DISTRIBUTED MEMORY PROCESSING OF VERY LARGE GRAPHS

by

SARA RIAZI

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

September 2019

DISSERTATION APPROVAL PAGE

Student: Sara Riazi

Title: Distributed Memory Processing of Very Large Graphs

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Boyana Norris	Chair
Dejing Dou	Core Member
Stephen Fickas	Core Member
Peter Ralph	Institutional Representative

and

Janet Woodruff-Borden	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded September 2019

© 2019 Sara Riazi

DISSERTATION ABSTRACT

Sara Riazi

Doctor of Philosophy

Department of Computer and Information Science

September 2019

Title: Distributed Memory Processing of Very Large Graphs

Big graphs such as social networks or the internet network, biological networks, knowledge graphs appear in many domains. However, processing these graphs rely on the accessibility of high-performance frameworks which are able to handle these large graphs. One aspect of this accessibility is the usability of the frameworks for a broad community of researches who do not have sufficient expertise to work with these frameworks. To address this issue, we introduce GraphFlow framework, a workflow-based framework that provides several graph mining components. GraphFlow benefits from data-parallel Apache Spark and its GraphX library, as the back-end, so it processes very large graphs. GraphFlow also supports the construction of experiment pipelines that involve running several components.

Integrated into our GraphFlow framework, we also introduce a novel vertex-centric network embedding algorithm, which can learn low-dimensional vectors for vertices of very large graphs. Our network embedding algorithm can scale to graphs with billions of edges, while previous algorithms do not scale to the graphs of this scale.

GraphFlow also supports dynamic graphs using graph snapshots and batch updates. We provide SSSPIncJoint, a novel algorithm for computing single-source shortest paths (SSSP) for dynamic graphs. SSSPIncJoint is significantly more efficient than running SSSP for each snapshot of a dynamic graph.

CURRICULUM VITAE

NAME OF AUTHOR: Sara Riazi

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Shahid Beheshti University, Tehran, Iran

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2019, University of Oregon
Master of Science, Computer and Information Science, 2019, University of Oregon
Bachelor of Science, Software Engineering, 2007, Shahid Beheshti University

AREAS OF SPECIAL INTEREST:

Big Data
Data Mining

PROFESSIONAL EXPERIENCE:

Graduate Research & Teaching Assistant, Department of Computer and Information Science, University of Oregon, 2013 to 2019

GRANTS, AWARDS AND HONORS:

Grace Hopper Student Scholarship, 2019

Marthe E. Smith Memorial Science Scholarship, College of Art and Sciences, University of Oregon, 2017

PUBLICATIONS:

S. Riazi and B. Norris, GraphFlow: Workflow-based Big Graph Processing, In Proceedings of 2016 IEEE International Conference on Big Data (Big Data), Washington, D.C., 2016.

S. Riazi, S. Srinivasan, S. K. Das, S. Bhowmick, and B. Norris, Single-Source Shortest Path Tree For Big Dynamic Graphs, In Proceedings of 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA.

S. Srinivasan and S. Riazi and B. Norris and S. K. Das and S. Bhowmick, A Shared-Memory Parallel Algorithm for Updating Single-Source Shortest Paths in Large Dynamic Networks. In Proceedings of the 25th IEEE International Conference on. High Performance Computing, Data, and Analytics, 2019.

S. Riazi, B. Norris, Vertex-Centric Network Embedding via Apache Spark. In Proceedings of the 19th Industrial Conference on Data Mining, New York, NY, 2019.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my advisor, Boyana Norris, that her extensive support makes all these possible. I also want to thank my dissertation committee, Steve Fickas, Dejing Dou, and Peter Ralph, for their constructive comments and help through these years. I also want to thank our collaborators Sanjukta Bhowmick, Sriram Srinivasan, and Sajal Das, for discussions and brainstorming, and joint works that we have done together. Last but not least, I want to thank my husband, Pedram, whose supports and understanding made this journey much easier for me.

To my husband, Pedram,

to my mother, Mina,

and to the memory of father, Reza.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1
1.1. Contributions 4
1.2. Dissertation Outline 5

II. BACKGROUND 6
2.1. Data-Parallel Systems 7
2.2. Distributed Graph Systems 7
2.3. Comparison and Discussion 22
2.4. Conclusion 28

III. GRAPHFLOW 29
3.1. Data Description 30
3.2. Interaction Model 31
3.3. GraphFlow Components 32
3.4. Use Cases 37
3.5. Conclusion 45

IV. GRAPH EMBEDDING 46
4.1. Sequence Representation 47
4.2. Network Embedding 51
4.3. Vertex-Centric Network Embedding 54

Chapter Page

4.4. Experiments 59

4.5. Conclusions 67

V. PROCESSING BIG DYNAMIC GRAPHS 68

5.1. Introduction 68

5.2. Related Work 70

5.3. Static SSSP on Spark 71

5.4. Dynamic SSSP on Spark 72

5.5. SSSPIncJoint 77

5.6. Experiments 79

5.7. Conclusion 84

VI. CONCLUSION AND FUTURE DIRECTIONS 85

6.1. Potential future directions 86

REFERENCES CITED 89

LIST OF FIGURES

Figure Page

1.1. An example of workflows in GraphFlow. 2

1.2. Using graph embedding for visualizing the structure of a graph. 3

2.1. Finding connected components of a graph using message passing. . . . 8

2.2. Edge-cut vs. Vertex-cut. 13

2.3. Compact adjacency list representation. 21

3.1. The architecture of GraphFlow. 30

3.2. The Query component. 35

3.3. The workflow of hierarchical clustering. 37

3.4. CDF of the shortest path lengths. 39

3.5. The workflow of finding the CDFs of shortest paths. 39

3.6. The workflow of coarsening a graph using clustering. 39

3.7. CDF of the shortest path length in the coarse graph. 40

3.8. The degree distribution of Wikepedia graph. 40

3.9. The workflow of ranking universities using Wikipedia graph. 43

4.1. Hierarchical softmax. 50

4.2. Node2vec neighborhood exploration. 53

4.3. The embedding of the Reddit graph generated by VCNE. 62

4.4. Average runtime for one training step of VCNE. 65

4.5. The effect of embedding dimension on the running time. 65

4.6. The effect of the number of negative samples on the running time. . . . 66

5.1. The SSSPBase and GraphInc algorithms based on the GAS model. . . 73

5.2. The execution time comparison of SSSPIncJoint and baselines. 82

Figure Page

5.3. Total number of GAS supersteps for running each algorithm. 82

5.4. Total number GAS supersteps for SSSPInc. 83

5.5. Apache Spark workers participation in SSSPInc. 83

LIST OF TABLES

Table Page

2.1. Comparison of distributed graph processing systems. 20

3.1. Top 10 universities found using workflow of Figure 3.9.. 44

4.1. The number of vertices and edges of the used real-world graphs. 60

4.2. F_1 score of vertex classification tasks. 61

4.3. Link Prediction for LiveJournal 63

4.4. The performance of link prediction using VCNE 63

5.1. Vertices and edges of the used real-world and synthetic graphs. 81

5.2. The characteristics of update batches for different graphs. 81

CHAPTER I

INTRODUCTION

Many real-world problems are represented as graphs or networks in different computational domains, such as bioinformatics (Borgwardt et al., 2005; Baldi and Pollastri, 2003), chemical informatics (Ralaivola et al., 2005; Wale et al., 2008), vision (Shi and Malik, 2000; Felzenszwalb and Huttenlocher, 2004), or social networks analysis (Liben-Nowell and Kleinberg, 2007; Backstrom and Leskovec, 2011; Agrawal et al., 2013). These graphs may scale to billions of vertices and edges, for example, Facebook has more than one billion active users. The complexity of graph algorithms is usually polynomial in the number of vertices of the graph. As a result, running graph algorithms over large graphs is very time-consuming. Few network analysis software tools support parallel algorithms, and the set of available methods is also small.

Moreover, a single machine may not be able to load the entire graph representation into memory, so processing very large graphs requires distributed memory and out-of-core processing, which is not widely supported by graph analysis frameworks. In response, distributed and parallel graph processing frameworks have emerged recently, which benefit from advances in high-performance and parallel computing. However, these frameworks have different performance in the presence of resources available to them, such as the number of processors, and the amount of available memory.

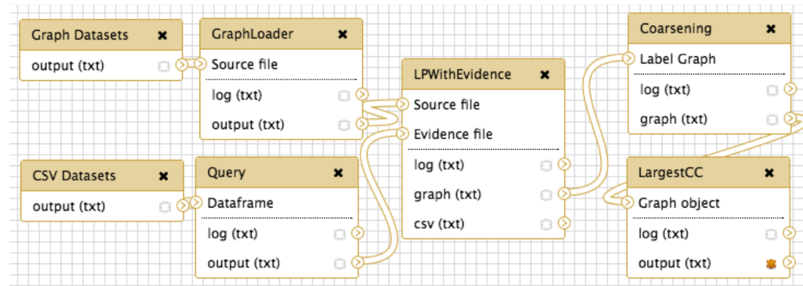

FIGURE 1.1. An example of workflows in GraphFlow. This workflow is used to create a coarse graph given some evidence nodes.

Unfortunately, in practice, there is a significant gap between the services provided by the graph-parallel frameworks and the actual needs of the domain experts. Most graph-parallel frameworks only offer a small set of algorithms that can be used as a black box. However, with the increasing diversity of data formats and solution requirements, there are no high-level reusable solution approaches. Instead, each data analysis instance can have a different workflow based on the underlying analysis framework, typically requiring domain expert involvement at each step. Current graph-parallel frameworks do not provide sufficient support for creating, reusing, and extending complex workflows required for analyzing large diverse datasets. Moreover, they rarely provide support for auxiliary, but necessary tasks, such as creating graphs from raw data, filtering metadata, selecting heuristics and comparing multiple results.

Our response to the aforementioned problems is GraphFlow, a big graph framework that is able to encode complex data science experiments as a set of high-level workflows. GraphFlow combines the Spark big data processing platform and the Galaxy workflow management system to offer a set of components for graph processing using a novel interaction model for creating and using complex

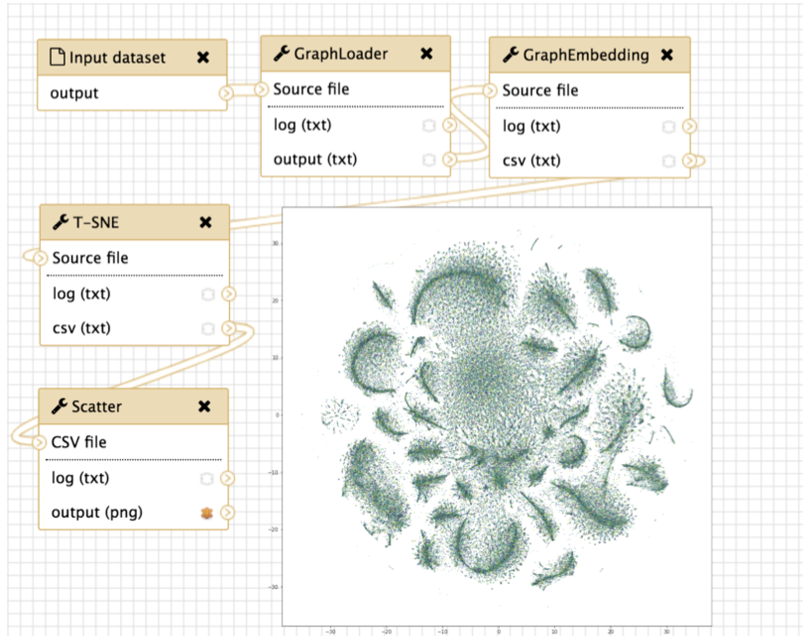

FIGURE 1.2. Using graph embedding for visualizing the structure of a graph.

workflows. GraphFlow contributes an easy-to-use interface and scalable algorithms for big graph analytic (see Figure 1.1. for an example). We discuss the architecture and components of GraphFlow in Chapter III.

We also extend the fundamental graph algorithms provided by GraphX. Among the added algorithms, one of the most challenging components is graph embedding (a.k.a. network embedding). Graph embedding can be used as a major component in many workflows providing vertex features for downstream tasks such as link-predication and vertex classification. Figure 1.2. shows a workflow for visualizing the structure of a graph using graph embedding.

However, supporting graph embedding in a data-parallel framework is not trivial since it includes propagating of large messages across workers, which prohibits learning meaningful embedding. In Chapter IV, we introduce a novel algorithm that addresses the problem of using data-parallel frameworks

especially Apache Spark for training graph embedding, while learning a meaningful representation.

Another important feature of GraphFlow is the support for very large dynamic graphs. We advise a novel algorithm for computing SSSP over very large dynamic graphs, while addressing the challenges of processing very large dynamic graphs in Apache Spark. Chapter V discusses these challenges and our response to them.

1.1. Contributions

The main contributions of this dissertation include

- We introduce GraphFlow framework, a workflow-based framework for processing big graphs using Apache Spark. The GraphFlow framework extends the map-reduce paradigm to high-level components, which maps a graph to another graph, or reduce a graph to values. An example of this high-level maps is coarsening component, which maps a graph to another graph that is the coarse version of an input graph with fewer edges and vertices.
- We introduce vertex-centric network embedding (VCNE) to compute network embedding for very large graphs. Network embedding becomes an integral part of graph analysis pipelines such as vertex classification or link prediction. However, most of existing network embedding approaches do not scale to very large graphs. In response, we introduce VCNE: a vertex-centric approach that is developed on top of Apache Spark and can scale to very large graphs.
- We introduce SSSPIncJoint, a data-parallel approach for computing single-source shortest path (SSSP) for very large dynamic graphs. SSSPIncJoint

addresses the problem of recomputing SSSP for every snapshot of big dynamic graphs by introducing a novel algorithm that updates the SSSP tree based on the incoming changes.

1.2. Dissertation Outline

Chapter II describes the graph-parallel frameworks, and in more details, Apache Spark and its graph processing library, GraphX. We also describe the necessary concepts such as map-reduce computation in this chapter. In Chapter III, we introduce our GraphFlow framework including its architecture and its components. We also show some case studies that motivate the usage of GraphFlow. Chapter IV is dedicated to the graph embedding components, its algorithm, and comparisons. We discuss the challenges of supporting dynamic graphs in Apache Spark in Chapter V, and introduce a novel algorithm for computing single-source shortest path for dynamic graphs. Finally in Chapter VI, we conclude this dissertation and discuss the future direction for extending this work.

CHAPTER II

BACKGROUND

Many real-world problems are described using networks and graphs such as social networks, Internet maps, and protein interactions. These graphs may scale to billions of vertices and edges, for example, Facebook has more than one billion active users. The complexity of graph algorithms is usually polynomial in the number of vertices of the graph.

As a result, running graph algorithms over very large graphs is very time-consuming. Moreover, a single machine may not be able to load the entire graph representation into the memory.

Therefore, to address the problem of processing very large graphs, many distributed and parallel graph processing frameworks have emerged recently, which benefit from advances in high-performance and parallel computing. However, these frameworks have different performance in the presence of resources available to them, such as the number of processors, and the amount of available memory. Understanding the architectural properties of these frameworks is essential for determining which framework is more suitable for different problems. For example, if the ratio of graph representation size to the available memory is high, we need a framework that supports out-of-core processing, which means that it partially loads the graph into the memory of the machine, and writes back the updated representation to its permanent external memory as soon as it requires to process other parts of the graph.

In this Chapter, we study a set of well-known distributed graph processing systems including Pregel (Malewicz et al., 2010), PEGASUS (Kang et al.,

2009), GraphLab (Low et al., 2012), Powergraph (Gonzalez et al., 2012), GraphX (Gonzalez et al., 2014), TurboGraph (Han et al., 2013), GraphCT (Ediger et al., 2013), Pregelix (Bu et al., 2014). Moreover, we also consider two other general approaches for distributed graph processing using GPUs (Harish and Narayanan, 2007) and MPIs (Plimpton and Devine, 2011).

2.1. Data-Parallel Systems

One of the most significant advances in distributed data processing is the map-reduce programming model (Dean and Ghemawat, 2008). In map-reduce, data is converted to key-value pairs and then partitioned to nodes. A map-reduce system consists of a set of workers that are coordinated by a master process. The master process assigns partitions to workers, and then workers apply a user-defined map function to the key-value pairs, resulting in intermediate key-value pairs stored on the local disks of workers. The intermediate key-value pairs are passed to another set of workers that group the key-values by keys and apply a user-defined reduce function on the group of values associated with a particular key. The workers then apply the reduce function and store the output on their local disks, so one can combine different partitions of the output together and create a single output file, or pass the output as the input to another map-reduce call.

Many graph frameworks are developed on top of data-parallel systems, in order to benefit from their optimized parallel processing.

2.2. Distributed Graph Systems

In this section, we describe a set of important distributed graph processing frameworks.

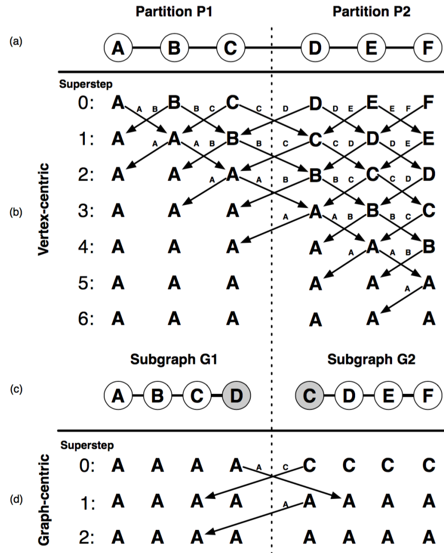

FIGURE 2.1. Finding connected components of a graph (figure from Tian et al. (2013)). a) The original graph that placed over a cluster of two computers. b) The vertex-centric computation. c) Graph-centric computation.

2.2.1. Pregel

Pregel (Malewicz et al., 2010) is a distributed graph processing framework, which introduces the important think-like-a-vertex paradigm and vertex-centric programming model for distributed graph processing.

The idea of the vertex-centric programming model is to distribute graph algorithms over vertices, so the system runs the program or function associated with each vertex in parallel. Vertices can communicate with each other in order to produce the final result of the designed algorithm. For example, suppose we want to find the connected components of a graph. Each vertex sets its value to be its ID number, and then, sends the vertex value to all of its neighbors. A vertex collects the messages from its neighbors and selects the minimum of the received values, then it updates the vertex value using the new value. At this point, the vertex sends the new value to the neighbors again if the updated value is different

from the previous value. Finally, after several communication steps, the value of vertices shows their component ID which is the smallest ID of the vertices in that component. The Figure 2.1. shows the message passing steps for finding the connected components of a chain.

Pregel iteratively runs user-defined function *compute* for each vertex simultaneously. Each iteration of the algorithm is called a superstep, in which a vertex gathers all messages from the previous superstep, and prepares messages for its neighbors that will be delivered on the next superstep. Each vertex can decide to deactivate itself by voting to halt. A vertex is reactivated again if it receives a message from other vertices. The program terminates when there are no more messages and all vertices are inactive. To reduce the number of messages passed among machines or workers, Pregel includes another user-defined function called *combine*. The combine function, if provided, is applied to the messages that have been sent for a vertex. Pregel also allows the users to provide a user-defined *aggregator* function, which acts similar to fold semantic in functional programming languages. The aggregator function is applied to the value of all vertices at the end of each superstep and aggregates them together, for example by computing sum or max value. The result of aggregation is available to all vertices in the next superstep. Pregel includes a master node and a set of workers. The master is responsible for coordinating the supersteps such that all workers complete the current superstep, and then next superstep starts. To force synchronization, each superstep ends with a *barrier* in which the workers wait for the master in order to get the permission of entering the next superstep.

Each worker is responsible for a partition of the underlying graph and calls the compute function for every vertex in its partition, and exchanges the

produced messages with the other workers. Pregel is only able to do the in-memory computation, so the number of workers should be selected accordingly such that each worker is able to keep the graph partitions and the corresponding messages in memory.

Pregel achieves fault tolerance by putting checkpoints at the beginning of each superstep. At each checkpoint, all workers are responsible for storing the value of vertices, edges, and outgoing messages on their local disk. The master also stores the value of aggregators. In the case of failure, the master coordinates the workers to rollback to the last successful checkpoint. Pregel is developed by Google Inc. as a closed source framework. However, Apache Giraph[1] is an open source implementation of Pregel that provides similar specifications.

2.2.2. GraphLab

GraphLab (Low et al., 2012) is another distributed graph processing framework. Similar to Pregel, GraphLab is based on the vertex-centric programming model, but instead of message passing it uses shared memory. GraphLab defines a scope of a vertex to be the value of vertices and edges in the graph that are needed for updating the value of the vertex. Two or more vertices that have intersecting scope can be considered to communicate with each other through the intersecting scope.

If two vertices that are located on different workers or machines have intersecting scopes, then each worker keeps immutable copy variables of the shared scopes. Whenever an original variable changes, its corresponding worker sends the updated value to the workers that keep the copies of the variable.

[1] http://giraph.apache.org

GraphLab is based on the pull model, in which a vertex uses the values in its scope to update its value, unlike Pregel that each vertex pushes the messages for its neighbors.

In GraphLab, users provide a stateless *update* function, which can be applied on values in the scope of the vertex. Applying the update function on two adjacent vertices may result in a collision, so GraphLab provides different consistency methods to control the mutual-exclusion. Full consistency model is the most restricted method, in which two neighboring vertices cannot run in parallel. The other model is edge consistency, which allows two adjacent vertices to run in parallel as long as each vertex is only read and modify the values that are associated with the vertex and all incident edges. Finally, GraphLab provides vertex consistency model that ensures each vertex only modifies its local values. The users may select one of these consistency types based on their computation to maximize efficiency.

When a vertex runs the apply function, it triggers its neighbors to call their apply functions. Therefore, the execution of vertices is asynchronous, comparing to synchronous supersteps in Pregel.

2.2.3. Powergraph

Powergraph (Gonzalez et al., 2012) is based on vertex-centric programming model and supports both synchronous execution, similar to Pregel, and asynchronous execution, similar to GraphLab.

The main difference of Powergraph with Pregel and GraphLab is that it supposes that large natural graphs follow the power-law degree distribution, which means that a small fraction of vertices is incident to a large fraction of the edges.

Power-law degree distributions define the probability that a vertex having degree d (d neighbors) as $P(d) = d^{\alpha}$, where α is a positive constant that determines the sparsity of the graph.

Since the complexity of the apply or compute function is linear in the degree of a vertex, the graphs with power-law degree distribution suffer from imbalanced workload and communication overhead. To address imbalanced workload, Powergraph runs the program associated with each vertex (vertex-program) in parallel in order to reduce the delay of processing on high-degree vertices.

Powergraph introduces Gather-Apply-Scatter (GAS) model, based on the vertex-centric programming model of Pregel and GraphLab. In the GAS model, the algorithm runs over three stages: data preparation, iterations of vertex-program, and output generation. A vertex-program consists of gather, sum, apply, and scatter. A vertex-program applies the gather function in parallel on the value of every edge that is incident with the corresponding vertex and then aggregates the values using sum function. Then the apply function is executed given the aggregated value in order to update the value of the vertex. Finally, the vertex program calls the scatter function in parallel for all edges incident to the vertex in order to update their values.

Since the vertex-program can be executed in parallel for different edges incident to the vertex, so in order to address the imbalanced network overhead, Powergraph distributes the edges evenly among the workers and allows each worker to keep a mirror of vertex data of each edge's end-points if they are not located at the same machines.

Distributing edges evenly among machines and mirroring the vertices is called vertex-cut (Gonzalez et al., 2012) as opposed to edge-cut, in which the vertices of

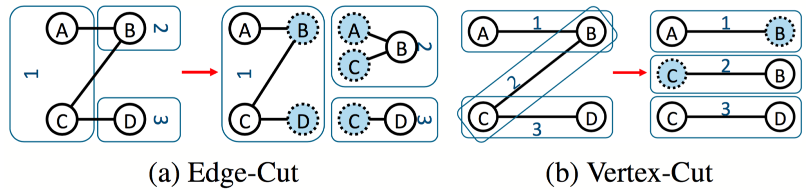

FIGURE 2.2. Edge-cut vs. Vertex-cut. The shaded vertices are mirrors. (figure from Gonzalez et al. (2012)).

a graph are evenly assigned to different machines, and mirroring happens for the adjacent vertices located at different machines. Figure 2.2. shows an example of vertex-cut vs. edge-cut. In the given example, distributing a graph of four vertices over three machines needs five mirroring variables if edge-cut is used, while it needs only three mirroring variables in the case of using vertex-cut.

Gonzalez et al. (2012) theoretically show that vertex-cut reduces the network overhead needed for synchronizing copy variables in compare to edge-cut for graphs with power-law degree distributions. so vertex-cut addresses the imbalanced communication workload.

2.2.4. GraphX

GraphX (Gonzalez et al., 2014) is a distributed graph processing framework developed on top of Apache Spark,[2] which is a fast growing framework for large data processing.

Spark supports a distributed architecture, in which an application is running as a set of processes. The main program, called the driver, consists of an object called *SparkContext* which coordinates the execution of the application's processes on Spark workers through a Spark master node, which manages the

[2]https://spark.apache.org/

workers. The most important concept in Spark is its resilient distributed datasets (RDDs). RDDs (Zaharia et al., 2012) are immutable collections of objects that are partitioned across different Spark workers in the network.

Since Spark is a data-parallel computation system, GraphX implements graph operations based on data-parallel operations available in Spark such as join, map, and reduces. GraphX represents graphs using two RDDs, one for vertices and another for edges.

However, handling graphs in a data-parallel computation system is more complex than map-reduce operations since the vertices should be processed in the context of their neighbors. To address that, GraphX introduces the triplet concept, which joins the structure of vertices and edges. Each triplet carries the value of an edge and the values of vertices that incident with that edge. Therefore, by grouping triplets on the id of the source or destination vertex, one can access the value of all the neighbors of each vertex. Moreover, since the triplets are distributed, if the neighbors of a vertex are located on different machines, then Spark workers communicate with each other to construct the group by the result. Therefore, strategies for distributing graphs over different partitions become important in terms of communication overhead and storage overhead. GraphX supports both edge-cut and vertex-cut graph partitioning strategies.

2.2.5. Pregelix

Pregelix (Bu et al., 2014) offers the same vertex-centric model as Pregel, but it is developed on top of Hyrack data-parallel system. Hyrack offers select, join, and group by operators as external memory operations. These operators are used by Pregelix to implement the message-passing model as it is provided by Pregel. For

example, considering a table of messages that includes the destination vertex id and the message value: (destId, msg), a group-by operator on the destId, groups all the messages that are sent for each vertex together. Then it can combine the messages using the combine user-defined function similar to Pregel.

Pregelix partitions the vertex data among the Hyrack workers using a selectable partition function over vertex id. Each worker applies the same operations as Pregel in each superstep using the relational operations and distributes messages based on the same partition functions on the destination vertex id. In this way, the message data that targets a vertex will locate at the same worker.

The main advantage of Pregelix over Pregel is out-of-core support because of using Hyrack. Therefore, Pregelix can scale better than Pregel if the graph size to available memory ratio is high.

2.2.6. MR-MPI

MR-MPI (Plimpton and Devine, 2011) provides map-reduce functionality using the message passing interface (MPI) for parallel processing. The framework is not specific for graph data, but the authors provide different graph algorithms such as random graph generation (with power-law degree distribution), PageRank, single-source shortest path, and triangle count as well as performance evaluation of the algorithm on random graphs with up to 268 million vertices and 2 billion edges. The framework provides three main functions *map*, *reduce*, and *collate*. The execution of these functions, similar to the other map-reduce frameworks, is synchronous, which means all processors wait until one stage of map, reduce, or collect gets completed before proceeding to the next stage.

The data representation that is used by MR-MPI is either key-value or key-multivalue pairs. A map function can generate key-value pairs or map existing key-value pairs. Each processor stores the resulting key-value pairs. The collate function first reassign the key-value pairs to different processors based on a hash function on the key part. This data movement happens using MPI communication. Then, the collate function identifies the unique keys on each processor and combines the value part to create key-multivalue pairs. The reduce function, finally, applies a function on the multivalue part of each pair, resulting in a key-value pair.

Since MR-MPI is intended to process large graphs, it supports out-of-core processing, which needs that a processor stores some parts of key-value on a disk because of memory limitation. The collate function becomes extremely expensive for out-of-core processing since it needs to load pairs from disk to construct key-multivalue pairs. By increasing the number of processors, the number of pairs that are assigned to each process decreases, so we can control the amount of out-of-core processing to boost the overall performance.

2.2.7. Giraph++

Giraph++(Tian et al., 2013) is similar to Pregel, but it introduces a graph-centric model to reduce the number of supersteps and synchronization points, and to increase the scalability and efficiency of the systems given a good graph partitioning.

In graph-centric models, instead of running vertices in parallel, the system runs subgraphs in parallel, and for each subgraph, it applies a sequential version of the algorithm. If a subgraph spans over more than one machine, then each worker keeps a mirror of the boundary variables. The workers need to synchronize the

value of boundary variables. Figure 2.1..d shows an example of finding connected component using graph-centric model for the partitioning given in Figure 2.1..c. Subgraph A, B, C and subgraph D, E, F are located on machine P1 and P2, respectively. Since C and D are connected in the original graph, then P1 keeps a mirror of D, and P2 keeps a mirror of C as boundary variables in their subgraphs.

During a superstep, each worker runs breadth-first search as a sequential algorithm for finding the connected components in its subgraph, then workers communicate with each other similar to Pregel vertex-centric approach. Therefore, at the first supper step, each worker finds the connected components in the subgraph, and then the connected components exchange the component id using the boundary variables. A component updates its component id if it is not smaller than the received component ids through boundary variables.

Supposing the most vertices of a subgraph are located at the same machine, the graph-centric model is significantly faster because it needs fewer synchronization point. However, to optimize the efficiency using graph-centric models, the system should use prior knowledge about the partitioning of the graph and its subgraphs to reduce the number of copy variables.

2.2.8. PEGASUS

PEGAUSUS (Kang et al., 2009) offers generalized matrix-vector multiplication, which can be used to efficiently implement many graph algorithms such as PageRank, random walk, and diameter estimation. PEGASUS includes three functions: *combine2*, *combineAll*, and *assign*. Suppose the m_{ij} represents the entries of a matrix and v_j represents the entries of a vector. Combine2 applies a user-defined function on each pair of (m_{ij}, v_j) and produces an intermediate

results x_i. Then, combineAll aggregates all x_i values and produces the new value v_i'. Finally, an assign operator replaces the old value v_i by v_i'. If we define combine2 as a product, combineAll as a sum, then these three operations compute matrix-vector multiplication.

For the algorithms that are representable using matrix-vector multiplication, PEGASUS iteratively applies these three operations until it meets algorithm-specific convergence criteria. PEGASUS applies these operators using Hadoop map and reduce functions, so it inherently synchronous.

In the basic model, each edge is described as one line in the Hadoop data file, which reduces the complexity of computation to be the same as vertex-centric models, however, PEGASUS also provides more optimized matrix-vector multiplication by encoding the matrix as block matrix through edge clustering.

2.2.9. TurboGraph

Similar to PEGAUSS, TurboGraph (Han et al., 2013) also provides matrix-vector multiplication. However, TurboGraph is intended to process very large graphs on a single PC using a disk-based approach. It uses adjacency list for representing the graph, but since for very large graphs the adjacency list may not fit in memory, TurboGraph uses fixed-size pages for storing adjacency lists and only keeps a small record table in memory. The record table indicates the first vertex of each page as well as the number of pages if adjacency list of one vertex spans over more than one page. The pages are stored on FlassSSD, which offers asynchronous parallel IO. TurboGraph has several threads that processing adjacency lists. When a thread needs a page that is not resident in memory, it creates an asynchronous IO request with a callback function for the buffer manager. Whenever the requested

page is ready a callback thread runs the passed function to process the page. After processing a page, the page becomes unpin, so the buffer manager can replace it with another page based on received requests. This process is called pin-and-slide. TurboGraph implements matrix-vector multiplication as an engine-level graph primitive based on the mention pin-and-slide model. It also includes the breadth-first search for graph traversing, which takes a user-defined function and applies it on every vertex in the graph.

2.2.10. GraphCT

GraphCT (Ediger et al., 2013) is based on Cray XMT multithread processors. Cray XMT provides GraphCT with a massive global shared memory using several physically distributed memories. This massive global shared memory eliminates the requirements of evenly graph partitioning for huge natural graphs since it can load the entire graph in the global memory and make it available to several threads of programs. Cray XMT reduces the memory access time through a fast one-cycle context-switch to the next ready thread. Therefore, to reduce the memory access delays, the user program should benefit from fine-grained parallelism, so Cray XMT offers many fine-grained primitives to instruct the compiler. For example, *progma* instructs the compiler that the loop iterations are independent and each iteration can be executed using separate threads. Cray XMT also offers coarse-grained parallelism, for example, running multiple breadth-first-search in parallel on the graph. For mutual-exclusion of shared memory, Cray XMT also includes many fine-grained synchronization primitives to reduce the delay and increase the throughput of threads.

TABLE 2.1. Comparison of distributed graph processing systems using their architectural properties. The list of used abbreviation for programming model (PM): vertex-centric (VC), graph-centric (GC), General-purpose (GP), Matrix-centric (MC), and for communication model (CM): message-centric (CM), shared-memory (SM), data-centric (DC), and disk-based (DB).

Name	PM	Execution Model	CM	Partitioning Strategy	Out-of-core support	Fault tolerance
Pregel	VC	Synch.	MC	Edge-cut	No	Yes
Pregelix	VC	Synch.	DC	Edge-cut	Yes	Yes
GraphLab	VC	Asynch.	SM	N/A	No	Yes
PowerGraph	VC	Both	SM	Vertex-cut	No	Yes
Giraph++	GC	Synch.	MC	Edge-cut	No	Yes
GraphX	VC	Synch.	DC	Vertex-cut, Edge-cut	Yes	Yes
GraphCT	GP	Asynch.	SM	N/A	Yes	Yes
PEGASUS	MC	Synch.	DC	Edge-cut	Yes	Yes
TurboGraph	MC	Synch.	DB	Edge-cut	Yes	Yes
MR-MPI	GP	Synch.	MC	Edge-cut	Yes	No
GPU-based	GP	Asynch.	SM	N/A	No	No

GraphCT provides an implementation for computing many graph metrics such as diameter estimations, betweenness centrality, and clustering coefficients benefiting from the fine-grained and coarse-grained parallelism and massive shared memory.

GraphCT is also partially offered for parallel processing platforms other than CrayXMT.

2.2.11. GPU-based

Harish and Narayanan (Harish and Narayanan, 2007) introduces a set of graph algorithms using CUDA programming interface for Nvidia 8800 GTX GPUs. 8800 GTX consists of 16 multiprocessor units and each multiprocessor has eight processors, so 8800 GTX contains 128 processors in total. Each multiprocessor provides shared memory for its all eight processors. Each processor

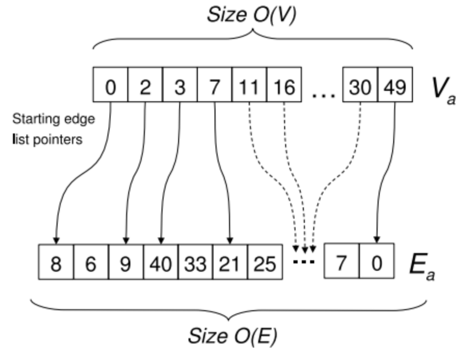

FIGURE 2.3. Compact adjacency list representation (from Harish and Narayanan (2007)).

of a multiprocessor executes the same instruction but over different data. Different multiprocessors can communicate with each other using a device memory. The main limitation of GPUs is that the amount of memory accessible for processors is less than the maximum texture size supported by the graphic card, which is 768 MB on 8800 GTX.

CUDA (computed unified device architecture) is a programming interface for using GPUs as multicore co-processors for general-purpose programming. The CUDA interface offers all memory accessible to the processors to program without any restriction on the data representation.

For the graph representation, the authors used a vertex array and an edge array to represent compact adjacency list. The edges that are incident with each vertex are located at consecutive entries in the edge array, and the vertex keeps the index of the first edge of its incident edges. Figure 2.3. demonstrates an example of a compact adjacency list. In the given example, vertex 0 has an edge to vertex 8 and 6, and vertex 2 has only one neighbor, vertex 9.

The authors provide asynchronous algorithms for breadth-first search, single-source shortest path, all shortest path. All the algorithms run one thread for each

vertex and are rely on atomic read and write to global memory for ensuring the consistency. Each thread is executed until the algorithm converges, and based on the nature of the implemented algorithms the final results do not depend on the execution order of threads.

The larges graph used by the authors has six million vertices and 15 million edges.

2.3. Comparison and Discussion

In this section, we compare the discussed distributed graph processing frameworks and libraries in the Section 2.2. based on their design properties. Scalability analysis of these frameworks and libraries also would be beneficial, but it is out of the scope of this paper. We compare the distributed graph processing systems, based on their programming model, execution model, communication model, graph partitioning strategy, their support of out-of-core computation and fault tolerance computation. Table 2.1. summarizes these properties for the surveyed frameworks.

2.3.1. Programming Model

Programming model determines how a framework offers the graph processing functionalities to its users. In other words, it shows the logical view of the framework for parallelizing the computation over the distributed framework.

The surveyed frameworks support vertex-centric, matrix-centric, graph-centric, and general-purpose programming models.

Vertex-centric based frameworks such as Pregel, Pregelix, GraphLab, GraphX, and PowerGraph run a user-defined vertex-program in parallel for each vertex

in the graph. It is also possible to distribute the vertex-program to improve the efficiency of the algorithm, as it is suggested by Powergraph (Gonzalez et al., 2012). In order to distribute the vertex-program, it must be decomposable over the edges of each vertex. Therefore, Powergraph gains extra parallelism in compare to the other vertex-centric models, especially for natural graphs with power-law degree distributions.

In frameworks that support matrix-centric models, the graph is represented as a sparse matrix of edges and a vector of vertex values such that the designed graph algorithm is representable as matrix-vector multiplication. PEGASUS and TurboGraph are two frameworks that work based on the matrix-centric model.

The graph-centric model, introduced by Giraph++ (Tian et al., 2013) is a generalization of vector-centric model, which is based on think-like-a-graph paradigm instead of think-like-a-vertex paradigm. Graph-centric approaches define subgraph-program instead of vertex-program, and the subgraph-programs run a sequential algorithm over the vertices in the subgraph. Subgraphs also include a set of boundary variables from other subgraphs if they have common vertices. These boundary variables are mirrored, so they need synchronization. This generalization reduces the number of synchronization points in compare to vertex-centric models, and in turn, speed up the execution of parallel algorithms. However, the performance of graph-centric models in general and Giraph++ specifically depends on graph partitioning.

General-purpose approaches do not have any specific model for parallelizing the computation, and they are mostly based on data-parallel processing frameworks or fine-grained level of parallelism. GraphCT (Ediger et al., 2013) and GPU-based

libraries (Harish and Narayanan, 2007) are categorized as examples of general-purpose models.

2.3.2. Execution Model

The scope of the vertex-programs or subgraph-programs are the values in the graph representation that are accessible to the programs. For example, in a vertex-centric model, a vertex-program can only access the value of its vertex, the value of out-going edges, and the values of its neighbors.

If two programs that are running in parallel have shared scope, then the race-condition may happen. The execution model discusses how different frameworks control the execution of the parallel programs in order to avoid race-conditions. The main execution models are synchronous and asynchronous.

In synchronous models, the execution of the processes is separated using synchronization points. A synchronization point stops a program from passing it while the execution of the other programs has not reached the same synchronization point yet. In the Pregel framework, the synchronization points happen at the end of each superstep.

The synchronous model is simple, scalable, and provably correct since each process only accesses the previous value of the other processes, so running processes in parallel in synchronous models generates the same result as running them sequentially.

The main disadvantage of the synchronous model is that all programs must wait for the slowest program in the system to pass the synchronization point, which introduces a considerable delay when the amount of computation needed for different programs varies significantly.

GraphX, PEGASUS, Pregelix, and in general, the distributed graph processing systems that are running on top of data-parallel systems are based on synchronous execution model since data-parallel systems run one stage on all data before executing the next stage.

Asynchronous models, on the other hand, do not stop programs at synchronization points, and each program can determine which programs will be executed next as soon as it finishes with its computation. For example, in GraphLab, when a vertex completes its execution, it becomes inactivate until another vertex activates it by sending a message to it.

Asynchronous models lead to faster execution comparing to synchronous models, specifically when the workload of processes are imbalanced. However, race-condition may happen for accessing shared memory, so the system may restrict the parallel execution of programs with shared scope.

GraphLab is the most famous framework that uses asynchronous execution model. PowerGraph can use both asynchronous and synchronous models.

2.3.3. Communication Model

The communication model discusses how the parallel programs in the distributed systems communicate with each other. The communication models of the surveyed frameworks can be categorized to data-centric, message-centric, shared-memory, and disk-based.

Distributed graph processing systems with data-centric communication models usually run a logical view of a program using the data-parallel functions such as a map and reduce. If a program applies map or reduce functions over data that is not local to the worker, the data-parallel system is responsible for moving

the requested data to the worker. GraphX, PEGASUS, and Pregelix use a data-centric model for communicating among programs.

In the message-centric model, different programs communicate with each other through messages. Message-centric model relies on the synchronization points, so the programs can make sure that they received all of the necessary messages. Pregel and MR-MPI are based on message passing. However, MR-MPI uses message passing to provide parallel-data processing functions for the algorithms on top of it.

Shared-memory is the other common communication model, in which the programs have common access to a set of shared variables. If two programs that need communication are not on the same worker, then each uses some copy variables that shadow that state of the other programs. The framework makes sure that the copy variable is consistent with the original variable. GraphLab, Powergraph, GraphCT, and GPU-based library Harish and Narayanan (2007) are using shared memory for communication.

Frameworks with disk-based models serialize the state of programs at every iteration of the computation, so each program can access the previous state of the other programs through loading the previously serialized data. TurboGraph uses disk-based communication in order to process very large graphs using a single PC. Since the graph representation of a very large graph is usually larger than the available memory of a single PC, the data needed by two programs usually do not reside in the memory at the same time. Therefore, each program needs to load the required data, runs the program, and updates its value.

2.3.4. Graph partitioning

Two main partitioning strategies exist for graphs: edge-cut and vertex-cut. In edge-cut, the vertices of a graph are evenly assigned to different machines, so the edges may span across different machines. We can optimize the partitioning to reduce the number of edges that span on different machines (reduce the number of cuts). Vertex-cut, on the other hand, evenly distributes the edges over the machines and may keep multiple copies of a vertex on different machines if the edges that incident with the vertex are assigned to different machines. Here, the communication overhead is to synchronize the information of copied vertices on the machines that store the copies.

Using vertex-cut reduces the communication overhead for a natural graph with power-law degree distributions. Pregel, GraphLab, GraphCT, and Pregelix using edge-cut, while GraphX and Powergraph are using vertex-cut.

2.3.5. Out-of-core computation

When the graph representation is very large, compared to the available memory, out-of-core computation allows the frameworks to use external memory in order to be able to run the programs. Therefore, the frameworks that support the out-of-core computation scale better in existence of the huge amount of data. Pregelix, TurboGraph, MR-MPI, and PEGASUS support out-of-core computation.

2.3.6. Fault tolerance

Fault tolerance is an important feature of any distributed systems since any worker may fail or become inaccessible during its execution. It is more important for distributed graph processing systems to be fault-tolerant since re-running the

graph algorithms on very large graphs is very time-consuming. The frameworks that are developed on the top data-parallel systems such as GraphX, PEGASUS, and Pregelix relies on the data-parallel systems for keeping track of data in case of failure. Other frameworks such as Pregel and GraphLab, Powergraph uses checkpoints for serializing the state of programs into local disks, so in the case of failures, they can restart the execution from the last successful checkpoint. MR-MPI and GPU-based library are not fault-tolerant, so in case of failures, the whole algorithm should be executed again.

2.4. Conclusion

In this chapter, we reviewed a set of distributed graph processing systems and analyzed their architectural aspects. These architectural properties are important in order to choose an appropriate framework for analyzing very large graphs. Based on the discussed properties especially out-of-core processing and fault-tolerance, also the community support and availability of the required hardware, we have decided to use Apache Spark and GraphX as the platform for processing our very large graphs.

CHAPTER III

GRAPHFLOW

The work presented in this chapter has been previously published in Riazi and Norris (2016), and Riazi is the primary contributor to the paper.

In this chapter, we introduce GraphFlow, a workflow-based big graph processing toolkit. The GraphFlow toolkit is a set of new Galaxy compatible tools and offers the rich GraphX graph algorithm API through the higher level of abstraction of Galaxy workflows, which improves usability, reuse, and reproducibility of graph analysis tasks while adding fine-grained parallelism to Galaxy for the first time.

Using GraphFlow we can construct complex data science experiments as a workflow of Spark-based components. Although throughout this paper we focus on Spark as the data-processing engine, we can incorporate other data-processing frameworks in the future.

Figure 3.1. shows the general architecture of GraphFlow. Each new Galaxy tool submits a Spark application to cluster systems or a local machine through the cluster-adapter. The cluster-adapter is a set of cluster system dependent scripts that prepares the inputs of Spark application and wraps the application call from Galaxy with cluster dependent information such as the address of the Spark master and accessible memory to Spark nodes. The cluster-adapter is also responsible to provide Galaxy with the output of the application. This new architecture enables GraphFlow to separate the workflow interface from the data processing. Therefore, Galaxy workflow can be placed on a local machine, e.g., a laptop, while the data engine resides on the cluster system.

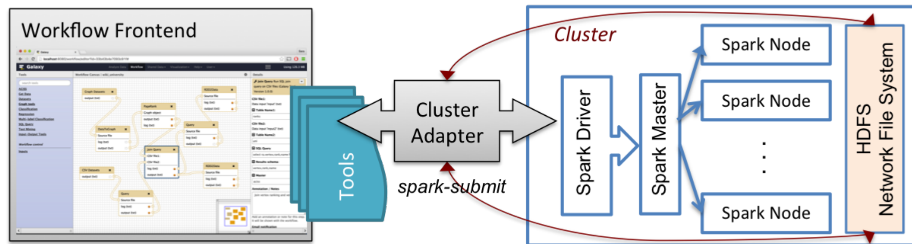

FIGURE 3.1. The architecture of GraphFlow. The Spark-based tools in Galaxy interact with Spark nodes on the cluster system using a cluster-adapter.

3.1. Data Description

The input data provided by Galaxy must be made accessible to Spark applications and output data generated by Spark applications must be accessible to Galaxy. The new cluster-adapter is responsible for this data migration.

In addition, Galaxy expects the input data to be stored as single local file in a conventional file system (not a distributed file system such as HDFS). By contrast, Spark partitions data into multiple files, which may also be distributed over many separate machines (virtual or real).

To address this inconsistency in a MapReduce context, Pireddu et al. (2014) introduce a new functional and extensible integration layer, which enables the users of Galaxy to combine Hadoop-based tools with conventional sequential tools in their workflows.

Their adaptation layer combines the HDFS address of input data files as a *pathset*, which is the list of URIs that defines the input dataset, and passes the constructed pathset to a Hadoop-based tool, which outputs another pathset for the output dataset. The output pathset can be the input of another Hadoop-based tool.

We build on this indirect referencing and introduce the *Metafile* as the input and output format of GraphFlow components. A Metafile is an XML description of the objects, the address type, and object address. By using the information about the address type, the cluster-adaptor can determine whether the object is stored locally, on HDFS, or on a network file system, and can then post the application to the requested cluster system or local machines if the data is available to it. Moreover, to avoid data migration, the address type is used for allocating space for the output data at the same file server as the input data.

Metafiles also include the schema of the data, which helps users attain general understanding about the underlying values because only Metafiles are accessible to users through the Galaxy experiment history.

3.2. Interaction Model

The ultimate goal of GraphFlow is to provide a workflow-based environment that is capable of encoding complex graph analytic experiments. Each GraphFlow component is a Spark application that manipulates a distributed collection of objects stored as dataframes. By using this representation, we define each GraphFlow component as either: (a) a complex map function that transforms a dataframe or a graph object to another dataframe, graph or a combination of these; or (b) a reduce function of a dataframe or a graph into a single data file, a set of aggregated values or charts.

Loading and storing typed collection objects such as RDDs reduces the generalization of the each component because RDDs have to be manipulated differently based on the type of the objects they are encapsulating. For better generality, each GraphFlow component expects the input to be in a named column

format, such as a CSV file. Each GraphFlow component loads the input CSV file into a dataframe and maps it to another dataframe. Finally, the component stores the dataframe as another CSV file. The CSV files are multi-part files, so GraphFlow components expect a Metafile as input that contains the schema of these CSV files and their addresses, and outputs another Metafile. The schema of an output Metafile may be different from the schema of the input Metafile. We use the Spark-CSV library[1] for I/O of dataframes.

3.3. GraphFlow Components

The GraphFlow components are grouped into general input/output tools, graph analytic tools, relational tools, and plotting tools. All GraphFlow components return a log file in addition to their expected output. This log output is a single text file understandable by Galaxy. The log files usually includes a small sample of output dataframes and the execution log of the tool (useful for debugging). For simplicity, we do not explicitly mention the log output in the description of each tool. Next, we describe GraphFlow components in more detail.

GraphFlow's **I/O tools** include components used to convert single-file data into dataframes and graph objects, and also to convert them back to single-file data. Since the aim of GraphFlow is to process big graph data, we expect GraphFlow's users to upload their big data files directly to the cloud storage (e.g., Amazon S3 if using Amazon AWS) instead of uploading through the Galaxy Web interface, and use their corresponding Metafile of their data as input to the GraphFlow's components. Therefore, we provide a basic *MetaLoader* component, which takes the file information from users and constructs a Metafile for it. The

[1] `https://github.com/databricks/spark-csv`

MetaLoader component can be used as the initial component of any workflow. *DFDump* can be used for converting a distributed dataframe back to a single file, which is downloadable through Galaxy interface. GraphFlow has two more similar components *GraphLoader* and *GraphDump* for loading a distributed graph object from a single file and for dumping a graph object into a single file, respectively.

GraphFlow provides a collection of **graph tools** that include algorithms for generating and processing big graphs: *GraphGen*, *PageRank*, *DegreeCount*, *TriangleCount*, *Subgraph*, *LargestCC*, *GraphCluster*, *ClusterEval*, and *GraphCoarsen*. The *GraphGen* components support generating random graphs using log-normal degree distribution and RMAT (Chakrabarti et al., 2004).

PageRank is a well-known graph vertex ranking algorithm introduced by Google for ranking Web pages. GraphFlow's *PageRank* component takes a graph object and outputs a dataframe with two columns of vertex IDs and rank value, for which the rank values are computed using the PageRank algorithm provided by Apache Spark's GraphX library.

Similar to the *PageRank* component, the *DegreeCount* and *TriangleCount* components take a graph object and return a dataframe of vertex IDs and degree counts, and a dataframe of vertex IDs and triangle counts, respectively.

The subgraph function in GraphX constructs a subgraph of the original graph. The user must provide either an edge or a vertex indicator function. The purpose of the indicator function is to determine whether the given edge or vertex belongs to the resulting subgraph. In order to utilize the indicator function, we represent any discrete function f as a dataframe of x and $f(x)$. For the vertex indicator, x is a vertex ID, and f is a boolean function. The *Subgraph* component in GraphFlow takes a graph object and a dataframe representing an indicator

function, and returns two graph objects: one for the subgraph corresponding to the indicator function and the other for the complement of that. Another component is *LargestCC*, which outputs the subgraph of a graph's largest connected components.

GraphFlow also includes a set of graph clustering algorithms such as PIC (Lin and Cohen, 2010), spectral clustering (Spielmat and Teng, 1996), and label propagation. We use the Spark implementation for PIC and label propagation, and add our implementation for spectral clustering. *GraphCluster* takes a graph and returns two outputs. The first output is a graph object called a cluster graph, in which the attribute of each vertex is the cluster number of that vertex. The other output of *GraphCluster* is a dataframe of vertex IDs and cluster numbers, which can be transformed to an indicator function using the query component (described later), so one can easily create a subgraph of nodes belonging to a particular cluster.

To measure the quality of a clustering, we created a GraphFlow *ClusterEval* component that implements two clustering metrics, modularity (Brandes et al., 2008) and normalized cut (Shi and Malik, 2000). This tool takes a cluster graph (as described above) and computes the modularity and normalized cut. We can consider the *ClusterEval* component as a reduce function that reduces a distributed graph object to a single value. We implemented the modularity and normalized cut computations using the Spark GraphX API.

Finally, we created a *GraphCoarsen* component that can be used to simplify a big graph. *GraphCoarsen* takes a cluster graph as input and replaces a set of vertices in a cluster with a super vertex. The output is a graph object where each super vertex attribute is the number of vertices that form the supper vertex. The

FIGURE 3.2. The Query tool expects a table name and query on the given table name. Providing the Query tool with the output schema is optional.

coarsening implementation is based on the pseudocode provided in Gonzalez et al. (2014).

The **Relational tools** consist of *Info*, *Query*, *JoinQuery*, and *PredefinedQueries* components and are an important part of every workflow represented in GraphFlow because we can use them to transform or constrain dataframes or to join the output of multiple components.

The *Info* tool collects the schema, the number of available data points, and some samples of data points from the given dataframe in order to guide the user in constructing valid queries.

The *Query* component runs an SQL query over the given input dataframe. In order to run a query over a dataframe, it first registers the input dataframe as a relational table with the given name, and then executes the query on the relational table. Figure 3.2. shows the parameter of page of the *Query* tool and an example SQL query. The *Query* component also expects the schema of the output dataframe in order to construct appropriate named columns. The given names are specifically useful when we want to run other queries on the output dataframe.

To simplify using the relational queries, we provide a set of common queries in *PredefinedQueries*.

The *JoinQuery* component is similar to *Query*, except it accepts two dataframes as inputs, so we can run join queries on both dataframes. Similar to *Query*, we provide names for the tables, schema for the results, and the SQL query. *JoinQuery* is specifically useful when we want to combine the information of two dataframes.

Statistics tools: the goal of these components is to collect statistics from the dataframe. Cumulative density function (CDF) has been extensively used in practice to study the data distribution, which is also provided here.

Plotting tools: includes different plotting components such as ScatterPlot, and HistogramPlot which summarizes a dataframe for further analytic studies.

We can also create more complex components by combining these tools, for example, we can create a hierarchical clustering workflow by unrolling few iterations of the recursive calls of *GraphCluster*[2] in combination with *Subgraph*, and *Query* components as shown in Figure 3.3.. In this workflow, the *GraphCluster* is configured with a maximum of two clusters. Then, we use the cluster assignment to select the vertices that belong to one cluster and feed that to *Subgraph* along with the cluster graph. *Subgraph* partitions the given graph into two subgraphs, each belonging to one cluster. Finally, we apply *GraphCluster* to each of these subgraphs.

[2]Galaxy workflow engine do not support recursive diagrams.

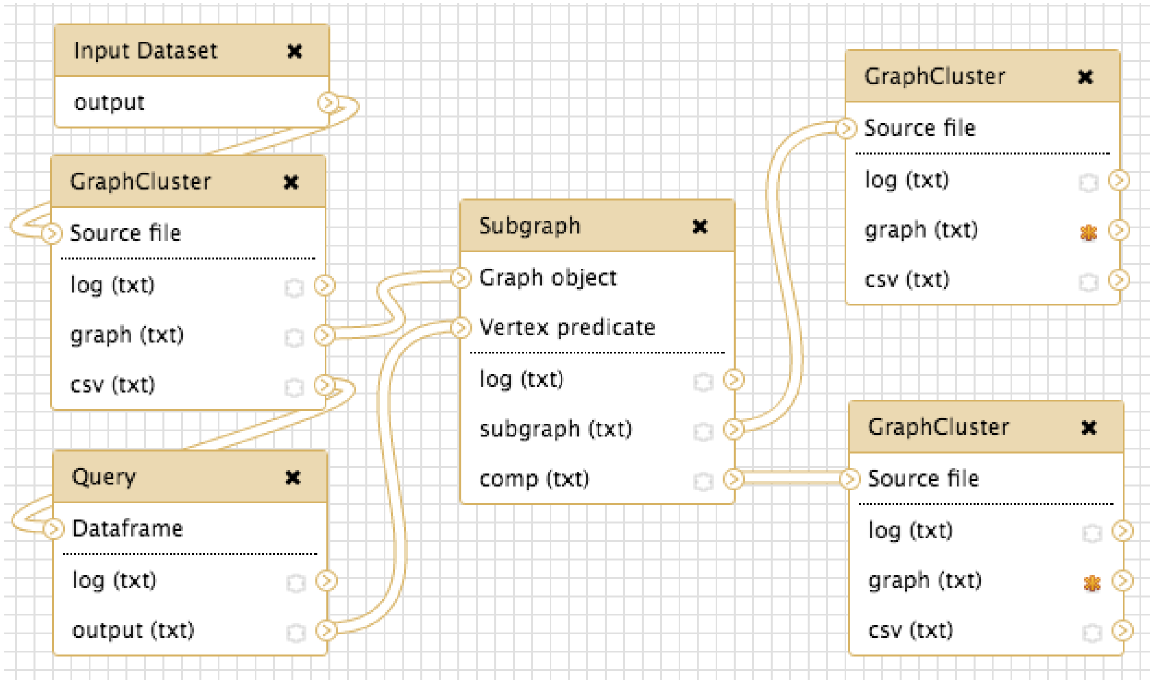

FIGURE 3.3. The workflow of hierarchical clustering using Subgraph, GraphCluster, and Query.

3.4. Use Cases

In order to show the expressiveness of the GraphFlow components, we construct different workflows to study the structural properties of graphs constructed from the Wikipedia datasets[3]. This dataset is a crowd-source gathered information from Wikipedia and includes several data files such as page links and abstracts. Each line in the page link dataset contains a pair of URIs such that the second URI appears in the Wikipedia Web page of the first URI. As an example of URIs, "http://dbpedia.org/resource/Stanford_University" is the URI of Stanford University Wikipedia page. The abstracts includes the URI of a Wikipedia page and the main section of each page.

[3]http://wiki.dbpedia.org/

In order, to construct the Wikipedia graph, we assign a unique ID to each URI, which identifies a vertex in the graph. Two vertices are connected if the pairs of their URIs appear together in the page links dataset. We simply ignore the order of URIs in each pair, so the final graph is undirected. The constructed graph consists of more than 20 million vertices and 159 million edges. Moreover, we keep the URIs and the assigned IDs in a CSV file as URI data file, which we use for finding the corresponding URI assigned to each vertex.

For these experiments, we ran the cluster system (Figure 3.1.) on the ACISS cluster[4], and we ran the Galaxy front-end on a laptop. We used five Spark nodes, each running on an Intel(R) Xeon(R) CPU X5650@2.67GHz with access to a total of 50GB of memory.

3.4.1. Degree Distribution

Degree distribution is well-studied metric for graphs. In order to find the degree distribution, we first use the Node Degree components to get the degree of each vertex as a CSV file with schema "vertex,degree", then the following SQL queries gives us the distribution:

```
SELECT degree, count(degree)
FROM degreeTable
GROUP BY degree,
```

where the degreeTable is the relation name that we use to register the input degree CSV file. Finally we redirect the output to the plotting component.

The degree distribution of Wikipedia graphs mostly follows power-law degree distribution, Figure 3.8..

[4] http://aciss-computing.uoregon.edu/

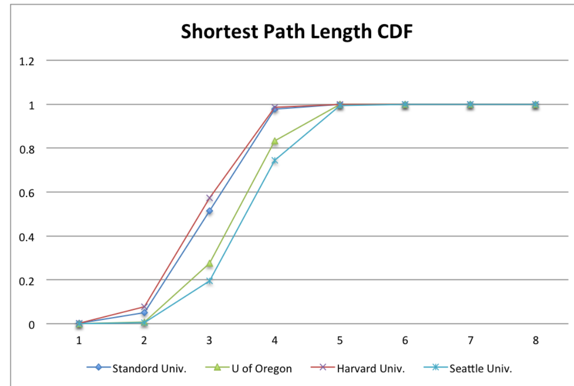

FIGURE 3.4. CDF of the shortest path length from the all nodes of the graph to vertices of Harvard University, Stanford University, University of Oregon, and Seattle University.

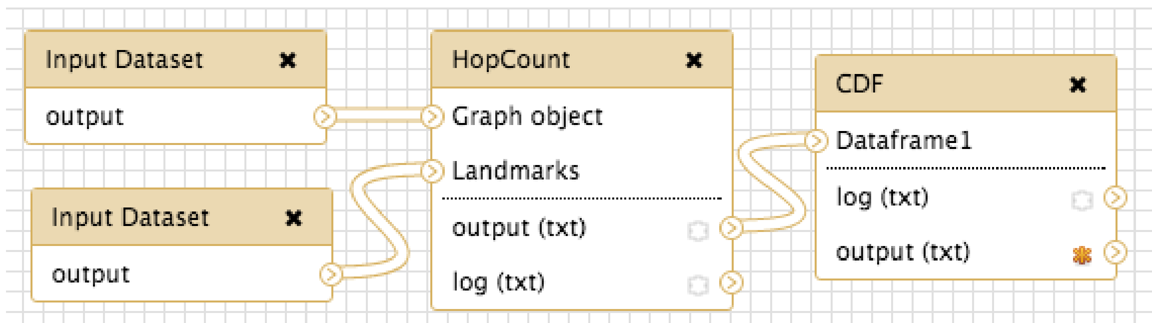

FIGURE 3.5. The workflow of finding the CDFs of shortest paths.

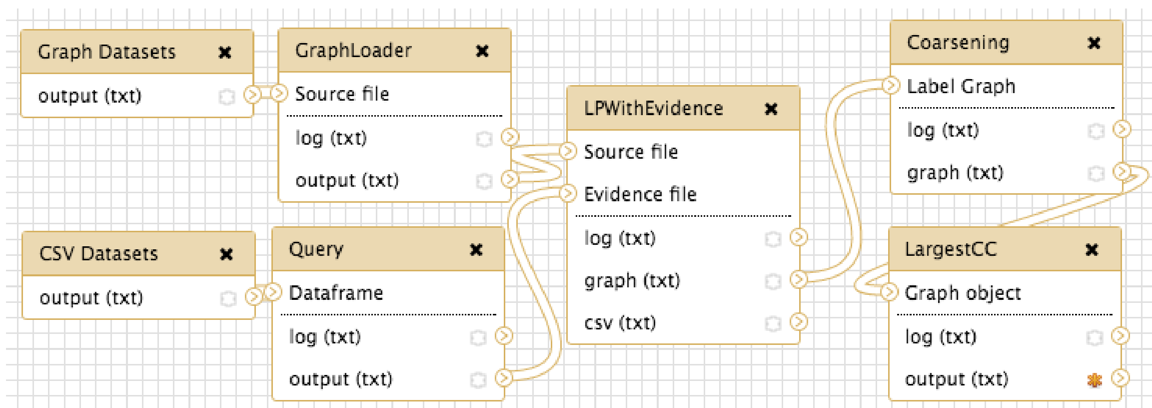

FIGURE 3.6. The workflow of coarsening a graph using clustering.

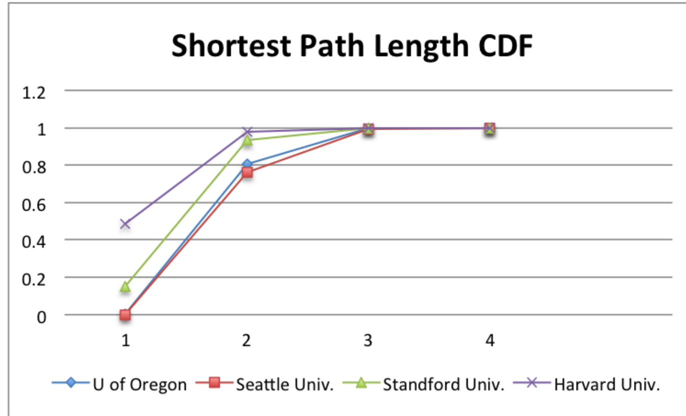

FIGURE 3.7. CDF of the shortest path length in the coarse graph.

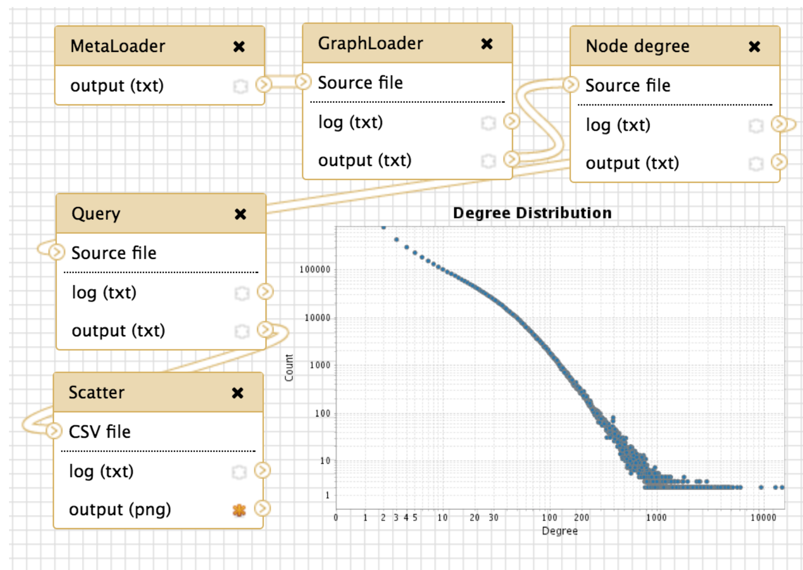

FIGURE 3.8. The degree distribution of Wikepedia graph along with the corresponding workflow.

3.4.2. Shortest-Path Length Distribution

Shortest paths length in a graph has been used for defining the closeness centrality, which shows the relative positions of a given vertex with respect to all other vertices in the graph. However, looking at the shortest-path length distribution is more informative. The *ShortestPath* component generates the shortest path lengths from each vertex in the graph to a set of predefined landmarks. We use the set of vertices corresponding to different universities as landmarks, and generate the cumulative density function (CDF) for each of these universities. Figure 3.4. shows these CDFs, which indicate the closeness of the landmarks with respect to other vertices in the graph. For example, approximately 45% of shortest paths toward the Standford University page have length smaller or equal to 3, while this value is only 20% for Seattle University.

3.4.3. Coarsening

Coarsening of very large graphs enables analysis with fewer resources. However, the coarsening process should preserve the properties of the original graph. For example, suppose we are interested in a subgraph of the Wikipedia graph that includes the pages of universities, colleges, institutes, and related pages. We select the pages if their URIs include University, Institute, or College, and refer to them as academic pages. Using the *Subgraph* component may result in removing all pages not belonging to set of vertices of the interest and ignoring their effect on the coarse graph. For example, the Oregon Ducks Football team page will not appear in the set of vertices and *Subgraph* ignores the paths that connected University of Oregon to universities thorough their football pages. Therefore, we need to find a community around each page of interest. This is similar to local

clustering. For this purpose, we modify the label propagation algorithm and put a weight on each label. Setting uniform weights reduces the local label propagation to original label propagation. For our purpose, we set the weights of labels belonging the academic pages to large values, while all other weights are set to one. This forces communities to be formed around the academic pages.

We feed the output of the local label propagation algorithm, which is a cluster graph (where the attribute of every vertex is its cluster ID) to the coarsening component and obtain its largest connected components. This workflow is shown in Figure 3.6.. There are 140K academic pages, however, the largest connected component of the coarse graph has only 14K vertices, compared to the 20M vertices of the original Wikipedia graph. To check whether the coarse graph preserves the structure, we look at the CDF of the shortest paths of the same universities studied in previous section. We can easily feed the output of the coarsening workflow, Figure 3.6. to the shortest path workflow, Figure 3.5.. Figure 3.7. shows the resulting CDF of the shortest paths to the given landmarks, indicating that the coarse graph has structure similar to that of the original graph.

3.4.4. Pagerank Centrality

PageRank is a well-known variation of eigenvector centrality. With PageRank, we can sort the vertices based on their rank score. Our goal in this use case is to rank universities based on their appearance in the Wikipedia using the PageRank algorithm Lages et al. (2015). Therefore, the rank of a university depends on the Wikipedia pages that have links to the Wikipedia page of the university and importance of those pages based on the ranking.

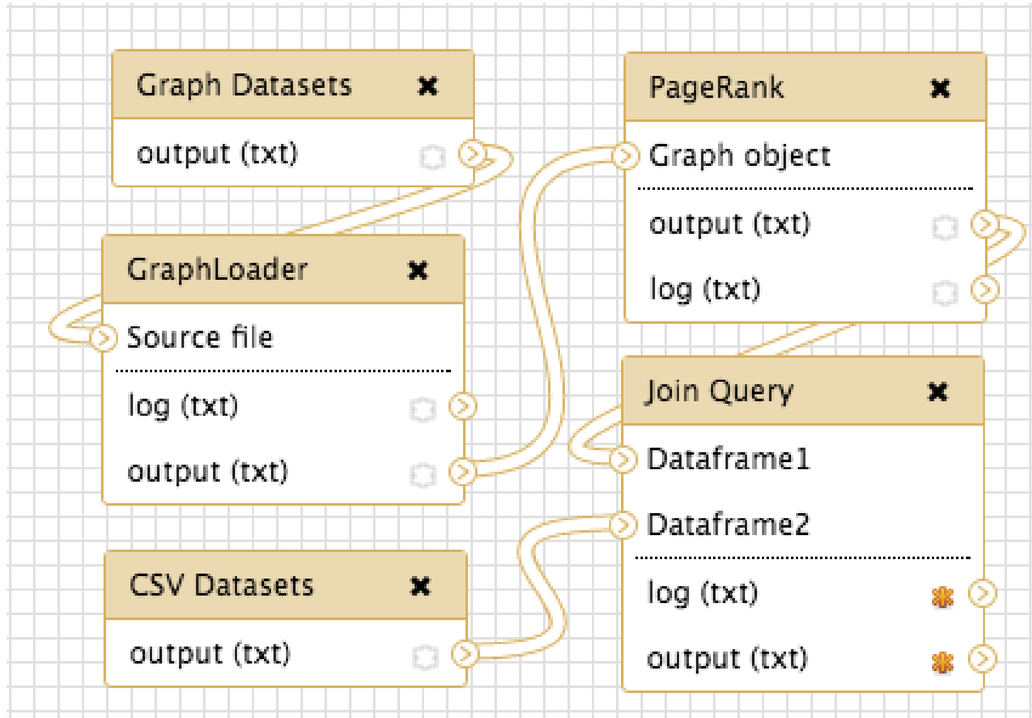

FIGURE 3.9. The workflow of ranking universities using Wikipedia graph.

To find the Wikipedia pages of universities we simply use the URI name and look for related words such as University, Institute, or College. An alternative approach would be to use the abstract file, but here the URI name seems sufficient. Therefore the result of the search is a dataframe that includes the ID and URI of universities.

Figure 3.9. shows the workflow of the experiment. The graph dataset points to the edge-view of the Wikipedia graph constructed from the Pagelink file, and the CSV dataset points to the URI data file.

The *PageRank* tool ranks the vertices of the Wikipedia graph, and the output dataframe is given to *JoinQuery* tool. The SQL query given to the *JoinQuery* joins two dataframes, so we access the URI of each vertex as well as its rank. We can then restrict the results to the URIs. The *JoinQuery* register the output of the

TABLE 3.1. Top 10 universities found using workflow of Figure 3.9. compared to Wikipedia university ranking from Lages et al. (2015) and the survey-based rankings Times Higher Education (2016).

	Ranking from Lages et al. (2015)	GraphFlow	QS Ranking Times Higher Education (2016)
1st	Univ. of Cambridge	Harvard Univ.	MIT
2nd	Univ. of Oxford	Univ. of Oxford	Stanford Univ.
3rd	Harvard Univ.	Columbia Univ.	Harvard Univ.
4th	Columbia Univ.	Univ. of Cambridge	Univ. of Cambridge
5th	Princeton Univ.	Yale Univ.	CalTech
6th	MIT	Stanford Univ.	Univ. of Oxford
7th	Univ. of Chicago	UC Berkeley	Univ. College London
8th	Stanford Univ.	MIT	ETH Zurich
9th	Yale Univ.	Univ. of Michigan	Imperial College London
10th	UC Berkeley	Princeton Univ.	Univ. of Chicago

PageRank and *Query* tools as relational tables *ranks* and *univ*, respectively, and runs the following SQL query on them:

```
SELECT name, rank from uri, ranks
WHERE ranks.vertex = uri.vertex
AND (name LIKE "%University%"
OR name LIKE "%Institute%"
OR name LIKE "%College%" )
ORDER BY ranks.rank DESC limit 100
```

Table 3.1. includes the top 10 of the final ranking result produced by our example workflow, the Wikipedia ranking reported by Lages et al. Lages et al. (2015), and the QS survey-based ranking Times Higher Education (2016). The Wikipedia-based top 10 lists have nine common entries. The difference in Wikipedia-based rankings is most likely attributable to the fact that we only used English Wikipedia pages while Lages et. al use all provided Wikipedia pages.

3.5. Conclusion

We introduced the GraphFlow toolkit, a workflow-based system for large-scale distributed graph analysis. GraphFlow provides the user with a set of Spark-based tools that can be combined together using the intuitive Galaxy workflow manger in order to describe complex data science experiments. Using GraphFlow, researchers can re-run their complex experiments with different parameter settings and over different input data. Moreover, workflows can be shared, reused, or composed into larger applications, as shown in the case studies. GraphFlow hides the complexity of interacting with cluster systems and data-parallel processing frameworks, significantly simplifying large-scale graph analysis.

CHAPTER IV

GRAPH EMBEDDING

Most of the work presented in this chapter has been previously published in Riazi and Norris (2019), and Riazi is the primary contributor to the paper.

Graph embedding (a.k.a. network embedding) is an important step in solving many graph problems including link prediction, vertex classification, and clustering. Network embedding aims to learn a low dimensional vector representation for vertices of a graph. However, existing approaches do not scale to very large graphs with billions of vertices and edges. One solution is to use distributed memory systems and out-of-core computation. Among distributed memory systems, frameworks such as the Apache Spark-based GraphX (Gonzalez et al., 2014) are of particular interest to us because they offer a map-reduce-based approach to expressing distributed-memory parallel algorithms for graph computations.

However, to take advantage of such distributed graph processing frameworks, we need to design new map-reduce (Dean and Ghemawat, 2008) network embedding algorithms. In general, following the previous work for learning general network embedding (Perozzi et al., 2014; Tang et al., 2015; Grover and Leskovec, 2016), we use the structural properties of a network to train an embedding. A common assumption underlying existing methods and our new algorithm is that we expect that the embedding of a vertex is more similar to the embeddings of its neighbors rather than to the embedding of a random vertex outside of its neighborhood. We enforce this objective with approximate maximum likelihood training of the embedding in which the partition function is approximated using negative samples. This training requires lookup access to the embedding of

vertices in a neighborhood, as well as vertices that lie outside of the neighborhood. However, lookup access in map-reduce frameworks is prohibitively expensive, which necessitates careful consideration in developing a map-reduce based network embedding algorithms. In this chapter, we introduce such an algorithm and experimentally show that we can train network embedding for very large graphs. We evaluate the new algorithm's accuracy and parallel scalability on a set of real-world networks.

4.1. Sequence Representation

The structure of graphs can be captured by sets of random walks starting at every vertex of the graph. Each of these random walks forms a sequence of vertices. Therefore, the algorithms for word representation that uses sequences of words (sentences) as the input can also be exploited for graph representation (Perozzi et al., 2014). Word representation or word embedding is an important tool in language modeling Bengio et al. (2003), which helps algorithms to extract similar words. The idea is that given a corpus, similar words would appear within similar context. A context of a word is the set of surrounding words in the same sentence.

The basic word representation is a 2-gram or one-word context, in which we only care about the co-occurrence of a word and its context that includes only one word. Basically two words are similar if they appear within similar context more often. An N-graph is the generalization of 2-gram which focuses on the appearance of similar words in similar contexts that have more than one word. The most well-known word representation learning algorithm is Skip-gram (Mikolov et al., 2013b,a), which we discuss in more detail.

Given a corpus with n words, originally each word w is represented using an one-hot-encoding vector, which is an n-dimensional binary vector that has one entry for each word in the corpus. The one-hot vector representation of w has only one non-zero element located in the column corresponding to word w. The objective is to learn a d-dimensional vector representation for each word, such that $d \ll n$ and similar words have close vector representations.

Skip-gram measures the similarity of words based on their context. The context of a target word w is a window of words surrounding the target word, which is called the context words c. The objective function of Skip-gram is to maximize the probability of predicting the context words given target words:

$$\max \sum_{w \in \mathcal{V}} \sum_{c \in V_c} \log P(c|w), \quad \text{(Equation 4.1)}$$

where $\mathcal{V}$ is word vocabulary and and $\mathcal{V}_c$ is the context vocabulary, which may be considered to be equal to $\mathcal{V}$. Skip-gram estimates $P(c|w)$ using a softmax function:

$$P(c|w) = \frac{exp(\Phi(w)^T \Phi(c))}{\sum_{c' \in \mathcal{V}} exp(\Phi(w)^T \Phi(c'))}, \quad \text{(Equation 4.2)}$$

where $\Phi(.)$ is a function from vocabulary space to a d-dimensional vector representation.

However, because the size of the context vocabulary is often very large, computing the denominator in the above softmax is prohibitive. To overcome this obstacle hierarchical softmax (Morin and Bengio, 2005) and negative sampling (Mikolov et al., 2013c; Dyer, 2014) have been widely used.

The idea of hierarchical softmax is to group words into classes in order to reduce the summation. If we can predict the class of each word, then we only

have to do the summation for the words belonging to that class, which reduces the required computation significantly. Morin and Bengio (2005) propose using hierarchical clustering, in which they form a binary tree of classes, and each intermediate node only predicates whether the word belong to the left or right sub-classes. They use softmax, for prediction at intermediate node:

$$P(b_l = 1|w) = \sigma(\Psi(b_l)^T \Phi(w)), \qquad \text{(Equation 4.3)}$$

where $\Psi(.)$ is the vector representation of each intermediate node and $\sigma(.)$ is the sigmoid function: $\sigma(x) : 1/(1 + \exp(-x))$. Since the variable of the intermediate nodes are binary, we don't need to compute the normalization constant by simply selecting $P(b_l = 0|w) = 1 - P(b_l = 1|w)$. Using hierarchical softmax, Relation Equation 4.2 can be computed using:

$$P(c|w) = \prod_{l=1}^{\lceil \log |\mathcal{V}| \rceil} P(b_l|w), \qquad \text{(Equation 4.4)}$$

which needs evaluating $\lceil \log |\mathcal{V}| \rceil$ different softmax functions and is exponentially more efficient than computing Equation 4.2. For example, Figure 4.1. shows a factorization for computing $P(v_3|\Phi(v1))$ as $P(b1 = 0|\Phi(v1))P(b2 = 1|\Phi(v1))P(b5 = 0|\Phi(v1))$.

Mikolov et al. (2013c) introduce negative sampling as another way to deal with the computational complexity of the normalization constant of the softmax relation (Equation 4.2). Negative sampling penalizes the co-occurrence of random context words and the target words. Therefore, the objective function of skip-gram

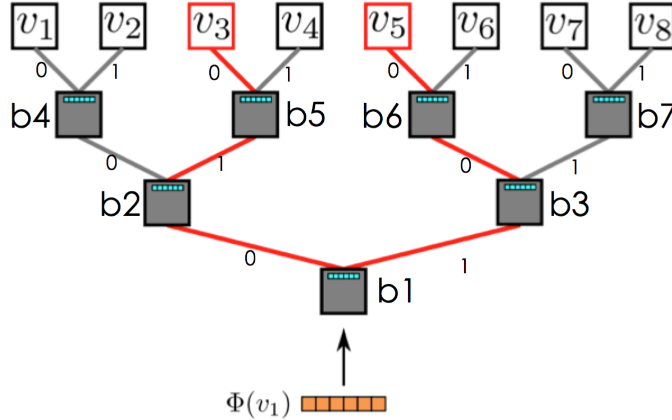

FIGURE 4.1. (Perozzi et al., 2014) Hierarchical softmax for computing $P(v_i|\Phi(v))$. Each intermediate node defines $P(b_l|\Phi(v))$, which be trained using logistic regression.

becomes the following:

$$\sigma(\Phi(c)^T\Phi(w)) + \sum_{i=1}^{k} E_{c'\sim P_D}[\log\sigma(-\Phi(w)^T\Phi(c'))], \qquad \text{(Equation 4.5)}$$

where P_D is an empirical unigram distribution: $P_D(c) = \frac{\#(c)}{D}$.

Levy and Goldberg (2014) show that optimizing the above objective function is similar to factorization of matrix M, whose elements, M_{ij}, are shifted point-wise mutual information (PMI) of words and contexts: $M_{ij} = \log\frac{\#(w,c)|D|}{\#(w)\#(c)} - \log k$. To learn the representation using matrix factorization the goal is to reconstruct the matrix M as the linear product matrix $U \in \mathbb{R}^{|\mathcal{V}|\times d}$ and $V \in \mathbb{R}^{|\mathcal{V}|\times d}$:

$$\min_{U,V} M - UV^T, \qquad \text{(Equation 4.6)}$$

where there rows of U and V are the vector representations of target words and context words, respectively.

4.2. Network Embedding

Similar to word representation, the goal of network embedding (a.k.a graph representation) is to learn a low-dimensional vector for each vertex in the graph such that the vector representation carries the structural properties of the graph. Formally, for graph $G(\mathcal{V}, \mathcal{E})$ of vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, we want to learn a d-dimensional vector representation $\Phi(v)$ for each $v \in \mathcal{V}$ such that $d \ll |\mathcal{V}|$.

DeepWalk (Perozzi et al., 2014) suggests using a model similar to the Skip-gram model for learning $\Phi(v)$, which maps vertex v to its vector representation. DeepWalk relates each vertex to one word and the set of random walks on the graph G to the corpus. Using this relationship, DeepWalk successfully applies the Skip-gram model for learning the graph representation. DeepWalk generates a set of fixed-length random walks R_v starting at every vertex v of the graph. Then for every vertex v_j of random walk R_v, it considers the vertices surrounding v_j (in a window centered at v_j) as the context of v_j. Finally, the representation vector of v_j, $\Phi(v_j)$, is calculated by optimizing the following relation:

$$\max_{\Phi} \sum_{i \in \{j-w, \cdots, j-1, j+1, \cdots j+w} \log P(v_i | \Phi(v_j)), \qquad \text{(Equation 4.7)}$$

where the size of the window is $2w$.

DeepWalks uses hierarchical softmax to compute the probability of $P(v_i|\Phi(v_j))$.

Node2vec (Grover and Leskovec, 2016) is another successful representation learning approach for graphs which is similar to DeepWalk in term of objective function and using random walks, however it uses negative sampling instead of hierarchical softmax to overcome the intractability of Equation 4.2. Moreover,

Node2vec defines the neighborhood of node v_j as its context, and introduce methods for extracting the neighborhood of a vertex. The main difference between the random walk exploration and neighborhood exploration is in introducing a search bias α, which controls selecting the next node to visit not only based on the current node, but also on the previous node. To select the next node to visit we need to sample from:

$$P(v_j = x|v_{j-1} = v, v_{j-2} = t) = \left\{ \begin{array}{cc} \frac{\alpha(t,x)w_{xv}}{Z} & \text{if (v,x)} \in E \\ 0 & \text{otherwise} \end{array} \right\}, \qquad \text{(Equation 4.8)}$$

where Z is the normalization constant, and α is defined based on the shortest path distant d_{tx} between node t and x as the following:

$$\alpha(t,x) = \left\{ \begin{array}{cc} \frac{1}{p} & \text{if d}_{tx} = 0 \\ 1 & \text{if d}_{tx} = 1 \\ \frac{1}{q} & \text{if d}_{tx} = 2 \end{array} \right\}, \qquad \text{(Equation 4.9)}$$

where p and q are positive parameters that control neighborhood exploration. p controls how often the random walk revisits the previous node, and q controls how often the random walk explores nodes that are not immediate neighbors of the previous node. For example, for $p \gg 1$ and $q \ll 1$ results in random walks which are more likely emulating depth-firth search, while the random walks generated with $p > g \gg 1$ are more likely emulating breadth-first search.

Although DeepWalk and Node2vec are successful in learning graph representations, they mostly suffer from the fact that random walks only capture local structural properties of graphs; therefore, what they learn mostly depends on what random walks can capture. Moreover, to learn the representation of

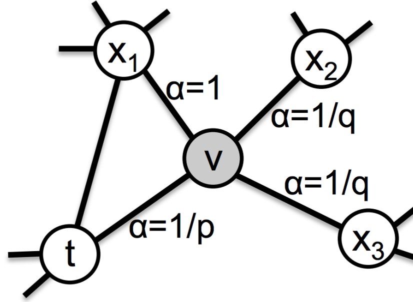

FIGURE 4.2. (Grover and Leskovec, 2016) Neighborhood exploration using search parameters p and q. The goal is to select the next node to visit given that the current and previous nodes are v and t, respectively.

large-scale graphs, they may need many random walks for each vertex which is prohibitive.

Tang et al. (2015) address these problems by defining an objective function which directly depends on the structure of graph instead of relying on random walks for capturing the structure of input graphs. This objective function is based on the definition of proximity in graphs, which includes first-order proximity and second-order proximity.

First-order proximity is the pairwise similarity between two vertices v_i and v_j, defined as the joint probability distribution over both of them: $P(v_i, v_j) = \sigma(\Phi(v_i)^T \Phi(v_j))$.

Second-order proximity is the pairwise similarity between two vertices v_i and v_j that share similar context or neighborhood, and is defined using $P(v_i|v_j)$:

$$p(v_i|v_j) = \frac{\exp(\Phi(v_i)^T \Phi(v_j))}{\sum_{k=1}^{|\mathcal{V}|} \exp(\Phi(v_k)^T \Phi(v_j))} \qquad \text{(Equation 4.10)}$$

Tang et al. (2015) define empirical distributions based on the graph structure for both $P(v_i, v_j)$ and $P(v_i|v_j)$, and then minimize the distant between empirical distribution and the model defined as KL-divergence.

The empirical distribution for $P(v_i, v_j)$ is defined as $\frac{w_{ij}}{W}$, where W is the total weights of the edges in the graph and w_{ij} is the weight of the edge between vertex i and vertex j. Similarly, the empirical distribution for $P(v_i|v_j)$ is defined as $\frac{w_{ij}}{\sum_i w_{ij}}$.

Therefore, minimizing KL-divergence for these two models results in the objective functions O_1 and O_2 for the first-order and second-order proximity, respectively:

$$\begin{aligned} O_1 &: \min_{\Phi} - \sum_{(i,j)\in\mathcal{E}} w_i j \log P(v_i, v_j) \\ O_2 &: \min_{\Phi} - \sum_{(i,j)\in\mathcal{E}} w_i j \log P(v_i|v_j) \end{aligned} \qquad \text{(Equation 4.11)}$$

Tang et al. (2015) experimentally show that optimizing either O_2 or $O_2 + O_1$ is learning a better representation comparing to DeepWalk.

4.3. Vertex-Centric Network Embedding

The existing network embedding algorithms do not scale to very large graphs, so to address this problem we introduce vertex-centric network embedding based on GraphX. The goal of vertex-centric network embedding is to learn a low-dimensional vector for each vertex in the graph such that the vector representation carries the structural properties of the graph. Formally, for graph $G(\mathcal{V}, \mathcal{E})$ of vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, we want to learn a d-dimensional vector representation $\mathbf{u}_i$ for each $i \in \mathcal{V}$ such that $d \ll |\mathcal{V}|$.

Many approaches have been introduced to learn a vector representation (Perozzi et al., 2014; Tang et al., 2015; Grover and Leskovec, 2016; Hamilton et al., 2017; Veličković et al., 2018) aiming to encode a vertex's neighborhood (its structural properties) into a low-dimensional space. Other properties of vertices, such as attributes, labels, and relations can also incorporated into the vector representation of the vertex (Duran and Niepert, 2017; Lin et al., 2015; Yang et al., 2015; Pan et al., 2016).

In general, a graph embedding approach is vertex-centric friendly if the embedding of each vertex is a function of only the embeddings of its neighbors. For example, LINE-1st (Tang et al., 2015) computes the embedding using first-order proximity by optimizing the following objective function:

$$\max_{\mathbf{u}} \sum_{(i,j)\in E} w_{ij}\sigma(\mathbf{u}_i^T\mathbf{u}_j), \qquad \text{(Equation 4.12)}$$

in which $\mathbf{u}_i$ and $\mathbf{u}_i$ are vector representations of vertex i and j, respectively, σ is a sigmoid function, and w_{ij} is the edge weight. We can rewrite Equation 4.12 as

$$\max_{\mathbf{u}} \sum_{i} \sum_{j\in N(i)} w_{ij}\sigma(\mathbf{u}_i^T\mathbf{u}_j), \qquad \text{(Equation 4.13)}$$

where $N(i)$ is the set of neighbors of vertex i.

More powerful representation learning methods, such as LINE second-order proximity, consider the embeddings of neighbors *and* the embeddings of random vertices selected among non-neighbor nodes (negative samples), contrasting them to learn the embedding of each vertex:

$$\max_u \sum_{j \in N(i)} w_{ij} \sigma(\mathbf{u}_i^T \mathbf{u}_j) + \frac{-1}{k} \sum_{j \notin N(i)}^{k} w_{ij} \sigma(\mathbf{u}_i^T \mathbf{u}_j) \qquad \text{(Equation 4.14)}$$

Negative samples make sure that the objective function does not find a trivial solution (e.g., the embedding of all vertices become the same). Negative sampling simply forces the embeddings of non-neighbor nodes to become different.

In a vertex-centric paradigm, we are required to decompose the algorithm such that each vertex is responsible for its part of the objective function evaluation, providing all the necessary information, e.g., the current state of its neighbors. In other words, we look at the computation from a vertex point of view. We can simply view network embedding of Equation 4.14 in a vertex-centric paradigm: "As a vertex, I want my embedding to be similar to my neighbors' embeddings, while it differs from the embeddings of other non-neighbor vertices". A vertex-centric network embedding requires the objective function to decompose as partial objectives computable at individual vertices, but unfortunately the objective of Equation 4.14 does not decompose over vertices.

In a vertex-centric setting for optimizing based on Equation 4.14, each vertex needs to access the embeddings of vertices that are not directly connected to it (negative sampling). Parallel graph frameworks do not provide efficient lookup of random vertices that are distributed among different machines. Moreover, each computing node does not have a lookup dictionary that can be used to locate and ship the attributes of required vertices, but there are routing tables for vertices based on the edges that are connecting them, so accessing the neighboring vertices is efficient (compared to random lookup access).

To benefit from this efficiency, we define a random graph, in which each vertex is connected to k randomly selected vertices in the graph with a negative weight, which can be uniformly set to one. We construct a new graph as the union of the current graph and the random graph. In the new augmented graph, each vertex has access to the embedding of k randomly chosen vertices. Therefore, we can rewrite Equation 4.14 with our augmented graph:

$$O_i = \max_u \sum_{j \in A(i)} w_{ij} \sigma(e_w \mathbf{u}_i^T \mathbf{u}_j), \qquad \text{(Equation 4.15)}$$

where e_w is negative one for negative samples and positive one for the actual neighbors, and $A(i)$ is the set of neighbors of vertex i in the augmented graph. We can derive Equation 4.15 from Equation 4.14 by using the symmetry in the sigmoid function: $\sigma(-x) = -\sigma(x)$ and absorbing k in the weights.

The objective function of Equation 4.15 decomposes over vertices in the augmented graph, so it can be computed in a vertex-centric approach unlike the negative sampling-based approach in the original graph, whose objective function is not decomposable.

4.3.1. Vertex-Centric algorithm

A data-parallel vertex-centric graph algorithm typically involves three steps: sending messages among neighbors (sendMessage), reducing all the messages to a single vertex to one message (mergeMessage), and executing a vertex related function given the final reduced message and the current state of the vertex (vertexProgram).

In order to compute the partial objective O_i on each compute node, a naive implementation sends the embedding of each neighbor to vertex i as sendMessage, keeps the union of embeddings as the reduceMessage, and optimizes O_i in the vertexProgram. However, in a map-reduce framework, combining the embedding vectors can result in prohibitively large collections since there is no bound on the degree of the vertices.

We use a simple trick to avoid the construction of these large collections by propagating the gradient instead of the embeddings. However, we first have to make sure that the total gradient of Equation 4.15 can be computed by the vertex programs.

The gradient of O_i can be written as

$$\nabla O_i = \sum_{j \in A(i)} \nabla O_{i \leftarrow j}, \qquad \text{(Equation 4.16)}$$

where

$$\nabla O_{i \leftarrow j} = e_w * \mathbf{u}_j * \sigma(e_w * \mathbf{u}_i^T \mathbf{u}_j)(1 - \sigma(e_w * \mathbf{u}_i^T \mathbf{u}_j). \qquad \text{(Equation 4.17)}$$

Finally we can update the embedding using gradient ascent:

$$\mathbf{u}_i = \mathbf{u}_i + \eta \nabla O_i \qquad \text{(Equation 4.18)}$$

Using edge triplets, each vertex in the augmented neighborhood $A(i)$ has access to data structures needed for computing $\nabla O_{i \leftarrow j}$. Therefore, defining $\nabla O_{i \leftarrow j}$ as a sendMessage function and sum as the mergeMessage operation, the final reduced message for vertex i is Equation 4.16. Finally, vertexProgram executes

Algorithm 1 Vertex-Centric Network Embedding

//e_{ji} : edge from j to i.
//d: embedding dimension
//msg: (m: $|\mathbb{R}|^d$)
//vertex attributes: (u: $|\mathbb{R}|^d$)
//$m_{i \to j}$: means the message from i for j
procedure sendMessasge(e_{ij}, u_i, u_j)
    $m_{j \to i}$: $\nabla O_{i \leftarrow j}$ //Eq. Equation 4.17
end procedure
procedure MergeMessages($m_{i \to j}$, $m_{k \to j}$)
    $m_{i \to j}$ + $m_{k \to j}$ //Eq. Equation 4.16
end procedure
procedure vertexProgram(u, m)
    $u \leftarrow u + \eta m$ // Eq.Equation 4.18
end procedure

the gradient update. In this vertex-centric design, the size of the data structures remains bounded and no large collection would be constructed in the intermediate steps. Therefore, we can optimize Equation 4.15 for very large graphs with large vertex degrees. Algorithm 1 shows the definition of these functions.

4.4. Experiments

We compare our network embedding algorithm, VCNE, with LINE (Tang et al., 2015), Node2vec (Grover and Leskovec, 2016) and PyTorch-BigGraph (Lerer et al., 2019) on mid-size datasets to show the capability of VCNE to learn meaningful representation. Then, we apply VCNE to very large graphs for the task of link prediction. Table 5.1. reports the characteristics of the graphs used in our experiments.

4.4.1. Vertex Classification

The goal of vertex classification is to place each vertex into different groups, which includes both multi-class and multi-label classification. In multi-class

TABLE 4.1. The number of vertices and edges of the real-world graphs in our test suite.

Name	Num. of Vertices	Num. of Edges
Friendster	68,349,466	2,586,147,869
Twitter-MPI	52,579,682	1,963,263,821
Twitter	41,637,597	1,453,833,084
LiveJournal	5,193,874	48,682,718
Reddit	232,965	11,606,919
PPI	56,944	793,632

classification, the problem is to label a vertex with one of the possible classes, while in multi-label classification, the problem is to assign a subset of possible labels to a vertex. For the multi-label setting, we can allocate one class variable for each label that can be on if the label present in the subset and off otherwise.

We use two datasets of protein-protein interaction (PPI) and Reddit posts. In PPI, the goal is to assign a set of activated protein functions to each vertex, which are represented using positional gene sets, motif gene sets, and immunological signatures (Hamilton et al., 2017). The total possible protein functions are 121 and the vertex feature set size is 50.

Reddit is an online discussion forum in which people publish posts and comment on others' posts. In the Reddit graph, the vertices are the posts and two vertices are connected with an edge if a user comments on the posts corresponding to the vertices (Hamilton et al., 2017). The node features include the average word embedding of the title, all comments of the post and the score of the post as well as the number of comments on the posts. The total number of features is 602, and the goal is to assign each vertex to one of 41 communities. For both PPI and Reddit, we used the same set of train/val/test as provided by Hamilton et al. (2017). Table 5.1. shows the characteristics of these two graphs.

We first generate vertex embedding using LINE, Node2Vec, Pytorch-BigGraph and VCNE, and then concatenate the vertex embedding to the vertex features, and use it as input to a logistic regression classifier to predict labels. As a baseline, we also train logistic regression using only the vertex features. Although more complex classifiers such as multi-layer perceptron would be possible and may result in higher accuracy, we use simple logistic regression to better isolate the impact of vertex embedding.

We used an embedding dimension of 100 for all algorithms.

TABLE 4.2. F_1 score of vertex classification tasks using different embedding algorithms.

	PPI	Reddit
Vertex features	43.3	51.2
LINE	53.08	63.9
Node2vec	49.8	65.4
PyTorch-BigGraph	52.70	66.3
VCNE	**53.28**	**66.7**

Table 4.2. shows the performance VCNE, LINE, Node2Vec, and raw vertex features in terms of F_1 score. For all embedding algorithm, using embedding in addition to vertex features helps, so we can conclude that the embedding is meaningful and encodes structural properties of the graph. For both Reddit and PPI, VCNE is more accurate than all the baselines. We also show the learned embedding by VCNE using t-SNE (Maaten and Hinton, 2008) in Figure 4.3.. VCNE can capture clear clusters in the graph.

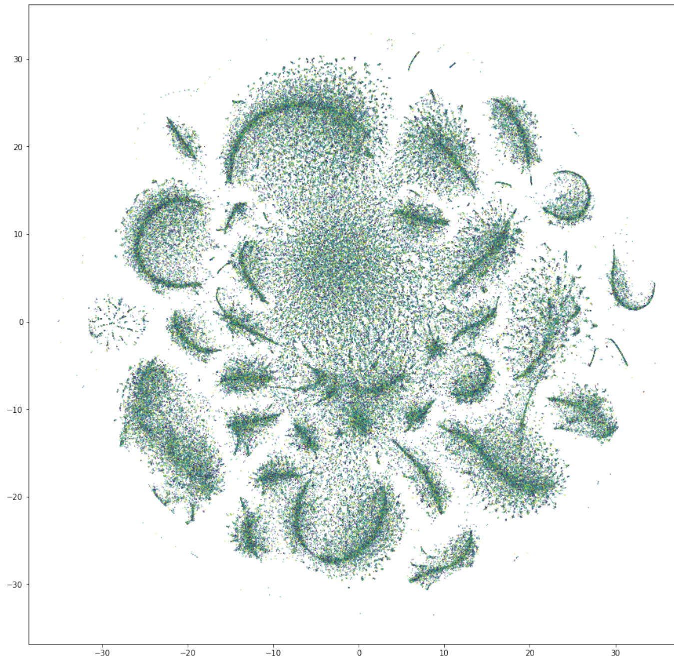

FIGURE 4.3. The embedding of the Reddit graph generated by VCNE.

4.4.2. Link Prediction

Link prediction is an import graph analytic problem, in which we wish to predict the potential edges in the network. This problem is of particular interest for social network friend suggestion or predicting the evolution of graphs in the future.

We constructed a synthetic link prediction dataset, for which we dropped one percent of the current edges of the graph and kept the dropped edges as the test set combined with another set of vertex pairs as the true negative. The size of our negative set is equal to the size of the dropped set making sure that we have balanced test set. We generate a similar train and validation set. The remaining edges of the graph constitute the core graph, which the network embedding algorithms have been trained on. We emphasize that the training algorithms have not seen the dropped edges. We first compare LINE, PyTorch-BigGraph and VCNE on the LiveJournal graph.

TABLE 4.3. Link Prediction for LiveJournal

	Precision	Recall	F_1
Jaccard	99.9	82.6	90.4
LINE	90.8	84.9	87.8
Pytorch-BigGraph	92.0	80.7	86.0
VCNE	93.3	88.1	**90.6**

TABLE 4.4. The performance of link prediction using VCNE

	Precision	Recall	F_1
Friendster	84.8	93.5	88.9
Twitter MPI	87.5	84.4	85.9
Twitter	80.7	90.0	85.1

We also use Jaccard index to predict an edge: $J(u,v) = \frac{N(u) \cap N(v)}{N(u) \cup N(v)}$, where $N(u)$ is the set of neighbors of vertex u. Computing the Jaccard index requires constructing triplets whose vertex attributes are sets of neighbor IDs, and for very large social networks, this results in prohibitively large messages given the power-law degree distribution of social networks. Nevertheless, we could compute the Jaccard index for LiveJournal graphs, but not for the other larger graphs. The cut threshold for deciding the existence of an edge is selected based on the validation data. For LiveJournal, using the Jaccard index results in 99.2% precision, 71.1% recall, and F_1 score of 83.1%. For the link prediction using embeddings, we train a 2-layer multi-layer perceptron, with 500 hundred hidden units and train it using the training pairs. We pick the best model based on the performance on the validation set, and report the model performance on the test set.

Table 4.3. the performance of link prediction for LiveJournal graph. Jaccard index has the highest precision, while VCNE has the best performance in overall F_1 score.

Next, we apply VNCE to the very large graphs and report the results in Table 4.4..

4.4.3. Scalability

To measure the scalability of VCNE over Apache Spark, we run VCNE for Friendster, Twitter MPI, Twitter, and LiveJournal with different numbers of Spark workers: 10, 20, 30, and 40. Each worker has access to 20 cores and 75GB of memory (for a total number of cores ranging between 200 and 800 and memory ranging from 750GB to 3TB). The University of Oregon Talapas cluster where we peformed the experiments consists of dual Intel Xeon E5-2690v4 nodes connected with an EDR InfiniBand network.

Figure 4.4. reports the average runtime for one learning iteration, which includes generating the random graphs, combining the random graphs with the original graph, and updating the embedding using Algorithm 1. We observe that the overhead of using data-parallel systems such as Apache Spark for processing mid-size graphs such LiveJournal is considerable, but increasing the number of workers significantly helps the processing of larger graphs such as Twitter-MPI and Friendster.

We also study the effect of the dimension of embedding and the number of negative samples on the running time of VCNE. These two factors directly affect the performance of the underlying map-reduce paradigm. As we increase the dimension of embedding the local memory required for map-reduced operation increase, thus imposing more overhead on the system. We measure the running time of 10 iteration of training VNCE for the LiveJournal graph. We used 10 workers with 20 cores and 80G of memory each. Figure 4.5. reports the

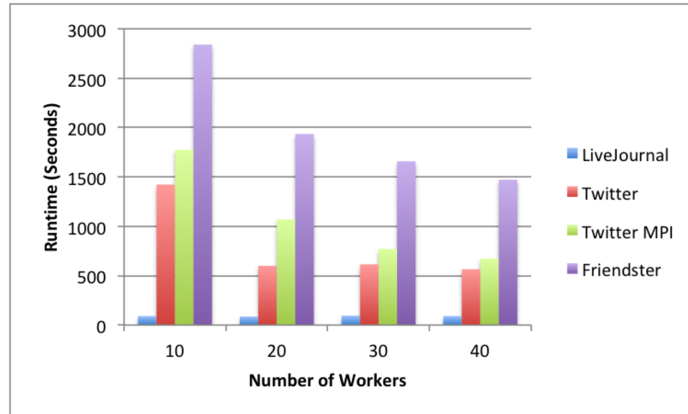

FIGURE 4.4. Average runtime for one training step of VCNE with 10 to 40 Spark workers.

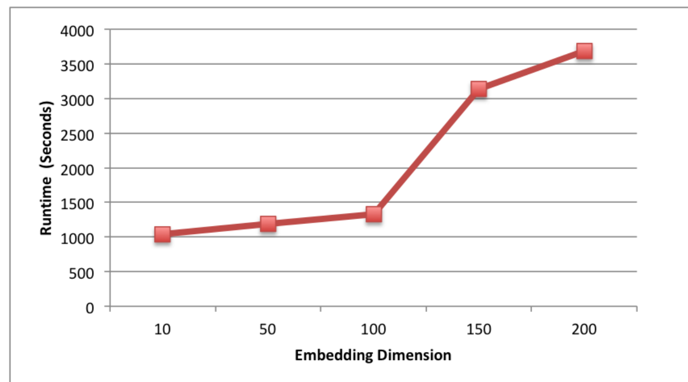

FIGURE 4.5. The effect of embedding dimension on the running time for the LiveJournal graph.

results, which shows the running time of VCNE with respect to the dimension of embedding.

We also study the effect of negative sampling by comparing the running times of VCNE on the LiveJournal graph with different numbers of negative samples. Negative samples increase the size of augmented graph, thus increasing the number of messages and increase the running time (see Figure 4.6.).

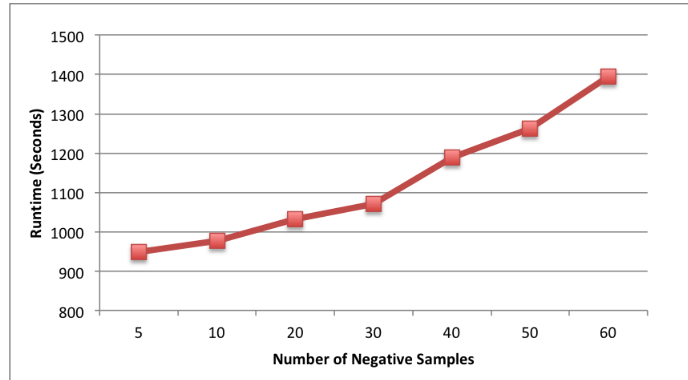

FIGURE 4.6. The effect of the number of negative samples on the running time for the LiveJournal graph.

4.4.4. Implementation Details

Working with iterative algorithm over very large graphs may result in replicating large collections such as EdgeRDDs in the memory. It is very important to unpersist the collection from memory in order to avoid exceeding the available memory capacity. For example, in the pipeline operations such graph construction followed by groupEdge, Apache Spark materializes the first graph and we lose the pointer to it as it is followed by map operation. It is necessary to observe the storage memory profile provided by Apache Spark as part of its Web UI to make sure that no large collections are left behind in an iteration.

We observe that unpersisting the RDDs may not force evacuating the memory and some RDDs may reside in the memory, waiting for the garbage collector. This behavior becomes critical for iterative algorithms: increasing the memory usage and activating out-of-core processing, when it is not necessary. Therefore, to enforce evacuating the memory, we serialize the working RDDs and close the Spark session at the end of each iteration. This trick is not necessary for mid-size graphs, however, for the consistency we apply it all of the reported experiments.

Moreover, operations such as aggregateMessage, which is used for message passing over graphs requires significant amount of data shuffling for shipping vertex attributes (embeddings) among workers. This results in a large amount of out-of-core data, which is stored in local storage accessible to the workers; this limits the size of vertex attributes given a fixed number of workers.

4.5. Conclusions

In this chapter, we introduced a new distributed-memory parallel vertex-centric algorithm for learning network embedding for very large graphs using GraphX and Apache Spark. Our algorithm, VCNE, can easily scale to handle very large graphs (billions of vertices and edges or larger) by increasing the number of Apache Spark workers that are accessible to it. We also show the VCNE can learn meaningful representations as demonstrated by the performance of classification and link prediction.

CHAPTER V

PROCESSING BIG DYNAMIC GRAPHS

The work presented in this chapter has been previously published in Riazi et al. (2018), and Riazi is the primary contributor to the paper.

Real-world graphs such as social networks, citation networks, road networks, or communication networks evolve as new edges and vertices come, and some of the existing ones are removed from the existing graphs. For an evolving or a dynamic graph, one needs to re-run static or sequential graph algorithm such as the single-source shortest path algorithm every time that the network changes. These changes are associated with a timestamp of their occurrence, so we can define static snapshots for a dynamic graph as the state of a graph at a specific time. Therefore, we can re-run the sequential graph algorithm for the latest or a specific snapshots. However, this re-running is expensive especially for very large dynamic graphs.

In this chapter, we focus on computing single-source shortest path for dynamic graphs to discuss the challenges of applying a sequential algorithm on evolving or dynamic graphs. We propose a novel distributed computing approach, SSSPIncJoint, to update SSSP on big dynamic graphs using GraphX. Our approach considerably speeds up the recomputation of the SSSP tree by reducing the number of map-reduce operations required for implementing SSSP in the gather-apply-scatter programming model used by GraphX.

5.1. Introduction

Discovering the single-source shortest path (SSSP) tree is a classical graph theory problem with many real-world applications such as finding routes in maps

and social network analysis. However, many graphs evolve over time, which necessitates the recomputation of the SSSP tree. For very large dynamic graphs, this recomputation requires significant resources and is time consuming, thus motivating the development of new algorithms that can quickly recover the updated SSSP tree without recomputing it from scratch.

This problem is exacerbated when graphs are so large that they do not fit in the memory of a single machine. In these cases, the graphs are analyzed using scalable parallel approaches that can use distributed memory and out-of-core processing. An example of such distributed memory software is GraphX Gonzalez et al. (2014), which has been developed on top of Apache Spark. GraphX enjoys the fault-tolerance and distributed computing provided by the data-parallel environment of Apache Spark. Apache Spark supports map-reduce (MR) operations over immutable distributed data structures called resilient distributed dataset (RDD) (Zaharia et al., 2012), which requires the algorithms such as SSSP to be defined as a set of (expensive) MR operations over RDD representation of graphs. However, GraphX does not come with built-in support for dynamic graphs; instead, we can apply batch updates by mapping the current snapshot of a graph to the next snapshot.

We can re-execute the SSSP algorithm on the new snapshot to obtain the new SSSP tree. However, reusing the computation from the previous snapshot may save significant execution time, especially for very large graphs. Reducing the execution time is even more critical when expensive computing services are being used for parallel processing.

In this chapter, we explore an algorithmic approach toward reusing the computation from previous snapshot in order to compute the SSSP tree for

the current snapshot. Specifically, we introduce SSSPIncJoint, a new parallel incremental SSSP algorithm, which recovers the SSSP tree over a series of graph snapshots that represent a dynamic graph. Our **key contribution** is this new algorithm, which reduces high-overhead data-parallel operations by tracking the changes among snapshots that affect the SSSP tree.

We experimentally show that SSSPIncJoint is more efficient (up to 2.2x speedup) than recomputing the SSSP for every snapshot of large dynamic graphs.

5.2. Related Work

GraphTau (Iyer et al., 2016) proposes a paradigm of pause-shift-resume, in which whenever a new batch of updates is ready, GraphTau pauses the current computation and updates the underlying graph and resume the computation with the previous state of the vertices. GraphTau cannot guarantee the correctness of the computation.

Chronos (Han et al., 2014) and ImmortalGraph (Miao et al., 2015) optimize GAS operations across different snapshots. They suppose accessing to all snapshots in advance and batch the operations for each vertex/edge over different snapshots and run batches in parallel using a locality-aware batch scheduling. In the incremental setting, when given a set of graph snapshots, it processes the first snapshot and batches the other snapshot reusing the computation of the first snapshot.

Similarly, Tegra (Iyer, 2017) operates over all snapshots, however, Tegra does not batch all snapshots together but runs every GAS round over all snapshots before continuing with the next round, so save the redundant computation.

BLADYG (Aridhi et al., 2017) is a block-centric framework, which partition the graph into blocks and assigns each block to a worker. When a new edge comes, it updates the corresponding block and the corresponding worker may communicate to other workers to propagate the update.

In Cai et al. (2012a) they use GraphInc which uses memoization and is not scalable for large networks which we use for our experiments. Among other related papers Wickramaarachchi et al. (2015) and Fan et al. (2017) do not report any experimental scalability results and their code base is not available for comparison.

There are a few implementations of sequential SSSP on dynamic networks such as Ramalingam and Reps (1996) and Narvaez et al. (2000). Bauer and Wagner (2009) propose SSSP algorithm for dynamic networks using batch updates. Vora et al. (2017) have proposed an approach that uses approximation while calculating SSSP on streaming graphs. Srinivasan et al. (2018) recently propose an approach for finding SSSP on dynamic networks, however it is based on shared-memory parallelism. Ingole and Nasre (2015) have proposed a GPU implementation of SSSP on dynamic networks.

5.3. Static SSSP on Spark

To enable computations on large graphs that do not fit in a single machine's memory, GraphX provides a vertex-centric gather-apply-scatter (GAS) distributed-memory parallel programming model (first introduced by Pregel Malewicz et al. (2010)). In a GAS model, an algorithm is developed from a vertex point of view, and in general includes three different steps: (i) gathering messages from its neighboring vertices, (ii) updating its state, and (iii) generating messages for its neighbors. GraphX iteratively executes these steps, and each iteration of these

steps is called a *superstep*. GraphX stores a graph as two RDDs, one for edges and another for vertices. It also provides *triplets* view as a joint representation of an edge attribute and the attributes on its incident vertices. As it provided by the name view, the triplets are dynamically constructed by shipping vertex attributes to the computation nodes where the corresponding edge partitions are located. This makes MR operations on triplets more expensive than MR operations on edge or vertex RDDs.

Each superstep of a GAS model can viewed as a set of MR operations over the triplet, edge and vertex RDDs. To gather the messages for each vertex, each triplet is mapped to messages using a *sendMessage* function that has access to edge and vertex attributes of its source and destination vertices, and then a *reduceMessage* function combines the messages to generate an RDD containing pairs of vertex ID and message data. To apply the messages, a new vertex RDD for the vertices that received any message is constructed by joining the existing vertex RDD and the new message RDD, and then the old vertex attributes and the message data are mapped to new vertex attributes using a *vertexProgram* function. Finally the graph's vertex RDD is updated by joining the new vertex RDD with the existing one to make sure that the vertex partitioning remains the same, otherwise, constructing the triplet view becomes very expensive for the next round.

5.4. Dynamic SSSP on Spark

Dynamic graphs can be viewed as a series of graph snapshots that evolve over time, where each snapshot is constructed by applying an update batch to its predecessor snapshot. In our setting, we assume that the update batches are

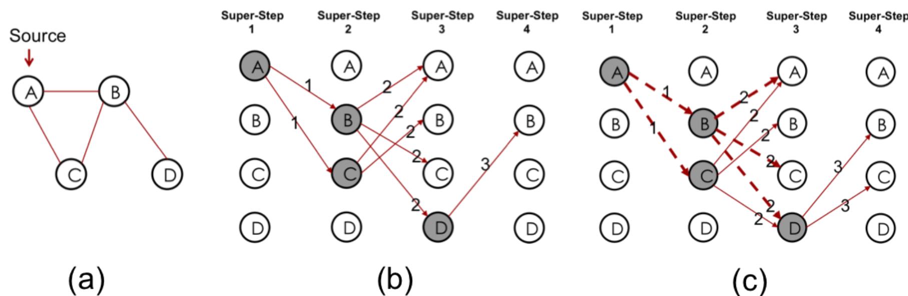

FIGURE 5.1. a) The original graph, weights not shown for readability. b) The SSSPBase algorithm based on GAS model. The gray nodes indicate the vertices that participate in a superstep. The red arrows is the messages labeled with shortest paths. c) The GraphInc execution after adding an edge between vertices C and D. The dotted edges shows the memoized messages that have been saved.

queued until the computation on the current snapshot is completed. GraphX does not have built-in support for dynamic graphs since it depends on immutable RDDs for graph representation.[1]

An updated graph can be constructed by mapping the old edge RDD to the new one to reflect the new changes (edge insertion and deletion) and constructing a new graph using the new edge RDDs. A simple approach for computing SSSP over dynamic graphs is to re-run SSSP for each snapshot separately. However, the main goal is to expedite the repetitious computation on dynamic graphs by reusing the state of vertices in the current snapshot as much as possible, so we have to transfer the old vertex attributes to the new graph using the *join* operations over RDDs.

Reusing computation for GAS is introduced by GraphInc Cai et al. (2012b), which memoizes received messages and vertex states from all supersteps. In each superstep, a vertex participates in GAS if its current state is different from the memoized state for the same superstep on the previous snapshot. A vertex runs the

[1] IndexedRDD (`https://github.com/amplab/spark-indexedrdd`) was introduced to expedite modifying a graph, however, it is not officially supported by GraphX due to fault-tolerance issues. Therefore, we focused on constructing dynamic graphs merely using the functionality provided by GraphX.

vertexProgram using the received messages and also using the memoized messages from its neighbors that have participated in the same superstep of the previous snapshot, but not in the current snapshot. Therefore, GraphInc runs SSSP for the new snapshot for the same number of supersteps, but with fewer messages in each superstep as shown in Figure 5.1.c.

A naive implementation of GraphInc on top of GraphX suffers from **two problems**: **first**, it does not reduce the number of supersteps, which are executed using expensive join operations over large RDDs, and **second**, in an MR framework such as GraphX, we have to store the memoized information as the vertex attributes and frequently ship them across different computation nodes (workers in Spark), which makes memoization impractical for large networks, especially for the social networks with power-law degree distributions. In order to scale to large social networks, the size of vertex attributes must not depend on the degree of vertices, which motivates using fixed-size attributes such as tuples. An example of variable-size attributes would be if each vertex keeps the distance to source of all its neighbors. We only store the distance to source, the parent of each vertex, and an extra flag for capturing the affected vertices due to a batch update.

In contrast to GraphInc, our memoized state does not provide enough information to recompute the state of vertices, thus requiring message propagation to take place. However, we can limit the number of required messages by considering the details of the SSSP algorithm.

An update batch includes a set of edge insertions and deletions.[2] Inserting or deleting edges directly affects the target vertices of the edges (immediate affected) or indirectly affects the descendants of the target vertices (causal affected). We call

[2]For simplicity, we only discuss edge insertion and deletion, but the same reasoning applies for weight decrease and increase.

a vertex *insert-affected* or *delete-affected* if it is affected (immediate or causal) by edge insertion or deletion, respectively.

If the update batch only includes edge insertions, the states of affected vertices (both immediate or causal) converge to the correct states if we continue running the SSSP algorithm. This happens because edge insertion can only shorten the distance of a vertex to the source, and a vertex generates a message for its neighbor only if it can reduce the distance-to-source (DTS) of the target vertex, otherwise the vertex does not participate in the superstep. Therefore, the neighbors of insert-affected vertices will participate in the message passing in order to adjust the state of the insert-affected vertices. This update propagates to adjust the DTS of all insert-affected vertices.

The situation for edge deletion is more complicated because deleting an edge may increase the DTS of affected vertices, and in turn, the neighbors of delete-affected vertices may not participate in message passing because the DTS of delete-affected vertices is at least as large as their DTS before the edge deletion happening.

If an update batch contains any edge deletion, the SSSP algorithm may not correct the delete-affected vertices, so we have to mark or invalidate them to make sure that we can correct their states using the SSSP algorithm. This marking phase (invalidation) starts with the immediate delete-affected vertices and propagates to their descendants in the SSSP tree using the GAS model. Therefore, in each superstep of the invalidation phase, each marked vertex generates messages for its children in the SSSP tree. If a vertex receives a message, it changes its status to marked. After convergence, all the delete-affected vertices are marked.

By setting the state of the marked vertices to ∞, we can make sure the SSSP does converge to the correct values. Therefore, after the invalidation phase, we can rely on the SSSP algorithm as a correction phase to adjust the state of all affected vertices (insert-affected and delete-affected).

These two steps (invalidation and correction) comprise our vanilla SSSPInc Algorithm 2, which exactly computes the SSSP tree for dynamic graphs. However, the invalidation phase is also expensive since it requires join operations over large RDDs to propagate the marks to all delete-affected vertices, and experimentally we observe that the invalidation phase may take as long as the correction phase (that considers all delete-affected and insert-affected vertices). Therefore, we introduce two variations of the basic SSSPInc: SSSPIncApprox and SSSPIncJoint in order to reduce the required time for recomputing SSSP for large dynamic graphs.

Algorithm 2 High-level SSSPInc

```
Run SSSP on the primary graph.
for each update batch do
    Invalidate all delete-affected vertices.
    Apply the update batch.
    Adjust the state of vertices.
end for
```

Algorithm 3 High-level SSSPIncApprox

```
Run SSSP on the primary graph.
for each update batch do
    Invalidate the immediate delete-affected vertices.
    Apply the update batch.
    Adjust the state of vertices.
end for
```

5.5. SSSPIncJoint

The invalidation propagation phase of SSSPInc is extremely expensive because it requires multiple MR and join operations over very large RDDs in order to pass few messages (with respect to the size of the graph). The number of required supersteps for the invalidation phase depends on the position of immediate delete-affected vertices in the SSSP tree. Therefore, SSSPInc may have even more supersteps than running the SSSP algorithm from scratch on the current snapshot.

To expedite the message propagation for the invalidation phase, one can prune the graph to only the SSSP tree, which significantly reduces the size of the edge RDD. But we should also note that pruning requires an MR operation over triplets. The overhead cost of pruning may be amortized over message propagation steps, but in our setting we didn't find it useful.

An alternative approach is to ignore the incorrect state of causal delete-affected vertices. In that case, the DTS of the causal delete-affected vertices is only an approximation of the true value. We call this approach SSSPIncApprox, and is described in Algorithm 3.

To achieve the same efficiency as SSSPIncApprox, but with more accurate DTS values, we try to run the invalidation and correction phase jointly. Although the joint execution may result in inexact values, we can guarantee that if it converges, it would be to the exact values. We revisit the convergence assumption after describing the algorithm.

To jointly execute the invalidation and correction, we must make sure that the correction does not truncate the invalidation phase. We first mark all the immediate delete-affected vertices. In each superstep, a marked vertex sends marking messages to its children in the current SSSP tree. To make sure that a

delete-affected vertex at least remains marked for one superstep, the neighbors of a marked vertex do not send any DTS value for the marked vertex. Therefore, if a vertex is marked it can propagate the mark to its children in one superstep. After propagating the mark, the vertex clears itself and sets its DTS value to ∞, then removes its parent in the tree. This happens in the vertexProgram. Therefore, in the next supersteps, its neighbors start sending their DTS to the already cleared vertex. To avoid loops, a vertex never sends DTS to its current parent in the SSSP tree, however, longer cycles are still possible but less likely. Algorithm 3 shows the GAS model for SSSPIncJoint.

Proposition: SSSPIncJoint converges to the exact single-source shortest path value or never converges.

Proof. Suppose that the edge e_{uv} is removed and also suppose that there exists an edge e_{yv} such that y belongs to the subtree rooted at v. Based on these assumptions, there exists a cycle including v and y. Let z be any vertex in this cycle, including u and v, with an edge e_{xz} such that x belongs to the subtree rooted at the source of the original SSSP tree. Note that if the latter condition is not met, the graph is not strongly connected after removing edge e_{uv}. In SSSPIncJoint, v is marked, and y sends $y.distance + e_{yv}.weight$ to v and becomes the parent of v. However, y is also a descendant of v, so it will receive the mark token and a new DTS value from its parent based on the DTS of vertex v, and since node y is the parent of v, it passes the mark token to v and the new DTS value. This cycle monotonically increases the DTS values of vertices in the cycle. Therefore, eventually $x.distance + e_{xz}.weight < z.distance$, so z changes its parent to x and breaks the cycle. And after another round of message passing in the cycle, all DTS values become exact. If there is no such vertex x (i.e. the graph is not

strongly connected after removing the edges), then DTS values of vertices in the cycle increase infinitely, and the algorithm never converges. □

5.6. Experiments

We evaluate the performance of SSSPInc, SSSPIncApprox, and SSSPIncJoint on three very large real-world social network graphs: Friendster, Twitter-MPI, and Twitter[3]. We also run our experiments on a very large syntactic random graph generated by R-MAT: with parameters: a=0.55, b=0.15, c=0.15, d=0.15. Table 5.1. shows the characteristics of these datasets.

We assume that a primary graph and an update batch in the form of edge events (insert or delete) are given as input. To construct a primary graph and update batch from a static graph, we randomly select an α fraction of edges of the static graph without replacement. β percent of events are edge deletion, and the rest are edge insertion. A primary graph is formed by removing the edges corresponding to insertion events from the static graph. The number of edge insertions and deletions for each update batch is shown in Table 5.2..

Inserting an edge may introduce a new vertex if the source or destination vertices are not in the graph. Therefore, we remove standalone vertices appearing as a result of edge removal from the static graph; they will be added to the graph as new vertices when we add the edges back.

The baseline is to re-run the SSSP algorithm for each snapshot without considering the dynamic nature of the graph. We call this method SSSPBase.

We use the vertex with the highest degree as the source for the SSSP algorithm. All algorithms are implemented using the GraphX library of Apache

[3]These graphs are the three largest graphs available on the Konnect graph repository: `http://konect.uni-koblenz.de/networks/`

Algorithm 4 SSSPIncJoint

```
//s: Source vertex for the SSSP algorithm
//e_uv : edge from u to v.
//msg: (source, distance, mark)
//vertex attributes: (isMarked, distance, parent)
//u → v : msg means u generates msg for v
procedure sendMessasge(e_uv)
    if u.isMarked then
        if v.isMarked then
            No message
        else if v.parent = u then //v is a child of u
            u → v: (u, ∞, true)
        else
            No message
        end if
    else if v.isMarked then
        if u.parent ≠ v  then //v is not the parent of u
            u → v: (u, e_uv.weight + u.distance, false)
        else
            No message
        end if
    else if e_uv.weight + u.distance < v.distance then
        u → v: (u, e_uv.weight + u.distance, false)
    else
        No message
    end if
end procedure
procedure MergeMessages(a, b)
    mark ← a.mark or b.mark
    if a.distance < b.distance then
        (a.source, a.distance, mark)
    else
        (b.source, b.distance, mark)
    end if
end procedure
procedure vertexProgram(u, msg)
    if  u = s  then
        (false, 0.0, s)
    else
        if msg.mark then
            (true, ∞, ∞)
        else if  u.distance > msg.distance then
            (false, msg.distance, msg.source)
        else
            (false, u.distance, u.source)
        end if
    end if
end procedure
```

TABLE 5.1. Vertices and edges of the real-world and synthetic graphs in our test suite.

Name	Num. of Vertices	Num. of Edges	Type
RMAT	339,201,984	4,252,445,904	Directed
Friendster	68,349,466	2,586,147,869	Directed
Twitter-MPI	52,579,682	1,963,263,821	Directed
Twitter	41,637,597	1,453,833,084	Directed

TABLE 5.2. The characteristics of update batches for different graphs.

	$\alpha = 0.1\%, \beta = 1\%$		$\alpha = 0.1\%, \beta = 10\%$		$\alpha = 1\%, \beta = 1\%$	
	Insert	Delete	Insert	Delete	Insert	Delete
RMAT	4,250,418	42,647	3,867,390	429,833	42,521,392	429,474
Friendster	2,559,344	25,949	2,327,102	258,373	25,592,403	258,992
Twitter-MPI	1,941,750	19,856	1,764,937	196,681	19,444,189	196,481
Twitter	1,453,304	14,796	1,321,018	146,888	14,532,098	147,270

Spark v. 2.3. For GraphX, we use ten Spark workers on a cluster with ten dual Intel Xeon E5-2690v4 processors. Each worker has access to 20 cores (for a total of 200 cores) and 120GB of memory (total 1.2TB memory).

We do not use GraphInc in our comparison because by using the suggested memoization, we have to store all messages in attributes of vertices. This would dramatically increase the size of vertex attributes, making shipping the vertices to the computation nodes very costly. Moreover, the number of messages depends on the degree of vertices, thus for social network graphs with power-law degree distributions, some of vertices have to store prohibitively large number of messages.

5.6.1. Results and Discussion

We report the execution time of SSSPBase, SSSPInc, SSSPIncApprox, and SSSPIncJoint for our three different update batches in Figure 5.2.. The execution time depends on the number of supersteps, as well as the size of the

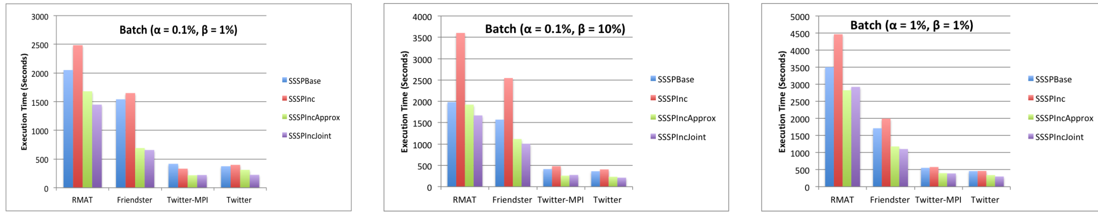

FIGURE 5.2. The execution time (in seconds) of SSSPBase, SSSPInc, SSSPIncApprox, and SSSPIncJoint for different update batches.

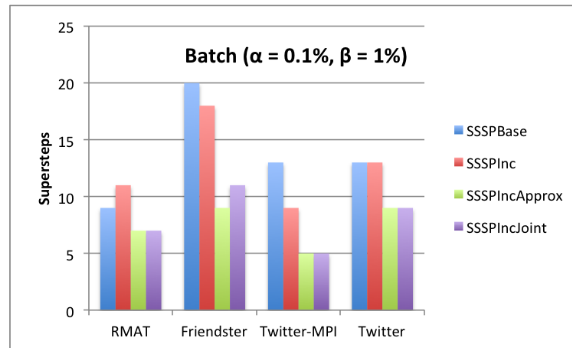

FIGURE 5.3. Total number of GAS supersteps for running each algorithm.

graph (number of edges and vertices), which determines execution time of each superstep. Figure 5.3. shows the number of supersteps of different algorithms for batch $\alpha = 0.1\%, \beta = 1\%$. Comparing to the same execution time for the same batch in Figure 5.2., we conclude that the ranking of algorithms with respect to the number supersteps is often the same as their ranking with respect to the execution time. The differences are explainable by the execution time of each superstep, which also depends on the number of active vertices participating in the message passing.

In general, SSSPInc is often slower than SSSPBase, and the difference is significant when we increase the number of edge deletions as in the batch $\alpha = 0.1\%, \beta = 10\%$. This happens because the invalidation phase is expensive since it needs to run several supersteps. We also show the number of supersteps required for the invalidation and correction phases, as well as the execution time for each

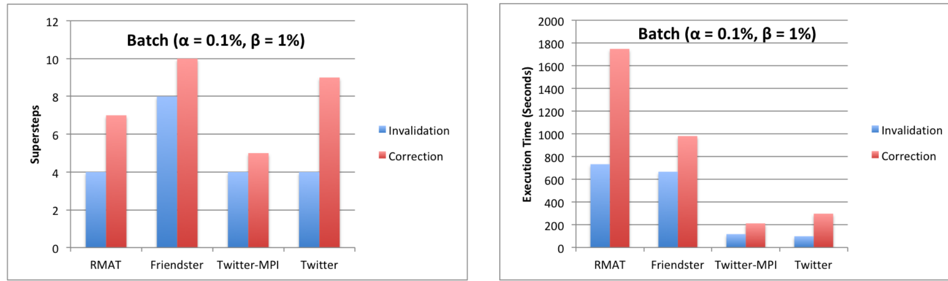

FIGURE 5.4. Total number GAS supersteps (left) and execution time (right) for invalidation and correction phase in SSSPInc.

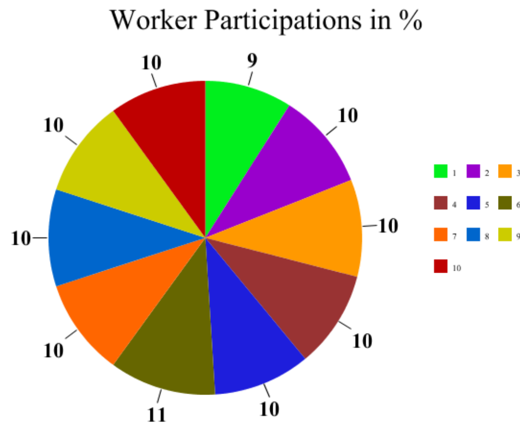

FIGURE 5.5. Apache Spark workers participation in SSSPInc for update batch $\alpha = 0.1\%, \beta = 0.1\%$ for Friendster graph.

phase in Figure 5.4.. The reported numbers are for batch $\alpha = 0.1\% \beta = 1\%$. The execution time of invalidation phase is considerable comparing to the execution time of the correction phase.

SSSPIncApprox, which only has one step of invalidation (for immediate delete-affected vertices), is always better than SSSPInc by saving multiple supersteps of invalidation phase. SSPIncApprox is also always better than SSSPBase. The one-step invalidation of SSSPIncApprox has not been included in the number of supersteps required for SSSPIncApprox. We notice, from the execution of SSSPInc, that the number of invalidated vertices is negligible compared to the number of the vertices in the graph (less than 0.001% of vertices),

which indicates that the accuracy of shortest-path distance values found by SSSPIncApprox is above 99.9% comparing to the exact SSSP.

SSSPIncJoint is always better or equivalent to SSSPIncApprox and is always better than SSSPBase and SSSPInc. SSSPIncJoint and SSSPIncApprox often share the same number of supersteps, which suggests that SSSPIncJoint successfully combines the correction and invalidation phases.

As we mentioned earlier, SSSPIncJoint may not converge if deleting the edges partitions the graph into disconnected components, but SSSPIncJoint in all of the experiments converges and finds the exact DTS for all the vertices comparing to our SSSPBase.

Finally, to see how balanced the workload distribution over the workers is, we show the processing time of each worker for batch $\alpha = 0.1\%, \beta = 1\%$ applied to the Friendster graph in Figure 5.5.. We find that the workload is evenly distributed among the workers.

5.7. Conclusion

We introduce an algorithmic approach to compute the SSSP tree for dynamic graphs on GraphX. Our approach, SSSPIncJoint,[4] jointly finds the vertices with incorrect state and corrects their states. SSSPIncJoint is computationally more efficient than computing the SSSP from scratch and also more efficient than two-phase approaches that complete finding the vertices with incorrect states before start correcting their values.

[4]Source code and instructions to reproduce our scalability results are available on https://github.com/DynamicSSSP/SSSPIncJoint-

CHAPTER VI

CONCLUSION AND FUTURE DIRECTIONS

In this dissertation, we introduce GraphFlow as a framework for processing very large graphs. GraphFlow encapsulates the detail of working with data-parallel systems and introduces high-level components to process big graphs. These components follow the same map-reduce paradigm, but they map a graph to another graph or dataframe or reduce it to scalar values.

In order to expand the functionality of GraphFlow beyond traditional graph algorithms, we introduce vertex-centric network embedding (VCNE) for learning graph representation for very large graphs since existing algorithms do not scale well.

In addition to static graphs, GraphFlow supports processing very large dynamic graphs with batch updates. We developed a novel algorithm SSSPIncJoint, which efficiently computes single-source shortest paths (SSSP) for different snapshots of a graph (determined by the update batches).

GraphFlow has many potentials to facilitate social sciences, especially for researches that they do not want to involve in complicated development of low-level pipelines. Moreover, GraphFlow can be used for educational purposes, where the goal is to mine graph data without requiring in-depth knowledge of big data processing systems. Similar frameworks such as Weka[1] has been widely used for machine learning algorithms. GraphFlow can also provide components that store intermediate graph data, such as learned embedding, which can be used in the

[1] https://sourceforge.net/projects/weka/

pipelines in order to reduce required computation for repetitive experiments. This reduction is significant for very large graphs.

6.1. Potential future directions

In this section, we discuss the potential future direction to extend this dissertation. We can group these directions into framework extension and algorithmic extension. The former regards the potential extension of the GraphFlow architecture, while the latter target its functionality.

6.1.1. Workflow Expansion

Although GraphFlow has been developed over the Galaxy workflow system, GraphFlow may benefit from ad-hoc workflow system that is aware of the underlying data-parallel system, here Apache Spark. For example, two consecutive components in the workflow may share a SparkContext, which allows the system to keep the objects in memory. This reduces the overhead of serialization and de-serialization of objects between two consecutive components. Apache Spark gains a similar advantage over the Hadoop map-reduce by adding in-memory computation.

The other limitation of Galaxy is the lack of support for streaming data. Galaxy runs each workflow component separately after executing its dependence. However, for streaming data, all components should be executed in parallel. This requires significant modification in the engine of the Galaxy workflow system.

6.1.2. Application of graph embedding for dynamic graph components

Graph embedding shows promising results on predicting the incoming edges of an evolving graph. In advance knowledge of potential incoming edges can be

used for pre-computation of target algorithms such as SSSP on the predicted future graph. Therefore, we can reuse such computation when the actual graph arrives. The predicted graph can be treated as the base snapshot for the actual new graph. Therefore, we can reuse the computation similar to what discussed in Chapter V. However, for these types of applications, we require to have embeddings with high precision and low recall since for high recall and low precision results in predicting more edges that may not appear in the actual arriving snapshots. Therefore, we have to delete edges from the predicted base snapshot to get to the actual arriving snapshots, and handling deleted edges in an incremental setting is more difficult in general. Such an embedding can be achieved by increasing the negative sampling of the proposed VCNE algorithm.

We may also need to re-train the embedding after visiting a new snapshot. Recently, finding network embedding for dynamic graphs has gained more attention from the community Sankar et al. (2018); Goyal et al. (2018); however, the current algorithms are not scalable. Therefore, learning scalable network embeddings for big dynamic graphs is also a potential extension to this dissertation.

6.1.3. Multi-resolution SSSP for large dynamic graphs

In Chapter V, we discuss computing SSSP for dynamic graphs with fixed batch updates. However, for mining purposes, one may be interested in modifying the batching window to study the behavior of the dynamic graph. For example, to see how the distribution of shortest paths is modified yearly, monthly, or for an ad-hoc interval. Similarly, we can share the computation among different windowing to save computation. For example, if we have access to the shortest paths of daily snapshots, we can construct the shortest paths of monthly snapshots without

running SSSP algorithm by merely taking the computation of the last day of the month. However, computing shortest paths for daily snapshots would be costly, so we can capture some key snapshots in the evolution of the graph and compute the shortest paths for those key snapshots. For example, as discussed in Chapter V, handling edge insertion for computing shortest paths is cheaper than handling edge deletion, so we can define a key snapshot as the one that has the most rate of edge deletion with respect to the previous snapshot. We can define the rate as the number of deleted edges over the difference in the current timestamp versus the timestamp of the previous snapshot. Therefore, we can expect that any batch after a key snapshot and before the next one has mostly edge addition. Consequently, we can simply compute the shortest paths for each snapshot by updating the shortest paths of a key snapshot.

REFERENCES CITED

Agrawal, P., Garg, V. K., and Narayanam, R. (2013). Link label prediction in signed social networks. In *IJCAI*.

Aridhi, S., Montresor, A., and Velegrakis, Y. (2017). BLADYG: A graph processing framework for large dynamic graphs. *Big Data Research*, 9:9–17. arXiv: 1701.00546.

Backstrom, L. and Leskovec, J. (2011). Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644. ACM.

Baldi, P. and Pollastri, G. (2003). The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *Journal of Machine Learning Research*, 4(Sep):575–602.

Bauer, R. and Wagner, D. (2009). Batch dynamic single-source shortest-path algorithms: An experimental study. In Vahrenhold, J., editor, *Experimental Algorithms*, pages 51–62, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S., Smola, A. J., and Kriegel, H.-P. (2005). Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl 1):i47–i56.

Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. (2008). On modularity clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 20(2):172–188.

Bu, Y., Borkar, V., Jia, J., Carey, M. J., and Condie, T. (2014). Pregelix: Big(ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment*, 8(2):161–172.

Cai, Z., Logothetis, D., and Siganos, G. (2012a). Facilitating real-time graph mining. pages 1–8. ACM Press.

Cai, Z., Logothetis, D., and Siganos, G. (2012b). Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*, pages 1–8. ACM.

Chakrabarti, D., Zhan, Y., and Faloutsos, C. (2004). R-MAT: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Duran, A. G. and Niepert, M. (2017). Learning graph representations with embedding propagation. In *Advances in neural information processing systems*, pages 5119–5130.

Dyer, C. (2014). Notes on noise contrastive estimation and negative sampling. *arXiv preprint arXiv:1410.8251*.

Ediger, D., Jiang, K., Riedy, E. J., and Bader, D. A. (2013). Graphct: Multithreaded algorithms for massive graph analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 24(11):2220–2229.

Fan, W., Xu, J., Wu, Y., Yu, W., Jiang, J., Zheng, Z., Zhang, B., Cao, Y., and Tian, C. (2017). Parallelizing Sequential Graph Computations. pages 495–510. ACM Press.

Felzenszwalb, P. F. and Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. *International journal of computer vision*, 59(2):167–181.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Hollywood, CA, USA.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI 14, pages 599–613, Broomfield, CO, USA.

Goyal, P., Chhetri, S. R., Mehrabi, N., Ferrara, E., and Canedo, A. (2018). Dynamicgem: A library for dynamic graph embedding methods. *arXiv preprint arXiv:1811.10734*.

Grover, A. and Leskovec, J. (2016). node2vec: Scalable feature learning for networks. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034.

Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. (2014). Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM.

Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. (2013). Turbograph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85, Chicago, IL, USA.

Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*, pages 197–208. Springer.

Ingole, A. and Nasre, R. (2015). Dynamic shortest paths using javascript on gpus.

Iyer, A. P. (2017). Time-evolving graph processing on commodity clusters.

Iyer, A. P., Li, L. E., Das, T., and Stoica, I. (2016). Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems - GRADES '16*, pages 1–6, Redwood Shores, California. ACM Press.

Kang, U., Tsourakakis, C. E., and Faloutsos, C. (2009). PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the 9th IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Miami, FL, USA.

Lages, J., Patt, A., and Shepelyansky, D. L. (2015). Wikipedia ranking of world universities. *arXiv:1511.09021 [cs.SI]*.

Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. (2019). PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of the 2nd SysML Conference*, Palo Alto, CA, USA.

Levy, O. and Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185.

Liben-Nowell, D. and Kleinberg, J. (2007). The link-prediction problem for social networks. *journal of the Association for Information Science and Technology*, 58(7):1019–1031.

Lin, F. and Cohen, W. W. (2010). Power iteration clustering. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 655–662.

Lin, Y., Liu, Z., Sun, M., Liu, Y., and Zhu, X. (2015). Learning entity and relation embeddings for knowledge graph completion. In *Twenty-ninth AAAI conference on artificial intelligence*.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. (2012). Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727.

Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146.

Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., and Chen, W. (2015). Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):14.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013c). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer.

Narvaez, P., Kai-Yeung Siu, and Hong-Yi Tzeng (2000). New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746.

Pan, S., Wu, J., Zhu, X., Zhang, C., and Wang, Y. (2016). Tri-party deep network representation. In *IJCAI*.

Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652.

Pireddu, L., Leo, S., Soranzo, N., and Zanetti, G. (2014). A Hadoop-Galaxy adapter for user-friendly and scalable data-intensive bioinformatics in Galaxy. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 184–191. ACM.

Plimpton, S. J. and Devine, K. D. (2011). Map-Reduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632.

Ralaivola, L., Swamidass, S. J., Saigo, H., and Baldi, P. (2005). Graph kernels for chemical informatics. *Neural networks*, 18(8):1093–1110.

Ramalingam, G. and Reps, T. (1996). On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1-2):233–277.

Riazi, S. and Norris, B. (2016). Graphflow: Workflow-based big graph processing. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 3336–3343. IEEE.

Riazi, S. and Norris, B. (2019). Large-scale vertex-centric network embedding via apache spark. In *19th Industrial Conference on Data Mining*.

Riazi, S., Srinivasan, S., Das, S. K., Bhowmick, S., and Norris, B. (2018). Single-source shortest path tree for big dynamic graphs. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4054–4062.

Sankar, A., Wu, Y., Gou, L., Zhang, W., and Yang, H. (2018). Dynamic graph representation learning via self-attention networks. *arXiv preprint arXiv:1812.09430*.

Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905.

Spielmat, D. A. and Teng, S.-H. (1996). Spectral partitioning works: Planar graphs and finite element meshes. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 96–105. IEEE.

Srinivasan, S., Riazi, S., Norris, B., Das, S., and Bhowmick, S. (2018). A shared-memory algorithm for updating single-source shortest paths in large weighted dynamic networks. In *Proceedings of the 25th IEEE International Conference on. High Performance Computing, Data, and Analytics (HIPC)*.

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. ACM.

Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204.

Times Higher Education (2016). World university rankings. `https://www.timeshighereducation.com/world-university-rankings/2017`.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. (2018). Graph attention networks.

Vora, K., Gupta, R., and Xu, G. (2017). KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. pages 237–251. ACM Press.

Wale, N., Watson, I. A., and Karypis, G. (2008). Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14(3):347–375.

Wickramaarachchi, C., Chelmis, C., and Prasanna, V. K. (2015). Empowering fast incremental computation over large scale dynamic graphs. pages 1166–1171. IEEE.

Yang, C., Liu, Z., Zhao, D., Sun, M., and Chang, E. Y. (2015). Network representation learning with rich text information. In *IJCAI*, pages 2111–2117.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association.