  FilterState (DP st) ()
buildDW w_t w_t' l l' =
let pair       = Pair (min l l') (max l l')
      paramBuild = if l < l' then (w_t, w_t') else (w_t', w_t)
      ut         = uncurry buildDW' paramBuild
in  if (IS.size w_t > 1) && l /= l' && not (IS.null (IS.intersection
↪  w_t w_t')) && not (nullUT ut)
      then modify $ flip modifyDWState (DW pair ut)
      else pure ()

buildDW' :: IntSet -> IntSet -> UT
buildDW' !w_t !w_t' = (w_t IS.\\ w_t', IS.intersection w_t w_t',
↪  w_t' IS.\\ w_t)
```

Source Code 6.16: [BTriangle.hs] actor$_2$

```haskell
actor3 :: Edge
       -> ReadChannel (UpperVertex, LowerVertex)
       -> ReadChannel W
       -> ReadChannel Q
       -> ReadChannel BT
       -> ReadChannel BTResult
       -> ReadChannel W
       -> WriteChannel (UpperVertex, LowerVertex)
       -> WriteChannel W
       -> WriteChannel Q
       -> WriteChannel BT
       -> WriteChannel BTResult
       -> WriteChannel W
       -> StateT FilterState (DP st) ()
actor3 (_, l) _ _ _ _ _ rfb _ _ _ _ _ wfb = do
  state' <- get
  case state' of
    DoubleWedges dwtt -> do
      modify $ const $ BiTriangles mempty
      foldM_ rfb $ \w@(W l' w_t') -> do
        push w wfb
        when (hasDW dwtt) $ do
          let (DWTT dtlist) = dwtt
          forM_ dtlist $ \(DW (Pair l_l l_u) ut) ->
            let triple = Triplet l_l l' l_u
                result =
                  if l' < l_u && l' > l_l then filterUt w_t' ut else
                  ↪ Nothing
            in  maybe (pure ()) (modify . flip modifyBTState . BT
            ↪ triple) result
      finish wfb
    _ -> pure ()

filterUt :: IntSet -> UT -> Maybe UT
filterUt wt (si, sj, sk) =
  let si' = IS.filter (`IS.member` wt) si
      sj' = IS.filter (`IS.member` wt) sj
      sk' = IS.filter (`IS.member` wt) sk
      sij' = si' `IS.union` sj'
      sjk' = sk' `IS.union` sj'
      wtInSome = not (IS.null sij' || IS.null sjk')
  in  if wtInSome then Just (sij', sj, sjk') else Nothing
```

Source Code 6.17: [BTriangle.hs] actor$_3$

```haskell
actor4 :: Edge
       -> ReadChannel (UpperVertex, LowerVertex)
       -> ReadChannel W
       -> ReadChannel Q
       -> ReadChannel BT
       -> ReadChannel BTResult
       -> ReadChannel W
       -> WriteChannel (UpperVertex, LowerVertex)
       -> WriteChannel W
       -> WriteChannel Q
       -> WriteChannel BT
       -> WriteChannel BTResult
       -> WriteChannel W
       -> StateT FilterState (DP st) ()
actor4 (_, l) _ _ query _ rbtr _ _ _ wq _ wbtr _ = do
  state' <- get
  case state' of
    BiTriangles bttt -> do
      rbtr |=> wbtr
      foldM_ query $ \e -> do
        push e wq
        unless (hasNotBT bttt) $ sendBts bttt e wbtr
    _ -> pure ()

sendBts :: MonadIO m => BTTT -> Q -> WriteChannel BTResult -> m ()
sendBts (BTTT bttt) q@(Q c _ _) wbtr = case c of
  ByVertex vx    -> forM_ bttt (\bt -> filterBTByVertex bt vx (flip
  ↪  push wbtr . RBT q))
  ByEdge   edges -> forM_ bttt (\bt -> filterBTByEdge bt edges (flip
  ↪  push wbtr . RBT q))
  AllBT          -> forM_ bttt (R.mapM_ (flip push wbtr . RBT q) .
  ↪  buildBT)
  Count          -> forM_ bttt (flip push wbtr . RC q . R.length .
  ↪  buildBT)
  _              -> pure ()
```

Source Code 6.18: [BTriangle.hs] actor$_4$

```haskell
filterBTByVertex :: MonadIO m => BT -> IntSet -> ((Int, Int, Int,
↪  Int, Int, Int, Int) -> IO ()) -> m ()
filterBTByVertex bt vertices f =
  if inLower bt vertices then
     liftIO
     . mapConcurrently_ f
     . buildBT
     $ bt
  else
    if inUpper bt vertices
      then buildBT' bt vertices f
      else pure ()

filterBTByEdge :: MonadIO m => BT -> Set Edge -> ((Int, Int, Int,
↪  Int, Int, Int, Int) -> IO ()) -> m ()
filterBTByEdge bt edges f =
  when (getAny $ foldMap (`hasEdge` bt) edges)
   $ liftIO . mapConcurrently_ f . R.filter (isInSetEdge edges) .
   ↪  buildBT $ bt
```

SOURCE CODE 6.19: [Edges.hs] `filterBTByVertex` and `filterBTByEdge`

6.6 Chapter Summary

In this chapter we first introduced some basic definitions. These definitions are the foundation to represent the graph index created from the input bipartite graph and to define the IEBT algorithm. Second, we presented a general view of the algorithm. Third, we deeply described the definitions of the stages of the $\mathsf{DP}_{\mathsf{BT}}$. Then, we provided a correctness proof of the IEBT algorithm. Finally, we give the main details of the implementation of the IEBT algorithm using DPF-Haskell.

Chapter 7

Empirical Evaluation

This chapter reports on the experimental study conducted to assess the efficiency of DP-BT-Haskell. This study aims at answering the research questions that emerged from the motivation of this work.

The fundamental part of this work focuses on incremental enumeration BT in BG. In spite of this, there are other important areas for doing empirical analysis such as memory consumption, thread scheduling, and execution time.

Regarding that, we have asked ourselves the following research questions that guide the empirical evaluation analysis, and we try to answer them with the conducted experiments: **RQ1)** Does DP-BT-Haskell generate incremental results regardless of the size of the graph? **RQ2)** Does the type of query Q impact on the execution of DP-BT-Haskell? **RQ3)** How effectively DP-BT-Haskell implements a *pay-as-you-go* model? **RQ4)** Does DP-BT-Haskell handle memory and threads efficiently?

7.1 Experiments Configuration

We have conducted different kinds of experiments to answer our research questions and verify the behavior of the approach in various benchmarks. **i)** *Continuous behavior Analysis*: using `dief@t` and `dief@k` [26] to assess the continuous behavior capabilities of the implemented algorithm (generation of incremental results). We run this experiment only once for each scenario per network (see subsection 7.1.3). **ii)** *Benchmark Analysis*: to identify how

the behavior of DP-BT-Haskell varies depending on the type of query command defined in Definition 6.9. This experiment have been conducted with `criterion` [37] which runs each scenario 1000 times for each network. **iii)** Finally, we have executed a *Performance Analysis* in which we have to gather profiling data from Glasgow Haskell Compiler (GHC) for one of the graphs, to measure how the program performs regarding multithreading and memory allocation. This experiment has been run only once for memory analysis and once for multi-threading analysis. On each case we selected the biggest network and most time consuming scenario according to the results of the *Benchmark Analysis*. In the following subsections, we detail the different aspects of the configuration such as hardware, Haskell compilation flags, metrics, and benchmark to conduct these experiments.

7.1.1 Benchmark

The experiments have been evaluated over the networks that composed the benchmark Konect Networks [68]. Specifically, the networks used in the literature have been selected [69–72].

Network	$\|U\|$	$\|L\|$	$\|E\|$	**Wedges**	**#BT**
Dbpedia	18422	168338	233286	1.45×10^8	3.62×10^8
Moreno Crime	829	551	1476	4816	211
Opsahl UC Forum	899	522	33720	174069	2.2×10^7
Wang Amazon	26112	799	29062	3.4×10^6	110269

TABLE 7.1: This table shows the different networks used in the experiments. We provide some metrics of the networks used in order to understand a little more about the topology of each BG. In particular, we are showing in the last column 2 metrics that are important and could affect results which are a number of wedges and bitriangles

The criteria for selecting those networks have followed the idea of conducting the analysis on one of the big networks [69] used on the BT counting work [8]. The rest of the networks, from the same data source [68], have been selected randomly but taking into consideration different sizes and topologies.

7.1.2 Metrics

This section describes the different metrics that we have gathered during the experimental analysis, as well as the tools used to obtain those metrics.

`dief@t` and `dief@k` The definition of this metric has been discussed on section 2.4, although we provide here a quick reminder. `dief@t` measures the continuous efficiency during the first t time units of execution regarding the results generated by DP-BT-Haskell. Time t in our experiments represents the number of nanoseconds elapsed to deliver that result from the moment the DP-BT-Haskell finishes the execution of actor_3 and it started executing actor_4, so actor_4 is able to start processing commands. A higher value of `dief@t` indicates better continuous behavior. Therefore, time 0 is equal to the start of actor_4 execution. `dief@k` measures the continuous efficiency while producing the first k answers regarding the results generated by DP-BT-Haskell. A lower value of `dief@k` indicates better continuous behavior.

Average Running Time The average of the total running time of 1000 resamples using `criterion` tool [37]. In each sample, the running time is measure from the beginning of the execution of the program until when the last answer is produced.

Total Running Time Total running time of one execution set over each experimental setup and benchmark. The total running time is measured from the beginning of the execution of the program until the end, i.e., the last answer is produced. This is provided by default in Haskell by enabling flag `-s` in the execution command-line argument.

GHC productivity Measures the proportion of Mutator (MUT) execution time vs. Garbage Collection (GC) time. In Haskell Programming Language Mutator is the acronym of a thread evaluating an Haskell expression – running computations of our program –. GC time is the amount of time that Garbage Collector is running. The closest the value to 100%, the better because it indicates that the program was executing all the time without being paused for running Garbage Collector. This is provided by default in Haskell by enabling flag `-s` in the execution command-line argument.

Distribution of Threads per Core It measures the number of threads per processor on each time slot of execution. This metric is gathered by `TreadScope` [39] tool.

Distribution of Allocated Memory per Data Type It measures the amount of allocated memory per Haskell Data Type. This metric is gathered by `eventlog2html` [41] tool.

7.1.3 Scenarios

An experimental scenario is a specific configuration of the Query Operator that we have defined in Definition 6.9. That means, that for each network that we run an experiment with a different configuration of the QO to obtain different results.

Definition 7.1 (Incidence Level)**.** Let G be a Bipartite Graph. Let v be a vertex such that $v \in V$ of G. Let e be an edge such that $e \in E$ of G. A *low, medium, or high incidence level* for vertices is defined as following: Low A vertex v has low incidence if its degree is less than 1% of $|V|$, Medium A vertex v has medium incidence if its degree is between $1\% - 25\%$ of $|V|$, and High A vertex v has high incidence if its degree is more than 25% of $|V|$ A *low, medium, or high incidence level* for edges is defined as following: Low A edge $e = (u, l)$ has low incidence if any of its vertices u or l has degree less than 1% of $|V|$, Medium A edge $e = (u, l)$ has medium incidence if any of its vertices u or l has degree between $1\% - 25\%$ of $|V|$, and High A edge $e = (u, l)$ has high incidence if any of its vertices u or l has degree more than 25% of $|V|$

Having the previous Definition 7.1, we define the following scenarios to conduct all the experiments regardless of the network.

Selection of values for QO The selection of values, either vertices V or edges E, has been done pseudo-randomly, i.e., given Definition 7.1 the next steps were followed: **i)** Sort the vertices by its degree. **ii)** Randomly select following a uniform distribution, a vertex or edge depending on the scenario, from the subset of vertices or edges that fulfill the Definition 7.1. **iii)** Execution of a sample of experiments to check if that selections provides results or not. If not, the test case is eliminated.

Scenario ID	Name	Search by
E-H	Edge High	edge with high incidence
E-L	Edge Low	edge with low incidence
E-M	Edge Medium	edge with medium incidence
VL-H	$l \in L$ High	vertex in lower layer with high incidence
VL-L	$l \in L$ Low	vertex in lower layer with low incidence
VL-M	$l \in L$ Medium	vertex in lower layer with medium incidence
VU-H	$u \in U$ High	vertex in upper layer with high incidence
VU-L	$u \in U$ Low	vertex in upper layer with low incidence
VU-M	$u \in U$ Medium	vertex in upper layer with medium incidence

TABLE 7.2: The first column of the table is an identifier to be reference in the rest of the section. E and V indicate if the QO scenario contains edge or vertex query, respectively. VL or VU indicates if those vertices belongs to L (lower layer) or U upper layer. After the $-$ symbol, the letters L, M, H indicate the incidence level defined in Definition 7.1

7.1.4 Implementation

Hardware Platform All the experiments have been executed in the *HPC Cluster at UPC*. The nodes' architecture running in the cluster is $x86$ 64 bits with a 24-*Core Intel(R) Xeon(R) CPU X5650* processor of 2.67 GHz. Regarding memory, the allocated nodes have been requested from $40GB$ up to $120GB$ of RAM for the biggest Dbpedia Network (Dbpedia) graph. These machines also have 256 KB of L2 cache memory, and 12 MB of L3 cache.

Haskell Setup The implementation uses GHC 8.10.4 plus the following set of Haskell libraries: `dyanmic-pipeline` 0.3.2.0 [16], `bytestring` 0.10.12.0 [73], `containers` 0.6.2.1 [38], `relude` 1.0.0.1 [74] and`unagi-chan` 0.4.1.3 [66] . The `relude` library is utilized because `Prelude` was disabled from the project with the language extension `NoImplicitPrelude` [75]. We have compiled our program using `stack` version 2.5.1 [76] with the following command and option flags[1]: `stack build --ghc-options "-threaded -O3 -rtsopts -with-rtsopts=-N"`. Flag `threaded` indicates GHC to compile the program with thread support enable. `-O3` is the highest optimization level for the compiler. Regarding `-with-rtsopts=-N`, it allows us to change dynamically on each runtime execution command, the number of processors, and other execution flags that we will explain in section 7.1.4.

[1]For more information about package.yaml or cabal file, please check https://github.com/jproyo/upc-miri-tfm/tree/main/bt-graph-dp

Optimal Execution Parameters GHC enables different flags parameters to speed up the execution. Unfortunately there is no recipe to tune those parameters in the correct way and each program needs to be analyzed to take advantage of GHC capabilities.[2] The most important parameters to be tuned are memory and the number of processors. In the case of the number of processors, we have run all the experiments between 6 and 12 cores. The detail of each run can be found in Table 7.3. Setting up memory allocation is not a straightforward task, because the combination of two parameters needs to be considered. `-A` flag which indicates the allocation area for the garbage collector, which is fixed and never resized, and `-H` which is the heap size. We used the tool `ghc-gc-tune` [77] to find the best combination of those parameters; it implements a heuristic algorithm that tries different setups until it finds an *suboptimal* combination for memory allocation. *Suboptimal combination* here refers to find the fastest total execution time with the less amount of allocated memory, as it can be seen in Figure 7.1. We have run DP-BT-Haskell with that tool, obtaining the following result that can be appreciated in Figure 7.1

In Figure 7.1 we can appreciate that the tool runs sereval times the same program with different configurations on flags `-A` and `-H` until if it finds a minimum. The dark blue shows the better performance where the curve find its minimum execution time. This indicates two possible suboptimal setup, either when `-A` is equal to `-H`, or either when `-A` is $\frac{1}{4}$ of the `-H`. For all our experiments we have selected the first option which `-A` is equal to `-H`, because choosing any of the two best combinations provides the same results as it can be seen in Figure 7.1. It is important to remind the reader that `ghc-gc-tune` [77] uses an heuristic algorithm, providing suboptimal results.

7.2 Experimental Results

7.2.1 E1: Continuous behavior Analysis

Goal In this experiment, we assess the ability of DP-BT-Haskell to generate results incrementally. In order to do that, we use the `diefpy` Tool Tool [27]

[2]For the sake of reproducibility, the execution parameters details are explained in Appendix A.2

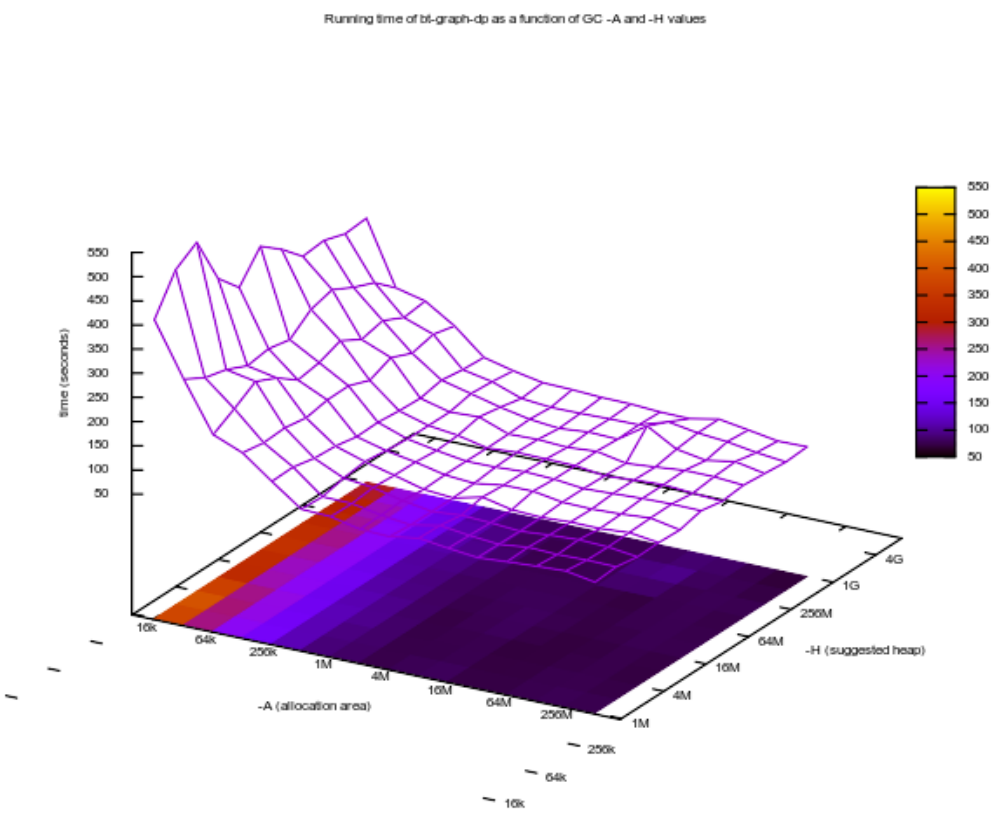


FIGURE 7.1: This figure shows the results obtained after running DP-BT-Haskell with the `ghc-gc-tune` tool in order to find the suboptimal configuration for Memory allocation. y axis shows the total execution time, x axis is the `-A` configuration flag and z axis is the `-H` configuration flag.

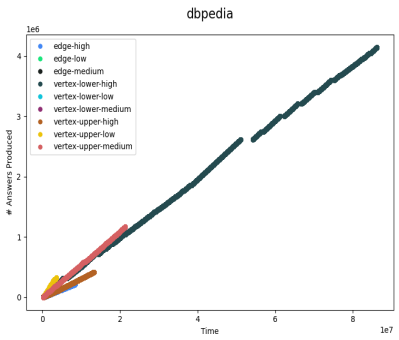
which implements *Diefficiency Metric* [26] measurement analysis. This experiment allows us to answer research questions [R1] and [R3] defined in Research Questions 7.

Procedure We execute this experiment for each of the networks described in subsection 7.1.1 and for each scenario described in subsection 7.1.3. This experiment has been executed five times on each case until we found the proper vertex or edges in the selection described in section 7.1.3. The criteria followed by the selection of the vertices or edges are detailed in section 7.1.3. The metric Diefficiency Metric `dief@t` –described in section 2.4– is used to measure the continuous behavior of the proposed approach in a given time frame. Table 7.3 depicts the different configurations evaluated in this experiment. The timeout for all the configuration scenarios setups that appears in Table 7.3 have been set in 48 hours.

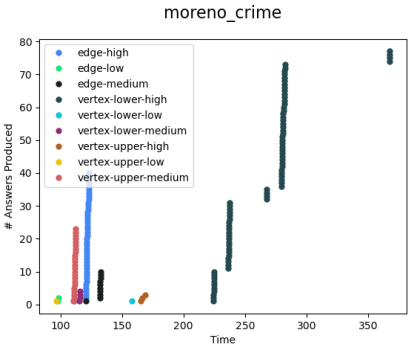
Network	Scenario ID	Exec Flags	Query
Dbpedia	E-H	+RTS -A5G -N8 -c -H5G -RTS	by-edge (921, 4)
	E-L	+RTS -A5G -N8 -c -H5G -RTS	by-edge (383, 397)
	E-M	+RTS -A5G -N8 -c -H5G -RTS	by-edge (540, 60)
	VL-H	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 9
	VL-L	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 809
	VL-M	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 511
	VU-H	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 921
	VU-L	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 93
	VU-M	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 540
Moreno Crime	E-H	+RTS -A5G -N6 -c -H5G -RTS	by-edge (413, 419)
	E-L	+RTS -A5G -N6 -c -H5G -RTS	by-edge (361, 19)
	E-M	+RTS -A5G -N6 -c -H5G -RTS	by-edge (531, 196)
	VL-H	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 95
	VL-L	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 187
	VL-M	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 97
	VU-H	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 2
	VU-L	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 793
	VU-M	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 533
Opsahl UC Forum	E-H	+RTS -A5G -N6 -c -H5G -RTS	by-edge (213, 33)
	E-L	+RTS -A5G -N6 -c -H5G -RTS	by-edge (398, 10)
	E-M	+RTS -A5G -N6 -c -H5G -RTS	by-edge (129, 171)
	VL-H	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 289
	VL-L	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 258
	VL-M	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 433
	VU-H	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 395
	VU-L	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 390
	VU-M	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 207
Wang Amazon	E-H	+RTS -A5G -N6 -c -H5G -RTS	by-edge (839, 9)
	E-L	+RTS -A5G -N6 -c -H5G -RTS	by-edge (10987, 36)
	E-M	+RTS -A5G -N6 -c -H5G -RTS	by-edge (19630, 84)
	VL-H	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 124
	VL-L	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 321
	VL-M	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 64
	VU-H	+RTS -A5G -N8 -c -H5G -RTS	by-vertex 1727
	VU-L	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 9970
	VU-M	+RTS -A5G -N6 -c -H5G -RTS	by-vertex 73

TABLE 7.3: This table shows all the different experiments setups conducted for E1. Execution flags are the Runtime execution flags which needs to be set on the execution command as it is detailed in section A.2. On the last column we can see for each scenario, the query command executed for BT enumeration (see Definition 6.9). The timeout configured for all this experiments setup were 48 hours

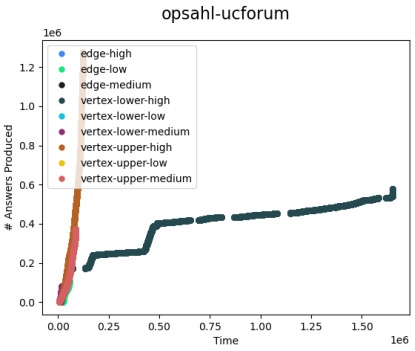
Observed Results As observed in the results above in Figure 7.2a, Figure 7.2b, Figure 7.2c and Figure 7.2d in all the networks and experiments setups, BT are incrementally enumerated and delivered.



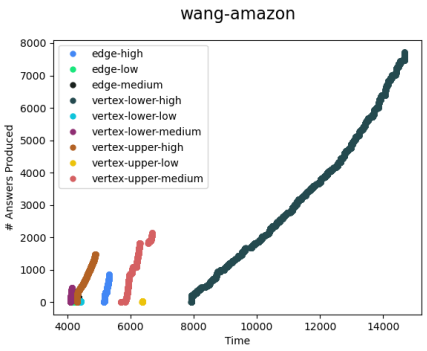
(A) `dief@t` metric results after running all experiment scenarios described in Table 7.3 on the DBpedia network. Each color represents one scenario. In the figure it is shown how VL-H scenario (dark green line) seems to show more continuous behavior compared with the rest because of the level of incidence



(B) `dief@t` metric results after running all experiment scenarios described in Table 7.3 on Moreno Crime network. Each color represents one scenario. The same that is exhibit in dbpedia, we can see here how VL-H seems continuously delivering results although with some gaps between some t



(C) `dief@t` metric results after running all experiment scenarios described in Table 7.3 on Opsahl UC Forum network. Each color represents one scenario. In this case VU-H with orange line is showing more continuous behavior. Green darks seems to move towards the end of the time but is not growing vertical which is the amount of answers produced.



(D) `dief@t` metric results after running all experiment scenarios described in Table 7.3 on Wang Amazon network. Each color represents one scenario. Although all shows a good level of continuous behavior, also VL-M is the one with most continuous behavior in terms of t

FIGURE 7.2: These figures show `dief@t` observed results after running all the scenarios described in Table 7.3 for each network. y axis represents the number of Answers produced and x axis is the t time of the `dief@t` metric describe in section 2.4. For example in Figure 7.2a we can see that dark green color shows VL-H scenario on DBpedia network. The more data points distributed throughout the x axis, the higher, the continuous behavior.

The continuous behavior can be appreciated in Figure 7.2 having the fact that any of those points draw a straight vertical line. If that were the case, the plot would have indicated that all the generated answers are being produced at the same t time.

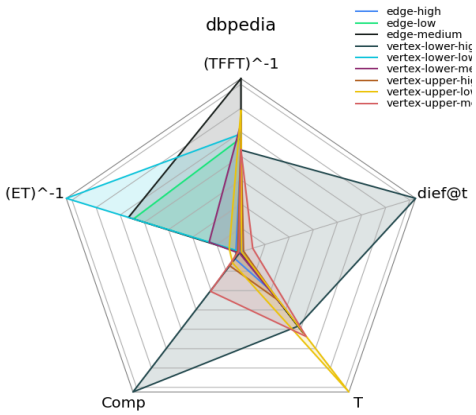
On the one hand, we observe higher `dief@t` values in Table 7.4 for all the networks with more bitriangles (see Table 7.1) in the nine scenarios indicating continuous behavior. As we have describe in section 2.4 a `dief@t` with higher values indicates better continuous behavior. Therefore, from observed values we can state that 100% of the experimented scenarios are generating continuous answers provided by DP-BT-Haskell. On the other hand, we also observe from the blue light highlighted values in Table 7.4, how the results obtained in experiments VU-H for Opshal with 1.99×10^{12}, E-H for Moreno with 9.85×10^{3}, VL-H for dbpedia with 1.81×10^{14} and VL-H for Wang Amazon with 2.24×10^{7} (see Table 7.2), are more continuous compare to the rest of the experiments setups. It can also be observed from results in Table 7.4 that in most of the scenarios and networks, vertices and edges higher incidence scenarios are reporting more continuous behavior compared with the other scenarios, highlighted in cells with light blue color. There are some interesting cases that are being reported as well which are the cells highlighted in yellow light color. In those cases, for example in Wang Amazon, VU-H is supposed to have the highest `dief@t` value with respect to VU-L and VU-M of the same network. However, we can appreciate that VU-M is reporting slightly higher `dief@t` value for Wang Amazon compare with VU-H. The same behavior can be appreciated in the rest of the network for some specific scenarios highlighted in yellow light color.

Regarding `dief@k` values reported in Table 7.4, we can appreciate that in networks with small number of bitriangles like Wang Amazon and Moreno Crime, `dief@k` values are low indicating continuous behavior as well. In the case of Wang Amazon lowest value of `dief@k` is in VL-M scenario. Moreno Crime are reporting all `dief@k` with same value 0. In the case of Dbpedia the lowest `dief@k` value is VL-H, which matches with the highest `dief@t` values of the same scenario and network. Finally, Opsahl UC Forum is reporting the lowest `dief@k` value in VL-M with 1.05×10^{5} but the difference with the rest of scenarios is not significant.

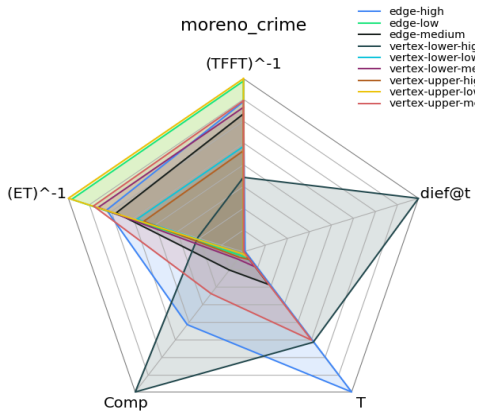
Another plots that `diefpy` tool provides are shown in Figure 7.3a, Figure 7.3b, Figure 7.3c and Figure 7.3d. Here we can observe how all the networks under

Network	Scenario ID	dief@t Metric	dief@k Metric
Moreno Crime	VU-H	6.05×10^{2}	0.00
	VL-L	2.10×10^{2}	0.00
	VL-H	7.95×10^{3}	0.00
	VL-M	1.01×10^{3}	0.00
	E-L	5.40×10^{2}	0.00
	E-M	2.37×10^{3}	0.00
	VU-L	2.71×10^{2}	0.00
	VU-M	5.89×10^{3}	0.00
	E-H	9.85×10^{3}	0.00
Dbpedia	VU-H	3.32×10^{13}	1.97×10^{5}
	VL-L	1.62×10^{10}	3.65×10^{6}
	VL-H	1.81×10^{14}	2.34×10^{4}
	VL-M	4.07×10^{12}	4.41×10^{5}
	E-L	3.33×10^{11}	2.90×10^{5}
	E-M	1.68×10^{12}	4.99×10^{5}
	VU-L	2.74×10^{13}	3.60×10^{5}
	VU-M	8.80×10^{13}	1.48×10^{5}
	E-H	1.75×10^{13}	3.28×10^{5}
Opsahl UC Forum	VU-H	1.99×10^{12}	1.27×10^{5}
	VL-L	2.17×10^{9}	4.92×10^{5}
	VL-H	6.44×10^{11}	1.90×10^{5}
	VL-M	1.30×10^{11}	1.05×10^{5}
	E-L	1.71×10^{11}	1.11×10^{5}
	E-M	7.09×10^{10}	1.20×10^{5}
	VU-L	1.58×10^{10}	1.18×10^{5}
	VU-M	5.93×10^{11}	1.06×10^{5}
	E-H	1.02×10^{11}	2.93×10^{5}
Wang Amazon	VU-H	1.50×10^{7}	43.6
	VL-L	3.90×10^{5}	41.0
	VL-H	2.24×10^{7}	63.1
	VL-M	4.74×10^{6}	34.5
	E-L	3.55×10^{6}	46.2
	E-M	1.61×10^{6}	41.2
	VU-L	3.41×10^{5}	51.4
	VU-M	1.83×10^{7}	2.48×10^{3}
	E-H	8.06×10^{6}	42.3

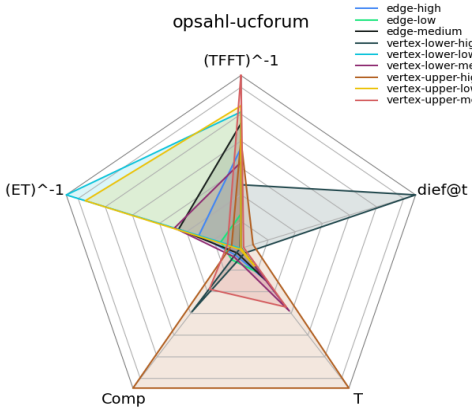
TABLE 7.4: This table shows all the `dief@t` and `dief@k` values for all the experiments setups conducted for E1. Higher values of `dief@t` indicates better continuous behavior. Lower values of `dief@k` indicates better continuous behavior. Highlighted blue lines indicates scenarios that are behaving according to the incidence level. Yellow highlighted lines are scenarios that reports values lower than expected according to incidence level. For experiment reference setup see Table 7.3



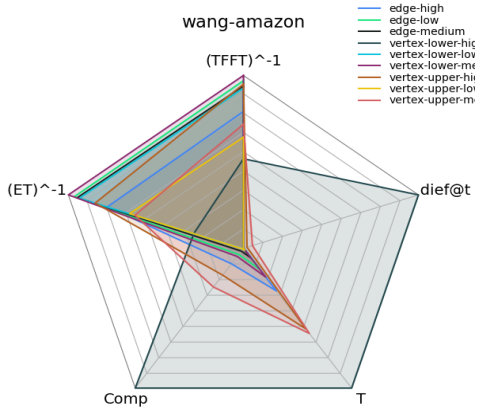
(A) `dief@t` metric results in radial plot after running all experiment scenarios described in Table 7.3 on DBpedia network. Each color represents one scenario. An important thing to remark here is that `Comp` was only completed with 100% by scenario VL-H as it is shown the dark green section. The same happen with `dief@t` which is as well at 100%



(B) `dief@t` metric results in radial plot after running all experiment scenarios described in Table 7.3 on Moreno Crime network. Each color represents one scenario. Here we can observe the same as dbpedia radial plot. The only scenario that completes and shows `dief@t` both at 100% is VL-H. Also E-H has the better `T`



(C) `dief@t` metric results in radial plot after running all experiment scenarios described in Table 7.3 on Opsahl UC Forum network. Each color represents one scenario. In this case VU-H has the better `Comp` and `T` with a 100%, but the only showing a 100% in `dief@t` is VL-H



(D) `dief@t` metric results in radial plot after running all experiment scenarios described in Table 7.3 on Wang Amazon network. Each color represents one scenario. For this case `Comp`, `T` and `dief@t` with 100% is also for VL-H scenario.

FIGURE 7.3: Radial plots show how the different dimensions values provided by `diefpy` tool such as T, TFFT, `dief@t`, ET and Comp are related each other for each experimental case. These figures show radial plot observed results after running all the scenarios described in Table 7.3 for each network. `dief@t` is described in section 2.4.

the VL-H scenario cover `dief@t` metrics with the dark green area, which indicates that it is continuously delivering results for each network. The rest of the data setup experiments indicates that the level of throughput, completeness, and execution time is less than Diefficiency Metric `dief@t`, and the results can be delivered faster without appreciating continuous behavior properly in the plot. These radial plots obtained from `dief@t` shows how ET, TFFT, Comp, T and `dief@t` measured by `diefpy` tool, relate each other in the same setup. For our case analysis, the higher the area that is cover in Diefficiency Metric `dief@t` the better, indicating a high level of continuous behavior.

7.2.2 E2: Benchmark Analysis

Goal Regarding benchmarking, we aim at answering research question [R2] and assess the impact of the type of command query Q on the BT enumeration. The results of this evaluation will also provide evidence to answer [R3] since we have provided evidence with the benchmarking that the execution time varies depending on the command query q proving that we are effectively implemented a *pay-as-you-go* model.

Procedure This experiment measures *Average Running Time* and *Total Running Time* as described in subsection 7.1.2. For gathering *Average Running Time*, the experiment scenarios described in Table 7.5 were conducted; the *Dbpedia* network was not considered, but the `criterion` [37] benchmark tool were followed. The command run for `criterion` benchmark analysis is `stack exec benchmark`.

Furthermore, *Total Running Time* is gathered with the time measured after running all the scenarios described in Table 7.3. Also the timeout configured for all the scenarios in Table 7.5 have been setup in 48 hours.

Observed Results As it can be seen in Figure 7.4, yellow bars are all the experiments related to Lower Layer vertices, blue bars are related to Upper Layer, and red bars are the experiments related to edge search. The longest bars are from network opsahl-ucforum, which we already know by Table 7.1 it is the biggest of the four networks in terms of the number of BT and wedges. So, it needs to enumerate more BT.

Network	Scenario ID	Query
Moreno Crime	E-H	`by-edge (413, 419)`
	E-L	`by-edge (361, 19)`
	E-M	`by-edge (531, 196)`
	VL-H	`by-vertex 95`
	VL-L	`by-vertex 187`
	VL-M	`by-vertex 97`
	VU-H	`by-vertex 2`
	VU-L	`by-vertex 793`
	VU-M	`by-vertex 533`
Opsahl UC Forum	E-H	`by-edge (213, 33)`
	E-L	`by-edge (398, 10)`
	E-M	`by-edge (129, 171)`
	VL-H	`by-vertex 289`
	VL-L	`by-vertex 258`
	VL-M	`by-vertex 433`
	VU-H	`by-vertex 395`
	VU-L	`by-vertex 390`
	VU-M	`by-vertex 207`
Wang Amazon	E-H	`by-edge (839, 9)`
	E-L	`by-edge (10987, 36)`
	E-M	`by-edge (19630, 84)`
	VL-H	`by-vertex 124`
	VL-L	`by-vertex 321`
	VL-M	`by-vertex 64`
	VU-H	`by-vertex 1727`
	VU-L	`by-vertex 9970`
	VU-M	`by-vertex 73`

TABLE 7.5: This table shows all the different experiments setups that we have conducted for E2. Execution command is the same for each experiment because it is handled by `criterion` tool. On the last column we can see for each scenario the query command executed for BT enumeration (see Definition 6.9), although the output is not used in this case, because we focus on average running time. The timeout configured for all this setups were 48 hours.

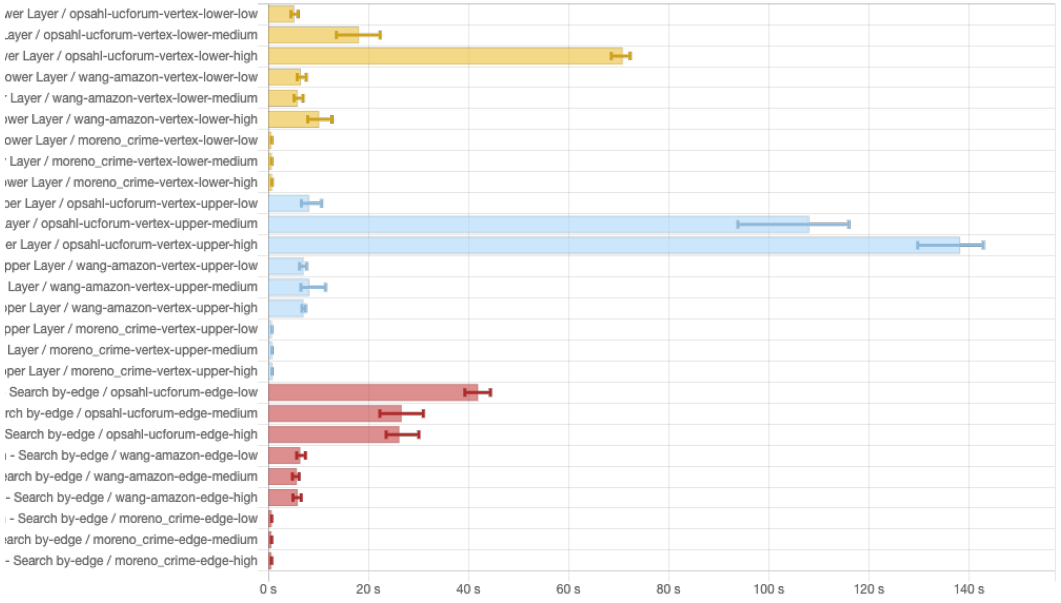


FIGURE 7.4: This plot depicts the results after running `criterion` the tool over the experimental setup described in Table 7.5. Yellow bars are the experiments over Opsahl UC Forum network. Blue light bars represents the experiments on Moreno Crime network, and red bars on Wang Amazon

In Table 7.6 we can see both the *Average Execution Time* and the *Standard deviation* of that average. As we can see and according to this data, all the scenarios with higher incidence are taking longer time in executing compared with the rest of the scenarios for the same network. *Average Execution Time* for Moreno Crime network is reporting 723, 745 and 655 milliseconds average execution time for high incidence (see subsection 7.1.3), compared with lower and medium incidence. The same behavior is shown for Wang Amazon which high incidence scenarios report 10.2 and 8.16 seconds on VL-H and VU-H respectively. Moreover, Opsahl UC Forum reports VL-H and VU-H with 70.8 and 138 seconds of average running time which is higher than other scenarios with lower incidence. The only cases that this behavior do not hold is for E-H in both Wang Amazon and Opsahl UC Forum networks. E-L scenario shows the highest value for both networks with 6.4 and 41.9 seconds respectively.

Regarding Figure 7.5 which is showing the *Total Running Time*, it is observed with the red bars that Dbpedia was the one which took more time in all scenarios. We can also see in Figure 7.5 that scenario E-H took 8756 seconds in Dbpedia network compare with the E-L and E-M, which took 596 seconds and 577 seconds, respectively. Also, in Figure 7.5, we can see that Opsahl

Network	Scenario ID	Average Execution Time	Standard deviation
Moreno Crime	VL-L	656 ms	13.0 ms
	VL-M	669 ms	18.7 ms
	VL-H	723 ms	72.2 ms
	VU-L	695 ms	16.2 ms
	VU-M	726 ms	14.0 ms
	VU-H	745 ms	19.6 ms
	E-L	694 ms	4.7 ms
	E-M	655 ms	14.9 ms
	E-H	655 ms	21.1 ms
Opsahl UC Forum	VL-L	5.6 s	940 ms
	VL-M	18.1 s	5.03 s
	VL-H	70.8 s	2.03 s
	VU-L	8.12 s	2.43 s
	VU-M	108 s	13.7 s
	VU-H	138 s	8.19 s
	E-L	41.9 s	2.87 s
	E-M	26.7 s	5.63 s
	E-H	26.2 s	4.05 s
Wang Amazon	VL-L	6.5 s	1.05 s
	VL-M	5.91 s	994 ms
	VL-H	10.2 s	2.66 s
	VU-L	8.12 s	2.43 s
	VU-M	7.04 s	739 ms
	VU-H	8.16 s	3.25 s
	E-L	6.4 s	1.08 s
	E-M	5.71 s	863 ms
	E-H	5.91 s	1.04 s

TABLE 7.6: This tables shows a detailed reported data of all the average execution time of all the networks and all scenarios except Dbpedia networks. It is also show the standard deviation of all average execution time. The measurement unit indicated in the table are: *s* for seconds, and *ms* for milliseconds. Highlighted blue lines indicates scenarios that are behaving according to the incidence level. Yellow highlighted lines are scenarios that reports values lower than expected according to incidence level.

UC Forum took 1660 seconds for VL-H scenario, 11 seconds for VL-L, and 22 seconds for VL-M, also showing the same behavior as Dbpedia. Moreover, in Figure 7.5 it can also be appreciated how Wang Amazon network and Moreno Crime are showing the same behavior for VL-H, VL-L, and VL-M scenarios, where Wang Amazon networks report $21,8$ and 8 seconds, respectively and Moreno Crime $3,3$ and 2 seconds. The same behavior repeats for all networks and all scenarios, indicating that the experiments with higher incidence take more time to finalize the execution compared to the experiments with lower incidence. This means that the type of query command Q impacts on the execution of the program; the higher the incidence, the more bitriangles to be enumerated and the longest the program will take to finish.

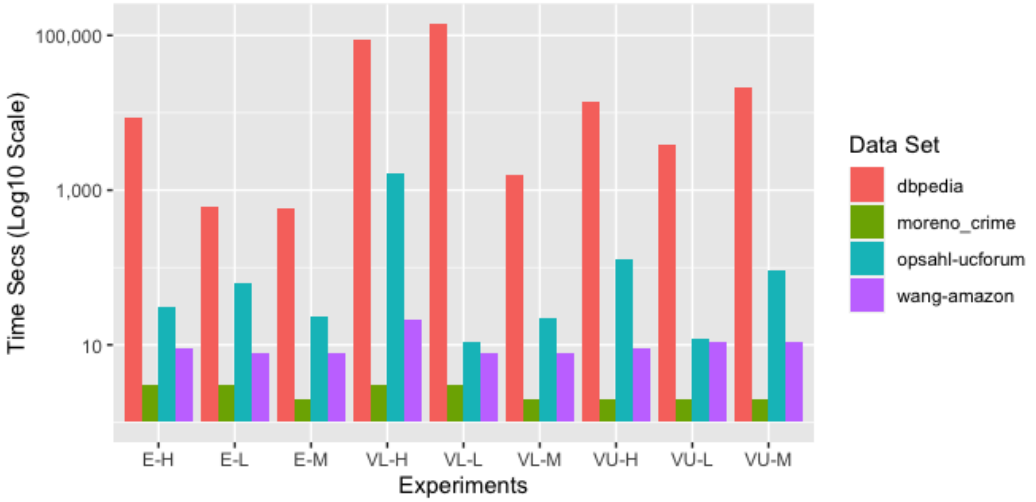


FIGURE 7.5: This plot shows all the experiment scenarios run in Table 7.3 and the comparison of the Total running execution time for each case. y axis shows the total time in ln scale and x axis is each Scenario ID describe in Table 7.2

7.2.3 E3: Performance Analysis

Goal In this experiment, we take measurements on one network, to gather data about the use of memory allocation and threads on GHC during the execution of DP-BT-Haskell.

Procedure This experiment gathers three metrics describe in subsection 7.1.2: *GHC Productivity*, *Distribution of Threads per Core* and *Distribution of Allocated Memory per Data Type*. Using `ThreadScope` [39] tool we measure *GHC Productivity*, *Distribution of Threads per Core*, and using `eventlog2html` [41] tool, we measure *Distribution of Allocated Memory per Data Type*. For both cases, we have run these tools using Dbpedia network only and VU-L scenario. The query command executed was `by-vertex 93` and we setup a timeout of 24 hours. This is because both tools enable profiling flags at compilation level on GHC, penalizing performance.

As observed in Table 7.7, although we have selected the biggest network, we are only running the tools to gather data using VU-L, which enumerate less BT. Therefore, this allows for the retrieval of profiling information, such as multithreading details and memory allocation.

Tool	Exec Flags
`ThreadScope`	`+RTS -A10G -H10G -c -N8 -l -s -RTS`
`eventlog2html`	`+RTS -A10G -H10G -c -N8 -l -s -RTS`

TABLE 7.7: This table shows the experiments scenario run for each of the tools. Notice the increase of `-A` and `-H` to support more memory allocation due to the profiling analysis

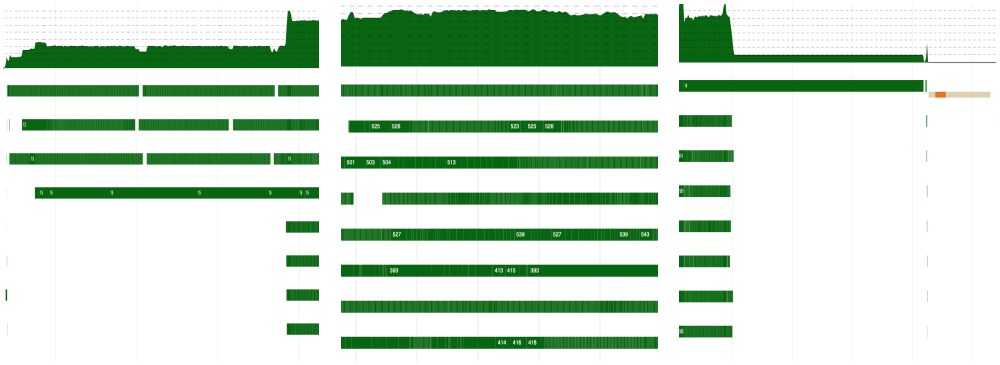
Observed Results We can divide the analysis of this section into two: memory consumption and multithreading.

Multithreading Regarding multithreading, we have gathered multithreading metrics of different time slots of the total execution time of Moreno Crime network run. We could not analyze bigger networks due to the huge amount of data gathered that make the program timeout for this experiment. The timeout on this case was set on 24 hours and the program timed out running out of memory. As we can see in the overview, execution in Figure 7.6 all the cores (8) are running Mutator time in threads almost during the whole execution of the program. Running Mutator most of the execution of the program, also indicates that there are few GC pauses, and running time is overtaken by MUT time and not GC. In fact, GHC productivity on this run indicated 99.8%.

ThreadScope [39] output allows us to zoom in different portions of the execution time to analyze the results better. If we zoom in on the execution threads at the beginning and at the end, we are going to see that there is a moment when only one core is executing. At the middle of the execution, we are seeing more processing distributing evenly among cores with less use GC and higher MUT time.

Memory Consumption In the case of memory consumption, we have been able to measure the memory consumption for the biggest graph, Dbpedia. As it is known, enabling profiling downgrades the performance of execution time. Because of that, the program runs out of memory after 24 hs., as we are going to see in the image. Although this, we have still been able to gather memory allocation data to conduct the analysis.

FIGURE 7.6: This is a general overview of ThreadScope results over the experiments running. Green bar indicates MUT time. The distribution between different 8 green bars means that it is executing on the 8 assigned cores.



(A) [EE] Thread Metrics: Initial execution (B) [EE] Thread Metrics: Middle of execution (C) [EE] Thread Metrics: End of execution

FIGURE 7.7: These plots depict different moments of the execution of the program after doing a zoom of the images allowing to scale down execution time view. The left image indicates the beginning of the execution. Middle image is the middle of the execution and right image is the end of the execution.

As we can appreciate in Figure 7.8 the darkest blue area belongs to `MUT_ARR_PTRS_CLEAN`. These types of objects are pointers to function. It is clear that `MUT_ARR_PTRS_CLEAN` is allocating $2G$ at most, and it is more than the rest of the objects. This also represents more than 50% of the accumulated memory allocation during the

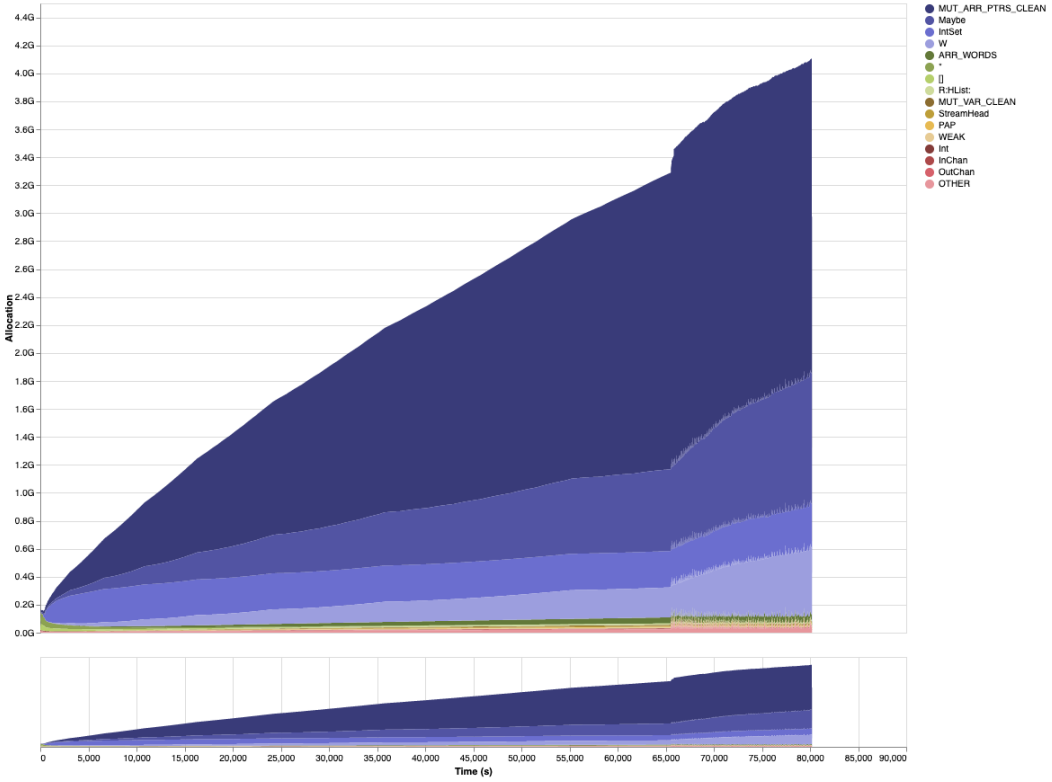


FIGURE 7.8: This plot is showing the accumulated memory allocation size of each Haskell Data Type throughout the execution of the program.

whole execution time. Below dark blue `MUT_ARR_PTRS_CLEAN`, there is a lighter blue area, which represents the allocation of `Maybe` type. `Maybe` is the type that transfers data between stages through channels. It also represents the 25% of the total allocated memory with less than $1G$ in total. Continue below `Maybe` type, it is `IntSet` data type memory allocation, which is the type for storing the ADW and ABT as we have described on section 6.1. This is even lower than the previous one, allocating up to $0.3G$ of memory that does not represent more than 7% of the total memory. Alongside this, it is the `W` type which is storing AW; it represents $0.5G$ which is a 10% of the total allocation. The rest 10% of the total memory is evenly distributed among other data types that are not part of the specific DPF-Haskell implementation, but from Haskell in general.

7.3 Discussion

E1: Continuous behavior section 7.2.1 reports show that continuous behavior can be appreciated better in some scenarios and networks than others. For example, the scenarios containing queries with higher incidence shows more continuous behavior than the rest as we have seen in Table 7.4. The reason of higher incidence are more continuous than the rest can be explained because bitriangles are aggregated based on triple $\ell = (l_l, l_m, l_u)$ (see Definition 6.8). Then, if the requested $l \in L$ matches with some of these vertices in ℓ, all the $\hat{U}_l$ cartesian product need to be enumerated in a 6-cycle path BT (see Definition 6.2). Therefore, if there are a large number of BT because the incidence level is high, there are more data to be delivered through channels and more data is arriving to $\mathsf{Sk}_{\mathsf{BT}}$ sooner, incrementally generating results (see section 6.2). Moreover, the cases that exhibit less continuous behavior are Wang Amazon and Moreno Crime networks in the majority of their scenarios. The reason for less continuity behavior on those networks could be explained by the fact of the topology of both graphs. Wang Amazon and Moreno Crime networks are the smallest of all the graphs used, in terms of the number of BT as we can see in Table 7.1. Therefore, results are delivered extremely fast and incremental results only can be appreciated in the vertices with high incidence. Another interest discussion point that we can provide based on the results is that there are some scenarios with higher incidence level that are showing less continuous behavior compared with scenarios of the same networks with smaller incidence, as we have discussed in section 7.2.1. This behavior repeats in all networks with some scenario but lets take an example, Opsahl UC Forum in Edges Scenarios. In this network we can see that E-H has `dief@t` value lower than E-L. The same happens in Dbpedia and Moreno Crime for VU-H compared with VU-M where VU-H has smaller `dief@t` value than VU-M. Also Wang Amazon exhibit the same on VU-H compared with VU-M. The explanation on why some higher incidence scenarios reports lower `dief@t` values compare with smaller incidence, it is due to the fact of the *pseudo-random* selection mechanism of the command Q value (vertex or edge) – as it is commented in section 7.1.3. We have detected that, even when the *pseudo-randomly* chosen vertex or edge has a high degree, it is not a vertex with a high incidence as it should be – not participating in the majority of bitriangles –, according to the Definition 7.1. In conclusion, we are able to answer [R1] and asses that we have built an incremental algorithm

for enumerating Bitriangle. The same conclusion can be obtained regarding question [R3]. We verify that, depending on the incidence of the vertex or edge, DP-BT-Haskell is enumerating BT in a continuous manner. This shows us DP-BT-Haskell effectively implements a *pay-as-you-go* model.

E2: Benchmark Regarding section 7.2.2 on the one hand, we have noticed that for the case of edges with high incidence scenarios in both Opshal UC Forum and Wang Amazon networks, the execution time is not consistent with what we expected regarding the incidence level. This could be explained by the *pseudo-randomly* selection detailed in section 7.1.3 which is outlying the samples. By outlying we mean that we select *randomly* from the subset with higher vertex degree assuming that this will enumerate more bitriangles, but in the experimentation has been shown it has not. On the other hand, all the rest of the scenarios reports *Average Execution Time* according to the level of incidence of the experiment, the higher the incidence the longest the average execution time. In what relates to *Total Execution time*, we have pointed out that DBpedia network is the one taking longer. This is perfectly explained by the characteristics and topology of the graph, since it is both the biggest graph in terms of edges and vertex, and it is the graph which proportionally has more BT. We can answer the question [R2] because it can clearly be seen in the benchmark analysis and in Figure 7.5 that as long as the user request for command queries Q that have more incidence in the graph and participates in more BT, the execution time increases.

E3: Performance In section 7.2.3 we have noticed the suitable distribution of threads among cores during the execution time. That exhibits a perfect fit with the DPP model since at the beginning, it considers all the filters and starts reading the input file, where there is less distribution of threads among the core. After that, we can see an even distribution and all cores busy when DPP executes the filters. Remember that each stage runs on its own thread. At the end, we observe a reduction in the distribution of the cores, because the last part of the execution is done by $\mathsf{Sk}_{\mathsf{BT}}$. Regarding memory allocation, we have seen that most of the allocation is done by `MUT_ARR_PTRS_CLEANER` data type. In Haskell Programming Language MUT is the acronym of a thread evaluating an Haskell expression. Therefore, having $2G$ of allocated memory for `MUT_ARR_PTRS_CLEANER` type means that there are many pointers allocated

waiting for evaluating expressions. DP-BT-Haskell implementation explains this behavior because it is spawning one thread per stage, and in particular, that means one thread per filter instance as well. In the case of Dbpedia which contains 168.338 vertices in L according to Table 7.1, in the worst case DP-BT-Haskell is spawning the same amount of threads for every run of this network. Since the execution of all these stages will not be released until it finishes the last actor_4 (processing queries), all of them are waiting for the queries to be processed and executed. Another important part of the allocated memory as we have seen in subsection 7.2.3 is distributed between `Maybe`, `IntSet` and `W` data types. This behavior is expected because `Maybe` carries data between stages which cannot be avoided due to the DPP model. Additionally, both `IntSet` and `W` are the compressed form DP-BT-Haskell uses for storing intermediate structures to build BT, as we have described in section 6.1. One of the proposed solutions for future work is to reduce the number of $\mathsf{F_{BT}}$ for bigger graphs in order to reduce the number of allocated pointers waiting for commands. Although memory allocation shows a linear growth in Figure 7.8 for the `MUT_ARR_PTRS_CLEANER` type in this experiment, the rest of the memory allocation is not growing linearly, as we can see also in Figure 7.8. This is a key factor on memory analysis since it is showing how DPF-Haskell is compressing intermediate objects like AW, ADW, and ABT without penalizing the rest. In conclusion, we can answer the question [R4] as we have shown that threads are efficiently handled by Haskell GHC scheduler supporting the parallelization level that DPP requires. We can also state that memory management is efficiently handled as well by the analysis exposed before. Additionally, memory allocation can be also improved by implementing a better matching algorithm for searching and deliver BT like for example the ones describe by Lai et al. [29]. It was out of the scope of this work to solve the efficiency of the queries, as well as the underlying data structures improvements.

7.4 Chapter Summary

In this chapter, we have explained the experiments conducted in order to answer our research question. We first started presenting a summary of the research questions. After that, we have described the experimental configuration. Then, the experimental procedure and results are reported. Finally,

we have discussed the observed outcomes in terms of the main properties of our proposed approach. To summarize, the main points observed during the experimentation are:

- High values of the metric dieft that is indicating the continuous behavior of DP-BT-Haskell exhibiting its capability of incrementally enumerates BT.
- Lower values of the metric diefk that is indicating the continuous behavior of DP-BT-Haskell exhibiting its capability of incrementally enumerates BT.
- High values of the metrics *Average Running Time* and *Total Running Time* for scenarios which enumerates more bitriangles, are suggesting an effective implementation of a *pay-as-you-go* model of DP-BT-Haskell.
- Results captured by the `ThreadScope` tool indicating an even distribution of the threads among processors, showing efficient use of the parallel model.
- Results gathered by the `eventlog2html` tool suggesting that memory consumption is efficiently handled in the intermediate objects that DPF-Haskell collects on each filter and transfers between them. A proposal on how also improve this part was exposed in section 7.3.

Chapter 8

Conclusions and Future Work

In this chapter we present the conclusions of our work. We also present some observed limitations and improvements to our proposal. Finally, we discuss some research lines we consider can be follow taking as starting point our contributions.

8.1 Conclusions

Enumerating bitriangles in large bipartite graphs is a significant problem that helps to detect important relations between entities in diverse areas. Some examples are pharmacology research, to establish relations between drugs and side effects; medicine and biology, to detect how some genes affect specific diseases; the entertainment industry, to relate user preferences and TV shows, allowing for certain predictions in that matter, etc. In general, any area requiring linking data could take advantage of this kind of enumeration of bitriangles in large bipartite networks.

Besides, introducing algorithms that incrementally deliver results –using a *pay-as-you-go* model approach combined with DPP – brings users the possibility of obtaining results without being exhaustive exploring the solution space. Moreover, users can obtain results without consuming resources unnecessarily. We have explored Dynamic Pipeline Paradigm as a model of computation to solve that problem using the *pay-as-you-go* approach.

We have introduced, implemented and empirically evaluated an Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network under the Dynamic Pipeline Paradigm. To reach the main objective of this work, we have combined different techniques: (i) the creation of an index graph to support the querying process based on the representation of compact structures defined in chapter 6, (ii) the definition and implementation of an algorithm under the DPP, using parallel Haskell and, (iii) the measuring of the its implementation using the Diefficiency Metrics. Putting all these features together gives insights about the high technical level we have dealt with. Finally but no less important, we implemented and left available for the community a generic Dynamic Pipeline Framework implemented in Haskell.

We think the design and implementation of algorithms delivering results incrementally and the possibility of measuring its continuous efficiency as we have done for the IEBT, is also a stimulus for tackling some other similar problems. In the experimental analysis, we showed the continuous behavior capabilities of the implementation using, Diefficiency Metric `dief@t` giving support to our research assumptions and motivations. The designed algorithm DP-BT-Haskell can process and enumerate large networks like the Dbpedia Network which contains more than 300 millions of bitriangles.

We have shown that Haskell Programming Language suits well for implementing a dynamic pipeline for solving Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network. After empirically assessing the implementation of the IEBT we think the suitability of the Dynamic Pipeline Paradigm as an effective computational model for deployment a parallel algorithm for incrementally enumerate Bitriangle in Bipartite Graph has been demonstrated. In particular, one important motivation to develop our own framework is that we not only wanted to satisfy our research needs but, as a novel contribution, we wanted to deliver a DPF to the Haskell community as well. We hope this contribution encourages and helps writing algorithms under the Dynamic Pipeline Paradigm.

To finish, we think the achieved results clearly show we have implemented a suitable *Algorithm for Incrementally Enumerating Bitriangles in Large Bipartite Network*, opening a wide range of possibilities not only to improve the existing framework and algorithm but also for tackling, using this approach, other complex problems.

8.2 Observed Limitations

We have obtained positive results during the experimental analysis, however we have detected some weaknesses of the implementation. These weakness can be addressed in the future to improve the solution. The first limitation, exposed in chapter 7, is memory consumption efficiency. Although the part of the memory allocation that belongs purely to the IEBT specificities are showing an acceptable use of the resources, there is still a problem with the number of filters that are being spawned and not free by GHC. This is causing a big increase of `MUT_ARR_PTRS_CLEANER` object. This limitation imposes that if we want to handle networks bigger than DBpedia it is necessary to worry about the amount of memory that we have at our disposal.

Another weakness also related to memory management, is the data structures used for managing the queries. In the DPF-Haskell implementation of the pseudo-code algorithm as we have seen on section 6.5 we have been careful about Haskell techniques and data types used to improve the search performance. However, it is important to remark that there are more advanced techniques to implement this kind of joins and querying process [29]. Finally, there are currently some limitations on Haskell Dynamic Pipeline Framework itself, since the threading model we are using is the *green threads* [62]. This threading model provided by Haskell has a limitation in the number of threads that the framework can spawn. There is other threading model also supported by Haskell, *sparks*, which potentially allows hundreds of millions of threads.

8.3 Future Work

Regarding complexity of the IEBT, the formal analysis is beyond the scope of this work. We left the realization of a complete and proper analysis for future work. In particular, to get a robust complexity analysis of the IEBT must consider many statistical parameters of the bipartite networks. Additionally, the parallelism of the implementation imposes the use of specific techniques for this family of algorithms. This is one of the most challenging task to be tackled after this work.

From the point of view of our algorithmic contribution under the DPP and using a *pay-as-you-go* approach, we think there are two main research lines.

On the one hand, considering the general problem of discovering motifs, and some related problems, in large graphs for linking data, it is interesting to study how the kind of compact representation we use to create a graph index could improve some similar discovering problems. On the other hand, showing the effectiveness of implementing graph algorithm under the DPP encourages the creation of a benchmark for many well-known graph problems. This is now really feasible since this work provides to the research community availability of a Dynamic Pipeline Framework.

Regarding implementation issues, the future work is mainly oriented to address the efficiency and the capability of the system handle bipartite networks larger than DBpedia. As we have seen in chapter 7, we have detected an increase in the memory consumption as long as the network grows. Regarding that, we think it is extremely important to improve this aspect in order to be able to process huge networks efficiently.

Moreover, we have seen evidence of the previous statement in section 7.2.3, where it is exhibited that execution of larger graphs requires paying an extra cost. The selection of appropriate data structures for optimizing handling the querying answering process over aggregated bitriangles Aggregated Bitriangle is an issue that needs to be tackled in the future.

Performing indexing over the edges and vertices could lead to significant improvements in search, although there is a trade-off in terms of memory allocation, that can be addressed using fast external storage. Another important aspect to be addressed in the future is to make DP_{BT} in Haskell distributed and parallel. Distribution of the stages between machines and not only between threads would enable sharing memory-intensive allocation between machines and not using a single memory unit for all the threads. We think that moving from a multithreading model to a distributed model will allow Dynamic Pipeline Paradigm to reduce the gap of memory consumption for huge network instances, as well as distribute the stage according to the amount of memory required by them. We cannot avoid that there is a trade-off in terms of transfer data and the delay that this kind of distributed computation brings in, but since DP_{BT} in Haskell is delivering incremental results in a *pay-as-you-go* model, the distribution cost could be amortized. We should also take into consideration that the speed-up and reliance on network communication is extremely high nowadays, enabling the exploration of the commented approach.

In another aspect of the computational model and more related with the Haskell Programming Language implementation, there are several improvements to be conducted in the Haskell Dynamic Pipeline Framework as well. One of those improvements could be to delegate the distribution to other stream processing systems like `Kafka` [78] for example and do the parallel processing of the split data with Haskell Programming Language acting as a consumer. In addition to that, some other radical improvements can also be done into Haskell Dynamic Pipeline Framework. From designing more abstractions to help the user to write pipelines with less effort and errors, to modifying the thread scheduler and memory management in GHC to improve performance. We can also improve the speed-up in Haskell Dynamic Pipeline Framework moving some Boxed types to Unboxed types [79] which would reduce memory footprint.

As we can appreciate, this work gives rise to interesting research lines and improvements that are worth to be explored.

Appendix A

Appendix

A.1 Source Code

All the source code of this research project can be found in `https://github.com/jproyo/upc-miri-tfm`, and it is publicly available for download. In that source code there are three folders:

- **connected-comp**: This contains the Haskell source code done for the chapter 4 contribution related to Weak Connected Components (WCC) using DPP.
- **bt-graph-dp**: This contains the Haskell source code done for the specific problem of this work which is incremental enumeration of Bitriangle in Bipartite Graph.
- **doc**: Contains this document in LaTex format as well as the paper written for the proof of concept [35].

A.2 Running Experiments

All the scripts and data for running the experiments are under `bt-graph-dp/experiments` folder. It is important to mention that we are not including in the source code distribution the networks themselves because you can search them on Konect [68] by the reference in this work. We are going to describe how to run the different experiments exposed on chapter 7.

E1 In this case we have different experiments setups and we are going to describe how to run one case only. For example for running the experiment setup $E-H$ on Dbpedia graph, assuming that you download the graph and call the file as `input.txt` and it is inside of folder `bt-graph-dp/experiments/diepfy/dbpedia`.

```
>>> cd bt-graph-dp
>>> stack build
>>> stack exec bt-graph-dp -- +RTS -A1G -H1G -N6 -c -RTS -f
↪  ./experiments/diepfy/dbpedia/input.txt -c
↪  ./experiments/diepfy/dbpedia/c-edge-high.txt -e dbpedia
```

E2 In the case of benchmark analysis it is simple the following command

```
>>> cd bt-graph-dp
>>> stack build
>>> stack exec benchmark
```

This is going to left the results in HTML file format under `benchmark`.

E3 In the case of Memory and Thead Measurement you need to enable profiling flags.

For Memory

```
>>> cd bt-graph-dp
>>> stack build --profile
>>> stack exec bt-graph-dp -- +RTS -A10G -H10G -c -N12 -hy
↪  -l-agu -RTS -f ./experiments/diepfy/moreno_crime/input.txt
↪  -c ./experiments/diepfy/moreno_crime/c-edge-high.txt -e
↪  moreno_crime
```

For ThreadScope

```
>>> cd bt-graph-dp
>>> stack build --profile
>>> stack exec bt-graph-dp -- +RTS -A10G -H10G -c -N12 -l -s
↪  -RTS -f ./experiments/diepfy/moreno_crime/input.txt -c
↪  ./experiments/diepfy/moreno_crime/c-edge-high.txt -e
↪  moreno_crime
```

A.3 Diefficiency Metrics - Traces

```
test,approach,tfft,totaltime,comp
dbpedia,edge-high,255736.822265625,8485928.7265625,214038
dbpedia,edge-low,224763.88452148438,389846.43212890625,3880
dbpedia,edge-medium,147658.31372070313,372466.23828125,19534
dbpedia,vertex-lower-high,251050.91967773438,86167082.25341797,4143936
dbpedia,vertex-lower-low,216835.52563476563,239741.81201171875,188
dbpedia,vertex-lower-medium,199219.95922851563,1318776.70703125,47714
dbpedia,vertex-upper-high,225575.84912109375,13363175.830078125,419107
dbpedia,vertex-upper-low,181239.228515625,3579096.3217773438,324662
dbpedia,vertex-upper-medium,250305.59741210938,21314163.650390625,1167826
```

Source Code A.20: CSV file that contains the metrics for all the scenarios of Dbpedia network that feed `diefpy` tool in order to obtain Diefficiency Metrics. This file contains the minimum and maximum t where an answer was produced for each scenario

```
test,approach,answer,time
dbpedia,edge-high,1,255736.822265625
dbpedia,edge-high,2,255737.4765625
dbpedia,edge-high,3,255737.59545898438
..
..
dbpedia,vertex-upper-medium,1167666,21313172.78540039
dbpedia,vertex-upper-medium,1167667,21313172.837890625
dbpedia,vertex-upper-medium,1167668,21313172.888183594
dbpedia,vertex-upper-medium,1167669,21313262.267333984
..
..
```

Source Code A.21: CSV file that contains all data points where an answer was generated for all the scenarios of Dbpedia network that feed `diefpy` tool in order to obtain Diefficiency Metrics. This file contains the scenario, the answer number, and the time t where that answer was produced

Bibliography

[1] Kwang-Il Goh, Michael E Cusick, David Valle, Barton Childs, Marc Vidal, and Albert-László Barabási. The human disease network. *Proceedings of the National Academy of Sciences*, 104(21):8685–8690, 2007.

[2] Mohan Timilsina, Meera Tandan, Mathieu d'Aquin, and Haixuan Yang. Discovering links between side effects and drugs using a diffusion based method. *Scientific Reports*, 9:10436, 07 2019. doi: 10.1038/s41598-019-46939-6.

[3] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9, 01 2005. doi: 10.7155/jgaa.00108.

[4] Jesper Nederlof. Detecting and counting small patterns in planar graphs in subexponential parameterized time, 2019.

[5] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, Jan 2003. ISSN 1095-7200. doi: 10.1137/s003614450342480. URL `http://dx.doi.org/10.1137/S003614450342480`.

[6] Matthieu Latapy, Clémence Magnien, and Nathalie Del Vecchio. Basic notions for the analysis of large two-mode networks. *Social networks*, 30 (1):31–48, 2008.

[7] Tore Opsahl. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients, 2011.

[8] Yixing Yang, Yixiang Fang, Maria E. Orlowska, Wenjie Zhang, and Xuemin Lin. Efficient bi-triangle counting for large bipartite networks. *Proc. VLDB Endow.*, 14(6):984–996, February 2021. ISSN 2150-8097.

doi: 10.14778/3447689.3447702. URL `https://doi.org/10.14778/3447689.3447702`.

[9] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[10] Jérôme Kunegis, Ernesto W De Luca, and Sahin Albayrak. The link prediction problem in bipartite networks. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems*, pages 380–389. Springer, 2010.

[11] Yike Guo, Moustafa Ghanem, and Rui Han. Does the cloud need new algorithms? an introduction to elastic algorithms. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 66–73. IEEE, 2012.

[12] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 62–73, 2013. doi: 10.1109/ICDE.2013.6544814. URL `http://dx.doi.org/10.1109/ICDE.2013.6544814`.

[13] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[14] I Lee, Ting Angelina, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17, 2015.

[15] C. Zoltán and et al. The dynamic pipeline paradigm. In *Jornadas sobre Programación y Lenguajes. Actas de las XIX Jornadas de Programación y Lenguajes (PROLE 2019): Cáceres, septiembre de 2019*, pages 1–11, San Sebastián, Guipúzcoa: Sociedad de Ingeniería de Software y Tecnologías de Desarrollo de Software (SISTEDES), 2019.

[16] Juan Pablo Royo Sales. Haskell dynamic pipeline library. `https://hackage.haskell.org/package/dynamic-pipeline`. Accessed: 2021-09-07.

[17] haskell.org. Hackage: The haskell package repository. `https://hackage.haskell.org`. Accessed: 2021-05-09.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `https://doi.org/10.1145/1327452.1327492`.

[19] Apache Foundation. Apache hadoop. `https://hadoop.apache.org/`. Accessed: 2021-04-30.

[20] Apache Foundation. Apache spark. `https://spark.apache.org/`. Accessed: 2021-04-27.

[21] Henriette Röger and Ruben Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52:1 – 37, 2019.

[22] Michael Snoyman. Haskell conduit types. `https://hackage.haskell.org/package/conduit`. Accessed: 2021-05-09.

[23] Gabriel Gonzalez. Haskell pipes library. `https://hackage.haskell.org/package/pipes`. Accessed: 2021-05-09.

[24] Harendra Kumar. Haskell streamly types. `https://hackage.haskell.org/package/streamly`. Accessed: 2021-05-09.

[25] Harendra Kumar. Haskell stream libraries benchmarks. `https://github.com/composewell/streaming-benchmarks`. Accessed: 2021-05-09.

[26] Maribel Acosta, Maria-Esther Vidal, and York Sure-Vetter. Diefficiency metrics: Measuring the continuous efficiency of query processing approaches. pages 3–19, 10 2017. ISBN 978-3-319-68203-7. doi: 10.1007/978-3-319-68204-4_1.

[27] Maribel Acosta. Python diefficiency metric tool. `https://github.com/SDM-TIB/diefpy/`. Accessed: 2021-05-03.

[28] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 607–614, New York, NY, USA,

2011. Association for Computing Machinery. ISBN 9781450306324. doi: 10.1145/1963405.1963491. URL https://doi.org/10.1145/1963405.1963491.

[29] Longbin Lai, Wenjie Zhang, Ying Zhang, Zhengping Qian, Jingren Zhou, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, and Lu Qin. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12:1099–1112, 06 2019. doi: 10.14778/3339490.3339494.

[30] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522738. URL https://doi.org/10.1145/2517349.2522738.

[31] Xiaodong Li, T. N. Chan, Reynold Cheng, Caihua Shan, Chenhao Ma, and Kevin Chang. Motif paths: A new approach for analyzing higher-order semantics between graph nodes. 2019.

[32] Tam Nguyen, Matthias Weidlich, Hongzhi Yin, Bolong Zheng, Nguyen Huy, and Nguyen Hung. Factcatch: Incremental pay-as-you-go fact checking with minimal user effort. 05 2020. doi: 10.13140/RG.2.2.27201.58722.

[33] Simon Marlow. Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *CEFP 2011*, volume 7241 of *LNCS*, pages 339–401. O'Reilly Media, Inc., 2012.

[34] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, 2011. ISSN 0362-1340. doi: 10.1145/2096148.2034685.

[35] Juan Pablo Royo Sales, Edelmira Pasarella, Cristina Zoltan, and Maria Esther Vidal. Towards a Dynamic Pipeline Framework implemented in (parallel) Haskell. In N. (Ed.) Martí Oliet, editor, *PROLE2021*. SISTEDES, 2021. URL http://hdl.handle.net/11705/PROLE/2021/017.

[36] Stanford University. Snap datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/index.html. Accessed: 2021-04-15.

[37] Bryan O'Sullivan. Haskell criterion tool. `https://hackage.haskell.org/package/criterion`. Accessed: 2021-04-17.

[38] Haskell.org. Haskell containers library. `https://hackage.haskell.org/package/containers`. Accessed: 2021-04-17.

[39] S. Singh, S. Marlow, D. Jones, D. Coutts, M. Konarski, N. Wu, E. Kow. Haskell threadscope tool. `https://wiki.haskell.org/ThreadScope`. Accessed: 2021-04-17.

[40] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for ghc. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, page 107–118. Association for Computing Machinery, 2007. ISBN 9781595936745. doi: 10.1145/1291201.1291217.

[41] M. Pickering, D. Binder, C. Heiland-Allen. Haskell eventlog2html tool. `https://mpickering.github.io/eventlog2html/`. Accessed: 2021-04-17.

[42] Haskell.org. Haskell ghc-heap types. `https://downloads.haskell.org/~ghc/8.10.4/docs/html/libraries/ghc-heap-8.10.4/GHC-Exts-Heap-ClosureTypes.html`. Accessed: 2021-05-09.

[43] WordPress.com. Wordpress. `https://wordpress.com/`. Accessed: 2021-09-27.

[44] RubyonRails.org. Ruby on rails. `https://rubyonrails.org/`. Accessed: 2021-09-27.

[45] VMware, Inc. or its affiliate. Spring boot. `https://spring.io/projects/spring-boot`. Accessed: 2021-09-27.

[46] Cake Software Foundation, Inc. . Cake php. `https://cakephp.org/`. Accessed: 2021-09-27.

[47] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, 2010. ISBN 978-0-321-71294-3. URL `https://www.safaribooksonline.com/library/view/domain-specific-languages/9780132107549/`.

[48] RedHat. Hibernate query language. `https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`. Accessed: 2021-09-27.

[49] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71, 11 2015. doi: 10.1016/j.infsof.2015.11.001.

[50] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[51] Juan Pablo Royo Sales. Haskell dynamic pipeline library - github repository. `https://github.com/jproyo/dynamic-pipeline`. Accessed: 2021-09-02.

[52] R. Hinze, J. Jeuring, and Andres Löh. Type-indexed data types. In *MPC*, 2002.

[53] E Meijer and Simon Peyton Jones. *Henk: a typed intermediate language*. Types in compilation edition, May 1997. URL `https://www.microsoft.com/en-us/research/publication/henk-a-typed-intermediate-language/`.

[54] Coq Consortium. The coq proof asistant. https://coq.inria.fr/, 2021.

[55] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, September 2005. ISSN 0362-1340. doi: 10.1145/1090189.1086397. URL `https://doi.org/10.1145/1090189.1086397`.

[56] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 53–66, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311205. doi: 10.1145/2103786.2103795. URL `https://doi.org/10.1145/2103786.2103795`.

[57] Olivier Danvy and Lasse Nielsen. Defunctionalization at work. volume 8, pages 162–174, 06 2001. doi: 10.1145/773184.773202.

[58] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions, April 2009. URL `https://www.microsoft.com/en-us/research/publication/fun-type-functions/`. Presented at Tony Hoare's 75th birthday celebration, Cambridge, 17 April 2009.

[59] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, Haskell'04*, pages 96–107, 01 2004. doi: 10.1145/1017472.1017488.

[60] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47599-6.

[61] S. Marlow, R. Newton. Haskell monad-par library. `https://hackage.haskell.org/package/monad-par`. Accessed: 2021-05-11.

[62] Glasgow Haskell Compiler. Glasgow haskell compiler user's guide. `https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/parallel.html#annotating-pure-code-for-parallelism`. Accessed: 2021-04-28.

[63] Haskell.org. Haskell base library. `https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Concurrent.html#v:forkIO`. Accessed: 2021-04-26.

[64] Simon Marlow. Haskell async library. `https://hackage.haskell.org/package/async`. Accessed: 2021-04-17.

[65] Tim Harris, Simon Marlow, and Simon Peyton Jones. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM Press, January 2005. ISBN 1-59593-080-9. URL `https://www.microsoft.com/en-us/research/publication/composable-memory-transactions/`.

[66] Brandon Simmons. Haskell unagi-chan library. `https://hackage.haskell.org/package/unagi-chan`. Accessed: 2021-04-17.

[67] Brandon Simmons. Haskell unagi-chan benchmark. `https://github.com/jberryman/unagi-chan`. Accessed: 2021-04-17.

[68] Jérôme Kunegis. The konect project. http://konect.cc/networks/, 2021.

[69] Record labels network dataset – KONECT, October 2017. URL `http://konect.cc/networks/dbpedia-recordlabel`.

[70] Crime network dataset – KONECT, October 2017. URL `http://konect.cc/networks/moreno_crime`.

[71] Uc irvine forum network dataset – KONECT, October 2017. URL `http://konect.cc/networks/opsahl-ucforum`.

[72] Amazon (wang) network dataset – KONECT, November 2017. URL `http://konect.cc/networks/wang-amazon`.

[73] D. Stewart, D. Coutts. Haskell bytestring library. `https://hackage.haskell.org/package/bytestring`. Accessed: 2021-04-17.

[74] D. Kovanikov, V. Romashkina, S. Diehl, Serokell. Haskell relude library. `https://hackage.haskell.org/package/containers`. Accessed: 2021-04-17.

[75] Glasgow Haskell Compiler. Glasgow haskell compiler user's guide. `https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#language-options`. Accessed: 2021-05-09.

[76] FP Complete. Haskell stack tool. `https://docs.haskellstack.org/en/stable/README/`. Accessed: 2021-05-11.

[77] Don Stewart. Haskell ghc-gc-tune tool. `https://hackage.haskell.org/package/ghc-gc-tune`. Accessed: 2021-05-09.

[78] Apache Foundation. Apache kafka. `https://kafka.apache.org/`. Accessed: 2021-04-27.

[79] Haskell.org. Haskell unboxing. `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/primitives.html#`. Accessed: 2021-04-27.