Top-k Most Probable Triangles in Uncertain Graphs

Dimitrios Tourgaidis, 66
Panagiotis Papaemmanouil, 56

MSc, Data and Web Science (Spring Semester 2021)

Course: Mining of Massive Datasets

Professors: A. Gounaris, A. Papadopoulos

SCHOOL OF INFORMATICS

Data & Web Science MSc Program

TABLE OF CONTENTS

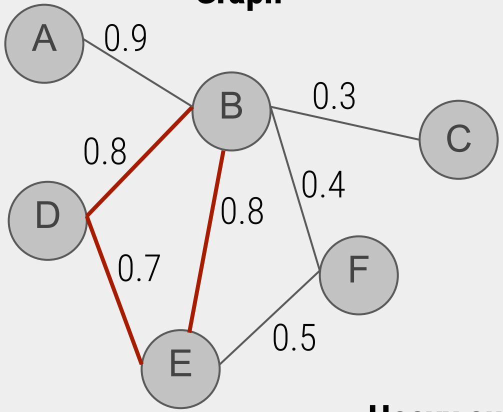
Problem Description

Given a probabilistic graph detect the k most probable triangles.

Definitions

- ➔ Probabilistic Undirected Graph, *G(V,E,w)*
- ➔ Edges’ weights distribution
- ➔ Triangles and Triangle probability (weight)
- ➔ Top-k heaviest triangles
- ➔ Heavy and Light subgraphs

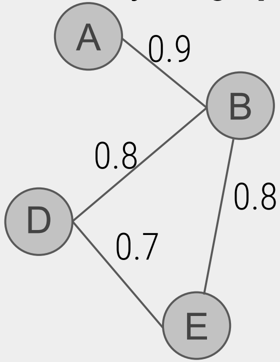
Graph



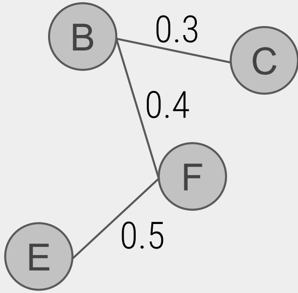
Example:
Triangle “BDE”, **weight**=0.9*0.7*0.8

Threshold = 0.6

Heavy subgraph



Light subgraph



Algorithms (x3)

01 Baseline

02 Optimization 1
Optimizes the baseline

03 Optimization 2
Optimizes the Optimization 1

Implementations (x2)

01 GraphFrames

02 RDD

ALGORITHMS: Baseline

Step 1

Identify all triangles in the graph

Step 2

Calculate the triangles' probabilities.

Step 3

Return the top-k heaviest triangles

Advantages

Easy to understand and implement.

Major Drawback

Doesn't Scale on large Graphs, because it calculates **all** the triangles to find the k heaviest

Theorem to optimize Baseline

- Let G=(V,E,w) be a probabilistic uncertain graph
- Let g=(V',E',w') be a subgraph of G
- Let Topk=(t1,...,tk) be the topk heaviest triangles of g
- Let t_{min} be the triangle with the lowest probability in TopK
- Let e1' and e2' the edges with the highest probability in E'

Let $e_x \in E - E'$ and w_x the corresponding probability

If for w_x is valid that

$$w_x \times w'_{max1} \times w'_{max2} < t_{min}$$

Then for $g' = g\left(V', E \cup e_x, w \cup w_x\right)$ is valid that

$$TopK' = TopK$$

Consequently no need to calculate $TopK'$

Theorem (Optimization 1)

(1)

Let $G_h = \left(V_h, E_h, w_h\right)$ be the heavy subgraph

and $G_l = \left(V_l, E_l, w_l\right)$ be the light subgraph of $G = (V, E, w)$

Let $\mathrm{TopK}_h = \{t_1, t_2, \ldots, t_k\}$ be the topk heavist triangles of G_h

and t_{min} the triangle with the minimum probability in TopK_h

Let e_{max1} and $e_{max2} \in E_h$ be the 2 edges with the highest probabilities w_{max1} and $w_{max2} \in w_h$

Minimum Probability **(2)**

$$x \times w_{max1} \times w_{max2} > t_{min} \Leftrightarrow$$

$$x > \frac{t_{min}}{w_{max1} \times w_{max2}}$$

(3)

If $G_h{}' = G_h \cup G_l{}'$, where $G_l{}' = \left(V_l{}', E_l{}', w_l{}'\right) \subseteq G_l$, $\forall\ edge\ probability \in w_l{}'$ is valid that

$$edge\ probability > x\ (\ minimum\ probability)$$

Then the $TopK_{h'}$ contains the global topk heaviest triangles of G

ALGORITHMS: Optimization 1

Steps

1. Decompose the graph into **Heavy and light subgraphs** based on Threshold T
2. Calculates the **topk heaviest triangles** of Heavy subgraph
3. Calculates **Minimum Probability**
4. Creates the **updated** Heavy subgraph based on the Minimum probability
5. Calculates the **topk heaviest triangles** of the updated Heavy subgraph

Hyperparameter T. The assignment of the Threshold value and its reduce steps in Case 2 has great impact in its efficiency

Case1

Optimization 1 implements only **steps**, if initial Heavy subgraph has at least k triangles

Case2

- If initial Heavy subgraph has less than k triangles
  - Reduce threshold T
  - Executes **Steps** again
  - If new heavy subgraph has less than k tringles, executes **Case2** again until a heavy subgraph has K triangles

Drawback. In Case2 already calculated triangles are calculated again leading to recomputations

ALGORITHMS: Optimization 2

Steps

1. Decompose the graph into **Heavy and light subgraphs** based on Threshold T
2. Calculates the **topk heaviest triangles** of Heavy subgraph
3. Stores in **Memory** the topk heaviest triangles calculated at step 2
4. Calculates **Minimum Probability**
5. Creates the **updated** Heavy subgraph based on the Minimum probability
6. Calculates the topk heaviest triangles of the updated Heavy subgraph, **except those stored at step 3**
7. Stores in **Memory** the triangles from step 3 and 6

Case1

Optimization 1 implements only **steps**, if initial Heavy subgraph has at least k triangles

Case2

❖ If initial Heavy subgraph has less than k triangles
  - ➢ Reduce threshold T
  - ➢ Executes **Steps** again (at step 2 calculates not calculate edges that were stores in memory at step 7)
  - ➢ If new heavy subgraph has less than k tringles, executes **Case2** again until a heavy subgraph has K triangles

Limitation. The memory capacity must be able to store the triangles

Algorithms Implementations

- Implementation based Spark's **GraphFrames** and **RDD** APIs

- **Focus** on triangles identification and their probability calculation.

* **GraphFrames optimization 2,** can not be implemented exactly as described in the previous section, because the incapability of the ***find()*** function to utilize the stored triangles in memory.

Implementations: Graphframes

Steps

1. Re-order the edges direction in **Edge Dataframe**

2. Create ***GraphFrame()*** object based on Nodes and Edges (Dataframes)

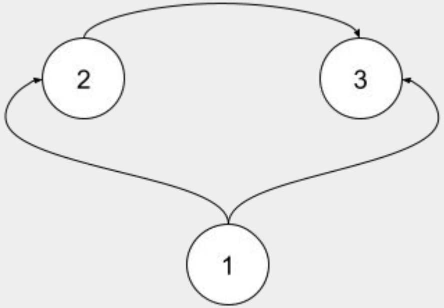
3. Find triangles using the ***find()*** function on GraphFrame() object and the corresponding **query**. This will return a **subgraph Data Frame** with each distinct triangle information in rows.

   ().find("(a)-[e]->(b); (b)-[e2]->(c); (a)-[e3]->(c)")

4. Use ***withColumn()*** function to create a new column that will store the triangle's label.

5. Use ***withColumn()*** function to create a new column that will store the triangle's probability

src	dst	probability
0	1794	0.9916502091617779
0	3102	0.42097990780244077
0	16645	0.02903458011468718
0	23490	0.18777644751934774
0	42128	0.009018990933542304
0	3822	0.9642845396877506
0	9555	0.7110197427615368
0	18602	0.5092125751882496
0	14473	0.48305600322478526
0	25929	0.04222184842363452
0	47676	0.522070150471605
0	9667	0.12149358078264472
0	29643	0.9587198811702391
0	15705	0.4799571579818369
1	43397	0.521340660415808
1	45983	0.5312038417347966
1	38922	0.1170472574186554



Triangles edge structure after edges direction re-ordering.

```
>>> subgraph.show()
```

a	e	b	e2	c	e3
{23}	{23, 7226, 0.3502...	{7226}	{7226, 28753, 0.8...	{28753}	{23, 28753, 0.077...
{35}	{35, 2386, 0.7851...	{2386}	{2386, 39082, 0.5...	{39082}	{35, 39082, 0.779...
{35}	{35, 271, 0.83462...	{271}	{271, 39082, 0.31...	{39082}	{35, 39082, 0.779...
{35}	{35, 41737, 0.150...	{41737}	{41737, 39082, 0....	{39082}	{35, 39082, 0.779...
{53}	{53, 31602, 0.091...	{31602}	{31602, 33767, 0....	{33767}	{53, 33767, 0.052...
{53}	{53, 8690, 0.9544...	{8690}	{8690, 33767, 0.5...	{33767}	{53, 33767, 0.052...
{53}	{53, 17045, 0.559...	{17045}	{17045, 33767, 0....	{33767}	{53, 33767, 0.052...
{53}	{53, 27821, 0.396...	{27821}	{27821, 33767, 0....	{33767}	{53, 33767, 0.052...
{53}	{53, 31686, 0.532...	{31686}	{31686, 33767, 0....	{33767}	{53, 33767, 0.052...
{53}	{53, 1799, 0.1711...	{1799}	{1799, 33767, 0.0...	{33767}	{53, 33767, 0.052...
{53}	{53, 26643, 0.790...	{26643}	{26643, 33767, 0....	{33767}	{53, 33767, 0.052...
{78}	{78, 12271, 0.170...	{12271}	{12271, 13506, 0....	{13506}	{78, 13506, 0.346...
{78}	{78, 4148, 0.7451...	{4148}	{4148, 13506, 0.1...	{13506}	{78, 13506, 0.346...
{93}	{93, 30598, 0.100...	{30598}	{30598, 11204, 0....	{11204}	{93, 11204, 0.645...
{93}	{93, 17045, 0.187...	{17045}	{17045, 11204, 0....	{11204}	{93, 11204, 0.645...
{93}	{93, 14014, 0.976...	{14014}	{14014, 11204, 0....	{11204}	{93, 11204, 0.645...
{93}	{93, 7397, 0.5229...	{7397}	{7397, 11204, 0.4...	{11204}	{93, 11204, 0.645...
{93}	{93, 10379, 0.974...	{10379}	{10379, 11204, 0....	{11204}	{93, 11204, 0.645...
{93}	{93, 27503, 0.003...	{27503}	{27503, 28926, 0....	{28926}	{93, 28926, 0.668...
{93}	{93, 16355, 0.375...	{16355}	{16355, 28926, 0....	{28926}	{93, 28926, 0.668...

Implementations: RDD (I)

(1)

1, 2, 0.80
5, 1, 0.56
1, 8, 0.44
2, 5, 0.40
6, 2, 0.79
2, 8, 0.30

Map()

(2)

(1, (2, 0.80))
(2, (6, 0.79))
(1, (5, 0.56))
(1, (8, 0.44))
(2, (5, 0.40))
(2, (8, 0.30))

groupByKey()

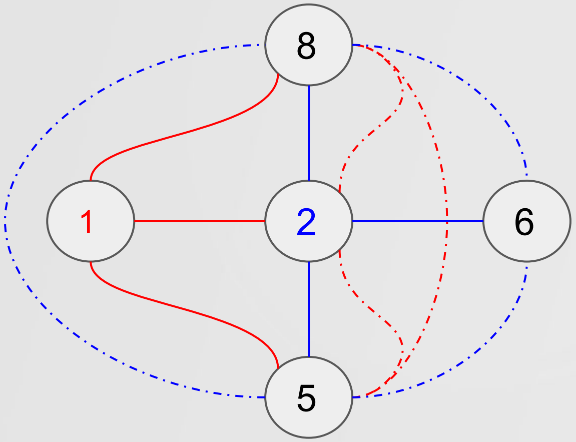
(3)

(1, [(2, 0.80),(5, 0.56),(8, 0.44)])
(2, [(5, 0.40),(6, 0.79),(8, 0.30)])

flatMap()

(4)

((1,2), (0.80,-1))
((1,5), (0.56,-1))
((1,8), (0.44,-1))
((2,5), (0.40,-1))
((2,6), (0.79,-1))
((2,8), (0.33, -1))
((2,5), (X, 1))
((2,8), (X, 1))
((5,8), (X, 1))
((5,6), (X, 2))
((5,8), (X, 2))
((6,8), (X, 2))



(4)

Candidate Edge

Implementations: RDD (II)

(4)

((1,2), (0.80,-1))
((1,5), (0.56,-1))
((1,40), (0.44,-1))
((2,5), (0.40,-1))
((2,6), (0.79,-1))
((2,40), (0.33, -1))
((2,5), (X, 1))
((2,40), (X, 1))
((5,40), (X, 1))
((5,6), (X, 2))
((5,40), (X, 2))
((6,40), (X, 2))

groupByKey()

(5)

((1,2), [(0.80,-1)])
((1,5), [(0.56,-1)])
((1,40), [(0.30,-1)])
((2,5), [(X,1), (0.40,-1)])
((2,40), [(X,1), (0.30,-1)])
((2,6), [(0.79,-1)])
((5,40), [(X,1), (X,2)])
((6,40), [(X,2)])
((5,6), [(X,2)])

flatMap()

(6)

(1,((1,2),0.80))
(1,((1,5),0.56))
(1,((1,40),0.44))
(2,((2,5),0.40))
(1,((2,5),0.40))
(2,((2,40),0.30))
(1,((2,40),0.30))
(2,((2,6),0.79))

groupByKey()

(7)

(1, [((1,2),0.80), ((1,5),0.56), ((1,40),0.30), ((2,5),0.40), ((2,40),0.30)])
(2, [((2,5),0.40), ((2,40),0.30), ((2,6),0.79)])

flatMap()

(8)

((1,2,5), 0.1792)
((1,2,40), 0.1056)

Experiments

Artists Dataset

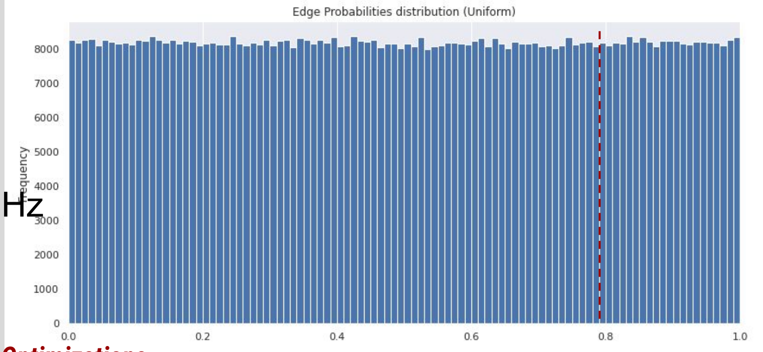
- ➔ 50.515 nodes
- ➔ 819.306 edges
- ➔ 2.273.700 triangles

Hardware setup

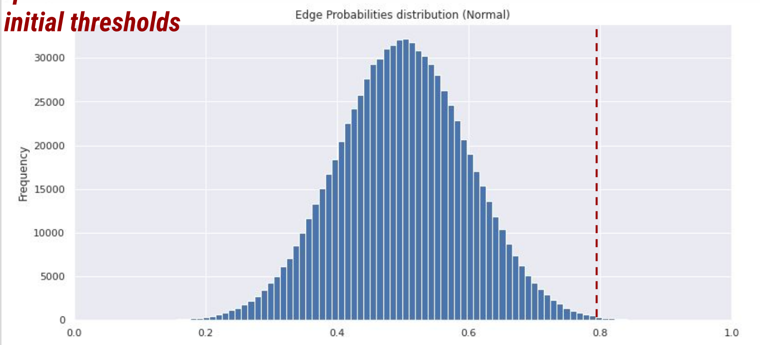
- ❑ Ubuntu 18.04.5 LTS
- ❑ Intel® Core™ i7-8550U CPU @1.80GHz
- ❑ 8 cores
- ❑ 20Gb RAM

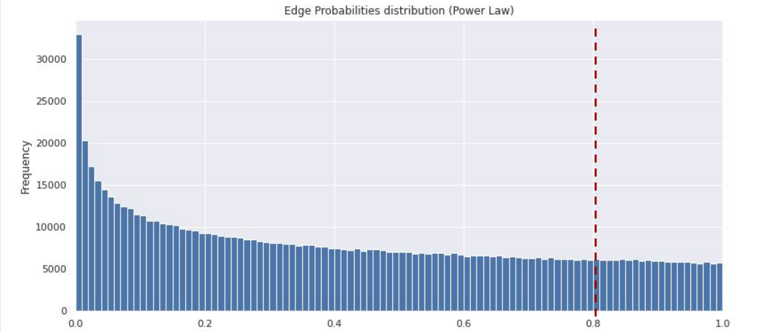
Experimental setting 6 x 3 x 3 x 3 = 162 experiments	
Algorithms	1. Baseline GraphFrames 2. Optimization 1 GraphFrames 3. Optimization 2 GraphFrames 4. Baseline RDD 5. Optimization 1 RDD 6. Optimization 2 RDD
Edges weight Distribution	● Uniform ● Normal ● Power Law
Cores	1, 2, 8
Total triangles to return (k)	10, 100, 10.000

https://snap.stanford.edu/data/gemsec-Facebook.html

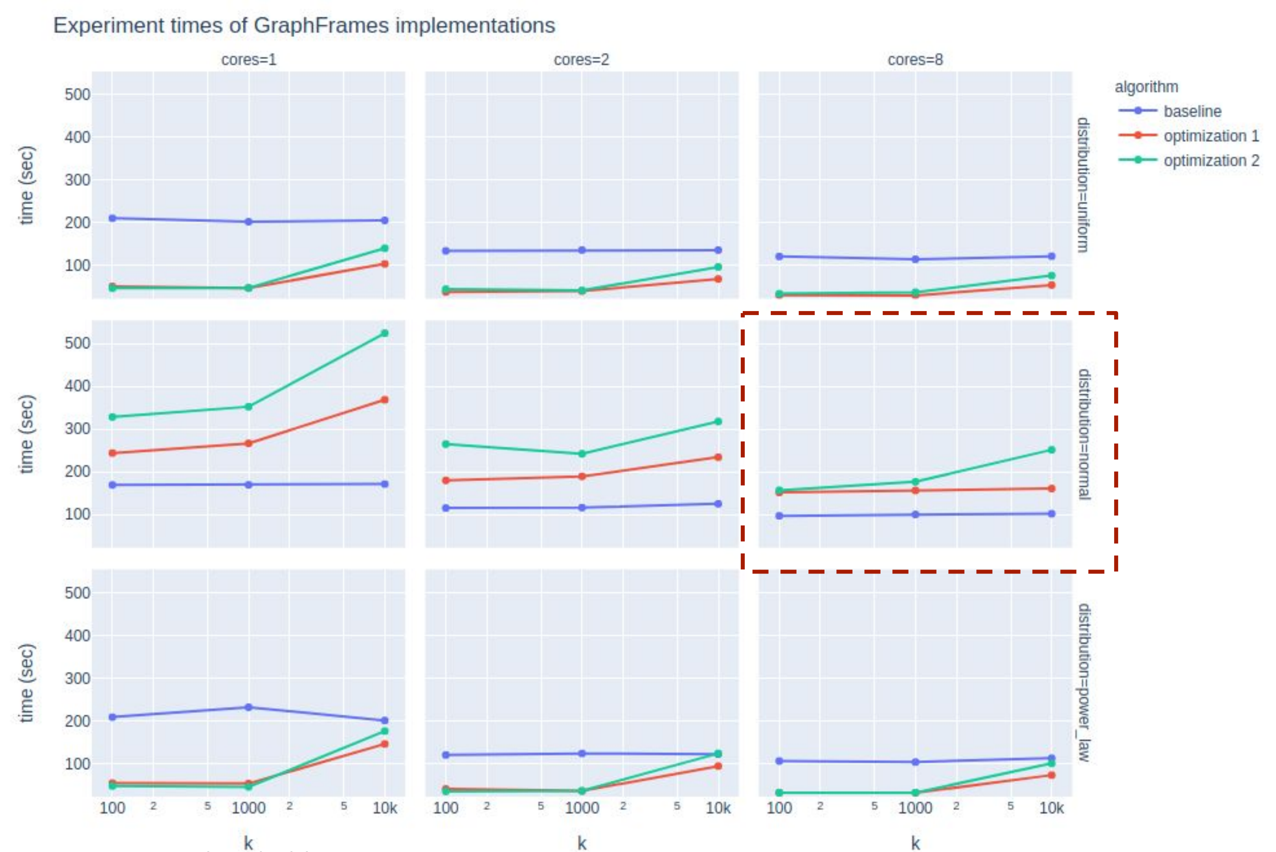


Optimizations initial thresholds





Results: GraphFrames

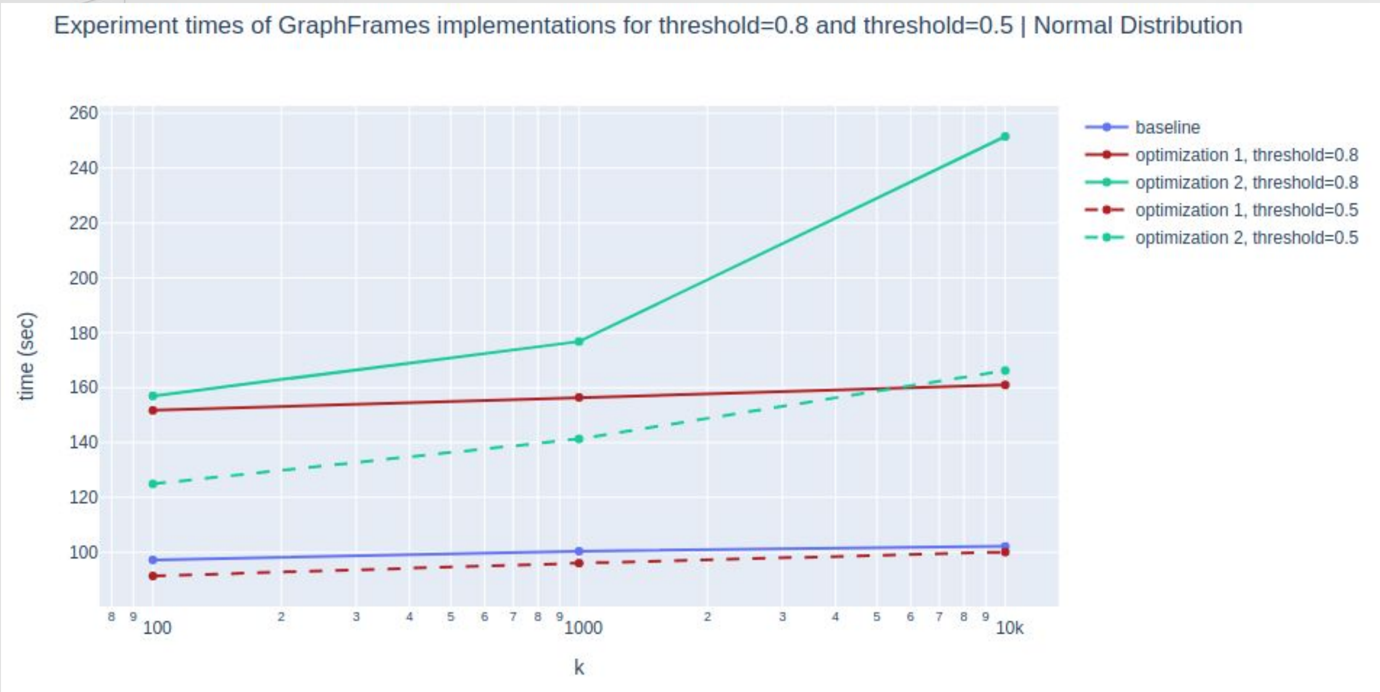


Optimizations threshold = 0.8

- Algorithms ***scale*** well based on number of cores.
- The ***2 optimizations*** implementations are faster than the ***baseline*** implementation, except the ***Normal distribution*** case, because the optimizations algorithms can not find the top-k heaviest triangles with the first ***decomposition*** (heavy and light subgraphs), leading to re-computations.
- ***Optimization 2*** is slower compared to ***Optimization 1***, because of GraphFrames drawback in optimization 2 implementation.

GraphFrames: Different initial Threshold

(Normal distribution)



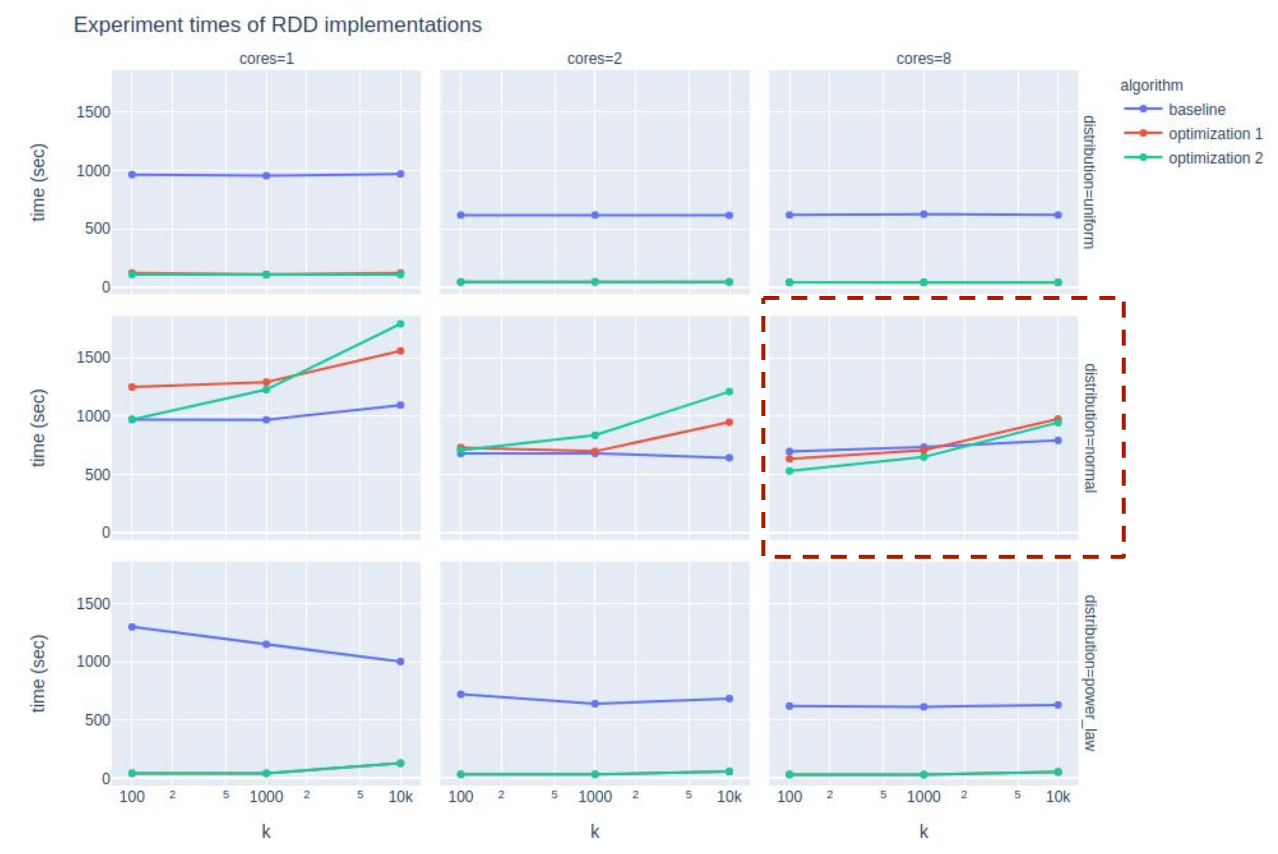
Giving as input a ***lower initial threshold***, the corresponding ***optimizations*** cases outperform the ***baseline-case***.

Conclusion

Selecting the appropriate initial threshold, based on edges' weight distribution, has a great impact in implementations' performance.

- The same applies for the RDD implementations.

Results: RDD

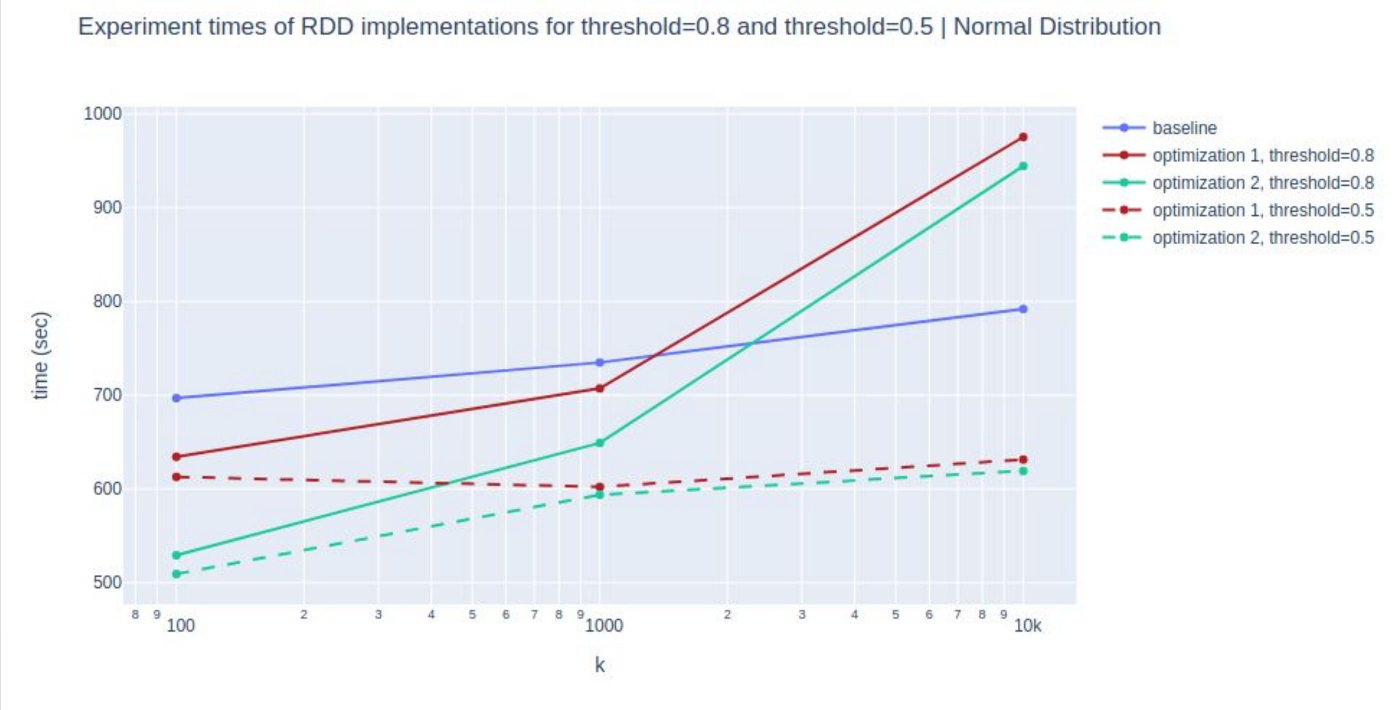


Optimizations threshold = 0.8

- ***Baseline*** execution times do not affected by the value of k.
- The ***2 optimizations*** implementations are faster than the ***baseline*** implementation, except the ***Normal distribution*** case, because the optimizations algorithms can not find the top-k heaviest triangles with the first ***decomposition*** (heavy and light subgraphs), leading to re-computations.
- ***Optimization 1*** and ***Optimization 2*** have equal execution times,for Uniform and Power Law distributions. In general, there **isn't a clear winner** between optimization 1 and 2.

RDD: Different initial Threshold

(Normal distribution)



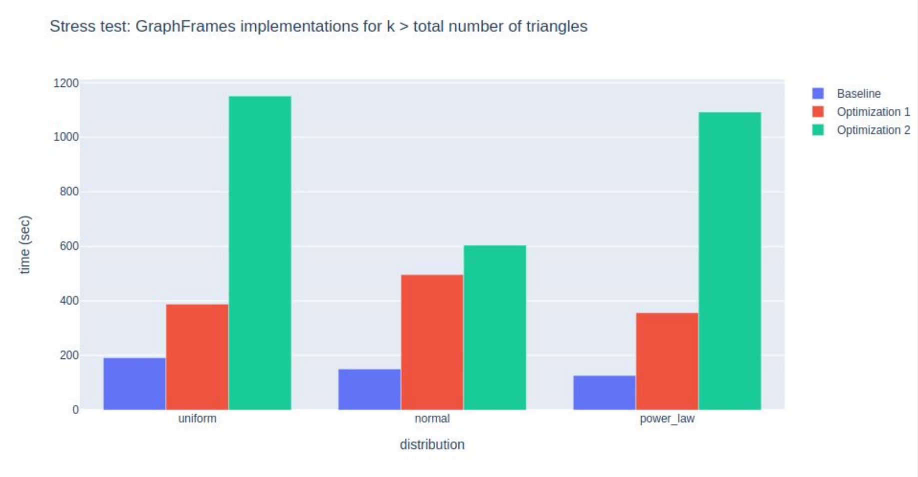
Giving as input a ***lower initial threshold***, the corresponding ***optimizations*** cases outperform the ***baseline-case***.

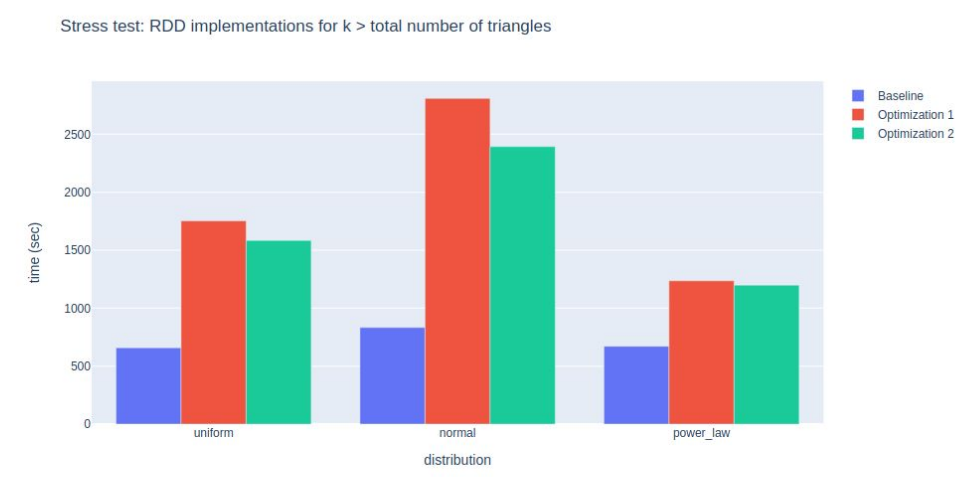
Conclusion

Selecting the appropriate initial threshold, based on edges' weight distribution, has a great impact in implementations' performance.

Stress Test: GraphFrames & RDD

k>Total triangles





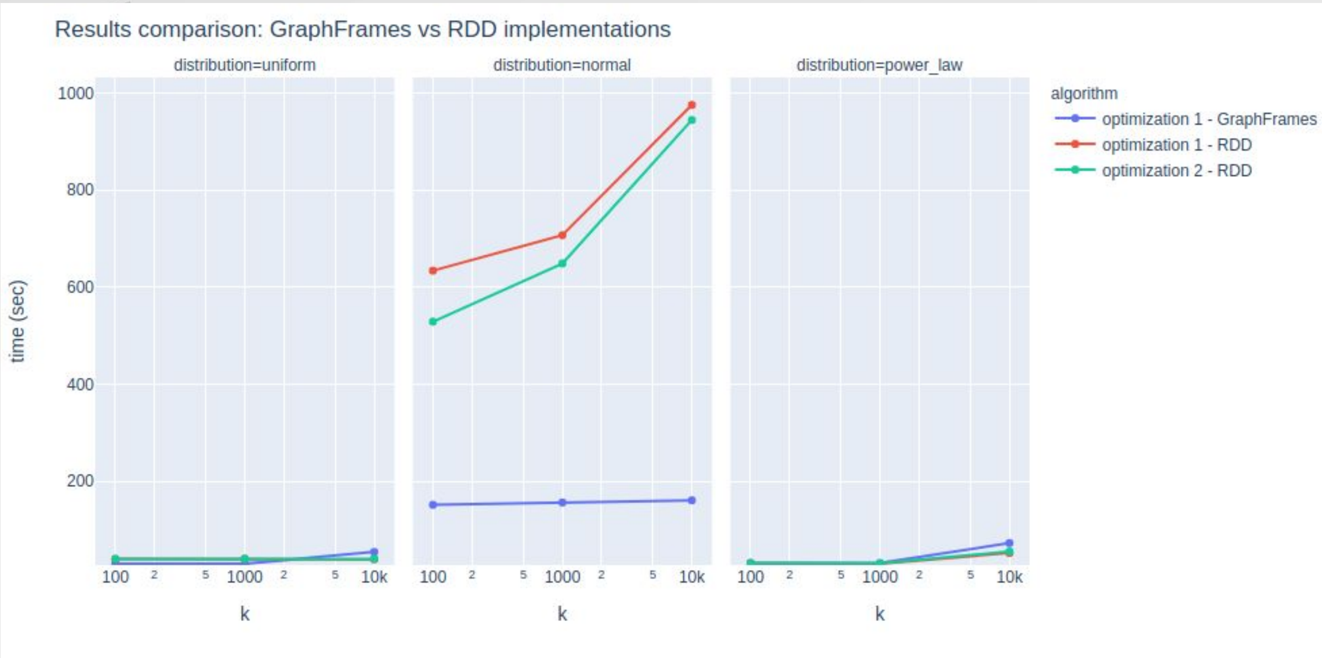
CONCLUSION

Baseline is always faster than **Optimizations**, for both GraphFrames and RDD implementations.

WHY?

Because the whole graph will be surely examined. The ***baseline*** iterates the graph once, while the ***optimized implementations*** will make re-computations for each updated (reduced) threshold, until finally the threshold will equal to 0 (examine full graph).

Comparison:
GraphFrames and RDD implementations



- GraphFrames runs faster than RDD implementation, in Normal distribution.
- RDD implementations are slightly better in Uniform and Power-Law distributions.

CONCLUSIONS

- ➔ The decision which implementation to use is probability-distribution dependent.
- ➔ The threshold value has great impact on the execution times of the optimizations implementations.
- ➔ Baseline implementation runs always faster than the optimizations implementations for ***k > all triangles exist in the graph.***

"In order to understand the true behavior of the optimizations implementations, they must be examined on a true massive probabilistic uncertain graph!"

For example, the baseline implementation will not be faster than optimizations implementation in normal distribution

THANK YOU!

Any questions?