CS5234 Project Report

Chan Wai Hap (Axxxxxx) xxxxx@u.nus.edu

Lu Wei (A0040955E) wei.lu@u.nus.edu

November 18, 2018

Contents

1 Overview

The density of a graph is defined as the number of edges divided by the number of nodes.

$$\rho(G) = \frac{m}{n} \tag{1}$$

Where m is the number of edges and n is the number of nodes of graph G. This is a convention we are going to use consistently for the rest of this report.

A dense subgraph is induced by a set of nodes in the original graph with many edges connecting them. The problem of finding the densest subgraph for undirected graphs is first formalized by Andrew Goldberg in 1984 (REFERENCE). He proposed a polynomial time exact algorithm for both unweighted and weighted graphs. It has been an active area research since. This class of problems are interesting not just from a theoretical perspective, they also have a lot of practical applications, such as community detection in social networks, web link spam detection for search engines, and correlation mining for gene, item or time series datasets. Such graph datasets are usually large with millions and even billions of nodes, which makes poly-time algorithm impractical. This prompts research in the direction of both distributed and approximation algorithms.

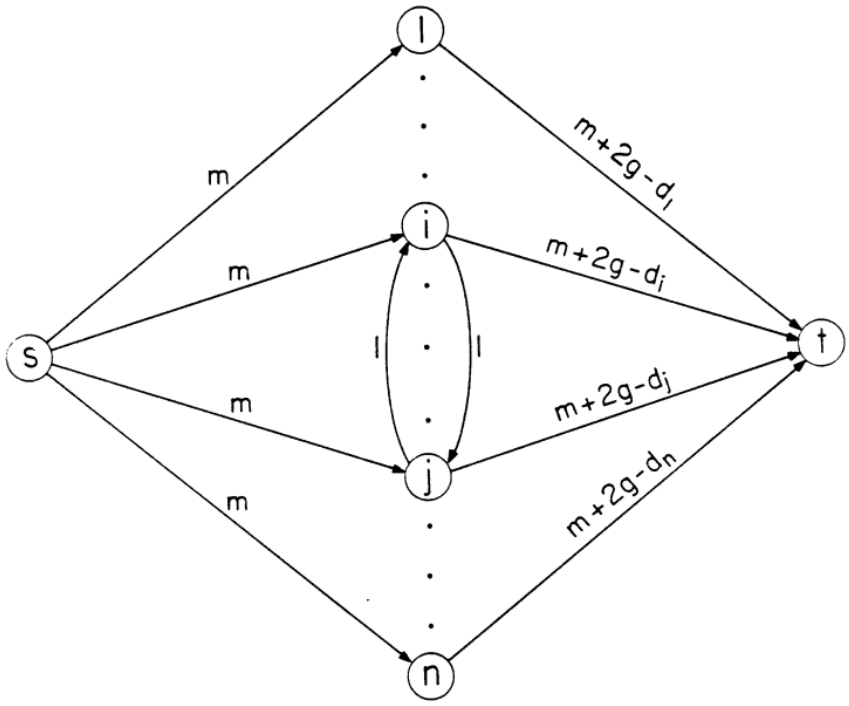
In our project, we limit the scope of our investigation to unweighted and undirect graphs. First, we explored both exact and approximation algorithms from existing literature for the densest subgraph problem, and formulated a distributed exact algorithm. For experimentation, we implemented Goldberg's exact algorithm, which we used to produce baselines for various datasets we experimented with. We then replicated and improved a MapReduce implementation of an approximation algorithm by Bahman et al in Hadoop – a distributed big data processing framework built on top of the MapReduce computational model. Last but not least, we also adopted the same approximation algorithm to a distributed graph processing framework named Giraph. We will present challenges and key lessons learned and compare the Hadoop and Giraph implementations.

2 Algorithms and Theory

In this section, we will introduce Goldberg's exact algorithm and briefly cover several approximation algorithms. Lastly, we will present our distributed exact algorithms and two possible realizations.

2.1 Exact Algorithm

The key idea of Goldberg's exact algorithm is to convert the original undirected, unweighted graph into a flow network as illustrated below:



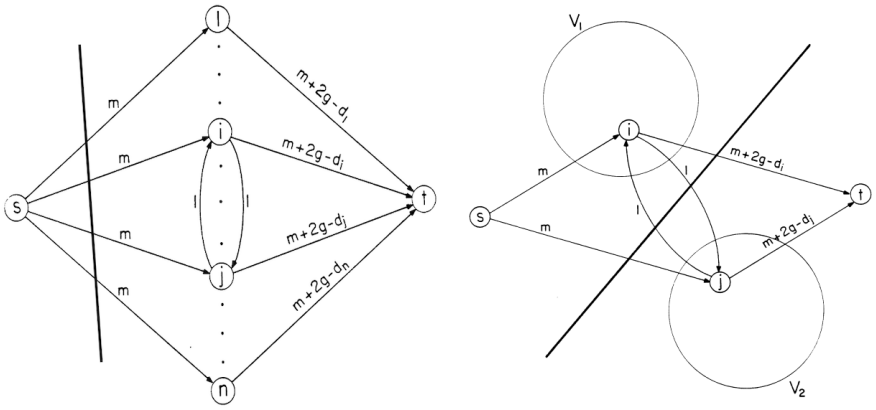
Comparing to the original graph with nodes V and edges E, the network has two nodes added: the source s and the sink t: $V_N = V + \{s,t\}$, and $m+2n$ edges added: $|E_N| = 2|E| + 2|N| = 2m+2n$. Every original unweight and undirected edge is converted to two directed edges each with capacity 1. The capacity from s to every node in the original graph is m; and the capacity from every node i in the original graph to t is assigned $m+2g-d_i$, where d_i is the degree of node i in the original graph, and g is a guessed value which the algorithm is going to iteratively "sandwich" by finding the min-cut until g converges to the density of the densest subgraph.

The pseudocode for the algorithm is as below:

```
l=0, u=m, V1 = ∅
while u − l > 1/(n(n−1)) do
  g = (u+l)/2
  Construct network the updated g: N = (VN, EN, g)
  Find min−cut (S, T)
  if S = \{s\}
    u = g
  else
    l = g
    V1 = S − s
end
```

The intuition here is that the min-cut either goes throught all the out-edges of source

node s, which looks like Case 1 below, or it doesn't (Case 2). In Case 1, the min-cut $c(S,T) = mn$; while for Case 2, the min-cut $c(S,T) = mn + 2n_1(g - \rho_1)$, where n_1 is the number of nodes in the subgraph induced by V_1 and ρ_1 is the density of the same subgraph. For detailed derivation please refer to the original paper (REFERENCE). Notice that the mn term appears in both cases, so for Case 2 to be the min-cut, it has to be the case that $2n_1(g - \rho_1) < 0 \Rightarrow g < \rho_1 \leq \rho^*$ where ρ^* is the densest subgraph density. It follows that when Case 1 is the min-cut, $g > \rho^*$.



(a) Case 1 (b) Case 2

Since the value of g is determined through binary search in the range of $[\frac{1}{n(n-1)}, m]$, it requires $log(n(n-1) \cdot m) = O(log(n^4)) = O(logn)$ iterations to terminate. Each iteration's running time is bounded by the time spent finding the min-cut in the contructed flow network.

There's extensive literature on finding s-t min-cut. The best known exact algorithm for general graphs is by Nagamochi and Ibaraki, which runs in time $O(mn + n^2 logn)$ (REFERENCE). In comparison, the classic push-relabel algorithm by Goldberg and Tarjan (REFERENCE) runs in $O(n^3)$ time for a sequential implementation. Therefore for very dense subgraphs $m \approx n^2$, the best known min-cut algorithm's time complexity is also bounded by $O(n^3)$. Using $O(n^3)$ as the upper bound, the Goldberg densest subgraph algorithm described above has running time of $O(n^3 logn)$.

We apprciate the cleverness of this algorithm in which one graph problem (densest subgraph) is translated to another graph problem (min-cut) through converting the original graph to a network flow with carefully assigned capacities. This is a common and effective technique we have come across while conducting literature review for this project.

2.2 Approximation Algorithms

We would like to focus on one approximation algorithm for finding densest subgraph in this section. It is proposed by Bahmani, Kumar, and Vassilvitskii (REFERENCE). The algorithm provides a $2 + 2\varepsilon$ approximation solution that completes in $O((m+n)logn$

time for a sequential implementation. We find this algorithm most impressive among its kind for the following reasons:

1. Its distributed implementation completes in $O(logn)$ rounds,
2. The algorithm itself is extremely simple, and
3. It provides great close-to-truth approximation on real-world data.

The key idea of this algorithm is to repeatedly remove small-degree nodes until the graph is empty. The intermediate subgraph with the highest density is the approximate solution to the densest subgraph problem. Below is the pseudocode:

```
V_S, Ṽ = V
while V_S ≠ ∅ do
    V_R = {i ∈ V_S | deg_S(i) ≤ 2(1+ε)ρ_S}
    V_S = V_S V_R
    if ρ̃ < ρ_S
        Ṽ = V_S
end
```

Intuitively, this algorithm works because dense graph has large average degree, $\rho(G) = \frac{m}{n} = \frac{\sum_{i \in V} deg_G(i)}{2n} = \frac{1}{2}$*(average degree)*. Therefore by removing small degree nodes, we expect the density of the subgraph induced by the remaining nodes to be large. The correctness and approximation bounds are established by setting the degree threshold carefully.

Let V^* be the set of nodes in the true densest subgraph, ρ^* be the true density. Removing any node i from V^* will weakly reduce the density: $\rho^* \geq \rho(V^* \ \{i\}) = \frac{m^* - deg_{V^*}(i)}{n^* - 1}$. Since $n^* > 1$, we can multiply $n^* - 1$ on both sides of the inequality:

$$\begin{aligned} \rho^* \cdot n^* - \rho^* &\geq m^* - deg_{V^*}(i) \\ m^* - \rho^* &\geq m^* - deg_{V^*}(i) \\ \rho^* &\leq deg_{V^*}(i) \end{aligned} \tag{2}$$

Let i be a node in the true densest subgraph $i \in V^*$, but we removed from V_S: $i \notin V_S$. This is bound to happen as V_S is empty in the end. Since $V^* \subset V_S$ and i is removed because $deg_{V_S}(i) \leq 2(1+\varepsilon)\rho_S$. Combined with the inequality above, we have $\rho^* \leq deg_{V^*}(i) \leq deg_{V_S}(i) \leq 2(1+\varepsilon)\rho_S$, dividing $2(1+\varepsilon)$ on both sides:

$$\rho_S \geq 2(1+\varepsilon)\rho^* \tag{3}$$

This proves that the algorithm produces a $2(1+\varepsilon)$ approximation solution.

We can see that for every round of the algorithm, it reduces the graph size by a factor of at least $1+\varepsilon$. For node $i \in V_{St}$:

$$
\begin{aligned}
\sum deg_{V_{St}}(i) &\geq n_{St} 2(1+\varepsilon)\rho_{S(t-1)} \\
2m_{S(t-1)} > \sum deg_{V_{St}}(i) &\geq n_{St} 2(1+\varepsilon)\frac{m_{S(t-1)}}{n_{S(t-1)}} \\
\frac{n_{S(t-1)}}{n_{St}} &\geq (1+\varepsilon)
\end{aligned} \tag{4}
$$

Therefore the algorithm terminates in $O(log_{(1+\varepsilon)}n)$ rounds. At each round, we check every node, which needs to count all its neighboring edges to get its degree, so the work per round is $O(m+n)$. Since degree is a node local property, the algorithm can therefore be distributed at every round.

2.3 Distributed Exact Algorithms

As a thought exercise, we explored possible formulations of a distributed exact algorithm for the densest subgraph problem. Our idea is to build the algorithm on top of Goldberg's min-cut based algorithm – we couldn't get around the outer loop which repeats $O(logn)$ times, so we looked into ways to distribute the min-cut finding algorithm, similar to how Bahmani's algorithm distributed the work within each round.

We came across a distributed min-cut algorithm by Nanongkai & Su (REFERENCE). They proposed a $1 \pm \varepsilon$ approximation algorithm based on finding a greedy tree packing: $(1-\varepsilon)\rho^* \leq \rho \leq (1+\varepsilon)\rho^*$. To obtain the exact min-cut, they proposed to first obtain a 3-approximation value ρ' for the min-cut using another distributed algorithm by Ghaffari & Kuhn (REFERENCE), which guarantees that $\rho \leq \rho' \leq 3\rho^*$; then they set their error term $\varepsilon = 1/(\rho'+1)$ to obtain an exact solution:

$$
\begin{aligned}
(1-\varepsilon)\rho^* \leq \rho &\leq (1+\varepsilon)\rho^* \\
(1-\frac{1}{\rho'+1})\rho^* \leq \rho &\leq (1+\frac{1}{\rho'+1})\rho^* \\
(1-\frac{1}{\rho^*+1})\rho^* \leq \rho &\leq (1+\frac{1}{\rho^*+1})\rho^* \\
\rho^* - 1 < \rho &< \rho^* + 1 \\
\rho &= \rho^*
\end{aligned} \tag{5}
$$

The same paper claims that the algorithm runs in $O(\rho^{*4} log^2 n(D+\sqrt{n}log^* n))$ rounds, where D is the diameter of the graph. For detailed proof on the running time, please refer to the original paper.

Recall that for the converted network flow used in Goldberg's exact algorithm, $\rho^* = O(mn)$, therefore making the above time complexity $O(m^4 n^4 log^2 n(D+\sqrt{n}log^* n))$ which is much worse than the time complexity for the sequential version of the original algorithm's $O(n^3 logn)$, so it's impractical in our use case. Regardless, we still want to highlight the merit of Nanongkai's approach of using two approximation algorithms to arrive at an exact solution by setting appropriate value for the error term.

Next, we turned into the classic push-relabel algorithm by Goldberg & Tarjan (REFERENCE). Turned out, this algorithm can be implemented in a distributed setting by breaking the push step into two substeps and perform the label update in between these two substeps. The pseudocode of the algorithm is presented as below. Note that a vertex v is active if it's label $0 < d(v) < n$ and it has excess flow: $e(v) > 0$.

```
For all active nodes v do
  <Push substep 1>
    push flow from v until flow excess e(v) = 0
      or there's no more residual capacity between v and
      any of its neighbor w: d(w) = d(v) - 1, r(v,w) = 0
    reduce e(v) without increasing e(w)
  <Label update>
    if e(v) > 0
      d'(v) = min(d(w) + 1 | ∀w, r(v,w) > 0)
      if d(v) ≠ d'(v)
        d(v) = d'(v)
        send message d(v) to all v's neighbors
  <Push substep 2>
    if there's any flow pushed to v in push substep 1
      increase e(v)
end
```

The above procedure is repeated until there's no more active node. Since the described procedure only concerns node local properties and sending messages to the node's neighbors, it can be performed in parallel among all the active nodes.

The original paper stated and proved that the above procedure can be completed in $O(n^2)$ rounds, which makes our distributed exact algorithm for the densest subgraph problem run in $O(n^2 logn)$ time. This is an improvement from the $O(n^3 logn)$ time complexity of the original Goldberg min-cut based algorithm, however it's still n^2 factor slower than the $2+2\varepsilon$ approximation algorithm. This makes a good thought exercise, but this distributed exact algorithm is infeasible for large graphs with millions and billions of nodes.

3 Implementations and experiments

Among the algorithms described above, we've implemented Goldberg's min-cut based exact algorithmin python, and Bahmani's $(2+2\varepsilon)$ approximation algorithm in python, Hadoop and Giraph. In the following section, we will describe our implementations including the challenges we've faced and resolved and lessons learned.

3.1 Baseline Implementation

First, in order to provide the ground truth for any experiment datasets we may want to play with, we implemented Goldberg's exact algorithm. We chose to use the library graph-tool (REFERENCE) for this implementation for the following reasons:

1. It provides an intuitive and effective python interface for handling graphs.

2. Internally, its data structures and algorithms are implemented in C++, which makes it very fast. This makes it possible for us to play with bigger datasets, including graphs with tens of thousands of nodes.

3. Conveniently, it comes with several min-cut algorithms which we can use directly without having to implement it ourselves.

Below is a sample output of our implementation executed on some of the SNAP (REFERENCE) graph datasets:

```
TODO: update me
python densest_subgraph_goldberg.py
Processing data/ca-HepPh.txt
original: # of nodes: 12008, # of edges: 118521, density: 9.8702
subgraph: # of nodes: 239, # of edges: 28442, density: 119.0042
function [process_graph] finished in 135807 ms
Processing data/ca-GrQc.txt
original: # of nodes: 5242, # of edges: 14496, density: 2.7654
subgraph: # of nodes: 46, # of edges: 1030, density: 22.3913
function [process_graph] finished in 16932 ms
Processing data/ca-AstroPh.txt
original: # of nodes: 18772, # of edges: 198110, density: 10.5535
subgraph: # of nodes: 565, # of edges: 18147, density: 32.1186
function [process_graph] finished in 219283 ms
Processing data/email-Enron.txt
original: # of nodes: 36692, # of edges: 183831, density: 5.0101
subgraph: # of nodes: 555, # of edges: 20726, density: 37.3441
function [process_graph] finished in 1033592 ms
Processing data/ca-HepTh.txt
original: # of nodes: 9877, # of edges: 25998, density: 2.6322
subgraph: # of nodes: 32, # of edges: 496, density: 15.5
function [process_graph] finished in 70048 ms
Processing data/ca-CondMat.txt
original: # of nodes: 23133, # of edges: 93497, density: 4.0417
subgraph: # of nodes: 30, # of edges: 404, density: 13.4667
function [process_graph] finished in 369477 ms
```

This exercise also help us better understand Goldberg's exact algorithm, which gave us the idea of formulating the distributed version of the algorithm described above.

3.2 Hadoop Implementation & Improvement

Hadoop is an open source implementation of the MapReduce programming model proposed by Dean and Ghemawat from Google. The MapReduce model is designed to facilitate general purpose distributed big data processing [1]. A MapReduce job is made of two phases: the Map phase reads key-value pairs from a distributed file system and perform operations such as filtering and transformation and emit intermediate key-value pairs as output. The framework then takes care of shuffling the intermediate output and group them by key and send those with the same key to the same reducer for the Reduce phase. The Reduce phase is analogous to the reduce function in functinoal programming. This is where values of the same key can be aggregated or tallied according to the

user defined reduce function. The output of the Reduce phase is then written into the distributed file system. Multiple MapReduce jobs can be chained to form a workflow to handle general purpose data processing at scale.

The MapReduce model heavily relies on the underlying distributed file system to efficiently manage data distribution and replication. One groundbreaking idea of MapReduce is to bring the computation to data whenever possible, rather than moving data from a separate storage to computation clusters. This in a distributed setting saves time spent on network IO which speeds up the data processing. Hadoop comes with its own implementation of the distributed file system named Hadoop Distributed File System (HDFS).

We used Hadoop Streaming with Python to implement the densest subgraph approximation algorithm. Hadoop Streaming is a Hadoop utility that enables creating and running MapReduce jobs with any executable or script as mappers/reducers. The scripts simply need to read the streaming input through `stdin`. This utility allows MapReduce jobs to be written in other programming languages besides the de facto framework language Java.

In order to process the graph data in a distributed fashion, we first uploaded the edge list data to HDFS. There are three essential MapReduce stages for our implementation: degree counting, threshold calculation, and node removal.

To count the degrees for each node of input graph, the mapper maps each undirected edge (u, v) in the graph into two key-value pairs: $(u, v), (v, u)$; the reducer then simply counts the number of values for each key that represents the node id for each node. The reducer emits both the original node pairs as well as the tallied degrees formatted as $(u, "d", degree)$. Having two fields as value for the degree output allows us to identify and filter such key-value pairs in our subsequent MapReduce stages.

The threshold mapper reads the output from the previous MapReduce job and filter only the key-value pairs with degrees and emits $("n", 1)$ and $("2m", degree)$ for each key-value pair. The reducer then aggregates all the "n" values to get the total number of nodes, and "2m" values to obtain 2 times the total number of edges. It then calculate the density of the current graph and the degree threshold needed for node removal, and store the output on HDFS.

To perform node removal, we need to remove every edges in the graph that are associated to the nodes with degree below threshold during each iteration. It requires two MapReduce passes. For each edge pair (u, v), the mapper first uses u as the mapping key, and it emits an additinal special key-value pair $(u, \$)$ when the mapper reads a node degree below the threshold, otherwise it just copies the input key-value pair. When the reducer later receives input with values containing the special character \$, it omits all the values associated with that key, otherwise, it copies every key-value pair and emit them. In the second pass, the mapper then uses v as the mapping key, and perform the same action; and so does the reducer.

TODO Xavier: add experiment results on actual improvement from merging degree counting & node removal step. Our attempted improvement is that, for each iteration instead of running all these two stages as three different MapReduce phases, we have managed to combine the degree counting into the second phase of node removal, which significantly eliminates the overhead of running an extra MapReduce phase.

3.3 Giraph Adaptation

Giraph is an open source distributed data processing framework specifically designed for processing graphs. It implements the Pregel computational model [2], which in turn is inspired by Valiant's Bulk Synchronous Parallel model [3]. It models the data processing as a sequence of iterations, each iteration is called a "superste". At each superstep, a user defined function is executed, conceptually in parallel for every node of the graph. A vertex can perform update and mutation actions such as changing its label data and removing its neighboring edges, as well as sending messages to its neighboring nodes over outgoing edges and processing any incoming messages from the previous superstep.

Giraph is built on top of Hadoop, since both distributed computational models let user focus on local actions and a superstep can naturally be translated to a MapReduce step. Giraph also added additional features beyond what is described in the original Pregel paper, such as master computation which can be used to facilitate global computation and coordination before the execution of the node centric user defined function of a superstep. This is particularly useful in our case as the algorithm requires calculating the density of the remaining graph at every iteration. During our implementation we find it conceptually helpful to think of a superstep as made of a `MasterCompute` substep followed by a `BasicCompute` substep.

To adapt the densest subgraph approximation algorithm to Giraph, first we need to duplicate and reverse the input edge list in order to represent undirect graphs. Giraph comes with a built-in java class `IntNullReverseTextEdgeInputFormat` which handles this exact case. Then we define our vertex centric function as part of the `BasicComputate` substep:

1. Get degree threshold value computed by `MasterCompute` if there is any
2. If the node's degree (`getNumEdges`) is smaller than the threshold:
   (a) Remove the current node in the next superstep (`removeVertexRequest`)
   (b) Message all its neighboring nodes to remove in-edges (`sendMessageToAllEdges`)
3. For every message received from previous superstep, remove out-edge (`removeEdges`).

And for our `MasterCompute`:

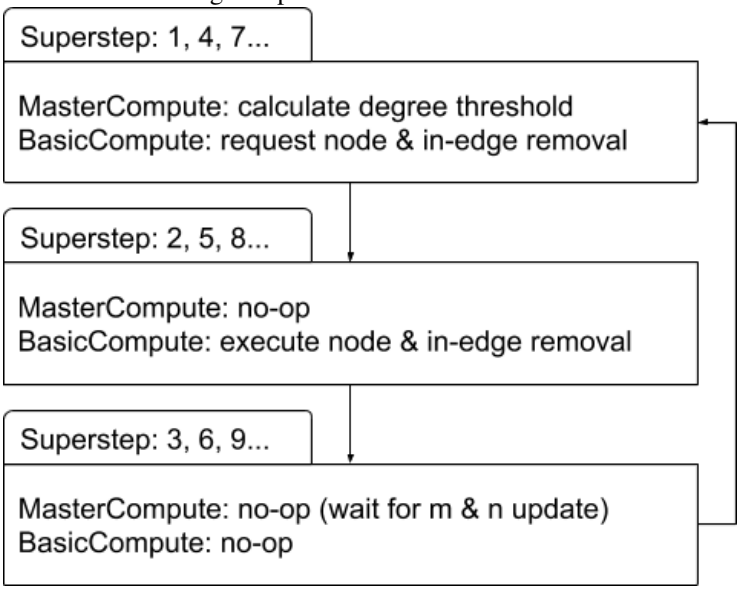
1. If total number of nodes (`getTotalNumVertices`) is zero, terminate
2. Calculate current density and degree threshold
3. Update global `Aggregator` value
4. Read maximum density from a global `Aggregator`. If it's smaller than current density, update it to be current density value.

Communication between `MasterCompute` and `BasicCompute` is achieved through `Aggregators` which is initialized and registered by `MasterCompute` upon its own

initialization. Internally, it reads and writes values onto the distributed file system (HDFS).

Note that every node owns its outgoing edges, but not its incoming edges. This means that removing a node also removes its outgoing edges, but not its incoming edges. Since each `BasicCompute` substep is node local, any attempt at removing an edge that a node doesn't own will not take any effect, including any of its incoming edges. Therefore we utilize messaging to signal the edge owner to remove the incoming edges of any removed node.

Giraph also keeps track of every node's degree and totally number of edges and nodes of the graph by default. However, it's worth pointing out that although total number of edges and nodes are accessible at any time for both `MasterCompute` and `BasicCompute`, the values are only consistent with the state of the graph from the previous superstep. In addition, not all mutation operations are executed instantly: In our case, `removeEdges` is considered a vertex-local operation so it's instantly executed in the same superstep, but `removeVertexRequest` is only carried out by the framework in the next superstep. Combining the lagged view and delayed operation, it means that the `MasterCompute` will only observe the updated total number of nodes and edges resulted from node removal 2 supersteps later; while `MasterCompute` will observe the updated total number of nodes and edges resulted from edge removal 1 supersteps later. This is important to note as it was a source of bug when synchronization is required to guarantee correctness of the algorithm. A common technique used is to check the current superstep count (`getSuperstep`), and perform different actions accordingly, e.g. only perform action 1 on even supersteps. Our adoption of the same technique resulted in the following Giraph workflow:



Another caveat to note is that by default, Giraph will recreate a deleted node when any message is received by that node. This behavior can be disabled through job run configuration option `giraph.vertex.resolver.create.on.msgs`.

In the process of implementing the Giraph adaption of the approximation algorithm, we realized that it's difficult to reason and debug such a distributed program due to unfamiliarity with the framework. One thing that really helped speed up the debugging process is to implement the approximation algorithm in pure python and print out the expected number of nodes, edges and degree threshold at every iteration. Since the algorithm is simple and runs fast, the python implementation is quick to implement and easy to run on a small dataset. By comparing the debug output from the python program and the Giraph program, we are able to track down the sources of the issues and understand quirks and features of the Giraph framework.

4 Conclusion

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[3] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.