ID2222 – Data Mining

Assignment 3 - Mining Data Streams

Course Coordinator:
Vladimar Vlassov & Sarunas Girdzijauskas

Teaching assistants:
Edward Tjörnhammer
Kambiz Ghoorchian
Mohamed Gabr

Group 2

Óttar Guðmundsson

Örn Arnar Karlsson

2018-11-24

Description

The world is producing more data than ever before. Information comes in different forms and structures; graphs are one of them. A graph is a data structure composed of nodes that are connected by edges.

Social networks use graph data to represent their users as nodes and their connections to other users as edges. When a user is suggested, new friends or exciting topics, some process on the graph has been executed, noticing that the two nodes have mutual connections through one or more edges, or the nodes share similar attributes. Other interesting topics such as fraud detection and epidemiology research also involves the use of graphs by counting triangles in a graph structure.

For this assignment, we implemented a space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. This algorithm only requires a single pass through all edges in a dataset and only needs to store $O(\sqrt{n})$ edges (n is the number of vertices) to provide accurate estimates

How to run

We wrote our solution in Scala using a Jupyter notebook with Spark Kernel. A guide to installing Jupyter Notebook, install Spark Scala and connect the kernel to Jupyter can be done using Toree (guide here)

Open the notebook and click “Run all cells” on the toolbar. This will load in all of the nodes in the US Power Grid graph (found here) or the Zachary karate club (found here)as an RDD in Spark, and read them all as an array of tuples that represents an edge. After reading the data, a simulated stream will be generated that reads 1-5 lines every second.

Note: In some cases, we encountered dead kernels in the Notebook. Attached is the file assign3.scala that has all of the code in a pipeline, computing all of our calculations. It can be run by opening the location of the file in terminal and writing *spark-shell -i assign3.scala*.

Solution

We started out using Spark to read in the whole dataset to simulate a stream. A scheduled job would read in 1 to 5 edges every second and for each edge, we would perform the algorithm. This way we made sure to access the list only once with a single pass. We worked in parallel where one of us created the first algorithm that would take care of creating and finding wedges, as well as initializing all the variables and data structures needed. The other one implemented the second algorithm that would take care of updating the edge and wedge list. The code was then merged and a few alterations were made to get things working.
There were some problems to begin with as we were unsure of if our implementation was working correctly. After debugging some of the basic bugs like int to double and other things, we still weren't sure if our solutions was still working as we had different results for every run but we'll talk more about that in the bonus part section.

Bonus

1. **What were the challenges you have faced when implementing the algorithm?**

    One of the biggest challenges was definitely debugging. It can be hard to visualize how the algorithm is running since it relies on probability. One of the problems we had was due to us keeping too small of a reservoir pool size. That meant that the triangle count estimate jumped too rapidly and would often go to zero. We resolved this by tripling the pool size from 100 to 300 for the power grid data set.

    Another problem was when we implemented the initialization for the algorithm. In the beginning it wasn't clear to us that the wedge_res and edge_res where supposed to be empty and we filled them up initially with the first edges. However, we quickly realized that this was wrong.

    We initially chose a data set with over 6000 edges which made debugging even harder. To help with that we switched to a smaller one of considerably smaller size. We also implemented logging logic to make it easier for us to see what was happening.

2. **Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.**

It would be possible to parallelize parts of the algorithm. For example when checking if a certain edge closes any of the reservoir wedges. It would be possible to split the wedges into multiple parts and run that part of the algorithm in parallelization. However, it is not possible to run the whole algorithm in parallelization on different nodes like using map reduce or something similar. The algorithm only works for simple graphs and by running it parallel it would need to create two or more graphs that might be disconnected. This will results in a final merge of both the edge and wedge resorvier, which the algorithm does not take into account. This would also performe worse than running it in on a single node as more space it required. Lets say that you had a graph of 500 vertices and 5000 edges, meaning you needed to store sqrt(500) = 25. If you would split it into two graphs, each with 2500 edges, you wouldn't know how many vertices are in each separate graph. This means that you would need to make sure that the edge resorvier is at least the size you need, so both nodes need to store an array of sqrt(N). Thats's a total of 2x25 = 50, twice as much storage!

3. **Does the algorithm work for unbounded graph streams? Explain.**

   No. For the initialization we need to know the number of vertices. That means that we need to know the bound in order for the algorithm to work.

4. **Does the algorithm support edge deletions? If not, what modification would it need? Explain.**

   The authors themselves mention that this paper does not consider edge/vertex deletions or repeated edges. They design the algorithm in this way in order to simplify the problem.

   We could probably support edge deletions but we would have to make a couple of changes:

   - When edge is deleted we need to close wedges that are closed by that edge
   - We need to remove wedges from the wedge reservoir that contain that edge
   - We need to remove edges from the edge reservoir that are that edge

This hints that the algorithm would need to keep track of way more data structures and attributes which might not be optimal, as the benefits here is to reduce elements needed to store.

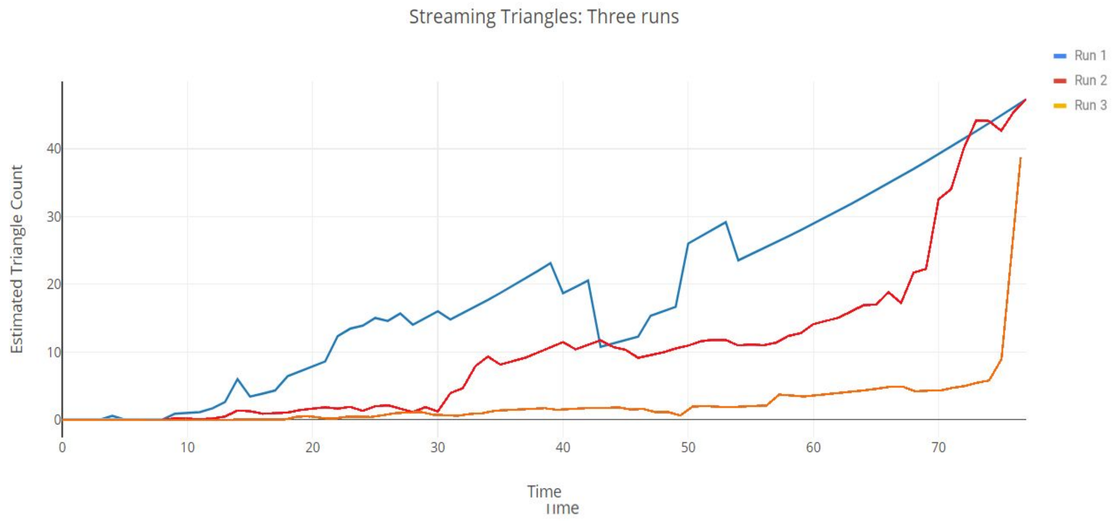
Results

Zachari

We ran our algorithm 3 times for the Zachari karate club data set that we got from this website: http://konect.uni-koblenz.de/networks/.

The results can be seen in the graph and table below.

	Est. Triangle Count	Actual Triangle Count	Accuracy
Run 1	47	45	95%
Run 2	45.6	45	98%
Run 3	38	45	84%

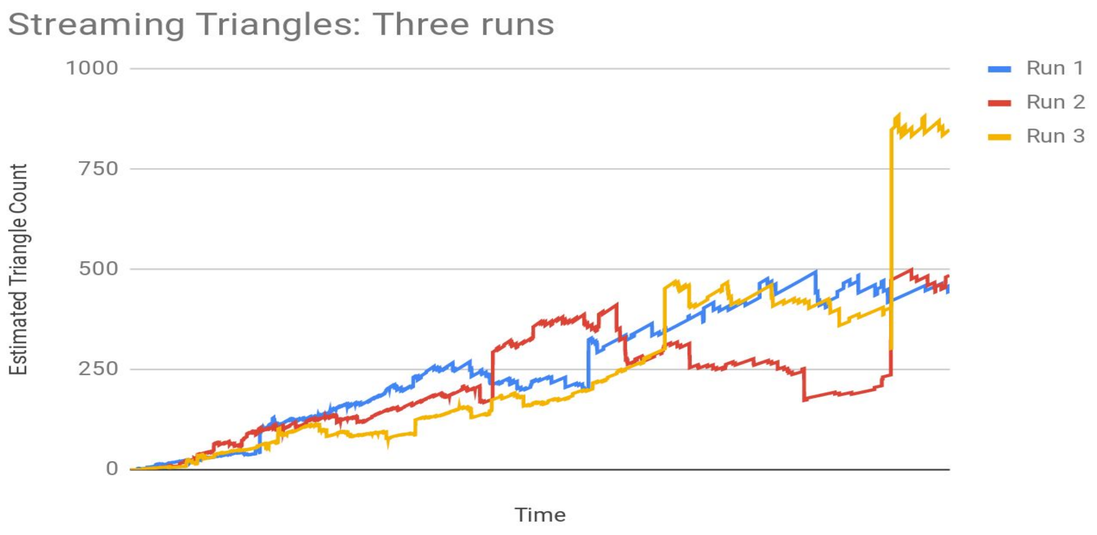


US Power Grid Data Set

We ran our algorithm 3 times for the Us power grid data set we got from this website: http://konect.uni-koblenz.de/networks/.

The results can be seen in the graph and table below.

	Est. Triangle Count	Actual Triangle Count	Accuracy
Run 1	445	651	68%
Run 2	485	651	74%
Run 3	849	651	70%


Streaming Triangles: Three runs
Estimated Triangle Count
Time
Run 1
Run 2
Run 3