Social Network Graph Analytics

Find Blocking And Ghosting

Researcher
George Mitsos, undergraduate student in Computer Science, University of Crete

Supervisor & Director
Dr. Polyvios Pratikakis, associate professor at the University of Crete

1. Introduction

With more than 3.5 billion users worldwide, social media are an integral part of most people's everyday lives. The variety of platforms increases as well as the social groups that enter the social network world. In this ever-expanding ecosystem, we need to find ways to spot unusual behaviors and provide fine experience and service for most users while also analyzing the way people interact in the Social Network. To achieve this, we have to study the behavior and actions taken between users and understand what caused them and why. Then we will have a great tool that will be able to predict unconventional behavior and avert it from the platform before any harm is done.

In this paper, we propose a simple way to spot blocking between users efficiently by embedding people's behavior and need to interact in social media into a graph analytics software that reduces the internet social media interactions into a real-life interaction model.

To achieve this, we first need to understand the theoretical background and technology behind problems like this and how we represent a social network in a human-comprehensible way.

The most widely used technique in representing networks in Computer Science is with the utilization of **Graphs.** A graph is a simple diagram consisting of nodes (or Vertices) and edges. Each node (users) may or may not be connected (by an edge) with other nodes. This connection (edge) represents a relation between the users, and we call this type of diagram a **Graph.** A simple example is shown in Figure 1.

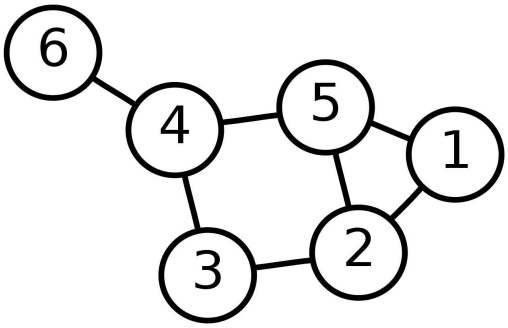

Figure 1. Example of an Undirected Graph.

In these graphs, we can represent various information on social media such as likes, mentions, quotes, etc. By applying simple procedures on a graph, we can draw useful information for users, trends, interactions and more. We can later use this information to our advance to make a platform more reliable, useful and friendly.

The power of this technique is much greater when **Multilayer Graphs** are used to draw this type of information. A **Multilayer Graph** (or Multilevel) is a graph that is represented in subsets of a network, each of these representing a relation. Social Media Networks are more sensible to represent on a Multilayer Graph since they consist of many different kinds of interactions between users and it is easier to cope with grouped data as long as the layers serve the purpose. An example of a **Multilayer Graph** is shown in Figure 2.

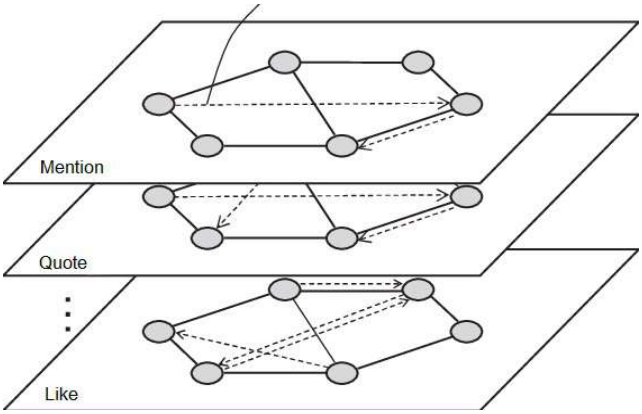

Figure 2. Example of a Multilayer graph with layers representing Quote, Mention and Like relations between Users.

In social network data analytics, when utilizing multilayer graphs, we can extract specific information while also reducing the noise of our results. We can also combine all layers for extracting a complete series of interactions between users, that we can later process and retrieve useful information about user relationships, habits, liking or disliking and even identify possible blocking between users.

In our case, since we are investigating blocking on Twitter, we don't need to process each layer separately, as blocking a user means total disruption of any interaction and so combining the multilayer graph into a central "general interaction" graph is efficient, resource-efficient while also more manageable and much simpler to work with.

To achieve this, we used Apache Spark, a distributed processing system commonly used for big data workloads that offer excellent optimization and monitoring tools while also automating the process of breaking the data into pieces that get distributed among a cluster. Using Spark is essential since the workloads of these problems could get enormous and, therefore, would out-scale single-node clusters that inevitably would struggle to calculate the output.

The biggest disadvantage when dealing with this type of data is that there is no simple way to intersect the output of the analysis with reality. Since there is no "ground truth", the outcome of this method is approximate.

2. Implementation

As mentioned above it is common for Social Media Networks to be represented as a multilayer graph and then processed. The implementation we are about to introduce, is based on a multilayer graph that consists of 5 layers :

1. Favorite
2. Mention
3. Quote
4. Reply
5. Retweet

Each of these layers is a graph that represents the traffic of a specific relation throughout the social network in a day. In addition we have two years span of these files that gives us a representative sample to work with and finally more accurate results and conclusions.

The technique we used for implementing the blocking prediction algorithm consists of 4 simple steps:

- Reducing the daily relation graphs into a day graph consisting of all layers.
- Reducing the day graphs into a main graph while holding the needed information.
- Grading scheme.
- Filtering the data by certain criteria.

For the implementation of the program we used Apache Spark combined with Scala's powerful API.

2.1. Reducing the daily relation graphs

Reducing the daily relation graphs into a single day graph consisting of all five layers is compulsory for execution timings and efficiency as we only have to deal with only one graph per day instead of five while also as mentioned in Introduction we don't need to investigate each layer separately.

Since we only care about the interaction itself and not the type of it, during this stage we only need to collect the list of edges (interactions). The output of this stage then proceeds for further processing in the next stage.

The function is responsible for delivering us this list is described below in Scala.

```
1  def getDay(noDay: Int) :RDD[( (Long,Long), (Int,Int,Boolean,Int,Int,Double,Double,Int,Int))] ={
2
3      val favorite = spark.sparkContext.textFile(fileDir + "favorite/day" + noDay.toString + ".txt")
4      val favoriteEdges = favorite.map(e ⇒ e.split(" ").map(i ⇒ i.toLong ))
5
6      val mention = spark.sparkContext.textFile(fileDir + "mention/day" + noDay.toString + ".txt")
7      val mentionEdges = mention.map(e ⇒ e.split(" ").map(i ⇒ i.toLong ))
8
9      val quote = spark.sparkContext.textFile(fileDir + "quote/day" + noDay.toString + ".txt")
10     val quoteEdges = quote.map(e ⇒ e.split(" ").map(i ⇒ if(i≠"quote-deleted")i.toLong else -1.toLong))
11
12     val reply = spark.sparkContext.textFile(fileDir + "reply/day" + noDay.toString + ".txt")
13     val replyEdges = reply.map(e ⇒ e.split(" ").map(i ⇒ i.toLong ))
14
15     val retweet = spark.sparkContext.textFile(fileDir + "retweet/day" + noDay.toString+".txt")
16     val retweetEdges = retweet.map(e ⇒ e.split(" ").map(i ⇒ i.toLong ))
17
18     val temp = favoriteEdges
19       .union(mentionEdges)
20       .union(quoteEdges)
21       .union(replyEdges)
22       .union(retweetEdges)
23
24     val dayEdges = temp.map(e ⇒ ( (e(0),e(1)),(1,0,false,1,0,0.0,0.0,0,0) ))
25
26     val returnDay = dayEdges.reduceByKey((a,b)⇒a)
27
28     returnDay
29 }
```

In line #1 is the definition of function **getDay**, which is responsible for retrieving the list of the edges that interacted in a day. It takes a single Int argument **noDay,** which represents the number of the day to retrieve data from.

The return value is a resilient distributed dataset (RDD), with each entry consisting of two tuples. The first tuple is of type (Long, Long) and represents the srcID and dstID respectively. The second tuple consists of the attributes to be monitored. These attributes represent (in order) :

- Int : Consecutive Active Days
- Int : Consecutive Non Active Days
- Boolean : Defines whether the edge is on a Non Active Streak
- Int : Maximum Positive Streak
- Int : Maximum Negative Streak
- Double : Average Positive Streak
- Double : Average Negative Streak
- Int : Total Positive Streaks
- Int : Total Negative Streaks

As shown above, most of the attributes are Streak variables. A Streak variable counts users' consecutive active (Positive) or non-active (Negative) interactions. By doing this, we can understand how strong a relationship between two users is.

In lines #3 - #16 the program collects the data from the edge files by opening each file using the spark function textFile which returns a RDD of strings each of these representing an edge and additional information contained by the graph file.

The strings are then processed using the spark map() function. Map() is a transformation operation that is used to apply transformation on every element of a RDD, DataFrame or a Dataset and returns a new data structure with the same type. By using map(), the program is able to iterate through the elements of the RDD that created in the previous step while tokenizing the strings by the space character. Then another map comes in place to transform the string tokens into Long values.

In lines #18 - #22 the program merges the RDDs of the five interaction types into one using the function union().

In line #24 by using map(), the program is removing any additional information the graph file contains and keeping only the IDs of the nodes consisting the edge, while also attaching the attributes mentioned above with their initiative values. These initiative values will be explained in-depth in section **2.2** .

In line #26 the program utilizes the reduceByKey() function which is a frequently used transformation operation that performs aggregation of data that have the same Key value. The key value in our RDD is the first tuple (srdID, dstID), so basically the edge itself. While reduceByKey() is able to perform actions between the entries that get reduced in this stage it is compulsory to filter the RDD from duplicates and so it returns one of the two entries intact. After this step we have the final product of each day which is returned after every call of the function **getDay().**

2.2. Reducing the day graphs into the main graph

In this stage we calculate the attributes of the edges that occurred in the previous stage that helps us monitor the daily interactions between users. These attributes help us understand the way nodes interact over a period of time, eg. How frequent is an interaction and how long is an interaction. In this stage we can also keep any additional information that can help us identify a motif behind an interaction.

The function responsible for delivering us the list with the updated attributes is described below in Scala.

```
1  def mergeDays(  central:RDD[( (Long,Long), (Int,Int,Boolean,Int,Int,Double,Double,Int,Int))],
2                  newDay:RDD[( (Long,Long), (Int,Int,Boolean,Int,Int,Double,Double,Int,Int))]
3               )  :RDD[( (Long,Long), (Int,Int,Boolean,Int,Int,Double,Double,Int,Int))] = {
4
5
6                  val temp = central.union(newDay)
7                  val notActiveToday = central.subtractByKey(newDay)
8                  val activeToday = temp.subtractByKey(notActiveToday)
9                  val activeTodayReduced = activeToday.reduceByKey( (a,b) => (
10                     a._1 + b._1,    //consecutiveActive days
11                     0,              //consecutiveNotActive days - Will not be reduced
12                     false,          //turn (true since first time no see edge) - Will not be reduced
13                     {   //Longest Positive Streak
14                         if((a._1 + b._1)> a._4.max(b._4)) a._1 + b._1
15                         else (a._4.max(b._4))
16                     },
17                         //Longest Negative Streak - Will not be reduced
18                         if(a._5.max(b._5)<(a._2 + b._2))a._2.max(b._2)
19                         else a._5.max(b._5),
20                         //Average Positive Streak
21                         a._6.max(b._6).toDouble,
22                         //Average Negative Streak
23                         {
24                             if(a._1 == 0 || b._1 == 0) ((a._7.max(b._7) * a._9.max(b._9) + a._2.max(b._2)) / (a._9.max(b._9) +1)).toDouble
25                             else a._7.max(b._7).toDouble
26                         },
27                         //Total Positive Streaks
28                         a._8.max(b._8),
29                         //Total Negative Streaks
30                         {
31                             if(a._1 == 0 || b._1 == 0) a._9.max(b._9) +1
32                             else a._9.max(b._9)
33                         }
34                     ))
35                 val notActiveTodayMap = notActiveToday.map(e=> (e._1,  (0,              //consecutiveActive days
36                                                                         e._2._2 + 1,    //consecutiveNotActive days
37                                                                         true,           //turn (true since first time no see edge)
38                                                                         e._2._4,        //Longest Positive Streak
39                                                                         e._2._5,        //Longest Negative Streak
40                                                                         {   //Average Positive Streak
41                                                                             if(e._2._1>0) ((e._2._6 * e._2._8 + e._2._1) / (e._2._8 + 1)).toDouble
42                                                                             else e._2._6.toDouble
43                                                                         },
44                                                                         e._2._7.toDouble,    //Average Negative Streak
45                                                                         {   //Total Positive Streaks
46                                                                             if(e._2._1>0) e._2._8+1
47                                                                             else e._2._8
48                                                                         },
49                                                                         e._2._9
50                                                                         ) ) )
51                 var newCentral = activeTodayReduced.union(notActiveTodayMap).repartition(partitions)
52                 newCentral
53 }
```

In line #1 - #3, is the definition of the function mergeDays which is responsible for updating the attributes and merge the daily RDDs. It takes two arguments which are both day RDDs consisting of the tuples described in section **2.1**. The central argument is the day number n and the newDay argument is the day n+1. The return value is the updated day RDD.

In line #6, the program merges the central and newDay into one RDD called temp using the union() function.

In line #7, the extraction of the RDD consisting of the edges that did not interact in the newDay takes place, by using subrtractByKey() function. SubtractByKey() returns a new RDD consisting of the of entries that have pair with matching key between the inserted data. Since the newDay can only have edges that were active in the last day, by subtracting the entries with the same key from the central RDD we get the inactive edges.

In line #8, by using subtractByKey() in the union product of line #6 while subtracting the inactive nodes the

program extracts the complete list of active edges through the whole graph. By this time the edges that are currently on an active streak are contained twice in the structure. One instance contains the edge with the outdated data, and the second instance contains the edge with the initiative attribute values.

In line #9, by utilizing reduceByKey() the program removes duplicate edges while keeping track of the attributes. The letters a and b resembles the attributes of the two instances of the same edge, while the "=>" symbol stands for the initiation of the aggregation and transformation sequence.

- In line #10, addition of a._1 and b._1 happens which are the first attribute values of the edges that is the consecutive active days. Inevitably one of these two values is always 1 (the initiative value). The other can either be 0, which means that the edge interacted after a not active streak, or greater than zero which means that the edge was and still is on an active streak. The addition of these two values gives us the days of the current consecutive active streak.
- In line #11, the value 0 resembles the days of the current consecutive not active streak. Since any edge contained in the RDD is contained in the newDay RDD and therefore is active, this value in this reduction stage will always be 0.
- In line #12, the value false resembles that the edge is not currently on an inactive streak. Using the same logic as line #11 this value in this reduction stage will always be false.
- In lines #13 - #16, the program calculates the new Longest Positive Streak and replaces it with the old one in case it is greater.
- In lines #17 - #19, the program calculates the new Longest Negative Streak and and replaces it with the old one in case it is greater.
- In line #21, the program keeps track of the Average Positive Streak. It is important to notice that in this stage the average positive streak must not be calculated since the current positive streak is not completed and will eventually affect the data.
- In lines #24 - #25, the program calculates the Average Negative Streak, if the edge currently reduced is coming from a negative streak.
- In line #28, by using the similar logic as in line #21 the program keeps track of the Total Positive Streaks
- In lines #31 - #32, the program calculates the Total Negative Streaks if the edge comes from one, else the Total Negative Streaks remain the same.

At this point we have the Current Active Edges of the day, with their attributes updated. The remaining edges of the total graph are the edges that are currently inactive and proceed to mapping but in a Negative Streak state. The way this is done is similar to the Positive Streak reduction and is described below. It is essential to notice that in this stage the notActiveToday dataset does not contain any duplicates, and therefore we utilize map() function instead of reduceByKey().

- In line #35, the program by utilizing map() keeps intact the edge itself while updating its attributes. Since the edge is currently on a Negative Streak the consecutive active days counter will always be 0.
- In line #36, the consecutive not active days counter is updated by adding 1 to the current value.
- In line #37, the value true resembles that the edge is currently on a negative streak. At this stage this value will always be true.
- In lines #38 - #39, the program carries the counters for Longest Negative and Positive Streak that were previously calculated.
- In lines #41 - #42, the program calculates the new Average Positive Streak if the edge currently mapped came from a Positive Streak state.
- In line #44, the program carries the counter of the Average Negative Streak that was previously calculated.
- In lines #46 - #47, the program updates the counter of Total Positive Streaks by adding 1 to the current value if the edge currently mapped came from a Positive Streak state, else it keeps intact the previously calculated value.
- In line #49, the program carries the counter of Total Negative Streaks that was previously calculated.

At this point we have Current Not Active Edges with their attributes updated. The next step is combining the active and not active edges into a main graph called newCentral by using the union() function. In addition to optimize the performance of the program we used the repartition() function which allows us to change the distribution of the data among the Spark cluster. The repartition strategy that gave us the best results was dividing the data into (AvailableCores/2) partitions.

Finally, the function is completed while returning the newCentral. By repeating the execution cycle of these two functions we are able to retrieve data from any amount of days.

A flow diagram of the iteration is shown in Figure 3.

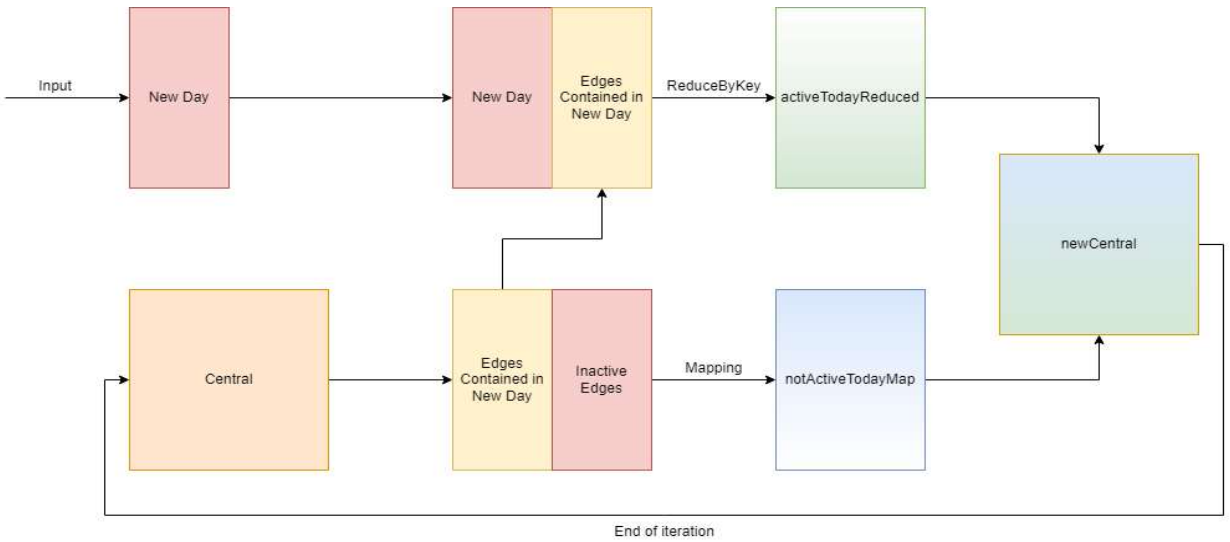

Figure 3. A flow diagram of an iteration of the main program

2.3. Grading Scheme

When all daily data are processed and combined, there are various ways to get the information we are looking for. A simple yet effective way to do this, is by using a grading scheme. The grading scheme we used is a new "Penalty" attribute for each edge. The penalty attribute represents a pure number; the greater this number is, the greater the chance that a user in an edge is blocked.

The way we add values in the Penalty variable is pretty arbitrary but still represents a way the real life interactions are. If we think of our real-life interactions with others, we all interact under the same attributes. We all have an average interaction streak and we all had a time when we were constantly close to a single friend or family member, so there is a maximum positive streak. In the same way we all had a maximum negative streak when a chain of events did not allow us to contact with our circle of friends and family. These exact values are the ones that we are interested in since blocking a user means you are no longer able to contact in any way.

In the end, it is efficient enough to model penalty numbers by the average and maximum negative streaks of an edge. But first we need to attach an importance grade for these two attributes.

As we reduce the problem into a real life situation model, we have no reason to think that other principles are taking place in this. So we can conclude that a current negative streak could be around the average negative streak so we can attach the importance grade of ***1*** (arbitrary). In the same way we can attach the importance grade of ***3*** (arbitrary) since the possibility of hitting a greater negative streak than the current maximum negative streak in a healthy relationship should be relatively small. We finally end up with this grading scheme:

Penalty = (CurrentNegativeStreak – AverageNegativeStreak) * 1 + (CurrentNegativeStreak – MaximumNegativeStreak) * 3

Since there is no ground truth, we cannot fit more accurate importance grades at this point. However, executing the Grading Scheme combined with a linear regression model is achievable for more precise grading, which will eventually give us more optimized and accurate results.

2.4. Filtering data by certain criteria

The final step is filtering the data and keeping the ones that are currently on a negative (inactive) streak. In addition as the grading scheme can give us negative values, we can filter those out considering that these values are produced from negative streaks that are smaller than the average negative streaks of the edge and therefore, most likely to happen even in a healty relationship.

The product of the filtering will be a list of edges with their attributes (2.2), a penalty variable for each edge that is greater than zero, and are currently in a negative streak. If we sort this list in descending order by penalty, the first elements will be the most likely blocked edges.

3. Additions and Upgradability

Dealing with such large data sets is memory and processing demanding, so in this research we made a relatively simple prediction model. By adding a variety of new attributes and a larger data set we could possibly achieve a better accuracy in our predictions.

Some attributes that would help us eliminate noise in our final data set, would be checking if the nodes of a likely blocked edge were active on other edges. By this way we could assure that the node was active in the period of

negative streak and not in a social media abstention state.

3. Performance and Efficiency

The script described above was tested on a cluster consisting of 4 PCs (or Workers) in various configurations using Spark. The parallelization that gave the best results was dividing the graph in **AvailableCores/2** partitions.
The specifications of each machine are described below:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Model name:            Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
CPU max MHz:           2400.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
Memory:                256GiB System memory
```

The graphs below represent the execution time of the script. In Figures 4 & 5, X-axis represents the number of executors for 1, 2 and 4 Workers (PCs). Each executor utilizes ten (10) cores.

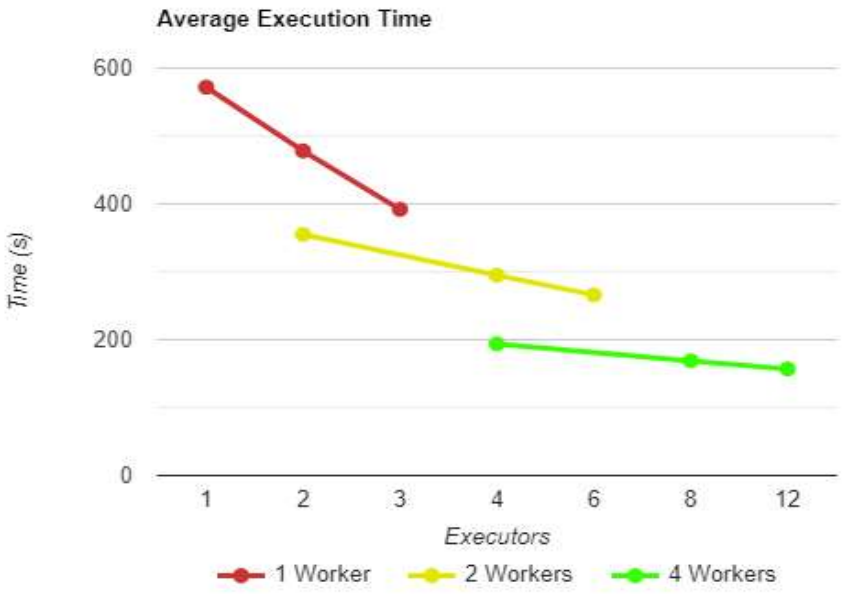

Figure 4. Average Execution Time by utilizing 1, 2 and 4 Workers
(10 cores / Executor)

In **Figure 4**, we represented the average execution time of each configuration on a line graph, with each line representing one (red), two (yellow) and four (green) workers. We can observe a flattening pattern while Workers increase, which means we approach the optimal execution time for this dataset. It is possible that adding Workers at this state while keeping the dataset size the same will eventually slow down the execution time.

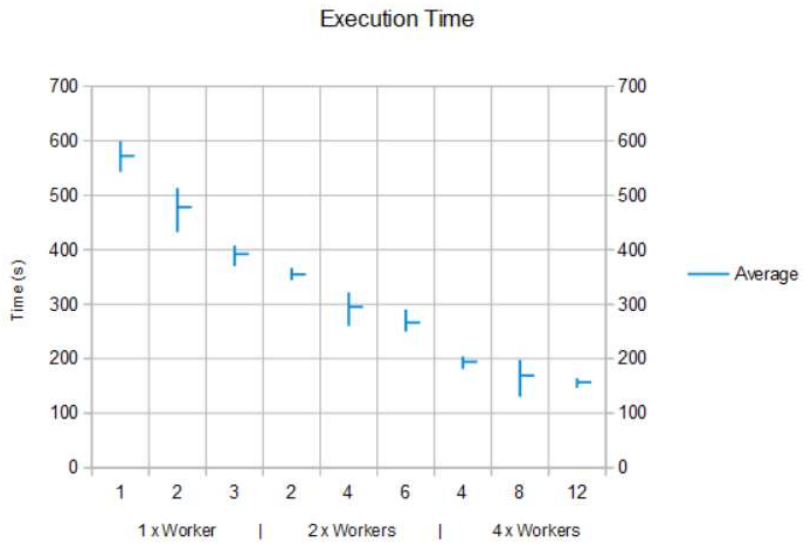

Figure 5. An Execution Time diagram with each point representing minimum, maximum and average execution time of each configuration
(10 cores / Executor)

In **Figure 5**, we again represent the average execution time of each configuration and the variance of each execution. The horizontal lines represent the average time, and the vertical represents the variance.

Number of Workers	Executors x Cores	Number of Partitions	1st Execution Time (s)	2nd Execution Time (s)	3rd Execution Time (s)	Average Execution Time (s)
4 x Workers	12 x 10	60	162	164	146	157
	8 x 10	40	130	198	179	169
	4 x 10	20	198	181	204	194
2 x Workers	6 x 10	30	258	290	250	266
	4 x 10	20	305	260	322	295
	2 x 10	10	367	355	344	355
1 x Worker	3 x 10	15	370	399	408	392
	2 x 10	10	432	490	513	478
	1 x 10	5	599	543	575	572

Figure 6. A complete table consisting of all timings, of all configurations

In **Figure 6**, there is a complete table consisting of Average, first, second and third execution times, the layout (Executors x Cores) of each configuration, and the number of partitions per configuration. The partition strategy that worked best was dividing the data into Total Cores / 2 partitions. We see that while adding resources in our cluster, the average execution time was speeding up. We can also see that single execution times were better in some cases than the configuration that utilizes more cores. This is probably due to the operating system and scheduling policies. The best single execution time was found in the first execution of 4 Workers and 8 Executors, achieving 130 seconds. In contrast, the worst was found in the first execution time of 1 Worker and 1 Executor achieving 599 seconds.

3.1. Utilizing More CPUs And Memory

By analyzing the graphs and the table, we observe that our script scales while we add more CPUs in the cluster. It is essential to point out that in Figures 4 & 5, we observe a continuous flattening pattern, which means that we are reaching the point that has the most optimal execution time. This means that if we attempt to add more CPUs, the total overhead of the process will negatively impact the execution time, assuming the dataset remains the same.

Although more CPUs seem to improve the execution timings, the **total** memory usage seems to increase as well. A graph representing memory usage is shown below.

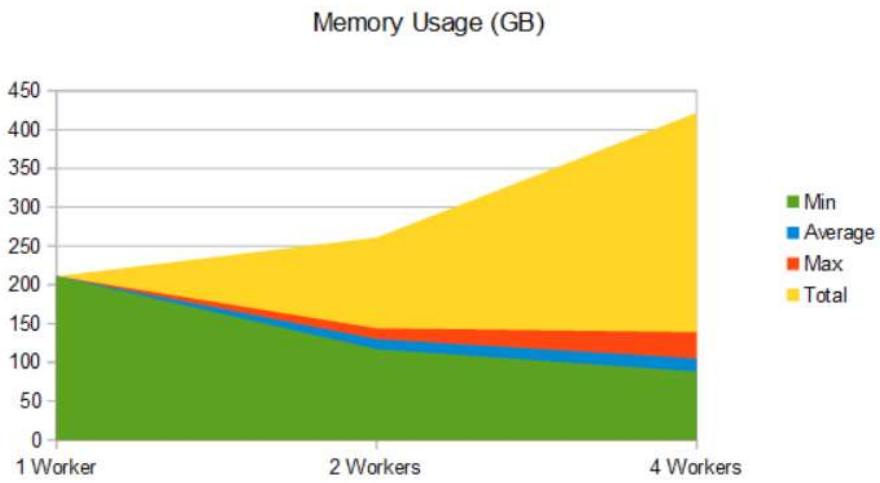

Figure 7. A graph representing minimum, average, maximum (per node) and total memory usage for each configuration

As we observe **Figure 7**, we see that the total memory usage is increasing while more workers are utilized but the memory usage of each worker drops significantly. This happens due to how Spark distributes data among the cluster. Since the script is using RDDs, Spark divides them into partitions and stores them across different nodes, that's why memory usage per node drops when more nodes participate in the cluster.

4. Results

The evaluation process of this script, since we don't have access to blocking catalogs of Twitter, is difficult, and the results might differ from reality.

The program described above was executed with a month's (28 days) of data from Twitter. The first step was to run it for the first 25 days of the month and filter the edges with the highest penalty value and so the highest blocking probability. A total of 104 edges had the greatest penalty value of 56,667. Then we compared these edges with the edges contained in the next three days of the month and found that 21 of these edges were active; 4 of them on day 26, 9 of them on day 27 and 8 of them on day 28. The final step was to eliminate duplicates in these 21 edges, so we ended up with a total of 17 edges that our program did not predict correctly and were currently active.

This gives our program an 83,65% success rate in predicting blocking. Although the results seem promising, we must keep in mind that no method can provide a high certainty result since there is no “ground truth” as described in the Introduction. Accessing more data and repeating the process will eventually increase the accuracy of these results.

5. Conclusions and Future Work

The Social Media Network is a massive bucket of information that can be extracted using simple yet effective techniques. This paper presented a scalable program that can effectively spot blocking between users with a success rate of 83.65%. Although the evaluation process was not optimal since we don't have a way to be confident for each block, these results are promising for larger datasets as we will be able to have greater accuracy. We see that extracting simple attributes like Streaks from a graph gives us a lot of data that can be processed and extract information that would else be impossible to know.

Parallel programming and the continuously increasing development of computing gives us ways to understand and observe more complex problems, and their solution will have a positive impact on our lives. More advanced techniques like machine learning algorithms and linear regression models will further increase the accuracy. In addition, minor optimizations can take place to eliminate noise in our output, like checking whether nodes of a likely blocked edge were active on other edges. In this way, we could assure that the node was active in the period of a negative streak and not in a social media abstention state. These are some options that we plan to embed in our technique in future work.