UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

MAINTAINING THE MINIMUM SPANNING FOREST OF FULLY DYNAMIC GRAPHS ON THE DYNAMIC PIPELINE COMPUTATIONAL PATTERN

DANIEL BENEDÍ GARCÍA

Thesis supervisor
AMALIA DUCH BROWN (Department of Computer Science)

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

Degree
Master's Degree in Innovation and Research in Informatics (Advanced Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

28/06/2024

I have attempted science - Strange Planet. Nathan W Pyle

Abstract

In this work we tackle the problem of using the Dynamic Pipeline Approach to parallelize the Kruskal's algorithm and thus compute and maintain the Minimum Spanning Tree of dynamic graphs in parallel and distributed settings. To achieve this, we introduce a characterization of graphs that we call an *underlying forest* of a graph. This characterization gives the basis for representing graphs in a distributed way along a dynamic pipeline.

The proposed algorithm, named `DP_Kruskal`, utilizes a pipeline architecture where the graph is partitioned into trees and distributed across a sequence of stages each handling different aspects of computation. Under this algorithm it possible to keep efficiently graph updates as well as to apply Kruskal's Algorithm at every stage of the pipeline to compute incrementally the minimum spanning tree of the stored graph.

A working implementation was developed to analyze the suitability of `DP_Kruskal` against other algorithms. The programming language `Go` was chosen because of its concurrency features, as its `goroutines` and communication channels naturally align with the Dynamic Pipeline Approach.

Several key optimizations were introduced in `DP_Kruskal`. Some of them regarding the implementation and others the Dynamic Pipeline Approach. Notably, we propose a way to disentangle the message passing from computational tasks which is a general optimization that enhances the entire Dynamic Pipeline Approach framework, not just for `DP_Kruskal`. These optimizations not only improve program's efficiency but also address and resolve memory management issues within `Go`.

Extensive experimental evaluations were conducted to compare `DP_Kruskal` with both sequential and parallel algorithms. Kruskal's algorithm was selected for sequential comparisons, while `Filter_Kruskal` and a message-passing implementation of `Prim`'s algorithm for parallel algorithms. The results demonstrate that `DP_Kruskal` significantly outperforms its counterparts in both single-core and parallel environments, showcasing its superior performance and scalability for handling large and dynamic graphs.

This work substantiates the effectiveness of the Dynamic Pipeline Approach in addressing complex graph problems and underscores its potential for wider application in parallel computing.

Acknowledgements

I would like to express my deepest appreciation to the supervisors of this work, Amalia Duch and Edelmira Pasarella, for their invaluable patience, insightful feedback, and refreshing motivation throughout this journey. Your guidance has been instrumental in the completion of this project.

Additionally, I extend my sincere thanks to the /RDLab-UPC for providing the computational resources necessary for the experimental evaluation. Your support made the technical aspects of this research possible.

I am also grateful to my colleagues and friends from the master program. You provided a warm welcome to this new step in my life, tolerated my moments of stress, and inspired me to dream big. And to my friends from Zaragoza for accompanying me up to this point no matter what happened.

Last but not least, I could not have undertaken this journey without my family and their constant support throughout this research. To my parents, thank you for always supporting me and allowing me to return to my roots; to my sister, for being a beacon of inspiration; and to my significant other, for your constant love and understanding.

Contents

Chapter 1

Introduction

"The Future is Big Graphs" states the CACM paper [52] of the same title, where forty-one experts from the data management and large-scale-systems communities settled –after Dagstuhl Seminar 19491 held in Dec. 2019– their conclusions about the opportunities and challenges of graph processing in the next decade. Beyond the corroboration that graph algorithms should be scalable to deal with the huge amount of data required by nowadays applications, this group of experts claims, among other issues, that, in order to succeed, graph algorithms have to deal with dynamic and streaming aspects of graph processing. This is, coping with updates such as edge insertions, changes, and deletions (dynamicity), as well as indefinitely growth and/or evolution as new data arrive (stream model). Additionally, one of the challenges these experts identified was to define a reference architecture for big graph processing.

Some representative examples are server networks, mobile phone networks, or sensor networks [41], where the devices involved in the network are represented as nodes of a graph and the communications among devices are the weighted edges of the graph, since linking a pair of devices has an associated cost. The resulting resulting graph is fully dynamic as it captures the evolution of the networks along time: new devices can be added to the network, some others might be removed, and the links among them might be modified, involving therefore the insertion and deletion of edges of the graph.

Given an undirected and connected graph with weighted edges, a minimum weight spanning forest, minimum spanning forest, or simply minimum spanning tree (MST); is a subset of the edges that connects all vertices and has minimum total weight among all such sets. The MST problem is not only of theoretical interest, but also has a wide range of uses, including but not limited to clustering [8, 33], image segmentation [38, 61], and the design of networks [36].

To comprehend and forecast the dynamics of evolving networks effectively, it is essential to maintain a minimum weight spanning forest of the dynamic graph. This ensures that the network can be adjusted and reconfigured as new connections are established or existing ones are disconnected. Moreover, in current applications, it is crucial to efficiently compute and maintain a large number of network updates involving potentially a huge number of devices that might be distributed in distant spatial locations [56, 57].

An example of dynamic graphs in which we need to maintain a minimum weight spanning forest can be found in the standard IEEE 802.1Q [28]. This standard outlines the support of the Media Access Control (MAC) Service in Bridged Networks, the operational principles of such networks, and the functioning of MAC Bridges and VLAN Bridges, encompassing aspects like management, protocols, and algorithms. In order to provide simple and full connectivity throughout a bridged network comprising arbitrarily interconnected bridges, the Spanning Tree Protocol and some of its variants, such as Rapid Spanning Tree Protocol (RSTP) or the Multiple Spanning Tree Protocol (MSTP), are used in each bridge. The primary purpose of STP is to avoid bridge loops and the broadcast storms they cause. Additionally, Spanning Tree enables a network architecture to incorporate redundant links to ensure system reliability in case a primary link malfunctions.

This work focuses on applying the Dynamic Pipeline Approach (DPA) [46] to maintain a dynamic graph as well as its MST providing a parallel Kruskal's algorithm. The study includes the development, optimization, and analysis of the `DP_Kruskal` algorithm. Several optimizations are explored from the implementation of the characterization to optimizations of the DPA itself. The performance of `DP_Kruskal` is compared with `Filter_Kruskal` and a message-passing implementation of `Prim`'s algorithm on various graph sizes and densities to assess scalability and efficiency.

The study is limited to specific types of graphs, focusing on random static and real-world dynamic graphs. Experiments are conducted primarily on single-core and multi-core machines, not extending to distributed computing environments. While comparisons with `Filter_Kruskal` and `Prim`'s algorithm are included, not all existing MST algorithms are covered since mostly are theoretical, little implementations are available and it some experimental results show that they do not behave properly in large graphs.

The research does not encompass all possible optimizations or address comprehensive memory management techniques beyond those relevant to `Go` and DPA. By defining these boundaries, the study aims to provide a focused investigation into the parallelization of Kruskal's algorithm using DPA within the specified context.

This work comprises five chapters, with this introductory chapter laying the foundation. Chapter 2 -Preliminaries- provides an overview of the state of the art, as well as relevant algorithms used throughout the research. Chapter 3 -MST characterization in the DPA- explains the main contribution of this project: how the DPA is utilized to solve the MSTproblem.Chapter 4 -Experimental Study- describes the experiments conducted, their motivation, the results, and the subsequent analysis. Finally, a conclusion is presented in Chapter 5, summarizing the findings and suggesting future research directions.

Chapter 2

Preliminaries

In this chapter, we lay the groundwork for the research conducted in this thesis by presenting an overview of the fundamental concepts and algorithms related to the MST problem. We begin with a detailed examination of classical MST algorithms including `Prim`'s algorithm, `Kruskal`'s algorithm—along with its associated `Union-Find` data structure—and `Boruvka`'s algorithm. Following this, we explore parallel and distributed approaches to the MST problem, providing a general overview before delving into specific methods such as the `Filter_Kruskal` algorithm and the Message Passing `Prim` algorithm.

The chapter also covers the topic of fully dynamic MSTs, discussing both sequential and parallel/distributed approaches, most of them theoretical since there is little work in experimental. Finally, we introduce the Dynamic Pipeline Approach, setting the stage for its application in solving the MST problem, which will be further elaborated in the subsequent chapters. This comprehensive review equips the reader with the necessary background to understand the innovations and experimental analyses presented later in the thesis.

2.1 Minimum Spanning Tree

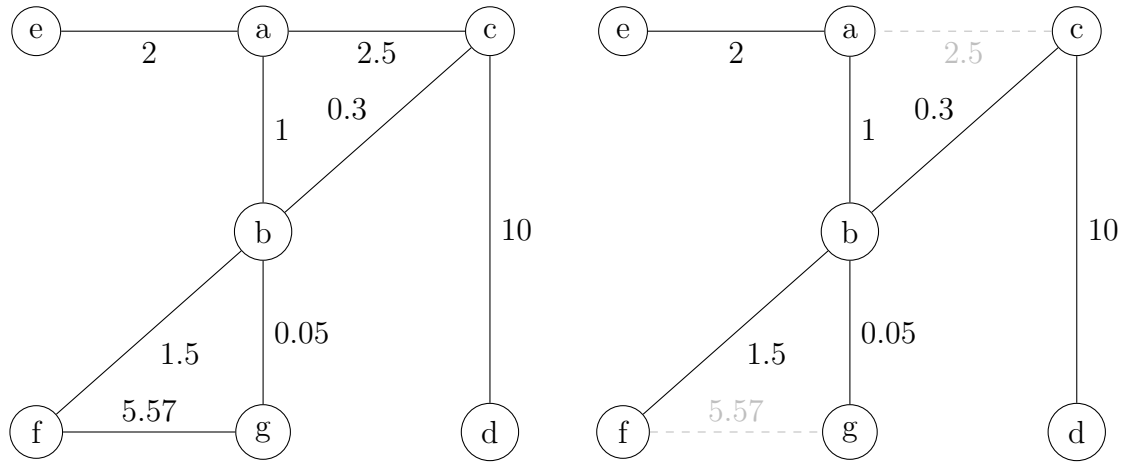
Definition 1. *A graph G consists of two sets: a set of vertices V and a set of edges E, where E is a set of unordered pairs from V such that $E \subseteq \{\{u, v\} \mid u, v \in V\}$. G is usually denoted as $G = (V, E)$.*

From now on, unless stated otherwise, for any graph $G = (V, E)$, we will refer to the elements of V indistinctly as the vertices or the nodes of G and to its size as n ($|V| = n$). Likewise we will refer to the elements of E as the set of edges of G and we will say that the size of E is m ($|E| = m$).

Definition 2. *A weighted graph $G = (V, E)$ is a graph in which its edges are assigned numerical values (known as weights) given by a function $\omega : E \to \mathbb{R}^+$.*

An example of a weighted graph is shown in Figure 2.1a.

Definition 3. *A spanning tree T of a graph G is a subgraph of G that is a tree and*



(a) A weighted graph. (b) A spanning tree.

Figure 2.1: An example of a weighted graph and one of its spanning trees

includes all the vertices of G.

For the purposes of our work, if a graph G is not connected, we will work with its *spanning forest*. A spanning forest of a graph G is a subset F of G conformed by a set of trees such that for every connected component of G, exactly one spanning tree of the connected component is in F. From now on, abusing of notation, we will refer to a spanning forest of a non connected graph G as a spanning tree of G. Figure 2.1b shows a spanning tree of the graph of Figure 2.1a. The light-gray dashed edges have not been added to the spanning tree.

Definition 4. *[53] Given a weighted graph $G = (V, E)$, $\omega : E \to \mathbb{R}^+$, we say that $T = (V, E')$, with $E' \subseteq E$, is a minimum spanning tree of G if T is a spanning tree of G that minimizes:*

$$\omega(T) = \sum_{e \in E'} \omega(e)$$

Let us observe that the spanning tree shown in Figure 2.1b is a minimum spanning tree of the graph shown therein.

From now on, abusing of notation, as long as it does not generate ambiguities and unless stated differently, we will use MST as an abbreviation of minimum spanning tree and MST(G) as the minimum spanning tree of graph G.

The finding and computation of MST*s* remains as a common challenge in graph theory, leading to the development of various algorithms to address it. Notable among these are Prim's [48], Kruskal's [35], and Borůvka's [10] algorithms, each leveraging different useful properties of MST*s*. These algorithms will be succinctly discussed in the subsequent subsections.

MST*s* have some useful properties: the cut-property and the cycle-property [1]. A *cut* is a partition of the vertices of a graph into two disjoint subsets. The *cut-set* of a cut is the set of edges that have one endpoint in each subset of the partition. The cut-property states that for any cut C of the graph, if the weight of an edge e in the cut-set of C is strictly

smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MSTs of the graph. A *cycle* is a finite sequence of edge which joins a sequence of vertices in which only the first and last vertices are equal. The cycle-property states that for any cycle C in the graph, if the weight of an edge e of C is larger than any of the individual weights of all other edges of C, then this edge cannot belong to an MST.

Prim's Algorithm

`Prim`'s Algorithm separtes the vertices in two subsets, those already added to the MST and those that are not added yet. It starts adding any given vertex to the MST. Then, in every iteration, it chooses the lightest edge that connects the vertices in the MST with the vertices not added yet and repeats until all the vertices are added. Algorithm 1 shows a sketch in pseudo-code adapted from [1]. Prim's algorithm implemented as described has a running time of $O(n^2)$, but can be reduced up to $O(m\frac{\log n}{\log(2+m/n)})$ [1].

Algorithm 1 Prim's algorithm

procedure PRIM(G)
  $u \leftarrow$ Select one of $V(G)$
  $F \leftarrow \varnothing$
  $Q \leftarrow V(G) \setminus \{u\}$
  $V(F) \leftarrow \{u\}$
  while $Q \neq \varnothing$ **do**
    $e \leftarrow \arg\min_{\{e=\{u,v\} \in E(G) | u \in V(F) \wedge v \in Q\}} \omega(e)$
    $Q \leftarrow Q \setminus \{v\}$
    $F \leftarrow F \cup \{e\}$
  end while
  return F
end procedure

Kruskal's Algorithm

This algorithm uses the cycle property of MSTs.

The idea behind the algorithm is the following: in every iteration add the lightest edge to the forest unless it forms a cycle - which happens when two vertices belong to the same connected component, CC. Algorithm 2 shows a sketch in pseudo-code adapted from [1]. The sorting step requires $O(m \log n)$ time and the time for the connected components is $O(m\alpha(m,n))$. The function $\alpha(m,n)$ is the inverse Ackermann function which grows much slower than the logarithm function and usually it is considered constant since $\alpha(n)$ is less than 5 for any practical input size n. Thus the total running time is $O(m \log n)$ [1].

Algorithm 2 Kruskal's algorithm

```
Require: E to be in weight-ascending order
  procedure KRUSKAL(G = (V, E))
      F ← (V, E' = ∅)
      for e = (u, v) ∈ E do
          if CC_F(u) ≠ CC_F(v) then        ▷ CC_F(u), the connected component of u in F
              F ← (V, E' ∪ {e})
          end if
      end for
      return F
  end procedure
```

Borůvka's Algorithm

This algorithms uses two properties of the MST*s*: the cut property already explained and the contraction property. The contraction property states that iff T is a tree of MST edges, then we can contract T into a single vertex while maintaining the invariant that the MST of the contracted graph plus T gives the MST for the graph before contraction.

The idea behind this algorithm is that initially the MST is empty and every vertex is single component in the MST. Then, in every iteration for every component, the cheapest edge that connects it to some other component is found and added to the MST. Algorithm 3 shows a sketch in pseudo-code adapted from [1]. Each iteration can be done in $O(\log n)$ and there is at most m iterations, thus the total running time is $O(m \log n)$ [1].

Parallel and/or Distributed Algorithms

The problem of computing the MST of a connected graph has also been studied for parallel and concurrent models of computation leading to several algorithms [5, 16, 59].

In the parallel model the objective is to minimize the maximum number of steps of computation that are performed by each processor as well as the number of processors involved, while in the distributed model the objective is to minimize the number of rounds of communication between the processors that are supposed to be of unbounded computational power. Also the algorithms in the parallel case assume that the processors have shared memory and this is not the case in distributed model. In particular, Bader et al. [5] implement three parallel variants of Borůvka's MST algorithm [13] arguing that this algorithm is more naturally paralelizable than Prim's and Kruskal's ones. The authors in [5] claim that –by the first time– they present a general parallel MST algorithm that runs efficiently in the SMP computer architecture[1]. In the distributed memory model such as the BMP computer architectue [60], the authors in [37] provide working an experimental comparison of algorithms based on Prim and Kruskal.

Independently of the topology of the input graphs, all these algorithms are designed for working on *in-memory* and/or static graphs –contrary to dynamic ones. To overcome the

[1]SMP computer architecture is a multiprocessor hardware and software architecture that has multiple identical processors. Processors share main memory and have access to all I/O devices.

Algorithm 3 Borůvka's algorithm

```
procedure BORŮVKA(G = (V, E))
    F ← (V, E' = ∅)
    Completed ← False
    while ¬Completed do
        Find the connected components (CC) of F
        Initialize the cheapest edge for each component to ∅
        for e = {u, v} ∈ E, CC(u) ≠ CC(v) do
            e' ← cheapest_edge(CC(u))
            if ω(e) < ω(e') then
                cheapest_edge(CC(u)) ← e
            end if
            e' ← cheapest_edge(CC(v))
            if ω(e) < ω(e') then
                cheapest_edge(CC(v)) ← e
            end if
        end for
        Completed ← True
        for i ∈ CC do
            e ← cheapest_edge(i)
            if e ≠ ∅ then
                F ← (V, E' ∪ {e})
                Completed ← False
            end if
        end for
    end while
    return F
end procedure
```

memory-bound problem, MST distributed algorithms have been proposed [44] where it is necessary to deal with message passing overhead, memory latency and fault-tolerance issues [11].

Filter Kruskal

`Filter_Kruskal` is a divide-and-conquer variant of Kruskal's algorithm inspired by the principles of Merge Sort [43]. This algorithm effectively partitions the graph's edges to facilitate parallel processing. The main idea is to choose a pivot edge, then separate the remaining edges into two subsets: those with weights less than the pivot and those with weights greater than the pivot. The algorithm recursively processes the lower-weight subset to build a partial MST. For the higher-weight subset, it removes edges that do not contribute to the MST (i.e., edges that connect vertices already in the same component). This approach leverages parallelism by dividing the problem into smaller, independent subproblems.

The `Filter_Kruskal` algorithm is particularly advantageous for its simplicity in parallelization, as noted by its authors [43] and has an expected running time of $O(m + n\log(n)\log(m/n))$. Below is the detailed pseudocode:

Algorithm 4 Filter Kruskal's algorithm

procedure FILTER_KRUSKAL($G = (V, E)$)
  if $|E| <$ threshold **then**
    return KRUSKAL(G)
  else
    $p \leftarrow$ pivot in E
    $E_{\leq} \leftarrow \{e \in E : \omega(e) \leq \omega(p)\}$
    $T_{\leq} = (V, E'_{\leq}) \leftarrow$ FILTER_KRUSKAL($(V, E_{\leq})$)
    $E_{>} \leftarrow \{e = (u, v) \in E : \omega(e) > \omega(p) \ \wedge \ CC_T(u) \neq CC_T(v)\}$
    $T_{>} = (V, E'_{>}) \leftarrow$ FILTER_KRUSKAL($(V, E_{>})$)
    return $(V, E'_{\leq} \cup E'_v)$
  end if
end procedure

In this algorithm, if the number of edges $|E|$ is below a certain threshold, the algorithm switches to the standard Kruskal's algorithm for simplicity and efficiency. A pivot edge p is selected from the graph G. This pivot helps in dividing the graph's edges into two subsets. The subset of edges $E_{\leq}$ containing edges with weights less or equal than p is recursively processed to build the MST. For the subset of edges $E_{>}$ with weights greater than p, edges that do not contribute to the MST (those connecting vertices already in the same component) are filtered out. The remaining edges are recursively processed.

This approach not only ensures the correctness of the MST but also enhances efficiency by leveraging parallel processing, making it suitable for large-scale dynamic graphs.

Parallel Prim

Lončar et al. [37] provide a parallelization of Prim algorithm (basic algorithm explained in Section 1). Only two steps can be parallelized: selection of the minimum-weight edge connecting a vertex not in MST to a vertex in MST, and updating array d -an auxiliary array of length n to store distances from each vertex to MST- after a vertex is added to MST.

The algorithm partitions the input set V into k subsets, where each subset contains n/k consecutive vertices along with their associated edges. Each process is assigned a distinct subset. In addition to their vertex subsets, each process also manages a portion of the array d, which tracks distances for the vertices in its partition. Let V_i represent the subset assigned to process p_i, and d_i the corresponding segment of the array d that p_i maintains.

Each process p_i identifies the minimum-weight edge e_i (the candidate edge) that connects a vertex in the MST to a vertex in V_i. These candidate edges are then sent from each process p_i to the root process using an all-to-one reduction. The root process selects the edge with the minimum weight from the received candidates, adds this edge to the MST, and broadcasts the selected edge to all other processes. Each process then updates its local portion of the array d and marks the vertices connected by this edge as part of the MST.

This process is repeated iteratively until every vertex is included in the MST.

Finding a minimum-weight edge and updating d_i during each iteration costs $O(p/n)$. Each step also performs a communication between the processes which takes $O(\log p)$. Combining the cost of one iteration and since there are n iterations, the total parallel time is: $O(n^2/p + n \log p)$.

2.2 Algorithms for Fully Dynamic MST

A static weighted graph according to Definition 2 involves a set of nodes V, a set of edges E, and a weight function ω. A weighted *dynamic graph* is obtained when any of these four entities changes over time [25]. A dynamic graph allows three kind of operations: insert a new edge, delete an edge of the graph and update the weight of an edge. If only one of this operations is allowed, we say that it is a *Partially Dynamic Graph*. On the other hand, if any operation is allowed, it is a *Fully Dynamic Graph*. We call any instance of these operations an *event*:

(i) `update`(G, e, ω_e) where the previous weight of edge $e \in E$ is replaced by w_e

(ii) `insert`(G, e, ω_e) that inserts edge e in E and assigns to it weight ω_e

(iii) `remove`(G, e) that removes e from E.

The study of dynamic graphs and, in particular, of algorithms that maintain a MSTof the dynamic graph is not new. In the extensive survey of Hanauer, Henzinger and Schulz [24] there is a reasoned state of the art on these algorithms that divides them clearly into the-

oretic algorithms and practical ones. Regarding specifically the maintenance of the MST of dynamic graphs, there has been very little work done on both theoretical algorithms and their empirical evaluation.

Frederickson [21] proposed a data structure for online updating of MST with a cost of $O(\sqrt{m})$ per update. Their proposal uses as base a previous work by Sleator and Tarjan [54] for maintaining a dynamic tree. Frederickson's solution applies a preprocessing step that forces all the graphs to have degree at most 3, by adding new vertices that are connected with weight zero. Holm et al. [26] gave the first fully dynamic algorithm with poly-logarithmic time per operation improving it later to $O(\log^4 n/ \log\log n)$ amortised time per operation [27]. The lower bound of $\Omega(\log n)$ per operation on graphs of n vertices for connectivity of dynamic graphs was extended by Patrascu et al. [47] to maintaining MSTs.

On the empirical evaluation, we can find the work of Amato [2] or Cattaneo et al. [12] that provide an extensive experimental study that assesses different approaches:

a) Eppstein et al.'s sparsification [17–19]

b) Frederickson's partitions and topology trees [20, 21]

c) Holm et al.'s logarithmic decomposition [27]

The experiments therein [12] have shown that, except for very particular instances, a simple $O(m \log n)$-time algorithm has the best performance in practice (for graphs of n vertices and m edges) and runs substantially faster than the poly-logarithmic algorithm of Holm et al. [27]: the latter was faster than the first for only one out of five different classes of instances, namely those in k-clique graphs, in which small cliques are connected by some inter-clique edges and all updates involve only the inter-clique edges. Despite its good theoretical amortized running time, the complex chain of data structures supporting Holms et al. [27] algorithm does not seem to lead to fast implementations and may require too much memory.

For the specific case of graphs with changing weights –i.e. the edges of the graph are constant, but the edge weights can change dynamically– Ribero and Toso [49] proposed a $O(m)$ amortized running time algorithm –for graphs of m edges– that reduces the computation time observed for the algorithm of Cattaneo et al. [12], yielding the fastest experimental algorithm for the above mentioned graphs.

Up to our knowledge, the only parallel and /or distributed algorithm dealing with the maintenance of the MST of fully dynamic graphs was given by Nowicki [42]. The algorithm is based on the massively parallel computation (MPC) computation model (MapReduce) [31], but it is restricted to sequences of updates of sub-linear size (with respect to the graph size).

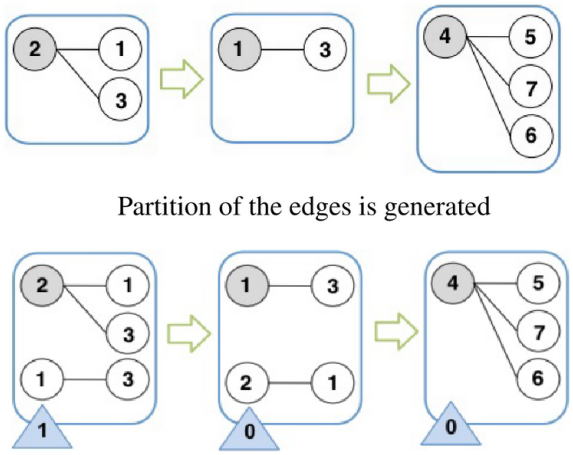
2.3 Dynamic Pipeline

Big data and the Internet of Things (IoT) are pivotal in driving both industrial digitization and interdisciplinary scientific research. Data-driven frameworks, which adapt

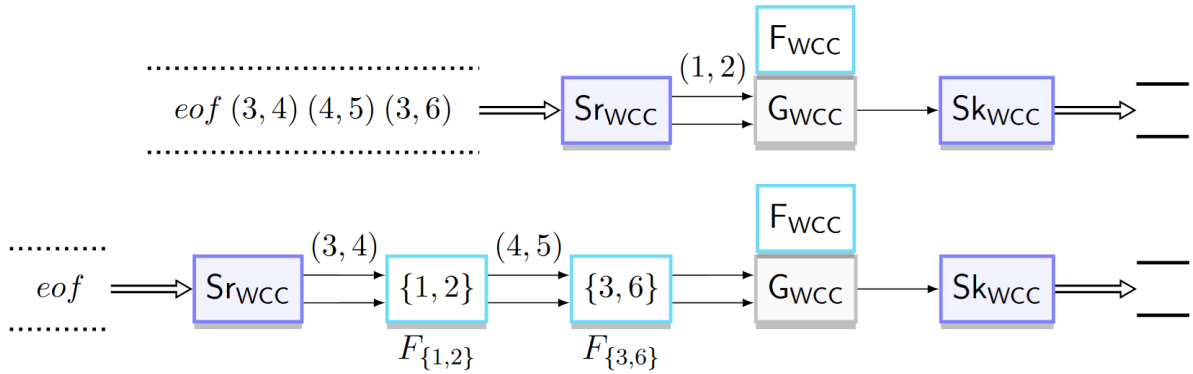
execution schedulers and computational structures to current input data conditions, are essential for managing continuous data streams. Data Streaming (DS) has been approached through various methods to efficiently process large volumes of data with high parallelization. Two distinct parallelization models have been identified: Data Parallelism (DAP) and Pipeline Parallelism (PP). Data-driven frameworks, which adapt execution schedulers and computational structures to current input data conditions, are essential for managing continuous data streams. However, traditional parallel paradigms like MapReduce [15, 31], which rely on the Divide & Conquer approach, have limitations. They partition problems into subproblems that must be fully executed before proceeding, resulting in a blocking behavior that impedes incremental result generation and limits the flexibility needed for real-world applications. In contrast, the Dynamic Pipeline Approach [46, 62] (DPA) is a PP computational model that operates as a one-dimensional, unidirectional chain of stages implemented as an asynchronous model of computation synchronized by channels, enabling the simultaneous execution of dependent tasks on different data items.

In the DPA stages can be categorized into several types: input stages, filter stages, generator stages, and output stages. Input stages are responsible for reading and feeding data into the pipeline. Filter stages process the data, applying transformations or filtering operations as needed. Generator stages can produce new filter stages based on the current data flow. Finally, output stages handle the end result, such as writing data to storage or presenting it to the user. Modeling a problem within the DPA involves breaking it down into filters. For example, a graph-processing problem can be approached by having filter stages apply partitioning to the graph and algorithms to process the vertices and edges, ensuring that the solution can be built incrementally from the output of the previous filter.

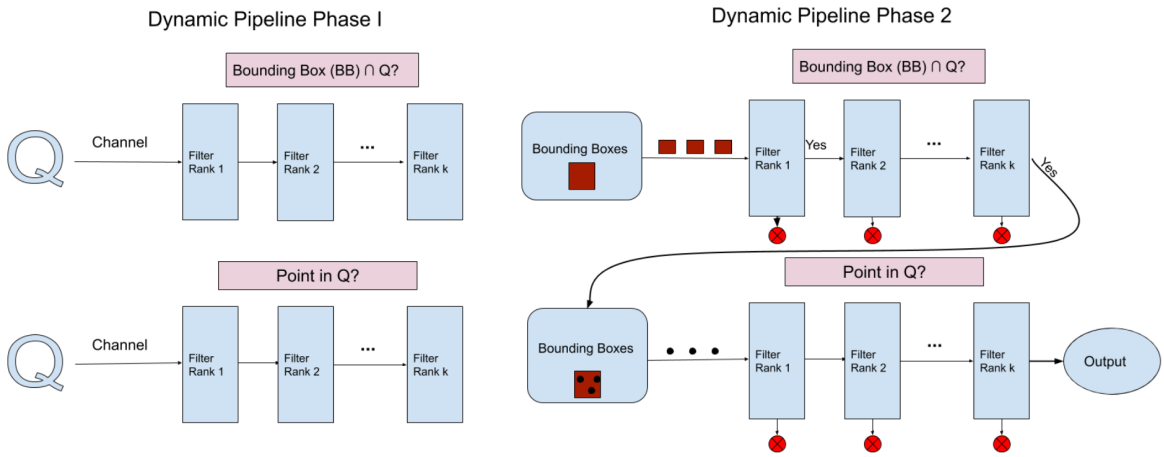
Many different problems have been implemented in the DPA, see Figure 2.2. Authors in [45] use DP for the well-known problem of counting triangles in a graph. Results suggest that pipeline allows for the implementation of an efficient solution of the problem of counting triangles in a graph, particularly, in dense and large graphs, drastically reducing the execution time with respect to the MapReduce implementation. The problem of enumerating bitriangles in large bipartite nerworks [50, 51] have also been implemented efficiently in the DPA. Another example of problems solved in DPA is given in [39]. In this work, authors propose a working implementation for solving the problem of multidimensional range queries, the retrieval of points of a multidimensional space that fall inside a given region.



(a) Dynamic Pipeline for triangles in graphs. Images from [45]



(b) Dynamic Pipeline for bitriangles in bigraphs. Images from [51]



(c) Dynamic Pipeline for Quadtrees. Image from [39]

Figure 2.2: Examples of different problems in DPA.

Chapter 3

Graphs and MST*s* in the DPA

In this chapter we introduce a representation of static and dynamic graphs suitable to characterize graphs in the DPA. A particular property of MST*s* represented as we propose will allow us to compute the MST of a graph in the DPA. We show also the `DP_Kruskal` Algorithm that uses this representation for tailor the DPA, and delve into its detailed implementation and various optimizations. We begin by explaining the representation and proving the above mentioned property, followed by its usage in the dynamic pipeline with a comprehensive breakdown of the implementation, which includes the Communication, Source Stage, Filter Stage, Generator Stage, and Sink Stage. Afterwards we include a cost analysis section, where we evaluate the time and memory complexity of `DP_Kruskal`, providing insights into its efficiency and scalability. We then introduce several key optimizations designed to enhance the performance of `DP_Kruskal`. These include the use of multiple roots per filter, Decoupled Event Handling, adaptive MST caching, and improved memory management. Notably, the Decoupled Event Handling optimization is highlighted as a general improvement that can be applied to any problem implemented using the DPA, not just the MST problem.

Through these sections, we aim to demonstrate how the `DP_Kruskal` algorithm, bolstered by these optimizations, offers a robust and scalable solution for maintaining the MST in dynamic graphs. This sets the stage for the experimental analyses presented in subsequent chapters, where the effectiveness of these enhancements is empirically validated.

3.1 Underlying Forests and MST*s*

In this section we present our algorithm for computing minimum spanning forests of dynamic weighted graphs according to the dynamic pipeline approach. In order to simplify the following definitions, we assume that the graphs have no isolated vertices (vertices with no incident edges), but can be easily adapted. We start by introducing the preliminaries to define the algorithm.

In the literature [14], there are many possible computational representations of graphs: as adjacency lists, as adjacency matrices, as a computable function, as a list of edges and many more. We propose a representation based on its *underlying forest*.

Definition 5. *Given a graph $G = (V, E)$, we say that the sequence $F_G = \langle T_1, \ldots, T_k \rangle$, $k \geq 1$, is an* underlying forest *of G if $T_i \subseteq E$ $\forall i$, $\cup_{i=1}^{k} T_i = E$, $\forall i, j, i \neq j, T_i \cap T_j = \emptyset$ and $\forall i$ $T_i \in F_G$ there exists a distinguished vertex $v_i \in V$, called the* root *of T_i, such that $\forall e \in T_i$, e is incident to v_i.*

Notice that every element of an underlying forest of G is a tree of height 1. Indeed, $\forall T_i \in F_G$ $\mathsf{height}(T_i) = 1$. Notice also that if T_i consists of a unique edge any of its two vertices could be distinguished as the root of T_i. Moreover, G can be characterized as $G = (V, \bigcup_{j=1}^{k} T_j), k \geq 1$ and hence, in what follows, by an abuse of notation we might also write $G = \bigcup_{i=1}^{k} T_i$. Likewise, we will also refer by G_i to the graph of G given by $G_i = \bigcup_{j=1}^{i} T_j$, $1 \leq i \leq k$.

In Figure 3.1, we show a weighted graph G together with one of its possible underlying forests. Edges are represented by 3-tuples of the form $(u, v, \omega(\{u, v\}))$, where $\omega(\{u, v\})$ is the cost of the edge $\{u, v\}$ (according to Definition 2). The graph therein is $G = (V, E)$, with $V = \{a, b, c, d, e\}$ and $E = \{(a, b, 2), (a, e, 1), (b, e, 1), (c, e, 1), (b, d, 7), (c, d, 4), (b, c, 1)\}$.

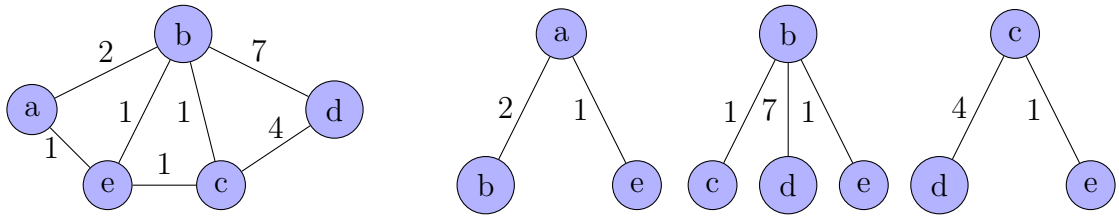


Figure 3.1: The sequence of trees on the right form an underlying forest of the graph shown in the left.

This characterization allows to analyze various properties of graphs in the DPA, however, our interest is to study the way to compute and maintain dynamically the MST of a dynamic graph. To do so, next proposition is crucial.

Proposition 1. *Given a weighted graph $G = (V, E)$ represented by the underlying forest $F_G = \langle T_1, \ldots, T_k \rangle$ $(k \geq 1)$ and the subgraphs G_i, $1 \leq i \leq k$, of G such that $G_i = \cup_{j=1}^{k} T_i$, it holds that*

$$\mathsf{MST}(G_i) = \begin{cases} T_1 & \text{if } i = 1 \\ \mathsf{MST}(T_i \cup \mathsf{MST}(G_{i-1})) & \text{if } i > 1 \end{cases}$$

where MST*(G) is any correct procedure to compute a* MST *of G.*

Proof. We proceed by induction on the size of F_G.

Base case, i= 1: $F_{G_1} = \langle T_1 \rangle$ and thus $G_1 = T_1$. Since T_1 is a tree containing all the edges of G_1, it corresponds to its own minimum spanning tree. Let us observe that for the case $i = 2$ the property follows trivially since $\mathsf{MST}(G_2) = \mathsf{MST}(T_1 \cup T_2) = \mathsf{MST}(\mathsf{MST}(T_1) \cup T_2)$.

Induction Hypothesis (HI): Let us suppose that $\mathsf{MST}(G_i) = \mathsf{MST}(T_i \cup \mathsf{MST}(G_{i-1}))$ is, effectively, a MST of G_i.

We want to proof that if (HI) holds then $\mathsf{MST}(G_{i+1}) = \mathsf{MST}(T_{i+1} \cup \mathsf{MST}(G_i))$.

Induction Step: Let $e \in E$ be any edge of $\mathsf{MST}(G_{i+1})$, then e is either in G_i or in T_{i+1} If e belongs to T_{i+1}, then $\mathsf{MST}(G_{i+1})$ will consider it (since it looks at all the edges of T_i) and add it to $\mathsf{MST}(G_{i+1})$.

If, on the other hand, e belongs to G_i then there are two options for it to belong to $\mathsf{MST}(G_{i+1})$:

1) e does not belong to any cycle in G_{i+1}, which implies that the edge does not belong neither to any cycle in G_i, so it will also be part of $\mathsf{MST}(G_i)$ and hence will be considered by $\mathsf{MST}(G_{i+1})$.

2) e belongs to at least one cycle of G_{i+1}. If e belongs only to cycles that include edges from T_{i+1}, then as G_i does not include such edges, e does not belong to any cycle in G_i and hence has to be part of $\mathsf{MST}(G_i)$. The last option left is that the e belongs to cycles in G_i. In such case, it cannot be the weightiest edge in any of that cycles, otherwise it would not be part of $\mathsf{MST}(G_{i+1})$, hence it has to be in $\mathsf{MST}(G_i)$.

With all this cases covered, we can state that we can build the $\mathsf{MST}(G_{i+1})$ as the MST of the union of the tree T_{i+1} with the $\mathsf{MST}(G_i)$.

We have not considered here the case in which the graphs G_i, $1 \leq i \leq k$, are not connected. If this is the case, then the final result is the sequence of $\mathsf{MST}s$ of every connected component of G_i. Since the proposition holds for any connected component, it is valid for the union of all connected components. □

Proposition 1 gives us the foundation to compute the MST of graphs represented by any of their underlying forests. To get insights about how this can help us to design an algorithmic proposal, let us consider the Example 1.

Example 1 (Computation of $\mathsf{MST}(G)$ from F_G)**.** *Let G be the forest $F_G = \langle T_1, T_2, T_3 \rangle$, where $T_1 = \{(a,b,2),(a,e,1)\}$, $T_2 = \{(b,c,1),(b,d,7),(b,e,1)\}$ and, $T_3 = \{(c,d,4),(c,e,1)\}$ shown in Figure 3.1. The computation of $\mathsf{MST}(G)$ is done according to Proposition 1. This is, $\mathsf{MST}(G)$ is $\mathsf{MST}(T_3)$ and is computed as follows:*

1. $\mathsf{MST}(G_1) = T_1 = \{(a,b,2),(a,e,1)\}$
2. $\mathsf{MST}(G_2) = \mathsf{MST}(T_2 \cup \mathsf{MST}(G_1)) = \{(a,e,1),(b,c,1),(b,d,7),(b,e,1)\}$
3. $\mathsf{MST}(G_3) = \mathsf{MST}(T_3 \cup \mathsf{MST}(G_2)) = \{(a,e,1),(b,c,1),(b,e,1),(c,d,4)\}$

3.2 DPA and MSTs of Dynamic Graphs.

In the classical sequential model of computation one could easily implement the traversal of forest F_G by a simple loop running over its trees. However, thinking in alternative ways of computation, it would be natural to distribute the trees of F_G along a pipeline, distributing the sequence among several processors. Then, the sequence of computations given in Example 1 would also been distributed along the processors holding the trees implied in the computation. That is exactly the idea of the *dynamic pipeline*, `DP_Kruskal`. The benefit of using this graph representation is threefold:

(i) The graph can be updated in a very simple way, just by adding or deleting edges of one element of F_G. Moreover, as we will see, this representation is adequate for the DPA [46] since the trees of F_G can be very easily spread out across a *dynamic pipeline* producing the distributed representation of the graph that we will refer to as `DP_Kruskal`. Hence, while the dynamic pipeline is active, G remains updated along time.

(ii) At any time t, the concurrent (and parallel) computational structure of `DP_Kruskal` is able to compute effectively the MST(G) of the current graph when required.

(iii) During the computation of the MST(G), the priority queue required for the implementation is of size at most $O(n)$ in contrast with the sized m priority queue required by the implementation of Kruskal's sequential algorithm because there will be at most $2n$ edges.

Following the definition in [46], to define a dynamic pipeline, we have to define (i) the configuration and the behaviour of the four kind of different stages (*stateful functions*): source or input, generator, (instances of) filter and sink or output, in this order; and (ii) the channels connecting stages. Each filter instance's state contains a tree of the forest F_G and is parameterized by the root of this tree. Besides filter instances' behaviour correspond to the operation defined in Proposition 1. The different stages of the pipeline are connected by means of two communication channels carrying the *events* and the MST(T_k) for passing the temporal MST(G), respectively.

When a pipeline is initialised, only the Source, the Generator and the Sink stages exist. Then, the pipeline shrinks and expands dynamically depending on the sequence of insertions and deletions of the edges of the graph but all these operations can modify only the number of filter stages, the underlying structure is not modified, unless the `DP_Kruskal` terminates. Figure 3.2 depicts the state of an instance of `DP_Kruskal`, and below we describe all the stages with more precision.

Source (I): Whenever this stage receives an event it passes it to the rest of the pipeline through the event channel, if $\texttt{op} = \texttt{mst}$ it also passes the empty set through the graph channel.

Filter ($F(v)$): The parameter v is called the *root* of the filter stage. Whenever an event $(\texttt{op}, \texttt{e} = \{\texttt{v}, \texttt{w}\})$ arrives to $F(v)$, the following things can happen: (i) If $\texttt{op} = \{\texttt{insert}, \texttt{update}, \texttt{delete}\}$ and e is incident to the root v, the event is treated in $F(v)$ according to `op`. On the contrary, if it is not incident, the event is passed through the event channel to the next stage of the pipeline; (ii) If $\texttt{op} = \texttt{mst}$, a partial MST is computed and passed to the rest of the pipeline through the graph channel by computing the MST of the graph resulting of the union of the MST that arrives through the graph channel with the tree in $F(v)$. As explained, a possible implementation would consist on joining the temporal graph from the graph channel with the filter's tree and the compute the MST with Kruskal's algorithm.

Generator (Gen): Whenever an event $(\texttt{op}, \texttt{e} = \{\texttt{v}, \texttt{w}\})$ arrives to this stage by the event channel, if $\texttt{op} \in \{\texttt{insert}, \texttt{update}\}$, a new instance of Filter stage, $F(v)$, is spawned and added to the pipeline between the last filter and *Gen*. The edge e is stored in the local

memory of $F(v)$. Besides, whenever `op` = `mst`, it passes the MST that arrives by the graph channel to the sink stage O.

Sink (O): When an event `mst` arrives to this stage, it outputs the MST_G.

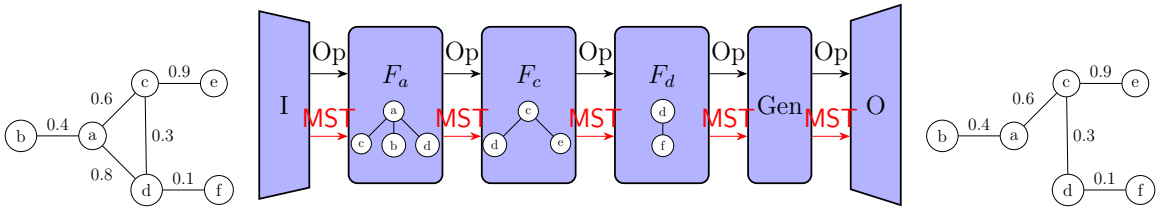


Figure 3.2: A `DP_Kruskal` at some point of its execution, the input graph and the MST that it returns. The graph is distributed along filters F_a, F_c and F_d.

Complexity. Conducting a precise cost analysis is inherently challenging due to the numerous influencing factors. In particular, the order in which graph modifications occur and the frequency of each type of operation significantly impact the structure of the pipeline and the overall performance. The following analyses provide a rough approximation based on the worst-case scenario and some amortised cost, ignoring the fact that we need to pass through all the filters. The `insert` event is simply $O(1)$, since we are appending to the tree. The `update` and `remove` require to scan the list of adjacent edges of a tree, so it will be $O(n)$. The `mst` can be implemented using the Kruskal Algorithm of Section 2.1. Each tree has at most n edges and the Kruskal Algorithm takes $O(n \log n)$ as mentioned in Section 2.1. This operation is performed at most n times as there are at most n trees in the underlying forest. So it has a cost $O(n^2 \log n)$, but the each operation in the trees can be performed in parallel for several MSTs, which implies that we have an amortised cost of $O(n \log n)$. Each `mst` operation takes at most $O(n)$ memory since the resulting MST has that size, the input graph as well and the priority queue needed for Kruskal takes also $O(n)$.

3.3 Implementation

The implementation was done using `Golang` 1.20 and it is available at Github [1]. The choice of `Go` was influenced by several factors. Firstly, the language supports lightweight green threads called `goroutines`, which are efficiently managed by the `Go` runtime scheduler. Additionally, the communication between stages is naturally facilitated by channels, enabling unidirectional communication between `goroutines`.

Communication

To communicate between stages, as already explained, we use `Go` channels. To avoid undesired synchronizations, we use buffered channels with up to 4096 item, a totally arbitrary number. Then, we packed the channels in `in_comm` if the channels are only incoming and `out_comm` for out-coming as shown in Figure 3.3.

[1] https://github.com/danielbenedi6/MasterThesis

```
type in_comm struct {
        Req   <-chan cmn.Request
        Graph <-chan cmn.Graph
}

type out_comm struct {
        Req   chan<- cmn.Request
        Graph chan<- cmn.Graph
}
```

Figure 3.3: Source code for the Channels in `Go`.

Source Stage

The source stage consists of an element that will feed the pipeline either reading an input file or by generating random graphs, so-called in-memory.

When reading from a file, we defined a file format in order to unify the different formats from all the dynamic graph databases that we used. Although we started with a version that stated the operation with a keyword, we opted to change this definition because of performance issues. Instead, the operation is defined by an integer: the insert an edge operation is represented by number 1, the update the weight operation is number 2, the delete an edge operation is defined by number 3, the operation to obtain the minimum spanning tree is number 4, the number 5 represents the end of the file and the number 6 returns the whole graph. The operations of inserting and updating will be followed by three arguments representing the two vertices and the weight of the edge. The operation of deleting will be followed by the two vertices that form the edge.

`Golang` offers different methods to parse data from a file. The most straightforward is based on the package `fmt` in which we can simply define the format of the desired input and it will manage all the exceptions and ensure the parsing is correct. Another option is the package `bufio` that allows to implement a parser and uses buffers to access to the files. An experimental analysis of both packages and why `bufio` was chosen is given in 4.2.

When the source stage is chosen to be a generator of random graphs, we opted to use binomial random graphs. A binomial random graph is a graph constructed by connecting nodes randomly in which each edge is included in the graph with probability p, independently from every other edge. A detailed explanation in Section 4.1.

Since both vertices of an edge can be the root in the underlying forest, before sending the edge through the `DP_Kruskal`, a "normalization" operation is done in which the vertices are ordered lexicographically, so the smallest one will be the root. This order, instead of a random order for instance, affects the shape of the pipeline and it will be studied in Section 4.2.

Filter Stage

The filter stage is a functional stage, as defined in [46], responsible for processing input data. The size of the `DP_Kruskal` structure depends entirely on the number of filter

stages present.

The grouping property, $\mathcal{R}$, of our filter stage ensures that each edge is adjacent to the root of the filter. Consequently, the filter retains only events with an adjacent edge. Instead of checking both vertices, we opted to check only the first vertex of the edge. The `insert` operation adds the edge to the end of the edge list, T_i. The `remove` operation replaces the edge with the last one in the list and reduces the array size. The `update` operation scans the edge list and updates the edge weight.

The `mst` operation is more complex as it involves implementing an MST algorithm and merging two trees. We assume the incoming MST, K, is a sorted list of edges by weight. The operation then performs the Kruskal algorithm, as explained in Algorithm 2. To find the connected component in which a vertex belongs to, we proposed the usage of the Union-Find data structure described in Algorithm 6. Instead of selecting the lightest edge of the graph, the algorithm selects the lightest edge from either K or T_i. Algorithm 5 provides a pseudocode of a possible implementation of this process.

Algorithm 5 Possible implementation of Kruskal for Filter Stage of `DP_Kruskal`

Require: T_i and $\mathsf{MST}(G_{i-1})$ are represented as a sequence of the edges and $\mathsf{MST}(G_{i-1})$ is already in weight-ascending order.

```
procedure KruskalDP(T_i, MST(G_{i-1}))
    F ← ∅
    Sort(T_i)                                   ▷ Sort T_i by weight
    CC ← 0                                      ▷ Number of connected components
    I ← []                                      ▷ DS for Union-Find
    M ← {}                                      ▷ Map for vertex id to conected component id
    j, k ← 0, 0                                 ▷ Indices for T_i and MST(G_{i-1})
    while j < |T_i| ∧ k < |MST(G_{i-1})| do
        e = (u, v) ← arg min_{T_i[j], MST(G_{i-1})[k]} ω(e)
        if u ∉ M then M[u] ← CC,  CC ← CC + 1
        if v ∉ M then M[v] ← CC,  CC ← CC + 1
        if Find(M[u], I) ≠ Find(M[v], I) then
            F ← F ∪ {e}
            Union(M[u], M[v], I)
        end if
        if e = T_i[j] thenj ← j + 1
        else k ← k + 1
    end while
    return F
end procedure
```

UnionFind

The sketch for Kruskal's algorithm given in Algorithm 2 needs a data structure to keep record of the connected component in which each vertex belongs to. Such data structure can be the Union-Find. The Union-Find data structure, also known as Disjoint Set Union (DSU), is a fundamental data structure used to manage and merge disjoint sets. It plays a crucial role in the implementation of Kruskal's algorithm for finding the MST

of a graph and it is used in Kruskal's variation for `DP_Kruskal` Algorithm 5. The origins of the Union-Find data structure trace back to algorithms proposed by Bernard Galler and Michael Fischer in [22]. Their seminal work introduced efficient methods for union and find operations, which have become essential in various areas of computer science, including network connectivity. This section delves into the mechanics and efficiency of the Union-Find algorithm, detailing its union by rank and path compression techniques that ensure nearly constant time complexity for union and find operations.

The usual description of the algorithm is making use of pointers and trees, but in Algorithm 6 we provide a sketch for implementing in a single array, I, of length the number of elements. In order to find the set to which the i-th element belong, the procedure `Find` will go to the i-th position of the array I, and the value in that position is the parent of that element, so the algorithm repeats until it finds an element that has itself as parent. On its way, the algorithm performs path compressing which consists on assign as parent of the i-th element the parent of its parent. This implementation of the `Find` algorithm is based on Tarjan and Van Leeuwen algorithm in [58]. The `Union` algorithm for joining the i-th element and the j-th element is as simple as finding the roots of the subtrees with the `Find` algorithm and assigning a new parent – since $\texttt{Find}(i) = I[\texttt{Find}(i)]$ and $\texttt{Find}(j) = I[\texttt{Find}(j)]$. When a new elements is needed to be added to the Union Find data structure, we append it to the array I and assign an id equal to the position in the array.

Algorithm 6 Both methods needed in Union-Find.

procedure FIND(p, I) ▷ I is passed by reference
  while $p \neq I[p]$ **do**
    $I[p] \leftarrow I[I[p]]$
    $p \leftarrow I[p]$
  end while
  return p
end procedure

procedure UNION(p, q, I) ▷ I is passed by reference
  $i \leftarrow$FIND(p, I)
  $j \leftarrow$FIND(q, I)
  $I[i] \leftarrow j$
end procedure

Generator Stage

In Figure 3.4, we present an implementation of the generator stage in `Go`. As previously described, when a generator stage receives a `insert` or `update` event, it instantiates a new filter and assigns the event to it. To prevent any event loss, this new filter is equipped with the input channels of the generator stage. Subsequently, the generator replaces these input channels with new ones, which serve as the output channels of the filter and the input channels for the generator itself.

```
func generator(in in_comm, out out_comm) {
        for {
                r, ok := <-in.Req
                if !ok {
                        break
                }
                switch r.Op {
                case cmn.Insert, cmn.Update:
                        out_req := make(chan cmn.Request, channelSize)
                        out_grph := make(chan cmn.Graph, channelSize)
                        new_out := out_comm{Req: out_req, Graph: out_grph}
                        new_in := in_comm{Req: out_req, Graph: out_grph}

                        go filter(in, new_out, r.E)
                        in = new_in
                case cmn.Delete:
                        // Do nothing, asked to delete unexistent edge
                case cmn.KMST:
                        g, _ := <-in.Graph

                        out.Req <- r
                        out.Graph <- g
                case cmn.EOF:
                        out.Req <- r
                        break
                default: //something's wrong
                        fmt.Println("Unknown operation in generator")
                        break
                }
        }
        close(out.Req)
        close(out.Graph)
}
```

Figure 3.4: Source code for the Generator

Sink Stage

The sink stage implementation is not relevant, it will simply read both channels and output it in standard output.

3.4 Optimizations

Multiple roots per filter

The first optimization we applied was the introduction of multiple roots per filter. This approach aimed to reduce the number of stages in the pipeline, thereby minimizing the overhead for the `Go` runtime scheduler.

In the original implementation, each filter in the pipeline had a single root, which often led to an excessive number of stages with really little number of edges. Each stage, being a separate goroutine, added overhead to the `Go` runtime scheduler, impacting overall

performance.

To address this, we modified the filter design to support multiple roots. By allowing each filter to manage several roots, we effectively reduced the number of required stages. This optimization brings several benefits. Firstly, it reduces overhead by minimizing the number of stages, resulting in fewer goroutines and less scheduling overhead managed by the Go runtime. This leads to more efficient utilization of computational resources and improved performance. Secondly, it enhances scalability by allowing each filter to handle a larger portion of the graph, thereby distributing the workload more evenly and maintaining performance as the graph size increases.

While the introduction of multiple roots per filter offers numerous benefits, it also brings about an additional parameter that requires consideration during filter creation: the number of roots it will handle. This parameter introduces complexity to the configuration process, necessitating careful deliberation to achieve optimal performance. Selecting the appropriate number of roots for each filter is important, as an insufficient number may lead to under-utilization of computational resources, while an excessive number can result in unnecessary overhead and decreased efficiency. Consequently, configuring the number of roots for each filter adds a layer of complexity to system setup and management. We delve into the impact of this parameter on performance in Section 4.2.

Decoupled Event Handling

Another optimization we implemented is the Decoupled Event Handling. During our experiments, we observed that events were being held up in a filter until it completed processing, even when the subsequent operation did not affect it. To address this inefficiency, we divided the filter into two substages: the MailBox and the Worker, each running in separate goroutines.

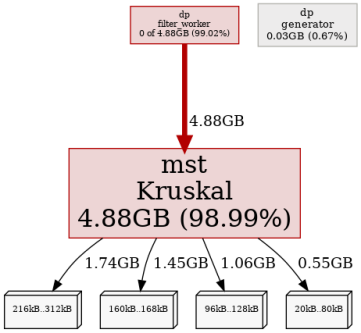
The MailBox is responsible for reading incoming operations and determining their relevance. It performs preliminary operations to decide whether an event is pertinent to the filter. If an event is relevant, the MailBox forwards it to the Worker; if not, it passes the event directly to the next filter in the pipeline. This ensures that irrelevant operations do not block the processing pipeline.

The Worker, on the other hand, focuses solely on performing operations on the internal graph. By isolating the Worker from the task of filtering incoming events, we ensure that it can operate more efficiently, without unnecessary interruptions.

Adaptive MST Caching

One improvement we implemented in the dynamic pipeline is called Adaptive MST Caching. The core idea behind this optimization is to store the last computed MST within each filter stage. This approach allows the system to output the cached MST if there have been no changes within the filter or the incoming MST.

In detail, the MST channel has been modified to pass both the graph and a flag indicating whether there has been a change since the last computation. Similarly, each filter now maintains a flag to indicate any changes since the last MST event. When an `insert`,



(a) Memory allocation of an instance.

```
         .          .   33:func Kruskal(root map[int64]*cmn.Graph, mst cmn.Graph) cmn.Graph {
         .          .   34:    ...
         .          .   82:        if !ok1 || !ok2 {
   4.45GB     4.45GB   83:            returnMST = append(returnMST, e)
         .          .   84:            if !ok1 {
   9.20MB     9.20MB   85:                m[e.X] = cc
   1.52MB     1.52MB   86:                id = append(id, cc)
         .          .   87:                ...
         .          .   88:            }
         .          .   89:            if !ok2 {
 305.08MB   305.08MB   90:                m[e.Y] = cc
 121.94MB   121.94MB   91:                id = append(id, cc)
         .          .   92:                ...
         .          .   93:            }
         .          .   94:            ...
         .          .   98:        } else if father(m[e.X], id) != father(m[e.Y], id) {
   3.25MB     3.25MB   99:            returnMST = append(returnMST, e)
         .          .  100:            ...
         .          .  103:        }
         .          .  104:    ...
         .          .  106:    return returnMST
         .          .  107:}
```

(b) Detailed profiling of memory allocation.

Figure 3.5: Information given by golang at runtime of resource allocation of an instance with 384,585 edges.

`remove`, or `update` event is applied to the filter, it performs the operation and sets its internal change flag.

Upon receiving an `mst` event, the filter checks both the incoming change flag and its internal change flag. If neither flag is set, the filter immediately passes the cached MST to the next stage. However, if either flag is set, the filter recomputes the MST and compares it with the cached version. If there is a difference, it updates the cached MST and propagates this information to the next stage, indicating that an update has occurred.

This optimization significantly reduces redundant computations by leveraging the stability of the MST across stages, thus enhancing the overall efficiency of the dynamic pipeline. The Adaptive MST Caching ensures that computational resources are utilized more effectively by only performing necessary updates, thereby improving performance in scenarios with frequent but minor changes.

Memory management

During the execution of some experiments, we observed that the system tended to run out of memory before processing the input graph, even though the graph theoretically fit in memory. Profiling techniques revealed an issue with memory management in the `Go` runtime. Figure 3.5 illustrates an example where a graph with 384,585 edges, which should occupy approximately 4.4 MB, consumes significantly more memory.

`Go` provides automatic dynamic memory management with garbage collection. This feature enhances program security by enabling runtime verification, bounds checking, and other benefits. However, garbage collection introduces a performance overhead, which is usually negligible. When a programmer defines an object that is placed on the heap, the required amount of memory is allocated, and a pointer to it is returned. Once the object is no longer needed, the memory region is marked for collection, and the garbage

collector will eventually free it, depending on the language. [2]

In `Go`, the internal structure representing a slice consists of a pointer to the heap where the elements are stored, an integer indicating the number of elements in the slice, and another integer representing the actual size of the underlying array. When a new element is appended to the slice and the number of elements equals the slice size, a new region of memory, twice the previous size, is allocated, and all elements are copied to this new region[3]. Since elements are appended within Kruskal's loop, numerous increases occur in a single call to Kruskal. Given the parallel execution, this results in multiple copies in each filter, leading to a substantial memory allocation.

Several solutions were considered to address this issue. The simplest was to force the garbage collector to run after every `mst` event. However, this approach is inefficient as it halts all goroutines until the garbage collection completes. A more effective solution involved transferring the slice ownership to the worker. At the beginning of Kruskal's algorithm, the slice size is reset to zero, preventing memory from being freed. Additionally, this solution reduces the number of copies, as the returned MST will generally be of similar size at each filter.

[2]A detailed explanation of the garbage collector implementation can be found in https://github.com/golang/go/blob/master/src/runtime/mgc.go#L5

[3]More detials about its internals in https://go.dev/blog/slices-intro

Chapter 4

Experimental Study

This chapter details the comprehensive experimental study conducted to evaluate the performance of the `DP_Kruskal` algorithm. We begin by discussing the test cases, which include experiments on random graphs and how they have been created, real dynamic graphs; and the implementation of a hot start technique to enhance efficiency without compromising results.

Next, we describe the running environment and validate the random graph generator to ensure the reliability of our experiments. The chapter then delves into various aspects of the `DP_Kruskal` design, including input reading, edge ordering, Decoupled Event Handling, filter size, and pre-processing steps.

Following this, we present a comparative analysis of the `DP_Kruskal` algorithm against other algorithms on both static and real dynamic graphs. The section on static graphs examines the performance of parallel algorithms, while the section on real dynamic graphs highlights the effectiveness of the `DP_Kruskal` algorithm in maintaining the MST over time.

Through these experiments, we aim to provide a thorough evaluation of the `DP_Kruskal` algorithm, demonstrating its strengths and identifying areas for further optimization.

4.1 Generation of Test Cases

We have two different suites of test cases. One of synthetic random graphs which are static and the other one of real dynamic graphs.

We generated random graphs where each edge has a probability p of being present of different number of vertices using the Algorithm 7. For each combination of nodes $(10^3, 2 \cdot 10^3, 10^4, 5 \cdot 10^4)$ and edge probability $(0.10, 0.25, 0.50, 0.75, 0.9)$, 20 random graphs were generated. This suite allows to understand different relationships between edges, densities and distribution along the pipeline.

Additionally, we obtained some realistic dynamic graphs (Table 4.1) collected by the authors in [24] and available in `https://DynGraphLab.github.io/`. We had to adapt

them to our format and some of them did not have weights or they were artificially generated by the repository, so we added a weight picked uniformly at random in the range 0 to 1.

Dataset	Vertices	Operations
amazon-ratings	495,452	476,728
as-caida	31,379	19,468
frwiki	2,212,682	31,624,375
movielens10m	49,847	384,585
simplewiki	100,312	889,016

Table 4.1: Propoerties of the real dynamic graphs used

Random graphs algorithm

A binomial random graph is constructed by connecting labeled nodes randomly with each edge being included in the graph with probability p, independently from every other edge. This type of random graphs was presented by Gilbert in [23], but the proposed implementation in Algorithm 7 is based on [7]. The idea is to generate a random number to determine how many edges to skip based on the probability p.

Algorithm 7 Generating a binomial random graph

procedure RANDOMGRAPH(N, p)
  $G \leftarrow \varnothing$
  $v \leftarrow 1$
  $u \leftarrow -1$
  while $v < N$ **do**
    $u \leftarrow u + 1 + \lfloor \frac{\log(1-\text{Rand}())}{\log(1-p)} \rfloor$
    while $u \geq v \wedge v < N$ **do**
      $u \leftarrow u - v$
      $v \leftarrow v + 1$
    end while
    if $v < N$ **then**
      $G \leftarrow G \cup \{(u, v)\}$
    end if
  end while
  return G
end procedure

Random generator validation

Our graphs and operations generator utilizes the `Go` programming language's `math/rand` package. This package employs the ChaCha8 algorithm, a well-regarded stream cipher designed by Daniel Bernstein [9]. The generator produces 63-bit numbers which are then scaled by dividing by 2^{63}, resulting in uniformly distributed random numbers between 0

and 1. Further details on the internal workings of the generator can be found on the `Go` Blog[1].

To ensure the quality and reliability of this pseudo-random number generator, we conducted several validation tests. We generated over a million random numbers using the seed '1707242827149724922' and performed two statistical tests: the Chi-square test and the Kolmogorov-Smirnov test.

The Chi-square test [34, 40] is a statistical method used to compare the distribution of a sample of random numbers to an expected uniform distribution. This test involves dividing the range of possible values into several bins, counting the number of random numbers that fall into each bin, and then computing the Chi-square statistic to assess how well the observed distribution matches the expected distribution. A p-value is then calculated; in our case, we used a significance level of 0.05. If the p-value is below this threshold, we reject the null hypothesis that the numbers are uniformly distributed. Our test results indicated no rejection with a p-value of 0.001, confirming that the distribution of our generated numbers was consistent with uniformity at the 0.05 significance level.

The Kolmogorov-Smirnov (K-S) test is another statistical test used to compare a sample distribution to a reference probability distribution (in our case, the uniform distribution) [30, 34]. This test calculates the maximum distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. A p-value is then derived from this maximum distance. For our generated numbers, we performed the K-S test and obtained a p-value of 0.02. Since this p-value is above the common significance threshold of 0.05, we do not reject the null hypothesis, indicating that our random number generator produces numbers that follow the expected uniform distribution.

Both the Chi-square test and the Kolmogorov-Smirnov test results validate the high quality and reliability of the `math/rand` pseudo-random number generator in Go for our application needs.

Hot start

In our experimental study, we implemented two additional operations in the `DP_Kruskal` algorithm specifically for static graph testing: `savestate` saving the current state of the filters and $\widehat{\text{loadstate}}$ loading this saved state. These operations enable what we term a "Hot Start", allowing the algorithm to resume from a pre-saved state rather than initializing from scratch for each test run. This method significantly enhances efficiency during experimentation without compromising the validity of the results. By using Hot Starts, we can isolate the performance of the algorithm itself from the overhead of repeated initializations, providing a clearer assessment of the algorithm's behavior under various conditions.

These test cases were designed to evaluate the performance improvements achieved by the optimizations in `DP_Kruskal` and to ensure that the results are both accurate and reproducible.

[1] https://go.dev/blog/randv2

4.2 Analysis of DP_Kruskal Architecture and Optimizations

In this section, we delve into the experimentation and optimization of the DP_Kruskal algorithm. Through a series of tests, we examine various design elements including input handling, edge ordering, event decoupling, filter sizing, and pre-processing. Our goal is to refine the pipeline architecture and enhance its performance for computing and maintaining the MST.

Read Input

As discussed in Section 3.3, Go provides two standard library packages for handling program input and output. The fmt package implements formatted I/O with functions similar to C's printf and scanf, while the bufio package implements buffered I/O. bufio wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but adds buffering and assists with textual I/O.

Since bufio uses buffers to read from a file and includes specific readers for various data types without needing a parser to infer the format as scanf does, it intuitively offers better performance than fmt. To verify this, we generated a file with the same format as our input graphs and measured the time taken to read it. This experiment was repeated 100 times to obtain stable time measurements.

Figure 4.1 shows that, as expected, bufio outperformed fmt. On average, bufio required around 1.04 microseconds per line, whereas fmt needed 18.69 microseconds per line. This indicates that using bufio instead of fmt resulted in a speed-up of approximately 18 times. Consequently, due to these results, bufio was chosen for our implementation.

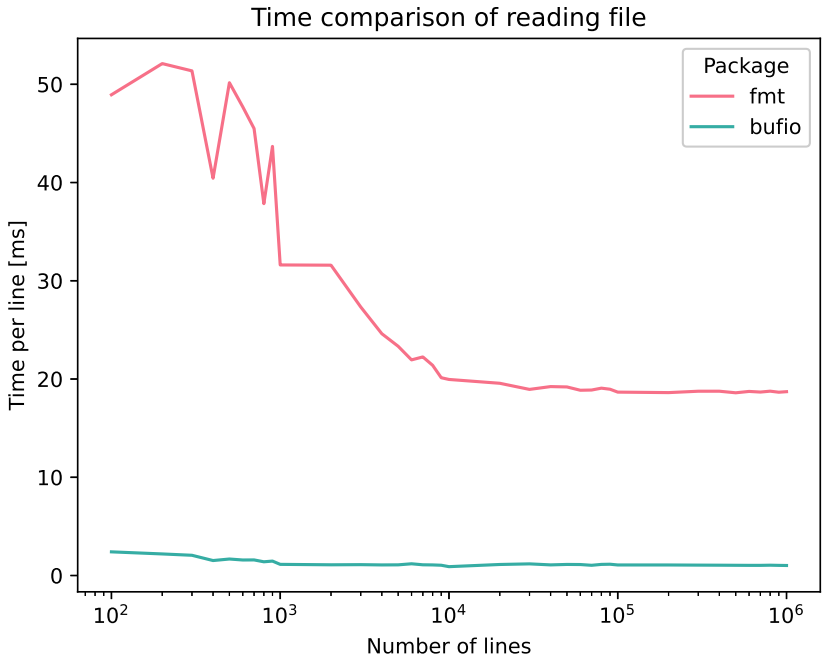


Figure 4.1: Running time comparison between bufio and fmt

Edge order

We observed that the ordering of the vertices in an edge has a significant impact on the overall performance of the system. Several possible orderings can be proposed, such as a random order of the vertices, a lexicographical increasing order, or an order based on the vertex that has appeared the least number of times, and many more. Each of these strategies or heuristics has its own advantages and complexities.

For our analysis, we focused on two ordering methods: random order and lexicographical order. These methods were chosen because they are relatively simple to implement and do not require prior knowledge of the graph's history. Instead of performing the analysis on the complete `DP_Kruskal`, we created a simplified model to simulate the behavior of the pipeline. This model allowed us to count where each filter ended up and provided insights into the performance impact of each ordering strategy. The code used for simulate can be found in Appendix A and Appendix B.

Figure 4.2 illustrates the simulation results for a random graph with 100,000 vertices and a density of 0.9. Figure 4.3 shows the simulation results for a real dynamic graph with 2,212,682 vertices and 31,624,375 operations. In both cases, we observed that the lexicographical order resulted in fewer filters, which helps to reduce the overhead associated with managing multiple goroutines. However, this reduction in filters comes at a cost: the remaining filters tend to accumulate more edges, leading to more costly operations.

The results indicate that while the lexicographical order can help streamline the pipeline by reducing the number of filters, it can also increase the workload on each filter. This trade-off highlights the importance of carefully choosing the vertex ordering strategy based on the specific characteristics and requirements of the application. Further research and experimentation may reveal additional strategies that balance these trade-offs more effectively. But at this point of time, we chose the lexicographical order for the rest of the experiments.

At this point in time, we chose the lexicographical order for the rest of the experiments. This decision was based on its demonstrated ability to minimize the number of filters, thereby reducing the goroutine overhead despite the increased workload on each filter.

Decoupled Event Handling

As detailed in Section 3.4, the MailBox/Worker optimization, termed "Decoupled Event Handling", was introduced to address inefficiencies in the original filter stage implementation. By separating the event handling and processing responsibilities into distinct stages —the MailBox and the Worker- we aimed to enhance the overall performance and responsiveness of the system.

To evaluate the impact of this optimization, we conducted an experiment that involved generating files of different sizes and measuring the execution time with and without the optimization. The results are shown in Table 4.2. For a file with 7,500 random requests, we observed a speed-up of 20% with the optimization. When the file size increased to 10,000 random requests, the speed-up increased to 38%. These results clearly indicate that the optimization significantly reduces the locking of the pipeline, allowing

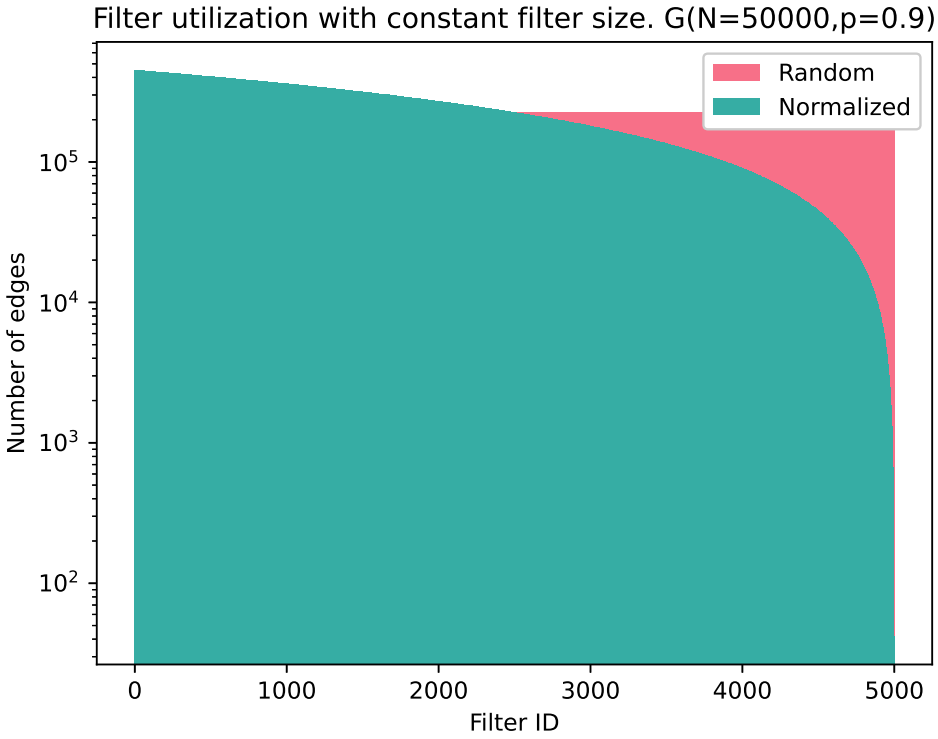


Figure 4.2: Simulation of the `DP_Kruskal` with 10 roots per filter using a random graph.

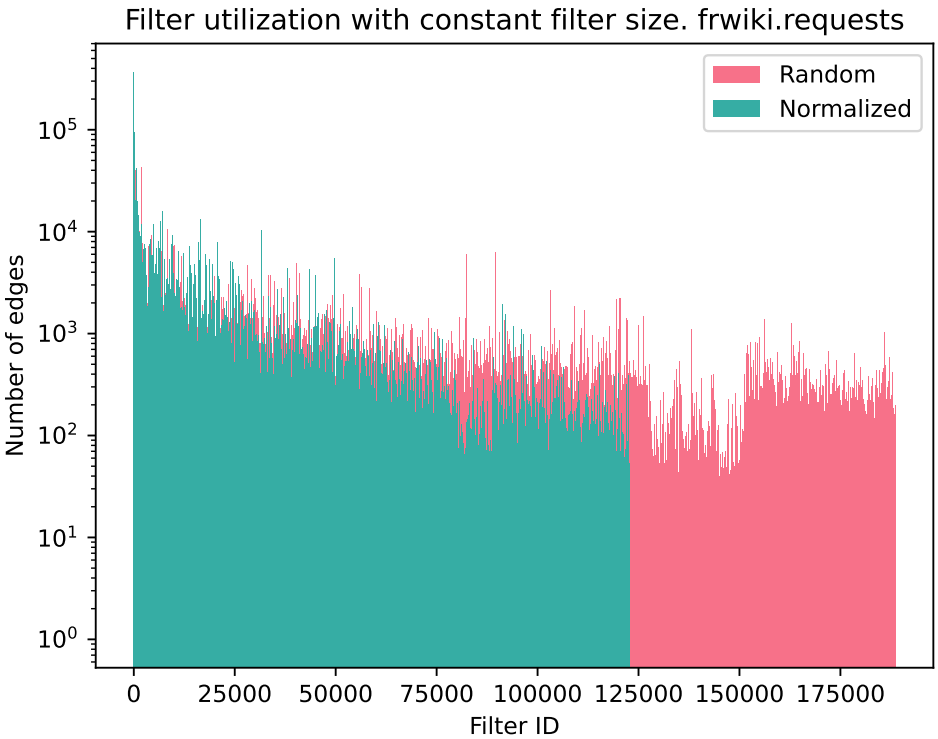


Figure 4.3: Simulation of the `DP_Kruskal` with 10 roots per filter using a real dynamic graph.

for more efficient processing as the number of operations increases. The more operations there are, the more pronounced the performance improvement becomes, demonstrating the effectiveness of the MailBox/Worker optimization in handling larger workloads and enhancing system throughput.

Operations	No optimization	With optimization
7500	1h 9m 31.48s	57m 28.23s
10000	3h 46m 46.73s	2h 44m 14.19s

Table 4.2: Execution time of `DP_Kruskal` of different file sizes with and without the MailBox/Worker optimization.

Filter size

As mentioned in Section 3.4, having multiple roots per filter can lead to a more efficient utilization of computational resources, but it also brings an additional parameter that can affect the performance of the `DP_Kruskal`.

To study the effect of the number of roots in each filter, we consider three options: a constant number of roots `DP_Kruskal_const`, $\log(n)$ roots `DP_Kruskal_log`, and $\sqrt{n}$ roots `DP_Kruskal_sqrt` (where n is the number of vertices of the input graph). We will initialize the `DP_Kruskal` and then measure the performance in a single core machine in order to compare their behaviour.

Figure 4.4 shows the corresponding experimental results. We can observe that, as expected, this value influences the performance of the algorithm. For small graphs it seems that the version that works with filters with a $\sqrt{n}$ number of roots is better and faster than the others, but as the graph becomes larger and denser the versions with a logarithmic or even constant number of roots per filter become more efficient.

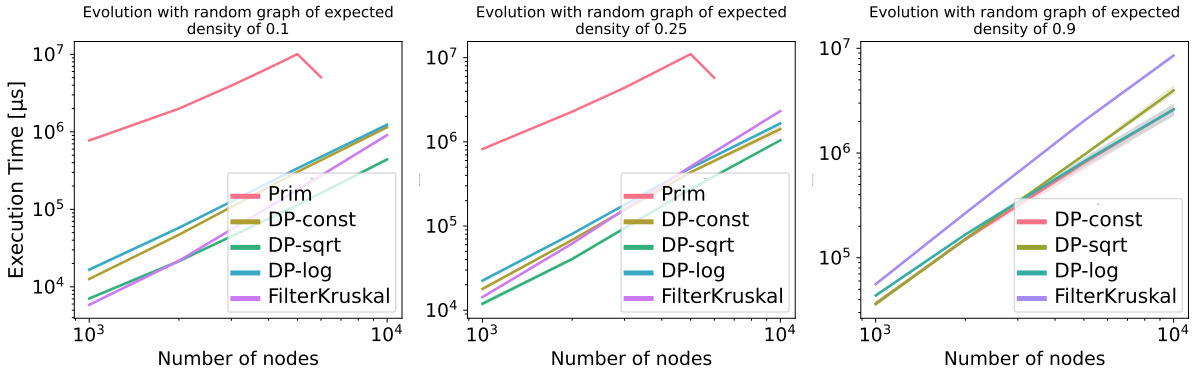


Figure 4.4: A comparison of three versions of the `DP_Kruskal` algorithm varying the number of roots at each filter: `DP_Kruskal_const`, `DP_Kruskal_log` and `DP_Kruskal_sqrt` with a single core

This trend is more evident when we group the results by different densities and the expected number of edges, computed as $n \cdot (n-1) \cdot p$. Figure 4.5 illustrates that there

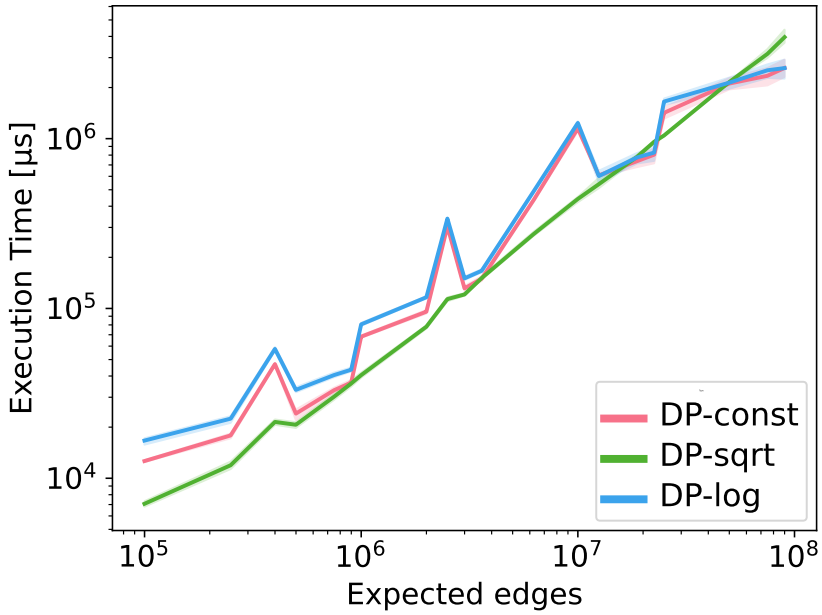


Figure 4.5: A comparison of three versions of the `DP_Kruskal` algorithm varying the number of roots at each filter and focusing on the number of edges: `DP_Kruskal_const`, `DP_Kruskal_log` and `DP_Kruskal_sqrt` with a single core

is a clear point at which the efficiency of the $\log(n)$ and constant root versions surpasses that of the $\sqrt{n}$ version. This transition point depends on the graph size and density, highlighting the importance of choosing an appropriate number of roots based on the specific characteristics of the input graph.

Pre-processing

Some experimental work [2, 12, 21, 29] comparing different algorithms to maintain a dynamic MST involves a pre-processing step, which is typically considered insignificant. These studies often compare data structures based on Euler Trees [54], which require vertices to have a maximum degree of 3. To meet this requirement, vertices with a degree greater than 3 are "exploded" into multiple vertices, equal to the initial vertex's degree, and connected in a loop with edges of weight 0. This ensures that all but one edge will be added during processing. An example of this process is shown in Figure 4.6.

To verify the insignificance of this pre-processing step and to determine whether the `DP_Kruskal` could benefit from it, we conducted our own experiments. Figure 4.7 demonstrates that this pre-processing step actually results in a higher initialization time compared to processing without it. For instance, in a binomial graph originally consisting of 1000 vertices with a density of 0.75, transforming the graph so that no vertex has a degree greater than 3 resulted in a 100% slow-down (The average time with preprocess divided by the time without it). Similarly, for a graph with 7500 vertices and the same density, this transformation led to a 75% slow-down.

These results indicate that the pre-processing time is not negligible and that this approach is not advantageous for the `DP_Kruskal`.

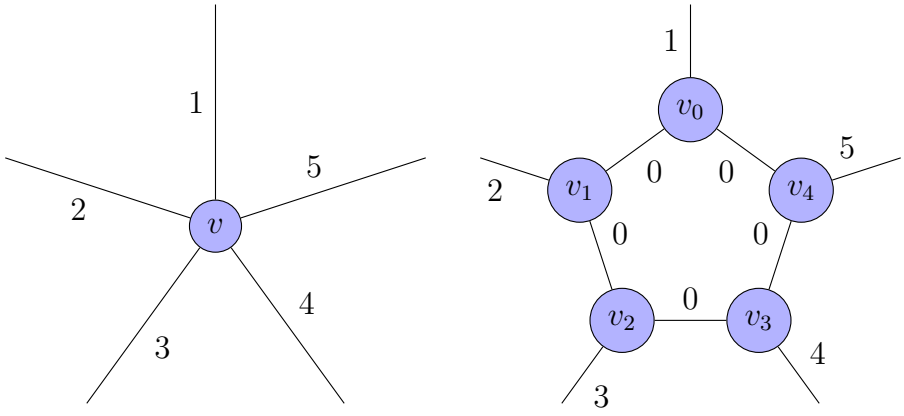


Figure 4.6: Example of the pre-processing step on a vertex with degree 5.

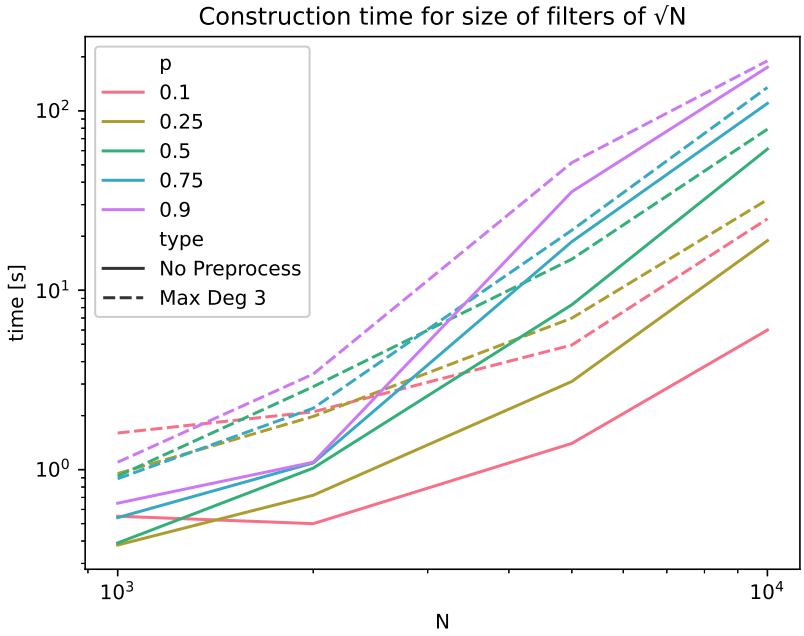


Figure 4.7: Comparison of `DP_Kruskal` initialization with and without preprocessing.

4.3 Comparison with other algorithms

In this section, as usual in parallel applications [6] we have captured the elapsed wall-clock time $T(k, n, p)$, which is the time elapsed from the moment in which the first processor started working to the moment in which the last processor completed the computation for a graph of n vertices and expected density of p using k processors. We will refer to absolute speed up and efficiency as:

- *Absolute speedup*: $speedup_a(k) = T(1, n, p)/T(k, n, p)$.
- *Efficiency*: $speedup_a(k)/k$.

where $T(1, n, p)$ is the execution time of the MST sequential implementation.

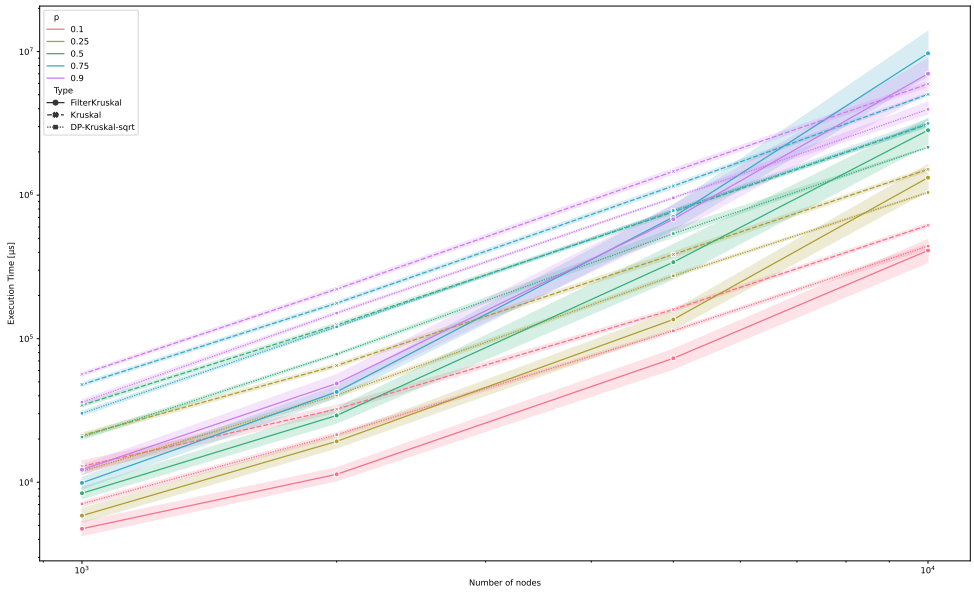


Figure 4.8: A comparison of DP-MST with different algorithms with a single core.

Static graphs

In Figure 4.8, we compare the performance of the Dynamic Pipeline (`DP_Kruskal`) against the standard Kruskal algorithm and the Filter-Kruskal (`Filter_Kruskal`) algorithm, as explained in Section 2.1. The results clearly demonstrate that `DP_Kruskal` offers a significant reduction in execution time across all graph densities. This indicates that the partitioning strategy utilized by `DP_Kruskal` is highly effective, providing superior performance regardless of the density of the graph. This efficiency is evident from the consistently lower execution times of `DP_Kruskal` compared to `Kruskal`.

Parallel algorithms:

If we focus on the algorithms that can be easily parallelized, we compare `DP_Kruskal` with our implementation of `Filter_Kruskal` and the results presented in [37] of an implementation of `Prim` using message-passing parallelism. In Figure 4.9, we can observe that `Prim` is far from optimal. Although `Filter_Kruskal` is faster at the beginning, for larger graphs it is easily outperformed by `DP_Kruskal`. This comparison highlights the superior scalability and efficiency of `DP_Kruskal` in handling large, dynamic graphs. Our experiments demonstrate that while `Filter_Kruskal` can provide competitive performance on smaller datasets, its efficiency diminishes as the graph size increases.

In evaluating the performance of `DP_Kruskal` and `Filter_Kruskal` on random graphs, the experimental results reveal significant insights into the scalability and efficiency of these algorithms. By examining the speed-up achieved with an increasing number of cores, as illustrated in Figure 4.10, we observe that `DP_Kruskal` consistently demonstrates superior parallel performance compared to `Filter_Kruskal`. For instance, in graphs with varying sizes and densities —such as $G(n = 1000, p = 0.25)$, $G(n = 5000, p = 0.75)$, and $G(n =$

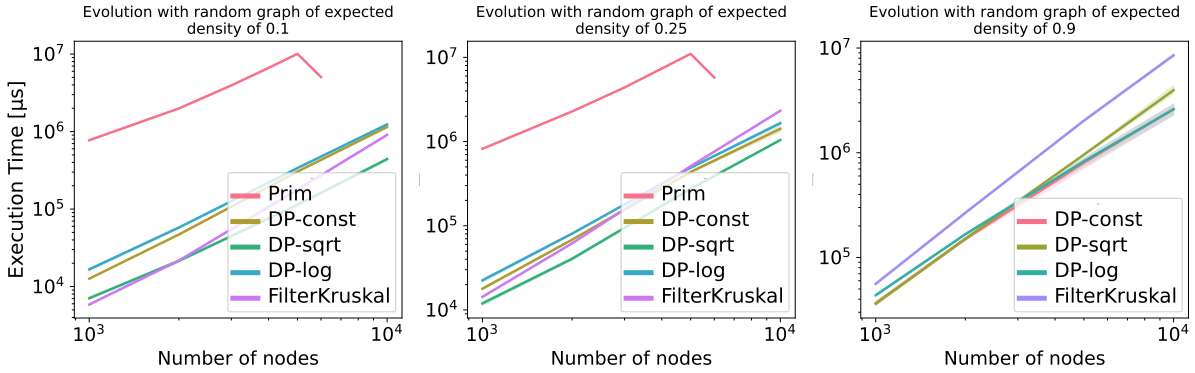


Figure 4.9: Comparison of `DP_Kruskal` with `DP_Kruskal_const(10)`, `DP_Kruskal_log`, `DP_Kruskal_sqrt`, `Prim` and `Filter_Kruskal` on a single core.

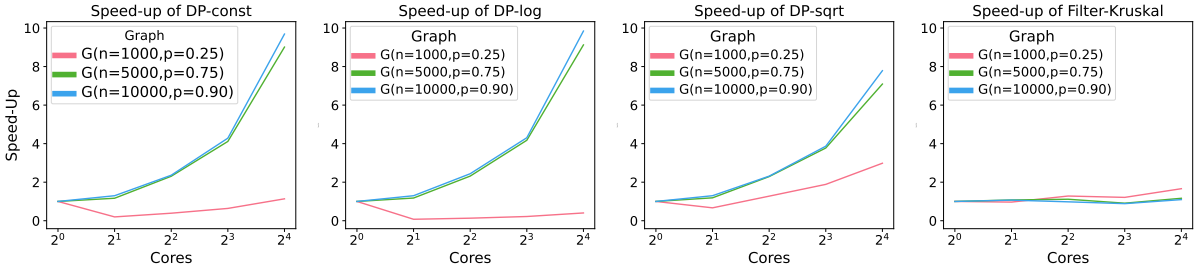


Figure 4.10: Speed-up comparison of `DP_Kruskal` and `Filter_Kruskal`

$10000, p = 0.90$)— `DP_Kruskal` exhibits substantial speed-ups across all configurations. Notably, `DP_Kruskal` with a logarithmic number of roots per filter (`DP_Kruskal_log`) achieves near-linear speed-up and almost a perfect efficiency (Figure 4.11) as the number of cores increases, highlighting its effectiveness in leveraging parallelism. In contrast, while `Filter_Kruskal` also benefits from parallel execution, its performance gains are comparatively modest, particularly for larger and denser graphs. This underscores the efficiency of the `DP_Kruskal` architecture in distributing computational workload and minimizing synchronization overhead, thereby making it a highly scalable solution for computing MST in parallel environments.

Real dynamic graphs

We compare `DP_Kruskal` and `Filter_Kruskal` on realistic dynamic graphs to evaluate their performance in maintaining the minimum spanning tree (MST) after each edge insertion or deletion. For each operation, we request an update of the MST to assess how efficiently the algorithms handle dynamic changes.

In Table 4.12a, we observe that `DP_Kruskal` effectively maintains the MST across various datasets. This table highlights the execution times for different graph sizes and densities, illustrating how `DP_Kruskal` consistently outperforms `Filter_Kruskal` in maintaining the MST.

Figure 4.12b further demonstrates the scalability and efficiency of `DP_Kruskal` in a parallel processing environment. By leveraging the parallelism capabilities inherent in its design, `DP_Kruskal` achieves significant speed-ups, especially as the number of processing

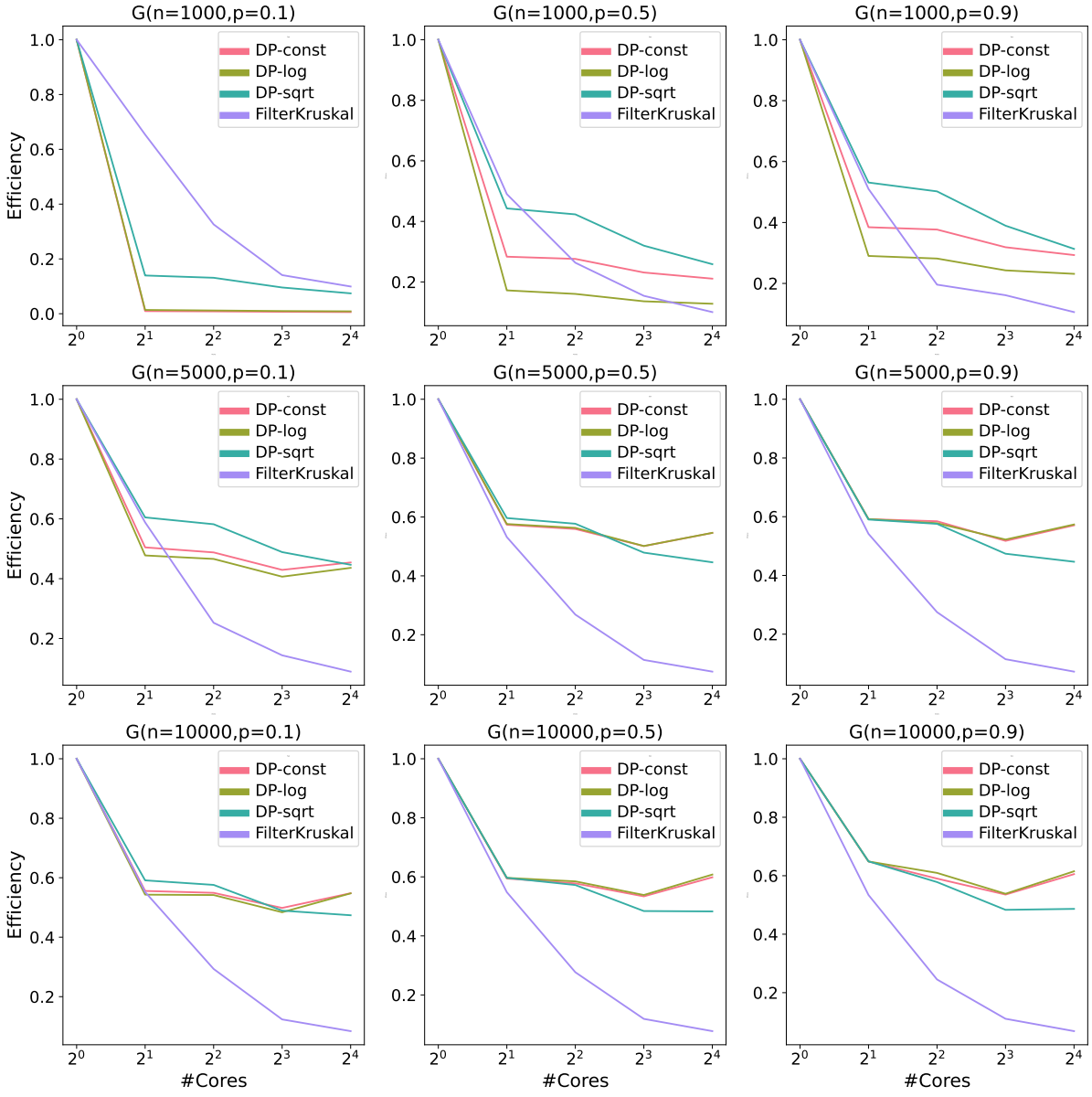
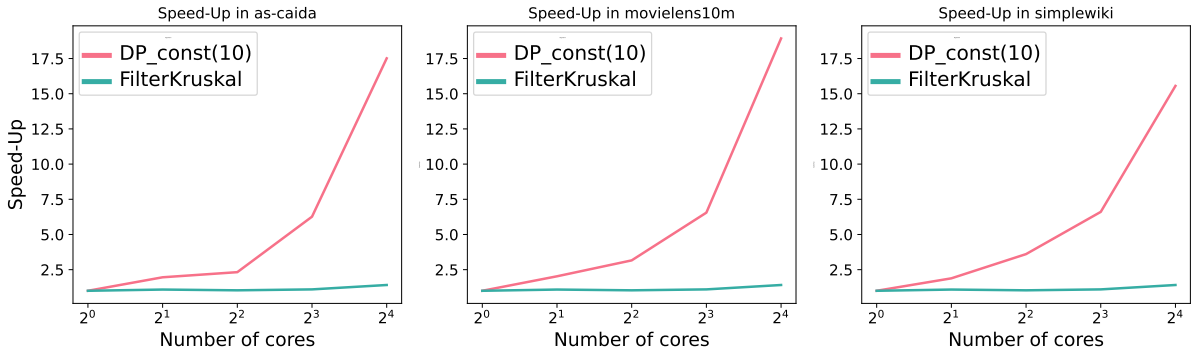


Figure 4.11: Efficiency comparison of `DP_Kruskal` and `Filter_Kruskal`

Dataset	`Filter_Kruskal`	`DP_Kruskal`
`as-caida`	1h 30min	1h 19min
`movielens10m`	1h 39min	1h 20min
`simplewiki`	17h 8min	11h 29min

(a) Execution time comparison with 1 core and 10 roots per filter



(b) Comparison of speed-ups in a multicore set-up

Figure 4.12: Experimental results on real dynamic graphs.

cores increases. This figure clearly shows that `DP_Kruskal` scales well with the addition of more cores, maintaining its performance advantage over `Filter_Kruskal` even as the computational load grows.

These results underscore the robustness and efficiency of `DP_Kruskal` in real-world scenarios, where dynamic updates to the MST are frequent and computational resources need to be optimally utilized. The combination of effective maintenance of the MST and superior scalability in parallel environments makes `DP_Kruskal` a compelling choice for handling dynamic graph problems in various applications.

4.4 Running environment

Experiments were run at the cluster of the RDLab-UPC (`https://rdlab.cs.upc.edu/`) on different nodes with the processors Intel(R) Xeon(R) CPU X5675 @ 3.07GHz and 12 cores, Intel(R) Xeon(R) CPU X5670 @ 2.93GHz and 12 cores, Intel(R) Xeon(R) CPU X5660 @ 2.80GHz and 12 cores, Intel(R) Xeon(R) CPU X5550 @ 2.67GHz and 8 cores, and Intel(R) Xeon(R) CPU E5-2450 @ 2.50GHz and 16 cores. The configuration used for submitting jobs was up to 16GB of RAM and a maximum number of cores depending on the experiment. The same job was executed 10 times and the average was reported. The timeout was 24 hours.

Chapter 5

Conclusions and Future Work

During this project, we have studied the suitability of the *Dynamic Pipeline Approach* (DPA) for providing a parallelization of Kruskal's algorithm for computing the Minimum Spanning Tree (MST) and the capabilities of the framework to maintain the MST of dynamic graphs.

The *underlying forest* representation and proposition 1 allowed to propose a possible algorithm called `DP_Kruskal`. The chosen language, `Go`, facilitated a natural implementation of `DP_Kruskal`, which allowed for a detailed structural analysis and the proposal of several optimizations. One significant improvement was storing multiple roots per filter, which enhanced the utilization of computational resources by minimizing the creation of excessive goroutines, albeit introducing a new design parameter. Crucially, separating message passing from MST maintenance emerged as a pivotal optimization, applicable across all DPA implementations, significantly enhancing parallelization. This optimization not only addressed inefficiencies but also uncovered and resolved a memory management issue inherent in `Go`. Furthermore, it enabled the introduction of a new optimization: caching the MST to prevent redundant computations.

With these improvements, we conducted an extensive experimental comparison of the `DP_Kruskal` algorithm against `Filter_Kruskal` and a message-passing implementation of `Prim`'s algorithm. The results clearly demonstrated that `DP_Kruskal` outperforms its competitors in single-core performance and exhibits far superior scalability in parallel environments.

In summary, `DP_Kruskal` proved to be a highly effective algorithm for computing and maintaining an MST with substantial parallelization, surpassing other algorithms in both performance and scalability. This project validates the DPA's potential for solving complex graph problems and highlights its broader applicability in parallel computing and provides critical optimizations for the framework itself.

There are many open research directions to explore. For instance, our experiments did not determine the point at which the speed-up stalls due to hardware limitations, so it would be interesting to conduct the same experiments on larger machines. An implementation that allows communication between processes, such as MPI, could be beneficial for experiments with more cores.

Despite the existence of several algorithms for maintaining an MST of dynamic graphs, we could not find any open-source implementations, neither sequential nor parallel. Therefore, implementing and comparing these algorithms against `DP_Kruskal` could be valuable. Some existing algorithms only handle partially dynamic graphs (only insertions or deletions). Modifying the `DP_Kruskal` implementation to leverage such partially dynamic graphs could be advantageous. Another research avenue is to study if data structures proposed for fully dynamic graphs can be used efficiently within each filter and how they compare with just Kruskal's algorithm. Additionally, a more detailed analysis of how the number of roots affects performance would be interesting, as we observed that it depends significantly on the size of the graph and impacts overall performance. A detailed examination of the effect of edge order and proposing more heuristics could further improve performance.

We provided a rough upper bound analysis of the execution time of `DP_Kruskal`. Employing more sophisticated techniques to refine this analysis and achieve a tighter bound could lead to enhanced implementations. One potential approach is competitive analysis with an adversary model [32]. An online algorithm processes its input incrementally, much like `DP_Kruskal`. Competitive analysis compares the performance of an online algorithm against an optimal offline algorithm that has full knowledge of the input sequence in advance. This method has proven effective in various domains, including online algorithms for paging [55], scheduling [4], and distributed algorithms [3]. Applying these techniques to `DP_Kruskal` could yield valuable insights and further optimization opportunities.

Bibliography

[1] "6. Minimum Spanning Trees". In: *Data Structures and Network Algorithms*, pp. 71–83. DOI: 10.1137/1.9781611970265.ch6. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970265.ch6. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970265.ch6.

[2] Giuseppe Amato, Giuseppe Cattaneo, and Giuseppe F. Italiano. "Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms". In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '97. New Orleans, Louisiana, USA: Society for Industrial and Applied Mathematics, 1997, pp. 314–323. ISBN: 0898713900.

[3] James Aspnes. "Competitive analysis of distributed algorithms". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 118–146. ISBN: 9783540683117. DOI: 10.1007/bfb0029567. URL: http://dx.doi.org/10.1007/BFb0029567.

[4] Baruch Awerbuch, Shay Kutten, and David Peleg. "Competitive distributed job scheduling (extended abstract)". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '92. Victoria, British Columbia, Canada: Association for Computing Machinery, 1992, pp. 571–580. ISBN: 0897915119. DOI: 10.1145/129712.129768. URL: https://doi.org/10.1145/129712.129768.

[5] David A Bader and Guojing Cong. "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs". In: *Journal of Parallel and Distributed Computing* 66.11 (2006), pp. 1366–1378.

[6] David A Bader, Bernard ME Moret, and Peter Sanders. "Algorithm engineering for parallel computation". In: *Experimental Algorithmics*. Springer, 2002, pp. 1–23.

[7] Vladimir Batagelj and Ulrik Brandes. "Efficient generation of large random networks". In: *Physical Review E* 71.3 (Mar. 2005). ISSN: 1550-2376. DOI: 10.1103/physreve.71.036113. URL: http://dx.doi.org/10.1103/PhysRevE.71.036113.

[8] Mohammad Hossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, Mohammad Taghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. "Affinity clustering: hierarchical clustering at scale". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6867–6877. ISBN: 9781510860964.

[9] Daniel J Bernstein et al. "ChaCha, a variant of Salsa20". In: *Workshop record of SASC*. Vol. 8. 1. Citeseer. 2008, pp. 3–5.

[10] Otakar Borůvka. "O jistém problému minimálním". In: (1926).

[11] Paul Burkhardt and Christopher A Waring. "A cloud-based approach to big graphs". In: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2015, pp. 1–8.

[12] Giuseppe Cattaneo, Pompeo Faruolo, U Ferraro Petrillo, and Giuseppe F Italiano. "Maintaining dynamic minimum spanning trees: An experimental study". In: *Discrete Applied Mathematics* 158.5 (2010), pp. 404–425.

[13] Sun Chung and Anne Condon. "Parallel implementation of Borůvka's minimum spanning tree algorithm". In: *Proceedings of International Conference on Parallel Processing*. IEEE. 1996, pp. 302–308.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. ISBN: 978-0-262-04630-5. URL: http://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

[15] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 1557-7317. DOI: 10.1145/1327452.1327492. URL: http://dx.doi.org/10.1145/1327452.1327492.

[16] Suryanarayana Murthy Durbhakula. "Parallel Minimum Spanning Tree Algorithms and Evaluation". In: *arXiv preprint arXiv:2005.06913* (2020).

[17] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. "Sparsification –a technique for speeding up dynamic graph algorithms". In: *Journal of the ACM (JACM)* 44.5 (1997), pp. 669–696.

[18] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Thomas H Spencer. "Separator-based sparsification II: Edge and vertex connectivity". In: *SIAM Journal on Computing* 28.1 (1998), pp. 341–381.

[19] David Eppstein, Giuseppe F Italiano, Roberto Tamassia, Robert E Tarjan, Jeffery Westbrook, and Moti Yung. "Maintenance of a minimum spanning forest in a dynamic plane graph". In: *Journal of Algorithms* 13.1 (1992), pp. 33–54.

[20] Greg N Frederickson. "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees". In: *SIAM Journal on Computing* 26.2 (1997), pp. 484–538.

[21] Greg N. Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications". In: *SIAM Journal on Computing* 14.4 (1985), pp. 781–798. DOI: 10.1137/0214055. eprint: https://doi.org/10.1137/0214055. URL: https://doi.org/10.1137/0214055.

[22] Bernard A. Galler and Michael J. Fisher. "An improved equivalence algorithm". In: *Commun. ACM* 7.5 (May 1964), pp. 301–303. ISSN: 0001-0782. DOI: 10.1145/364099.364331. URL: https://doi.org/10.1145/364099.364331.

[23] E. N. Gilbert. "Random Graphs". In: *The Annals of Mathematical Statistics* 30.4 (1959), pp. 1141–1144. DOI: 10.1214/aoms/1177706098. URL: https://doi.org/10.1214/aoms/1177706098.

[24] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. "Recent Advances in Fully Dynamic Graph Algorithms – A Quick Reference Guide". In: *ACM J. Exp. Algorithmics* 27 (Dec. 2022). ISSN: 1084-6654. DOI: 10.1145/3555806. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3555806.

[25] F. Harary and G. Gupta. "Dynamic graph models". In: *Mathematical and Computer Modelling* 25.7 (Apr. 1997), pp. 79–87. ISSN: 0895-7177. DOI: 10.1016/s0895-7177(97)00050-2. URL: http://dx.doi.org/10.1016/S0895-7177(97)00050-2.

[26] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 723–760.

[27] Jacob Holm, Eva Rotemberg, and Christian Wulff-Nilsen. "Faster Fully dynamic Minnimum Spanning Forest". In: *In Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras Greece, September 14–16 2015, Proceedings (Lecture Notes in Computer Science Vol. 9294), Nikhil Bansal and Irene Finocchi (Eds.)* Ed. by Springer. 2015, pp. 742–753.

[28] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)* (2022), pp. 1–2163. DOI: 10.1109/IEEESTD.2022.10004498.

[29] Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. "An Experimental Study of Polylogarithmic, Fully Dynamic, Connectivity Algorithms". In: *ACM J. Exp. Algorithmics* 6 (Dec. 2002), 4–es. ISSN: 1084-6654. DOI: 10.1145/945394.945398. URL: https://doi.org/10.1145/945394.945398.

[30] Frank J. Massey Jr. "The Kolmogorov-Smirnov Test for Goodness of Fit". In: *Journal of the American Statistical Association* 46.253 (1951), pp. 68–78. DOI: 10.1080/01621459.1951.10500769. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1951.10500769. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500769.

[31] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. "A Model of Computation for MapReduce". In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Jan. 2010. DOI: 10.1137/1.9781611973075.76. URL: http://dx.doi.org/10.1137/1.9781611973075.76.

[32] Richard M. Karp. "On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future?" In: *IFIP Congress*. 1992. URL: https://api.semanticscholar.org/CorpusID:37205055.

[33] Abdul Atif Khan and Sraban Kumar Mohanty. "A fast spectral clustering technique using MST based proximity graph for diversified datasets". In: *Information Sciences* 609 (2022), pp. 1113–1131. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2022.07.101. URL: https://www.sciencedirect.com/science/article/pii/S0020025522007903.

[34] Donald Knuth. *Art of Computer Programming, the: Seminumerical Algorithms, volume 2*. Pearson Education (US), 1997.

[35] Joseph B. Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50. ISSN: 1088-6826. DOI: 10.1090/s0002-9939-1956-0078686-7. URL: http://dx.doi.org/10.1090/S0002-9939-1956-0078686-7.

[36] Ning Li, J.C. Hou, and L. Sha. "Design and analysis of an MST-based topology control algorithm". In: *IEEE Transactions on Wireless Communications* 4.3 (2005), pp. 1195–1206. DOI: 10.1109/TWC.2005.846971.

[37] Vladimir Lončar, Srdjan Škrbić, and Antun Balaž. "Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures". In: *Transactions on Engineering Technologies*. Springer Netherlands, 2014, pp. 543–554. ISBN: 9789401788328. DOI: 10.1007/978-94-017-8832-8_39. URL: http://dx.doi.org/10.1007/978-94-017-8832-8_39.

[38] Xiaodong Long and Jian Sun. "Image segmentation based on the minimum spanning tree with a novel weight". In: *Optik* 221 (2020), p. 165308. ISSN: 0030-4026. DOI: https://doi.org/10.1016/j.ijleo.2020.165308. URL: https://www.sciencedirect.com/science/article/pii/S003040262031144X.

[39] Daniel Lugosi Enes. "Concurrent Implementation of Multidimensional Range Queries". UPC, Facultat d'Informàtica de Barcelona, Departament de Ciències de la Computació, July 2019. URL: http://hdl.handle.net/2117/169246.

[40] M. Donald MacLaren and George Marsaglia. "Uniform Random Number Generators". In: *J. ACM* 12.1 (Jan. 1965), pp. 83–89. ISSN: 0004-5411. DOI: 10.1145/321250.321257. URL: https://doi.org/10.1145/321250.321257.

[41] M.J. Neely, E. Modiano, and C.E. Rohrs. "Dynamic power allocation and routing for time-varying wireless networks". In: *IEEE Journal on Selected Areas in Communications* 23.1 (2005), pp. 89–103. DOI: 10.1109/JSAC.2004.837349.

[42] Krzysztof Nowicki and Krzysztof Onak. "Dynamic graph algorithms with batch updates in the massively parallel computation model". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 2939–2958.

[43] Vitaly Osipov, Peter Sanders, and Johannes Singler. "The Filter-Kruskal Minimum Spanning Tree Algorithm". In: *2009 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2009, pp. 52–61. DOI: 10.1137/1.9781611972894.5. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972894.5. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972894.5.

[44] Gopal Pandurangan, Peter Robinson, Michele Scquizzato, et al. "The distributed minimum spanning tree problem". In: *Bulletin of EATCS* 2.125 (2018).

[45] Edelmira Pasarella, Maria-Esther Vidal, and Cristina Zoltan. "Comparing MapReduce and Pipeline Implementations for Counting Triangles". In: *PROLE*. 2017. URL: https://api.semanticscholar.org/CorpusID:11937418.

[46] Edelmira Pasarella, Maria-Esther Vidal, Cristina Zoltan, and Juan Pablo Royo Sales. "A computational framework based on the dynamic pipeline approach". In: *Journal of Logical and Algebraic Methods in Programming* 139 (2024), p. 100966. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2024.100966. URL: http://dx.doi.org/10.1016/j.jlamp.2024.100966.

[47] Mihai Patrascu and Erik D. Demaine. "Logarithmic Lower Bounds in the Cell-Probe Model". In: *SIAM Journal of Computing* 35.4 (2006), pp. 932–963.

[48] R. C. Prim. "Shortest connection networks and some generalizations". In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401. DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[49] Celso C. Ribeiro and Rodrigo F. Toso. "Experimental Analysis of Algorithms for Updating Minimum Spanning Trees on Graphs Subject to Changes on Edge Weights". In: *Experimental Algorithms*. Ed. by Camil Demetrescu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 393–405. ISBN: 978-3-540-72845-0.

[50] J. Royo, Edelmira Pasarella, C. Zoltán, and Maria-Esther Vidal. "Towards a dynamic pipeline framework implemented in (parallel) Haskell". In: *Actas de las XX Jornadas de Programación y Lenguajes (PROLE 2021)*. Sociedad de Ingeniería de Software y Tecnologías de Desarrollo de Software (SISTEDES), 2021, pp. 1–17. URL: http://hdl.handle.net/2117/365477.

[51] Juan Pablo Royo Sales. "An algorithm for incrementally enumerating bitriangles in large bipartite networks". UPC, Facultat d'Informàtica de Barcelona, Departament de Ciències de la Computació, Oct. 2021. URL: http://hdl.handle.net/2117/361615.

[52] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. "The future is big graphs: a community view on graph processing systems". In: *Communications of the ACM* 64.9 (2021), pp. 62–71.

[53] Maria J. Serna. *Lecture notes in Algorithmics*.

[54] Daniel D. Sleator and Robert Endre Tarjan. "A data structure for dynamic trees". In: *Journal of Computer and System Sciences* 26.3 (1983), pp. 362–391. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(83)90006-5. URL: https://www.sciencedirect.com/science/article/pii/0022000083900065.

[55] Daniel D. Sleator and Robert E. Tarjan. "Amortized efficiency of list update and paging rules". In: *Commun. ACM* 28.2 (Feb. 1985), pp. 202–208. ISSN: 0001-0782. DOI: 10.1145/2786.2793. URL: https://doi.org/10.1145/2786.2793.

[56] Chenhao Tan, Jie Tang, Jimeng Sun, Quan Lin, and Fengjiao Wang. "Social action tracking via noise tolerant time-varying factor graphs". In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '10. Washington, DC, USA: Association for Computing Machinery, 2010, pp. 1049–1058. ISBN: 9781450300551. DOI: 10.1145/1835804.1835936. URL: https://doi.org/10.1145/1835804.1835936.

[57] John Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. "Temporal distance metrics for social network analysis". In: *Proceedings of the 2nd ACM Workshop on Online Social Networks*. WOSN '09. Barcelona, Spain: Association for Computing Machinery, 2009, pp. 31–36. ISBN: 9781605584454. DOI: 10.1145/1592665.1592674. URL: https://doi.org/10.1145/1592665.1592674.

[58] Robert E. Tarjan and Jan van Leeuwen. "Worst-case Analysis of Set Union Algorithms". In: *J. ACM* 31.2 (Mar. 1984), pp. 245–281. ISSN: 0004-5411. DOI: 10.1145/62.2160. URL: https://doi.org/10.1145/62.2160.

[59] Alok Ranjan Tripathy and BNB Ray. "A New Parallel Algorithm for Minimum Spanning Tree (MST)". In: *International Journal of Advanced Studies in Computers, Science and Engineering* 2.5 (2013), p. 7.

[60] Leslie G. Valiant. "A bridging model for parallel computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: https://doi.org/10.1145/79173.79181.

[61] Jan Wassenberg, Wolfgang Middelmann, and Peter Sanders. "An Efficient Parallel Algorithm for Graph-Based Image Segmentation". In: *Computer Analysis of Images and Patterns*. Ed. by Xiaoyi Jiang and Nicolai Petkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1003–1010. ISBN: 978-3-642-03767-2.

[62] Cristina Zoltan, Ana Edelmira Pasarella Sánchez, Julián Arturo Aráoz Durand, and Maria-Esther Vidal. "The dynamic pipeline paradigm". In: 2019. URL: https://api.semanticscholar.org/CorpusID:208111957.

Appendix A

Code for simulate static random graphs in the DP_Kruskal

```
from math import exp, log
import random

def simulate_random_graph(N,p, normalize):
  filters = [0]
  roots = [0]
  mapToFilter = dict()
  v = 1
  u = -1
  while v < N:
    u += 1 + int( log(1 - random.random()) / log(1 - p))

    while u >= v and v < N:
      u = u - v
      v = v + 1
    if v < N: # Edge is in the graph and should be added to DP
      e = (u,v)
      # Normalization operation
      if normalize:
        if e[0] > e[1]:
            e[0], e[1] = e[1], e[0]
      else:
        # Random order of vertices
        if random.choice([True, False]):
            e[0], e[1] = e[1], e[0]

        if e[0] not in mapToFilter: # If vertex has not been assigned to a filter yet
            if roots[-1] == Fsize: # If last filter has reached its maximum, we should make a new one
                filters.append(0)
                roots.append(0)
            roots[-1] += 1
            mapToFilter[e[0]] = len(filters) - 1
        filters[mapToFilter[e[0]]] += 1

  return filters
```

Appendix B

Code for simulate real dynamic graphs in the DP_Kruskal

```
from math import exp, log
import random

def simulate(path, normalize):
    filters = [dict()]
    mapToFilter = dict()

    file = open(path, "r")
    line = file.readline()

    V = set()
    E = set()

    while line:
        line = line.split()
        for i in range(3):
            line[i] = int(line[i])
        V.add(line[1])
        V.add(line[2])
        # Normalization operation
        if normalize:
          if line[1] > line[2]:
              line[1], line[2] = line[2], line[1]
        else:
          # Random order of vertices
          if random.choice([True, False]):
              line[1], line[2] = line[2], line[1]

        if line[0] == 1: # Insert Operation
            E.add((line[1],line[2]))
            if line[1] not in mapToFilter: # If vertex has not been assigned to a filter yet
                if len(filters[-1]) == Fsize: # If last filter has reached its maximum, we should make a new one
                    filters.append(dict())
                filters[-1][line[1]] = set()
                mapToFilter[line[1]] = len(filters) - 1
            filters[mapToFilter[line[1]]][line[1]].add(line[2])
        elif line[1] == 2:
            # Delete Operation
            if line[1] in mapToFilter and line[2] in filters[mapToFilter[line[1]]][line[1]]:
                filters[mapToFilter[line[1]]][line[1]].remove(line[2])
        line = file.readline()

    density = [ sum([ len(filter[root]) for root in filter ])  for filter in filters ]
    return density
```