Maintaining the Minimum Spanning Forest of Fully Dynamic Graphs on the Dynamic Pipeline Approach

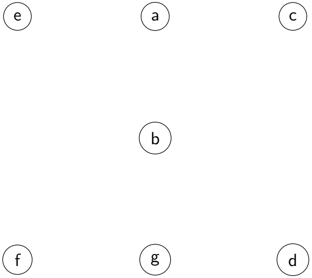
Daniel Benedí

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

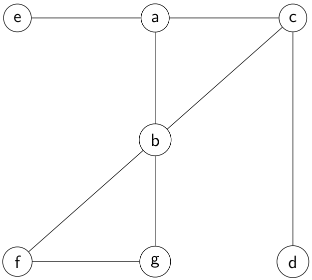
June, 2024

Overview

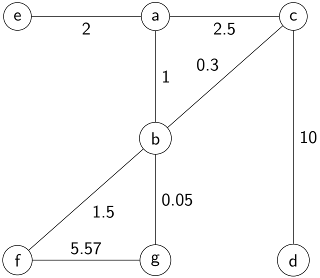
Preliminaries: Dynamic Graphs



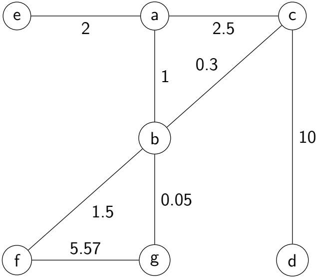
Preliminaries: Dynamic Graphs



Preliminaries: Dynamic Graphs

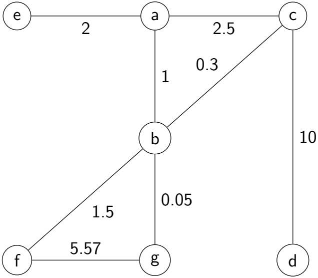


At the beginning:

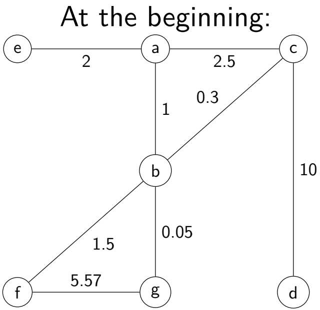


- Initial Graph

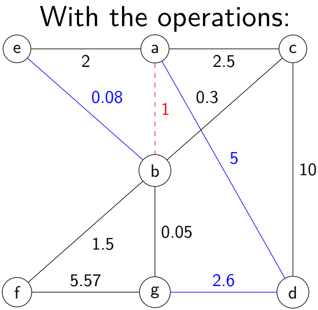
With the operations:



Preliminaries: Dynamic Graphs

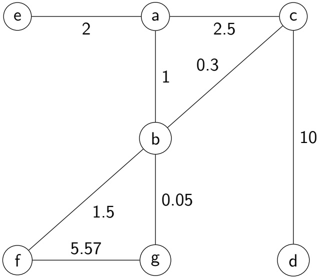


- Initial Graph
- INSERT,b,e,0.08
- INSERT,a,d,5
- INSERT,d,g,2.6
- DELETE,a,b



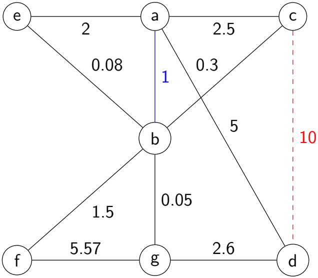
Preliminaries: Dynamic Graphs

At the beginning:



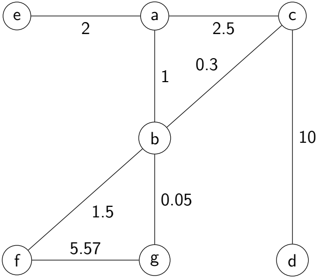
- Initial Graph
- `INSERT,b,e,0.08`
- `INSERT,a,d,5`
- `INSERT,d,g,2.6`
- `DELETE,a,b`
- `DELETE,c,d`
- `INSERT,a,b,1`

With the operations:



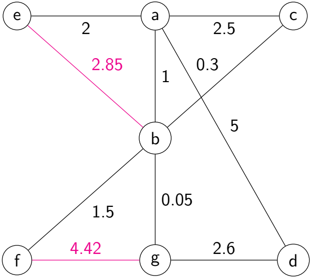
Preliminaries: Dynamic Graphs

At the beginning:



- Initial Graph
- `INSERT,b,e,0.08`
- `INSERT,a,d,5`
- `INSERT,d,g,2.6`
- `DELETE,a,b`
- `DELETE,c,d`
- `INSERT,a,b,1`
- `UPDATE,f,g,4.42`
- `UPDATE,b,e,2.85`

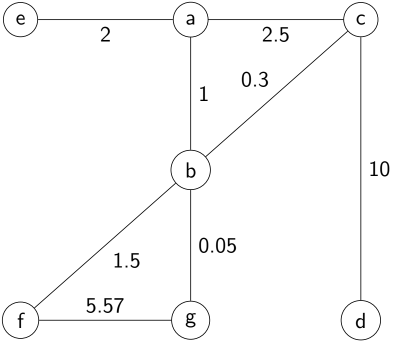
With the operations:

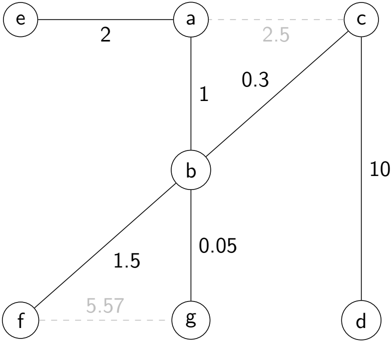


Preliminaries: Minimum Spanning Forest

Definition

A *spanning tree* T *of a graph* G is a subgraph that is a tree and includes all the vertices of G.





Preliminaries: Motivation

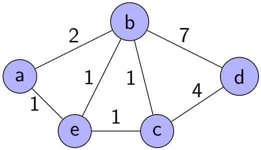
There are many theoretical and experimental results. Cattaneo et al. (2002) showed that simpler algorithm are faster in practice than those with better asymptotically behaviour. Those results used small graphs due to limitations of the date.

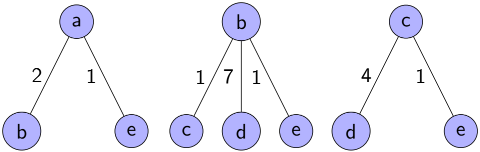
Almost no algorithms for maintaining dynamic MST*s* on concurrent or parallel set-ups. Up to our knowledge, only one with batch updates in MapReduce.

Graphs and MSTs in DPA: Underlying forest and MST (I)

Definition

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.





Graphs and MST*s* in DPA: Underlying forest and MST (II)

Proposition

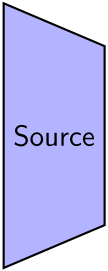
Given a weighted graph $G = (V, E)$ represented by the underlying forest $F_G = \langle T_1, \ldots, T_k \rangle$ ($k \geq 1$) and the subgraphs G_i, $1 \leq i \leq k$, of G such that $G_i = \cup_{j=1}^{k} T_i$, it holds that

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

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

Graphs and MSTs in DPA: Stages of Dynamic Pipeline

- **Source:** This stage manages the in-connection with the outside: streaming input, file reading, random generation...

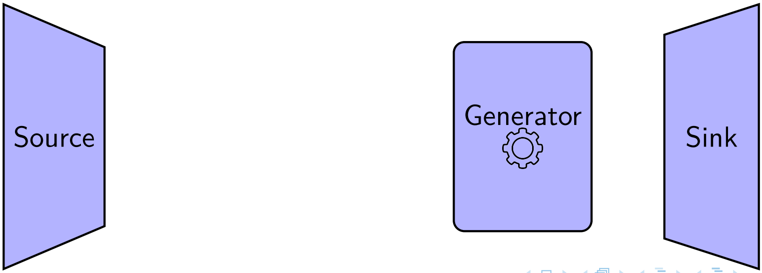


Graphs and MSTs in DPA: Stages of Dynamic Pipeline

- **Source:** This stage manages the in-connection with the outside: streaming input, file reading, random generation...
- **Sink:** This stage manages the out-connection. Usually stdout

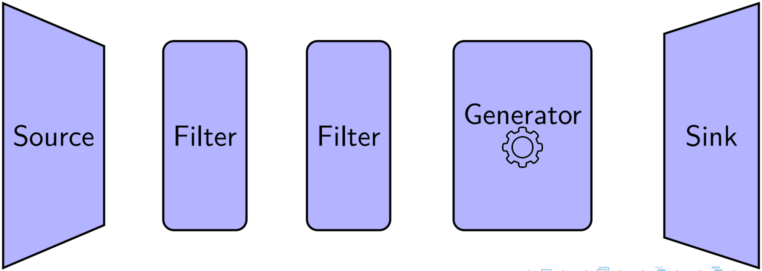
Graphs and MSTs in DPA: Stages of Dynamic Pipeline

- **Source:** This stage manages the in-connection with the outside: streaming input, file reading, random generation...
- **Sink:** This stage manages the out-connection. Usually stdout
- **Generator:** If operation requires to be processed by a filter, it will generate a new filter.



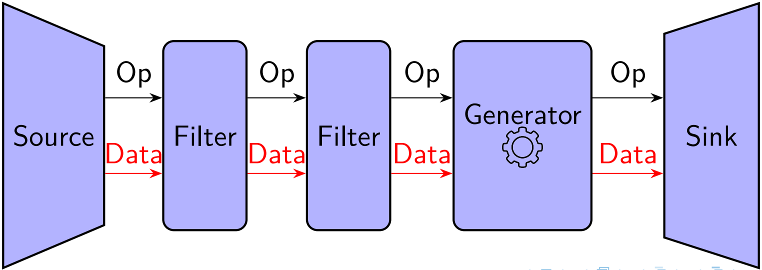
Graphs and MSTs in DPA: Stages of Dynamic Pipeline

- **Source:** This stage manages the in-connection with the outside: streaming input, file reading, random generation...
- **Sink:** This stage manages the out-connection. Usually stdout
- **Generator:** If operation requires to be processed by a filter, it will generate a new filter.
- **Filter:** This (statefull) stage will select which operations to perform and wich ones have to be passed to the next filter. It will do the main work.

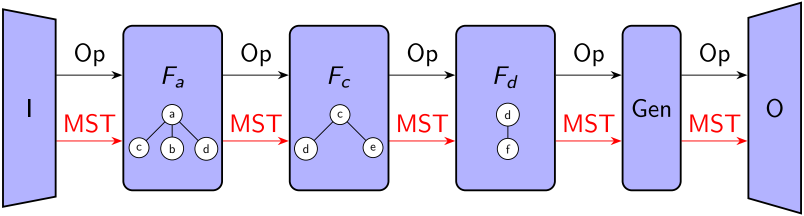


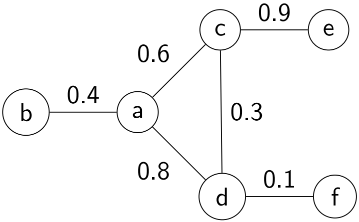
Graphs and MSTs in DPA: Stages of Dynamic Pipeline

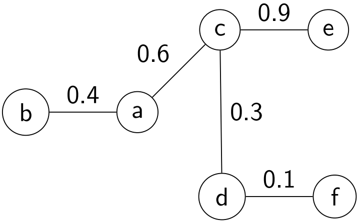
- **Source:** This stage manages the in-connection with the outside: streaming input, file reading, random generation...
- **Sink:** This stage manages the out-connection. Usually stdout
- **Generator:** If operation requires to be processed by a filter, it will generate a new filter.
- **Filter:** This (statefull) stage will select which operations to perform and wich ones have to be passed to the next filter. It will do the main work.



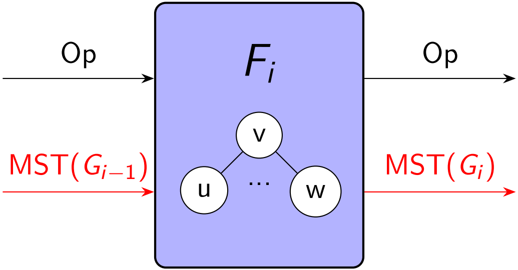
Graphs and MSTs in DPA: `DP_Kruskal`







Graphs and MSTs in DPA: `DP_Kruskal`



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

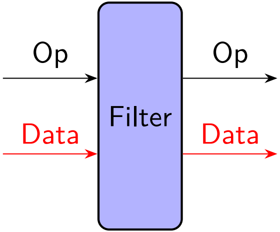
$\mathrm{MST}(T_i \cup \mathrm{MST}(G_{i-1}))$ is computed using `Kruskal`.

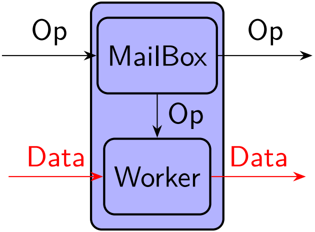
Experimental Study: Optimizations

We proposed different optimizations that could be applied to the `DP_Kruskal`.

- Multiple roots per filter
- Decoupled Event Handling
- Adaptive MST Caching
- Memory management
- Preprocessing

MST is more costly than other operations. This can hold all the incoming operations although they are not related. A possible solution: separate operation reception and data structures modification.





Experimental Study: Decoupled Event Handling (II)

We generated instances with different number of operations and measure the execution time of `DP_Kruskal` of different file sizes with and without the MailBox/Worker optimization.

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

Experimental Study: Comparison

- Experimental Study
  - Static Case
    - Random Graphs
  - Dynamic Case
    - Real Graphs

We measured elapsed wall-clock time $T(k, n, p)$ and we refer to absolute speed up and efficiency as:

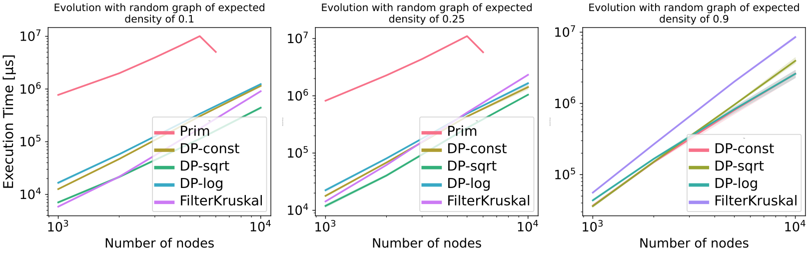
- *Absolute speedup*: $speedup_a(k) = T(1, n, p)/T(k, n, p)$.
- *Efficiency*: $speedup_a(k)/k$.

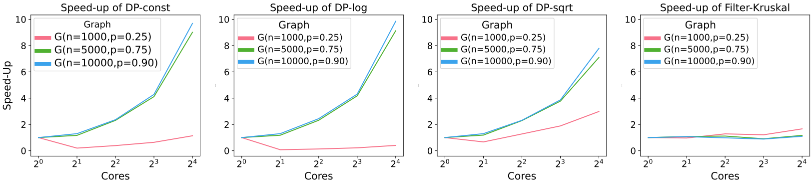
Experimental Study: Static Case (I)

20 static binomial random graphs for each combination of:

- Number of vertices: $n \in \{10^3, 2 \cdot 10^3, 10^4, 5 \cdot 10^4\}$
- Edge probability: $p \in \{0.25, 0.50, 0.75, 0.90\}$

Experimental Study: Static Case (II)





Experimental Study: Dynamic case (I)

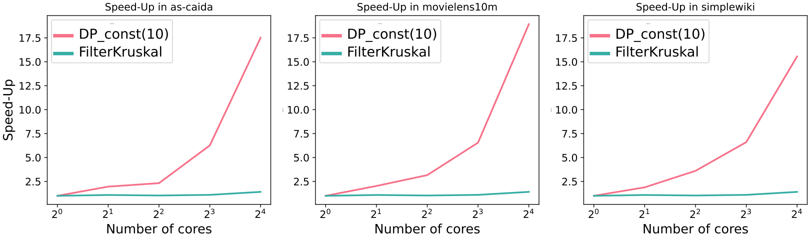
We obtained some realistic dynamic graphs collected from a public repository. [1].

Dataset	Vertices	Operations	Density
amazon-ratings	495,452	476,728	$1.94 \cdot 10^{-6}$
as-caida	31,379	19,468	$1.21 \cdot 10^{-4}$
frwiki	2,212,682	31,624,375	$6.46 \cdot 10^{-6}$
movielens10m	49,847	384,585	$1.55 \cdot 10^{-4}$
simplewiki	100,312	889,016	$8.84 \cdot 10^{-5}$

[1] `https://DynGraphLab.github.io/`

Experimental Study: Dynamic case (II)

Dataset	Filter_Kruskal	DP_Kruskal	SpeedUp
as-caida	1h 30min	1h 19min	1.14
movielens10m	1h 39min	1h 20min	1.24
simplewiki	17h 8min	11h 29min	1.48



Conclusion & Future Work

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:

- Extend the implementation to use several machines
- Broader experiments with algorithms for maintaining an MST of dynamic graphs
- Explore other data structures inside the filter stage
- More precise analysis of cost
- ...

Conclusion & Future Work

A short-paper of this work has been accepted for presentation in the Poster session at the Euro-Par 2024 conference.