Spectral Graph Analysis with Apache Spark

Davor Šutić
Faculty of Technical Sciences, University of Novi Sad
Trg Dositeja Obradovića 6,
21000 Novi Sad, Serbia
+381-69-3964860
davor.sutic@gmail.com

Ervin Varga
Faculty of Technical Sciences, University of Novi Sad
Trg Dositeja Obradovića 6,
21000 Novi Sad, Serbia
+381-64-6819076
evarga@uns.ac.rs

ABSTRACT

Graphs are the cornerstone of many algorithms pertaining to various network analyses. When the problem's dimensionality is relatively small, expressed in the number of vertices and edges of a graph, then most methods perform adequately well. As the problem size increases, more compute power is required. Distributed computing is a one viable option to address this issue, but it cannot scale indefinitely. At one point, it is necessary to turn to heuristic approaches. Spectral graph theory is an example of such approximate scheme. In this paper, we combine spectral analysis with distributed computing using Apache Spark. The paper is accompanied with a publicly available proof of concept implementation. The system was extensively performance tested, and the results show a superb fit of Apache Spark to the purpose of spectral graph analysis. Furthermore, the resulting code is straightforward thankfully to Spark's intuitive distributed programming model, and well-designed APIs.

CCS Concepts

Networks → Distributed architectures; Software → Middleware; Computing methodologies → Massively parallel and high-performance simulations

Keywords

Apache spark; Distributed computing; Graph; Scala; Spectral graph theory

1. INTRODUCTION

Graphs and the associated analysis methods are the foundation of advanced algorithms in many domains. For example, electrical grid may be represented as a graph, and most power functions are based upon operations on a graph. Graphs in real setups may become huge. These scalability issues require novel, and sometimes even approximate, approaches for dealing with an increased dimensionality. A typical case would be a graph partitioning problem, which is N-P complete. Therefore, after some point classical methods aren't applicable anymore. One possibility is to switch over to heuristic algorithms.

Spectral graph theory is a promising approximation technique, that tries to relate the eigenvalues of the matching graph's matrices (like its adjacency matrix or Laplacian matrix) with its inherent combinatorial attributes [1]. One of the main use cases of spectral analysis are graph partitioning and sparsification [4]. Despite promising expectations of this method, the accompanying implementation must be performant, possibly using distributed resources. The aim of this paper is to find a good basis for developing algorithms based upon spectral graph analysis. The solution must be straightforward to realize, and deliver expected performance.

ICoMS 2018, July 15–17, 2018, Porto, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6538-3/18/07…$15.00
DOI: https://doi.org/10.1145/3274250.3275111

Apache Spark is a very popular open-source cluster-computing framework [5] for tackling Big Data related problems. It has an intuitive programming model with comprehensive and comprehensible APIs targeted for many mainstream programming languages. Moreover, Apache Spark comes with a powerful machine learning [6] and graph analysis libraries [7]. Therefore, it was a natural choice to realize our spectral graph analysis library.

The rest of this paper is structured as follows: the second section describes the solution, the third section presents the performance testing result, and the conclusion elaborates about future work. The POC is freely available at https://bitbucket.org/sutícd/spectralgraphanalysistool.

2. SOLUTION

The proof of concept (POC) implementation extends Apache Spark with a new module, that relies on the existing MLib and GraphX components (see Figure 1). Spark's machine learning library already contains a clusterization algorithm based upon spectral analysis [8]. Nonetheless, separating spectral graph analysis inside a separate module aids reuse and keeps the architecture clean. The code is written in Scala with proper unit tests, that may serve as an additional documentation about its usage.

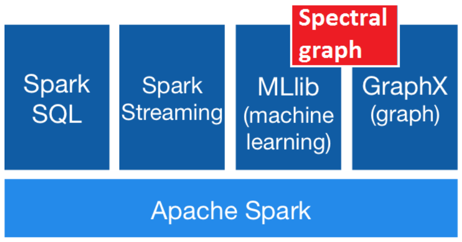


Figure 1. Extension of Apache Spark's stack with the new module for spectral graph analysis

Apache Spark was chosen as the foundation of the solution, because it already provides strong support for distributed linear algebra and graph analysis. Further, Spark is an open-source framework, natively written in Scala. So, our solution contributes a logical extension to the project.

Below is the abridged source code (imports and embedded comments are omitted) of the class implementing the core set of operations related to spectral graph analysis. The main methods are bolded, as these will be separately mentioned in the next section about performance. It is a perfect testimony how easy it is to extend Apache Spark, and rely on its fundamental abstraction called Resilient Distributed Dataset (RDD) [9] to hide complexities surrounding distributed computing.

```
class SpectralGraph[VD, ED] (graph : Graph[VD, ED]) extends PartialOrdering[SpectralGraph[VD, ED]] {

  val numberOfVertices = graph.vertices.count().toInt

  def adjacencyMatrix(): CoordinateMatrix = {
    val matrixEntriesUpper = graph.edges.map(edge => MatrixEntry(edge.srcId, edge.dstId, 1.0))
    val matrixEntriesLower = graph.edges.map(edge => MatrixEntry(edge.dstId, edge.srcId, 1.0))

    new CoordinateMatrix(matrixEntriesUpper ++ matrixEntriesLower, numberOfVertices, numberOfVertices)
  }

  def laplacianMatrix(): CoordinateMatrix = {
    val matrixEntriesUpper = graph.edges.map(edge => MatrixEntry(edge.srcId, edge.dstId, -1.0))
    val matrixEntriesLower = graph.edges.map(edge => MatrixEntry(edge.dstId, edge.srcId, -1.0))

    val vertexDegrees =
      graph
     .aggregateMessages[Int](ctx => { ctx.sendToSrc(1); ctx.sendToDst(1) }, _+_)
     .map(tuple => MatrixEntry(tuple._1, tuple._1, tuple._2))

    new CoordinateMatrix(
      matrixEntriesUpper ++ matrixEntriesLower ++ vertexDegrees, numberOfVertices, numberOfVertices)
  }

  def laplacianRayleighQuotient(vector : Vector[Double]) : Double = {
    if (vector.length != numberOfVertices) {
        sys.error("The vector's length (" + vector.length + ")"
          + "does not match the number of vertices (" + numberOfVertices + ").")
    }

    val x = laplacianQuadraticForm(vector)
    val y = vector.map(elem => elem * elem).reduce(_+_)
    x / y
  }

  def laplacianQuadraticForm(vector : Vector[Double]) : Double = {
    if (vector.length != numberOfVertices) {
        sys.error("The vector's length (" + vector.length + ")"
          + "does not match the number of vertices (" + numberOfVertices + ").")
    }

    graph.edges.map(edge => (vector(edge.srcId.toInt)
      - vector(edge.dstId.toInt))*(vector(edge.srcId.toInt) - vector(edge.dstId.toInt))).reduce(_+_)
  }

  def spectralComparison(x: SpectralGraph[VD, ED], y: SpectralGraph[VD, ED]) : (Double, Double) = {
    if (x.numberOfVertices != y.numberOfVertices) {
      sys.error("The number of vertices in the first graph (" + x.numberOfVertices + ")"
        + "does not match the number of vertices in the second graph (" + y.numberOfVertices + ").")
    }

    val vectorProbe = IndexedSeq.fill(x.numberOfVertices)(Random.nextDouble()).toVector
    val xQuadraticForm = x.laplacianQuadraticForm(vectorProbe)
    val yQuadraticForm = y.laplacianQuadraticForm(vectorProbe)

    (xQuadraticForm, yQuadraticForm)
  }

  def lteq(x: SpectralGraph[VD, ED], y: SpectralGraph[VD, ED]): Boolean = {
    val laplacianQuadraticFormValues = spectralComparison(x, y)
    laplacianQuadraticFormValues._1 <= laplacianQuadraticFormValues._2
  }

  def tryCompare(x: SpectralGraph[VD, ED], y: SpectralGraph[VD, ED]): Option[Int] = {
    val laplacianQuadraticFormValues = spectralComparison(x, y)
    Some((laplacianQuadraticFormValues._1 - laplacianQuadraticFormValues._2).toInt)
  }
```

Since this work is a logical extension of the GraphX and MLlib libraries, the input is a GraphX graph, while the outputs are either primitive types or distributed MLlib matrices. The scope of this paper are unweighted graphs. Although GraphX graphs support the notion of weighted edges, this is more a concept of tagging, as the "weight" is not restricted to numerical values and can practically be any object. At this stage of development, there is little to gain by limiting the suitable graphs to those with numerical weights. However, as more advanced algorithms are included, support for weighted graphs will be provided. The meaning of the bolded functions is enrolled below:

- `adjacencyMatrix`: adjacency matrix is the basic algebraic representation of a graph. It represents the adjacency relationship between vertices in the graph, i.e. a given matrix entry is equal to 1, if the vertices associated with the coordinates are adjacent. All other elements (including those on the diagonals) are equal to 0. Although substantial advances are being made, spectral graph theory is basically the study of the relationship between a graph and the spectrum of its adjacency matrix.

- `laplacianMatrix`: Laplacian matrix is the central matrix representation of a graph in spectral analysis. It can be instrumental in finding many useful properties and the cornerstone of prominent algorithms, e.g. the sparsest cut, minimal spanning tree, Cheeger's inequality. The construction is like that of the adjacency matrix. The differences are that adjacent vertices are represented by -1 instead of 1 and that the main diagonal contains the degrees of each individual vertex.

- `laplacianRayleighQuotient`: the eigenvectors and eigenvalues of the Laplacian matrix are important parameters of the spectral analysis. For example, the second smallest eigenvalue determines whether the graph is connected. Unfortunately, the Eigen decomposition of a given matrix is a tedious and computationally expensive process. Spark's MLlib provides an algorithm for the singular value decomposition (SVD). However, it produces good results for tall slim matrices and for identifying the largest eigenvalues. Given that the Laplacian matrices are square and that we are usually interested in the smallest eigenvalues, using the provided SVD might be limiting, especially for large graphs. Obtaining eigenvalues from the Rayleigh quotient is an optimization problem. Namely, if the vector parameter is an eigenvector of the Laplacian, the Rayleigh quotient will be equal to the corresponding eigenvalue. So, by maximizing and minimizing the Rayleigh quotient, one can systematically obtain the desired eigenvalues.

- `tryCompare`: the notion of partial order for graphs derives from a similar concept applied to symmetric matrices. The centerpiece for ordering graphs is their Laplacian quadratic form. Given any non-constant vector, if the Laplacian quadratic form of one graph is lesser or equal to the Laplacian quadratic form of the other graph, the first graph precedes the second.

3. RESULT

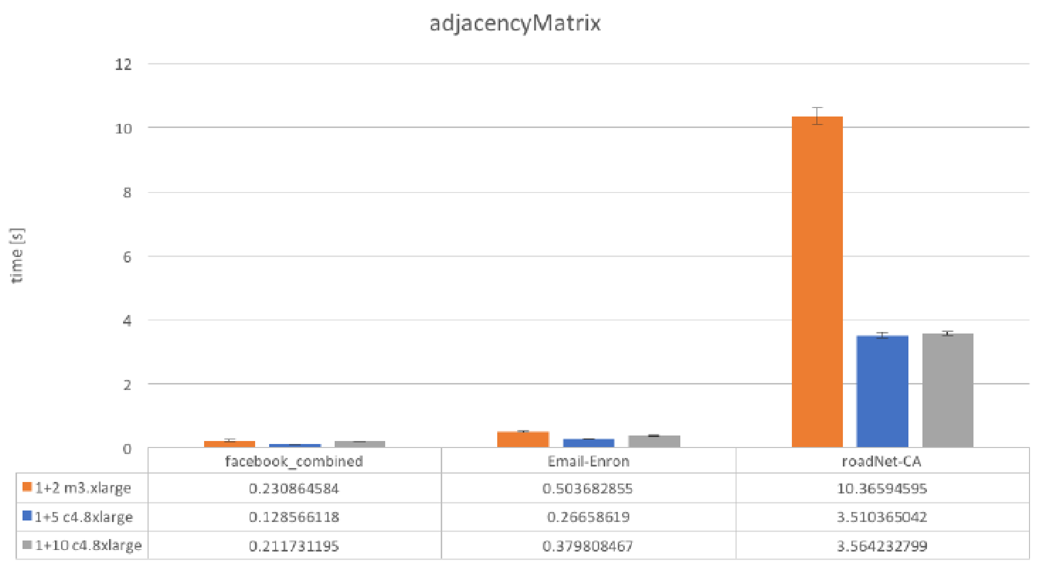
The POC was tested using 3 publicly available datasets. All of them are large enough to provide good feedback about the performance and scalability of the proposed solution.

The first one consists of 'circles' (or 'friends' lists') from Facebook [10]. It contains 4039 vertices and 88234 edges. Facebook data was collected from survey participants using a Facebook application. The dataset includes node features (profiles), circles, and ego networks. Facebook data has been anonymized by replacing the Facebook-internal ids for each user with a new value. Also, while feature vectors from this dataset have been provided, the interpretation of those features has been obscured. For instance, where the original dataset may have contained a feature "`political=Dummy Party`," the new data would simply contain "`political=Anonymized Feature 1`." Thus, using the anonymized data it is possible to determine whether two users have the same political affiliations, but not what their individual political affiliations represent.

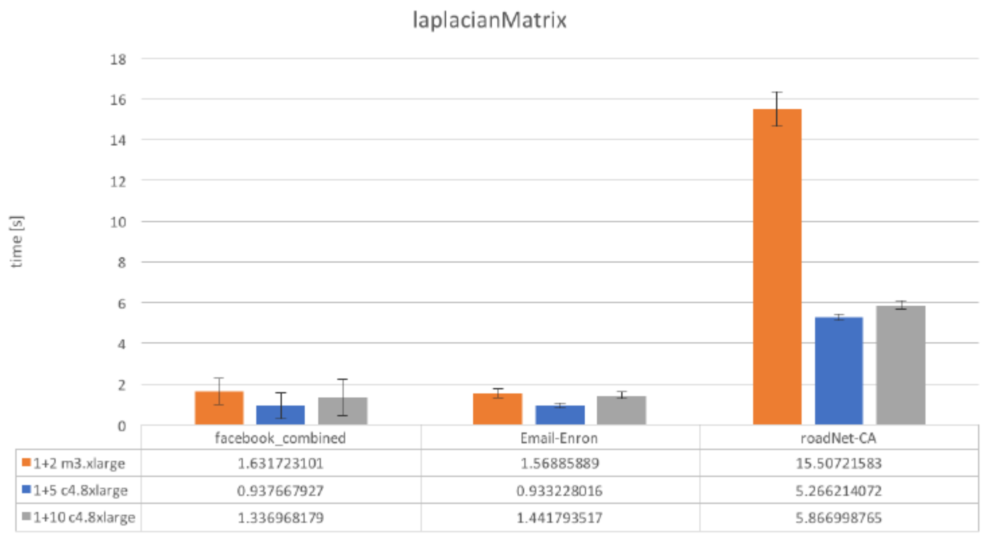
The second dataset is related to the Enron email communication network, which covers all the email communication within a dataset of around half million emails [11]. The data consists of 36692 vertices and 367662 edges. This data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation. Nodes of the network are email addresses and if an address *i* sent at least one email to address *j*, the graph contains an undirected edge from *i* to *j*. Note that non-Enron email addresses act as sinks and sources in the network as we only observe their communication with the Enron email addresses. The Enron email data was originally released by William Cohen at CMU.

Finally, the third dataset is about a road network of California [12]. Intersections and endpoints are represented by nodes; the roads connecting these intersections or road endpoints are represented by undirected edges.

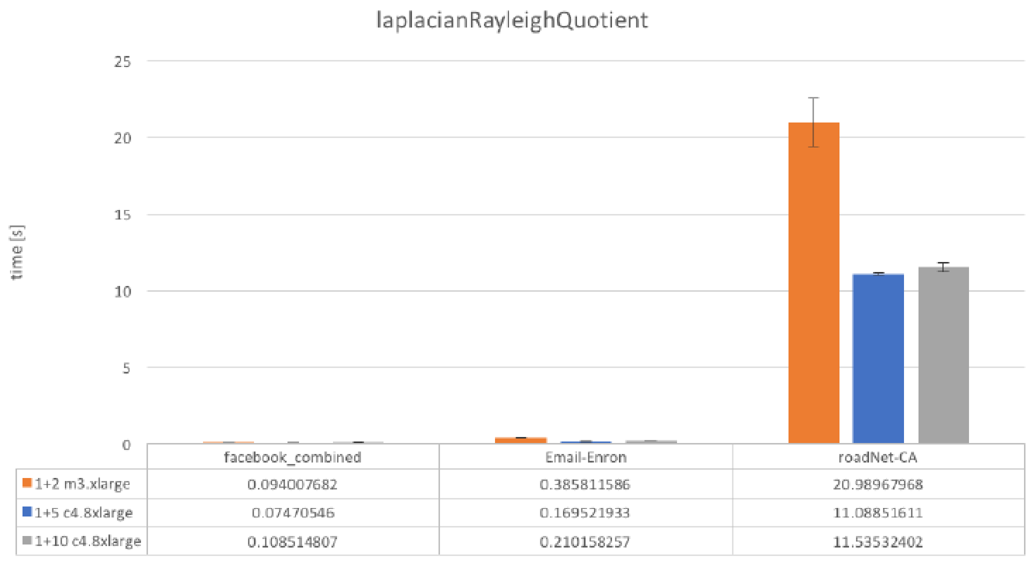
The measurements were done on 3 different AWS instance types: 1+2 m3.xlarge, 1+5 c4.8xlarge, and 1+10 c4.8xlarge. Figure 2 and 3 show the combined performance graphs for different datasets and operations (Laplacian matrix, adjacency matrix, Laplacian-Rayleigh quotient, and graph comparison). Each graph overlays average execution times per iteration for the various configurations. It is interesting to notice, that increasing the number of virtual machines (horizontal scaling) as well as choosing more powerful nodes (vertical scaling) is dependent upon the dataset. The Facebook and Enron sample doesn't show too much variance, but the road network shows a huge difference between the weakest configuration, and the other two. Moreover, there is even a negative effect for the road network case when the number of nodes is bumped from 5 to 10. This could be explained with the Amdahl's law for parallel computing. In this case, more nodes introduce extra overhead, that outweighs the benefits of elevated parallelism. Consequently, the configuration for each dataset must be separately tuned.



	facebook_combined	Email-Enron	roadNet-CA
🟧 1+2 m3.xlarge	0.230864584	0.503682855	10.36594595
🟦 1+5 c4.8xlarge	0.128566118	0.26658619	3.510365042
▨ 1+10 c4.8xlarge	0.211731195	0.379808467	3.564232799

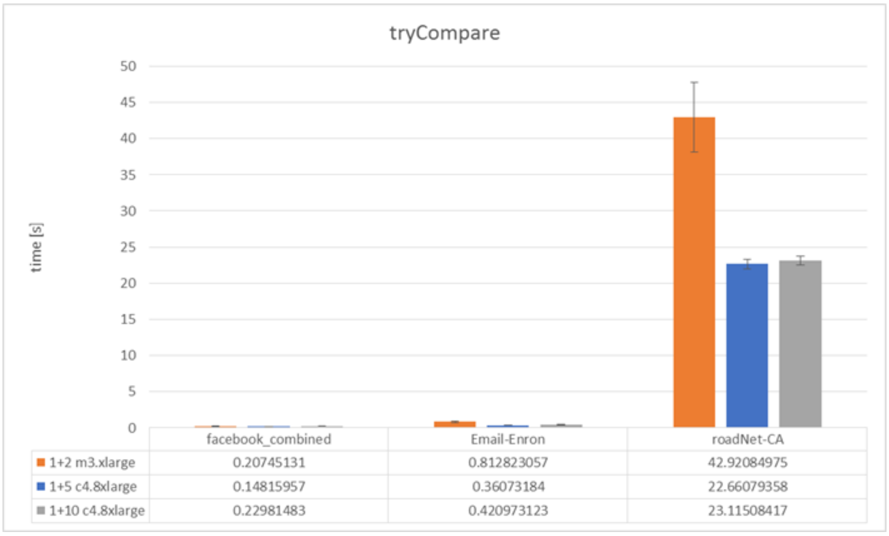


	facebook_combined	Email-Enron	roadNet-CA
🟧 1+2 m3.xlarge	1.631723101	1.56885889	15.50721583
🟦 1+5 c4.8xlarge	0.937667927	0.933228016	5.266214072
▨ 1+10 c4.8xlarge	1.336968179	1.441793517	5.866998765



	facebook_combined	Email-Enron	roadNet-CA
🟧 1+2 m3.xlarge	0.094007682	0.385811586	20.98967968
🟦 1+5 c4.8xlarge	0.07470546	0.169521933	11.08851611
▨ 1+10 c4.8xlarge	0.108514807	0.210158257	11.53532402

Figure 2. Performance figures for graph matrix operations. All execution time measures are in seconds



	facebook_combined	Email-Enron	roadNet-CA
🟧 1+2 m3.xlarge	0.20745131	0.812823057	42.92084975
🟦 1+5 c4.8xlarge	0.14815957	0.36073184	22.66079358
⬜ 1+10 c4.8xlarge	0.22981483	0.420973123	23.11508417

Figure 3. Performance figure of the graph comparison routine. All execution time measures are in seconds

4. CONCLUSION

This paper has presented the way to use Apache Spark to implement a module for spectral graph analysis. Based upon the performance results we may conclude, that the approach is feasible. Moreover, the source code shows how comfortable it is to do distributed computing using Spark comprehensible APIs.

The next steps would be to augment the POC code base with more advanced capabilities. Furthermore, we investigate ways to transform current power analysis methods to utilize spectral graph analysis.

5. REFERENCES

[1] Dan Spielman, "Spectral Graph Theory, Fall 2015," http://www.cs.yale.edu/homes/spielman/561/, Accessed 01 February 2018

[2] Bogdan Nica, A brief introduction to Spectral Graph Theory, https://arxiv.org/abs/1609.08072, Accessed 01 February 2018

[3] Luca Trevisan, "Spectral Graph Theory I: Introduction to Spectral Graph Theory," https://simons.berkeley.edu/talks/luca-trevisan-2014-08-26a, Accessed 01 February 2018

[4] J. Batson, D.A. Spielman, N. Srivastava, S.H. Teng, " Spectral Sparsification of Graphs: Theory and Algorithms," Communications of the ACM, Vol. 56 No. 8, Pages 87-94, 2013

[5] B. Chambers and M. Zaharia, Spark: The Definitive Guide, O'Reilly Media, Inc., 2018

[6] R.B. Zadeh, et al. "Matrix computations and optimization in Apache Spark," Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016

[7] R.S. Xin, et al. "GraphX: A resilient distributed graph system on Spark," First International Workshop on Graph Data Management Experiences and Systems, ACM, 2013

[8] Apache Spark , "Power iteration clustering (PIC)," http://spark.apache.org/docs/latest/mllib-clustering.html#power-iteration-clustering-pic, Accessed 01 February 2018

[9] M. Zaharia, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012

[10] J. McAuley and J. Leskovec, "Learning to Discover Social Circles in Ego Networks," NIPS, 2012

[11] B. Klimmt and Y. Yang, "Introducing the Enron corpus," CEAS 2004 - First Conference on Email and Anti-Spam, 2004

[12] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," Internet Mathematics, 6(1) 29--123, 2009