UNIVERSITÀ DEGLI STUDI DI VERONA

Master's degree in computer science and engineering / Big Data

Counting the number of triangles in a very large graph using the TTP algorithm

Elia Perantoni, VR472361
November 5, 2022

1 INTRODUCTION

A triangle $\Delta(u, v, x)$ in a undirected graph $G = (V, E)$ is a triple of nodes u, v, x such that all edges between them are in E. That is: $\{(u, v), (u, x), (v, x)\} \subseteq E$.

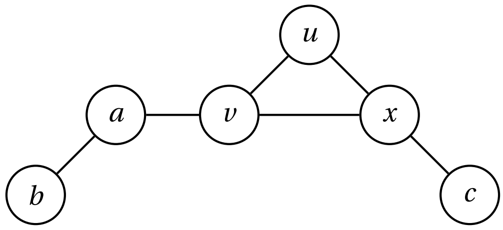


Figure 1: A triangle $\Delta(u, v, x)$ inside a larger graph

The problem of counting triangles in a graph inherits great importance from several interesting metrics on graphs that build upon it. The clustering coefficient, for instance, is a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that real world graphs, social networks in particular, tend to create tightly knit groups characterized by a relatively high density of ties. This metric, arguably a very insightful one, requires the number of triangles to be known, in order to compute it. Hence, its relevance.

In recent years, internal and external memory algorithms have been developed to solve the problem of triangle counting; but some of the graphs that we wish to analyze have become so large that they can no longer fit inside a single machine.

A popular technique for distributed computation is *Map-Reduce*: a *map* function is applied to each element in the input collection individually to produce key-value pairs, the shuffle phase then merges all values belonging to the same key and, finally, a *reduce* function is applied to each group of values (one for each key) to produce the final result. What makes Map-Reduce so powerful is the fact that data can be split in partitions to which the map and reduce functions can be applied independently. With this mechanism, processing enormous quantities of data becomes possible as the work can be distributed among a large number of machines.

This report comes with, and provides context to, a Jupyter notebook in which a recently developed Map-Reduce algorithm known as *TTP (Triangle Type Partition)* [1] is implemented, tested, analyzed and compared with one of its predecessor: *GP (Graph Partition)* [2].

We will first be introducing and discussing GP before moving to TTP because the two algorithms have many concepts in common but GP is easier to understand.

2 The tools

Arguably, the most known Map-Reduce implementation is *Apache Hadoop*, which also provides a distributed file system for data persistence: *HDFS (Hadoop Distributed File System)*. However, using Hadoop is somewhat tedious as the provided functionality is limited to plain Map-Reduce only. Also, HDFS comes with sup-optimal performance as secondary storage is used.

Apache Spark is a software based on Hadoop that provides the framework and unified set of APIs for performing a wide range of distributed computation operations that go beyond just simple Map-Reduce. Spark also improves on HDFS by using what's known as: *RDD (Resilient Distributed Dataset)*. It's a managed data structure that gets sharded and distributed (possibly with replication) among the machines belonging to the same cluster. What's different from HDFS is that RDDs' shards are stored in internal memory, thus noticeably improving the performance of read/write operations.

Our implementation is Apache Spark based and consumes its API via the Python SDK.

3 Graph Partition algorithm

In GP [2], the input graph $G = (V, E)$ is split into ρ partitions $(G_0, G_1, \ldots, G_{\rho-1})$ using a partitioning function P such that $\forall\, u \in V.\, P(u) \in [0, \rho-1]$. This gives $V = \bigcup_{i=0}^{\rho-1} V_i$ and $\forall_{i \neq j}.\, V_i \cap V_j = \emptyset$.

3 partitions G_i, G_j and G_k can be combined into one 3-partition G_{ijk} by taking the union of the nodes $V_{ijk} \triangleq V_i \cup V_j \cup V_k$ and inducing the edges from the original G. See fig. 3 for some examples.

The algorithm finds the triangles in every 3-partition G_{ijk} with $0 \le i < j < k \le \rho - 1$ and assigns a weight to each. The sum of weights across all partitions will give the correct number of triangles in G.

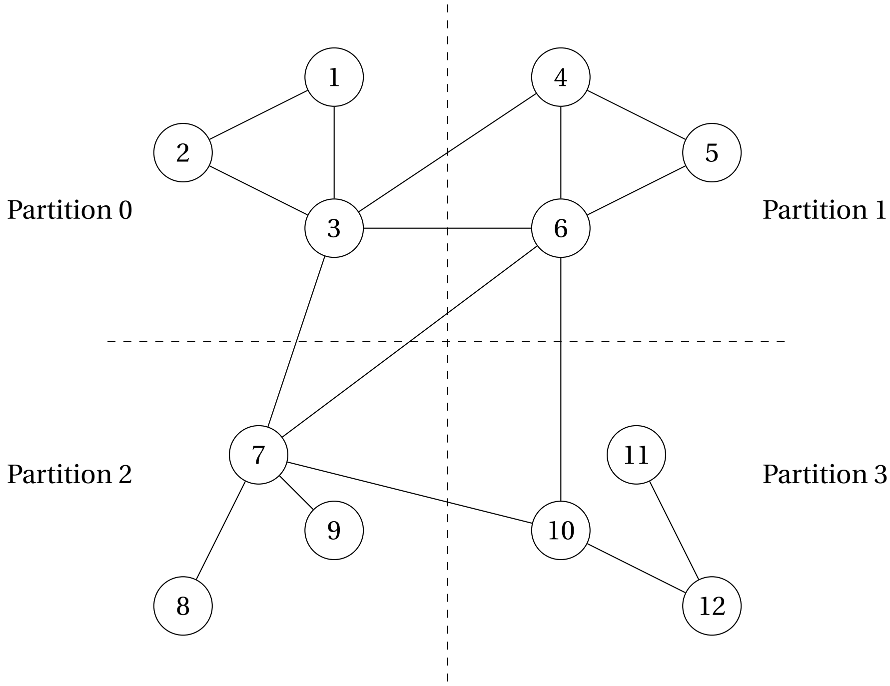


Figure 2: A simple graph divided into 4 partitions

Some triangles are seen in more than one 3-partition. Take a triangle $\Delta(u,v,x)$ entirely contained within partition $G_0 = (V_0, E_0)$ for instance, it will be observed repeatedly in any 3-partition G_{0jk} with $j,k \neq 0$. The purpose of the weight system then is to counteract this effect, ensuring that the sum of all weights emitted for a triangle is always 1.

The weight to emit for a triangle solely depends on the number of partitions that its nodes span:

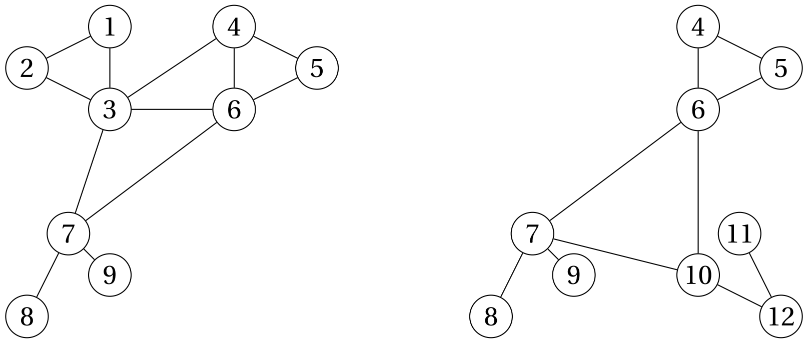
$$w(\Delta(u,v,x)) = \begin{cases} \frac{1}{\binom{\rho-1}{2}} & P(u) = P(v) = P(x) \\ \frac{1}{\rho-2} & P(u) = P(v) \vee P(v) = P(x) \vee P(u) = P(x) \\ 1 & \text{otherwise} \end{cases}$$

Proposition 1. *The sum of weights emitted by GP for any given triangle* $\Delta(u,v,x)$ *is* 1.

Proof. Trivially, exactly one of these must be true:

1. $\Delta(u,v,x)$ is entirely contained within a single partition G_A
2. $\Delta(u,v,x)$ is entirely contained within two partitions G_A and G_B
3. $\Delta(u,v,x)$ spans three partitions G_A, G_B and G_C with each node belonging to a different one

Therefore, we proceed with a proof by cases:



(a) G_{012} (b) G_{123}

Figure 3: Some 3-partitions from the graph in fig. 2

1. $\Delta(u, v, x)$ appears in any 3-partition G_{ijk} where either i, j or k is equal to A. Therefore, two variables remain free and their values are chosen between $\rho - 1$ possible partitions (all except A). Hence, $\Delta(u, v, x)$ appears in $\binom{\rho-1}{2}$ partitions. For each of those a weight equal to $\frac{1}{\binom{\rho-1}{2}}$ is emitted, giving $\binom{\rho-1}{2} * \frac{1}{\binom{\rho-1}{2}} = 1$.
2. $\Delta(u, v, x)$ appears in any 3-partition G_{ijk} with $\{i, j, k\} \subseteq \{A, B\}$. This means that one variable remains free and its value must be chosen among $\rho - 2$ possible partitions (all except A and B). Hence, $\Delta(u, v, x)$ appears in $\rho - 2$ partitions. For each of those a weight equal to $\frac{1}{\rho-2}$ is emitted, giving $(\rho - 2) * \frac{1}{\rho-2} = 1$.
3. $\Delta(u, v, x)$ appears just in G_{ABC}. The weight 1 will be emitted just once, trivially making the sum equal to 1.

$\square$

Since the sum of weights for any triangle amounts to exactly one, then the sum of all weights is exactly the number of triangles in the graph.

The *map* step consists of emitting an edge for every 3-partition that it belongs to. The *reduce* step is given a complete 3-partition and its job is that of enumerating the triangles and emitting a weight for each. A subsequent Map-Reduce pass is required to sum up the weights, giving the final result.

The distributed nature of GP emerges when observing that the actual triangle counting process can be performed on every 3-partition independently, using any internal memory algorithm.

Depending on the size of G, a sufficiently large number of partitions ρ will make sure that each is sufficiently small to fit inside a single machine.

The main issues with GP is that edges are emitted many times over, to account for all 3-partitions in which they appear, and triangles are processed redundantly. Take a triangle entirely contained within a single partition for instance, with $\rho = 10$. It will be processed in

$\binom{\rho-1}{2} = 36$ different 3-partitions. The improved algorithm, TTP, brings that number down to just 9.

4 TRIANGLE TYPE PARTITION ALGORITHM

The TTP algorithm [1] improves on GP by reducing the amount of redundancy and thus, the amount of data transferred in the cluster.

First, we formally classify triangles much in the same way as we did before for our proof:

Type I Is a triangle $\Delta(u, v, x)$ spanning 1 partition

Type II Is a triangle $\Delta(u, v, x)$ spanning 2 partitions

Type III Is a triangle $\Delta(u, v, x)$ spanning 3 partitions

For instance, in fig. 2 $\Delta(1,2,3)$ is a Type I triangle, $\Delta(3,4,6)$ is a Type II triangle and $\Delta(3,6,7)$ is a Type III triangle.

Then, edges are also classified as being either *inner* or *outer*. An edge (u, v) is said to be inner when u and v belong to the same partition. Otherwise, it's an outer edge.

A $3'$-partition (note the prime), denoted G'_{ijk}, is a 3-partition composed only of outer edges. TTP reduces redundancy by searching for Type III triangles in $3'$-partitions while and Type I and II triangles in 2-partitions. In the future, we may refer to 2-partitions and $3'$-partitions as *combinatorial partitions* to emphasize that they are a combination of multiple G_i ($i \in [0, \rho-1]$) partitions and to better distinguish them.

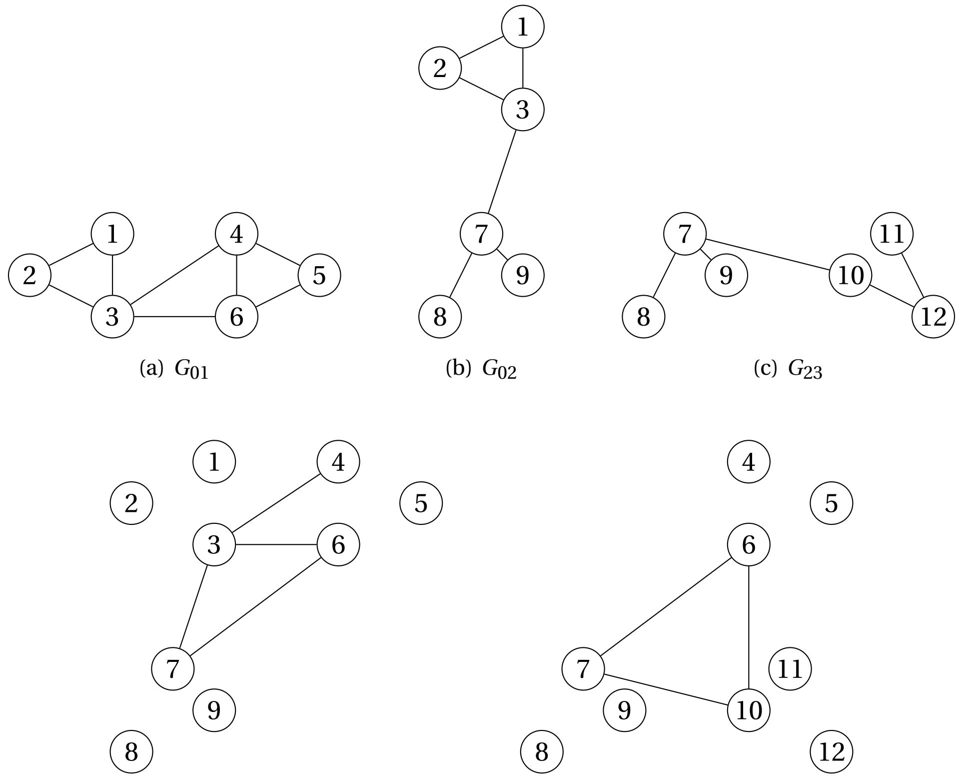
See fig. 4 for some examples of 2-partitions and $3'$-partitions.

The interesting observation to make here is that Type III triangles are entirely made up of outer edges and can therefore be correctly counted in $3'$-partitions that lack inner edges; thus making them smaller and quicker to process. Additionally, Type I and Type II have at least one inner edge, so they are not repeatedly processed in $3'$-partitions simply because they don't exist there. The number of $3'$-partitions partitions grows exponentially with ρ, so stripping them of inner edges and only processing Type III triangles there makes for a big performance improvement.

The overall structure of the TTP algorithm is similar to that of GP: the *map* step emits a pair (p, e) for every edge e that belongs to partition p, then the *reduce* step receives an entire partition (that might be a 2-partition or a $3'$-partition) and emits a weight for each triangle found there. The sum of all weights emitted gives the total number of triangles in the initial graph G.

In TTP, the weights are as follows:

$$w(\Delta(u, v, x)) = \begin{cases} \frac{1}{\rho-1} & P(u) = P(v) = P(x) \\ 1 & \text{otherwise} \end{cases} \tag{1}$$



(a) G_{01} (b) G_{02} (c) G_{23}

(d) G'_{012} (e) G'_{123}

Figure 4: Some examples of 2-partitions and $3'$-partitions from the graph in fig. 2

So Type I triangles get $\frac{1}{\rho-1}$ while Type II and Type III triangles get 1.

Proposition 2. *The sum of weights emitted by TTP for any given triangle* $\Delta(u,v,x)$ *is* 1.

Proof. Every triangle is either Type I, Type II or Type III. Let's proceed with a proof by cases.

Type I Suppose $\Delta(u,v,x)$ is entirely contained within partition G_A. Then, it appears in all 2-partitions G_{ij} with $i = A$ or $j = A$. It cannot appear in $3'$-partitions because a Type I triangle is entirely made out of inner edges, which $3'$-partitions lack. Therefore, there are $\rho - 1$ partitions in which the triangle appears. And every time that happens a weight equal to $\frac{1}{\rho-1}$ is emitted. Giving $(\rho-1) * \frac{1}{\rho-1} = 1$.

Type II Suppose $\Delta(u,v,x)$ is entirely contained within partition G_A and G_B (suppose also wlog that $A < B$). Then, it appears only in 2-partition G_{AB}. It cannot appear in $3'$-partitions because a Type II triangle has one inner edge, which $3'$-partitions lack. Therefore, there is just one partition in which the triangle appears. Since 1 is emitted just once, the sum is trivially equal to 1.

Type III Suppose $\Delta(u,v,x)$ spans three partitions G_A, G_B and G_C (suppose also wlog that

$A < B < C$). Then, it appears only in $3'$-partition G_{ABC}. It cannot appear in 2-partitions simply because the triangle is composed of nodes from three distinct partitions. Since 1 is emitted just once, the sum is trivially equal to 1.

□

5 Datasets

For this project, undirected graphs from the Standford Large Network Dataset Collection were used. In particular, the software is tested against these (in order of disk size):

1. ego-Facebook (1612010 triangles)
2. email-Enron (727044 triangles)
3. com-Amazon (667129 triangles)
4. roadNet-CA (120676 triangles)

To make the whole algorithm easier to understand and reason about, the same example from [1] is also used before getting to more complicated datasets.

Each dataset is given as a text file in the `datasets/` directory. Nodes are identified by means of a unique integer and for each edge $(u, v) \in E$ there is a line in the file with the node ids of u and v, separated by whitespace. *Edges are not reported twice*, meaning there is no `"v u"` line when `"u v"` is already there.

6 Implementation in depth

In this section, we are going to walk through the implementation provided along with this report as a Jupyter notebook. The same example graph used in [1] (also shown in fig. 2) will be used throughout.

First, we load the dataset from disk and perform some preprocessing to make everything easier.

```
edges = (sc
     .textFile(DATASET)
     .filter(bool)                    # Discard empty lines
     .map(lambda line: line.split())  # Split nodes in edge
     .map(lambda x: list(map(int, x))) # Parse node ids
     .map(tuple))                     # Convert list to tuple
```

The call to `textFile` returns an RDD whose elements are lines from the dataset's file. We filter out empty lines using the `bool` built-in function, then split each line by whitespace to get a pair of node ids, both of which are parsed to integers. Then the pair of nodes, thus far represented with a list of length 2, is converted to a tuple, just for personal preference.

If we were to collect `edges`, the result would look something like: `[(1, 2), (1, 3), (2, 3), (4, 5), (4, 6), ...]`. Each tuple in the list is a pair of node ids representing an undirected edge.

Next, we partition the graph by defining the number of partitions ρ and a partitioning function P. Note that the partitioning function initially used in the notebook only works for the example graph shown in fig. 2 and was chosen to mimic what's in [1]. A better, more general, partitioning function will be provided later:

```
RHO = 4
def P(node: int) -> int:
    return (node - 1) // (RHO - 1)
```

Now we need to build the 2-partitions and 3′-partitions that will be distributed among workers for independent triangle counting. This process works by producing a set $\{(p,e)\}$, where each element (p,e) is stating that edge e belongs to the combinatorial p, and grouping by p to obtain all the edges for that partition.

`edges_in_2p_map` and `edges_in_3p_map` are the mapping functions that take an edge $e \in E$ and map it to all (p,e) such that p is a 2-partition or a 3′-partition respectively, and e belongs to p:

```
def edges_in_2p_map(edge):
    u, v = edge
    res = []
    for a in range(0, RHO - 1):
        for b in range(a + 1, RHO):
            if P(u) in [a, b] and P(v) in [a, b]:
                res.append(((a, b), (u, v)))
    return res

def edges_in_3p_map(edge):
    u, v = edge
    if P(u) == P(v):
        return []
    res = []
    for a in range(0, RHO - 2):
        for b in range(a + 1, RHO - 1):
            for c in range(b + 1, RHO):
                if P(u) in [a, b, c] and P(v) in [a, b, c]:
                    res.append(((a, b, c), (u, v)))
    return res
```

Notice that, because inner edges never appear in 3′-partitions, if for an edge $e = (u, v)$ we have $P(u) = P(v)$ then `edges_in_3p_map` completely skips it.

We map all the edges in the graph using both functions in parallel and flatten the results. The flattening step is necessary because each edge produces not just one (p, e) pair but many of them instead.

We end up with two sets $\{(p, e)\}$: the first only contains pairs (p, e) where p is a 2-partition while the second only contains pairs (p, e) where p is a $3'$-partition. We take the union of those two sets:

```
edges_in_2p = edges.flatMap(edges_in_2p_map)
edges_in_3p = edges.flatMap(edges_in_3p_map)
edges_in_Xp = edges_in_2p.union(edges_in_3p)
```

If we now were to inspect `edges_in_Xp` we would see something like:

```
[((0, 1), (1, 2)),
 ((0, 2), (1, 2)),
 ((0, 3), (1, 2)),
 ((0, 1), (1, 3)),
 ((0, 2), (1, 3)),
 ...,
 ((0, 1, 2), (3, 6))
 ...,
]
```

Meaning that edge $(1,2)$ belongs to partitions G_{01}, G_{02} and G_{03}; edge $(1,3)$ belongs to partitions G_{01}, G_{02}, etc... But also, maybe more interestingly, edge $(3,6)$ belongs to partition G'_{012}.

Noticing that we now have a collection of pairs where the partition occupies the first position, we can `groupByKey` to collect all the edges belonging to the same partition:

```
# Merge together all the edges belonging to the same partition
edges_in_Xp_merged = edges_in_Xp.groupByKey()
```

Now we are finally ready to count the triangles in every partition, emitting a weight for each, and summing the results.

First, `assign_weight` is the function that takes a triangle and assigns a weight to it. It's exactly an implementation of eq. 1:

```
# Will be used later on to avoid floating point arithmetic errors.
from fractions import Fraction

def assign_weight(triangle):
    u, v, x = triangle
    if P(u) == P(v) == P(x):
        # If triangle is Type I, then the weight is 1/(RHO-1).
        return Fraction(1, RHO-1)
    else:
        # For all other triangles, 1 because they are observed only once.
        return Fraction(1, 1)
```

The usage of `Fraction`, instead of simply `float`, is due to the fact that floating pointer numbers do not have infinite precision and so slight errors can slowly creep up and lead to wrong results. Here's a toy example to demonstrate the fact (Python 3.10.8):

```
>>> 0.1 + 0.2
0.30000000000000004
```

`Fraction` is a great alternative when one wishes to store rational numbers with more precision because the numerator and denominator are stored separately as integers:

```
>>> Fraction(1, 10) + Fraction(2, 10)
Fraction(3, 10)
>>> float(Fraction(1, 10) + Fraction(2, 10))
0.3
```

Now getting back to our algorithm, recall that we have a collection of $(p, \{e\})$ elements. That is: for every combinatorial partition p we have a list of edges that belong to it. Now that we have grouped all edges belonging to the same p, we can actually strip that away because that's not needed for inducing a graph and counting triangles on it. Then we're left with a collection of edge-sets $\{e\}$, each of which can be processed independently by applying `total_partition_weight` to it:

```
weights = (edges_in_Xp_merged
    .map(lambda x: x[1])
    .map(total_partition_weight))
```

The job of `total_partition_weight` is that of inducing the subgraph of the combinatorial partition, using the provided edge-set, and emitting a weight for each triangle found there. Since this is performed on a single combinatorial partition, much smaller than the entire graph G, and on different workers, an internal memory algorithm for triangle enumeration can be used.

[1] cites many different internal memory algorithms in its "Related Works" section but it doesn't really matter which one is used. Since `total_partition_weight` is given a list of edges, we found the *edge-iterator algorithm* a pretty natural choice. It works by iterating through all edges and, for each $(u, v) \in E$, taking the intersection of the adjacency list of u and of v. For any node x in the intersection, a triangle $\Delta(u, v, x)$ is recorded. Notice that this algorithm observes each triangle three times. Therefore, proper measures should be in place to take this into account. We're simply relying on Python's sets to remove duplicate triangles.

```
def total_partition_weight(edges):
    # Given the list of edges, build an adjacency list.
    adj = dict()
    for edge in edges:
        u, v = edge
        adj[u] = adj.get(u, set()) | {v}
        adj[v] = adj.get(v, set()) | {u}
```

```python
    # Collect here the set of edges found in this partition.
    # Use a set to avoid duplication.
    triangles = set()
    for edge in edges:
        u, v = edge
        # Take the intersection of the adjacency list of u and v.
        # For each node x, a triangle (u,v,x) exists.
        triangles |= {frozenset((u,v,x)) for x in adj[u] & adj[v]}

    # For each triangle found in this partition, compute its weight.
    _sum = Fraction(0, 1)
    for triangle in triangles:
        _sum += assign_weight(triangle)

    # And return the sum of the weights.
    return _sum
```

We use the list of edges to build an adjacency list for every node, then we run the actual edge-iterator algorithm recording every triangle without duplication, and finally we compute the weight for each triangle found in this combinatorial partition and emit the partial sum.

`triangles` is a set of triangles but every triangle is, in turn, a set of nodes. This is because changing the order of nodes in a triangle does not make it any different. But standard sets cannot be added to other sets because they are not hashable; to overcome this problem `frozenset` must be used.

All that's left to do now is add all the partial sums coming the various partitions into one `Fraction` and converting that to an integer:

```python
# Sum the weight from every partition
round(weights.sum())
```

And sure enough, we get 5, which is what we expected for the graph in fig. 2.

As soon as we swap the simplified partitioning function P with a more general one such as this:

```python
def P(node: int) -> int:
    return node % RHO
```

Then we can start testing the implementation on more and larger graphs. We successfully conducted tests on all the datasets in sect. 5 and you can see the results in the attached notebook.

7 Impact of ρ

And now for some performance analysis: how does the number of partitions ρ affect the execution time and amount of data exchanged between workers in TTP? Given that the total number of combinatorial partitions is $\binom{\rho}{2} + \binom{\rho}{3}$, a lower value of ρ will result in significantly less combinatorial partitions to process (see fig. 5). But each combinatorial partition is now larger, potentially making it impossible for a combinatorial partition to fit inside a single machine anymore.

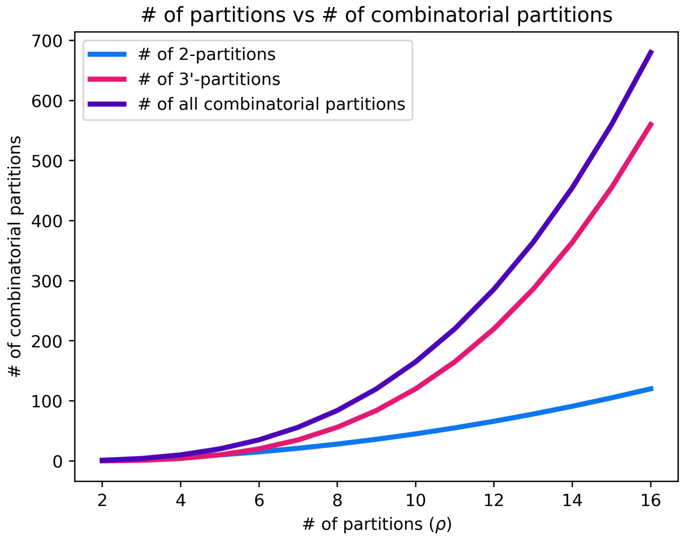
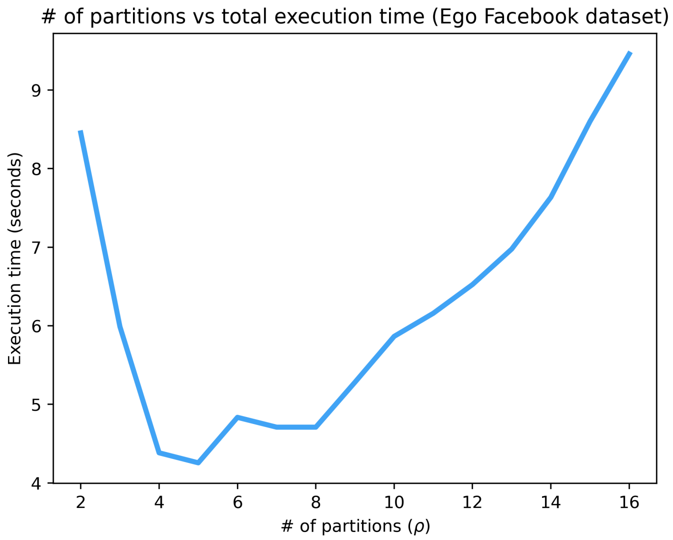


Figure 5: How ρ relates to the number of combinatorial partitions

Also, larger combinatorial partitions drive up the computational cost of the internal memory algorithm used. This suggests that a tradeoff is at play and one should not simply pick the lowest possible value of ρ that allows all combinatorial partitions to fit in the workers' memory.

To prove this point, we have benchmarked the algorithm with different values for ρ in $[2,16]$ on the Ego Facebook datasets (see sect. 5). The executions times are as observed by Python, since the Apache Spark admin console is only precise down to the second:

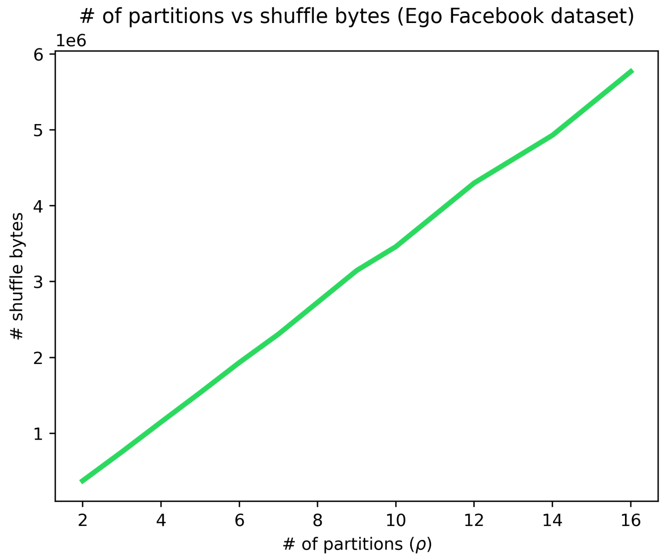


On my machine, the best tradeoff seems to be around $\rho = 4$ or $\rho = 5$ but this is definitely dependent on the number of machines in the cluster and their power. Point is: the lowest value for ρ is not necessarily the best.

Other factors that come into play are the amount of redundant processing of triangles and the amount of Type I & Type II triangles versus the amount of Type III triangles. We've seen that as ρ increases, the number of combinatorial partitions increases. Therefore, a Type I triangle $\Delta(u,v,x)$ in partition G_0, for instance, is repeatedly counted in all 2-partitions G_{0j} with $j \in [1,\rho-1]$ and the number of those clearly increases with ρ. On the other hand, a lower number of partitions means that more and more triangles become Type I or Type II. The great thing about Type III triangles is that they can be counted in stripped down subgraphs that are processed faster. Having less of them is detrimental to the performance of the algorithm.

As you can tell, there is quite a large number of effects going in opposite directions that depend on ρ, and predicting the best possible value is no easy task.

If we shift our focus from execution time to amount of data exchanged during the shuffle step, we see that it increases as ρ increases, as one would expect:



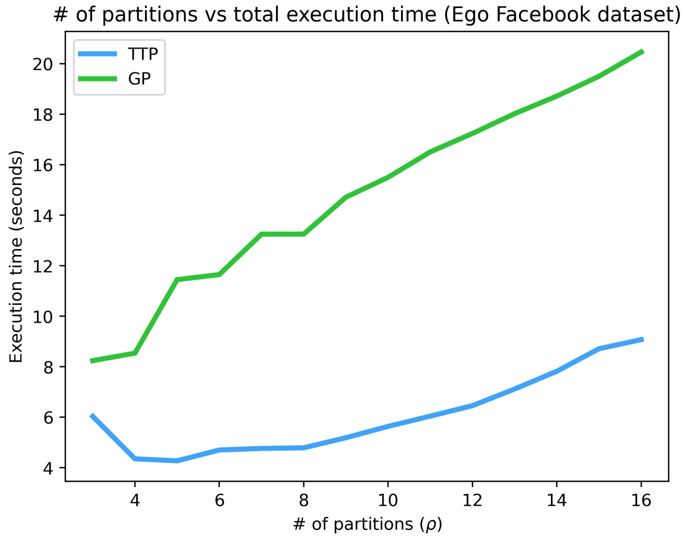
(The amount of data was scraped from Spark's admin console using Python packages `requests` and `beautifulsoup4`).

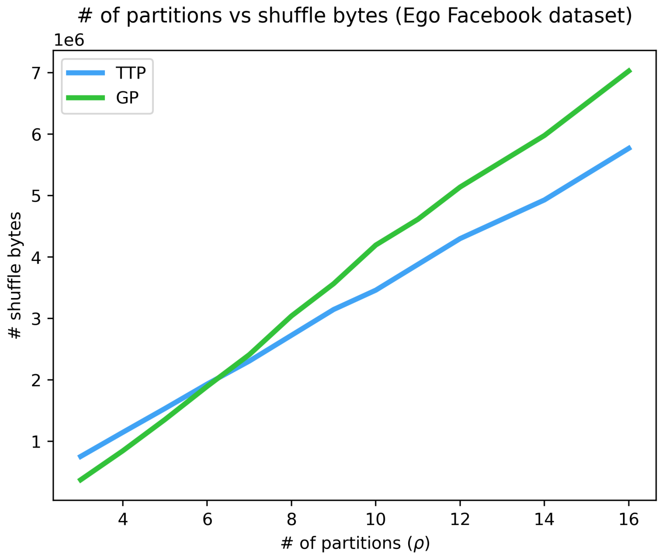
This can be explained by observing that a larger number of partitions ρ makes for a larger number of combinatorial partitions, and this in turn results in more duplication when emitting (p, e) pairs: imagine an edge (u, v) belonging to partition G_0, it is emitted in all G_{0j} with $j \in [1, \rho - 1]$ and in all G'_{0jk} with $j, k \in [1, \rho - 1]$. Obviously, with a larger ρ there are more G_{0j} and G'_{0jk} for which to emit an edge, driving up the amount of data exchanged.

8 Performance comparison TTP vs GP

We have covered the differences between the two algorithms before. The Jupyter notebook provides not just an implementation for TTP but also for GP.

Let's run them the two algorithms on the same dataset, and with various values for ρ, to see how they compare:





TTP consistently performs better than GP in terms of execution time. For a low number of partitions, GP needs to exchange slightly less data between workers. This is due to the fact that TTP emits not only 3-partitions but instead both 2-partitions and $3'$-partitions. This strategy might seem sub-optimal when one is working with a few partitions but proves to be very good when working with large graphs that need to be split in many partitions, which is the primary focus with this kind of algorithm anyway.

9 Conclusions

We started by explaining how the Graph Partition (GP) algorithm counts triangles in a graph by splitting the work across many, much smaller, 3-partitions. Then we moved to the Triangle Type Partition (TTP) and seen how it improves upon GP by classifying triangles and counting Type I and Type II triangles in different partitions than Type III to reduce redundant data flow and computation. We have tested the algorithm on many large graphs and experimented with different numbers of partitions to see how that affects performance. Finally, we have compared TTP against GP to prove that it actually performs better in practice.

References

[1] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 539–548, 2013.

[2] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614, 2011.