A Stream Processing Based Algorithm for Maintaining Minimum Spanning Forests of Evolving Graphs

Daniel Benedí, Amalia Duch and Edelmira Pasarella

Universitat Politècnica de Catalunya, Barcelona Tech, Barcelona, Spain

Abstract

In this work we study the feasibility of the Dynamic Pipeline concurrent approach of computation to maintain updated the minimum spanning tree (or forest) of evolving graphs. To do so we propose a concurrent/parallel adaptation of Kruskal algorithm since it has been shown to be the fastest algorithm in practice dealing with evolving graphs. So, in our algorithm, the input (undirected) graph is given by a stream of weighted edges and it is dynamically created and maintained in a distributed way along a pipeline of *stateful* stages. We show experimentally that our algorithm scales well to a large number of processes and that it is competitive i) against the classic sequential Kruskal algorithm and ii) a well known parallelized version of Kruskal algorithm. Our experimental study is done for a large class of evolving graphs both real as well as randomly generated –including several densities and sizes.

Keywords

Spanning trees, Kruskal algorithm, Stream processing, Evolving graphs, Dynamic pipeline approach

1. Introduction

"The Future is Big Graphs" states the CACM paper [1] 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 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 claim –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 arrives (stream model). Additionally, one of the challenges that these experts identified is to define a reference architecture for big graph processing. So, in this work we study the feasibility and suitability of the Dynamic Pipeline approach [2, 3] (DPA from now on) to support the new requirements of nowadays graphs.

Abusing terminology, we will indistinctly say evolutionary graphs or dynamic graphs, although the first is a slightly broader term than the second since it also includes graphs whose changes depend on a function of time (specific case, the latter, which we will not include in this work, but such that could be included with slight modifications to our algorithm).

AMW 2024: 16th Alberto Mendelzon International Workshop on Foundations of Data Management, September 30th–October 4th, 2024, Mexico City, Mexico

daniel.benedi@upc.edu (D. Benedí); duch@cs.upc.edu (A. Duch); edelmira@cs.upc.edu (E. Pasarella)
0009-0005-4295-9484 (D. Benedí); 0000-0003-4371-1286 (A. Duch); 0000-0001-8315-4977 (E. Pasarella)

The study of dynamic graphs and, in particular, of algorithms that maintain a minimum spanning forest (MST, from here on, abusing of notation) of the dynamic graph is not new. In the extensive survey of Hanauer, Henzinger and Schulz [4] there is a reasoned state of the art on these algorithms that divides them clearly into theoretic 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. The lower bound of $\Omega(\log n)$ for the time per operation on graphs of n vertices, given in [5], for connectivity extends to maintaining MSTs. Holm et al. [6] 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 [7], that is still far from the lower bound. The experimental work of Cattaneo et al. [8] provide an extensive experimental study that assesses different approaches: [9, 10, 11, 12, 13, 7] and shows that, except for very particular instances, a simple $O(m \log n)$-time algorithm, as Kruskal 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. [7]. 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 [14] 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. [8], yielding the fastest experimental algorithm for the above mentioned graphs.

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 [15, 16, 17]. 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. [15] implement three parallel variants of Borůvka's MST algorithm [18] arguing that this algorithm is more naturally paralelizable than Prim's and Kruskal's ones. The authors in [15] claim that –by the first time– they present a general parallel MST algorithm that runs efficiently in the SMP computer architecture[1]. 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 memory-bound problem, MST distributed algorithms have been proposed [19] where it is necessary to deal with message passing overhead, memory latency and fault-tolerance issues [20]. For fully dynamic graphs with sequences of updates of (restricted) sub-linear size (with respect to the graph size) MST algoritm –based on the massively parallel computation (MPC) computation model (MapReduce) [21] has been presented in [22] and, up to our knowledge, there are no more parallel and /or distributed algorithms dealing with the maintenance of the MST of fully dynamic graphs.

Summarizing all what we argued so far, we claim that current applications require algorithms to keep updated the MST of evolving graphs (the existing algorithms in parallel and distributed models are mainly for static graphs) that are competitive from a theoretical point of view,

[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.

susceptible to be implemented efficiently and scalable to maintain their efficiency when the graphs to be treated are large (existing algorithms have been tested only for small graphs, it has been experimentally proven that they are not useful in practice and that the theoretical predictions are not met).

Therefore, in this work we provide a parallel and concurrent algorithm –the `DP_Kruskal` algorithm– that maintains continuously along time the evolution of fully dynamic graphs together with their corresponding MSTs. `DP_Kruskal` keeps the graph distributed along a pipeline, according to the DPA [2, 3]. Additionally, we show experimentally that the `DP_Kruskal` algorithm introduced here is competitive in practice specially when dealing with dense graphs –where other algorithms fail– and that the $O(m \log n)$ running time per operation (when dealing with graphs of n vertices and m edges) of the best previous algorithms in practice is a very pessimistic (and unreachable) upper bound of our algorithm. Moreover, under this framework the computation of the Dynamic_MST problem turns out to be more natural and intuitive as well as more efficient in several cases since the algorithm discards –early in its execution– several edges.

Our work is organised as follows. In next section (Section 2) we introduce our algorithm together with its fundamental features, extensions and functioning. In Section 3 we discuss the results of our experimental study. Finally, in Section 4, we give our conclusions and lines for further work.

2. The `DP_Kruskal` Algorithm

An evolving graph $G = (V, E)$ is a graph (that might be weighted, in that case we assume that the weight of each edge e is a positive real value) that supports the following update operations:

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

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

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

Since operation (i) can be simulated by a deletion followed by an insertion, we focus our attention in operations (ii) and (iii). We call any instance of these operations an *event* and an evolving graph is therefore given (and/or defined) by a stream of events: the input of our `DP_Kruskal` algorithm.

The functioning of `DP_Kruskal` requires to represent the input graph as a partition of its edges. Any partition is valid but we use a representation based on the graph's *underlying forest*. In what follows we assume without loss of generality that the input graphs have no isolated vertices (vertices with no incident edges), although with slight modifications the algorithm could deal with them.

Definition 1. *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 vertex v_i.*

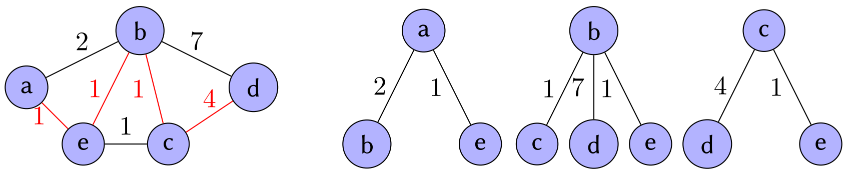

Figure 1: Example of graph G (extreme left). The edges marked in red belong to a MST_G. The sequence of trees on the right form an underlying forest of G, F_G.

In Figure 1 we show the weighted graph $G = (V, E)$ such that $V = \{a, b, c, d, e\}$ and $E = \{(a, b, 2), (a, e, 1), (b, e, 1), (c, e, 1), (b, d, 7), (c, d, 4), (b, c, 1)\}$ together with the sequence F_G, one of its possible underlying forests.

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 $G_i = \bigcup_{j=1}^{i} T_j$, $1 \leq i \leq k$.

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 an evolving graph (the edges in red in Figure 1 conform one of the possible MSTs of the graph therein). To do so, next proposition, that can be proved by induction on the size of the underlying forest, 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 $G_i = \cup_{j=1}^{i} T_j$, it holds that*

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

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

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 following example.

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 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. $\mathsf{MST}(G_3) = \mathsf{MST}(T_3 \cup \mathsf{MST}(G_2)) = \{(a, e, 1), (b, c, 1), (b, e, 1), (c, d, 4)\}$

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. For instance, the computation of $\mathsf{MST}_{F_G}(T_2)$ requires the tree T_2 but also the tree (or forest) corresponding to the MST of the previous trees in the sequence, T_1, in this case.

We connect the different stages of the pipeline by means of communication channels that are in charge of the synchronisation of the procedure.

We identify every tree T_i ($1 \leq i \leq k$) of F_G with a stage F_{v_i} ($1 \leq i \leq k$) of the pipeline that we call *filter*, and it is the filter corresponding to vertex v_i. In fact, it is possible to allow filters to contain as many vertices (and, in consequence, trees) as possible. This option does not change the algorithm, but shrinks or streches the longitude of the pipeline. So, further in our experiments we study the effect of having a constant number of vertices (`DP_Kruskal_const`), a $\log(n)$ number of filters (`DP_Kruskal_log`) and a $\sqrt{n}$ number of filters (`DP_Kruskal_sqrt`) for graphs of n vertices.

There are tree more kinds of stages in the pipeline: the *Input*, the *Output* and the *Generator*, all of them follow an ordering along the pipeline. The first one in the order is the Input stage, followed by the sequence of filters (that is empty when the process starts), then comes the Generator and finally the Output stage.

When a pipeline is initialised, only the Input, the Generator and the Output 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 `DP_Kruskal` terminates. Below we describe all the stages with more precision.

Data: The input data of the `DP_Kruskal` is a stream of events of the form $(\mathtt{op}, e)$, with $op \in \{\mathtt{insert}, \mathtt{delete}, \mathtt{mst}, \mathtt{eof}\}$ and $e \in E \cup \perp$ where $\perp$ is interpreted as an empty edge. When $\mathtt{op} \in \{\mathtt{insert}, \mathtt{delete}\}$ e is a weighted edge and $e = \perp$ in any other case. The event $(\mathtt{mst}, \perp)$ requests for the computation of the MST of the current graph while the event $(\mathtt{eof}, \perp)$ causes the deactivation of the `DP_Kruskal`.

Channels: There are two types of channels, the event channel and the graph channel carrying events and MSTs, respectively.

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

Filters (F_v) An instance $F(v)$ of the Filter stage stores F_v. Whenever an event $(\mathtt{op}, \mathtt{e})$ arrives to $F(v)$, the following things can happen: (i) If $\mathtt{op} = \{\mathtt{insert}, \mathtt{delete}\}$ and e is incident to v, the event is treated in $F(v)$ according to $\mathtt{op}$. In case of deleting, if the tree becomes empty $F(v)$ dies. If on the contrary, e is not incident to v, the event is passed through the event channel to the next stage of the pipeline; (ii) If $\mathtt{op} = \mathtt{mst}$, a partial MST is computed and passed to the rest of the DP_Kruskal through the graph channel by combining the MST that arrives through the graph channel together with the tree in $F(v)$ as stated by Proposition 1. The filter stage has been implemented with two different goroutines with a communication channel from the first to the second. The first goroutine is supposed to be light-weighted to avoid blocking the pipeline and it will only manage the event channel. In case $\texttt{op} = \{\texttt{insert}, \texttt{delete}\}$, the goroutine will check if the edge involved belongs to the filter. If so, it will send the event to the other goroutine, otherwise it will send the event to the next filter. If the event is $\texttt{op} = \texttt{mst}$, then it will be passed to the other goroutine and to the next filter. The second goroutine will perform the operation received from the first goroutine and in the case of $\texttt{op} = \texttt{mst}$ it will read the partial MST from the previous filter.

Generator (Gen): Whenever an event $(\texttt{op}, \texttt{e})$ arrives to the Generator stage by the event channel, depending on op, the following actions are taken: If $\texttt{op} \in \{\texttt{insert}, \texttt{update}\}$ and $e = (v_1, v_2, w)$, a new instance of Filter stage, $F(v_1)$, is spawned and added to DP_Kruskal between the last filter and Gen. The edge $e = (v_1, v_2, w)$ is stored in the local memory of $F(v_1)$. Besides, whenever $\texttt{op} = \texttt{mst}$, it passes the MST that arrives by the graph channel to the output stage O.

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

So, back to Example 1, when the request to insert edge $(a, b, 2)$ of G arrives to I it is passed through the event channel of the pipeline. Since there are no filters yet, it arrives to the generator that creates the first filter with root a. Afterwards, a request to insert edge $(a, e, 1)$ arrives to I, it follows the same procedure as edge $(a, b, 2)$ but this edge passes first through filter F_a that keeps it. The request to insert edge $(b, c, 1)$ provokes the creation of a new filter and the algorithm goes on in this way until all the edges are inserted. Afterwards, the request for the MST traverse the filters keeping the trees listed in Example 1. The first steps of this evolution are shown in Figure 2. As we have already observed, the order in which the edge insertion requests arrive into the input stream determines the configuration of the pipeline and may affect the efficiency of the algorithm.

3. Experimental Results

In order to evaluate the performance of DP_Kruskal we have implemented it in Golang 1.20 (https://golang.org/). All the programs are available at the GitHub repository and the details of the implementation can be found in [23, 24]. We have experimented with two different types of dynamic graphs: random graphs generated by ourselves and real graphs taken from the repository:https://DynGraphLab.github.io/ [4].

As is standard in parallel applications [25], in all our experiments, we recorded the elapsed wall-clock time $T(k, n, m)$—the time elapsed from the start of the execution of DP_Kruskal by the first processor until its end by the last processor—for a graph with n vertices and m edges using k processors. Notably, all experiments assumed no prior knowledge of the incoming graph, unlike the classic fully dynamic model where the number of vertices is known in advance, allowing ad-hoc optimizations.

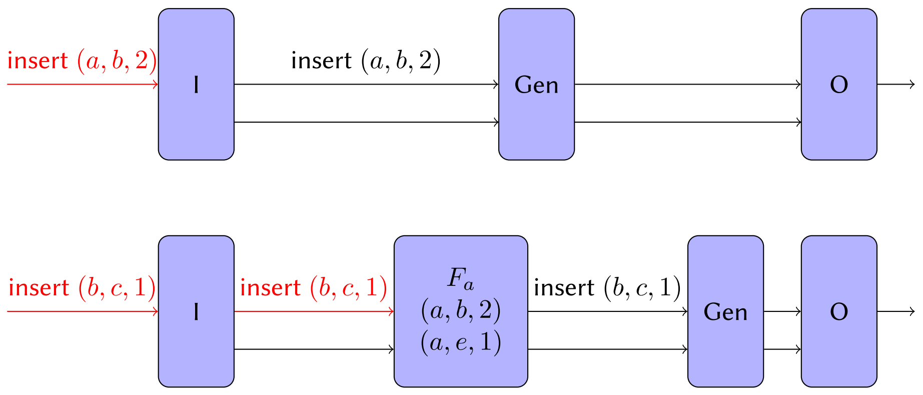

Figure 2: First two steps on the creation of the dynamic pipeline of the graph G shown in Figure 1.

Static graphs: We generated random graphs where each edge has a probability p of being present, as introduced by Gilbert [26]. Instead of visiting every edge individually, we used Batagelj's method [27], which involves generating a random number to determine how many edges to skip based on the probability p. 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. 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 3, 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 $G(n = 10000, p = 0.90)$— `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) 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. It is worth noting that the performances of `DP_Kruskal_log` and `DP_Kruskal_const` are very similar. This is because for very large graphs as the ones in the experiments the exact value of $\log(n)$ is so close (and equal in some cases) to the chosen constant value to appreciate any significant difference. In order to observe a substantial difference in performance n should have be much huger.

These results underscore 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 graphs: We compare `DP_Kruskal` and `Filter_Kruskal` on realistic dynamic graphs (from `https://DynGraphLab.github.io/` [4]) 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

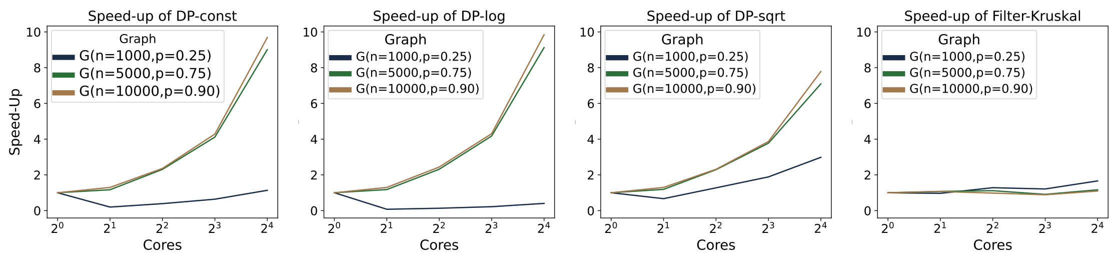

Figure 3: Speed-up comparison of `DP_Kruskal` and `Filter_Kruskal`

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

Table 1
Real dynamic graphs. Execution time comparison in single core set-up

changes. In order to not assume any prior knowledge of the input graphs, in the forthcoming experiments, we consider only the `DP_Kruskal_const` version of our algorithm. Nevertheless, since this version presented the best performance for static graphs we can expect a similar performance for real ones.

In Table 1, 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`.

Figure 5 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 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.

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.

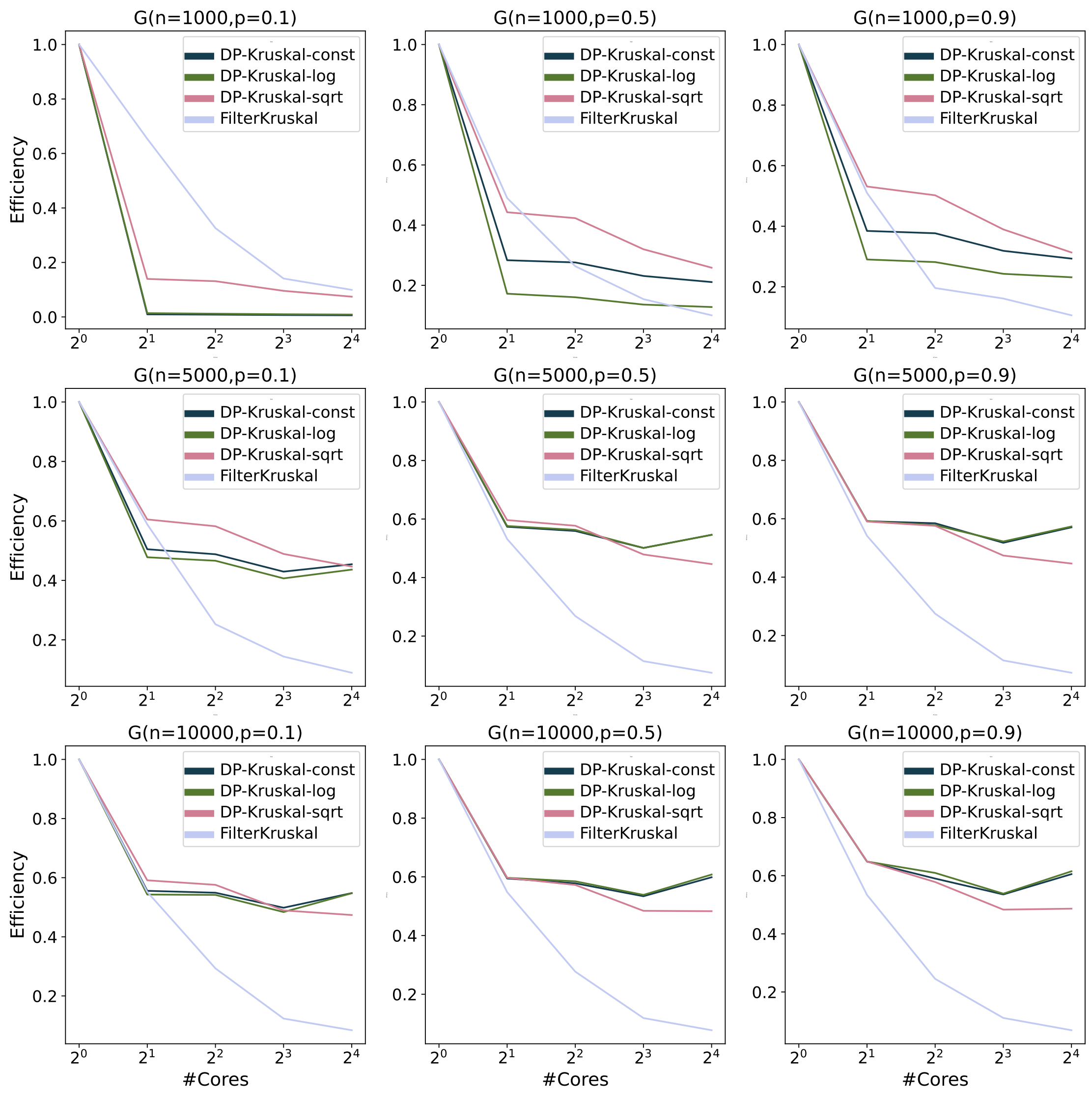

Figure 4: Efficiency comparison of `DP_Kruskal` and `Filter_Kruskal`

4. Final Remarks

We have studied the suitability of the `DP_Kruskal` model of computation to obtain the MST of fully dynamic graphs. To do so, we proposed an algorithm based on Kruskal's MST algorithm and implement it in the `Go` programming language –in order to take advantage of its built-in communication channels which allow to deal with dynamic pipelines in a very natural way.

We have conducted a series of preliminary experiments using two kinds of source data: a wide variety of random graphs and real graphs trying to cover a huge range of possible graph topologies and sizes. The results of these experiments show that our algorithm is competitive

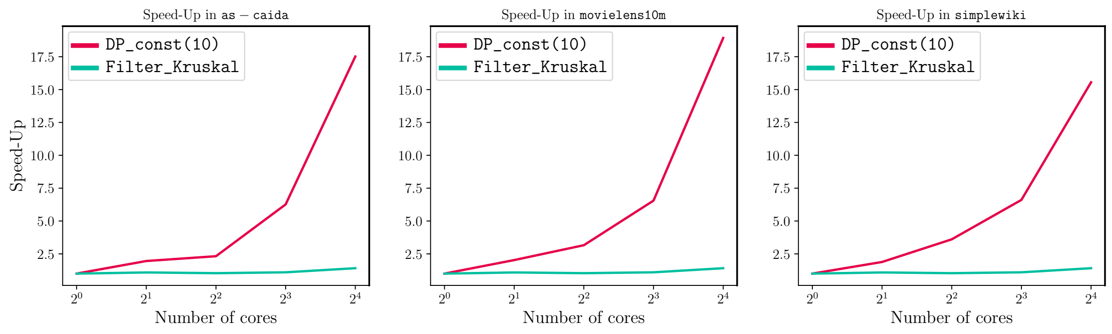

Figure 5: Real dynamic graphs. Comparison of speed-ups in a multicore set-up

specially for the case of dense graphs of any size. We have also observed that our algorithm is scalable and highly parallelisable; in particular in our experiments (i) for graphs from $5 \cdot 10^4$ edges up our algorithm has better performance metrics than the Filter-Kruskal algorithm and (ii) it seems that at least up to 16 cores it improves consistently its efficiency and speed up in both random and real graphs.

As future work, we plan to conduct further experiments according to the following guidelines: (i) To compare our algorithm against better baselines, specifically againts its unique (up to our knowledge) competitor in massive parallel computation (the MapReduce based algorithm [21]) although our algorithm has no restrictions on the number of updates, (ii) to evaluate other –apart from Kruskal's– MST algorithms and their adaptations to the `DP_Kruskal` model and compare the different algorithms among them, (iii) to evaluate the suitability of the programming language and consider other possibilities of implementation that allow the spawn of tasks, (iv) in order to study the scalability and real applicability of our model, we plan to conduct more exhaustive experiments with bigger real datasets and benchmarks, (v) finally, since there are also various dynamic graph models such as insertions-only algorithms, deletions-only algorithms, offline dynamic algorithms, algorithms with vertex insertions and deletions, kinetic algorithms and temporal algorithms, algorithms with a limit on the number of allowed queries, algorithms for the sliding-windows model, and algorithms for sensitivity problems (also called emergency planning or fault-tolerant algorithms) among others; it would be interesting to test the adaptability of our proposal to these approaches.

Acknowledgments

This work has been funded by MCIN/AEI/10.13039/501100011033 under grant PID2020-112581GB-C2.

References

[1] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al., The future is big graphs: a community view on graph processing

systems, Communications of the ACM 64 (2021) 62–71.

[2] C. Zoltan, E. Pasarella, J. Araoz, M. Vidal, The dynamic pipeline paradigm, 2019. URL: https://hdl.handle.net/11705/PROLE/2019/017.

[3] E. Pasarella, M.-E. Vidal, C. Zoltan, Comparing mapreduce and pipeline implementations for counting triangles, Electronic proceedings in theoretical computer science 237 (2017) 20–33.

[4] M. H. Kathrin Hanauer, C. Schulz, Recent advances in fully dynamic graph algorithms – a quick reference guide, ACM Journal of Experimental Algorithmics 27 (2022) 1–45. URL: https://doi.org/10.1145/3555806.

[5] M. Patrascu, E. D. Demaine, Logarithmic lower bounds in the cell-probe model, SIAM Journal of Computing 35 (2006) 932–963.

[6] J. Holm, K. De Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity, Journal of the ACM (JACM) 48 (2001) 723–760.

[7] J. Holm, E. Rotemberg, C. Wulff-Nilsen, Faster fully dynamic minnimum spanning forest, in: Springer (Ed.), 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.), 2015, pp. 742–753.

[8] G. Cattaneo, P. Faruolo, U. F. Petrillo, G. F. Italiano, Maintaining dynamic minimum spanning trees: An experimental study, Discrete Applied Mathematics 158 (2010) 404–425.

[9] D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig, Sparsification –a technique for speeding up dynamic graph algorithms, Journal of the ACM (JACM) 44 (1997) 669–696.

[10] D. Eppstein, Z. Galil, G. F. Italiano, T. H. Spencer, Separator-based sparsification ii: Edge and vertex connectivity, SIAM Journal on Computing 28 (1998) 341–381.

[11] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, M. Yung, Maintenance of a minimum spanning forest in a dynamic plane graph, Journal of Algorithms 13 (1992) 33–54.

[12] G. N. Frederickson, Data structures for on-line updating of minimum spanning trees, with applications, SIAM Journal on Computing 14 (1985) 781–798.

[13] G. N. Frederickson, Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees, SIAM Journal on Computing 26 (1997) 484–538.

[14] C. C. Ribeiro, R. F. Toso, Experimental analysis of algorithms for updating minnimum spanning trees on graphs subject to changes on edge weights, in: Springer (Ed.), In Experimental Algorithms, 6th International Workshop, WEA 2007, Rome Italy, June 6–8 2007, Proceedings (Lecture Notes in Computer Science Vol. 4525), Camil Demetrescu (Ed.), 2007, pp. 393–405.

[15] D. A. Bader, G. Cong, Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs, Journal of Parallel and Distributed Computing 66 (2006) 1366–1378.

[16] S. M. Durbhakula, Parallel minimum spanning tree algorithms and evaluation, arXiv preprint arXiv:2005.06913 (2020).

[17] A. R. Tripathy, B. Ray, A new parallel algorithm for minimum spanning tree (mst), International Journal of Advanced Studies in Computers, Science and Engineering 2 (2013) 7.

[18] S. Chung, A. Condon, Parallel implementation of Borůvka's minimum spanning tree algorithm, in: Proceedings of International Conference on Parallel Processing, IEEE, 1996, pp. 302–308.

[19] G. Pandurangan, P. Robinson, M. Scquizzato, et al., The distributed minimum spanning tree problem, Bulletin of EATCS 2 (2018).

[20] P. Burkhardt, C. A. Waring, A cloud-based approach to big graphs, in: 2015 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2015, pp. 1–8.

[21] H. Karloff, S. Suri, S. Vassilvitskii, A model of computation for mapreduce, in: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, SIAM, 2010, pp. 938–948.

[22] K. Nowicki, K. 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.

[23] D. Benedí, A. Duch, E. Pasarella, C. Zoltan, `DP_Kruskal`: a concurrent algorithm to maintain dynamically minimum spanning trees, 2024.

[24] D. B. García, Maintaining the minimum spanning forest of a fully dynamic graph in the dynamic pipeline pattern, 2024.

[25] D. A. Bader, B. M. Moret, P. Sanders, Algorithm engineering for parallel computation, in: Experimental Algorithmics, Springer, 2002, pp. 1–23.

[26] E. N. Gilbert, Random Graphs, The Annals of Mathematical Statistics 30 (1959) 1141 – 1144. URL: https://doi.org/10.1214/aoms/1177706098. doi:10.1214/aoms/1177706098.

[27] V. Batagelj, U. Brandes, Efficient generation of large random networks, Physical Review E 71 (2005). URL: http://dx.doi.org/10.1103/PhysRevE.71.036113. doi:10.1103/physreve.71.036113.