Enabling Scalable Social Group Analytics via Hypergraph Analysis Systems

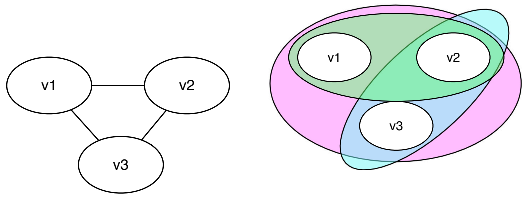
Benjamin Heintz and Abhishek Chandra
Department of Computer Science & Engineering
University of Minnesota

Abstract

With the rapid growth of large online social networks, the ability to analyze large-scale social structure and behavior has become critically important, and this has led to the development of several scalable graph processing systems. In reality, social interaction takes place not just between pairs of individuals as in the common *graph* model, but rather in the context of multi-user *groups*. Research has shown that such group dynamics can be better modeled through *hypergraphs*: a generalization of graphs. There are not yet, however, scalable systems to support hypergraph computation, and several challenges and opportunities arise in their design and implementation. In this paper, we present an initial attempt at building a scalable hypergraph analysis framework based on the GraphX/Spark framework. We use this prototype to examine several programmability and implementation issues through experiments with two real-world datasets on a 6-node cluster.

1 Introduction

The advent of online social networks and communities such as Facebook and Twitter has led to unprecedented growth in user interactions (such as "likes", comments, photo sharing, and tweets), and collaborative activities (such as video/document editing and shared quests in multi-player games). This has resulted in massive amounts of rich data that can be analyzed to better understand user behavior, information flow, and social dynamics. The traditional way to study social networks is by modeling them as *graphs*, where each vertex represents an entity (e.g., a user) and each edge represents the relation or interaction between two entities (e.g., friendship). Several graph analytics frameworks [11, 19, 20] have been introduced to scale out the computation on massive graphs comprising millions or billions of vertices and edges.



(a) Graph (b) Hypergraph

Figure 1: A hypergraph can model groups unambiguously compared to a simple graph. Here, we have three groups: two consisting of pairwise interactions, and one including all three vertices.

While graph analytics has enabled a better understanding of social interactions between users, there is a growing interest [16] to study a *group* or team of individuals as a distinct entity on its own. A group is an underlying basis for many social interactions and collaborations, such as users on Facebook commenting on a common event of interest or a team of programmers working together on a software project. In these cases, the interactions between users is driven primarily as a result of their group membership, and not as individual pairwise interactions happening in a vacuum. Further, the dynamics of many such systems may also be driven through group-level events, such as users joining or leaving groups, or finding others based on group characteristics (e.g., common interest). Since such group-based phenomena involve multi-user interactions, it has been shown that many natural phenomena can be better modeled using hypergraphs than by using graphs [7].

Formally, a *hypergraph* is a generalization of a graph, and is defined as a tuple $H = (V, E)$, where V is the set of entities, called *vertices*, in the network, and E is the set of subsets of V, called *hyperedges*, representing relations between one or more entities [1]. As illustrated in Figure 1, a hypergraph can model groups unambiguously compared to a graph[1]. Recent work [25] has shown that hypergraph models can also achieve a significant improvement in modeling accuracy compared to graph-based models. While hypergraph algorithms have received much less attention compared to graph algorithms, there has been work on developing hypergraph counterparts for problems such as centrality estimation [3], shortest path computation [9], and others. These algorithms will likely receive more attention as the study of group dynamics matures.

From a systems standpoint, a key challenge in enabling hypergraph analysis is the massive scale of the underlying data (millions or billions of vertices and hyperedges). As a result, similar to a graph analysis system, a *hypergraph analysis system* must also be scalable, both in terms of memory and storage utilization to support large data, as well as by enabling distributed computation across multiple CPUs and nodes for increased parallelism. The focus of this paper is on such systems-level challenges.

In this paper, we present an initial attempt at building a scalable hypergraph analysis system based on the GraphX framework [11] in Apache Spark [26] and use this prototype to illustrate the various challenges and opportunities involved. We propose a concise but expressive hypergraph API which we use to implement hypergraph extensions of the popular PageRank [21] algorithm. We then discuss a number of alternative techniques for implementation issues related to data representation and execution. We also examine the impact of various factors on the tradeoffs of selecting different alternatives. Some of the key factors include hypergraph characteristics, algorithm behavior, and the mechanisms provided by (as well as the efficiency of) the underlying computational platform. Our analysis is based on preliminary empirical results obtained by running the algorithms/prototype on a 6-node cluster with two publicly available datasets obtained from DBLP and Friendster.

2 A Hypergraph PageRank Algorithm

We begin by presenting an exemplar hypergraph algorithm based on PageRank [21], a widely used algorithm in graph analytics to determine the relative importance of different vertices in a graph. It is used in a variety of applications, such as search, link prediction, and recommendation systems.

We can imagine extending PageRank to the hypergraph context in many ways. First, it is possible to compute the PageRank for vertices based on their membership in different hyperedges in the hypergraph. In a social context, this would correspond to determining the importance of a user based on her group memberships (e.g., a user might be considered more influential if she is part of an exclusive club).

At the same time, it is possible to compute the PageRank for hyperedges based on the vertices they contain. This corresponds to estimating the importance of groups based on their members (e.g., a group with Fortune 500 CEOs is likely to be highly influential). This extension also illustrates the fact that hyperedges can be considered as first-class entities associated with similar state and computational functions as vertices in typical graph computation.

This elevation of hyperedges to first-class status suggests a further extension to PageRank: we can compute additional attributes for each hyperedge using arbitrary functions of its member vertices. For instance, we can use an entropy function to determine the uniformity of each hyperedge; i.e., the extent to which its members contribute equally to its importance.

3 A Hypergraph API

These example PageRank algorithms suggest the following key requirements of an API for a hypergraph analysis system. First, it must support hyperedges as first-class entities on par with vertices, with their own state and compute functions. Secondly, the computation can be logically carried out in a series of alternate steps involving computation on vertices and hyperedges respectively. Guided by these requirements, we propose a concise and expressive API for hypergraph computation (see Listing 1)[2]. The key modeling abstraction is the `HyperGraph`, which is parameterized on the hypervertex[3] and hyperedge attribute types[4].

The core computational method, `compute`, provides an iterative computational model similar to Pregel [19]. Using this method, users can easily express computation that proceeds iteratively in a series of alternate supersteps, during which hypervertices (resp., hyperedges) update their state and compute new messages, which are delivered to hyperedges (resp., hypervertices). In this model, hyperedges are clearly elevated to first-class status; they can maintain their state, carry out computation, and send messages just as hypervertices do.

To use the `compute` method to orchestrate their iterative computation, users encode their hypervertex (resp., hyperedge) behavior in the form of a `Program` comprising a `Procedure` for consuming incoming messages, updating state, and producing outgoing messages, as well as a `MessageCombiner` for aggregating messages destined to a common hyperedge (resp., hypervertex). The `Context` provides methods that enable the `Procedure` to update hypervertex (resp., hyperedge) state, and to send messages to neighboring hyperedges (resp., hypervertices).

Using this API, we are able to implement a hypergraph PageRank algorithm which computes ranks for hyperver-

```
trait HyperGraph[HVD, HED] {
  def compute[ToE, ToV](
    maxIters: Int,
    initialMsg: ToV,
    hvProgram: Program[HVD, ToV, ToE],
    heProgram: Program[HED, ToE, ToV])
    : HyperGraph[HVD, HED]
}

object HyperGraph {
  trait Program[A, InMsg, OutMsg] {
    def messageCombiner: MessageCombiner[OutMsg]
    def procedure: Procedure[A, InMsg, OutMsg]
  }

  type MessageCombiner[Msg] = (Msg, Msg) => Msg

  type Procedure[A, InMsg, OutMsg] =
    (Int, NodeId, A, InMsg, Context[A, OutMsg]) => Unit

  trait Context[A, OutMsg] {
    def become(attr: A): Unit
    def send(msgF: NodeId => OutMsg,
             to: Recipients): Unit
  }
}
```

Listing 1: Key abstractions from our hypergraph API (expressed in Scala).

tices using only 19 lines of code. Further computing hyperedge ranks requires only a single line of additional code. An even richer version which also computes the entropy of each hyperedge requires fewer than 25 lines of code.

Note that a user might be able to implement the same algorithms without using the hypergraph abstraction explicitly or using our API. For instance, for the simplest PageRank variant, where we compute only vertex ranks, it is possible to define a transformation function from the input hypergraph to a weighted graph such that an existing graph PageRank algorithm yields identical results. We refer to this weighted graph as the *one-mode projection* of the original hypergraph. Such a transformation, however, is highly non-trivial requiring significant developer time and effort, and the resulting graph consumes significantly more space than a hypergraph representation (see Section 5). Further, this approach does not apply to the richer algorithm variants, as they require modeling hyperedges as first-class entities. An alternative approach could be to use an affiliation network model explicitly capturing group affiliations of users [8]. However, such an approach is also tedious and error-prone, and many existing graph processing abstractions (such as GraphX's `Pregel` implementation) cannot be applied without modification. Yet another disadvantage of such approaches is that, by disguising hypergraphs as graphs, they preclude any hypergraph-aware optimization at the underlying system level.

4 Implementation Issues

Next, we explore two key issues involved in the implementation of our hypergraph computing API: how to represent the hypergraph at the system level, and how to partition this hypergraph for distributed computation.

4.1 Representation

The choice of the underlying platform has a major impact on how we represent hypergraphs at the system level. If using an underlying graph processing platform, we can represent the hypergraph as a bipartite graph, where one partition comprises exclusively hypervertices, and the other exclusively hyperedges, with low-level graph edges connecting hyperedges to their constituent hypervertices. If the underlying platform provides a more flexible *multigraph* abstraction, allowing multiple parallel edges between pairs of vertices, then we can represent hypervertices using graph vertices, and hyperedges using labeled graph edges: any two vertices that belong to a common hyperedge h are connected by an edge with label h. If we instead implement our API directly on a general distributed computing framework such as Hadoop or Spark, then there is much more flexibility of representation.

While these alternative representations are all equally valid, each has its strengths and weaknesses. One key advantage of the bipartite graph representation is its portability: it can be implemented on any graph computing system. For example, the underlying dataflow for our `compute` method is a natural extension of the dataflow in the existing GraphX `Pregel` implementation. However, a straightforward transformation that does not distinguish between the two distinct types of entities—hyperedges and hypervertices—can result in suboptimal performance. Hyperedges and hypervertices may have significantly different characteristics including attribute sizes, degree/cardinality distribution, and behavior of their respective `Programs`, resulting in poor I/O performance and load imbalance. As a result, the underlying system must use mechanisms and optimizations that are aware of these differences.

A multigraph representation, on the other hand, represents hypervertices and hyperedges using distinct underlying entities (viz., vertices and edges respectively), and as a result avoids these potential performance pitfalls. One limitation, however, is that only a few existing graph computing platforms (such as GraphX) provide a multigraph abstraction. Additionally, mapping our hypergraph API into this representation would require a completely different dataflow than is provided directly by existing systems. Compared to the bipartite representation, this representation might also be more storage-intensive, especially if the hypergraph contains many large hyperedges. This problem could be addressed through intelligent transformations to the underlying multigraph that might yield interesting performance trade-offs.

Finally, implementing a hypergraph directly on top of a general distributed computing platform such as Hadoop or Spark opens up an essentially limitless range of optimization opportunities, but it fails to take advantage of the many recent advances in graph computing systems. Further, graphs remain the right tool for a broad range of problems, and implementing a hypergraph API atop a graph system makes it much easier for application developers to mix and match graph and hypergraph computing as their applications require.

4.2 Partitioning

To scale to large hypergraphs, it is essential to distribute computation across multiple nodes. The decision of how to partition the underlying representation can have a significant impact on performance, in terms of both computation load balance, as well as network I/O. Graph computing systems such as Pregel partition graphs by assigning each vertex to a unique machine, and thus “cutting” edges; the volume of network I/O is then driven directly by the number of cut edges. PowerGraph [10] uses an alternative approach of assigning each *edge* to a unique machine and instead cutting vertices to achieve better load balance for graphs with highly skewed vertex degree distributions. PowerLyra [5] uses a hybrid approach that distinguishes between high- and low-degree vertices.

Similarly for hypergraphs, the space of options is broader than the two extremes of vertex-cut vs. edge-cut. Using a bipartite representation, for example, a graph-level vertex-cut partitioning approach would effectively cut *both* hypervertices and hyperedges. As a concrete example, GraphX allows users to specify a `PartitionStrategy`, which assigns each directed edge to a partition based on its source id, destination id, or both. In our implementation of the bipartite representation, edges are directed from hyperedges to hypervertices. As a result, if we partition exclusively by source (resp., destination) id, we are effectively cutting only hypervertices (resp., hyperedges). If we partition by both, then we cut both hyperedges and hypervertices, and the distribution of these cuts is driven by the details of our `PartitionStrategy`.

An interesting possibility is to base the partitioning strategy on the relative size of hyperedge and hypervertex attributes or degree/cardinality distributions. For example, if hypervertex attributes are much larger than hyperedge attributes, then we can reduce network I/O by cutting hyperedges rather than hypervertices. In an iterative computing model like ours, it may even be worthwhile to adapt partitioning decisions across iterations, dynamically repartitioning the hypergraph representation based on run-time performance metrics.

5 Experimental Evaluation

To understand the performance implications of various factors such as data and algorithm characteristics and partitioning strategies, we implement a proof-of-concept of our API using the GraphX framework in Apache Spark, and carry out experiments on a shared 6-node cluster consisting of machines with dual 6-core processors, 24GB RAM and 2TB disk. In our experiments, we use a bipartite graph as the low-level representation of a hypergraph. As inputs for our experiments, we use the two publicly available datasets described in Table 1.

The DBLP dataset describes more than one million publications, from which we use authorship information to build a hypergraph model where vertices represent authors and hyperedges represent collaborations between authors. The full DBLP dataset yields a hypergraph with roughly one million vertices and hyperedges, and its bipartite graph representation contains roughly 2.8 million edges. A one-mode projection that uses an edge between two authors to represent their coauthorship would contain more than 21 million edges.

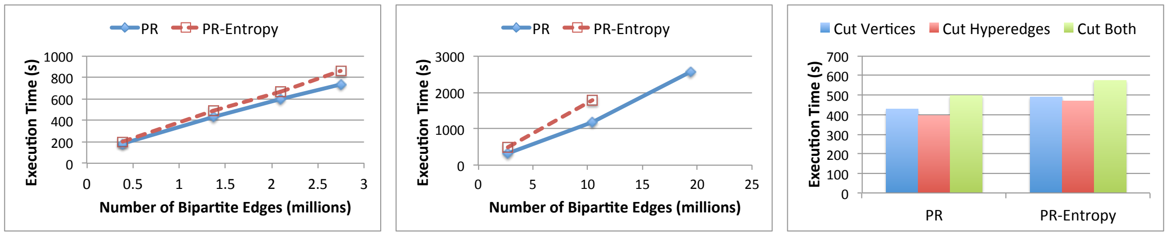
The Friendster dataset represents users of the Friendster social networking site. In our hypergraph model, vertices represent individual users, and hyperedges represent user-defined communities. Because membership in these communities does not require the same commitment as collaborating on a publication, this dataset has very different characteristics from the DBLP dataset. For one, it is much larger, containing nearly eight million vertices and more than 1.6 million hyperedges. A bipartite representation contains more than 23 million edges. A one-mode projection of this dataset is so large that we are unable to materialize it on our small cluster. A conservative lower bound on its size, computed by restricting our model to hyperedges with cardinality of 1,000 or greater, is in excess of 15 billion, or roughly three orders of magnitude larger than for the bipartite version. [5]

Using these datasets, we run two versions of our hypergraph PageRank algorithms for 30 iterations. The PR algorithm computes ranks for both vertices and hyperedges, and the PR-Entropy algorithm also computes the entropy of each hyperedge. The key difference is that, in the PR-Entropy version, in order to compute entropy of each hyperedge, messages from vertices to hyperedges must be concatenated rather than summed, leading to a higher volume of data movement.

Figures 2(a) and 2(b) show the execution time as we process increasingly large subsets of the DBLP and Friendster datasets, respectively. For DBLP, we generate subsets by filtering away publications prior to a threshold date, whereas for Friendster, we generate subsets by filtering away hyperedges with cardinality below a threshold value. For DBLP, we see that execution time grows

Table 1: Datasets used in our experiments.

Dataset	# Vertices	# Hyperedges	Max. Deg.	Max. Card.	# Bip. Edges	# 1-mode Edges	Max. 1-mode Deg.
DBLP	952,115	916,947	369	2,767	2,768,930	21,592,883	11,466
Friendster	7,944,949	1,620,991	1,700	9,299	23,479,217	$>$ *15.1 billion*	$>$ *843,000*



(a) DBLP (b) Friendster (c) Partitioning Strategies (DBLP)

Figure 2: Execution time for 30 iterations of our hypergraph PageRank algorithms.

approximately linearly with the number of edges in the underlying bipartite representation. The difference in algorithm characteristics manifests as a slightly higher runtime for the PR-Entropy algorithm due to its higher volume of communication. For the Friendster dataset, we run into scalability limits due to limited disk space on our cluster nodes, as Spark relies on disk to perform external sorting during the join operations that implement data movement between hyperedges and vertices. Even for the full Friendster dataset, however, we are able to process at least 20 PageRank iterations using our hypergraph algorithms, while a one-mode projection alternative is too large to even materialize, let alone use as input for iterative computation. As expected, we reach this limit at smaller input sizes for the PR-Entropy algorithm, due to its higher volume of communication.

Figure 2(c) illustrates the impact of partitioning strategies on execution time when processing the DBLP dataset (subset including publications since 2010). We see that, for both algorithm variants, there is a slight advantage to cutting hyperedges as opposed to cutting vertices, and that both of these strategies outperform a strategy that cuts both vertices and hyperedges.

6 Related Work

There has been an explosion of graph computing systems in recent years [4, 6, 15, 17, 18, 19, 22, 24], and along with them, a great deal of work on performance evaluation and optimization [10, 12, 23, 27]. In order to take advantage of these recent advances, and to ultimately produce a system that users can effectively integrate into their broader workflows, we have biased our study towards the opportunities and challenges of building on top of such systems rather than starting from scratch. In prior work [13], we have explored other issues surrounding hypergraph computing, such as application-level modeling considerations and the need for characterization of real-world hypergraphs. In this paper, on the other hand, we have proposed a concrete programming interface and explored some of the specific challenges that arise when implementing this interface on existing systems. Hypergraphs have been studied for decades [1, 2] and have been applied in diverse applications such as VLSI design [14]. Our hope is that scalable hypergraph computing *systems* will enable and encourage the development of novel and useful hypergraph *algorithms*.

7 Conclusion

The rapid growth in large online social networks has highlighted the need to analyze social structure and behavior at a massive scale. This has led to the development of scalable graph processing systems, but because social interaction takes place not just between pairs of individuals, but within *groups*, we have argued that hypergraphs are a more appropriate model for many applications. To enable users to work at this higher level of abstraction, we need scalable hypergraph computing systems, and implementing such systems presents interesting challenges and opportunities. In this paper, we have described our first steps toward building a scalable hypergraph analysis framework based on the GraphX framework in Apache Spark, and we have explored several implementation issues through experiments with two real-world datasets on a 6-node cluster.

8 Acknowledgments

The authors would like to acknowledge NSF Grant IIS-1422802, which supported this research. We further thank Jaideep Srivastava for initial discussions on hypergraph computing, and Shivangi Singh and Janani Thirunavukkarasu for their contributions to our proposed API and initial prototype.

9 Discussion

Feedback: While feedback on higher-level issues related to algorithms and applications would be welcome, we expect to get feedback primarily on the systems-level issues given the focus and expertise of participants at this workshop. For instance, what are the pros and cons of leveraging an existing graph processing framework (such as GraphX or GraphLab) vs. a more generic distributed processing framework (such as Spark or Hadoop)? Would a clean-slate design be better? What other kinds of underlying representations or partitioning heuristics would be useful to consider? What kind of scheduling and communication primitives or policies would be useful to support in such a system?

Controversy: We are examining a radically different way of analyzing social interaction, and proposing the development of a new kind of computational framework to support it. It might be controversial whether there is a need for such a radical approach in the first place, and whether a new computational model is required.

Discussion: This paper is likely to generate significant discussion regarding the differences and similarities between hypergraph and graph processing systems, as well as the related systems-level issues (both that can be applied from prior work, and those that would have to be invented).

Open issues: We do not address a number of application modeling and algorithm-level issues related to use of hypergraphs in applications. We also do not address programming model-level issues comprehensively (e.g., whether it is necessary to support asynchronous computation) though we propose an initial API.

Failure: This entire idea might fall apart if the algorithms community and more broadly, application developers, do not make sufficient progress in developing new hypergraph algorithms or applying them to interesting real-world phenomena.

References

[1] BERGE, C. *Graphs and hypergraphs*, vol. 6. Elsevier, 1976.

[2] BERGE, C. *Hypergraphs: combinatorics of finite sets*. Elsevier Science Publishers B. V., 1989.

[3] BONACICH, P., HOLDREN, A. C., AND JOHNSTON, M. Hyperedges and multidimensional centrality. *Social Networkrks 26*, 3 (2004), 189 – 203.

[4] BU, Y., BORKAR, V., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow. 8*, 2 (Oct. 2014), 161–172.

[5] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. of EuroSys* (2015), pp. 1:1–1:15.

[6] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proc. of EuroSys* (2012), ACM, pp. 85–98.

[7] ESTRADA, E., AND RODRIGUEZ-VELAZQUEZ, J. Complex networks as hypergraphs. *Arxiv preprint physics/0505137* (2005).

[8] FAUST, K. Centrality in affiliation networks. *Social networks 19*, 2 (1997), 157–191.

[9] GAO, J., ZHAO, Q., REN, W., SWAMI, A., RAMANATHAN, R., AND BAR-NOY, A. Dynamic shortest path algorithms for hypergraphs. *Arxiv preprint arXiv:1202.0082* (2012).

[10] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. of OSDI* (2012), pp. 17–30.

[11] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *Proc. of OSDI* (2014), pp. 599–613.

[12] HAN, M., DAUDJEE, K., AMMAR, K., ÖZSU, M. T., WANG, X., AND JIN, T. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow. 7*, 12 (Aug. 2014), 1047–1058.

[13] HEINTZ, B., AND CHANDRA, A. Beyond graphs: Toward scalable hypergraph analysis systems. *SIGMETRICS Perform. Eval. Rev. 41*, 4 (Apr. 2014), 94–97.

[14] KARYPIS, G., AGGARWAL, R., KUMAR, V., AND SHEKHAR, S. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on VLSI Systems 7*, 1 (March 1999), 69–79.

[15] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *Proc. of OSDI* (2012).

[16] LAZER, D., ET AL. Computational social science. *Science 323*, 5915 (2009), 721–723.

[17] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[18] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. GraphLab: A new parallel framework for machine learning. In *Proc. of the Conf. on Uncertainty in Artificial Intelligence (UAI)* (July 2010).

[19] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proc. of the SIGMOD International Conference on Management of Data* (2010), pp. 135–146.

[20] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proc. of SOSP* (2013), pp. 456–471.

[21] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab, 1999.

[22] SALIHOGLU, S., AND WIDOM, J. GPS: A graph processing system. In *Proc. of the 25th International Conference on Scientific and Statistical Database Management* (2013), SSDBM, pp. 22:1–22:12.

[23] SALIHOGLU, S., AND WIDOM, J. Optimizing graph algorithms on pregel-like systems. *Proc. VLDB Endow. 7*, 7 (Mar. 2014), 577–588.

[24] SATISH, N., SUNDARAM, N., PATWARY, M. M. A., SEO, J., PARK, J., HASSAAN, M. A., SENGUPTA, S., YIN, Z., AND DUBEY, P. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), ACM, pp. 979–990.

[25] SHARMA, A., CHANDRA, A., AND SRIVASTAVA, J. Predicting multi-actor collaborations using hypergraphs. In *arXiv* (2014).

[26] ZAHARIA, M., ET AL. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI* (2012).

[27] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. PrIter: A distributed framework for prioritized iterative computations. In *Proc. of SOCC* (2011), ACM, pp. 13:1–13:14.

Notes

[1]In this paper, we use "graph" to refer to a simple or dyadic graph.

[2]Scala `traits` are analogous to Java `interfaces`, and the `object` keyword here is used to define a module namespace.

[3]To avoid ambiguity, we use "hypervertex" to denote a vertex within a hypergraph.

[4]While not shown here, our interface provides a number of other methods such as those for accessing hypervertex and hyperedge attributes and for transforming the hypergraph in different ways.

[5] An interesting note is that we carried out this computation by using our hypergraph `compute` method to compute the approximate set (using a HyperLogLog) of vertices adjacent to each vertex.