Comparison of Graph Centrality Algorithms in GraphX

J.T. Liso, Sean Whalen

***Abstract*— GraphX is a popular graph processing system developed at UC Berkeley in 2014. Although the system is widely used, it has received some criticism for its performance and is far from an industry standard. In their paper, GraphX is shown to perform PageRank at competitive runtimes with regards to its peers, but this is only one of many graph centrality algorithms. We compare the performance of other graph centrality algorithms used in research in addition to the PageRank algorithm that GraphX has optimized. We attempted to find whether or not GraphX has proportional performance to PageRank in other graph centrality algorithms. We tested this with centrality algorithms of varying time complexities, namely Degree Centrality, Closeness Centrality, and Eigenvector Centrality. We also compared the runtime of these algorithms with our implementations in the distributed environment of GraphX to a single machine Python environment. From our results, we discovered that PageRank is indeed the fastest centrality measure in GraphX, but that other algorithms are not as amenable to parallelization and do not receive the same performance benefits as PageRank run on a cluster. The time complexities of all four graph centrality algorithms followed a linear pattern in our testing. As expected GraphX implementations were significantly faster than single machine implementations in Python with an exception of degree centrality.**

I. INTRODUCTION

Graph processing has become an integral part of data analysis, especially in the analyses of social networks. GraphX [1] was proposed in 2014 as a graph processing solution built on top of Apache Spark. However, their analyses of the system were limited to only the graph algorithms of PageRank and connected components. PageRank is a graph centrality algorithm focused on finding the most used website in a search result. Although this algorithm is widely used (especially by Google), many other graph centrality algorithms exist and are necessary for other applications. We want to analyze the GraphX system through a variety of graph centrality algorithms of different time complexities and compare their performance of both large and small graphs to those of PageRank.

Graph centrality is a process of finding the most important vertex or group of vertices in a graph. Different algorithms have different definitions of what the most important vertices are in a graph, and consequently have varying purposes. We compare the PageRank algorithm used in the GraphX paper to three other graph centrality algorithms, namely, Degree Centrality, Closeness Centrality, and Eigenvector Centrality. We will only consider simple, undirected, finite graphs, with V vertices and E edges. Additionally, only connected graphs were considered.

Each graph centrality algorithm is very different and relays various information about a graph. The "central" vertex found in one algorithm will most likely not be the "central" vertex in another algorithm. This is due to the computations within each algorithm, so it is important to have the ability to use different graph centrality algorithms quickly. The graph centrality algorithms we analyzed are described in the rest of this section with their corresponding time complexities.

A. PageRank

PageRank [2] is an algorithm developed by Larry Page, Sergey Brin and co-authored with Rajeev Montwani and Terry Winograd at Stanford University in 1996. Originally it was used to supplement the ranking of websites in search results by Google, but can be generalized to determine the ranking of vertices in any graph by importance. Simply speaking, it uses probability distributions to represent likelihood of a certain vertex (website) being visited. The PageRank, *PR(u)*, for a vertex *u* can be found by

$$PR(u) = 1 - d + d \sum_{v \epsilon B_u} \frac{PR(v)}{L(v)}, \tag{1}$$

where B_u is the set of vertices connected to *u* and *L(v)* is the number of edges from vertex *v*. In the original construction the damping factor *d* was fixed at 0.85, empirically determined as the probability a web-surfer randomly clicks on a link from his/her bookmark bar. Intuitively, PageRank works through a kind of peer-review, where a page that is linked to frequently by other reputable pages will have a high PageRank. A page with no backlinks will have a base PageRank of 0.15. PageRank typically runs in *O(V + E)*.

B. Degree Centrality

Degree centrality is probably the most conceptually simple graph centrality measure. The degree centrality of a vertex *v* is found by finding the degree of the vertex, meaning the number of edges connected to *v*. The vertex with the highest centrality, i.e. degree, is considered to be the center of the graph. This algorithm runs in $O(V^2)$ for dense graphs and $O(E)$ for sparse graphs [3].

C. Closeness Centrality

Closeness centrality measures how close all other vertices are to a vertex. Closeness centrality is the inverse of the farness measure [4]. Finding the closeness of a vertex involves the sum of all shortest distances between it and every other vertex, but with the reciprocal taken so that a lower sum creates a higher closeness value. Consequently, closeness can be defined by the equation

$$C(v) = \frac{1}{\sum_u d(v,u)}, \tag{2}$$

where $d(v,u)$ is the distance from vertex v to vertex u. Large closeness centrality indicates vertices are closer than a smaller closeness centrality. Graphs of different sizes can be compared by taking the average rather than the sum

$$C(v) = \frac{N-1}{\sum_u d(v,u)}, \tag{3}$$

where *N-1* is the number of vertices minus one, to factor out the vertex *v* itself. Typically if closeness is run on a disconnected graph it will fail unless run separately on each connected component, though methods exist involving the harmonic mean instead of the arithmetic mean which are thought to model disconnected graphs better. As previously stated, we ignored disconnected graphs. Closeness runs in *O(E)* time after the shortest paths have been found. This means that its runtime depends on the method of finding the shortest paths between every pair of vertices.

D. Eigenvector Centrality

Eigenvector centrality measures the influence of a vertex based on the concept that vertices connected to high-scoring vertices themselves are high-scoring. The general equation for Eigenvector Centrality for a graph *G* with vertex *v* connected to another vertex *t* can be described as

$$x_v = \frac{1}{\lambda} \sum_{t \epsilon M(V)} x_t = \frac{1}{\lambda} \sum_{t \epsilon G} a_{v,t} x_t, \tag{4}$$

where $a_{v,t}$ is the value of the adjacency matrix between vertices *v* and *t*, which is 1, *M(v)* is the set of all connected components in a graph and λ is a constant [5]. However, this equation can be rewritten as the eigendecomposition of the adjacency matrix *A* using some rearranging:

$$Ax = \lambda x. \tag{5}$$

Since Eigenvector centrality is just an eigendecomposition, the time complexity of eigenvector centrality is the time complexity of the eigendecomposition method implemented. We focused on the most common eigendecomposition method used, Power Iteration [6]. Power Iteration has a time complexity of $O(V^3)$.

The rest of the paper is outlined as follows. Section II describes implementation of centrality algorithms in GraphX and Python. Section III explains the datasets we used in our analysis. Section IV displays and explains the results of our implementations with Section V providing concluding remarks. We propose future work for our project in Section VI.

II. METHODS

Our goal with our implementations was not to propose any new radical changes to the current best known centrality algorithms. Instead we chose to implement the most standard, commonly-used implementations of each algorithm in GraphX and Python so that the underlying systems could be compared.

A. Implementations in GraphX

PageRank, was easy to implement with the built-in functionality of GraphX. As one of the canonical examples of GraphX's utility we were happy to use the implementation directly from GraphX. We specified the tolerance, the smallest amount the "ranks" can change before convergence, to be 0.0001.

```
val ranks = graph.pageRank(0.0001).vertices
```

Degree centrality was also natural to implement in GraphX, we leveraged the degree operator of the Graph class, mapped these degrees against a normalization factor of the reciprocal of the number of vertices but one, and sorted in descending order.

```
val normalizationFactor:Float = 1f/(graph.vertices.count()
    -1)
val degrees: VertexRDD[Int] = graph.degrees.persist()

//sort vertices on descending degree value
val normalized = degrees.map((s => (s._1, s._2*
    normalizationFactor)))
```

Closeness centrality did not have a built-in function in GraphX. Our implementation follows the mathematical definition. First the shortest paths are calculated, and then the closeness is derived from the reciprocal of the sum of these distances per vertex.

```
//create a new RDD with just VertexId to be used for
    shortest paths algorithm
val vertexSeq = graph.vertices.map(v => v._1).collect().
    toSeq
val spgraph = lib.ShortestPaths.run(graph, vertexSeq)
val closeness = spgraph.vertices.map(vertex => (vertex._1
    , 1f/vertex._2
.values.sum))

//sort vertices on descending degree value
val sorted = closeness.sortBy(- _._2)
```

The overwhelming majority of the runtime for closeness centrality stems from the shortest paths function call, the operation of which though externally implemented merits some discussion. As is the case with many GraphX parallelized operations the underlying architecture is the Pregel operator, a bulk-synchronous parallel messaging abstraction where at each step vertices aggregate incoming messages, compute a new value, and relay said value to their neighbors. In the shortest paths function vertices maintain a map of the shortest distance to every other vertex.

```
private def makeMap(x: (VertexId, Int)*) = Map(x: _*)
val spGraph = graph.mapVertices { (vid, attr) => makeMap(
    vid -> 0) }
```

The aggregation function per vertex, or the "vertex-program", upon receiving a map of distances from a neighbor chooses the minimum distance between that map's entries and the currently stored one's.

```
private def addMaps(spmap1: SPMap, spmap2: SPMap):SPMap=
{

(spmap1.keySet ++ spmap2.keySet).map {
  k => k -> math.min(spmap1.getOrElse(k, Int.MaxValue),
    spmap2.getOrElse(k,Int.MaxValue))
}(collection.breakOut)

}
```

```
def vertexProgram(id: VertexId, attr: SPMap, msg: SPMap):
    SPMap = {
  addMaps(attr, msg)
}
```

Then sending messages adds one to the incoming distance to factor in that vertex's presence on the path. If the new shortest distance doesn't agree with the stored value, all the vertex's neighbors will be notified of the updated distance. Otherwise no messages are sent. When no more message are sent the algorithm converges.

```
def sendMessage(edge: EdgeTriplet[SPMap, _]): Iterator[(
    VertexId, SPMap)] = {
  val newAttr = incrementMap(edge.dstAttr)
  if (edge.srcAttr != addMaps(newAttr, edge.srcAttr))
    Iterator((edge.srcId, newAttr))
  else
    Iterator.empty
}
```

These functions along with an initial message, and message combinator, are fed into the Pregel operator and drive the computation of the shortest paths algorithm. With every vertex maintaining a map of distances to every other vertex at scale we discovered this implementation to be very computationally expensive.

Our eigenvector centrality method first initializes a graph's eigenvalues per vertex to one over the number of nodes, the minimum influence.

```
var oldValue = graph.mapVertices((vId, eigenvalue) => 1.0
    / nodeNumber)
```

Then we set the weight on the edges of the graph based on the outdegree, to prepare this information for transmission across triplets.

```
var rankGraph = oldValue
  .outerJoinVertices(graph.outDegrees) { (vid, eigenvalue,
      deg) => deg.getOrElse(0)}
  .mapTriplets(e => 1.0 / e.srcAttr)
  .outerJoinVertices(newVertices) { (vid, deg, eigenValue)
      => eigenValue.getOrElse(0.0) }
```

Finally MapReduce distributes and collates the information across the triplets. When the convergence, the sum of absolute differences of old and new eigenvalues for each vertex, dips below 0.00015 we terminate, or at a maximum of 1000 iterations.

```
newVertices = rankGraph.aggregateMessages[(Double)](

//calculate how much each vertex "contributes" to the
    destination vertex
triplet => {
  triplet.sendToDst(triplet.srcAttr*triplet.attr)
},

// Add all vertices old eigenvalues of inVertices to sum
    the eigenvalue of each vertex
(a, b) => (a + b)

)
```

B. Implementations in Python

Implementations in Python are directly from the Networkx suite of graph processing functions. Networkx served as an industry standard with which we could gauge the performance benefits of GraphX against. The Networkx (nx) functions used were:

- `nx.pagerank`
- `nx.closeness_centrality`
- `nx.eigenvector_centrality_numpy`
- `nx.degree_centrality`

We verified that all four of these algorithms had time complexities matching the algorithms. For example, Networkx's degree centrality algorithm did in fact have a $O(V^2)$ runtime.

III. DATA

In order to test the graph centrality algorithms in different environments, we generated several random graphs based on the Erdős-Rényi model *G(n,p)*, where *n* is the number of vertices and *p* is the probability of an edge between any two vertices exists [9]. Graphs were generated using Networkx in Python [7], and saved as an edge-list file. To maintain consistency across the graphs and test solely based on the number of vertices, we kept *p* constant with a value of .3. Graphs had vertices ranging from 100 to 8000 in increments of 100. We intended to generate larger graphs, but it was quite computationally expensive to generate large random graphs of larger sizes. The same graphs were used for testing the Python and GraphX implementations.

IV. RESULTS

All tests were run on the Clemson cluster of the EducationProject group on Cloudlab. Each node on this cluster has 16 cores (2 CPUs) with 256 GB of memory, running Ubuntu 16.04.1 (GNU/Linux 4.4.0-75-generic x86_64). Spark 2.3.0 was used on top of Hadoop 2.7.3 for the underlying GraphX MapReduce architecture. Scala 2.11.6 was used for the GraphX programming. For the Python implementations, Python 2.7.12 was used with version 2.1 of Networkx.

A. GraphX Algorithm Runtimes

As seen in Figure 1, the four different algorithms we tested have runtimes that differ greatly. PageRank is the fastest algorithm as was expected with a maximum runtime of about 5 seconds on 8000 vertices. Moreover, the PageRank algorithm follows a linear pattern based on the number of vertices, which we confirmed through finding a line of best fit.

Degree centrality would seem to scale quadratically compared to PageRank since its maximum runtime is about 35 seconds on 8000 vertices. This would be consistent with its quadratic time complexity of $O(V^2)$. However, when we calculated the quadratic curve of best fit, the quadratic coefficient was 0 with a non-zero linear coefficient meaning that the runtime follows a linear pattern. Additionally, both PageRank and degree centrality have a sharp jump in the runtime at 5000 vertices. This probably has to do with some underlying GraphX architecture where more storage is needed at this point, causing a more distributed approach to the code.

Eigenvector centrality seems to follow closely to the runtime of Degree centrality up until graphs with 7200 or more vertices where eigenvector centrality has a tremendous increase in runtime. Thus, just like degree centrality, it

follows a linear progression up until this point, unlike what it's cubic time complexity would suggest. This was once again confirmed through attempting to find the curve of best fit. When we tried to find a cubic curve for the data, we found that the cubic and quadratic terms were zeroed. After 7200 vertices, however, we found that the runtimes followed a quadratic relationship with the number of vertices. Though, this still is inconsistent with the cubic complexity of the algorithm.

Closeness centrality is significantly slower than the other algorithms as seen in Figure 1. The algorithm can only handle graphs with less than 3000 vertices as it crashes beyond that point with memory errors. As previously stated, the time complexity of closeness centrality is heavily dependent on the complexity of the shortest path algorithm used. We used the shortest paths algorithm in the GraphX library of Spark. This has a time complexity of $O(V^2)$, which we determined by reading the source code of the algorithm[1]. However, this does not explain the extremely slow runtimes for relatively small graphs. Our implementation is as simple as we could make it, only using basic Spark and GraphX function calls, but this still caused the extremely slow runtime. We believe it is due to the large data structures needed in closeness centrality for calculating and storing the shortest paths between all vertices. This causes slow access time and, consequently, slow computation. Closeness centrality is also the slowest algorithm in Python as seen in Figure 2, but the disparity between the algorithms is not as large in Python as it is in GraphX.

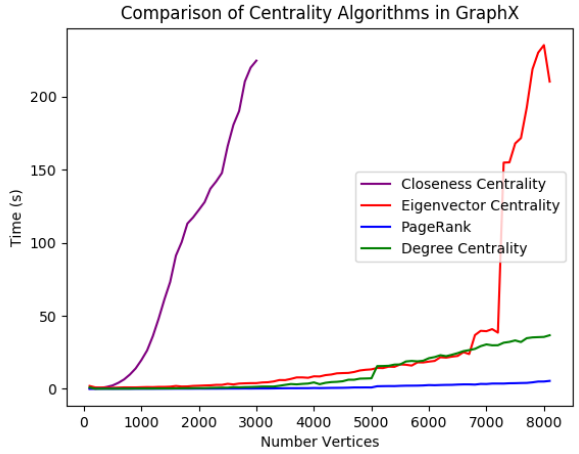

Fig. 1. Comparison of runtimes of graph centrality algorithms using the GraphX framework, labeled appropriately. The runtimes follow the a linear pattern with PageRank being the fastest algorithm with a maximum runtime of about 5 seconds for 8000 vertices. Closeness centrality failed on graphs with 3000 or more vertices.

B. GraphX Algorithm Robustness

PageRank, degree centrality, and eigenvector centrality never failed on any of the graphs we implemented. Because of the lack of failures, we cannot comment on the robustness of these algorithms fully. These three algorithms do not require much extra storage in their algorithms, so it is unlikely for them to fail due to memory errors on graphs of sizes we used.

Closeness centrality failed on graphs with 3000 or more vertices due to memory errors. We believe this is due to the large storage associated with closeness centrality. In addition to storing the entire graph, Closeness centrality also requires the storage of the shortest paths between all vertices. This data is both computationally expensive and memory expensive which leads to the crash at relatively small graphs compared to the other algorithms.

C. Comparison to Python

We compared these runtimes to the same algorithms implemented in Python. The graphs of the four algorithms' runtimes in GraphX and in Python can be seen in Figure 2 at the top of the next page. GraphX speeds up computation in three of the four algorithms - PageRank, closeness centrality, and eigenvector centrality. In PageRank and closeness centrality, GraphX is faster in graphs of all sizes. For eigenvector centrality, GraphX is actually slower than the Python implementation for up to graphs with about 1500 vertices. The runtime gap between the two implementations is rather small, so it is not too notable. The difference is most likely due to some convergence factor when we use Power Iteration.

Degree centrality is the outlier from the other algorithms in terms of comparison of GraphX and Python runtimes. In all cases, degree centrality runs faster in Python than it does in GraphX. This is due to the use of Networkx in our Python implementations. Networkx stores the degrees of each vertex as the graph is created. This way when we find the degree centrality, which just sorts the degrees in descending order, we do not need to calculate the degree of each vertex. On the other hand, in our GraphX implementation, we have to manually calculate the degree of each vertex after we create the graph. Then we can sort the list of degrees to find the center by degree centrality. This causes more computation and consequently a slower runtime in GraphX than in Python.

V. CONCLUSIONS

From our results, we have a better understanding of how the four graph centrality algorithms we studied compare in a distributed computing environment. As was expected, PageRank performed the quickest since GraphX referred to this algorithm for testing results in their initial paper. The various graph centrality algorithms do not match their indicated time complexities as we had originally thought. Additionally, the runtimes of the different algorithms scale linearly with the exception of closeness centrality. We are unsure of exactly what causes this linear complexity in all cases, but we believe it has to do with the distributed processing of GraphX. Since the processing is distributed among the Spark environment, a change in the number of

[1]https://github.com/apache/spark/

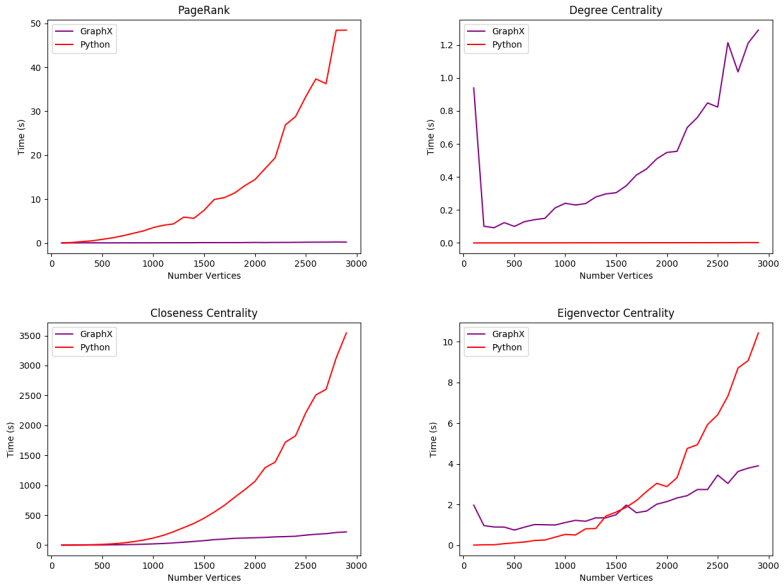

Fig. 2. As seen in the four graphs, GraphX is significantly faster than strict Python implementations for PageRank, closeness centrality, and eigenvector centrality. Python performs better on degree centrality due to the Networkx constant access to vertex degrees.

vertices will not have as great of an effect as it does in a non-distributed environment.

Comparing the runtimes of the graph centrality algorithms in GraphX to the non-distributed environment of Python, the algorithms are much faster in GraphX than in Python for most of the algorithms, as was expected. GraphX promises fast, distributed processing which is supported by these results. The only reason that Python performs better in degree centrality is due to Networkx's storage of degrees as a graph is created leading to quick access of degrees. This means that we simply need to just access this list and sort it to find the highest degree vertex. Thus, it seems that degree centrality may be best fit in a single system environment rather than a distributed system since it comes with unnecessary overhead.

GraphX does not solve the problem of robust processing for algorithms that require large amounts of bookkeeping such as closeness centrality. GraphX fails to efficiently distribute the load of the bookkeeping among the Spark framework, causing a crash at relatively small graphs. This issue could be solved with larger storage resources and perhaps a more efficient implementation. From this, we can conclude that storage capacity plays a large role in the effectiveness of certain graph centrality algorithms. If an algorithm with large bookkeeping is needed, an environment that can support large data transfers is required.

VI. FUTURE WORK

Due to time constraints, we were unable to test our algorithms on extremely large datasets. Consequently, we were unable to verify the robustness of degree centrality, eigenvector centrality or PageRank as they never crashed on the datasets we used. We want to generate extremely large graphs and find how robust these algorithms are. Additionally, we want to explore ways to make closeness centrality more efficient and avoid memory errors. Perhaps implementing our own shortest path algorithm can save some storage and computational complexity. Furthermore, we were unable to implement k-Betweenness centrality as we initially proposed due to computational errors. Like closeness centrality, it requires a good amount of storage space and computational power. With more time and better computing resources, we would like to implement k-Betweenness centrality as another centrality comparison.

ACKNOWLEDGEMENTS

We would like to thank the University of Tennessee for providing the computing resources at CloudLab, and more specifically, Dr. Qing "Charles" Cao for instruction leading toward the completion of this assignment. We owe our understanding of graph theory, and a number of non-synthetic graphs with which we could verify the correctness of our implementations to Dr. Michael Langston, also a faculty member of the University of Tennessee.

REFERENCES

[1] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: graph processing in a distributed dataflow framework. In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14). USENIX Association, Berkeley, CA, USA, 599-613.

[2] Page, L., Brin, S., Montwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999

[3] Freeman, Linton C. "Centrality in social networks conceptual clarification." Social networks 1.3 (1979): 215239.

[4] Sabidussi, G (1966). "The centrality index of a graph". Psychometrika. 31: 581603. doi:10.1007/bf02289527.

[5] M. E. J. Newman. "The mathematics of networks" (PDF).

[6] Richard von Mises and H. Pollaczek-Geiringer, Praktische Verfahren der Gleichungsauflösung, ZAMM - Zeitschrift fr Angewandte Mathematik und Mechanik 9, 152-164 (1929).

[7] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 1115, Aug 2008

[8] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8. See in particular Section 26.2, “The FloydWarshall algorithm”, pp. 558565 and Section 26.4, “A general framework for solving path problems in directed graphs”, pp. 570576.

[9] Erdős, P.; Rényi, A. (1959). “On Random Graphs. I” (PDF). Publicationes Mathematicae. 6: 290297.