# Top-k Most Probable Triangles in Uncertain Graphs

Mining Of Massive Datasets

Dimitrios Tourgaidis*
tourgaidi@csd.auth.gr
Aristotle University of Thessaloniki
Greece

Panagiotis Papaemanouil*
ppapaemm@csd.auth.gr
Aristotle University of Thessaloniki
Greece

**ACM Reference Format:**
Dimitrios Tourgaidis and Panagiotis Papaemanouil. 2021. Top-k Most Probable Triangles in Uncertain Graphs: Mining Of Massive Datasets. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Triangle discovery is an important task in Graph Mining, which is strongly related to more complex tasks such as dense subgraph discovery, community detection, link prediction and many more. The problem of triangle listing or triangle counting has been studied well in the literature. However, in many important applications, graphs are characterized by uncertainty, which introduces additional difficulties in the mining process. The edges of these graphs are assigned with a probability value, where, the higher the probability value the more probable is that the edge is present in the graph.

In this project, we focus on the discovery of the k most probable triangles in massive uncertain graphs. We implement 3 algorithms, where the first is a baseline approach, the second an optimization of the baseline case, and the third an even further optimization of the second algorithm.

In order to implement the 3 algorithms, we used Apache Spark. Once, we implement the algorithms using Spark's GraphFrames API, and once, using Spark's RDD API. Our objective is to make a comparison between the 2 implementations to see which is more suitable, solely more efficient, for our case. For both cases, we utilized Spark's PySpark, so our codes implementations are completely in python.

With aim to evaluate the efficiency of the 2 optimizations algorithms (compared to the baseline algorithm) we present in the report, we conducted an exhaustive experimentation process for a number of different scenarios; different number of: k triangles with highest probability to return, edges weight (probability) distributions, cores that Spark runs. This process leads to understand the general behavior of our algorithmic approaches. as well to identify which implementation is the most efficient.

**Report outline.** Next, in order to comprehend the rest of the report's structure, we present the outline of the remaining sections.

In **Section 2**, we present preliminary definitions of the field, along with 2 Theorems, which are used for the development of the 2 optimizations algorithms. In **Section 3** we provide a thorough description of the 3 algorithms we examine, namely the baseline and its 2 optimizations-cases. Additionally, for each algorithm, we provide its implementation in pseudo-code. In **Section 4**, we describe the development of the algorithms introduced in **Section 3**, both using Spark's GraphFrames API and RDD API. For each algorithm, we focus mainly on the triangle calculations process because there is where their core difference lies. Finally, in **Section 5** we present the results of the experimentation-phase, based on a real-world graph data set. We evaluate all implemented algorithms in terms of time efficiency and scalability. Furthermore, we provide a relative comparison accompanying with a rule of thumb for selecting the best algorithm for different situations.

Finally, all the source code, along with experimental results can be found on GitHub here[1].

## 2 PRELIMINARIES

This section presents important definitions of the field's concepts that the report regards; *top-k most probable triangles in uncertain graphs.* Furthermore, it presents 2 theorems we used for our optimization algorithms. These concepts are heavily used in the rest of the report, so their comprehension is mandatory.

**Definition 2.1 (Probabilistic Undirected Graph).** A graph $G = (V, E, w)$ is a probabilistic undirected graph if $\forall e \in E$ has no direction and $\forall e \in E$ is assigned with a corresponding weight (probability) $w_e \in w$. The probability of an edge shows the probability that the edge exist in the graph.

**Definition 2.2 (Triangle probability).** Let $G = (V, E, w)$ be a probabilistic undirected graph, and $t$ a triangle of $G$ that is constituted from 3 edges $e_1, e_2, e_3 \in E$. The probability of $t$

*All authors contributed equally to this research.

*Conference'17, July 2017, Washington, DC, USA*
© 2021 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY-NC-ND 4.0 License.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]GitHub Mining-of-Massive-Datasets-AUTh

is P(t) = $w_1 \times w_2 \times w_3$, where $w_1, w_2, w_3 \in w$ the corresponding probabilities of $e_1, e_2, e_3$

**Definition 2.3 (Top-k heaviest triangles).** Let $G = (V, E, w)$ be a probabilistic undirected graph, and $TopK = \{t_1, t_2, \ldots, t_k\}$ a set that contains triangles of $G$. $TopK$ is the set that contains the top-k heaviest (highest probabilities) triangles of $G$ if $\forall t \in TopK$ there is no other triangle $t' \in G$ and $t' \notin TopK$ that $P(t) < P(t')$. Solely, $TopK$ contains the triangles of $G$ with the highest probability.

**Definition 2.4 (Heavy and Light subgraphs).** Let $G = (V, E, w)$ be a probabilistic undirected graph. We define as heavy subgraph $G_h = (V_h, E_h, w_h)$ and light subgraph $G_l = (V_l, E_l, w_l)$, the 2 subgraphs of $G$ where the $G_h$ contains all edges from $G$ with probability more or equal than a threshold value, while the $G_l$ contains all the edges from $G$ with probability less than the threshold. Based on this definition, it is clear that $E_h$ and $E_l$ are mutually exclusive sets, while this is not applied for $V_h$ and $V_l$.

**Theorem 2.5.** *Let $G = (V, E, w)$ be a probabilistic undirected graph. Then, $TopK = \{t_1, t_2, \ldots, t_k\}$ be a set that contains the top-k heaviest triangles of a subgraph $G' = (V', E', w') \subset G$, where $t_{min} \in TopK$ the least heavy triangle in $TopK$ and $e'_{max1}, e'_{max2} \in E'$ the 2 edges with the 2 highest probabilities $w'_{max1}, w'_{max2} \in w'$. In case where for any edge $e_x \in E - E'$ is valid that $(w_x \times w'_{max1} \times w'_{max2}) < t_{min}$ (1), then no triangle can exist that is constituted from $e_x$ and 2 edges $\in E'$ that is heavier from $t_{min}$. Consequently, if $e_x$ is added to $E'$ and then the top-k heaviest triangles ($TopK'$) are again calculated for the updated $G'$, then $TopK = TopK'$.*

*On the other hand if (1) is not valid, it doesn't mean that for sure a corresponding triangle exist that is heavier compared to $t_{min}$, instead there is a possibility that such a triangle may exist. Consequently, $TopK = TopK'$ can be valid also in this case, but there is no guarantee for this until we recalculate again the top-k heaviest triangles in the updated subgraph, and compare them to the previous top-k heaviest triangles.*

**Theorem 2.6.** *Let $G = (V, E, w)$ be a probabilistic undirected graph and $G_h = (V_h, E_h, w_h)$ and $G_l = (V_l, E_l, w_l)$ the corresponding heavy and light subgraphs (2.4) of G based on a threshold T, where $0 < T < 1$. Next, $TopK_h = \{t_{h1}, t_{h2}, \ldots, t_{hk}\}$ is a set that contains the top-k heaviest triangles of the heavy subgraph. Based on theorem 2.5, in case where, $e_{l_{max}} \in G_l$ is the edge with the highest probability among all the the edges that $\in G_l$, then if its corresponding probability $w_{l_{max}} \in w_l$ validates (1) of theorem 2.5, then all the edges $\in G_l$ are validating (1), so $TopK_h$ is the global top-k set.*

*On the other hand, if $G'_l = (V'_l, E'_l, w'_l) \subset G_l$ and $\forall$ edge $\in G'_l$ the (1) is not valid, then the global top-k heaviest triangles is going to be the top-k heaviest triangles ($Topk_{global}$) of $G_h \cup G_{l'}$. As mentioned in theorem 2.5, $Topk_{global}$ can be equal to $TopK_h$, but we can't be sure about it in advance.*

## 3 Algorithms

At first, this section presents the algorithmic approaches we examined to solve the assignment, and then the implementations of the algorithms.

As for the algorithms, initially we implemented a *baseline* approach, which served as a base on which we examined various *optimizations*, concluding in 2 optimized approaches.

**Baseline algorithm**. The logic behind the *baseline* algorithm is very simple. Given a weighted graph G = (V,E,w), we calculate all the triangles of G, then we take the $K$ heaviest triangles that ultimately will be the top-k heaviest triangles of the graph. Algorithm 1 presents the *baseline* algorithm.

**Algorithm 1:** Baseline algorithm to find top-k heaviest triangles in a weighted graph

```
Input: Weighted graph $G = (V, E, w)$, top-k heaviest
       triangles of $G$ to return $K$
Output: The top-k heaviest triangles of $G$
1 $T \longleftarrow$ All the triangles of $G$ and their probabilities
2 $Topk \longleftarrow$ the top $K$ heaviest triangles of $T$
3 return $TopK$
```

**Optimization 1**. The drawback of the *baseline* approach is that it calculates all the triangles to return the top-k heaviest triangles of the graph. In order to avoid this drawback, we utilized the theorems 2.5, 2.6.

The optimized algorithm, at first, decompose the graph into a *heavy* and a *light subgraph* based on a threshold, and then calculates the top-k heaviest triangles in the *heavy subgraph*. Next, it checks if edges that belong to the *light subgraph* exist that validate (1) in theorem 2.5. If such edges exist, they are added to the *heavy subgraph* constructing a new *heavy subgraph*, for which the top-k heaviest triangles are recalculated again **from scratch**. The result of the aforementioned recalculation is the global top-k heaviest triangles of the graph. In case were such edges not exist, the initial top-k heavies triangles, of the *heavy subgraph*, are also the global top-k heaviest triangles.

In case where the *heavy subgraph* has in total less than k triangles, the value of the threshold is decreased and the graph is again decomposed into the *heavy* and *light subgraphs*, now, based on the decreased threshold. Afterwards, the procedure regarding the calculation of global top-k heaviest triangles is again executed for the new decomposition.

There is also the possibility that even the *heavy subgraph* of the new decomposition has less than k triangles in total. In such a case, the procedure of decreasing the threshold and trying to recalculate the global top-k heaviest triangles is executed until the *heavy subgraph* of a new decomposition will have k or more triangles or the threshold becomes equal to 0. If threshold becomes equal to 0, then the corresponding *heavy subgraph* will practical be the entire graph, consequently its top-k heaviest triangles are going to be the

**Algorithm 2:** Optimization 1 algorithm to find top-k heaviest triangles in a weighted graph

```
Input: Weighted graph $G = (V, E, w)$, Threshold $T$ ($0 < T < 1$), top-k heaviest triangles of $G$ to return $K$
Output: The top-k heaviest triangles of $G$
1  repeat
2      $HeavySG \leftarrow$ the heavy subgraph of $G$ based on $T$
3      $Topk_h \leftarrow$ the top-k heaviest triangles of $HeavySG$
4      if $T = 0$ then
5          return $Topk_h$
6      else
7          if number of triangles in $Topk_h$ is equal to $K$ then
8              break the repeat loop
9          else
10             $T \leftarrow$ decreased threshold, $0 \leq T <$ previous $T$
11         end
12     end
13 until $T = 0$
14 $e_{max1} \leftarrow$ the edge with the highest probability in $G$
15 $e_{max2} \leftarrow$ the edge with the second highest probability in $G$
16 $T_{min} \leftarrow$ the least heaviest triangle in $Topk_h$
17 $LightSG \leftarrow$ the light subgraph of $G$ based on $T$
18 $LightSG' \leftarrow$ the edges that validate $(w_i \times w_{max1} \times w_{max2}) > T_{min}$, $\forall e_i \in LightSG, i = 1, \dots n, n =$ number of edges in $LightSG$
19 $HeavySG' \leftarrow HeavySG \cup LightSG'$
20 $Topk'_h \leftarrow$ the top-k heaviest triangles of $HeavySG'$
21 return $TopK_h$'
```

global top-k heaviest triangles of the graph. For that case, if the total top-k heaviest triangles of *heavy subgraph* are less than k, then the entire graph has less triangles than k, so the existing triangles are just returned. Algorithm 2 presents the previous discussed optimization algorithm.

**Optimization 2**. The *optimization 1* approach can be optimized even further. In *optimization 1*, when the *heavy subgraph* of a decomposition has at least k triangles, namely it has a set of top-k heaviest triangles, edges of the *light subgraph* that validate (1) from theorem 2.5 are added to the updated *heavy subgraph* and the top-k heaviest triangles of the updated *heavy subgraph* are recalculated **from scratch** again. This event leads to recomputing triangles that were calculated previously. The same drawback occurs also when the *heavy subgraph* of a decomposition has less than k triangles, consequently, a new decomposition takes place leading to the recomputation of triangles that were calculated previously.

**Algorithm 3:** Optimization 2 algorithm to find top-k heaviest triangles in a weighted graph

```
Input: Weighted graph $G = (V, E, w)$, Threshold $T$ ($0 < T < 1$), top-k heaviest triangles of $G$ to return $K$
Output: The top-k heaviest triangles of $G$
1  $Topk_{mem} \leftarrow \emptyset$
2  repeat
3      $HeavySG \leftarrow$ the heavy subgraph of $G$ based on $T$
4      $Topk_h \leftarrow$ the top-k heaviest triangles of $HeavySG$
5      $Topk_{mem} \leftarrow Topk_h$
6      if $T = 0$ then
7          return $Topk_h$
8      else
9          if number of triangles in $Topk_h$ is equal to $K$ then
10             break the repeat loop
11         else
12             $T \leftarrow$ decreased threshold, $0 \leq T <$ previous $T$
13         end
14     end
15 until $T = 0$
16 $e_{max1} \leftarrow$ the edge with the highest probability in $G$
17 $e_{max2} \leftarrow$ the edge with the second highest probability in $G$
18 $T_{min} \leftarrow$ the least heaviest triangle in $Topk_h$
19 $LightSG \leftarrow$ the light subgraph of $G$ based on $T$
20 $LightSG' \leftarrow$ the edges that validate $(w_i \times w_{max1} \times w_{max2}) > T_{min}$, $\forall e_i \in LightSG, i = 1, \dots n, n =$ number of edges in $LightSG$
21 $HeavySG' \leftarrow HeavySG \cup LightSG'$
22 $Topk'_h \leftarrow$ the top-k heaviest triangles of $HeavySG'$
23 return $TopK_h$'
```

These 2 drawback-cases can be avoided. For the first case, the top-k heaviest triangles of the *heavy subgraph* are stored in the memory, and used to avoid to recalculate them again. Only the triangles that include the edges added from the *light subgraph* are calculated. As for the second case the same logic is applied. The triangles of the *heavy subgraph* for each decomposition are stored in the memory and used in the next decomposition (if there will be) to avoid the recomputation of already calculated triangles.

An obvious limitation of the this approach is the usage of memory required to store the triangles. The memory capacity of our environment must exceed the required memory needed to store the triangles. Algorithm 3 presents this optimization approach.

## 4 Algorithms Implementations

This section analyzes our implementations of the 3 algorithms introduced at section 3. As mentioned at the report's introduction, we implement all the 3 algorithms with both, Spark's Graphframes and Spark's RDD APIs.

Regarding the implementations analyzes, for both cases, we present only the part that concerns the triangles calculation in a graph. The substantial differentiation of the 2 APIs comes down to this part only. The remaining stages of the algorithms 1,2,3 are software engineering and independent of APIs or programming languages, so there is no need to be presented.

On both implementations, the graph is given in edge-list format (csv), which means that each line represents an edge of the graph. Each line contains three numbers, where the first 2 regard the edge's source and destination node, and the third one the corresponding probability of the edge.

### 4.1 Graphframes Implementations

GraphFrames[2] is an API for DataFrame-based graphs in Spark. Users can write highly expressive queries by leveraging the DataFrame API combined with easy to use functions for motif finding.

GraphFrames API handles a network through a GraphFrame() object, which takes as input the nodes and the edge of the network, where both are given in a Spark' Dataframe API data-structure. A key feature of GraphFrames is that it handles only directed graphs. GraphFrames provides the "find" function which takes as input an edges pattern, and returns the corresponding network's subgraph based on the pattern. Consequently, we utilize the "find" function to query the network"s subgraph that contains all the triangles of the network.

At fist we re-order the edges direction in our edge Dataframe that we give as input to the GraphFrame() object. For each edge, the source node becomes the one with the lower id, and the target node the one with the higher id. This re-ordering results that each network's triangles will have now the structure presented at Figure 1.

Now that each triangle has the structure presented at Figure 1, we apply the following find function at the GraphFrame() object (the network), which returns the network's subgraph that contains all the network's triangles.

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

The resulting subgraph is returned as a Dataframe. Each row of the returned Dataframe contains the data of a distinct triangle, but in a way where the probability has not been calculated yet. Afterwards, we apply a withColumn() function to create a new column in the Dataframe that will

[2]GraphFrames docs

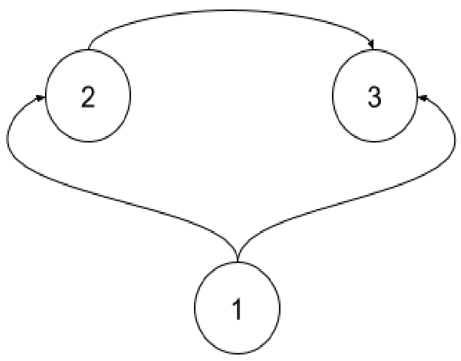

**Figure 1.** Each network's triangle edge directions after edge directions re-ordering

store the triangles label. This step can be avoided if we are not interested to return the triangles in a more human readable form. Lastly we execute again a withColumn() function that creates a new column that stores the probability of each triangle.

**Baseline implementation**. In order to implement the *baseline* algorithm, we execute the presented GraphFrames-based triangle calculation procedure at line 1 of algorithm 1.

**Optimization1 implementation**. In order to implement the *optimization1* algorithm, we execute the presented triangle calculation implementation (GraphFrames-based) at line 3 and 20 of algorithm 2. For both cases, the GraphFrames-based implementation finds all the triangles of the subgraph on which it is executed, and then the $K$ heaviest triangles are returned.

**Optimization2 implementation**. Unfortunately, GraphFrames API can not be utilized to implement the algorithm 3 exactly as described. The $TopK_{mem}$ in 3 is implemented through Apache Spark's *broadcast variable*. The way that the "find" function is implemented, we can not utilize the $TopK_{mem}$ to avoid to recalculate triangles that are stored in the $TopK_{mem}$. What we do is to avoid to recalculate the probabilities of those triangles, which we do at the second withColumn() function. This fact doesn't lead to the optimization of the algorithm 2 which is the objective of algorithm 3. The why is presented in more detailsl at section 5

### 4.2 RDD Implementations

Compared to the graphframes implementations, in the RDD case, we had to implement the top-k heaviest triangles calculation procedure on our own. To accomplish this task, we utilized the *MapReduce* technique.

In order to understand our implementation regarding the top-k heaviest triangles calculation procedure, we present the implementation through a simple example. Suggest we have the following graph, which csv-edge-list form is presented below.

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

All the algorithms (1, 2, 3) implementations execute the same (except the baseline) preprocessing procedure to the loaded dataset, where the result after the preprocessing procedure is presented below.

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

At first, each edge is stored as a key-value pair, with key the node with the smaller id and value, again a key-value pair, with key the id of the bigger node and value the probability of the edge. Furthermore, the edges are sorted in an ascending order based the edge probability. This sorting is not applied to the *baseline* implementation where all the triangles are calculated, so the ordering doesn't matter. It is need for the *optimizations* implementations which construct the *heavy* and *light subgraphs* based on the threshold.

Now that the edges are stored in the form presented previously, a pipeline of transformations is applied to this RDD, where after its execution, the final produced RDD will contain all the triangles of the graph and their probabilities. The transformations of the pipeline are presented in Figure 2.

Getting started with the first transformation of the pipeline, the produced RDD after the execution of the first *groupByKey()* is presented below.

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

Examining the first element of the above RDD, we have inside a key-value pair all the edges and their probabilities, where the one common node (that participates in all those edges) is the node with id=1. Handling the value (list) of this key-value pair we can find, which edges, if they exist, then the triangles exist, whose one node is the node with id=1 and the other 2 nodes are the 2 nodes of the edges we will search to examine if they exist. For example, taking the (2, 0.80) key-value pair we conclude that if the edges (2,5) and (2,40) exist, then the corresponding triangles (1,2,5) and (1,2,40) exist. Next, taking the (5, 0.56) key-value we conclude that if the edge (5,40) exist, then the triangle (1,5,40) exist. For each key-value pair in the list, we utilize only the key-value pairs after them (index-based in the list) to examine which edges if they exist, then the corresponding triangles exist.

Consequently, the first *flatMap()* of the pipeline finds and stores all these "candidate" edges for each node in a RDD, plus all the existing edges and their probabilities. The existing edges and their probabilities are reconstructed by using the key in each RDD element (key-value pair) and the key-value pairs in the value (list) of the corresponding RDD element. Below is presented the result of the first *flatMap()* execution for our example.

((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))

The above RDD contains key-value pairs, where their keys and values are key-value pairs too. In each key-value pair, the key are indicating which edge this key-value pair concerns. Two types of key-value pairs exist, which are going to be analyzed through the example; **i)** type of ((2,5), (0.40,-1)) and **ii)** type of ((2,5), (X, 1)). Type **i)** is interpret as following; the edge (2,5) exist and its probability is 0.40. The "-1" value is a "dummy" constant that works as a flag that shows that this edge exist. For an existing edge in the graph there will be 1 or 0 key-value pair of such type. Next, type **ii)** is interpret as following; if the edge (2,5) exist then the triangle (1,2,5) exist. The "X" value is a "dummy" constant that works as a flag that shows that if this edge exist, then the triangle exist, which nodes are the 2 nodes of the edge that the key-value pair concerns, and the one node which id is stored next to the "dummy" "X" value. For type **ii)** multiple key-value pairs can exist that concern the same edge. For example the key-value pairs ((5,40), (X, 1)) and ((5,40), (X, 2)) are concerning both the edge (5,40). This means that the graph can have 2 triangles that containing the edge (5,40), if the edge (5,40) exist. The 2 triangles will be (1,5,40) and (2,5,40). This is logical because an edge can participate to multiple triangles in a graph.

The 2 "dummy" values "-1" and "X" are need for the case where a probability of an edge is equal to 0 or 1. Suggest the key-value pair ((2,5),1). We would not be able to suggest if this key-value pair is of type **i)** or **ii)**. Solely, if its says that the edge exist and its probability is 1, or if it says that if this edge exist, then the triangle (1,2,5) exist. In case where the edges' id are strings or the graph has no edges with probability 0 and 1 or the graph is preprocessed so that all the $ids = id + 2$, then the "dummy" values can be avoided, resulting to less memory usage for the Spark cluster.

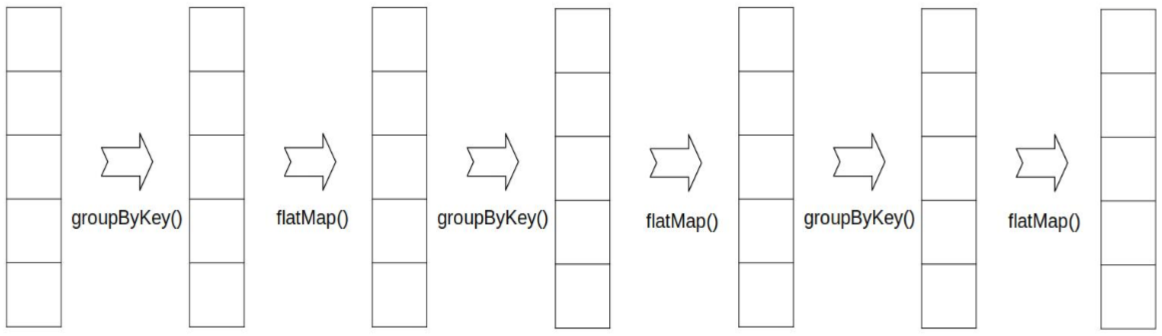

**Figure 2.** The RDD transformations pipeline for calculating all the triangles of a graph (subgraph)

After executing the second *groupByKey()*, the produced RDD, which elements are key-value pairs, is presented below.

((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)])

The key of a key-value pair shows which edge this key-value pair regards. A key-value pair in this case can be categorized based on its value (list), to some of the following 3 categories; **1)** it contains a key-value pair of type **i)** and none of type **ii)**, **2)** it contains no key-value pair of type **i)** and one/multiple of type **ii)**, **3)** it cointain of both types **i)** and **ii)**. Example ((1,2), [(0.80,-1)]) belongs to **1)**, so edge (1,2) exist, but no triangle exist in the graph that edge (1,2) participates. Next, ((5,40), [(X,1), (X,2)]) belongs to **2)**, so the edge (5,40) don't exist, so the nodes that had this edge as a "candidate" aren't constructing a triangle with this edge, which they could if edge (5,40) was exist. As for **3)**, the example ((2,5), [(X,1), (0.40,-1)]) belongs to **3)**, so edge (2,5) exist, and it participates in triangle (1,2,5). If the value of the key-value pair was [(X,1), (0.40,-1), (X,0)], instead of [(X,1), (0.40,-1)], then the edge (2,5) would participate also in the triangle (0,2,5), if a node with id=0 and the edges (0,2) and (0,5) were existing in our example graph.

Now, after the application of the second *flatMap()* the produced RDD is the follow.

(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))

The above RDD is constituted by key-value pairs, where the key reflects a node and the value is a key-value pair, whose key is an edge and the value the probability of that edge.

When the *flatMap()* is executed to a key-value pair that belongs to category **2)**, it returns an empty list, since the edge that the key-value pair concerns doesn't exist, so no corresponding triangles can exist. For the next 2 cases, solely, if the *flatMap()* is executed to a key-value pair that belongs to category **1)** or **3)**, the number of produced key-value pairs is going be *N*, where *N* = the number of elements in its value, which is a list. For key-value pairs of category **1)** the value always contains only one key-value pair that shows that this edge exist, so only one key-value will be returned in this case. The key is going to be the smaller id of the edge. Then the key of the value, which is a key-value pair, is going to be the edge, and its value the probability of the edge. For example ((1,2), [(0.80,-1)]) $\rightarrow$ (1,((1,2),0.80)). For the key-value pairs of category **3)** the key-value pair in the value (list) that is of type **i)**, which is exactly 1, is going to be mapped exactly as the previous example. Then as for the key-value pairs of type **ii)** instead of taking the smaller id of the edge an put it as the key to the produced key-value pair, their values are placed as the key, which are indicating which node has declared this edge as a "candidate". The rest is same as the type **i)** case. For example ((2,5), [(X,1), (0.40,-1)]) $\rightarrow$ [(1,((2,5),0,40)), (2,((2,5),0,40))].

In hence, what we accomplish with this *flatMap()* is that after executing the last *groupByKey()*, for each node, we will have in a list all the edges that participate in existing triangles that have the corresponding node as an "anchor". The "anchor" node of a triangle is the node with the smallest id among the 3 nodes that it is constituted. Consequently, after the execution of the last *groupByKey()* the produced RDD is as follow.

(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)])

The above RDD has the exact same form as the the RDD we had after the first *groupByKey()* execution, with one difference. Now the value (list) of the key-value pairs are containing also the "candidates" that each node had set after the execution of the first *flatMap()*, and actual exist in the graph. Consequently, handling this value (list) for each key-value pair, of the above RDD, through the last *flatMap()* of the pipeline, results to the calculation of all the triangles of our graph. The resulting RDD from the *flatMap()* contains key-value pairs, where the keys are showing the triangles and the values their corresponding probabilities.

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

Examining the result of the last *groupbByKey()*, we see that despite the fact that our example graph has 5 nodes in total, only for 2 nodes their corresponding triangles are calculated. Furthermore, in the final results, the triangles (1,2,5) and (1,2,40) are generated only once, despite that both triangles contain both nodes (id=1 and id=2), for which at the end we calculate all the triangles their participate. Looking at the value (list) of the RDD produced after the last *groupbByKey()*, we can see that the node (id=1) generates the 2 triangles. Consequently, our implementation finds each triangle only once correctly, avoiding unnecessary recomputations.

In order to achieve the aforementioned, our implementation follows the subsequent logic; for each triangle of the graph, the node with the smallest id knows the 2 edges (the 2 edges in which it participates) of the triangle, and set the third edge that it doesn't know if it exist as a "candidate" (which must be examined if it exist). At the end, for each node, their "candidates" that actual exist are send to them back to calculate the triangles their participate and their corresponding probabilities .

Summarizing, in our implementation the preprocessing procedure and the first *groupbByKey()* result that for each triangle, the node with the smallest id knows the 2 edges of the triangle (the 2 edges in which it participates). Then, the first *flatMap()* sets the "candidates" for each node. The rest transformations of the pipeline are sending the "candidates" that actual exist to the corresponding nodes, which then are calculating the triangles their participate and their corresponding probabilities.

**Baseline implementation**. In order to implement the *baseline* algorithm, we execute the above discussed RDD-based triangle calculation implementation at line 1 of algorithm 1.

**Optimization1 implementation**. In order to implement the *optimization1* algorithm, we execute the above discussed RDD-based triangle calculation implementation at line 3 and 20 of algorithm 2. For both cases, the RDD implementation finds all the triangles of the subgraph on which it is executed, and then the $K$ heaviest triangles are returned.

**Optimization2 implementation**. In order to implement the *optimization2* algorithm, we changed the above presented RDD-based triangle calculation implementation at one point. The first *flatMap()* that sets the "candidates" takes under consideration the $TopK_{mem}$ (algorithm 3, lines 1 and 5). When a "candidate" is found, we also know which triangle is going to be generated if it exist. Consequently, at first we examine if this triangle exist in the $TopK_{mem}$ or not. If it not exist, then it is set as a "candidate", else it is not set as a "candidate". If it is not set as a "candidate" then it will not be recalculated, avoiding that way recomputations. Furthermore, in lines 4 and 22 of algorithm 3, in order to have the top-k heaviest triangles for the corresponding subgraphs, the result of the presented RDD implementation has to be merged with the corresponding $TopK_{mem}$ and then return the $K$ heaviest triangles of this merged set. The $TopK_{mem}$ is implemented using the *broadcast variable* that Apache Spark offers.

## 5 Results

Intending to understand the algorithms behavior and efficiency and conclude which implementation (Graphframes or RDD API) is more time-efficient, this sections presents the execution results based on our experimentation and then presents the conclusion that are produced based on the results.

The executions comparisons are implemented in a way to examine which algorithm is the most efficient for each implementation type (Graphframes and RDD API) and examine, which type of implementation is the most efficient for our task eventually. Furthermore, we examine the impact of the threshold value assigment in the *optimization1* and *optimization2* algorithms. At last, we execute a stress-case for each algorithm-implementation case, solely, the number of k triangles to return is equal to "number of total triangles in the graph + 1".

We carry out a number of different executions based on different parameters; **1)** Different weight distributions; normal, uniform, power low **2)** different number of cores Spark utilize; 1, 2, 8 **3)** different number of k heaviest triangles to return; 100, 1000, 10000. All these different executions were carried out for both implementations types (GraphFrames and RDD APIs). Consequently, 162 executions were carried out in total.

In more detail, all the execution results are presented in corresponding diagrams. The interested reader can find the full experimentation results in the Github repository here [3].

**Dataset**. We evaluate our implementations through a Real World dataset provided from the Stanford Network Analysis Project repository (SNAP[4]), which is the artists dataset[5]. This dataset referred to Facebook pages and it was collected in

[3]Experiments
[4]Snap website
[5]Artist dataset link

November 2017. It represents blue verified Facebook page networks. The network's nodes represent the artists' pages and the network's edges represent the mutual likes among them.

This dataset is an undirected certain graph with no weights (probabilities) assigned to its edges. In order to create a probabilistic graph from it, we assign probabilities to the edges, based on various distributions; *uniform*, *normal*, and *power law*. Following, we present the structural characteristics of the graph.

- 50.515 nodes
- 819.306 edges
- 2.273.700 triangles

**Hardware specifications**. We executed our experiments in a machine with the following characteristics.

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

**Graphframes Results**. Figure 3 presents the results for the GraphFrames API case. Both optimization-cases were executed with threshold value equal to 0.8 for all the executions.

From the results we can see that the 2 optimizations implementations are faster than the baseline implementation, except the normal distribution case. In this case, edges distribution follows the normal distribution with mean=0.5 and standard deviation=0.1, consequently, the optimizations algorithms can not find the top-k heaviest triangles with the first decomposition (heavy and light subgraphs), leading to recomputations that makes them slower than the baseline-case in this dataset. Another observations is that in the cases (uniform and power low), where the optimizations algorithms calculate the top-k heaviest triangles with the first decomposition, the higher the k value, the higher the execution time. Next, regarding the optimization2-case, we can see that it is slower compared to the optimization1-case. This happens due to the characteristic described at "optimization2 implementation" paragraph in subsection 4.1. The time needed to look if a value exists in a data-structure is more compared to just calculate the product of 3 numbers (that the baseline does), especially if the k parameter is high.

**RDD Results**. Figure 4 presents the results for the RDD API case. Both optimization-cases were executed with threshold value equal to 0.8 for all the executions.

At first, in the cases (uniform and power low), where the optimizations algorithms calculate the top-k heaviest triangles with the first decomposition, the 2 optimizations-cases execute times are equal, so this is the reason why in the corresponding plots only the ones execute times are presented. Now, regarding the optimizations implementations efficiency compared to the baseline implementation, the same applies as the GraphFrames-case, except that for the normal-distribution case, for low k and high number of cores the optimizations-cases outperform the baseline-case. On the other hand, the k values doesn't impact the time efficiency in the uniform and power low distribution cases as it was in the GraphFrames-case. From the results in Figure 4 we can not understand which optimization-case is the optimal. The only indication that could make as believe that that the optimization2-case is more efficient is that from 2 to 8 cores the optimization2-case has more percentage reduction concerning the execution time than the optimization1-case. Consequently, as the Spark cluster scales, optimization2 would probably outperform optimization1.

**Graphframes vs RDD**. Above we presented the comparison of the algorithms implementations, individually for each type of API. Next we present a comparison between the 2 API types implementations.

In Figure 5 we compare the GraphFrames-based optimization1 implementation (that is better than the optimization2 implementation) with the 2 RDD-based optimization implementations. Figure 5 shows a comparison among them, for each distribution and only for the case where they had been executed with 8 cores.

What we see is that the GraphFrames-based case outperforms clearly the 2 RDD-based cases in the normal distribution circumstance. As for the other 2 distributions, the 2 RDD-based cases outperform slightly the GraphFrames-based case.

**Threshold impact**. The threshold value that we give as input in the optimizations algorithms has a great impact on their execution times. In the normal distribution case, in both implementations types, we saw that with threshold=0.8 the baseline-case outperforms the 2 optimizations-cases.

Next, Figures 6 and 7 present how with just giving as input a different threshold, for each implementation type, the corresponding optimizations-cases outperform now the baseline-case.

Consequently, it is crucial to examine the weight distribution of the graph to set the optimal threshold value in order to reduce the execution time of the optimization-cases as much as possible. The objective must be to set the threshold, so the top-k heaviest triangles to be found, if it is possible, with the first decomposition.

**Stress-case**. At last, we examine the worst-case scenario for our optimization algorithms. For this case, we run experiments for a k larger than the total number of triangles in the network. As one can see in Figures 8 and 9, the baseline line algorithm was the fastest in this case. This is the expected behavior because for the optimization algorithms any other decomposition with threshold > 0 was a waste of time. Consequently, for k larger than the total number of triangles in the network, the baseline algorithm should always be preferred than the optimizations algorithms.

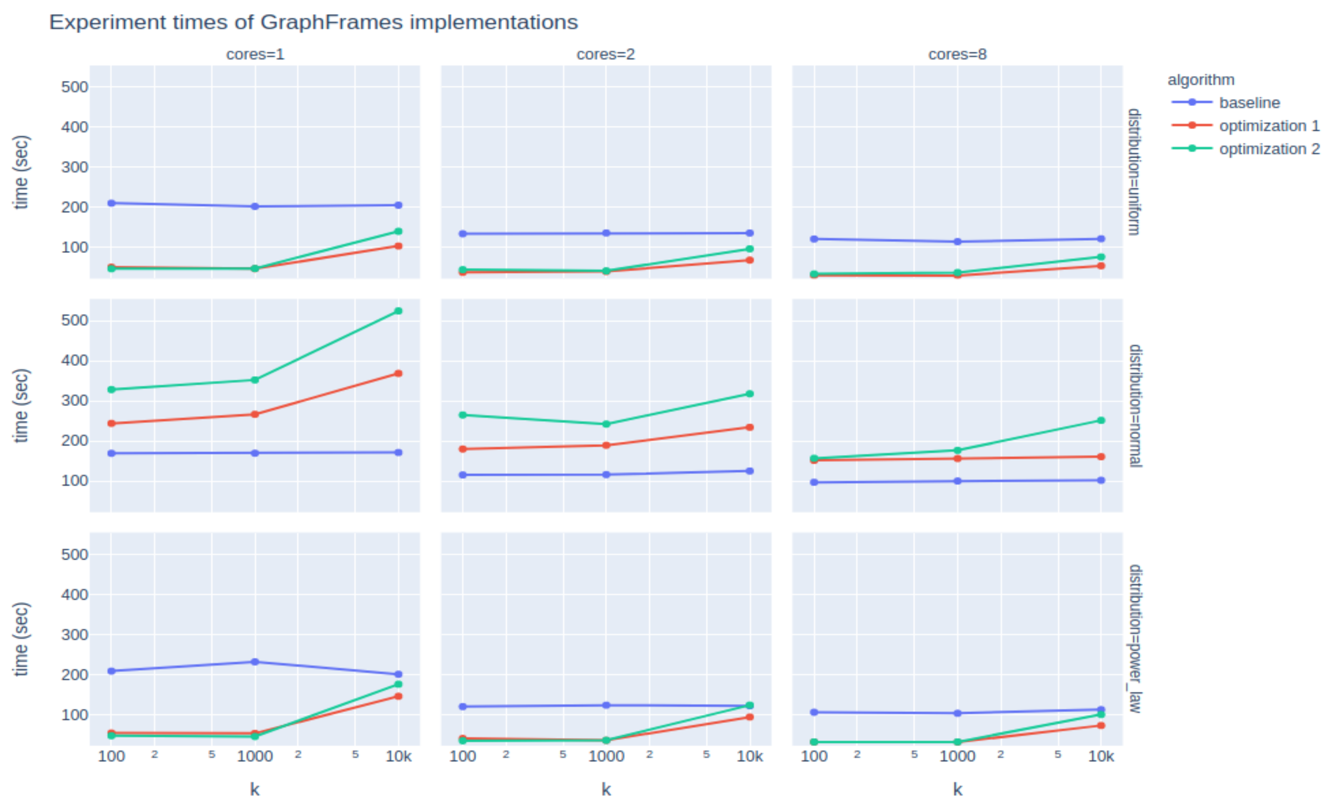

**Figure 3.** Experiment times for GraphFrames implementations (optimization threshold=0.8)

**Conclusion**. The results of the experimentation-phase gives us an indication of the algorithms implementations characteristics.

At first, there is no clear winner. The decision regarding which implementation to use is dependent from the edge probability distribution. Consequently, prior knowledge of probability distribution can be a significant help in selecting the appropriate algorithm and threshold value, both that lead to the reduction of the execution time. As for the extreme case, where the k value is higher than the number of the existing triangles in the network, it is better to execute the baseline implementation than the 2 optimizations implementations.

Finally, in order to examine the true nature of the algorithms implementations behavior, they must be executed on true massive probabilistic uncertain graphs. For example, for the normal distribution case, where for both API types implementations the baseline-case is faster than the 2 optimization-cases, in a true massive probabilistic uncertain graph that should not be true. Even for a reasonable number of different decomposition, namely reducing the threshold value because the current heavy subgraph doesn't contain k triangles, the optimizations implementations should be faster than examining the whole graph (baseline-case). Finally, examining the implementations in massive datasets could lead to declare a clear winner.

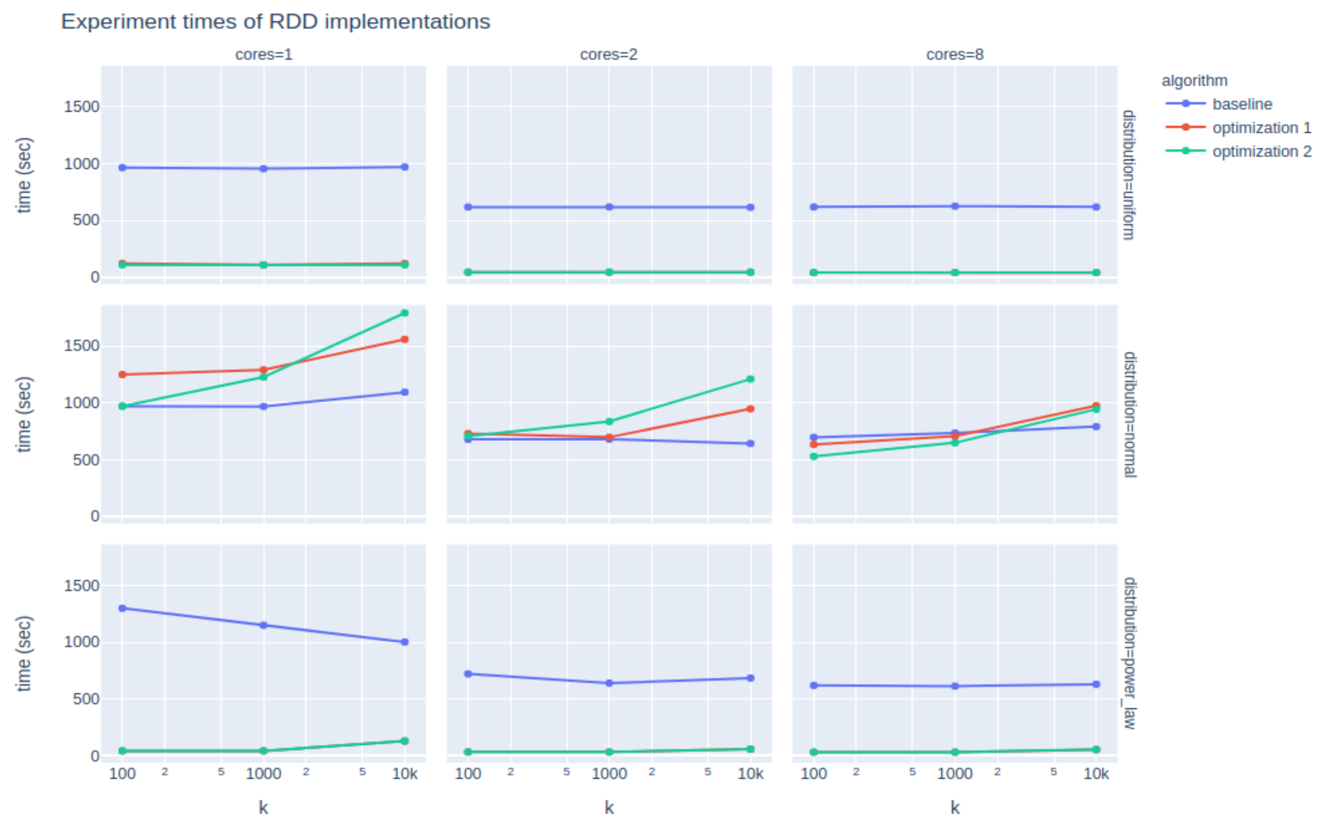

**Figure 4.** Experiment times for RDD implementations (optimization threshold=0.8)

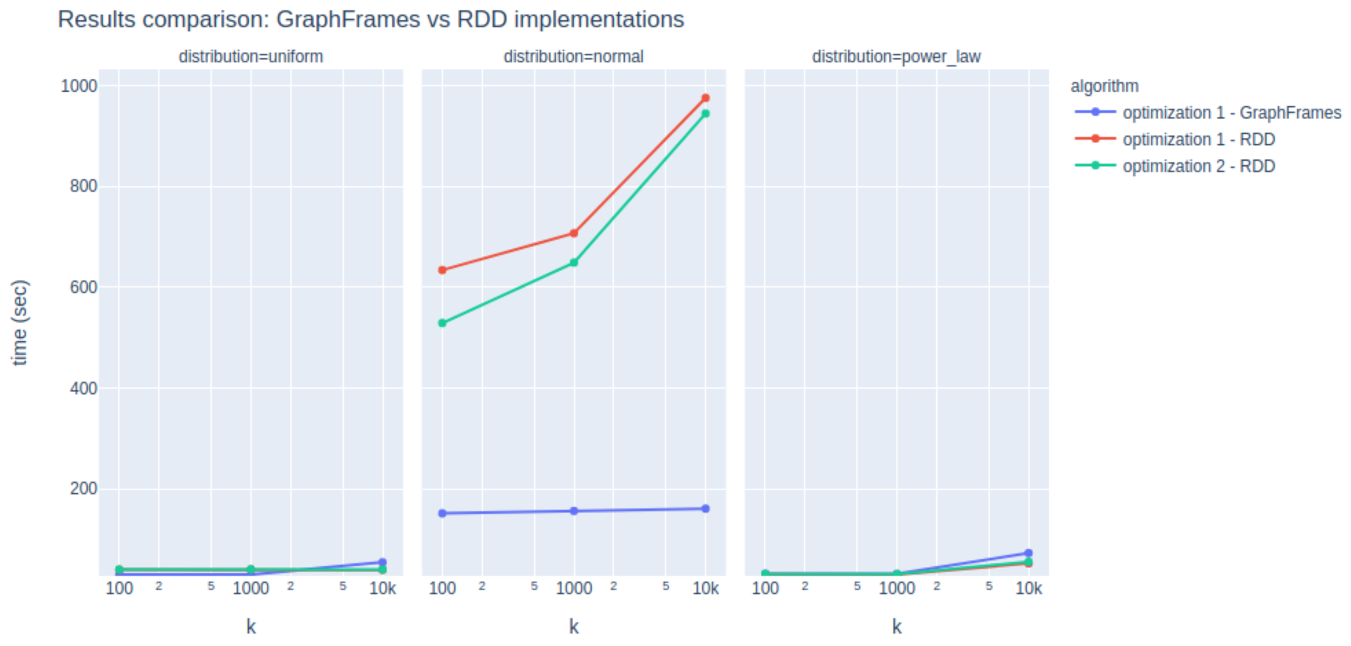

**Figure 5.** Results comparison: GraphFrames vs RDD implementations (cores=8, threshold=0.8)

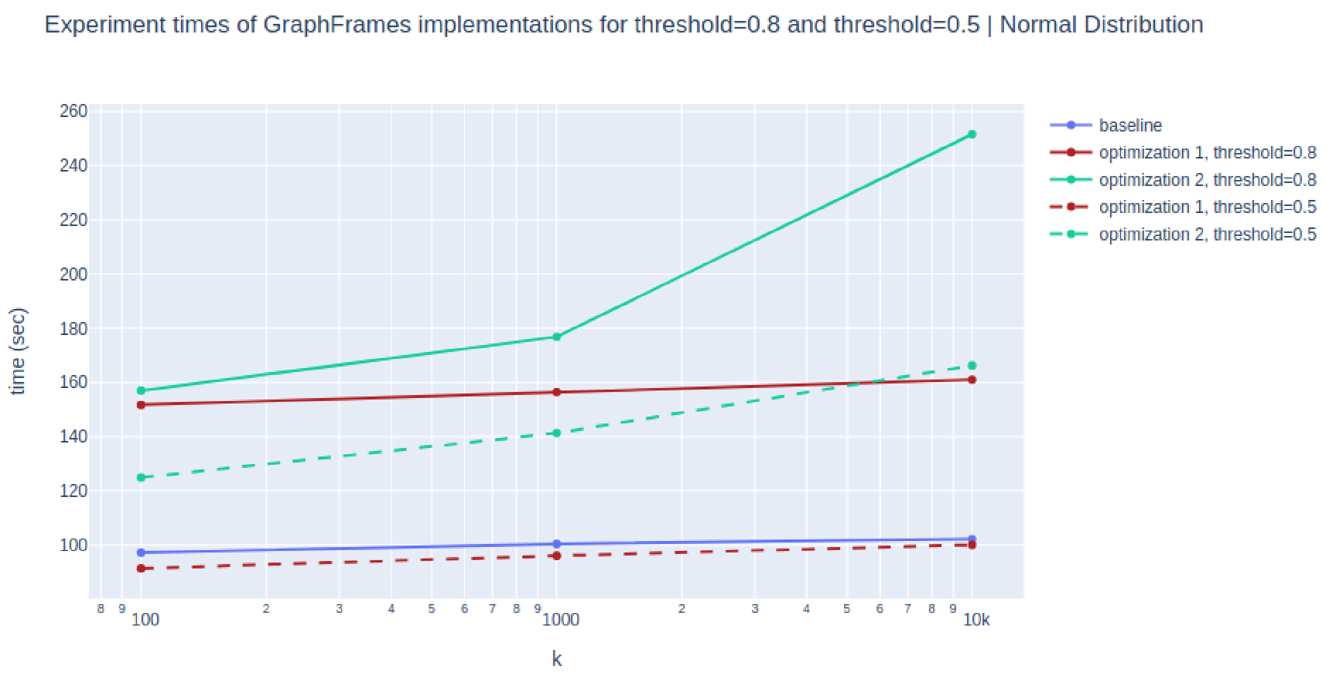

**Figure 6.** Experiment times for different thresholds

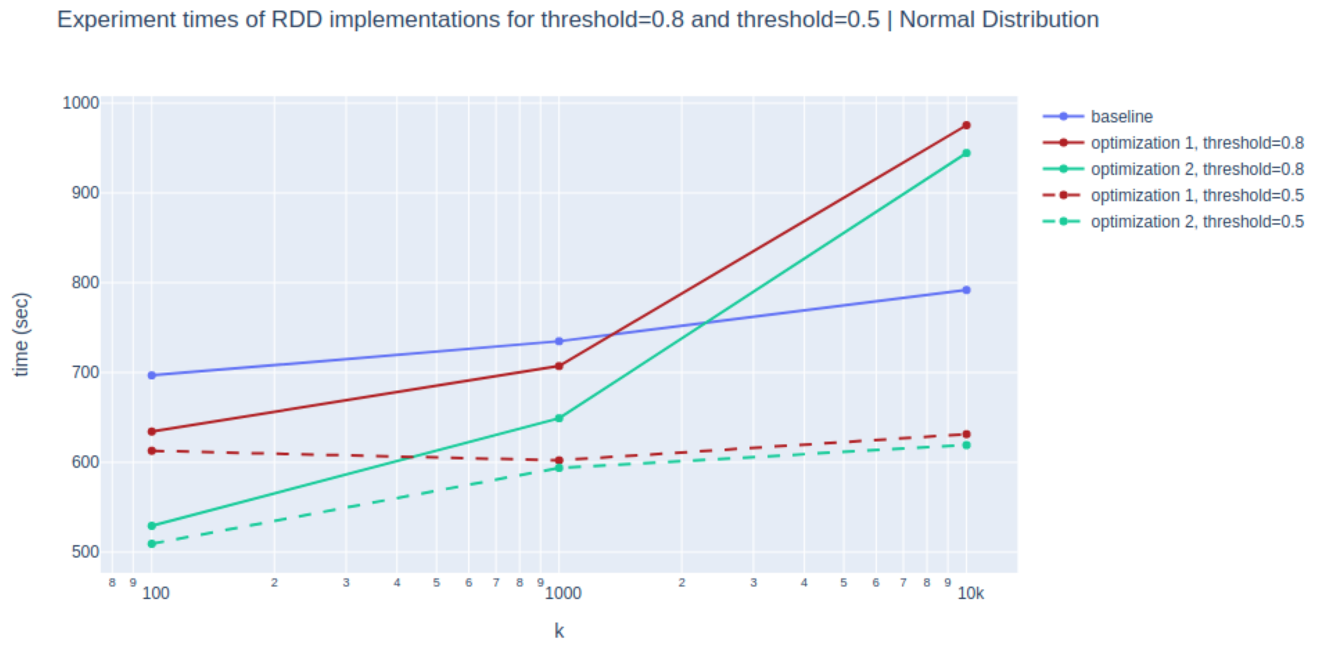

**Figure 7.** Experiment times for different thresholds

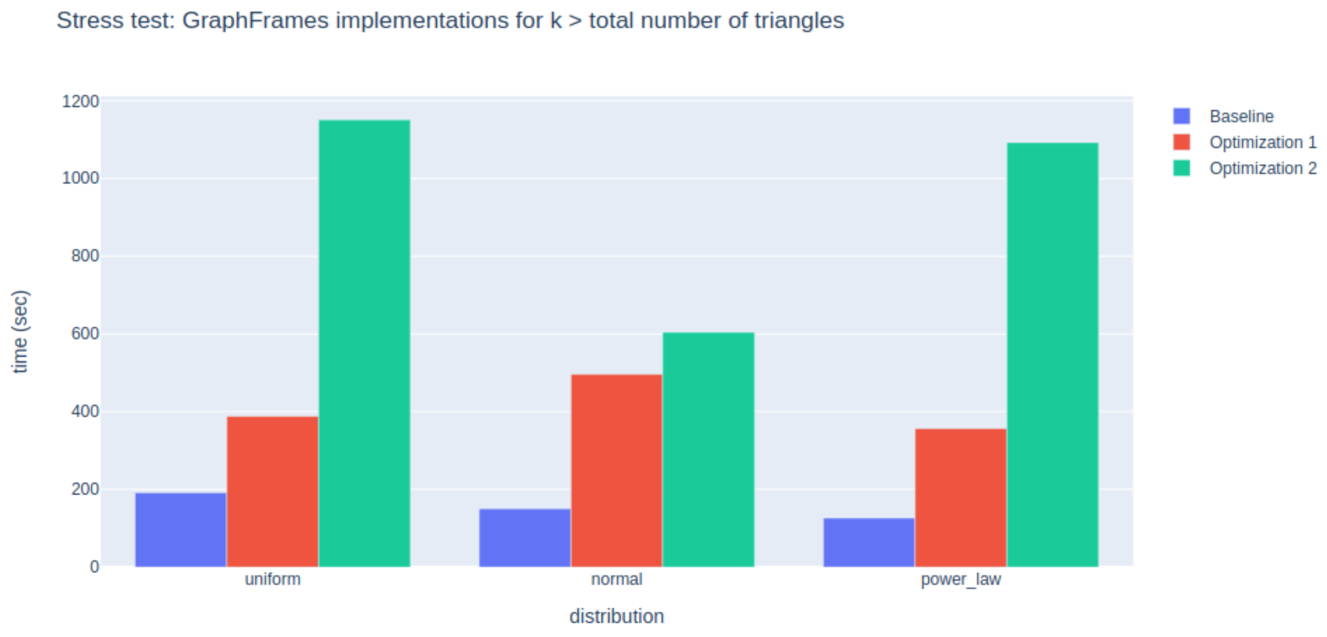

**Figure 8.** GraphFrames Stress Test case for k>total triangles

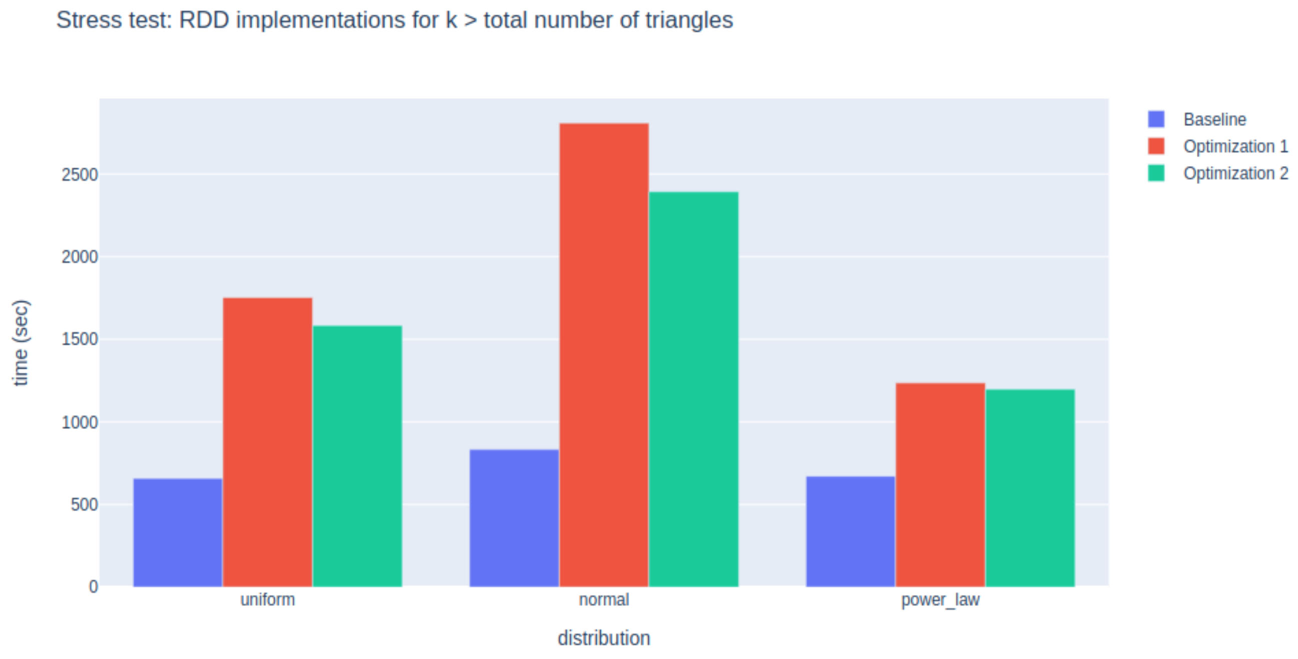

**Figure 9.** RDD Stress Test case for k>total triangles