THE UNIVERSITY OF CHICAGO

SYSTEMATIC UNDERSTANDING OF GRAPH COMPUTATION BEHAVIOR TO ENABLE ROBUST BENCHMARKING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S DEGREE

DEPARTMENT OF COMPUTER SCIENCE

BY
FAN YANG

CHICAGO, ILLINOIS
MARCH, 2015
# TABLE OF CONTENTS

LIST OF FIGURES ................................................. v
LIST OF TABLES .................................................. vi
ABSTRACT ............................................................ vii

1 INTRODUCTION ................................................... 1
  1.1 Motivation .................................................... 1
  1.2 Our Work .................................................... 3
  1.3 Contributions .............................................. 5

2 BACKGROUND .................................................... 7
  2.1 Graph and Graph Processing ................................ 7
  2.2 Computation Models ......................................... 8
  2.3 Graph-Processing Systems ................................. 9
  2.3.1 Pregel and Giraph ....................................... 9
  2.3.2 GraphLab ................................................ 10

3 EXPERIMENTAL METHODOLOGY ............................... 12
  3.1 Experiment Setup .......................................... 12
  3.2 Workload ................................................... 13
    3.2.1 Graph Algorithms ..................................... 13
    3.2.2 Graph Datasets ........................................ 15
    3.2.3 Graph Generators ...................................... 16
  3.3 Measurement ................................................ 18
    3.3.1 Fundamental Behaviors ............................... 19
    3.3.2 Performance Metrics .................................. 19

4 VARIATION IN GRAPH COMPUTATION BEHAVIOR ............... 21
  4.1 Variation across Graphs ................................... 21
    4.1.1 Graph Analytics ....................................... 21
    4.1.2 Clustering (K-Means) ................................... 22
    4.1.3 Collaborative Filtering ............................... 26
    4.1.4 Jacobi, LBP, and DD ................................... 27
  4.2 Variation across Algorithms ............................. 30

5 EXPLORING BEHAVIOR SPACE ................................. 32
  5.1 Behavior Space and Ensemble Metrics ..................... 32
  5.2 Efficient Exploration of Behavior Space ............... 35
    5.2.1 Single-Algorithm Ensembles ......................... 35
    5.2.2 Single-Graph Ensembles ......................... 37
    5.2.3 Unrestricted Ensembles ....................... 39
5.3 Implications for Robust Benchmarking ............................................. 40
  5.3.1 Understanding Diversity ....................................................... 41
  5.3.2 Reducing Benchmarking Complexity ........................................ 43

6 DISCUSSIONS AND RELATED WORK ................................................. 46
  6.1 Benchmarking Efforts .............................................................. 46
  6.2 Comparative Performance Studies ............................................ 47

7 SUMMARY AND FUTURE WORK ..................................................... 49

ACKNOWLEDGMENTS ................................................................. 51
LIST OF FIGURES

2.1 Bulk Synchronous Parallel (BSP) model. .............................................. 9
2.2 Gather-Apply-Scatter (GAS) model. .................................................. 10

4.1 GA active fraction for all graphs. In each figure, there are four curves of the same color, representing 4 experiments over graphs with same \( \alpha \) but different \( n_{edges} \). ........................................... 22
4.2 CC metric values across iterations. .................................................. 23
4.3 KC metric values across iterations. .................................................. 23
4.4 TC metric values across iterations. .................................................. 24
4.5 SSSP metric values across iterations. .............................................. 24
4.6 PR metric values across iterations. .................................................. 25
4.7 AD metric values across iterations. .................................................. 25
4.8 KM active fraction for all graphs. There are four curves of the same color, representing 4 experiments over graphs with same \( \alpha \) but different \( n_{edges} \). ........................................... 26
4.9 KM metric values across iterations. .................................................. 26
4.10 CF active fraction for all graphs. In each figure, there are four curves of the same color, representing 4 experiments over graphs with same \( \alpha \) but different \( n_{edges} \). Note that in NMF, SGD, and SVD, curves are on top of each other. ......... 27
4.11 ALS metric values across iterations. .................................................. 28
4.12 NMF metric values across iterations. .............................................. 28
4.13 SGD metric values across iterations. .............................................. 29
4.14 SVD metric values across iterations. .............................................. 29
4.15 Active fraction for Jacobi, LBP, and DD. ......................................... 30
4.16 Metric values for Jacobi, LBP, and DD. ......................................... 30
4.17 Metric Values for All Algorithms. .................................................. 31

5.1 Ensembles members in behavior space with low and high spread/coverage. ... 34
5.2 Spread for Single-Algorithm Ensembles. ........................................... 36
5.3 Coverage for Single-Algorithm Ensembles. ....................................... 37
5.4 Spread for Single-Graph Ensembles. ............................................... 38
5.5 Coverage for Single-Graph Ensembles. ........................................... 38
5.6 Spread for Unrestricted Ensembles. .................................................. 40
5.7 Coverage for Unrestricted Ensembles. ............................................. 40
5.8 Frequency of Appearance of Each Algorithm in Top100 Sets for Spread. .... 43
5.9 Frequency of Appearance of Each Algorithm in Top100 Sets for Coverage. ... 43
5.10 Spread for Limited Algorithms, Graphs, and Runtime. ....................... 44
5.11 Coverage for Limited Algorithms, Graphs, and Runtime. .................... 45
LIST OF TABLES

1.1 Comparative Graph Processing System Evaluations .......................... 2
3.1 Graph Feature Variables ......................................................... 18
5.1 Ensemble Members Achieving Best Spread and Coverage .................. 41
ABSTRACT

Graph processing is widely recognized as important for a growing range of applications, including social network analysis, machine learning, data mining, and web search. Recently, many performance assessments and comparative studies of graph computation have been published, all of which employ highly varied ensembles of algorithms and graphs. To explore the robustness of these studies, we characterize how behavior varies across a variety of graph algorithms (graph analytics, clustering, collaborative filtering, etc.) on a diverse collection of graphs (size and degree distribution). Our results show that graph computation behaviors, with up to 1000-fold variation, form a very broad space, and inefficient exploration of this space may lead to a narrow understanding of graph processing performance, or worse misleading conclusions.

Hence, we consider how to construct a high-quality benchmark set, which employs as few experiments as possible, and exhibit a wide range of graph computation behavior. We study different ensembles of graph-algorithm pairs, and define two metrics, spread and coverage, to quantify how efficiently and completely an ensemble explores the space. Our results show that: (1) an ensemble limited to a single algorithms or a single graph may unfairly characterize a graph-processing system, (2) an ensemble exploring both algorithm diversity and graph diversity improves the quality significantly (30% better coverage and 200% better spread), but must be carefully chosen, (3) some specific algorithms are more useful than others for exploring the space, and (4) it is possible to reduce benchmarking complexity (i.e. number of algorithms, graphs, etc.) while conserving the benchmarking quality.
CHAPTER 1

INTRODUCTION

1.1 Motivation

Rapid growth of World Wide Web and social networking has given rise to massive graph data sets that are used for modeling in domains such as web graphs, social networks, recommender systems, protein interaction networks, and more. These real-world graphs have remarkable size [27, 30] and various structures, which combined with complex algorithms create a critical need for efficient and scalable graph-processing systems.

To meet this need, the research community has created numerous graph-processing systems including Pregel [21], Giraph [1], GraphLab [7], SNAP [27], TurboGraph [12], Mizan [17], GPS [24], and GraphChi [19] to meet the challenges of graph processing: (1) extreme scale (100s of billions of edges), (2) irregular computation structure (difficult to aggregate multiple vertex operations), (3) poor locality, and (4) wide variation in parallelism.

Confounding factors are wide variation in both the properties of input graphs and graph algorithms. Graph features, including graph size, vertex degree, and vertex/edge weights, depend heavily on application domain. For example, the largest graph available from the Web Data Commons [30] has 3.5 billion vertices (web pages) and 128 billion edges (hyperlinks). Another web graph, Wikipedia vote network [27], contains only 7,115 vertices and 103,689 edges. Vertex degree also varies widely by domain. In social networks, popular Twitter or Facebook accounts have millions of followers, while others have only a few. But in a graph derived from a linear solver, vertices have a low, nearly uniform degree.

The dynamic properties of graph algorithms (variation in compute intensity, communication intensity, and vertex activity distribution) also exhibit great diversity. For example, computation at a vertex can vary widely based on algorithm. Communication intensity typically depends on vertex degree and algorithm. The active fraction of vertices is a strong function
of algorithm in PageRank, all vertices begin active and the fraction gradually decreases, whereas in Single-Source Shortest Path (SSSP), only the source vertex begins active, but the active fraction grows rapidly.

Numerous comparative studies of graph processing system employ a wide variety of algorithms and input graphs (see Table 1.1).

<table>
<thead>
<tr>
<th>Authors</th>
<th>Underlying Systems</th>
<th>Benchmarks</th>
<th>Input Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Giraph, GraphLab</td>
<td>ssition</td>
<td>com.Orkut</td>
</tr>
<tr>
<td>Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L.</td>
<td>Hadoop, YARN, Stratosphere, Gi-</td>
<td>Statistic algo-</td>
<td>Amazon, WikiTalk, KGS, Citation, DotaLeague, Synth, Friendster</td>
</tr>
<tr>
<td>Willke [10]</td>
<td>raph, Graphlab, Neo4j</td>
<td>rithm, BFS, CC,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CD, GE</td>
<td></td>
</tr>
</tbody>
</table>

Because each study measures performance on different sets of algorithms and graphs, the results of these comparative studies have produced either incomparable or in some cases conflicting results. For example, Table 1.1 summarizes how three studies assess the relative merits of Giraph and GraphLab. Elser [6] finds that GraphLab outperforms Giraph on all graph datasets, while Han [11] finds that their relative performance varies, but they are comparable performing overall. Guo [10] uses a different set of graphs, producing similar, but incomparable results; also finding that relative performance varies, and provides no overall conclusion.

Results such as these make it difficult for a practitioner of graph computing to gain a clear perspective on which systems are preferable and in which contexts (algorithm, graph, scale). More fundamentally, because of previously mentioned confounding factors, even experts are
challenged to achieve any deep understanding of graph computing system performance. Hence, it is important to systematically understand the performance impact of graph algorithm and structure on graph processing systems, and thereby enable robust, systematic benchmarking in the graph computing community.

1.2 Our Work

To understand and quantify diversity in graph computation behavior, we study the dynamic properties of a collection of 11 graph algorithms chosen from diverse application domains. Using GraphLab as an experimental vehicle, these algorithms are run on a collection of 20 synthetic graphs of varying size and degree distribution. In total, we conduct 215 experiments, represented as graph-algorithm pairs. To characterize the behavior of each graph-algorithm pair, we capture several key properties ranging from vertex activity to compute intensity, and measure the behavior variation across different algorithms and different graphs. Our experimental results show that, not only do graph algorithms behave differently, but also each graph algorithm exhibits radically diverse behavior over different input graphs. In fact, the captured behavior of graph computation varies by up to three orders of magnitude across experiments, and forms a very broad space. Any benchmarking effort that fails to thoroughly explore this space, i.e. exercise only a small class of graph computation behavior, will possibly lead to a narrow understanding of graph processing performance, and risk unfair comparisons and potentially erroneous conclusions. Therefore, understanding performance over such a broad behavior space is necessary and also a big challenge for robust benchmarking of graph-processing systems.

So, to understand graph-processing behavior systematically, we define a vector space using 4 fundamental graph computation properties that also affect performance. To capture how well an ensemble of graph-algorithm pairs explores the behavior space, we further define ensemble metrics, spread and coverage, that measure the efficiency and completeness of this
exploration. With these metrics, we evaluate the benchmarking quality of different ensembles of graph-algorithm pairs.

We first look into ensembles that consist of experiments with single algorithms and single graphs, which seem to be a simple choice for benchmarking. Our results show that, neither single algorithms with a variety of graphs, nor multiple algorithms over single graphs explore the behavior space well, giving rise to capricious evaluation of graph-processing system.

We then consider to construct a high-quality benchmark suite that can exercise a broad graph computation behavior with minimum number of experiments. To achieve this goal, we search all available graph-algorithm pairs for best spread and coverage. Our study suggests that, by exploring both algorithm diversity and graph diversity, a small ensemble of carefully chosen graph-algorithm pairs could sample the behavior space much more efficiently and completely, giving 200% better spread and 30% better coverage than single-algorithm ensembles. This presents an opportunity to build a high-quality benchmark suite.

By looking into the contributions of each algorithm to best spread and coverage, we also find that not all algorithms are equally useful in high-quality benchmarking. Some graph algorithms, including K-Means, Alternating Least Squares, and Triangle Counting, are better choices than other algorithms. The reasons may be that these algorithms exhibit larger behavior variation across graphs, and some of them explore a unique region of the behavior space, making them valuable and difficult to replace in a thorough benchmarking.

We even consider to construct more efficient benchmark sets by reducing the complexity in algorithms, graphs and runtime. We explore what spread and coverage can be achieved if we reduce the number of algorithms and graphs, and shorten the runtime of several algorithms that have unvaried behavior over time. Our results show it is possible to construct a simpler benchmark suite by reducing algorithm complexity and reducing runtime while still conserving a high benchmarking quality.
1.3 Contributions

Specific contributions of this thesis include:

1. We conduct a series of 215 experiments varying algorithm and graphs (size and degree distribution) on a 48-node system, which demonstrates 1000-fold variation across five dimensions of graph computation behavior, including vertex activity, compute and inter-vertex communication intensity.

2. We define two ensemble metrics, spread and coverage, to measure how efficiently and thoroughly an ensemble of graph computation explores the behavior space of graph computation.

3. Using these metrics, we demonstrate that an ensemble drawn from a single algorithm or a single graph is a poor benchmark set, providing poor spread and coverage. Adding more experiments over the same algorithm or graph to an ensemble is of little benefit.

4. By exploring both algorithm diversity and graph diversity, we demonstrate that a diverse ensemble gives 200% better spread and 30% better coverage than the single-algorithm ensembles and single-graph ensembles, which presents an opportunity to build a high-quality benchmark suite.

5. Our insights into the best ensembles with high spread and coverage show that some algorithms, including K-Means, Alternating Least Squares, and Triangle Counting, are more useful than other algorithms in behavior space exploration.

6. By exploring simpler ensembles, we find that careful reduction in algorithm diversity minimizes loss of spread and coverage with further optimization possible by reducing runtime.

The remainder of the thesis is organized as follows. In Chapter 2, we introduce some background on graph processing, computation models and graph-processing systems. In
Chapter 3, we detail our experimental design, including the setup and workload, and define our metrics to capture fundamental behaviors in graph computation. The experimental results with each algorithm and graph are documented in Chapter 4, showing the great diversity of behavior. In Chapter 5, we consider ensembles of graph-algorithm pairs, and how to best design them for thorough graph system evaluation. Related work is discussed in Chapter 6, and Chapter 7 summarizes our work and suggests future research directions.
CHAPTER 2
BACKGROUND

This chapter introduces the basic concepts in graph computation, some basic computation models, and several widely used graph-processing systems.

2.1 Graph and Graph Processing

A graph is an abstract data structure that consists of a set of vertices that are connected by directed or undirected edges. It is an excellent data structure for modeling relationships and connections. Commonly, graphs are used to describe natural graphs in our real life, such as social networks where vertices represent people and edges represent friend links. In clustering applications, graphs are used to represent lists of data points. In collaborative filtering applications, graphs are used to represent recommendation matrices. Linear solvers use graphs to represent matrices, where each element corresponds to a weighted edge in the graph. For some Machine Learning algorithms based on random fields, graphs can model Markov chains, encoding random variables as vertices and relationships between those variables as edges.

Here we define several basic terms used in graph representation. The degree of a vertex is the number of edges that are connected to this vertex. Two vertices are called adjacent if they are connected by an edge; similarly, two edges are called adjacent if they share a common vertex. A weighted graph is a graph in which each edge is associated with some values, called edge weights. A directed graph, or digraph, is a graph in which each edge has a direction.

Graph processing is widely employed to solve many real-world problems, such as machine learning and data mining applications, social network analysis, business intelligence, and more. For example, Google’s PageRank [34] algorithm can determine where a web page appears in search result ranking. Facebook needs to process large social networks with more
than 900 million users in order to find communities and perform advertisement propagation. [11]

2.2 Computation Models

To aggregate different graph computations, several different computation models have been proposed and applied in current graph-processing systems, including vertex-centric model and edge-centric model.

In a vertex-centric computation model [4, 28], computation is expressed at the level of a single vertex. A vertex receives data from its adjacent vertices, updates the state of itself and its adjacent edges, and then sends data to its adjacent vertices. This procedure is iteratively performed to each vertex in the graph, until some termination criterion is met [18]. Vertex-centric model is widely employed in many graph-processing systems, including Pregel, Giraph, GraphLab, GraphX, and more.

In an edge-centric model [23], computation is applied to graph edges. Each edge keeps its own state and data, and data is transferred between adjacent edges. Edge-centric model is used by graph-processing systems such as X-Stream [23], to avoid random access into the set of edges.

There are also other computation models used, such as graph-centric model [28], path-centric model [39], and more. Despite the diversity of computation models, the fundamental behavior of graph computation is conserved – transferring information through edges, performing computation on an independent unit (vertex, edge, sub-graph, etc.), and iteratively updating the state of an independent unit until a convergence condition is met. Hence, when we talk about graph computation behavior, we do not specify the computation model and assume that same behavior is conserved across computation models.
2.3 Graph-Processing Systems

In this section, we will introduce several widely used graph-processing systems, including Pregel, Giraph, and GraphLab.

2.3.1 Pregel and Giraph

Apache Pregel [21] is a distributed computing platform for processing large-scale graphs. It employs the Bulk Synchronous Parallel (BSP) model, in which computations consist of a sequence of iterations and are performed in a synchronous way. As shown in Figure 2.1, in each iteration, Pregel applies the user-defined compute function locally to each vertex in parallel, updates the vertex values and states, and send messages to neighbors that can only be received in the next iteration. Algorithm termination is based on the states of vertices in the graph. Only those have an active state can participate in the computation of current iteration, and algorithm will terminates if all vertices become inactive.

Apache Giraph [1] originated as the open-source counterpart to Pregel, and is also inspired by the BSP model.

![Bulk Synchronous Parallel (BSP) model.](image)

Figure 2.1: Bulk Synchronous Parallel (BSP) model.
2.3.2 GraphLab

GraphLab PowerGraph [7] is another widely-used distributed graph-processing framework. It employs the Gather-Apply-Scatter (GAS) model, in which computations consist of three phases (see Figure 2.2):

1. **Gather** phase, where vertices receive information about adjacent vertices;

2. **Apply** phase, where vertices execute compute function using the gathered information;

3. **Scatter** phase, where the program updates data on adjacent edges.

Similarly, each vertex maintains its own state indicating whether it is active or inactive. Vertices can be activated by receiving signals from a neighboring vertex. Only active vertices can perform computation. The program terminates if all vertices become inactive. More details will be introduced in Section 3.3.1.

GraphLab supports two types of computation, the synchronous mode and the asynchronous mode. In the synchronous mode, the GAS phases are run synchronously on all vertices with a barrier at the end. In the asynchronous mode, the GAS process is performed to each vertex asynchronously, while the data consistency is guaranteed by applying a fine-grained locking protocol to all neighboring vertices hence the computation on a single vertex is performed in a sequential order.

![Figure 2.2: Gather-Apply-Scatter (GAS) model.](image-url)
Overall, Pregel, Giraph, and GraphLab express graph computation in a vertex-centric fashion. They apply local computation to vertices and transfer information by sending/receiving data through edges. These graph-processing systems are widely used in the research community and in many performance studies.
CHAPTER 3
EXPERIMENTAL METHODOLOGY

To characterize the fundamental behavior of graph computations, we study the dynamic properties of a wide range of graph computation experiments. Using GraphLab as our experimental vehicle, we execute our experiments on a small cluster, and the experiment setup is described in Section 3.1. Each experiment is a graph-algorithm pair, in which we vary graph features (scale and structure) and graph algorithms across diverse application domains (Section 3.2). To capture the fundamental behavior of graph computation, we define several performance metrics that quantify both compute intensity and communication intensity (Section 3.3).

3.1 Experiment Setup

We employ GraphLab PowerGraph v2.2 [7] as our graph-processing platform, because of its broad application in many performance studies. The algorithms used in our experiments are all GraphLab Toolkits [7], which are implemented libraries of algorithms on top of GraphLab. We execute these programs using the synchronous mode of GraphLab, where the Gather, Apply, and Scatter phases are performed without overlap. For programs not supporting synchronous mode, such as K-Means clustering, we modify their implementations without any change to the algorithms.

Each graph algorithm is executed on a variety of graphs. We change the value of graph features one at a time to isolate impacts on behavior. To limit the number of experiments, we use four different sizes, and five different degree distributions for each graph.

All the experiments were performed on the Midway system in Research Computing Center of the University of Chicago [22]. We used up to 48 nodes, connected by a fully non-blocking FDR-10 Infiniband network. Each node has two eight-core 2.6GHz Intel Xeon E5-2670
“Sandy Bridge” processors with 32GB of memory.

3.2 Workload

We describe a variety of graph computing problems, and algorithms and graphs that arise in them. We select fourteen algorithms from distinct application domains for their widely varying behaviors, such as different inter-vertex communication patterns, different vertex-local computations, and different graph traversal patterns. Since each application domain requires different formats of graph datasets, we use generators to create synthetic graphs or select real-world graphs with varied features.

3.2.1 Graph Algorithms

In this section, we introduce algorithms from multiple domains, including graph analytics, clustering, collaborative filtering, linear solver, and graphic models.

(1) Graph Analytics (GA) focuses on data mining, especially relationships from large graphs. We choose six algorithms, including Connected Components (CC), K-Core decomposition (KC), Triangle Counting (TC), Single-Source Shortest Path (SSSP), PageRank (PR), and Approximate Diameter (AD).

- Connected Components [32] of an undirected graph are the subgraphs in which any two vertices are connected to each other. To find all connected components in a graph, the CC program compares the IDs of adjacent vertices and only update a vertex if its ID is larger than the minimum value. A vertex only receives data from neighbors that activate it.

- A K-Core [5] of a graph is the largest subgraph where vertices have at least k interconnections. To find all K-Cores of the input graph, the KC program recursively removes
all vertices with degree $d=0, 1, 2, \ldots$. Similarly, a vertex only receives data from neighbors that activate it.

- **Triangle Counting** counts the number of triangles formed by three adjacent vertices in an undirected graph. For each edge in the graph, the TC program counts the number of intersections of the neighbor sets on both endpoints.

- **Single-Source Shortest Path** finds shortest paths from one special vertex to all other vertices. The source vertex is active initially. In each iteration, an active vertex computes and updates distances for adjacent vertices.

- **PageRank** [34] ranks web pages, considering links as votes where a page linking to another page is casting a vote. Pages with higher rank have more weight in links [16]. All vertices are active initially. A vertex becomes inactive when its rank remains stable within a given tolerance.

- **Approximate Diameter** estimates the diameter of a graph, which is the longest distance (i.e. the longest shortest path) between any two vertices.

(2) **Clustering** classifies objects based on their similarity. We choose K-Means algorithm (KM) that partitions $n$ vertices into $k$ clusters such that each vertex belongs to the cluster with the nearest mean [35]. All vertices remain active through the whole lifecycle. In scatter, each vertex sends messages to neighbors when the cluster assignment has changed, and those who receive no messages become inactive in the next iteration.

(3) **Collaborative Filtering (CF)** is a technique used by recommender systems to predict the ‘rating’ or ‘preference’ that user would give to an item, such as a movie or a book [31]. Each user is associated with a user-factor vector, and each item is associated with an item-factor vector. Prediction is done by taking an inner product [15]. Therefore, a general method for CF is learning all the user-factor vectors and item-factor vectors through matrix factorization. We select four algorithms:
• Alternating Least Squares method (ALS) is used for explicit feedback datasets, where unknown values are treated as missing, leading to a sparse objective function [15].

• Non-negative Matrix Factorization (NMF) is used to factorize non-negative matrices.

• Stochastic Gradient Descent (SGD) is a gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions [37].

• Singular Value Decomposition (SVD) decomposes a matrix into the product of unitary matrices and a diagonal matrix using Restarted Lanczos algorithm.

(4) Other algorithms include Jacobi method, Loopy Belief Propagation (LBP) algorithm and Dual Decomposition (DD).

• Jacobi method is an iterative method to solve a diagonally dominant system of linear equations.

• Loopy Belief Propagation is a discrete structured prediction application that can be applied to a wide range of predictions.

• Dual Decomposition solves a relaxation of difficult optimization problems by decomposing them into simpler sub-problems.

3.2.2 Graph Datasets

The format of graph datasets varies across application domains due to the requirements of graph computations:

• The inputs to Graph Analytics algorithms are unweighted graphs.
• In the domain of Clustering, vertices are data points (in this paper they are 2D vectors) and edges are pairwise rewards between vertices. So the inputs to Clustering algorithms include unweighted graphs, and vertex data in the format of vector lists.

• Inputs for Collaborative Filtering are weighted graphs, where source vertices of edges are users, target vertices are items to be recommended, and the weight of an edge represents the rating that a user gives to an item. Hence the total number of vertices is the sum of users and items. To simplify our experiments, we assume the number of items is equal to the number of users.

• Inputs of Jacobi include a matrix (also a weighted graph with uniform degree for all vertices) and a vector. Inputs of LBP include a pixel matrix and vertex data, which are prior estimates for each pixel color. To simplify our experiments, we only generate square matrices as the input to Jacobi and LBP. Inputs of DD are Markov Random Field (MRF) graphs in the standard UAI file format. For DD we use real-world MRF graphs downloaded from [13].

Therefore, we use graph generators for Graph Analytics, Clustering and Collaborative Filtering, and matrix generators for Jacobi and LBP. DD is the only algorithm that takes real graphs as input. We describe the details of generators in Section 3.2.3.

3.2.3 Graph Generators

For algorithms in Graph Analytics, Clustering and Collaborative Filtering, we use a synthetic graph generator to create graphs varying both size and structure (degree distribution). Graph size is measured in number of edges (\(n_{edges}\)), which is much larger than number of vertices, and determines the scale of a graph problem.

The degree distribution \(P(k)\) of a graph is defined to be the fraction of vertices in the graph with degree \(k\). Thus if there are \(n_k\) out of \(n\) vertices that have degree \(k\), we have
\[ P(k) = n_k/n \] [33]. Degree distribution characterizes the graph structure, and hence affects the behavior of topology-sensitive algorithms. Even across application domains, many real-world graphs exhibit similar degree distributions. Typically vertices with smaller degrees are more frequent than those with larger degrees. This is called a Power Law distribution, and graphs with such distributions are called scale-free networks [36]. More precisely, the fraction \( P(k) \) of vertices in the graph with degree \( k \) goes for large values of \( k \) as

\[ P(k) \sim k^{-\alpha} \quad (3.1) \]

Where \( \alpha \) is a constant typically ranging \( 2.0 \leq \alpha \leq 3.0 \) in the real world, although it may also lie outside these bounds. In this paper, we use \( \alpha \) to represent the degree distribution of a graph.

Hence, parameters of our graph generator include both \( n_{edges} \) and \( \alpha \). We set the values of \( n_{edges} \) to different orders of magnitude \( (10^5 \text{ to } 10^9) \). We also set the values of \( \alpha \) to match real-world scale-free graphs \( (2.0 \text{ to } 3.0) \), accepting slight variation in the number of vertices. For each application domain, we generate as many as 20 graphs varying size and degree distribution.

For Jacobi and LBP that require matrices as inputs, we use different graph generators. Since there is no need to vary graph structure for matrices, the only parameter for these generators is the number of rows of the matrix to be generated \( (n_{rows}) \), which determines the size of a graph/matrix. We set the values of \( n_{rows} \) as an arithmetic sequence \( (5000, 10000, 15000, 20000) \) so that the largest graph can fit into memory.

The variables describing graph features are shown in Table 3.1.

For all the synthetic datasets, the vertex data and edge weights are generated randomly and not varied, while their data types are determined by application domains. The way we generate vertex and edge data is indicated below.
Table 3.1: Graph Feature Variables

<table>
<thead>
<tr>
<th>Domains</th>
<th>Algorithms</th>
<th>Variables</th>
<th>Values of Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Analytics</td>
<td>CC, TC, KC, SSSP, PR, AD</td>
<td>$nedges$</td>
<td>$10^6$, $10^7$, $10^8$, $10^9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\alpha$</td>
<td>$2.0, 2.25, 2.5, 2.75, 3.0$</td>
</tr>
<tr>
<td>Clustering</td>
<td>KM</td>
<td>$nedges$</td>
<td>$10^6$, $10^7$, $10^8$, $10^9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\alpha$</td>
<td>$2.0, 2.25, 2.5, 2.75, 3.0$</td>
</tr>
<tr>
<td>Collaborative</td>
<td>ALS, NMF, SGD, SVD</td>
<td>$nedges$</td>
<td>$10^5$, $10^6$, $10^7$, $10^8$</td>
</tr>
<tr>
<td>Filtering</td>
<td></td>
<td>$\alpha$</td>
<td>$2.0, 2.25, 2.5, 2.75, 3.0$</td>
</tr>
<tr>
<td>Linear Solver</td>
<td>Jacobi</td>
<td>$nrows$</td>
<td>$5000, 10000, 15000, 20000$</td>
</tr>
<tr>
<td>Graphical Model</td>
<td>LBP</td>
<td>$nrows$</td>
<td>$5000, 10000, 15000, 20000$</td>
</tr>
<tr>
<td>Graphical Model</td>
<td>DD</td>
<td>$nedges$</td>
<td>$1056, 1190, 1406, 1560$</td>
</tr>
</tbody>
</table>

- Vector data for K-means: 2D-vectors generated in Gaussian distribution (mean is generated in uniform distribution within [-10, 10], and standard deviation = 2)

- Edge data for ALS, NMF and SGD: the dot-product of a random user-factor vector and a random item-factor vector. (Both vectors are generated in Gaussian distribution with mean = 0 and standard deviation = 2)

- Edge data for SVD and Jacobi: generated in Gaussian distribution with mean = 0 and standard deviation = 2. The matrix for Jacobi is diagonally dominant.

- Vertex and edge data for LBP are generated using the synthetic image data generator provided by GraphLab.

3.3 Measurement

In this section, we describe the fundamental behaviors in graph computation and define several performance metrics to characterize these behaviors.
3.3.1 Fundamental Behaviors

Our algorithms are implemented in GraphLab using the Gather-Apply-Scatter (GAS) model [7, 28]. We capture several fundamental behaviors from the overall GAS process:

- **Edge read.** In Gather phase, central vertices collect data through adjacent edges. Both the data to collect and the set of involved edges (in-edges, or out-edges) are user-defined. In this paper, the operation of collecting data through an edge is called an *edge read*;

- **Vertex update.** In Apply phase, central vertices apply user-defined computation and update their values. In this paper, the operation of updating one vertex is called a *vertex update*;

- **Message transferring and edge write.** In Scatter phase, central vertices send signals to activate neighbors. In this paper, a signal is called a *message*. Only vertices that receive messages can be active in the next iteration. In some implementations, the scatter phase also updates the value of adjacent edge, which is called an *edge write*. Since edge write is not generally used, we do not consider it in our current work.

Hence, inter-vertex communication includes both edge reads and message transferring. These fundamental behaviors are not specified only for the GAS model, but are representative for all kinds of graph computations despite of the diversity of computation model.

In this paper, we call a complete GAS procedure in the synchronous model an *iteration*. In our experiments, most tested algorithms converge in finite number of iterations, except NMF and SGD. For algorithms that do not converge, we set a maximum number of iterations (20).

3.3.2 Performance Metrics

To capture the fundamental behaviors described above, we define five performance metrics, including active fraction, UPDT, WORK, EREAD, and MSG. We choose these metrics
because they can fully describe the dynamic properties of graph computation, including graph traversal pattern, local compute intensity, and inter-vertex communication intensity. On one hand, in order to capture the behavior variation over time, we define *active fraction*, which is the ratio of active vertices to all vertices in a single iteration that directly shows the activity of vertices across the whole lifecycle. Since only active vertices perform computation per iteration, active fraction determines the amount of computation over the entire graph. Moreover, since vertices are activated by receiving signals, active fraction also affects the communication intensity. For many algorithms, active fraction varies over time.

On the other hand, in order to capture the overall behavior, we define \{UPDT, WORK, EREAD, MSG\}, which reflect the compute intensity and communication intensity of a graph computation.

1. **UPDT** is the average number of vertex updates per iteration.
2. **WORK** is the average CPU time for computing and updating vertex values per iteration, which is measured by time spent in user-defined apply function. Both UPDT and WORK measure how much work needs to be done for updating vertex values.
3. **ERead** is the average number of edge reads (data transferring) per iteration.
4. **MSG** is the average number of messages (signaling) per iteration.

Both EREAD and MSG measure how much information is transferred between vertices. Note that in many algorithms like PR, data transferring and signaling are different types of communications, so both contribute to the communication intensity.

Moreover, because the values of \{UPDT, WORK, EREAD, MSG\} can differ by orders of magnitude and highly depend on the number of edges, we divide each of these four metrics by the number of edges to capture the per-edge behavior. In order to highlight the relative difference rather than absolute values, we also normalize these metrics by linearly mapping the absolute values to \([0.0, 1.0]\).
CHAPTER 4
VARIATION IN GRAPH COMPUTATION BEHAVIOR

In this chapter, we present experimental results of executing different graph algorithms on a variety of graphs, which highlights the large variation in graph computation behavior across both graphs and graph algorithms.

4.1 Variation across Graphs

For each algorithm from each application domain, we observe large variation of 5 performance metrics when varying graph size and graph structure.

4.1.1 Graph Analytics

In Figure 4.1, different GA algorithms exhibit different shapes of active fraction; for each GA algorithm, active fraction exhibits similar trends over various graph structures. In detail, the shape of trends is classified by degree distribution, especially for CC, SSSP, and AD — these algorithms exhibit a higher peak and a sharper drop, and converge faster with more uniform degree distribution (i.e. a smaller $\alpha$. $\alpha$ is a constant in Equation 3.1). In contrast, KC, TC, and PR are less sensitive to graph topology. Specially, TC converges after only one iteration for all kinds of graphs. AD behaves stably, with active fraction = 1.0 for the whole lifecycle.

In Figure 4.2, CC exhibits no significant changes in algorithm behavior across graph scales, but a greater intensity of communication over more uniform degree distribution. Besides, data transferring ($EREAD$) and message transferring ($MSG$) are the same in CC program.

In Figure 4.3, 4.5 and 4.6, all 4 metrics heavily depend on graph size and degree distribution. All metric values are positively correlated to $\alpha$ in KC, whereas communication intensity is negatively correlated to $\alpha$ in SSSP and PR.
Figure 4.1: GA active fraction for all graphs. In each figure, there are four curves of the same color, representing 4 experiments over graphs with same $\alpha$ but different $n_{edges}$.

In Figure 4.4 and Figure 4.7, neither TC nor AD exhibits significant variation in behavior across graph size; both have constant $EREAD$ for all graphs; also, there is less computation, less updates, and less messages transferred per iteration when degree distribution becomes more uniform. (Note: experimental results of AD with $n_{edges} = 10^9$ are missing in this thesis.)

4.1.2 Clustering (K-Means)

In Figure 4.8, KM exhibits similar trends as AD, as it activates all vertices all the time. KM converges much more slowly than GA algorithms, requiring more than 700 iterations.

In Figure 4.9, KM behaves differently across graph sizes and degree distributions. Similar to AD, all metric values are positively correlated to $\alpha$, except $EREAD$ that is constant.
Figure 4.2: CC metric values across iterations.

Figure 4.3: KC metric values across iterations.
Figure 4.4: TC metric values across iterations.

Figure 4.5: SSSP metric values across iterations.
Figure 4.6: PR metric values across iterations.

Figure 4.7: AD metric values across iterations.
Figure 4.8: KM active fraction for all graphs. There are four curves of the same color, representing 4 experiments over graphs with same $\alpha$ but different $nedges$.

Figure 4.9: KM metric values across iterations.

4.1.3 Collaborative Filtering

In Figure 4.10, all of our Collaborative Filtering algorithms except ALS have all vertices active for entire lifecycle. Neither NMF nor SGD can converge automatically so that a maximum number of iterations (=20) is set. As the only outlier, ALS exhibits different
trends of active fraction across graph sizes and degree distributions. It converges much more slowly over larger graphs, showing a nearly 60-fold difference in the number of iterations.

Figure 4.10: CF active fraction for all graphs. In each figure, there are four curves of the same color, representing 4 experiments over graphs with same $\alpha$ but different nedges. Note that in NMF, SGD, and SVD, curves are on top of each other.

Figure 4.11 explains why ALS is so interesting as a benchmark. ALS behavior strongly depends on graph size and degree distribution. We observe high variation in the average value of all 4 metrics.

Figure 4.12, 4.13 and 4.14 indicate that none of NMF, SGD and SVD exhibits significant changes in behavior across graph sizes, except for the outlier of nedges $= 10^6$; for all these 3 algorithms, compute intensity is positively correlated to $\alpha$; for NMF and SVD, MSG is also positively correlated to $\alpha$.

4.1.4 Jacobi, LBP, and DD

In Figure 4.15, both Jacobi and DD activate all vertices, and Jacobi converges much faster than DD. The convergence rates of Jacobi and LBP are not affected by graph size.

In Figure 4.16, the behavior of Jacobi highly depends on graph scale; LBP and DD are less sensitive to graph size, while WORK is the only varied metric when graph size changes.
Figure 4.11: ALS metric values across iterations.

Figure 4.12: NMF metric values across iterations.
Figure 4.13: SGD metric values across iterations.

Figure 4.14: SVD metric values across iterations.
4.2 Variation across Algorithms

As shown in Section 4.1, all algorithms have a characteristic shape of active fraction that varies significantly across algorithms; some algorithms exhibit a varying active fraction over time, while other algorithms, including AD, KM, NMF, SGD, and SVD, have a constant active fraction and always activate all vertices. Moreover, the convergence rate differs a lot across domains, by up to three orders of magnitude (TC vs. DD).

Moreover, in Figure 4.17, different algorithms also exhibit quite different shapes of \{UPDT, WORK, EREAD, MSG\}. The values of all 4 metrics are much smaller in ALS, SSSP, KC, PR and LBP than in other algorithms. AD requires the most work for updating vertices, KM requires the most data transferring, and SGD requires the most message transferring. Overall, these algorithms behaves very differently in all dimensions of vertex activity, compute intensity and communication intensity.
In conclusion, graph computation behavior exhibits a wide diversity across algorithms and graphs, forming a very broad space. Any single experiment only samples a small part of that space, leading to ad-hoc performance studies that are unable to systematically understand the performance of graph-processing systems. Hence, we need to consider a more efficient way to sample the behavior space of graph computation.
In this chapter, we introduce a systematic approach to characterize graph processing system performance. Using this approach, we measure how efficiently and thoroughly an ensemble of graph computations explores the behavior space, and demonstrate several key implications for robust benchmarking.

### 5.1 Behavior Space and Ensemble Metrics

As described in Chapter 4, the behavior of graph computations forms a broad space. Hence, to systematically understand graph computations and graph-processing systems, we begin by defining a vector space for graph computation behavior. A vector space is useful in modeling graph computation behavior, because it fully presents all the dynamic properties we have described in Chapter 4, without any loss of information. It also makes it simple to compare behaviors of different graph computations.

Vectors in this space are composed from four performance metrics that we used in Chapter 4 to capture the fundamental behavior of graph computations—vertex value updates, compute per update, and edge reads, and messages transferred [1, 7, 21]. So we define $\text{Behavior}(GC_i)$ as follows:

$$\text{Behavior}(GC_i) = \langle \text{UPDT}, \text{WORK}, \text{ERead}, \text{MSG} \rangle$$

(5.1)

Where \{\text{UPDT}, \text{WORK}, \text{ERead}, \text{MSG}\} are as defined in Section 3.3.2, and $GC_i$ is a graph computation represented as a graph-algorithm pair (e.g. an experiment in Chapter 4), while each graph is a $<\text{nedges}, \alpha>$ tuple.

While our behavior space is designed to capture fundamental graph computation behavior, doing so optimally is an open research challenge; we define only one vector performance space, and evaluate its efficacy. Possible uses of our graph computation behavior character-
ization include basic algorithm analysis, algorithm comparison, performance analysis, graph computation optimization, performance prediction, system benchmarking, and even system design. Here we consider how to use the behavior performance space to design efficient, high-quality benchmarking of graph-processing systems.

As mentioned in Chapter 4, due to the large variation in graph computation behavior, any single graph computation only samples a small part of behavior space. Hence, a robust benchmarking study should employ an ensemble of graph-algorithm pairs, for systematically understanding the behavior space. We define an ensemble as a set of graph computations, \{GC_1, GC_2, \ldots\}. Then we can model a benchmark suite as an ensemble, and really characterize any set of performance experiments on a graph processing system as an ensemble.

Generally, we would like an ensemble of graph computations that maximize the characterization of a graph processing systems performance —over its entire behavior space (or the graph computation behavior space) with minimum effort. More formally we define:

\[ Ensembles_k = \{GC_1, GC_2, \ldots, GC_N\} \]  

Where \(N\) is the size of that ensemble. Given this definition, we would like to describe the quality with which an ensemble characterizes graph computation behavior. To this end, we define two ensemble metrics, spread and coverage, that quantitatively characterize the quality of an ensemble to thoroughly explore graph computing system behavior, but including a wide range of graph computation behavior.

**Spread** of an ensemble is defined as the mean pairwise distance between the Behavior vectors in an ensemble (see below).

\[ Spread(Ensemble_k) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} d(Behavior(GC_i), Behavior(GC_j))}{N(N - 1)} \]  

33
Where $d()$ is the Euclidean distance between two graph computation Behavior vectors. Intuitively, spread represents a form of “dispersion” of an ensemble. As shown in Figure 5.1(a), if the ensemble members are tightly clustered, then spread will be low; and if ensemble members are uniformly dispersed, it will be high. From a thorough evaluation point of view, spread should be maximized.

**Coverage** of an ensemble is defined as average minimum distance from all points in the space to the nearest point in the ensemble. Formally:

$$Coverage(Ensemble_k) = \frac{N_S}{\sum_{i=1}^{N_S} \min_{j=1...N}\{d(Sample_i, Behavior(GC_j))\}} \quad (5.4)$$

Where sample points are taken randomly and uniformly throughout the space to compute the coverage. $Sample_i$ is a sample point, and $N_S$ is the number of sample points (we use 1 million). Intuitively, the notion of good coverage is the idea that no matter where a behavior falls in the space, it will be close to a point in the ensemble. For an ensemble, a high coverage indicates that we have sampled the behavior space thoroughly (Figure 5.1(b)). High coverage is desirable for an ensemble that characterizes graph processing system performance well.

![Figure 5.1: Ensembles members in behavior space with low and high spread/coverage.](image)

Overall, coverage is a quality measure that indicates whether the ensemble capture the full range of graph computation behavior; spread is an efficiency measure that indicates the amount of information per graph-algorithm pair. So, in order to construct an ensemble of
5.2 Efficient Exploration of Behavior Space

In this section, we study different ensembles, and use our ensemble metrics to quantify how efficiently and thoroughly an ensemble explore the behavior space. In particular, we propose several questions and try to find answers to them:

- Many studies assess graph computation performance with a single algorithm or a single graph, which seems to be a simple choice for benchmarking. So first, we consider the question: *How well can an ensemble using a single algorithm or a single graph explore the behavior space?* Answers are discussed in Section 5.2.1 and 5.2.2.

- Given our broad exploration of graph computation behavior, we consider efficient and effective ensemble design retrospectively. Then we ask the question — *What are the ensembles that would explore the behavior space most efficiently?* We answer this in Section 5.2.3.

5.2.1 Single-Algorithm Ensembles

First, we consider the question: *How well can an ensemble using a single algorithm explore the behavior space?* To construct single-algorithm ensembles, for each algorithm we consider our collection of 20 runs varying graphs (<nedges, α> tuples). Jacobi, LBP and DD are not considered because their graph structures do not vary. Hence, for eleven algorithms, we have a total of 215 runs over 11 algorithms from across three application domains (GA, Clustering and CF). Unfortunately, 5 runs of AD with largest nedges = 10^9 failed. Therefore, to understand how effective a single algorithm can be for achieving high spread, for
each algorithm, we exhaustively test, and pick the best ensemble for each given size. Figure 5.2 presents our results, and shows that as the ensemble size increases, for all 11 algorithms, spread decreases steadily. This indicates that additional runs of a single algorithm tend to be clustered, and even selecting the best ones does not produce a good spread.

To understand the quality of the achieved spread, we also plot an empirical upper bound for spread for each given ensemble size. These are computed assuming ensemble members uniformly and maximally distributed in the behavior space. The spread achieved by single-algorithm ensembles falls well below our empirical upper bound.

Next we consider coverage. For each algorithm, we select the best ensemble members to maximize coverage. Our results (see Figure 5.3) show that restricted to a single algorithm, coverage increases very slowly. That is, adding ensemble members from a single algorithm do not spread out and cover the graph computation space. While the coverage achieved varies some across algorithms, the overall trend is the same.

To understand the quality of the achieved coverage, we also plot an empirical upper bound for coverage for each given ensemble size. Again, these are computed assuming ensemble members uniformly and maximally distributed in the behavior space. The coverage achieved by single-algorithm ensembles falls well below our empirical upper bound.

![Figure 5.2: Spread for Single-Algorithm Ensembles.](image-url)
5.2.2 Single-Graph Ensembles

Second, we consider: \textit{How well can an ensemble using a single graph explore the behavior space?} To answer this question, we select fifteen graphs with varied $n_{\text{edges}} = 10^6, 10^7, 10^8$ and $\alpha = 2.0, 2.25, 2.5, 2.75, 3.0$. For each graph with certain size and structure, we consider 11 runs over 11 algorithms to construct a single-graph ensemble. Overall, we have 165 runs. Considering spread, we construct the ensemble that maximizes spread for each graph. These results are shown in Figure 5.4, while none of the graphs enables spread anywhere close to the upper bound, the achieved spread is significantly higher than with single algorithms. Graph appears to be a more important factor in behavior variation than algorithm.

Considering coverage, we construct the ensembles that maximize coverage for each graph. Our results (see Figure 5.5) again show a flattening trend: no single graph is sufficient to fully explore the behavior space. And further, none approaches our upper bound. Compared to the ensemble drawn from a single algorithm, the ensemble around a single graph spread into the behavior space better, but slightly less completely.

\textit{Implications: If restricted to a single algorithm or a single graph, only limited graph computation behavior can be evoked. Adding more runs based on either the same algorithm or}
the same graph, generally leads to similar graph computation behavior. If we look at single-algorithm ensembles, they exhibit even less behavior variation than single-graph ensembles. In short, particular graph algorithms often behave similarly across graph structures and scales. Ensembles created with these properties (as used in numerous published graph computation system performance studies) risk unfair comparisons or incomplete results, and potentially erroneous conclusions.

Figure 5.4: Spread for Single-Graph Ensembles.

Figure 5.5: Coverage for Single-Graph Ensembles.
5.2.3 Unrestricted Ensembles

Examining our 215 runs, we ask the question — What are the ensembles that would explore the behavior space most efficiently? As before, the best ensembles avoid clustering (spread decreases slowly) and maximize exploration (coverage increases fast). Intuitively, the best ensemble corresponds to the most efficient benchmark suite of various sizes.

Considering spread, allowed unrestricted choice across multiple algorithms and graphs, it is possible to sample the space much more efficiently. Figure 5.6 presents the achievable spread with unrestricted ensembles and compares it to our prior results on ensembles drawn from single algorithms and single graphs. For unrestricted ensembles, spread starts high (∼ 1.2), and declines slowly to 0.75 for 20 members. In contrast, with only single-algorithm ensembles, spread starts much lower, and decreases to 0.25 for 20 runs. Compared to single-graph ensembles, the story is different. The spread of single-graph ensembles begins equally high, but falls off much more rapidly. Overall, there is a clear benefit in drawing richly from both algorithm and graph diversity, with as much as a three-fold greater spread.

Considering coverage, Figure 5.7 presents the best coverage for ensembles without restriction of algorithms and graphs. As with spread, ensembles that are drawn on both algorithm and graph diversity have clear advantage, delivering 30% better coverage than single-algorithm ensembles. The unrestricted ensembles achieve coverage that is significantly higher at as few as 5 runs, and the advantage grows even at 20 runs, achieving 3.9 overall.

Implications: Both algorithm diversity and graph diversity contribute to the behavior variation of graph computations. Hence, by exploiting both, we can identify multiple graph-algorithm pairs that exhibit diverse behaviors. We can use this information to construct an efficient and representative benchmark suite by carefully selecting graph-algorithm pairs that exhibit very different behavior and thoroughly sample the whole behavior space. In this way, this high-quality benchmark suite can exercise a broad graph computation behavior and systematically evaluate the performance of a graph processing system.
5.3 Implications for Robust Benchmarking

As shown in Section 5.2, benchmark suites built from single origins (algorithms or graphs) are inefficient in exploring the behavior space and will possibly lead to ad-hoc performance studies. Also, there is little benefit in adding more runs. In contrast, we can gain a significant improvement by exploring both algorithm and graph diversity and carefully choosing a small set of runs to build a high-quality benchmark suite.

In this section, we will give some deeper insight into more efficient benchmarking, by ex-
ploring what aspects of algorithm and graph diversity contribute most to better quality, and whether we can reduce the benchmarking complexity to conserve a reasonable quality.

### 5.3.1 Understanding Diversity

Our results for unrestricted ensembles give significantly better results, so a natural question is *What aspects of diversity in algorithms and graphs enable them to achieve better spread and coverage?* Because algorithms (and the associated implementation effort and code porting) are a major element of benchmarking complexity, we consider the representation of algorithms in the best ensembles. Table 5.1 lists the members of the unrestricted ensembles achieving best spread and coverage.

<table>
<thead>
<tr>
<th>Type</th>
<th>Ensemble Size</th>
<th>Ensemble Members (Algorithm[, nedges, α])</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best Spread</strong></td>
<td>5 runs</td>
<td>⟨ALS, 10^6, 3.0⟩, ⟨SGD, 10^8, 2.0⟩, ⟨TC, 10^6, 2.0⟩, ⟨SSSP, 10^6, 3.0⟩, ⟨ALS, 10^5, 2.75⟩⟩</td>
</tr>
<tr>
<td></td>
<td>10 runs</td>
<td>ALS, SGD, TC, SSSP, ALS, TC, SGD, ALS, KM, SVD</td>
</tr>
<tr>
<td></td>
<td>15 runs</td>
<td>SSSP, ALS, KM, SGD, ALS, TC, SGD, ALS, TC, SSSP, ALS, SGD, TC, SVD, ALS</td>
</tr>
<tr>
<td></td>
<td>20 runs</td>
<td>SSSP, ALS, TC, SGD, ALS, TC, SGD, ALS, KM, SSSP, ALS, SGD, KM, SVD, ALS, TC, SGD, ALS, SGD, TC</td>
</tr>
<tr>
<td><strong>Best Coverage</strong></td>
<td>5 runs</td>
<td>⟨TC, 10^6, 2.5⟩, ⟨KM, 10^6, 2.25⟩, ⟨AD, 10^7, 3.0⟩, ⟨ALS, 10^8, 2.0⟩, ⟨KC, 10^6, 2.5⟩⟩</td>
</tr>
<tr>
<td></td>
<td>10 runs</td>
<td>AD, SVD, KM, ALS, TC, KC, KM, ALS, KM, NMF</td>
</tr>
<tr>
<td></td>
<td>15 runs</td>
<td>KM, NMF, ALS, AD, SVD, KC, KM, ALS, KM, KM, SVD, PR, ALS, TC, NMF</td>
</tr>
<tr>
<td></td>
<td>20 runs</td>
<td>AD, SVD, KM, ALS, TC, KC, KM, ALS, KM, SGD, NMF, KM, ALS, NMF, PR, TC, NMF, SSSP, ALS, AD</td>
</tr>
</tbody>
</table>

Interestingly, some algorithms are consistently represented, Alternating Least Squares for best spread, and K-Means for best coverage. The best ensembles are complicated involving large numbers of algorithms and graphs. For example, the best five-member ensemble for
spread includes 4 algorithms and 5 different graphs. The best five-member ensemble for coverage includes five algorithms and 4 graphs. At the level of ten-member ensembles, the complexity already exceeds that of most comparative graph performance studies (6 algorithms for spread and 7 for coverage) [6, 10, 11, 12, 25].

To reliably assess “diversity contribution” of an algorithm, we would like to minimize shadowing effects. That is, considering only the best ensembles, a particular algorithm that is useful for spread or coverage might be shadowed by others that are slightly better, but when they are removed, really contribute greatly to diversity. We consider the question: Which algorithms contribute most often to the best ensembles for spread and coverage? To minimize shadowing, we expand our consideration of the best ensemble of size \(n\) to the 100 best ensembles of size \(n\) for each – spread and coverage. Within the 100 best ensembles, we use the frequency of appearance of each algorithm as an indication of contribution to diversity. These results are shown in Figure 5.8 and 5.9, and demonstrate that not all algorithms contribute significantly to a good spread or coverage. For example, K-Means, Alternating Least Squares, and Triangle Counting among our suite contribute to efficient and thorough behavior space exploration.

Implications: not all graph algorithms are equally useful in high-quality benchmarking. Some algorithms exhibit a greater diversity of behaviors across graphs (e.g. ALS appears several times in one best-spread ensemble), while others exhibit behaviors varying little and thus characterize a narrow class of behaviors. On the other hand, some algorithms explore a unique region of the behavior space, making them valuable and difficult to replace in a thorough benchmarking (e.g. KM appears in almost all best-coverage ensembles). Therefore, these special algorithms should be the first choice in the construction of a high-quality benchmark suite.
5.3.2 Reducing Benchmarking Complexity

Because the best ensembles require complex combinations of algorithms and graphs, it is worthwhile to consider simpler combinations to reduce benchmarking complexity. So the next question is: How does restricting our ensemble complexity impact achievable spread and coverage? Here we consider three dimensions of constraints, limited algorithms, graphs, and runtime (see Figure 5.10 and 5.11).

First, we limit ensembles to three algorithms, selecting those that contribute most to both spread and coverage. In this case, it is the same set for both metrics – KM, ALS and TC. The algorithm-limited suites maintain a high spread, and a slight advantage over single
algorithms.

Second, we limit ensembles to three graphs. The best ensembles use the graphs of $nedges = 10^7, 10^8 \text{ and } 10^9$ with $\alpha = 2.0$. The results indicate that limiting the number of graphs decreases spread rapidly and produces poor coverage even lower than single algorithms.

Third, we consider shortening graph computation runs for even more efficient benchmarking. Some of our algorithms, including AD, KM, NMF, SGD, and SVD, have constant, repetitive behavior (i.e. a constant active fraction=1.0). Their runs could be shortened, so ensembles using these 5 algorithms, can much more efficiently probe the behavior space. Because of the variety of these repetitive algorithms, these runtime-constrained suites still achieve high spread and coverage.

Implications: 1) A small collection of special algorithms can exercise a broad computation behavior. By employing these algorithms, we can reduce the number of algorithms used in benchmarking with minimal loss of quality. 2) Restricting graph selection significantly decreases benchmarking quality, because many algorithms exhibit similar behavior on the same graph structure. 3) Some algorithms that have unvaried behavior also have a high benchmarking quality, which enables us to further reduce runtime. Overall, by choosing these constrained runs to sample the behavior space efficiently, we can benchmark systems with minimum computational effort.

![Figure 5.10: Spread for Limited Algorithms, Graphs, and Runtime.](image)
In conclusion, our consideration of both single-algorithm and single-graph ensembles all suggest that beyond a few, adding more runs with the same algorithm or the same graph gives little benefit. By exploring both algorithm and graph diversity, a small number of carefully selected graph-algorithm pairs can improve the benchmarking quality significantly. We can further reduce the complexity in algorithms, graphs and even runtime, to benchmark systems with much less effort.
In this chapter, we discuss related benchmarking work and comparative graph-processing system experiments. For each, we summarize the related work, and explain how it relates to our efforts.

### 6.1 Benchmarking Efforts

General benchmark suites such as SPEC [26], NPB [3], TPC [29], and HPCC [14] employ a wide collection of programs, often with single inputs of varied sizes. Criteria for selection typically include [26] programs (1) representing real problems, (2) with targeted properties, such as compute-bound for CPU benchmarks, and IO-bound for IO benchmarks, (3) easy portability, and (4) free availability.

Taking SEPC CPU suite as an example, all the benchmarks aim to test “real-life” situation, including compiler, chemistry program, weather program, etc. The suite tests CPU performance by measuring the run time of these benchmarks, and calculates the geometric mean to get an overall score [38]. However, it is yet unclear whether these benchmarks explore all possible “real-life situations” thoroughly and efficiently.

Another example is HPCC benchmark suite, which aims to measure floating-point performance, memory bandwidth, and many other performance parameters of underlying system by stressing bottlenecks on CPU, memory, etc. Similarly, it is not clear whether these benchmarks can well represent real applications.

Therefore, though these benchmarks are carefully selected and stress bottlenecks in targeted systems, none are designed to systematically characterize performance. And even with their diversity, efficient sampling of the full behavior space is beyond reach.

Closer, there are several graph-oriented benchmarking efforts, including Graph500 [9], LDBC
[20], and LinkBench [2]. However, they suffer from the same limitations. Graph 500 uses only a single Breadth-First Search (BFS) program, on a single graph typically. LDBC executes several benchmarks on a single synthetic data set generated by a data generator. Since benchmarks drawn from either single algorithm or single graph will result in low spread and coverage, it is unrealistic for Graph500 and LDBC to fairly characterize a graph-processing system. Moreover, LinkBench provides general infrastructure, but not a specific benchmark.

6.2 Comparative Performance Studies

Comparative graph processing systems studies are numerous. We summarize a few of the most complete efforts here. M. Han et al. [11] compare Giraph, GPS, Mizan, and GraphLab while applying graph and algorithm agnostic optimizations. They employ four simple benchmarks: random walk, sequential traversal, parallel traversal, and graph mutation and five real-world graphs of different sizes. S. Salihoglu et al. [25] made similar study on Pregel-like systems. However, neither Han nor Salihoglu give any clear rationale for benchmark and algorithm selection, and no claims of thoroughness of behavior space exploration are made. Taking Hans study as an example, all the chosen benchmarks seem requiring much communication and few computation, which would possibly overestimate the performance of platforms that perform poorly in per-vertex computation.

B. Elser et al. [6] compare Map-Reduce, Stratosphere, Hama, Giraph, and GraphLab using a single algorithm, K-core decomposition and 7 different graph inputs. They compare cluster executions with a single multicore machine with shared memory and use runtime as the only metric. W. Han et al. [12] compare TurboGraph and GraphChi using simple benchmarks (out-neighbor queries, PageRank and Connected Components) and three graphs of different scales. Our results show that, neither a single complex algorithm such as K-Core, nor several simple benchmarks such as PR and CC explore behavior thoroughly. For W. Hans experiments, the use of a small number of graphs also narrows their study.
Y. Guo et al. [10] propose a comprehensive experimental method for benchmarking graph-processing platforms, and employ three dimensions of diversity: dataset diversity, algorithm diversity, and platform diversity. Their benchmarking suite has five classes of algorithms and seven diverse graphs, and they use it to analyze and compare six platforms. To explore diversity, they select real-world graphs with varied numbers of vertices and edges, and different structures (such as various average degree), and select graph algorithms including general statistics, graph traversal, connected components, community detection, and graph evolution. Metrics include basic performance (edges per second, vertices per second, etc.), resource utilization, scalability, and overhead. Guo’s work is perhaps closest to ours. The authors recognize the need to explore fully, and claim to do so. However, they present no clear framework of formal metric for assessing the thoroughness of exploration. In contrast, we have formulated a space and clear metrics for assessing thoroughness.

Our study suggests how to construct ensembles of graph computations that thoroughly explores the behavior space of graph computations. Beyond thoroughness, we describe how to further optimize an ensemble for efficiency and simplicity—limiting graphs, algorithms, etc. These ensembles can of course be used for benchmarking, and by exercising graph computing system behavior broadly, such a benchmark suite promises to give a comprehensive view of the performance of graph-processing systems, as well as an objective comparison between systems.

To the best of our knowledge, our study is the first and the only one that quantitatively characterize the behavior of graph computation and to propose an evaluation methodology for systematic benchmarking.
CHAPTER 7
SUMMARY AND FUTURE WORK

We have studied the behavior of a variety of graph algorithms (graph analytics, clustering, collaborative filtering, etc.) on a diverse collection of graphs (varying size and degree distribution). Our results characterize the variation of behavior across graphs, application, domains, and graph algorithms. Given the broad range of graph computation behavior, we consider how well an ensemble of graph computations (algorithms and graphs) explores this space, in terms of spread and coverage. Since ad-hoc sets are unlikely to fairly characterize a graph-processing system, our study gives suggestions on systematically choosing a small set of algorithms and graphs structures to exercise broad computational behavior. We find that ensembles over single algorithms and single graphs are inefficient in sampling the behavior space, while a set of carefully selected runs exploring both algorithm diversity and graph diversity give significantly better results. We propose more efficient ensembles by considering constraints such as limited algorithms, graphs, and even runtime.

With understanding of how ensembles of experiments sample the behavior space, more ambitious goals are within reach. Here we suggest several directions for future research.

1. Besides the overall behavior, we can also study the temporal structure and spatial structure of graph computation behavior, which determine the convergence rate, load balancing, etc. and will probably affect the performance. In the case of temporal structure, we can look at the behavior variation between neighboring iterations, and learn how graph computation varies and converges over time. In the case of spatial structure, we can look into the different parts of a graph and figure out the behavior variation across regions.

2. Since there are many different platforms used for processing graphs, including general-purpose frameworks such as MapReduce, and specific graph-processing systems such
as Pregel, Giraph and GraphX, we can extend our work and redo our studies with these other platforms. We can also redo our work on multiple parallel computing systems and explore both multi-core architecture and distributed architecture.

3. We can use our framework to analyze existing comparative studies and evaluate performance studies. Based on our work, it is possible to figure out which study can fairly compare graph-processing systems and which system are more preferable. It is also possible to figure out what limitations of these benchmarking efforts make them ad-hoc studies, and their impacts on the performance characterization.

4. In addition to constructing an optimal benchmark suite that explores the behavior space most efficiently, we can also design an ensemble that works as a basis of the behavior space. That is, we can define a set of synthetic graph computations, which are linearly independent and can span the entire behavior space. With such a basis ensemble, it is possible to precisely model any kinds of graph computation behavior, and further predict its performance.

5. With a thorough understanding on the behavior of a graph computation, a new direction of our work is to figure out the optimal configuration of underlying platform, which can best fit this kind of computation and achieve highest performance while minimizing resource requirements. For example, we can build a configuration model, which, taking graph computation behaviors as input, dynamically determines partitioning of the graph, memory allocation and number of compute nodes.
ACKNOWLEDGMENTS

We are grateful for the support of the University of Chicago Research Computing Center for assistance with the calculations carried out in this work.

I thank my advisor and committee member, Andrew Chien, who is also the coauthor of the conference version of this thesis. My study is guided by him from the beginning to the end. I have learned a lot from him, not only research skills, but more importantly, how to set and achieve my goals, which of course will benefit the whole life.

I acknowledge the other two committee members, Prof. Ian Foster and Prof. Hank Hoffmann, for their guidance, comments and suggestions on my research.

I also thank my friends and colleagues, who have given me a lot of useful feedback on my master’s thesis.
REFERENCES


