Study on Hypergraphs for Large Networks

Thomas Woods, Anirudh Narayanan, Abdullah Al Raqibul Islam

1. Introduction

Hypergraphs are a useful data representation when considering group interactions. Consider a authorship network where a group of authors - i.e., Author A, Author B, and Author C who have co authored in a single publication. Representing this information using traditional graph edges where each edge connects two authors to show the co-authorship relationships bring two issues. First, it introduces misleading information where we can not differentiate the case of a single publication by three authors; and three authors co-authored three publications among them (i.e. Publication one co-authored by Author A and Author B; Publication two co-authored by Author B and Author C; Publication three co-authored by Author A and Author C). The second implication of representing hypergraph using traditional graph representation is the inflation of data in the graph storage. For example in our previous co-authorship network, we will need to add $N*(N-1)$ edges to represent the co-authorship among N authors for a single publication. By learning how to encode data into these structures and how to manipulate them to extract meaningful data, we may be able to gain insights about the data that would have not been possible before.

One of the purposes of this study is to understand the hypergraphs and find their adaptability in large networks. We have explored several representations of hypergraphs and understand pros and cons of each. In pursuing this direction, we made several exploratory searches on the existing libraries [7], [8] that work on this direction. These libraries allow to benchmark different algorithms using different encodings and implementations of hypergraph. But these libraries have flaws- especially when the data sources are large and unyielding like Tweets, mail, social network interactions, or financial data.

In rescuing, researchers made several attempts by proposing parallel hypergraph algorithms [9] which works well on large datasets on a single machine. There is a growing need to process hypergraph algorithms in distributed setup to tackle the challenges of the data beyond the memory limit. In our work, we port the existing parallel hypergraph algorithm implementations to a distributed environment using Apache Spark. We are building a Python module to explore the hypergraph page rank implementation in a distributed setup using Apache Spark. Existing research [10] has targeted the graph partitioning challenge in distributed hypergraph processing. We are also planning to explore the implications of hypergraph partitioning research for distributed computation of hypergraph algorithms. Our goal in this direction is to understand the existing work, explore novel approaches, and realize a solution in Python that utilizes Apache Spark for parallel computing.

2. Related Work

We explored research on hypergraph representation, configuration models, and parallel hypergraph algorithms. There are different encoding types of hypergraphs including Adjacency Matrix, Incidence Matrix, Lower-order representations of higher-order hypergraphs, etc. [1] gives an overview of the research, applications, issues, and advantages of using hypergraphs. The following figure from [1] gives a very concise example of the representations of higher-order interactions that we usually observe in hypergraphs-

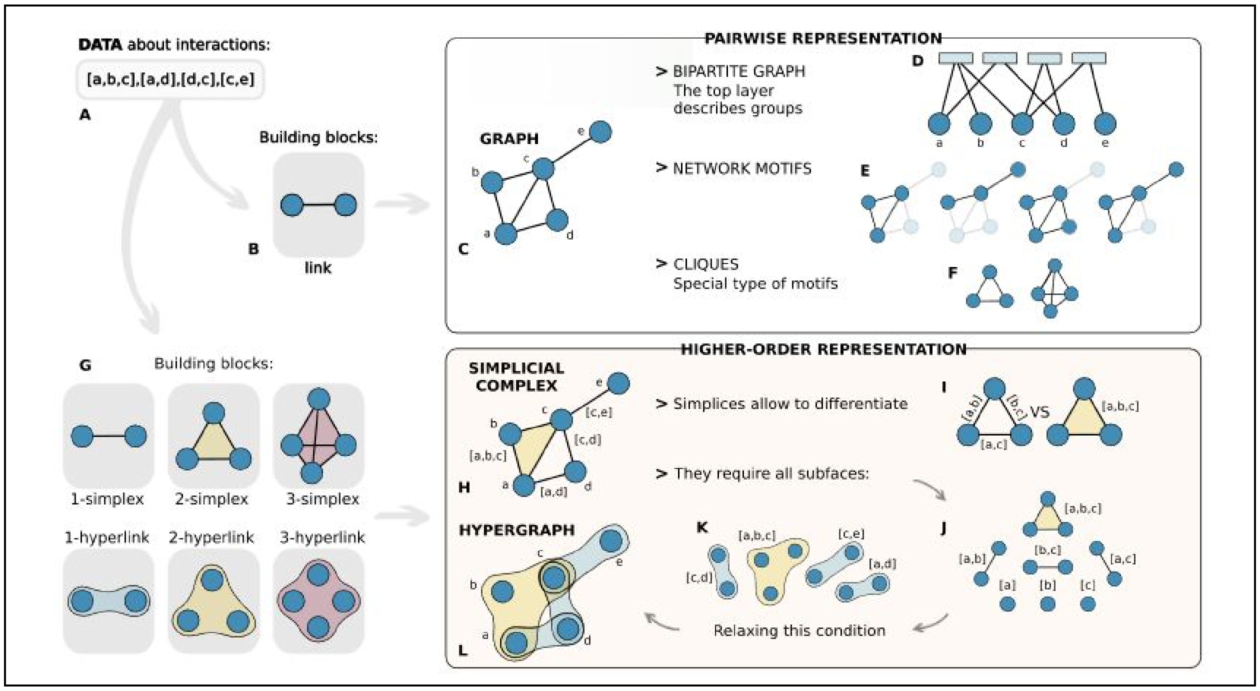


Figure 1 - Representations of higher-order interactions from Battiston et. al [1]

In this image, we can see how a set of interactions of heterogeneous order (A) can be converted into higher-order representations as a hypergraph. Hypergraphs can be built from lower-order hyperedges, all the way down to regular nodes and edges. Using only low-order blocks, the set of interactions can be described in the simplest way by using a graph showing in (C). Alternatively, interactions can be encoded as nodes in one partition of a bipartite graph, where the other partition contains the interaction vertices (D). Other examples of high-order coordinated patterns are in (E); i.e., using motifs, small subgraphs with specific connectivity structures, etc. Cliques are a special type motif and are particularly popular as they represent the densest subgraphs, which is equivalent to the higher-order bricks (F).

All these representations hide the information that was present in the original interaction data (A). A probable solution is to consider the higher-order building blocks explicitly, in the form of simplices and hyperedges (G). Collection of simplices form simplicial complexes (H). But given a simplex, simplicial complexes require the presence of all possible subsimplices (J), which can bring memory overhead in some systems.

The following figure shows in more detail a multipartite representation of a hyperedge to drive home the point of hyperedges being composed of lower-order hyperedges, all the way down to regular graphs. A simplicial complex (A) is constructed by the list of composing simplices. The structure of the simplices can be described as a graph (B), where nodes correspond to simplices and edges the inclusions.

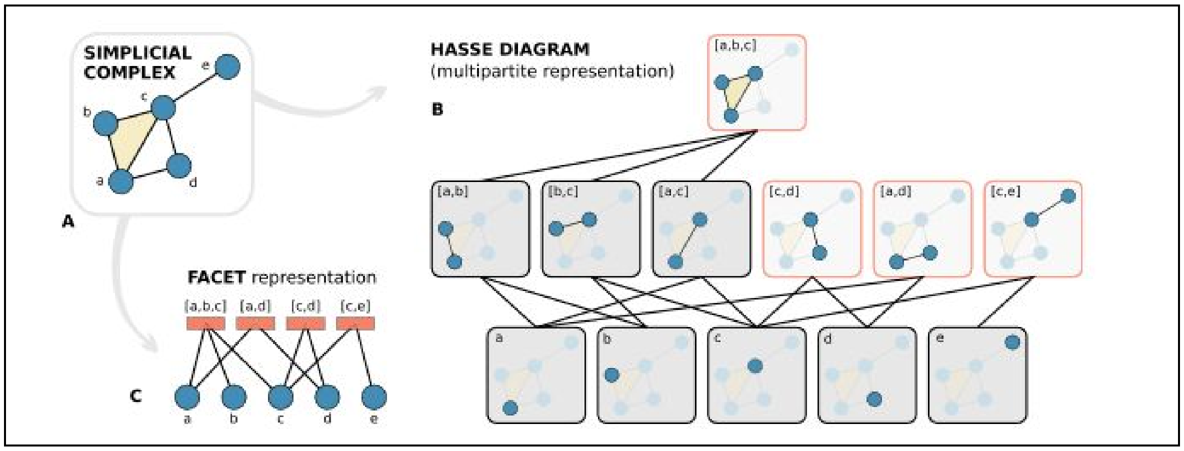


Figure 2 - Relations among different hypergraph representations from Battiston et. al [1]

Finally, these example hypergraphs are shown with their corresponding encodings, specifically incidence and adjacency matrices. One big takeaway was that, as the maximum edge order grew, so did the size of the matrix, by a factor of the number of nodes. This highlights why the curse of dimensionality causes these representations to be infeasible on large problems.

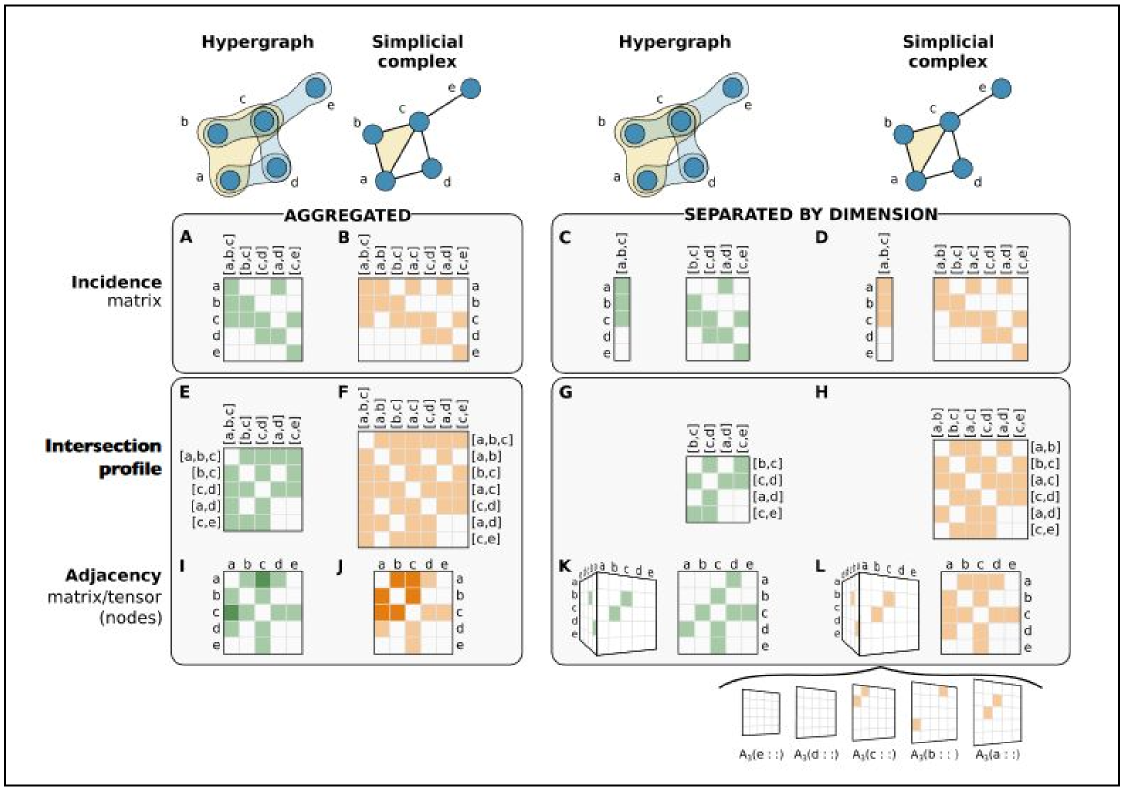


Figure 3 - Matricial and tensorial descriptions of higher-order systems from Battiston et. al [1]

We have further explored different hypergraph libraries and frameworks that aim to reduce the programming efforts by providing high-level APIs. For example, HyperNetX [7] provides a python package for hypergraph analysis and visualization. To keep track of the objects in a hypergraph each node and edge is instantiated as an Entity and given an identifier, uid. Since hypergraphs can be quite large, only these identifiers will be used for computation intensive methods, this means the user must take care to keep a one to one correspondence between their set of uids and the objects in their hypergraph. This library has support to create hypergraphs in a couple of ways; for example, empty instances, from a dictionary of iterables, from a NetworkX bipartite graph, etc. This library supports the incidence matrix and dictionary representation of hypergraph. It further adds APIs to access graph information, for example, degree, neighborhood, dimensionality, etc.

Finally we explored the existing high-level programming frameworks for hypergraph processing. Julian Shun proposed in-memory hypergraph processing [9] and implemented a couple of parallel graph algorithms; including algorithms for betweenness centrality, maximal independent set, k-core decomposition, hyper-trees, hyperpaths, connected components, PageRank, and single-source shortest paths. This word is an extension of their earlier study [13]. Similar to us, this work also considers the bipartite representation of hypergraph.

3. Hypergraph Algorithms

3.1. Pagerank

Pagerank is a commonly used graph analytic algorithm. It is used to find the relative importance of the vertices in a network. There are a wide range of applications where pagerank plays very important roles, for example in recommendation systems, link prediction, search engines, etc. We can think of the implication of pagerank in hypergraphs in two different ways. First, the importance of vertices in a network based on their group participation. For example, in a social network context we can measure the importance of a user based on the group membership, e.g., admin of a group with a minion of users might have a bigger influence in the whole network. Second, the importance of the hyperedges based on the vertices it is linked to. For example, from a co-authorship network we can find the most influential publications based on the relative importance of the authors in the network.

4. Datasets

Finding sufficiently large datasets for hypergraphs has been a problem due to the novelty of the field. For testing, we have used the dataset *email-enron* produced by the CALO project that creates a hyperedge between all nodes included in an email [12]. Email-enron has 143 nodes representing people, contains 10,883 timestamped simplices, 1,542 of which are unique in the nodes that they connect. Temporality is not currently of interest in our model, so we will only consider these unique hyperedges. We have also identified four more datasets that are large enough to provide useful benchmarking. These datasets are shown in Table 1 -

Dataset	Source	Node Count	Hyperedge Count
com-Orkut-group	SNAP	2,320,000	15,300,000
com-Friendster	SNAP	7,940,000	1,620,000
dbpl-5000	SNAP	317,080	1,049,866
email-enron	SNAP	36,692	183,831

** after removing duplicate hyperedges*

Table 1 - Large or Useful Hypergraph Datasets

The first two datasets, *com-Orkut-group* and *com-Friendster*, are representative of group membership in their respective social media networks. Each hyperedge represents a group, and the nodes that are connected together are the members of that group. The metrics for these two datasets were pulled from '*Practical Parallel Hypergraph Algorithms*' [9] Table 1 as it identifies the group metrics rather than person-to-person friendships. The *dbpl-5000* dataset represents co-authorship in academic papers. Each hyperedge represents an academic paper, with the nodes within the hyperedge representing the authors of that paper. The *email-enron* dataset represents a set of publicly available emails, with each hyperedge representing an email, with the nodes within each hyperedge representing the sender and receivers of the email.

One dimension of the data that was commonly removed from the first two datasets was the timestamps. It could be useful to explore whether collapsing this temporal component into hyperedge weight could yield useful information in the graph's metrics.

To supplement the drought of large hypergraph datasets, random hypergraphs can also be generated to observe algorithm performance. This may prove to be challenging on its own as many traditional random graph generation algorithms need additional information to generate hypergraphs.

5. Graph Configuration

A method for representing hypergraphs is to create a clique among all pairs of vertices of each hyperedge and store the result as a regular graph. As we mentioned earlier, this leads to a couple of issues, for example loss of information. Furthermore, the space overhead is significantly higher than that of the original hypergraph. Another approach is to use a bipartite graph representation with participating vertices in one partition and the hyperedges in the other partition. We keep hypergraph information using this bipartite graph representation where the hyperedges are connected to the participating vertices as we can see in the following figure.

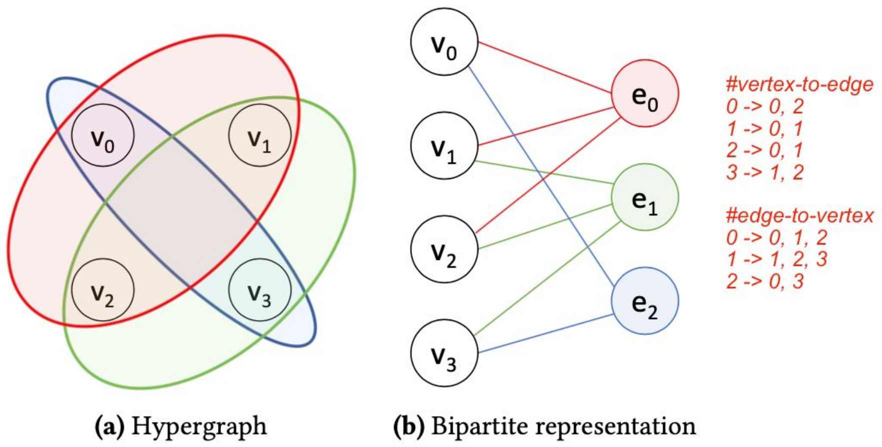


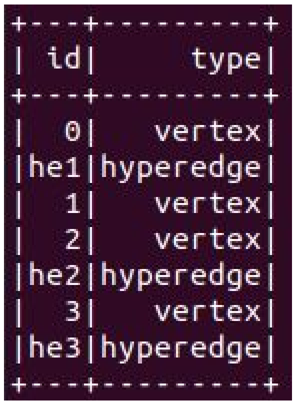
Figure 4 - An example hypergraph representing the groups {0,1,2}, {1,2,3}, and {0,3}, and its bipartite representation from [9]

To access the neighbors of a vertex, we need to loop over all the hyperedges where that vertex has membership and then access the set of participating vertices of those hyperedges. For example to access the neighbors of vertex V_0 in Figure 1, we have to loop over hyperedges e_0 and e_2 and then we will be able to retrieve the participating vertices $\{V_1, V_2, V_3\}$ of those hyperedges. By representing hypergraphs in this way, we would be able to solve the information loss and data duplication issue that we have discussed earlier.

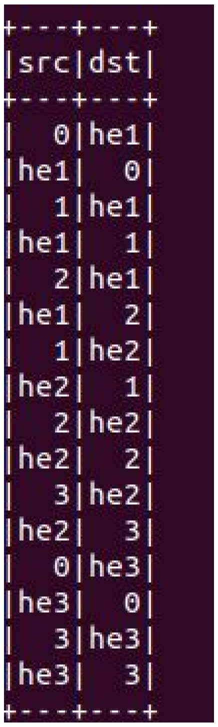
The graph storage is done in this form of dataframes. This is a relational database format in which the vertices are stored with their ID and type where the type refers to a hyperedge or a regular vertex. The edges are stored in the format of src and destination. Here is an example of how it is stored (undirected graph).

The relational style storage helps is basically a way of partitioning and storing vertices and edges amongst the distributed environment and the querying via the filtering/relational querying is done in parallel by default. The basic operations to perform major tasks like page rank, bfs and centrality are based on map reduce mechanism. This is done using a map transformation and reduction process which will yield these metrics.

Fig1, denotes the vertex dataframe, which is of id and type , the type which denotes the vertex or hyperedge. The Fig 2 is a simple undirected edge set each of which denote the bipartite edges. The neighbors of a given vertex can be derived from this bipartite representation.



```
+---+---------+
| id|     type|
+---+---------+
|  0|   vertex|
|he1|hyperedge|
|  1|   vertex|
|  2|   vertex|
|he2|hyperedge|
|  3|   vertex|
|he3|hyperedge|
+---+---------+
```



```
+---+---+
|src|dst|
+---+---+
|  0|he1|
|he1|  0|
|  1|he1|
|he1|  1|
|  2|he1|
|he1|  2|
|  1|he2|
|he2|  1|
|  2|he2|
|he2|  2|
|  3|he2|
|he2|  3|
|  0|he3|
|he3|  0|
|  3|he3|
|he3|  3|
+---+---+
```

Figure 5 - DataFrameVertex Representation *Figure 6 - DataFrame Edges Representation*

This partitioning is an important part of our work. The importance of partitioning the data is to take advantage of the distributed environment in the most effective and time-efficient way. Knowing the distributed node shards, where a node's neighbors exist, is the key challenge of any distributed graph processing framework. A good partitioning algorithm aims to solve this challenge. In the bipartite representation of hypergraph, it is relatively important to identify neighbors using hyperedges. Due to this, each shard/node in the cluster needs to know all hyperedges' information and the connecting neighbors. The most rudimentary way to solve this problem is to have the graph metadata stored in all the machines, but we go a step further and reduce the overhead by using a cluster-wide broadcast of only the essential hyperedge data. This way, the costs are less, and we can get the neighbor's information. The page rank runs on this partitioned data on this cluster.

6. Evaluation

We implemented a distributed PageRank algorithm for hypergraph. The experiments are carried out on an 8 node cluster running on AWS. Each node is a m4.2x large machine. Note that each worker corresponds to one core. One node acts as the master and the other 3 nodes act as slaves. The execution engine Apache Spark 2.4.7 is implemented in Python by utilizing Databricks GraphFrames API 0.8.1, which is a graph processing framework on top of which the hypergraph algorithms are implemented.

Using GraphFrames, which is a wrapper around apache spark in python (PySpark), which uses a variety of dataframe based functionality to provide for us the graph processing engine, we run distributed versions of our algorithms. Our Key evaluation metric would be successful distributed implementations of select algorithms. The final goal is to have a comparable library which runs on top of apache spark using the graph processing library to be a resulting hypergraph processing library which is an equivalent in terms of running distributed algorithms to the state of the art for graphs. The current state is the distributed storage using dataframes of hypergraphs in bipartite format. We also ran simulations of page rank algorithms in the distributed environment. Our goal is to implement algorithms like page rank, bfs and betweenness centrality in a distributed environment. In the end result we would like to have scalable distributed implementations of page rank with our own partitioning in order to evaluate the results of our hypergraph library.

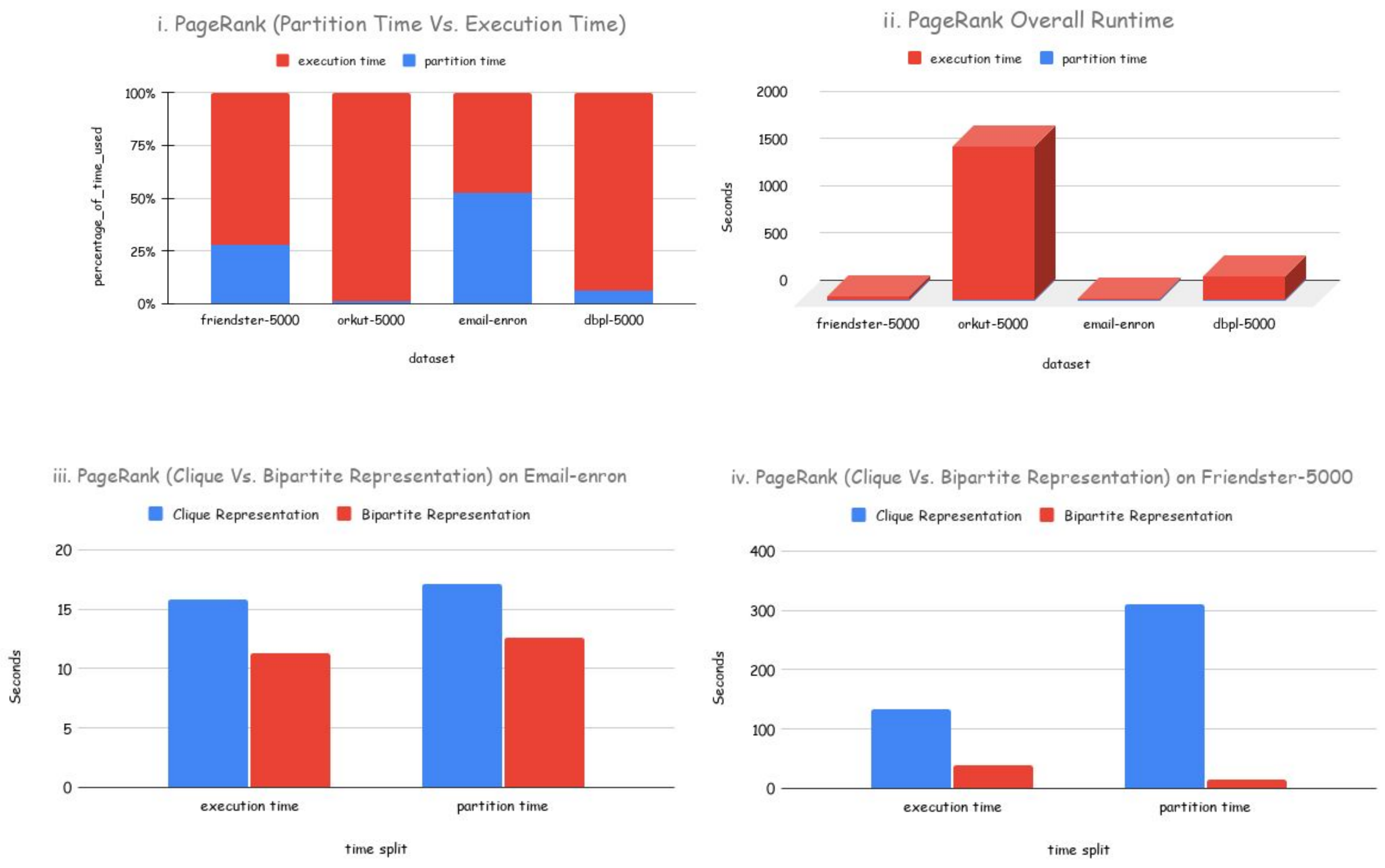


Figure 7 - Experiment results on PageRank

The implementation of page rank, is based on the rdd based map reduce framework in apache PySpark in Python, in which we parallelly parse factors like neighborhood of a vertex, computing vertex page ranks, and propagation. Prior to this algorithm running, we partition the graph in a way that it can be accessed easily in the future. We run on a 5000 version dataset of the most commonly used hypergraphs discussed in the datasets section. We achieved good performance in the partitioning, and as expected the bipartite representation performs better than the clique page rank version.

In figure 7(i), we compared the distributed partition and algorithm execution time for PageRank algorithm. For all the large datasets, partition cost is significantly lower than the algorithm execution time. In figure 7(ii), we plotted the overall performance of the PageRank algorithm for different datasets. Here, the orkut-5000 graph takes a longer running time compared to others. The reason is, orkut-5000 has much larger hyperedges (i.e. the number of participating nodes in the hyperedges). For example, compared with dbpl-5000, orkut-5000 have **12x** larger hyperedges. On the other hand, dbpl-5000 runs only **6x** times faster than orkut-5000. This actually implies the usefulness of running large hypergraphs in distributed environments. In figure 7(iii) and figure 7(iv), we compare two hypergraph representations (e.g. clique and bipartite representations) for email-enron and Friendster-5000 respectively. We weren't able to run PageRank on the clique implementation of the hypergraph for orkut-5000 and dbpl-5000, as those graphs are very large and it was not possible to fit them within our testbed setup. This further signifies the importance of our research. Figure 7(iv) indicates that the partition time in bipartite representation significantly reduces for larger hypergraphs compared to the execution time. This is quite expected, as in the clique representation we have to put a lot of edges and hence increases the partition time exponentially.

7. Conclusion

In conclusion, based on the current resources for hypergraph experimentation and implementation, building a library that allows for hypergraph creation and testing for the large networks in a distributed environment, along with a collection of hypergraph algorithms, would be a worthwhile endeavor. We plan to examine the performance of our current implementations and see how we can improve them. Further we will explore the implications of hypergraph partitioning research for distributed computation of hypergraph algorithms.

8. Reference

1. Battiston, F., Cencetti, G., Iacopini, I., Latora, V., Lucas, M., Patania, A., Young, J.-G., & Petri, G. (2020). Networks beyond pairwise interactions: Structure and dynamics. Physics Reports. https://doi.org/10.1016/j.physrep.2020.05.004
2. Benson, A. R. (2019). Three hypergraph eigenvector centralities. SIAM Journal on Mathematics of Data Science, 1(2), 293-312. https://doi.org/10.1137/18M1203031
3. Chodrow, P. S. (2020). Configuration models of random hypergraphs. Journal of Complex Networks, 8(3), cnaa018. https://arxiv.org/pdf/1902.09302.pdf
4. Kamiński, B., Poulin, V., Prałat, P., Szufel, P., & Théberge, F. (2019). Clustering via hypergraph modularity. PloS one, 14(11), e0224307. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0224307
5. D. Zhang, J. Yin, X. Zhu and C. Zhang, "Network Representation Learning: A Survey," in IEEE Transactions on Big Data, vol. 6, no. 1, pp. 3-28, 1 March 2020, doi: 10.1109/TBDATA.2018.2850013.

6. Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, “Hypergraph neural networks,” in AAAI, 2019.
7. Pacific Northwest National Laboratory, Python package for hypergraph analysis and visualization., (2020), GitHub repository, https://github.com/pnnl/HyperNetX
8. Leland McInnes, A library for hypergraphs and hypergraph algorithms., (2015), GitHub repository, https://github.com/lmcinnes/hypergraph
9. Julian Shun. 2020. Practical parallel hypergraph algorithms. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20). Association for Computing Machinery, New York, NY, USA, 232–249. DOI:https://doi.org/10.1145/3332466.3374527
10. Y. Gu, et al.,"Distributed Hypergraph Processing Using Intersection Graphs" in IEEE Transactions on Knowledge & Data Engineering, vol. , no. 01, pp. 1-1, 5555. doi: 10.1109/TKDE.2020.3022014
11. J. Yang and J. Leskovec. Defining and Evaluating Network Communities based on Ground-truth. ICDM, 2012.
12. Simplicial closure and higher-order link prediction. Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. *Proceedings of the National Academy of Sciences (PNAS)*, 2018.
13. Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 135-146, 2013.