Connected Components at Scale: A Comparison of Methods and Implementations

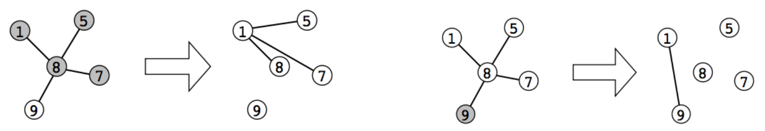
James Kelly

September 24, 2020

0.1 Introduction

Finding connected components in an undirected graph is a classical graph problem. The most well-known, oldest, solution is probably to do a breadth-first, or depth-first, search on the graph, sequentially checking every node until all nodes are marked as visited. This method is slow and infeasable on very large graphs (edges > 30m), where there is too much data to be stored on a single disk. Graphs of these sizes abound in many areas including internet search data, social media community data, and protein synthesis maps, to name a few. Techniques were developed to perform distributed computation of ConnectedComponents, as well as a number of other algorithms such as StronglyConnectedComponents and PageRank at this scale. Pregel mapping is one such technique, relying on a message passing interface to compute graphical relationships. GraphX is a popular Spark library that allows users a higher level interface to Pregel in order to find ConnectedComponents most efficiently. In 2014, Kiveris et alia published a method of computing connected components that outperformed the then s.o.t.a. Pregel Hash-To-Min algorithm on larger graphs with > 1b edges (though not on smaller graphs). This project attempted to reimplement the Kiveras algorithm, hereafter referred to as StarUnion on a Google cluster in 3 different ways: in Spark with scala, in Hadoop streaming, and with mrjob, a library for automating more complex MapReduce jobs. These implementations were ran on 3 real-world datasets and 3 synthetic datasets, along with the classic DFS algorithm, to better understand which tools are best suited for this problem of finding the number of connected components, at increasing scale and composition.

In the 2013 paper Kiveras, et al. test multiple versions of their algorithm. The basic parts are the LargeStar and SmallStar subroutines. These both have a mapper which takes in a single edge and outputs one or more edges, and a reducer which takes in a node and a list of its neghbors and outputs one or more edges. They differ only slightly in that LargeStar connects all strictly larger neighbors to the min neighbor, and SmallStar connects all smaller neighbors and self to the min neighbor including self. An image of both operations from Kiveras, et al. presentation slides is shown below:



(a) The small-star operation at node 8. (b) The large-star operation at node 8.

Figure 1. An illustration of small-star and large-star operations at node $v = 8$. In both cases, the minimum neighbor of 8 is 1. In Figure 1(a), small-star connects 1 to all nodes labels no larger than 8. In particular, 1 is connected to 8. In Figure 1(b), large-star connects 1 to all nodes with label strictly larger than 8. In particular, 1 is not connected to 8.

It became clear, however, that additional steps were needed to perform the algorithms described, namely that between the mapppers and reducers of both Large- and SmallStar, a reduce step would have to be performed to get the neighbors of each node, so that the minimum could be obtained for the reducer step. Additionally, components consisting of only a single node with a self-edge are not accounted for in this algorithm, as LargeStar only emits forward strictly larger edges from either direction. This was discovered in comparing output of our StarUnion algorithms with GraphX on a toy graph to find discrepancies, for validation of correctness purposes. It should be said as well that all three implementations of connected components used: GraphX, NetworkX, our implementation of StarUnion - required the graph to be represented as an adjacency list, read from a file with one edge per line, without duplicates.

Besides testing for differences in alternating vs a 2-phase structure, Kiveras also made 2 optimizations to the algorithm. 1. they kept a distributed hash table in order to save the min neighbor from the end of SmallStar, avoid recomputation in LargeStar, and 2. they introduce a form of load-balancing. Because of the way the algorithm transforms the graph into a union of stars, where every component in a cluster points to the min, this results in an increasing data bottleneck on a single reducer. To address this, they break up large stars (nodes with neighbors over size = Tau), add new nodes with dummy labels from the partitioned cluster to the original min. The first of these optimizations is implemented in a kind of way in our code as a 3rd value that is passed with the edges as the min neighbor.

0.2 Hardware

The largest limitation of this study was the hardware capacity. All experiments were run on a Google dataproc cluster with 5 nodes. While Kiveras, et al. do not explicitly state the hardware architecture used in their experiments, nor the runtimes obtained, instead opting to compare relative performance of the algorithms on the same system. They do allude to the hardware capacity when they mark certain results as unfinished after 24 hours of running. Due to resource limitations this scale of time wasn't possible, and in addition our MapReduce experiments would not run on graphs larger than 34 billion edges due to having each sharded task size still too large. This was unfortunately the result of poor design consideration on our part as this scale is exactly what is required to test the StarUnion algorithm, since they report lesser performance than Pregel on smaller graphs. Our experiments merely confirm this then, where in every case Pregel (often dramatically) outperformed our implementation.

0.3 Implementations

5 total implementations were considered for each graph dataset, 3 of StarUnion: Spark, Hadoop-Streaming, mrjob; 1 of Pregel: Spark+GraphX, 1 of local python with NetworkX. Needless to say the python local implementation with NetworkX was very simple and worked best on the smallest graphs likely due to the overhead of configuring the job on the cluster.

A note on convergence. Determination of convergence is not discussed at the algorithmic level in Kiveras, et al. The authors prove convergence occurs for LargeStar and SmallStar separately and together with an upper bound on the number of iterations on the 2-phase variation, $O(logloglogn)$. It's clear that LargeStar if left to itself woould keep connecting larger parents to smaller children until no situation where a node was connected to a larger node and a smaller node existed - in other words a DAG with a height of 2. SmallStar then connects all the nodes that are equal to or less than, further collapsing the graph. Convergence occurs when the graph does not collapse any further. To know whether the graph has changed we must step outside of the mapreduce framework to capture the state of the graph for comparison with the next iteration. A simple MapReduce job with a number of mappers and combiners and reducers would not suffice. In Spark this was easy to account for by simply putting the mapreduce steps inside of a loop. mrjob uses Hadoop and allows the user to write high-level programs, abstracting away some of the configuration details, and can be called in a routine python function for instance, allowing for a simple loop again. In Hadoop streaming, however, a script had to be written to make the command line calls for each job. This was much less elegant code and is very cumbersome compared to the ease of mrjob but did outperform mrjob slightly. It could be asked why a pure Hadoop implementation in Java wasn't also considered, and it was purely due to the limited experience of the author to this framework and in fact may have been faster than the methods tried, given that older evidence suggests that jobs with very large data requirements are faster with pure Hadoop, while jobs with more computational dependency may be faster on streaming related to language optimizations (Ding et al., 2011).

0.4 Datasets

3 real-world datasets and 3 synthetic datasets were used in these experiments. The 3 real-world graphs are: LiveJournal, Deezer, and Facebook. All real-world datasets were obtained from Stanfords Network database SNAP, and all synthetic graphs were generated with RMAT software.

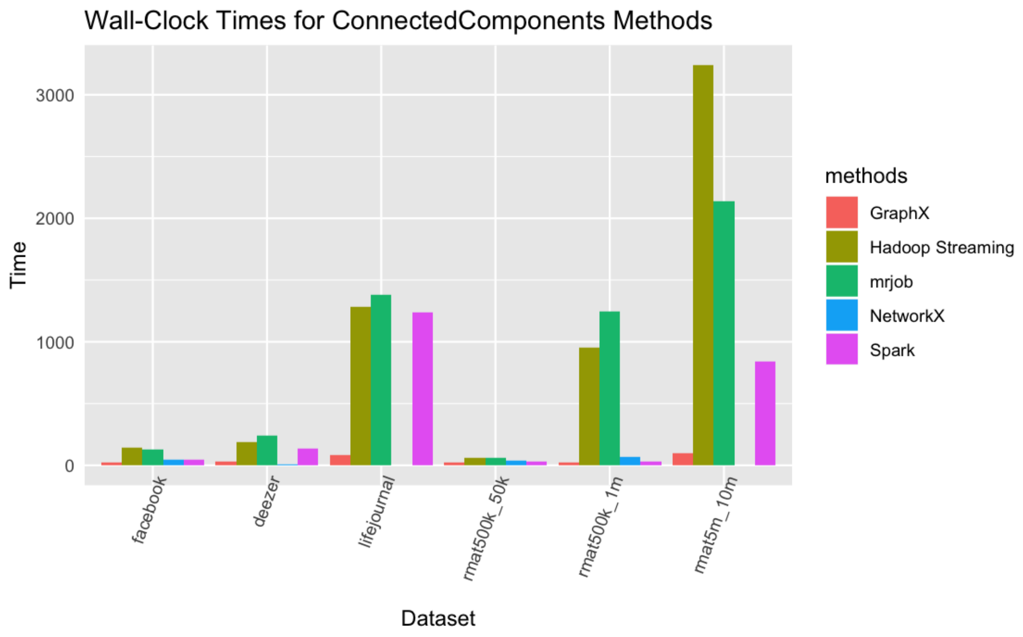
LiveJournal is a free on-line blogging community where users declare friendship each other and may form groups which other members can then join. This is the largest real-world dataset considered. The Deezer data represents an online music community from 3 European countries. The Facebook dataset is a small subgraph of a single component of nodes representing a network of "friends".

Real Dataset	Nodes	Edges	Components
LiveJournal	3997962	34681189	1
Deezer	143884	846915	1
Facebook	4039	88234	1

In order to obtain graphs of the scale required to test StarUnion, Kiveras et al. generated them using RMAT software. RMAT generates graphs by recursively partitioning a graph space into quadrants each with a defined degree probability until the parameters for number of nodes or edges has been met (RMAT source). In the late 1950s Erdos and Renyi proved that for truly random graphs, the probability of connecting to a node that is already connected to a component approaches 1 as the number of nodes approaches infinity (Erdos Renyi, 1958). But real-world graphs approach this much faster because they obey a power-law distribution. That is, for a given node, the probability that it will be connected to another node when an edge is added is proportional to the degree of that node. Friends tend to make more friends. RMAT attempts to mimic real world graphs by accounting for this tendency but ultimately still likely results in more components for very large n that would be present in real-world data. This is evident in the 3 graph datasets created using RMAT for our experiments. Though much much smaller than the >1b graphs created by Kiveras, the fact that they produce smaller and more diverse component make-ups than some of the real world graphs is interesting for the occasion when the data doesn't fit this assumption, or for instance when a strictly defined graph composition is required for a system. In this case, the RMAT creations could be very useful in understanding the way algorithms perform not just at scale, but with varying composition.

Synthetic Dataset	Nodes	Edges	Components
rmat500k_1m	500000	1000000	510
rmat500k_50m	500000	50000	23369
rmat10m_5m	10000000	5000000	455267

0.5 Results



All times are in seconds and taken from the real elapsed time from the time unix module inside the cluster, and reflect the mean times of 3 runs per experiment. None of the methods required more than 6 iterations. Datasets for which no bar is shown for a particular method reflect methods that either took longer than 120 minutes or whose processes were killed automatically. For instance, NetworkX connected components was killed after 15 minutes on the largest RMAT dataset as well as the lifejournal dataset.

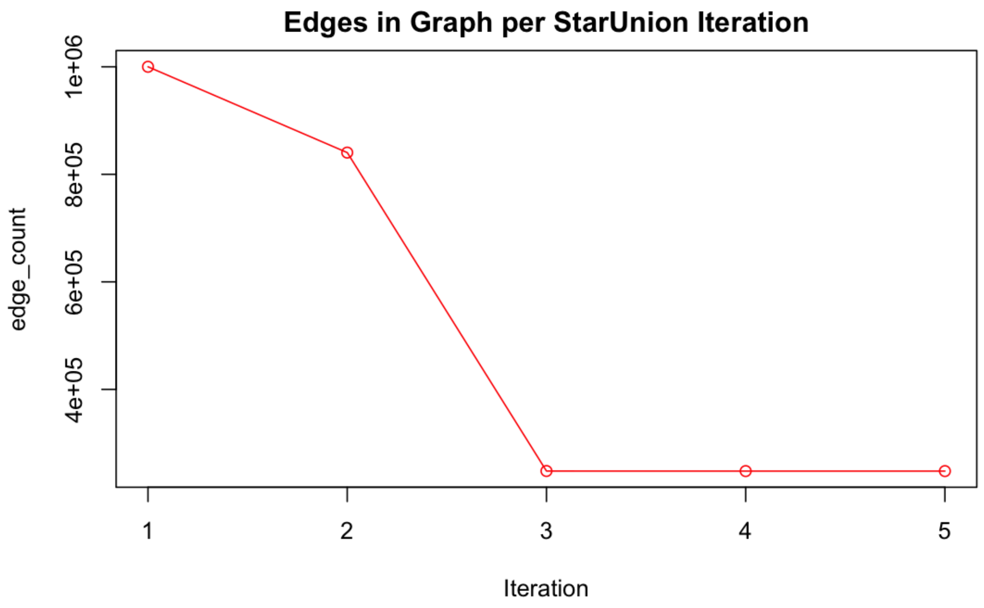
As shown, GraphX with Pregel mapping provided the shortest times for all datasets. Oddly mrjob was faster than the Hadoop streaming method for the RMAT graph with 10 mil. edges, but not for the RMAT graph with 1 mil. edges. This could be because for all experiments no number of executers was specified beyond the automated cluster configuration, and mrjob may have optimized this for smaller jobs, also the streaming method appears to require a significant more reading and writing to disk, though this is not exactly quantified here. For the smallest datasets, ($<$ about 1 mil. edges) NetworkX is very fast, but becomes impractical with larger graphs and certainly those that would not fit on disk.

As noted previously, attempts were made to build larger RMAT graphs having > 1 bil. edges, but met resource restrictions > 10 million edges.

0.6 Discussion

Not reflected in these times are the subjective user experience of using each method. Different users may have different levels of familiarity with a particular method given their history, however, some methods objectively require much less code to implement without significant execution trade-off. Spark with scala was a nice compromise in this sense, requiring a small amount more code than mrjob with vastly improved execution times. It's hypothesized that these time improvements are related to the ability of the Spark framework to provide a smoother flow of data through the mapreduce pipeline and less disk operations. However, GraphX requires less than 10 lines of code to implement, and even in this sense is still superior to the other methods. Once again though, this wasn't refuted by Kevaris, et al. and was never tested at true scale.

Beyond the task of finding the connected components, both the StarUnion and Pregel methods result in a graph transformation into a union of stars with every node pointing to the min node in the component. This almost always results in a graph with less edges. This guaranteed reduction in edges (and therefore graph size) is a key benefit Kevaris et al. claim over and against Pregel, which temporarily expands the graph before compression. Graph compression is itself very useful in some instances. Below is a plot of the number of edges decreasing over iterations of the Hadoop streaming StarUnion implementation on the RMAT500k_1m dataset:



Finally, as of the time of writing this another optimization has been published by a team that iterated on the StarUnion algorithm for load balancing by partitioning the graph and recombining later, reporting even faster times on very large RMAT graphs than Kiveras, et al. (Park, et al., 2020). It will be interesting to see if these results on synthetic graphs align with results on real-world datasets of this size, as we've shown that graph composition is significantly different in the latter case. In these and Kiveras RMAT graph generation, a fixed degree probability was given to each quadrant. It may also be worth exploring further whether these results hold for all possible graph compositions, perhaps by varying this degree. This of course is an infinite space, but could be itself partitioned for some value.

Large graphs today will likely be small graphs in the future and ongoing work in this area shows that improvements to classical graph algorithms like connected components exist at scale and may be vital in processing datasets going forward.

0.7 References

Ding, M., Zheng, L., Lu, Y., Li, L., Guo, S. Guo, M. (2011). "More Convenient More Overhead: The Performance Evaluation of Hadoop Streaming", *Association of Computational Machinery.*

Erdos, P. Renyi, A. (1958) "On Random Graphs I". Retrieved 9/22/2020 from: http://snap.stanford.edu/class/cs224w-readings/erdos59random.pdf

Kiveras, R., Lattanzi, S., Mirrokni, V., Rastogi, V. Vassilvitskii, S. (2013). "Connected Components in MapReduce and Beyond". Retrieved 9/20/20 from: https://arxiv.org/pdf/1203.5387.pdf

Park, H., Park, N., Myaeng, S. Kang, U. (2020). "PACC: Large scale connected component computation on Hadoop and Spark", Retrieved 9/20/20 from: https://doi.org/10.1371/journal.pone.0229936

Graph Datasets: https://snap.stanford.edu/data/
Code: https://github.com/semajyllek/GraphStuff