Graph Twiddling in Spark: Implementing MapReduce Algorithms for Network Analysis

Feiyi Xie
Electrical & Computer Engineering
Univiersity of Victoria
Victoria, Canada
feiyixie@uvic.ca

Mingxi Guo
Electrical & Computer Engineering
Univiersity of Victoria
Victoria, Canada
mingxiguo@uvic.ca

Weilong Qian
Electrical & Computer Engineering
Univiersity of Victoria
Victoria, Canada
wqian@uvic.ca

***Abstract*—Our project involves implementing a range of graph algorithms from the "Graph Twiddling in a MapReduce World" paper, utilizing Apache Spark for distributed computing. Our work encompasses multiple algorithms from the paper, highlighting Spark's resilience in executing intricate graph operations and offering valuable insights into distributed graph analytics.**

***Index Terms*—Graph Algorithms, Distributed Computing, MapReduce Model, Apache Spark, Network Analysis**

I. Introduction

The graph is a good representation of a collection of relationships as it naturally demonstrates network structure between social, traffic, etc.; with direction and weight, graphs can include sufficient details describing the information that's worth analyzing. However, when the graph size becomes very large, the memory and communication costs will become prohibitive physically and inevitably bring the challenge toward algorithmic complexity. To study and deal with this kind of situation, we need a better way of twiddling the data so that we can distribute the processing on the "cloud" and complete the task with a way better efficiency.

II. Graph Data Model and MapReduce

Graph data models are abstract representations of real-world entities and their interconnections. They are highly versatile, embodying relationships as edges and entities as vertices or nodes. This model is intuitive for representing networks like social interactions, molecular structures, or computer networks, making complex connections and interactions comprehensible and analyzable.

MapReduce is an effective programming framework for managing extensive data sets using a distributed, parallel approach across a computer cluster. It facilitates scalable and efficient analysis of vast and complex network datasets when applied to graphs. Decomposing graph operations into "map" (process and transform data) and "reduce" (aggregate results) phases allows for handling data-intensive tasks on graphs that are too large for traditional in-memory processing.

III. Our Work

We worked on a project that studied the graph algorithms discussed in the paper "Graph Twiddling in a MapReduce World" [1]. To implement the approach, we used PySpark on Colab. Additionally, we verified the algorithm's correctness by implementing it in NetworkX. For testing purposes, we used the well-known Zachary's karate club dataset.

The GitHub repo of our implementation can be found here: https://github.com/goosekeeper233/CSC-502-Project.

IV. Graph Algorithms using MapReduce

In this section, we will discuss the implementation of several graph algorithms from the target paper [1] using PySpark. These algorithms include Augmenting Edges with Degrees, Simplifying the Graph, Enumerating Triangles, Enumerating Rectangles, and Finding Trusses. We will briefly overview each algorithm and a portion of our implementation code.

A. Augmenting Edges with Degrees

The paper commences with a basic example of graph algorithms - Augmenting Edges with Degrees [1]. This function can be utilized as a helper function in the succeeding algorithms. As a result of this process, the records are enriched with vertex-degree information.

The process can be broken down as follows:

1) Mapper: Each edge in the graph is processed to emit two tuples, each with one of its vertex as key with a value of the edge itself.
2) Reducer 1: The vertices are then aggregated to sum their counts, which computes the degree of one of the vertex in the edge.
3) Reducer 2: The original edges are then combined with the computed degrees from the two vertices.

In Spark, this would typically implemented as the following steps:

```
def emit_edges(edge):
  src, des = edge
  return [(src, edge), (des, edge)]
map_1 = edge_rdd.flatMap(emit_edges)

def reducer_1(input):
  vertex, edges = input
  output = []
  length = len(edges)
  for edge in edges:
  if vertex == edge[0]:
    output.append( ((edge), (length, 0)) )
    else:
      output.append( ((edge), (0, length)) )
    return output

reduce_1 = map_1.groupByKey() \
                .mapValues(list) \
                .flatMap(reducer_1) \

def reducer_2(input):
    edge, degree_list = input
    d = [0, 0]
    for degree in degree_list:
    d[0] += degree[0]
    d[1] += degree[1]
    return (edge, tuple(d))

reduce_2 = reduce_1.groupByKey() \
                  .mapValues(list) \
                  .map(reducer_2)

return reduce_2
```

This approach leverages the distributed processing capabilities of Spark to handle large graph datasets efficiently, allowing for the scalable computation of vertex degrees and their association with edges.

B. Simplifying the Graph

For time & space efficiency, it is a good step to simplify the graph before we do the graph analysis; furthermore, it is also a necessary step for some algorithms which regard multiple edges as an indicator of connection strength.

When simplifying the graph, two cases need to be considered: whether the graph is an undirected graph or directed graph, and if the graph is a weighted graph or unweighted graph, and that would be 4 cases in total, such as:

1) Undirected unweighted graph: We may want to merge multiple edges that connect the same vertices to one representative edge.
2) Undirected weighted graph: We may want to sum up the weight for all multiple edges that connect the same vertices and merge that to one representative edge
3) Directed unweighted graph: We may want to leave the edge along since direction does grant it important information.
4) Directed weighted graph: We can sum up the weight for all multiple edges that connect the same vertices with the same direction and merge that to one representative edge.

Depending on the different requirements, the simplifying strategy should be different.

Since our karate club dataset has no direction or weight, we created a simple list to experiment with.

```
simple_list = [(6,7),(1, 2),
               (1, 2), (3, 4),
               (3, 4), (3, 4),
               (3, 3), (4, 5)]
edges_rdd_simplify = sc.parallelize(
                          simple_list);
```

We first want to remove the self-loop. As the paper suggested, if the pair has a repeating value, we need to drop it by creating an empty list.

```
removed_loop_rdd = edge_rdd.flatMap(
                      lambda edge: [edge] \
                      if edge[0] != edge[1] \
                      else [])
```

Then, we have to put the same edges into the same bin. Author [1] suggested we use hash; however, in PySpark, the default partitioner is hash-partition, so we don't need to do that explicitly [2].

```
binned_rdd = removed_loop_rdd.map(
               lambda edge: (
                 tuple(sorted(edge)),1))
```

Then, we reduce the same edges using the `reducebykey` function.

```
edges_grouped = binned_rdd.reduceByKey(
                    lambda a, b: a+b)
simplified_edge_rdd = edges_grouped.map(
                          lambda x: x[0])
```

The result is the same as the NetworkX function.

```
[(1, 2), (3, 4), (4, 5), (6, 7)]
[(1, 2), (3, 4), (4, 5), (6, 7)]
The outputs are the same: True
```

C. Enumerating Triangles

When we analyse a graph, the triangle plays a very important role as it's a fundamental structure in the graph. However, when we are dealing with a massive dataset, an improper way of searching for triangles will tremendously affect efficiency. To deal with this scenario, we must enumerate triangles following these steps.

Firstly, we want to find every open triad. An open triad is a structure having two edges connected to a shared vertex, or it can be considered a triangle "missing" a third edge. Once we found all open triads and their third closing edge, we found all triangles. By doing this, we optimized our problem from finding all triangles to finding all open triads.

In our `enumerating_triangle`, we take augmented edges as input and group all edges under the lowest vertex the edge has. Setting the lowest vertex as the key guaranteed there would be the least number of pairs being processed in each bin being emitted to the reducer as the lowest vertex means it connects to the least number of edges. However, as the number of triangles in a graph is fixed, we are going to have more reducers to do the job. This step is a good trade-off between reducer sizes and communication.

Then we ignore all the solo edges left (not an open triad) and find all the possible third closing edges:

```
def enumerating_triangle(edges_rdd):
lowest_degree_rdd =
  edges_rdd.map(lambda edge: (
      edge[0][0] \
      if edge[1][0] <= edge[1][1] \
      else edge[0][1],edge[0])) \
    .groupByKey() \
    .filter(lambda x:len(list(x[1])) > 1) \
    .flatMap(edge_wanted)
```

This function uses double-for-loop to search for all possible third-closing edges for every open triad. At this point, we only list all the closing edges; we are curious if they exist. The output is a key value pair taken sorted, with the wanted-third closing edge as the key.

```
def edge_wanted(grouped):
key, edges = grouped
edges_list = list(edges)
pairs = []
for i in range(len(edges_list)):
  for j in range(i + 1, len(edges_list)):
    sorted_pair \
      = tuple(sorted((
          edges_list[i][0] \
          if edges_list[i][0] != key \
          else edges_list[i][1],
          edges_list[j][0] \
          if edges_list[j][0] != key \
          else edges_list[j][1])))
    #sorted_pair = tuple(
      sorted(
        [edges_list[i][0],
        edges_list[j][0]]))
    pairs.append(((
      sorted_pair,(edges_list[i],
                   edges_list[j]))))
return pairs
```

Ultimately, we sort all our edges to remain consistent with previous work and then use join to find if a close triad exists.

```
third_edge_rdd = edges_rdd.map(
  lambda edge: (
    (edge[0][0], edge[0][1]) \
    if edge[0][0] <= edge[0][1] \
    else (edge[0][1], edge[0][0]),
    edge[0]))
close_triad_rdd = lowest_degree_rdd \
  .join(third_edge_rdd) \
  .map(lambda x: (
    x[1][0][0], x[1][0][1], x[1][1]))
return close_triad_rdd
```

The result is the same as NetworkX's function, 45.

```
[((0, 4), (4, 6), (0, 6)),
  ((0, 5), (5, 6), (0, 6)),
  ((0, 4), (4, 10), (0, 10)),
  ((0, 8),.....
45
```

D. Enumerating Rectangles

Enumerating rectangles is similar to enumerating triangles. To create a rectangle, we can combine two open triads with the same opening edge. However, unlike triangles, the permutations of the vertices in rectangles can be more complex.

Let us take the example of ordered vertices. For a triangle, there is only one permutation - for any vertex, the other two vertices are always neighbouring. However, for any four ordered vertices forming a rectangle, there are three permutations. Each permutation has a different vertex, not neighbouring a particular vertex. In order to define these permutations, the paper suggests that we classify a triad connecting two vertices with a higher order as a "low triad" and a triad connecting one vertex with a higher order and another with a lower order as a "mixed triad."

We can see from Fig. 1 that all three permutations can be found as any pairs formed by a low triad (labelled in blue) or mixed triad (labelled in orange) [1].

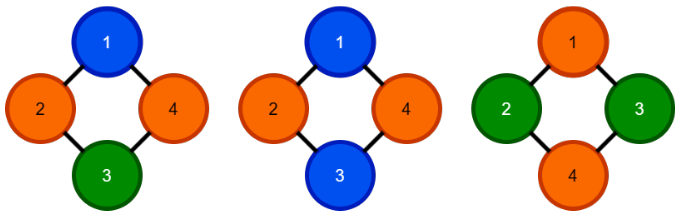

Fig. 1. Low and Mixed Triads in Rectangles [1]

Using these findings, we can design our MapReduce algorithm in three steps, as suggested in the paper [1]:

Step 1 Bin every edge by its high and low vertex, marking each output record as high or low, producing a binned edge file.

```
def transform_edge(edge):
((src, dst), (src_deg, dst_deg)) = edge
if src_deg > dst_deg:
    return [(src,
      ((src, dst), 'H')),
      (dst, ((src, dst), 'L'))]
else:
    return [(src,
      ((src, dst), 'L')),
      (dst, ((src, dst), 'H'))]
labeled_edges_rdd = augmented_edges_rdd \
  .flatMap(transform_edge)
grouped_labeled_edges_rdd = \
  labeled_edges_rdd.groupByKey()
```

Step 2 Go through each bin in the binned edge file, exporting every pair of distinct low records and every pair of a low record mixed with a high record in the bin to produce a triad file. Each record in the triad file is binned by the pair of vertices the triad connects, with the pair being ordered lexicographically.

```
def process_bin(bin_data):
    common_vertex, edges = bin_data
    lows = []
    highs = []
    for edge in edges:
        (src, dst), label = edge
        if label == 'L':
            lows.append((src, dst))
        else:
            highs.append((src, dst))
    # Generate pairs of low-low
    # and low-high,
    # excluding the common vertex
    triads = []
    for low in lows:
            other_vertex_low = low[1] \
              if low[0] == common_vertex \
              else low[0] \
            # Low-Low pairs
            for other_low in lows:
                if low != other_low:
                    other_vertex_low_2 \
                      = other_low[1] \
                        if other_low[0] \
                          == common_vertex \
                        else other_low[0]
                    triad_key = tuple(
                      sorted(
                        [other_vertex_low,
                        other_vertex_low_2]))
                    triads.append(
                      (triad_key,
                      (low, other_low)))
            # Low-High pairs
            for high in highs:
                other_vertex_high = high[1] \
                  if high[0] == common_vertex \
                  else high[0]
                triad_key = tuple(
                  sorted([other_vertex_low,
                          other_vertex_high]))
                triads.append((triad_key,
                               (low, high)))
        return triads
triad_edges_rdd = \
  grouped_labeled_edges_rdd \
  .flatMap(process_bin)
triad_grouped_rdd = \
  triad_edges_rdd.groupByKey() \
  .mapValues(list)
```

Step 3 Go through each bin in the triad file bin, exporting a rectangle for every triad pair.

```
def find_rectangles(bin_data):
    _, triads = bin_data
    rectangles = []
    # Remove duplicate triads
    unique_triads = list(set(triads))
    # Compare each pair of unique triads
    # to find rectangles
    for i in range(len(unique_triads)):
        for j in range(i + 1,
          len(unique_triads)):
            triad1 = unique_triads[i]
            triad2 = unique_triads[j]
            # Combine edges of both triads
            # and find all unique vertices
            all_edges = triad1 + triad2
            all_vertices = set(
              [vertex for edge \
                in all_edges for vertex \
                in edge])
            # A rectangle is formed
            # if there are exactly
            # 4 unique vertices
            if len(all_vertices) == 4:
                # Sort edges within
                # the rectangle
                # to remove redundancy
                sorted_rectangle = tuple(
                  sorted([edge for edge
                    in all_edges]))
                # don't really need to sort
                # vertices in edges since
                # we assume it's pre-sorted
                # in this project and we
                # keep this intact
                rectangles.append(
                  sorted_rectangle)
    return rectangles
# Map each bin through the
# find_rectangles function
flattened_rectangles_rdd \
  = triad_grouped_rdd \
  .flatMap(find_rectangles).distinct()
```

E. Finding Trusses

Trusses refer to sections of a graph that exhibit a greater degree of interconnectivity. They constitute a subset of the graph where each edge is integrated into a minimum number of triangles. Locating k-trusses within a graph is vital in comprehending the clustering and community structure. A k-truss is a subgraph in which each edge is present in at least k-2 triangles within the subgraph. Identifying these k-trusses is especially valuable in social network analysis, as they can reveal groups with strong ties.

Firstly, we need a list of all triangles in the graph. Then, we count how many triangles each edge is part of. Edges in fewer than k-2 triangles are not part of a k-truss and are, therefore, removed. This process repeats iteratively until no more edges can be removed.

Here is a step-by-step breakdown with code:

Step 1 Emitting Edge Triad Information: Every triangle in the graph will be broken down into its constituent edges, and each edge will be emitted with information about the triangle it is part of. This step is crucial for counting the number of triangles each edge participates in.

```
def emit_triad_edges(triangle):
    # Emit edges from a triangle with each \
    # edge tagged by the opposite vertex.
    for i in range(3):
        for j in range(i+1, 3):
            # Emit the edge with the third
            # vertex as a tag
            yield ((triangle[i],
              triangle[j]),
              triangle[3 - i - j])
```

Step 2 Counting Triangles for Each Edge: By using the countByKey() action, we count the occurrences of each edge across all the triangles. This gives us the number of triangles to which each edge belongs.

```
# Assumes 'triangles_rdd' is an RDD of
# tuples representing triangles
triangle_counts_rdd = triangles_rdd \
  .flatMap(emit_triad_edges).countByKey()
```

Step 3 Filtering for Truss Membership: In this step, edges not part of at least k-2 triangles are filtered out. The remaining edges constitute the k-truss.

```
def find_trusses(triangle_counts, k):
  """Filter edges to find the k-truss."""
  # Filter out edges that are not part of
  # at least k-2 triangles
  return [edge for edge, count \
    in triangle_counts \
    .items() if count >= k-2]
```

Step 4 Iterating to Convergence: The truss finding process is iterative. The edges that do not satisfy the k-truss condition are pruned in each iteration. The remaining graph is then used for the next iteration. This process repeats until no more edges can be pruned.

V. Conclusion

Our project aimed to implement and analyze a series of graph algorithms described in the ”Graph Twiddling in a MapReduce World” paper [1] using Apache Spark’s distributed computing capabilities. We successfully translated the theoretical MapReduce algorithms into practical Spark implementations and demonstrated their ability to process large-scale graph data efficiently.

To ensure the correctness of our implementations, we conducted a comparative analysis with similar algorithms in NetworkX, a Python-based graph processing library. Although NetworkX operates in a single-machine, C-based context, while Spark utilizes a distributed, JVM-based framework, our MapReduce algorithms consistently produced accurate results.

Comparing the performance of these two platforms was challenging due to their differences in operational paradigms and underlying technologies. However, our project highlighted the intuitive advantages of the MapReduce model in Spark, specifically its scalability and ability to handle vast datasets, which NetworkX, single-threaded, could not match.

The MapReduce model excels at processing large graphs by breaking down complex operations into smaller, manageable tasks, optimizing resource usage, and reducing computation time. Our project reinforced MapReduce’s theoretical benefits and provided practical insights into its application in modern distributed computing environments.

In conclusion, our project bridged the gap between theoretical algorithms and their practical implementation and demonstrated the comparative strengths of distributed versus single-node graph processing. It also opens the door for future research in optimizing graph algorithms for distributed systems and exploring the full potential of big data technologies in graph analytics.

References

[1] J. Cohen, “Graph Twiddling in a MapReduce World,” in Computing in Science & Engineering, vol. 11, no. 4, pp. 29-41, July-Aug. 2009.
[2] “Pyspark.rdd.reducebykey,” pyspark.RDD.reduceByKey - PySpark master documentation, https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.reduceByKey.html (accessed Apr. 7, 2024).