| | |
|---|---|
| Distributed Systems | Date: *23th April 2020* |
| Instructor: *Lini Thomas* | Author: *Shivam Singh* |

# Enumerative Algorithm for 4-cycles using GoFFish

Project Report : Shivam Singh (2018201015)

## 1 Introduction

With the ever-growing size of data, the problem of processing large datasets that do not fit in a single machine has become important. Graphs constitutes a particularly challenging class of data due to their irregular structure and asymmetric execution of graph algorithms.
Google introduced a vertex-centric distributed graph processing model called Pregel for processing large scale graphs on commodity clusters. Other frameworks have extend this to a coarser component-centric programming model, like GoFFish (Sub-graph-centric) including others.
These have exhibited better performance than former by reducing the number of Super-Steps and messages exchanged between workers.

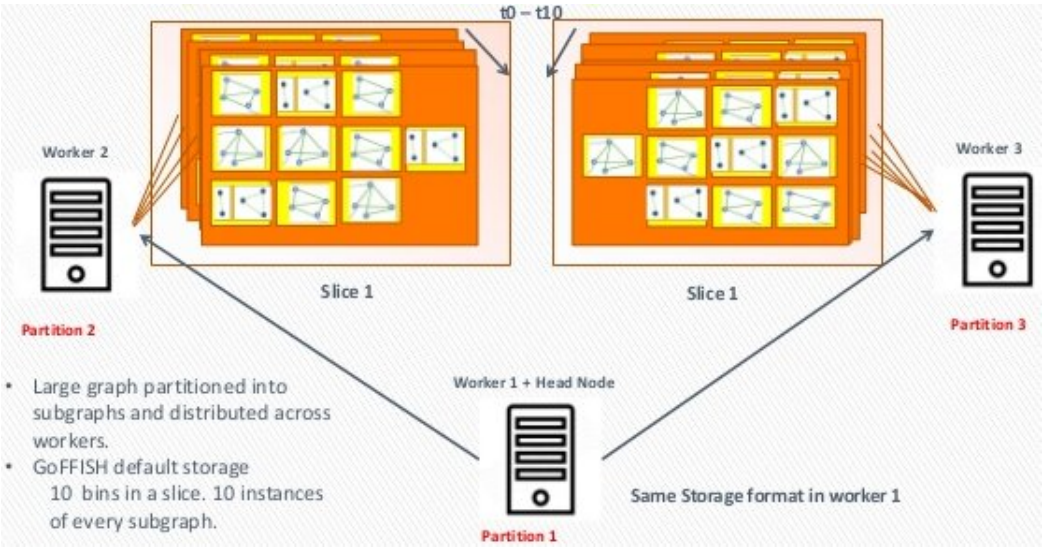
## 2 GoFFish

### 2.1 Overview

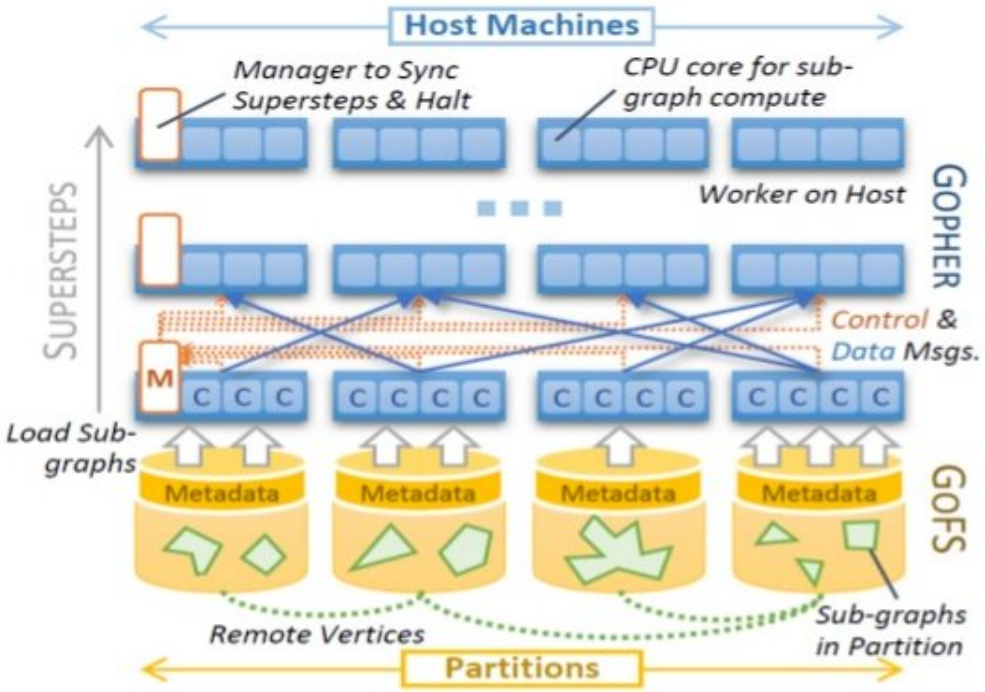
GoFFish is a scalable software framework for storing graphs, and composing and executing graph analytics in a Cloud and commodity cluster environment.
It consists of :-

- **GoFS** - It is a distributed store for partitioning, storing and accessing graph datasets across hosts in a cluster.
- **Gopher** - Gopher is a programming framework that offers sub-graph centric abstractions on a cloud or cluster in conjunction with GoFS.

GoFFish is implemented in Java using Maven build tool. GoFFish's Storage Architecture is shown below :



And the Algorithm workflow is as shown below:




So Sub-graph-Centric models helps in reduced number of Super-Steps which results in better performing algorithms.

## 2.2 Sub-graph Centric Programming Model

- **Algorithm** : Sequence of Super-Steps seperated by global barrier synchronization.
- **Super-Step i**
  1. Sub-graphs compute in parallel.
  2. Receive messages sent to it in Super-Step (i-1).
  3. Execute same user defined function.
  4. Send messages to other Sub-graphs (to be received in Super-Step (i+1)).
  5. Can vote to halt: I'm done or Deactivate.
- Global **Vote to Halt** check
  - Termination - All Sub-graphs voted to halt
- Active Sub-graphs participate in every computation.
- Deactivated Sub-graphs will not get executed/ activated unless it get new messages.

## 2.3 Programming Abstractions Provided by GoFFish

- **Compute** - User specified function to be run on each Sub-graph, for each Super-Step.
- **Send** - Sends a message to a remote Sub-graph, or vertex in a Sub-graph.
- **SendToAll** - Broadcasts a message to all Sub-graphs in the graph.

- **SendToMaster** - Sends message to the master Sub-graph.
- **VoteToHalt** - Sub-graph votes to halt. Application halts when all Sub-graphs halt and no messages were sent.

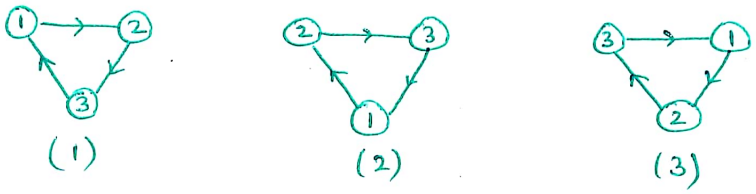
# 3 Triangle Counting Using GoFFish

A triangle in a graph is a 3-clique, that is, a set of three vertices where each pair of vertices is connected. Triangle counting is the problem of counting all triangles in a graph. The application of triangle counting includes finding clustering coefficient, transitivity, and community detection.

## 3.1 Pre-Concepts

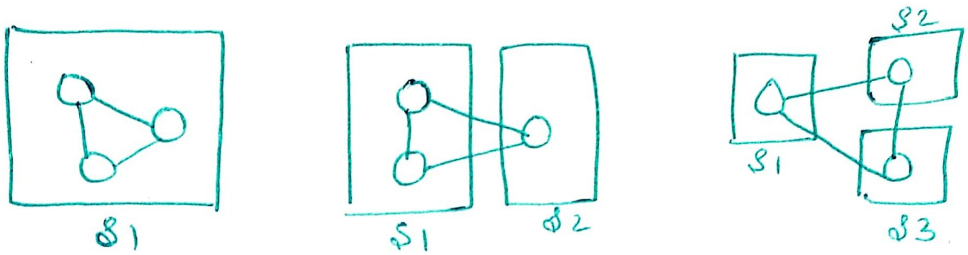
Before jumping to algorithm, first we can look to below mentioned observations:

- We want our algorithm to be correct i.e. we don't want to skip any triangle as well don't want to repeat any triangle in our count. Below mentioned pic shows same triangle in three ways:




  So no matter which vertex we start on we should be counting each triangle once. For that what we can do is go only that vertex whose id is strictly greater than the current one. i.e. above shown triangle can be written uniquely as 1-2-3-1.

- Second point we can notice is if our whole graph is partitioned into many Sub-graphs then only 3 types of triangles can be formed as shown below :




  1. All three vertices lie in the same Sub-graph.
  2. Two of its vertices lie in one Sub-graph, while the third is in a different Sub-graph, and
  3. All three vertices lie in different Sub-Graphs.

  Number of Super-Steps depends on how many times we jump to different Sub-graphs i.e. in 1 we'll need 0 Super-Steps, in 2 we'll need 1 Super-Step and in 3 we'll need 2 Super-Steps.

### 3.2 Algorithm

The Sub-graph centric triangle counting algorithm can be stated as follows:

- In the first Super-Step, it begins with finding and printing triangles whose vertices (v, w, u) are fully internal to this Sub-graph, or with just one vertex (u) in a remote Sub-graph [lines 3-11].
- It then sends messages to vertices in neighboring Sub-graphs whose IDs (w) are larger than the source vertex (v) in this Sub-graph [lines 12-16].
- In the second Super-Step, the received message with a pair of vertex IDs (v, w) is forwarded to a neighboring Sub-graph connected to the local vertex (w) if the remote Sub-graph's vertex ID (u) is higher than the source vertex (w) [lines 17-23].
- In the last Super-Step, we receive vertex IDs for a triple (v, w, u) and test if u is connected to v, and if so, print it as a Type (iii) triangle [lines 24-31].

Note that in both Super-Steps 2 and 3, one may end up in not finding a Type (iii) triangle if the vertex ID comparisons fail (w.id ¡ v.id, u.id ¡ w.id), or the first vertex is not connected to the third ($v.id \notin u.adjList$).

### 3.3 Pseudo-Code

```
1: procedure COMPUTE(Subgraph s, Message[] msgs)
2:     if superstep = 0 then                ▶ Find Types (i),(ii) triangles in subgraph
3:         for v ∈ s.vertices do
4:             for w ∈ v.adjList | !w.isRemote & w.id > v.id do
5:                 for u ∈ w.adjList | u.id > w.id do
6:                     if u ∈ v.adjList then          ▶ v, w, u form triangle
7:                         PRINT(v.id, w.id, u.id)
8:                     end if
9:                 end for
10:            end for
11:        end for
12:        for v ∈ s.vertices do
13:            for w ∈ v.adjList | w.isRemote & w.id > v.id do
    ▶ Potential Type (iii). Message subgraph 2.
14:                SEND(w.subgraphId, ⟨v.id, w.id⟩)
15:            end for
16:        end for
17:    else if superstep = 1 then
18:        for ⟨vId, wId⟩ ∈ msgs do
19:            w = s.getVertex(wId)
20:            for u ∈ w.adjList | u.isRemote & u.id > w.id do
    ▶ Potential Type (iii). Message subgraph 3.
21:                SEND(u.subgraphId, ⟨vId, wId, u.id⟩)
22:            end for
23:        end for
24:    else if superstep = 2 then
25:        for ⟨vId, wId, uId⟩ ∈ msgs do
26:            u = s.getVertex(uId)
27:            if vId ∈ u.adjList then          ▶ Found Type (iii) triangle
28:                PRINT(vId, wId, uId)
29:            end if
30:        end for
31:    end if
32: end procedure
```

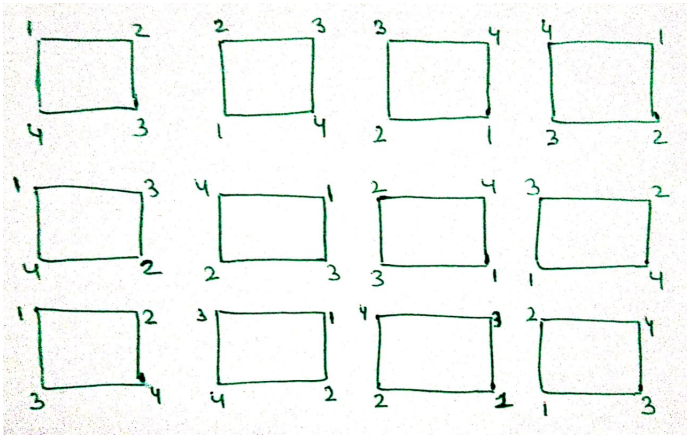
# 4 Counting 4-Cycles Using GoFFish

A 4-cycle or quadrilateral in a graph is a cycle containing 4 edges and vertices in which neither of them repeat. 4-cycle counting is the problem of counting all quadrilaterals in a graph. The application of 4-cycles counting includes finding clustering coefficient, transitivity, social network and community detection.

## 4.1 Pre-Concepts

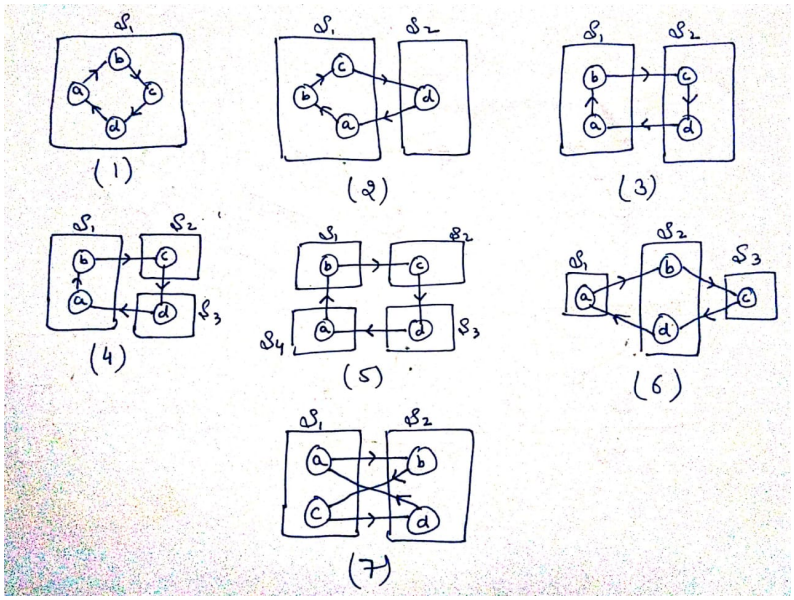
Before jumping to algorithm, first we can look to below mentioned observations:

- Similar to Triangle formation, there are many ways in which a 4-cycle can be formed:



Each row in above picture represents same 4-cycle but each column represents a different one. So like triangle counting we can't go with the selection of vertices in increasing order as then we'll be skipping many 4-cycles.

- Second point we can notice is if our whole graph is partitioned into many Sub-graphs then 7 types of 4-cycles can be formed which are shown below :



1. All four vertices lie in the same Sub-graph.
2. Fourth vertex lie in different Sub-graph.
3. First two vertices in one Sub-graph and other two in another Sub-graph.
4. First two vertices in same Sub-graph, rest two in another Sub-graphs that too in different ones.
5. All four vertices lie in different Sub-graphs.
6. 4-cycle is divided in 3 Sub-graphs, i.e. one of the pairs of diagonal vertices lie in same Sub-graph.
7. Both diagonal pair vertices lie in same Sub-Graph but mutually different.

Number of Super-Steps depends on how many times we jump to different Sub-graphs i.e. in 1 we'll need 0 Super-Steps, in 2,3 we'll need 1 Super-Step, in 4 we'll need 2 Super-Steps and in 5,6,7 we'll need 3 Super-Steps.

## 4.2 Assumptions

- The graph we are working on is a directed one i.e. direction of edges matters and they are not bi-directional.
- While counting 4-cycles, 2 4-cycles are assumed different if first one contains any edge which is not present in second one. eg. 1-2-3-4-1 and 1-3-2-4-1 are two different 4-cycles even if combination of vertices are same in both.
- Each vertex has an adjacency list of its neighboring vertices (adjList) that can be looked up in constant time.

## 4.3 Algorithm

The Sub-graph centric 4-cycle counting algorithm can be stated as follows:

- In the first Super-Step, it begins with finding and printing 4-cycles whose vertices (u, v, w, x) are fully internal to this Sub-graph.
- It then sends messages to vertices in neighboring Sub-graphs with a defined symbol (typeOfMessage), -1 for Type (ii) 4-cycles, -2 for Type (iii) and (iv) 4-cycles and -3 for Type (v), (vi) and (vii) 4-cycles (depending on number of vertices we have found till now for these type of 4-cycles). The message contains vertex (u) and the Sub-graph Id where (u) is.
- In the second Super-Step, the received message with the vertex ID and symbol we can count 4-cycles for Type (ii) and (iii). For others we'll find next vertex and process as step 2, i.e. sending message to neighbouring vertices. This time -1 is for Type (iv) 4-cycle, -2 for Type (v) and (vi) 4-cycles, and -3 for Type (vii) 4-cycle. This time the message contains vertex (u) and previous two Sub-graph Ids which have been visited since we don't want to repeat vertex and want the cycles to be as the Type has been decided.
- In the third Super-Step, we can have the count for Type (iv) 4-cycles. For the remaining types this will be the last vertex so we don't need any different type of messages, we just want to check is there an edge from (x) to (u). So this time message will only contain (u).
- In the last Super-Step, we receive vertex ID (u) and test if there is an edge from x to u, and if so, count it as a 4-cycle.

Note that in Last Super-Step we are not sending any typeOfMessage symbol so we won't be able to categorize it in different types of 4-cycles, but if we want to do so we can include typeOfMessage symbol in that too.

## 4.4 Code Snippets

Below are attached some code snippets from original code to introduce how the GoFFish API functions can be used in above mentioned algorithm (Type number is code is same as mentioned above in images):

- **Code Snippet for Super-Step 0, here we count all Type-1 4-cycles and send messages for other types.**

```
// Counting Type1 4-cycles where all are local vertices
for (IVertex<LongWritable, LongWritable, LongWritable, LongWritable> firstVertex : getSubgraph().getLocalVertices()) {
    for (IEdge<LongWritable, LongWritable, LongWritable> firstEdge : firstVertex.getOutEdges()) {
        IVertex<LongWritable, LongWritable, LongWritable, LongWritable> secondVertex = getSubgraph().getVertexById(firstEdge.getSinkVertexId());
        if(secondVertex.isRemote())
        {   // For Type - 5,6,7 4-cycles
            @SuppressWarnings("unchecked")
            long nextSGId = ((IRemoteVertex<LongWritable, LongWritable, LongWritable, LongWritable, LongWritable>) secondVertex).getSubgraphId().get();
            long currentSGId = getSubgraph().getSubgraphId().get();
            step1Zip(outputMessages, nextSGId, currentSGId, secondVertex.getVertexId().get(),firstVertex.getVertexId().get(), -3);
            continue;
        }
        if(secondVertex.getVertexId().get() == firstVertex.getVertexId().get())
            continue;
        for (IEdge<LongWritable, LongWritable, LongWritable> secondEdge : secondVertex.getOutEdges()) {
            IVertex<LongWritable, LongWritable, LongWritable, LongWritable> thirdVertex = getSubgraph().getVertexById(secondEdge.getSinkVertexId());
            if(thirdVertex.isRemote())
            {   // For Type - 3,4 4-cycles
                @SuppressWarnings("unchecked")
                long nextSGId = ((IRemoteVertex<LongWritable, LongWritable, LongWritable, LongWritable, LongWritable>) thirdVertex).getSubgraphId().get();
                long currentSGId = getSubgraph().getSubgraphId().get();
                step1Zip(outputMessages, nextSGId, currentSGId, thirdVertex.getVertexId().get(),firstVertex.getVertexId().get(), -2);
                continue;
            }
            if(thirdVertex.getVertexId().get() == firstVertex.getVertexId().get() || thirdVertex.getVertexId().get() == secondVertex.getVertexId().get())
                continue;
            for (IEdge<LongWritable, LongWritable, LongWritable> thirdEdge : thirdVertex.getOutEdges()) {
                IVertex<LongWritable, LongWritable, LongWritable, LongWritable> fourthVertex = getSubgraph().getVertexById(thirdEdge.getSinkVertexId());
                if(fourthVertex.isRemote())
                {   // For Type - 2 4-cycle
                    @SuppressWarnings("unchecked")
                    long nextSGId = ((IRemoteVertex<LongWritable, LongWritable, LongWritable, LongWritable, LongWritable>) fourthVertex).getSubgraphId().get();
                    long currentSGId = getSubgraph().getSubgraphId().get();
                    step1Zip(outputMessages, nextSGId, currentSGId, fourthVertex.getVertexId().get(),firstVertex.getVertexId().get(), -1);
                    continue;
                }
                if(fourthVertex.getVertexId().get() == secondVertex.getVertexId().get() || thirdVertex.getVertexId().get() == fourthVertex.getVertexId().get() |
                    continue;
                // Found all 4 vertices of 4-cycle in local Sub-graph, so increasing count
                if (fourthVertex.getOutEdge(firstVertex.getVertexId()) != null)
                    type1 += 0.25;
```

- **Code Snippet for last Super-Step i.e. 3, here we counted all Types of 4-cycles and program votes to halt.**

```
for (Map.Entry<Long, List<Long>> secondToFirst : transitionIdsMap.entrySet()) {
    IVertex<LongWritable, LongWritable, LongWritable, LongWritable> firstVertex = getSubgraph().getVertexById(new LongWritable(secondToFirst.getKey()));
    List<Long> firstList = secondToFirst.getValue();
    for(int i = 0; i < firstList.size(); i+=2)
    {   if(firstList.get(i) == -1)
        {   // For Type-5, 4-cycle
            for (IEdge<LongWritable, LongWritable, LongWritable> firstEdge : firstVertex.getOutEdges()) {
                IVertex<LongWritable, LongWritable, LongWritable, LongWritable> secondVertex = getSubgraph().getVertexById(firstEdge.getSinkVertexId());
                // Confirms that 4th vertex forms Type-5, 4-cycle
                if(secondVertex.getVertexId().get() == firstList.get(i+1))
                    type5 += 0.25;
            }
        }
        else if(firstList.get(i) == -2)
        {   // For Type-6, 4-cycle
            for (IEdge<LongWritable, LongWritable, LongWritable> firstEdge : firstVertex.getOutEdges()) {
                IVertex<LongWritable, LongWritable, LongWritable, LongWritable> secondVertex = getSubgraph().getVertexById(firstEdge.getSinkVertexId());
                // Confirms that 4th vertex forms Type-6, 4-cycle
                if(secondVertex.getVertexId().get() == firstList.get(i+1))
                    type6 += 0.5;
            }
        }
        else
        {   // For Type-7, 4-cycle
            for (IEdge<LongWritable, LongWritable, LongWritable> firstEdge : firstVertex.getOutEdges()) {
                IVertex<LongWritable, LongWritable, LongWritable, LongWritable> secondVertex = getSubgraph().getVertexById(firstEdge.getSinkVertexId());
                // Confirms that 4th vertex forms Type-7, 4-cycle
                if(secondVertex.getVertexId().get() == firstList.get(i+1))
                    type7 += 0.25;
```

- **Code Snippet for zipping message before sending it to next Super-Step.**

```
// Zipping message send by each Sub-graph from Super-Step 2 to 3
private void step3Zip(Map<Long, ByteArrayHelper.Writer> msg, long remoteSubgraphId, long secondVid, long firstVid, long typeOfMessage) {
    Writer writer = msg.get(remoteSubgraphId);
    if (writer == null) {
        // We are taking buffer of default size, for bigger input graphs we can customize the size of buffer y passing size as parameter
        writer = new Writer();
        msg.put(remoteSubgraphId, writer);
    }
    // Pushing elements in order we want
    writer.writeLong(secondVid);
    writer.writeLong(typeOfMessage);
    writer.writeLong(firstVid);
}
```

- **Code Snippet for unzipping message after receiving it and appending it in a list to access however we want.**

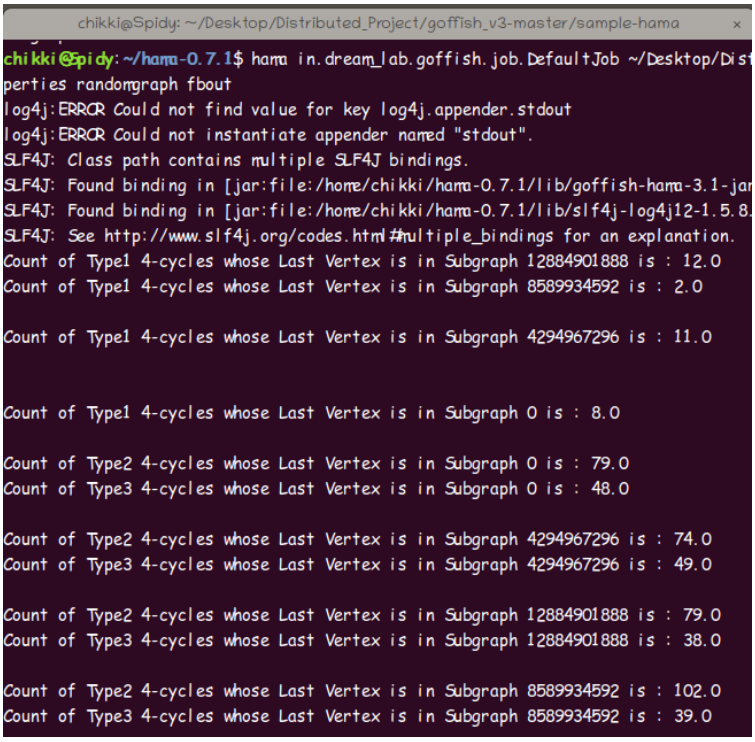
```
// Unzipping message at Super-Step 3
private void step3Unzip(Iterable<IMessage<LongWritable, BytesWritable>> messageList, Map<Long, List<Long>> transitionIdsMap) {
    for (IMessage<LongWritable, BytesWritable> messageItem : messageList) {
        BytesWritable message = messageItem.getMessage();
        ByteArrayHelper.Reader reader = new Reader(message.copyBytes());
        while (reader.available() >= 16) {
            // Getting elements in order we inserted
            long secondVid = reader.readLong();
            long typeOfMessage = reader.readLong();
            long firstVid = reader.readLong();
            List<Long> firstVidList = transitionIdsMap.get(secondVid);
            if (firstVidList == null) {
                firstVidList = new LinkedList<Long>();
                transitionIdsMap.put(secondVid, firstVidList);
            }
            // Pushing elements in order we want
            firstVidList.add(typeOfMessage);
            firstVidList.add(firstVid);
        }
```

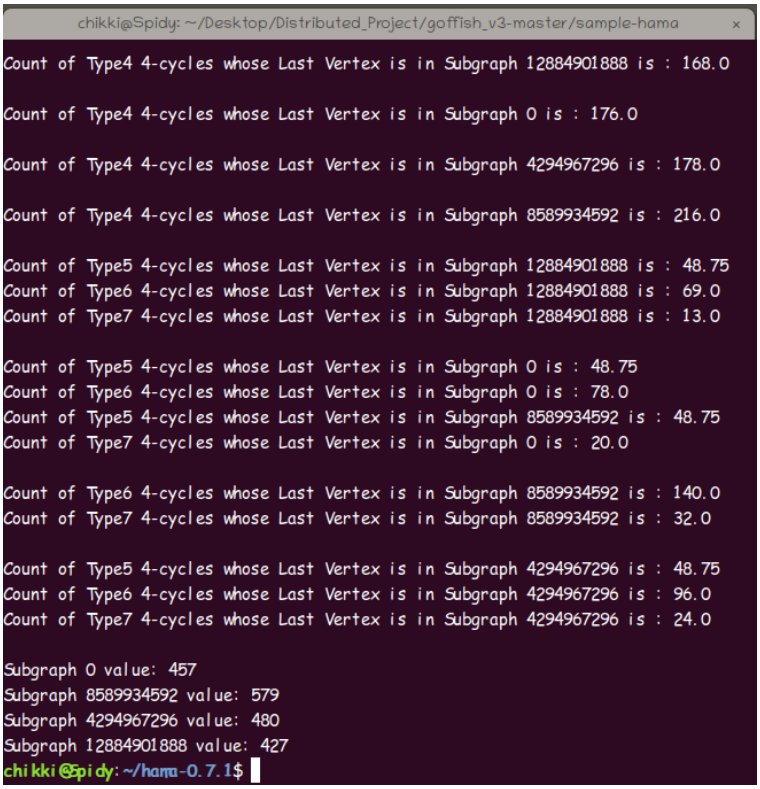
- **Code Snippet to send messages to all process for next Super-Step according to Sub-graph Id.**

```
void sendMessages(Map<Long, ByteArrayHelper.Writer> msg) {
    // Sending Message to all Sub-graphs according to Key set as Sub-graph Id to which it has to be sent
    for (Map.Entry<Long, ByteArrayHelper.Writer> m : msg.entrySet()) {
        BytesWritable message = new BytesWritable(m.getValue().getBytes());
        sendMessage(new LongWritable(m.getKey()), message);
    }
}
```

## 4.5 Result Snaps

**Below are attached some result snaps for a graph which has 100 vertices each having at most 10 outgoing edges and whole graph is divided in 4 Sub-graphs each containing 25 vertices.** (Type of 4-cycles are same as shown above in the images)



### 4.6 Other details

- Technologies Used : GoFFish-API provided function abstractions, Java for main Algorithm, Hama for graph processing, Maven as build tool and C++ to validate answers.
- Some useful commands needed to run this project apart from setting up GoFFIsh and Hama are mentioned below:
  - **Generating a Random Graph** : hama jar hama-examples-0.7.1.jar gen fastgen -v Number_Of_Vertices -e Max_Outgoing_Edge_from_any_Vertex -o randomgraph -t No_of_partitions/SubGraphs
    - eg. hama jar hama-examples-0.x.x.jar gen fastgen -v 100 -e 10 -o randomgraph -t 2
  - **Building your Project Using Maven** : mvn clean install (This command should be run inside the main Directory of project and after making properties file)
  - **Running any Sample Example or Your new Code using Hama**
    hama in.dream_lab.goffish.job.DefaultJob $Path_To_Properties_File_of_that_program$ $Path_To_randomgraph_generated_above$ $Path_to_Dump_file$
    (should be executed inside Hama Package Directory)
    - eg. hama in.dream_lab.goffish.job.DefaultJob /Desktop/Distributed_Project/goffish_v3-master/Docker/Hama/Goffish_Hama_Bin/properties/QuadCount.properties randomgraph fbout
- For code of counting 4-cycles you can refer to this link : `https://github.com/shivam496/Algorithms-Implemented-Using-GoFFish/blob/master/QuadCount.java`
- For other details like understanding algorithm or code you can refer to this video : abc.xyz

### 4.7 Improvements

- The loops in each Super-Step after the first one can be combined and the code can be made more efficient, but it is left that way for the sake of understanding and to see how code differs for each Type of 4-cycles.
- There is an ongoing work on BiVertex module, current we are using IVertex in GoFFish. In BiVertex along with Outgoing edges from a vertex we can also access Incoming edges which will decrease 1 Super-Step in each case and making our algorithm more efficient.

## 5 Reference


1. Diptanshu Kakwani and Yogesh Simmhan, "Distributed Algorithms for Subgraph-Centric Graph Platforms". `https://arxiv.org/pdf/1905.08051.pdf`
2. Github Repository for GoFFish V3. `https://github.com/dream-lab/goffish_v3`