APPROXIMATING GEODESIC DISTANCE AND GRAPH CENTRALITY ON
SHARED NOTHING ARCHITECTURES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

ATILLA SONER BALKIR

CHICAGO, ILLINOIS
DECEMBER 2013
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. vi
LIST OF TABLES ......................................................................... ix
ACKNOWLEDGMENTS ............................................................... x

ABSTRACT .................................................................................. xi

1 INTRODUCTION ................................................................. 1
  1.1 Problem Statement ......................................................... 1
  1.1.1 Mining Large Graphs .................................................... 1
  1.1.2 Shortest Paths and Distance Estimation ......................... 2
  1.1.3 Graph Centrality ......................................................... 4
  1.1.4 Distributed Graph Processing Tools .............................. 5
  1.1.5 Statistical Patterns in Large Graphs and Common Computational Issues .......................... 7
  1.1.6 Contributions .............................................................. 8

2 BACKGROUND .......................................................................... 13
  2.1 Preliminary Definitions ..................................................... 13
  2.2 Distance Estimation in Large Graphs ................................. 14
    2.2.1 Simple Scalar Methods .............................................. 15
    2.2.2 Distance Approximation with Network Coordinates .......... 17
    2.2.3 Path Concatenation Methods ...................................... 18
    2.2.4 Limitations of Path Concatenation Methods .................. 25
  2.3 Graph Centrality: Definitions and Approximations ............... 26
    2.3.1 Degree Based Approaches ........................................ 27
    2.3.2 Closeness Based Approaches ...................................... 28
    2.3.3 Flow Based Approaches ............................................. 29
    2.3.4 Centrality in Large Graphs ........................................ 30
    2.3.5 Large Scale Graph Processing .................................... 34
    2.3.6 Summary ............................................................... 36

3 METHODS FOR EXACT AND APPROXIMATE DISTANCE COMPUTATION IN LARGE GRAPHS ................................................. 38
  3.1 Overview ........................................................................... 38
  3.2 Datasets ............................................................................ 39
  3.3 Parallel-SSSP .................................................................... 41
    Representing Vertices .......................................................... 42
    Algorithm Overview ........................................................... 43
    MAP .................................................................................. 45
    REDUCE .......................................................................... 45
Active/Inactive Vertex Labelling ........................................ 49
Selective Push using Bit Vectors .................................... 52
Parallel Bit Vector Merge Algorithm ................................. 54
2-level Parallel Merge .................................................. 55
k-level Parallel Merge .................................................. 56
3.3.1 Load Balancing High-Degree Vertices ......................... 58
3.3.2 Path Sampling .................................................... 61
Breadth First Sampling .................................................. 62
Depth First Sampling ..................................................... 62
Uniform Random Walk Sampling ........................................ 64
Weighted Random Walk Sampling ...................................... 67
3.3.3 Complexity in MapReduce ....................................... 67
3.3.4 Extending Parallel-SSSP to Weighted Graphs ............... 68
3.4 Distance Estimation with PathCrawler ......................... 69
3.5 Experimental Evaluation .......................................... 73
3.5.1 Cluster Setup .................................................... 73
3.5.2 Accuracy ........................................................ 73
Medium Sized Networks .................................................. 73
Large Networks .......................................................... 76
3.5.3 Performance and Scalability ................................... 76
Selective Push .......................................................... 76
Effects of the Small-World Phenomena ............................... 78
Performance Optimizations in Parallel-SSSP ...................... 80
3.6 Summary ............................................................ 84
4 PARALLEL SHORTEST PATHS AND APPROXIMATING CENTRALITY 86
4.1 Overview ........................................................... 86
4.2 Parallel All Pairs Shortest Paths ................................. 86
4.2.1 Naïve Parallel Implementation ................................ 87
4.2.2 Bucket-APSP ...................................................... 88
Generalization to Disjoint Subset of Vertices ..................... 91
4.3 Approximating Centrality ......................................... 91
4.3.1 Approximate Parallel Closeness Centrality ................. 91
4.3.2 Approximate Parallel Betweenness Centrality ............. 92
Suggesting a Sample Size .............................................. 93
4.4 Experimental Evaluation .......................................... 94
4.4.1 Accuracy ........................................................ 94
Exact vs. Approximate Closeness .................................... 94
Overlap of Exact and Approximate Vertices ....................... 96
4.4.2 Performance and Scalability ................................... 99
Increasing Input Size .................................................. 99
Increasing Cluster Size ................................................. 101
4.4.3 Intermediate Data vs. Run Time .............................. 104
4.5 Summary ............................................................ 106
5 CLOSING REMARKS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.0.1 Future Research Directions . . . . . . . . . . . . . . . . . . . . . . . . 107
5.0.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
LIST OF FIGURES

2.1 Illustration of the upper (left) and lower (right) bounds for landmark-based pairwise distance estimation. Re-drawn based on the original paper. .......................... 16
2.2 Illustrating path concatenation on a toy graph with eight nodes. .......................... 20
2.3 The short-cutting optimization demonstrated on the toy graph. The initially estimated path is optimized down to three hops. .......................... 22
2.4 Finding the lowest common ancestor on a shortest path tree. Notice that this is equivalent to detecting and removing the cycle [0, 1, 3, 2] from the initially estimated path. .......................... 23
2.5 Landmark-BFS with vertices 0 and 5 selected as the landmarks. The approximate path π′(3, 8) has length four. This figure is redrawn based on an illustration from the original paper. .......................... 25

3.1 Roadmap of the large scale graph algorithms. .......................... 38
3.2 Power law degree distribution of the large graphs used in the experiments .......................... 40
3.3 a) Sample undirected graph with no edge weights and nine vertices. b) The subgraph for vertex 7 which contains the output of SSSP for the pair (0,7). Three shortest paths are highlighted with arrows: {0,1,3,7}, {0,1,4,7} and {0,2,5,7}. .......................... 41
3.4 Logical representation of a vertex record. .......................... 42
3.5 Delta encoding of a neighbor list. Neighbors are sorted in ascending order and only the difference between two consecutive vertices is stored on disk except for the first one. .......................... 43
3.6 Operation of Parallel-SSSP on a sample graph with 8 vertices with ℓ = 0. Iterations 0 and 1. .......................... 46
3.7 Operation of Parallel-SSSP on a sample graph with 8 vertices with ℓ = 0. Iterations 2 and 3. .......................... 47
3.8 Incremental Graph Serialization for the sample graph when ℓ = 0. .......................... 49
3.9 Redundant messages are colored red whereas required ones are green. Notice the redundant messages sent from B to C in iteration 0, and A to B in iteration 1. The current distance of each vertex is displayed on the table in the right hand side. .......................... 50
3.10 Only active vertices send a message to their neighbors. A is active in iteration 0, and B is active in iteration 1. Redundant messages are avoided by the active/inactive labelling scheme. .......................... 51
3.11 a) Vertex 4 when it is active, along with its neighbors. b) The corresponding bit vector for the sample graph. Only those fields that correspond to the vertices above are highlighted. .......................... 52
3.12 Illustration of the 2-level Parallel Merge algorithm. There are 15 vectors and four map tasks. The first three map tasks merge four vectors each and the last one merges three vectors. The single reducer combines all four vectors received from the map tasks and produces the final answer. Note that both levels take t amount of time, in this case t is the time to merge four vectors. .......................... 57
Sample range partitioning on a vertex with degree eight. There are four map tasks and all of them pull the same vertex record from the pool on DFS. Each map task processes a disjoint subset of neighbors where subsets are determined by a partitioning function. Note that a vertex with degree eight is far from being a fat-vertex in practice. Furthermore, there are usually hundreds of map tasks spawned for a MapReduce job. The sample vertex with degree eight and four map tasks are chosen to demonstrate the concept on a micro scale. In practice, the fat vertex pool contains thousands of vertices with degrees ranging from tens of thousands to millions.

Breadth first sampling on a sample graph.

Depth first sampling on a sample graph.

Uniform random sampling on a sample graph.

Weighted random sampling on a sample graph.

Shortest paths from the landmark ($\ell = 0$) to vertices 7 and 8. Note that Parallel-SSSP can compute multiple shortest paths for each $(\text{landmark}, \text{vertex})$ pair and the results are stored on the distributed file system. In this example, there are three shortest paths between each of the pairs $(0,7)$ and $(0,8)$.

Combination of shortest paths that yield the true distance between the pair $0,7$ in the sample graph. The first set of paths is $\{0,1,4,7\}$ and $\{0,1,4,8\}$; and the second set of paths is $\{0,2,5,7\}$ and $\{0,2,5,8\}$. All other combinations yield sub-optimal answers.

Distance estimation accuracy for Facebook. Median pairwise distance is 4 hops. The Facebook dataset produced almost perfect results.

Distance estimation accuracy for Enron. Median pairwise distance is 4 hops. Slight approximation errors are visible around the (3,4) and (4,5) regions.

Distance estimation accuracy for Wikipedia. Median pairwise distance is 3 hops. A noticeable approximation error is visible around the (3,4) region.

Error Rate vs. Number of Landmarks.

Number of distance messages with and without the bit-vector for the first 15 iterations of Parallel-SSSP. Note that the WWW graph converges in approximately 60 iterations.

Hop plot for a randomly chosen vertex.

Shuffled vertex records in Parallel-SSSP.

Map task run times for iteration #4 over the WWW graph. Median task completion times are 954 and 340 seconds for the regular and optimized versions respectively.

Map task run times for iteration #3 over the Twitter graph. Median task completion times are 2183 and 78 seconds for the regular and optimized versions respectively.

Map output for a small graph with six vertices.

Map output of Bucket-APSP for the same graph in the previous figure.

Correlation coefficient for Facebook: 0.9997.

Correlation coefficient for Enron: 0.9968.

Correlation coefficient for Wikipedia: 0.9724.
4.6 Betweenness Centrality: Overlap of the most central vertices in the true and estimated results. ................................................................. 97
4.7 Closeness Centrality: Overlap of the most central vertices in the true and estimated results. ................................................................. 97
4.8 Betweenness Centrality in billion-scale: Change in run time with increasing input size. ................................................................. 100
4.9 Closeness Centrality in billion-scale: Change in run time with increasing input size. 100
4.10 Betweenness Centrality in billion-scale: Change in run time with increasing hardware resources. ................................................................. 102
4.11 Closeness Centrality in billion-scale: Change in run time with increasing hardware resources. ................................................................. 102
4.12 Number of buckets vs. run time and intermediate data in Bucket-APSP . . . . . 104
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of datasets</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Facebook Centrality Overlap</td>
<td>98</td>
</tr>
<tr>
<td>4.2</td>
<td>Enron Centrality Overlap</td>
<td>98</td>
</tr>
<tr>
<td>4.3</td>
<td>Wikipedia Centrality Overlap</td>
<td>99</td>
</tr>
<tr>
<td>4.4</td>
<td>Closeness Centrality</td>
<td>103</td>
</tr>
<tr>
<td>4.5</td>
<td>Betweenness Centrality</td>
<td>103</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS
ABSTRACT

This thesis presents a parallel toolkit for pairwise distance computation in massive networks. Computing the exact shortest paths between a large number of vertices is a costly operation and serial algorithms are not practical for billion-scale graphs. We first describe an efficient parallel method to solve the single source shortest path problem on commodity hardware with no shared memory. Given a source vertex $s$, the traditional solutions to this problem typically return one shortest path for each $(s, v)$ pair in the graph where $v \in V$. We slightly modify the original problem definition and extract multiple shortest paths between each $(s, v)$ pair. An important characteristic of large real-world graphs is the skewed degree distribution where the fraction $p(k)$ of vertices with degree $k$ asymptotically follows a power law. In parallel frameworks, the skewed degree distribution often results in straggling tasks and under-utilization of the computing hardware. A large number of idle computers wait for a small number of busy ones that consume a lot of CPU cycles while processing a few vertices with very high degree. The presented algorithm for solving the single source shortest path problem includes a load balancing technique to deal with the skewed degree distribution in large graphs. We also describe two novel optimization methods called Selective Push and Incremental Graph Serialization to avoid generating and exchanging redundant messages between compute nodes while minimizing the amount of data shuffled through the network.

Using the output of the first algorithm as a building block, we introduce a new parallel algorithm to estimate shortest paths between arbitrary pairs of vertices. Our method exploits data locality, produces highly accurate results and allows batch computation of shortest paths with 7% average error in graphs that contain billions of edges. The proposed algorithm is up to two orders of magnitude faster than previously suggested algorithms on a single computer and it does not require large amounts of memory or expensive high-end servers. Furthermore, this algorithm runs in parallel where each compute node can work on a different subset of vertex pairs independently. We present alternative parallel implementations of this method.
and discuss their efficiency using a communication cost model. To the best of our knowledge, this is the first algorithm that can approximate pairwise distances in large batches based on an embarrassingly parallel scheme.

We finally look at alternative methods to approximate the closeness and betweenness centrality metrics, which involve systems challenges dealing with indexing, joining and comparing large datasets efficiently. Exact computation of these measures involves solving the all pairs shortest path problem. Existing parallel algorithms for approximating closeness and betweenness centrality are designed for high-end shared memory symmetric multiprocessor architectures. Graph size is still a limiting factor as these algorithms are memory intensive, and they do not exhibit high degrees of cache and memory locality. There is also the hardware cost and availability issue. Most network scientists do not have direct access to super computers or high-end shared memory multiprocessor architectures. On the other hand, commodity clusters and shared nothing architectures with independent compute nodes are becoming widely available. They are also offered through cloud computing vendors in the form of infrastructure as a service. We use the parallel distance estimation algorithm to approximate the closeness and betweenness centrality metrics in large graphs. In one experiment, we mined a real-world web graph with 700 million nodes and 12 billion edges to identify the most central vertices and calculated more than 63 billion approximate shortest paths in 6 hours on a 20-node commodity cluster.
CHAPTER 1

INTRODUCTION

1.1 Problem Statement

1.1.1 Mining Large Graphs

Graphs are among the most pervasively studied models in computer science with broad area of applications in many disciplines including biology, chemistry, physics and, more recently, psychology and sociology. With the rapid increase in collective data gathered from the Internet and advances in data intensive computing frameworks, more researchers outside of computer science are getting interested in mining large graphs to draw useful statistical information, identify communities and analyze characteristics of real world graphs such as shrinking diameter and densification. Perhaps the largest real world graph that is of interest to many researchers is the World Wide Web, where pages are represented by vertices and hyperlinks are represented by edges. Previous studies on identifying topics and detecting entities [86, 17] are some of the examples that modelled the web as a large graph. Wikipedia has been modelled as a category network for performing document characterization [79], and discovering hub and authority relationships [56] using network analysis techniques [7]. Social networks are also becoming an increasingly popular resource for graph mining. Recently, public data collected from Twitter messages have been used to examine diurnal and seasonal mood variations across multiple countries [42]. Another study has shown that social network data can provide real-time assessment of influenza-like illness activity [1]. However, such studies make little use of the underlying link structure of large networks. There is a great potential to extract much more useful information from large graphs. Following proper computer science practice, we can identify basic primitives and seek to make them available as building blocks for use in more complicated algorithms achieving a bottom-up design approach.
In this dissertation, we focus on two primitives in graph algorithms; Finding the shortest path between arbitrary pairs of vertices, and measuring the relative importance of a vertex by computing its centrality. Essentially, we present parallel approximation algorithms that work on billion-scale graphs and provide a widely available toolkit that can run on cheap and small commodity clusters which can be obtained by most network scientists who have limited computing resources.

1.1.2 Shortest Paths and Distance Estimation

Computing the geodesic distance, i.e., the length of the shortest path between arbitrary pairs of vertices is a prominent problem in computer science. This problem and its variations have been studied extensively for over 50 years in the computer science literature, resulting in efficient algorithms for obtaining exact answers. In general, given a weighted graph with \( n \) vertices and \( m \) edges, efficient implementations of Dijkstra’s algorithm [19] can solve the single source shortest path problem (SSSP) in \( O(m + n \log n) \) time complexity. The all pairs shortest path problem (APSP) seeks to find the shortest path between all pairs of vertices in the graph. Floyd-Warshall [32] algorithm solves this problem in \( O(n^3) \) time and \( O(n^2) \) space complexity. Alternatively, one can call Dijkstra’s Algorithm \( n \) times, once for each vertex which gives \( O(mn + n^2 \log n) \) time complexity and can perform better in sparse graphs. Other methods have been proposed to improve upon Dijkstra’s Algorithm combining bi-directional search and \( A^* \) algorithm [48] to prune the majority of the vertices that would otherwise be traversed [50] [41]. However, these algorithms can not be adapted to massive graphs containing millions to billions of vertices and edges as they require an enormous amount of memory space and computing power. A full breadth-first search traversal of a relatively small network having 4M vertices and 50M edges has been reported to take approximately one minute in a standard desktop computer [73] as of 2010. Computing all pairs shortest paths is a harder task and pre-computing them to further use in other algorithms is impractical since the same graph would require storing 16 trillion elements.
Even an optimistic yet unlikely scheme that requires only one bit per element would occupy more than 1.8 Terabytes of space, requiring a distributed storage mechanism to be efficient. However, we have recently shown [5] that in parallel applications where model parameters are rapidly queried against a distributed database, the network round trip time can introduce a severe performance bottleneck. In addition, several social networks already have hundreds of millions of vertices making the analysis much costlier than the presented numbers above. Thus, in large networks, computing the exact shortest paths between all pairs of vertices is not practical. To address this problem, several fast approximation algorithms have been developed recently.

The first step towards approximating shortest paths in large graphs is to perform some pre-computation to index and summarize the link structure of the graph. In landmark-based methods [73, 69, 23, 85, 44, 67], this involves selecting a set of vertices called landmarks and computing the exact shortest paths from the landmarks to the rest of the graph. Using the shortest path trees rooted at the landmarks, the distance between an arbitrary pair of vertices can be computed in almost constant time.

The main focus of landmark-based methods has been on providing fast approximations to shortest path queries. Performance gain is achieved by the assumption that the pre-computed trees from the landmarks are small enough to fit in memory. This information is typically stored in a high-end multi-core server for random memory access. The shared memory central server approach has inherent scalability limitations. The computation time increases as the graph gets denser. Also, the performance of these methods under heavy load with many distance estimation requests arriving in parallel has not been evaluated. Recent studies show that applications running on multi-core architectures tend to be limited by off-chip bandwidth due to shared resources in the memory hierarchy [25, 63]. There is no distance estimation algorithm in the literature that can handle billion-scale networks, run on cheap commodity hardware and compute approximate shortest paths in parallel without a central storage scheme.
1.1.3 Graph Centrality

In graph theory, measuring the relative importance of vertices within a graph is accomplished by computing its centrality. For example, the most well connected people in a microblogging social network create a hub effect by forwarding shared information to their peers. An army can consider the highway road map as a graph to detect the most crucial points for moving equipment. Some proteins have been shown to evolve more slowly and be more likely to be essential in a protein-protein interaction network [47]. Detecting such critical vertices in a graph structure is accomplished by defining and evaluating several centrality metrics. Common centrality measures include simple approaches such as degree centrality as well as more complicated ones including closeness, betweenness and Eigenvector centrality. Among these measures, some variations of Eigenvector centrality [8] have been studied well in the area of distributed systems. For example, the PageRank algorithm [12] is a variant of Eigenvector centrality and computing it efficiently in large-scale gave rise to ubiquitous data-intensive programming abstractions [24][51]. On the other hand, little work has been done on approximating betweenness and closeness centrality in large graphs. Finding shortest paths is an essential step for both of these measures and exact computation requires solving the all pairs shortest path problem (APSP), which is prohibitively expensive for billion-scale graphs.

Existing parallel algorithms for approximating closeness and betweenness centrality are designed for high-end shared memory symmetric multiprocessor architectures [4, 26]. Graph size is still a limiting factor as these algorithms are memory intensive, and they do not exhibit high degrees of cache and memory locality [66]. There is also the hardware cost and availability issue. Most network scientists do not have direct access to supercomputers or high-end shared memory multiprocessor architectures. Parallel programs running on supercomputers are typically sensitive to high latencies and require high bandwidth interconnects. MPI [84] can run on relatively cheap clusters of computers but fault tolerance is not addressed well, and usually requires additional effort from the programmer by check-
pointing. There is also a steep learning curve, which often results in experts outside of the parallel computing area repeatedly solving the same challenges that involve synchronization, communication and parallel I/O. A more appealing alternative would be to choose a widely available parallel framework that does not require low latency/high bandwidth interconnects and can run on cheap hardware while automatically dealing with node failures. Example parallel frameworks that meet these requirements are MapReduce [24], GraphLab [64] and Pregel [68]. Some parallel tool-kits built on top of MapReduce have been proposed to compute Eigenvalue centrality [53]. However, their counterparts for approximating closeness and betweenness centrality are currently missing aside from recently introduced similar measures that can be computed in parallel, but are not direct replacements to the original metrics [52].

1.1.4 Distributed Graph Processing Tools

Several frameworks have been proposed to solve problems that concern large graphs. We found particularly three of them related to the scope of this thesis which aims to solve distance based graph mining problems using cheap hardware.

GraphLab [64, 65] is a programming abstraction to express dynamic graph-parallel computations asynchronously. The computation is carried out as a distributed loop that processes vertices in parallel, based on a prioritized ordering scheme in shared memory environments. Pregel [68] is a functional programming based model that aims to solve large-scale graph problems. The unit of computation is called a superstep during which, a vertex exchanges messages with its neighbors. The graph structure including the vertices, edges and the computation state is stored in distributed memory. Both frameworks are built on simple yet comprehensive abstractions, provide fault tolerance and have demonstrated extensive usability for graph mining and machine learning problems.

MapReduce is a parallel programming abstraction widely used in processing large datasets using commodity clusters [24]. MapReduce applications are not limited by the total cluster
memory. Local disk is actively used for storing intermediate data during the computation. High throughput is achieved by carrying out the same computation on different parts of the data in large batches. These features of MapReduce allow scaling to arbitrarily large datasets at the cost of local disk access and extra computation time for grouping and managing the data in the cluster. In graph mining, MapReduce has been used for detecting connected components, enumerating triangles and subgraphs, diameter estimation and finding subgraphs of high connectivity [53, 18, 2, 83].

From a distributed graph computation standpoint, the fundamental \textit{vertex-centric} methodology behind all three abstractions above is the same and centers upon the following primitives:

- Partition the graph.
- Process each vertex independently.
- Send/receive messages between neighboring vertices.
- Synchronize and repeat the above steps until convergence.

In this dissertation, the main concentration is on the actual work done by the algorithms rather than potential speed-ups in execution time resulting from the platform of choice. We address performance issues related to large scale graph mining algorithms common to all distributed platforms. All algorithms in this dissertation are based on the MapReduce framework. We choose MapReduce for its availability as well as the extensive community, library and platform support. However, all methods and optimization techniques in this dissertation can be implemented using any other framework optimized for large scale graph processing with minor modifications as long as it supports the \textit{vertex-centric} parallel programming style.
1.1.5 Statistical Patterns in Large Graphs and Common Computational Issues

Real-world graphs are called **scale-free** when they exhibit skewed degree distributions where the fraction $p(k)$ of vertices with degree $k$ asymptotically follows a power law given by

$$p(k) = A k^{-\gamma}, \quad \gamma > 1 \text{ and } k \geq k_{\text{min}}$$  \hspace{1cm} (1.1)

Examples of scale-free networks include the World Wide Web graph [57] and the citation network between scientific papers [76]. A common characteristic often observed in scale-free networks is the **small-world** phenomenon (a.k.a six degrees of separation) which states that the diameter of large graphs is usually not too long. In addition, previous studies show that the graph diameter tends to get smaller as the graph grows over time [61, 54]. This is known as the **shrinking diameter** phenomenon in scale-free dynamic networks.

There are two consequences of the above characteristics and phenomena in processing large graphs. The **vertex-centric** computation methodology statically assigns each vertex in the graph to a particular process. Due to the skewed degree distribution, processes that are responsible from a few high-degree vertices may fall behind and take longer to complete. This results in the **straggling task problem**: A large number of computers/CPU cores stay idle and wait for the remaining ones to finish during the synchronization steps of a parallel algorithm.

The second issue arises from the **small-world** phenomenon. The **vertex-centric** approach requires sending messages from a vertex to its neighbors. Since the diameter of real world graphs are small, a large number of edges can be visited within a small number of passes over the input graph. In other words, the number of vertices that send/receive messages can grow exponentially resulting in massive message traffic between compute nodes. This may dramatically slow down the computation and further cause compute node failures.
This dissertation presents examples to these problems in large real-world graphs. We describe general optimization techniques to deal with these issues. The suggested optimizations can also be used by similar graph algorithms designed for other parallel platforms.

1.1.6 Contributions

This dissertation presents Hoba: A Large-Scale Graph Library for Approximating Distance and Centrality in Shared Nothing Architectures. HOBA is a parallel graph mining toolkit capable of scaling to arbitrarily large networks that have billions of vertices and edges. It contains efficient parallel algorithms for approximating distance and graph centrality on commodity hardware. The main contributions of this thesis are listed as follows:

- An optimized parallel algorithm for the single source shortest path problem

  We describe a new MapReduce algorithm to solve the single source shortest path problem. The output contains multiple shortest paths between all \((s, v)\) pairs where \(s\) is the source and \(v \in V\) is an arbitrary vertex. Since there are multiple shortest paths between each pair, the output of this computation can be quite large considering massive graphs with billions of vertices and edges. We present an efficient storage technique by combining multiple shortest paths into sub-graphs and applying graph compression methods to each sub-graph. The suggested algorithm involves three key optimization methods, each tackling a different problem related to large-scale graph processing.

  - A load-balancing technique in MapReduce which solves the straggling task problem resulting from the skewed degree distribution of a large network.
  - Selective Push technique to reduce the amount of messages exchanged between processes. This optimization essentially reduces the number of edges traversed globally and avoids side effects of the small-world phenomenon in large graphs.
Incremental Graph Serialization minimizes the distributed file system IO by gradually serializing the entire graph over multiple iterations.

The proposed algorithm produces exact answers, runs in nearly linear time for unweighted graphs and involves several optimization techniques to minimize the amount of data sent over the network. Experimental evaluation shows that it is up to seven times faster than naïve implementations.

- **Accurate estimation of pairwise distances**

  Distance estimation is an essential building block for deep network structure analysis. We describe a large-scale approximation algorithm by primarily following the landmark approach, which is based on using the outcome of basic pre-computation steps to obtain accurate estimates. These steps involve

  - Finding the shortest paths from a few landmarks to the rest of the graph.
  - Storing the results as pre-computation data on a distributed file system.

Note that the above two steps are handled by the first contribution, which solves the single source shortest path problem in large scale. The estimation algorithm then makes use of these data to estimate the distance between arbitrary pairs of vertices in the graph. To batch-process and rapidly compute all pairwise distances within a subset of the graph, we relax some primitives of the existing serial methods such as random access to the pre-computed results from memory, and fit these requirements into the MapReduce programming style. By exploiting the data intensive processing capabilities of MapReduce, the new algorithm is designed based on the shared-nothing principle to horizontally scale, and perform multiple distance estimation operations in parallel. The use of MapReduce enables multiple compute nodes to work on separate pairs of vertices independently without requiring random access to a shared-memory structure. In addition, the distance approximation algorithm is able to use more pre-computation
data to increase the accuracy. This would be infeasible without parallel programming practices as serial implementations rely on storing the entire pre-computation data in memory.

It is worth noting that the main concentration of this algorithm is to estimate a large number of pairwise distances using a batch-processing approach. This is different than quickly responding to a few distance estimation queries, which is what the existing serial implementations are designed for. In particular, we eliminate the random access to shared memory requirement in the previously suggested landmark-based methods and leverage more pre-computation data to increase the accuracy of the results. We achieve average error rates between 0.02% and 7% in several real-world graphs ranging from a few thousand to more than 700 million nodes in size. In a single compute node, the proposed method can return multiple approximate shortest paths between a pair of vertices in under 8 milliseconds on average, over 100 times faster than previously reported results on comparable graphs [85].

- **Detecting prominent vertices using centrality metrics**

  Deciding the importance of a vertex in a graph is a complex task. The concept of graph centrality tries to answer this question in several different dimensions including degree-based, flow-based, and diameter-based measures. Whereas the first can easily be extracted from the graph meta-data without much effort, the other measures are non-trivial to approximate in large graphs. Computing diameter-based and flow-based centrality measures involves discovering all vertices that are reachable from a given vertex within $k$ hops, knowing its average distance to the rest of the graph, and counting how many times it occurs in the critical paths. It is possible to bypass the inherent scalability limitations introduced by these requirements in billion-scale graphs by an accurate distance estimation algorithm and careful parallel design. In large networks, issues related to principal methods in centrality computation such as counting neigh-
boring vertices, comparing shortest paths, and choosing the most optimal values can be transformed into large-scale systems problems dealing with indexing, joining, and comparing large data sets efficiently.

We use the parallel distance estimation algorithm to approximate the closeness and betweenness centrality metrics in large graphs. Two alternative implementations of these methods in distributed environments are presented and their efficiency is discussed using a communication cost model. Experiment results suggest that we can identify the most central vertices with up to 96% accuracy on medium sized networks. The proposed algorithms can easily handle billion-scale networks running on small commodity clusters with less than 100 physical cores.

- A widely available toolkit running on cheap hardware

The choice of platform to implement massive scale graph algorithms is critical for availability. A useful system should be built on a higher level of abstraction, easy to set-up and have a gradual learning curve. MapReduce gained considerable recognition from the scientific community as it is highly abstract, comes coupled with a distributed file system, and shields the programmer from the lower level details of node-to-node communication and synchronization issues. Computers with cheap commodity hardware as well as several cloud computing platforms can run a MapReduce cluster with minimal effort. We present all algorithms to meet the objectives above in the form of a parallel framework built on top of Hadoop, an open source MapReduce implementation. We hope that the proposed distance and centrality estimation algorithms encourage the development of more accurate and complex graph analysis algorithms. The ultimate objective of this toolkit is to provide researchers from any discipline with a parallel graph mining system that allows users think on the actual graph problems rather than how to overcome parallel design challenges. The current version of Hoba is open source and it is compatible with the Amazon Elastic MapReduce cluster. This
allows any network scientist to mine massive graphs for moderate costs. With Hoba, we were able to handle billion-scale graphs easily on a 20-node commodity cluster. We encourage using Hoba for social network analysis and web mining especially in small labs or research groups with limited computing resources.
CHAPTER 2
BACKGROUND

Many solutions to the problems of distance estimation, computing centralities as well as
mining large networks in parallel have been researched and studied over time. This section
aims to review the prior work. In each category, basic definitions are introduced and notable
background work is summarized. The main course of this section is to show where previ-
ous studies fall short on each relevant category and how this dissertation addresses these
problems.

For the rest of this dissertation we assume the graphs are unweighted for simplicity,
unless stated otherwise. However, all algorithms described in this dissertation are applicable
to weighted graphs as well. We explain how to cover the weighted setting when we describe
each algorithm in the corresponding subsection.

2.1 Preliminary Definitions

Let $G$ be a graph with with $|V| = n$ vertices, and $|E| = m$ edges. Given two arbitrary
vertices $v_0$ and $v_k$, a path $\pi(v_0, v_k)$ of length $|\pi(v_0, v_k)| = k$ is an ordered sequence of
vertices $\{v_0, v_1, ..., v_k\}$ such that

\[
v_i \in V, \quad 0 \leq i \leq k \quad \text{and} \quad (v_i, v_{i+1}) \in E, \quad 0 \leq i < i + 1 \leq k, \quad i, k \in \mathbb{N}
\]

A graph is connected if and only if there exists a path between any pair of vertices. The
shortest path between two vertices $u$ and $v$ is denoted by $\pi^*(u,v)$. Their distance $d(u, v)$ is
the length of the shortest path, i.e., $d(u, v) = |\pi^*(u, v)|$ or $\infty$ if $v$ is not reachable from $u$.
The term geodesic path is also used to represent shortest paths.
The **diameter** $D$ of a graph is the longest shortest path between any two vertices. Note that $D < \infty$ for connected graphs.

The **effective diameter**, given by $D'$, is the minimum number of hops in which 90% of all vertex pairs can be reached [81]. In some resources, the 90% threshold can be higher or lower depending on the context. Essentially, effective diameter represents the minimum number of hops to cover the majority of the graph.

Given two paths $\pi(u, v)$ and $\pi(v, s)$, there exists another path $\pi(u, s)$ constructed by concatenating the two as

$$\pi(u, s) = \pi(u, v) \cdot \pi(v, s)$$  \hspace{1cm} (2.1)

where $\cdot$ is the path concatenation operator. Observe that $\pi(u, s)$ is not necessarily the shortest path between $u$ and $s$ and it may contain cycles. In general, the following triangle inequality

$$d(u, s) \leq d(u, v) + d(v, s)$$  \hspace{1cm} (2.2)

holds for any three vertices $\{u, v, s\} \in V$.

### 2.2 Distance Estimation in Large Graphs

In a directed graph with unbounded non-negative edge weights, Dijkstra’s algorithm [19] can compute the shortest path from a given source vertex to the rest of the graph in $O(m+n \log n)$ time complexity using a Fibonacci heap [34]. The all pairs shortest path (APSP) problem is more challenging and can be solved by invoking Dijkstra’s algorithm $n$ times, resulting in $O(nm + n^2 \log n)$ time complexity. However, the Floyd-Warshall [32] algorithm addresses this problem by progressively computing all shortest paths with a dynamic programming approach, which yields $O(n^3)$ time and $O(n^2)$ space complexity. Some large graphs such
as the Twitter graph that contains who-follows-who information, and the Facebook graph containing friendship relations with hundreds of millions of vertices can be represented as directed unweighted graphs, where exact single source shortest paths can be calculated with simple breadth first search (BFS) having $O(n + m)$ time complexity. Invoking the BFS algorithm $n$ times gives $O(n^2 + mn)$, which is still quadratic with the number of vertices. In addition, such graphs are dense and generally $m$ out-weighs $n$ by orders of magnitude. This is intuitive for social networks since people (vertices) tend to increase their interactions (edges) over time.

Many extensions and improvements to Dijkstra’s single source shortest path (SSSP) algorithm have been suggested. Recent methods combine bi-directional search and $A^*$ algorithm to prune the majority of the vertices that would otherwise be traversed [50] [41]. Nevertheless, the main limitation of these algorithms is their requirement of a shared data structure such as a priority queue that orders vertices based on their actual or approximate distance to the source. In a parallel setting, ordering vertices introduces an inherent serialization barrier. This justifies the need for computing pairwise shortest paths for massive graphs in parallel using approximation techniques. The main focus of previous work on distance estimation in large networks has been on providing fast distance approximations to shortest path queries.

### 2.2.1 Simple Scalar Methods

Potamias et al. [73] describe an algorithm to quickly estimate the length of the point to point shortest paths (PPSP) using a landmark (sketch) based method. They pre-compute shortest path trees from a sample set of landmarks and then estimate the distance between a pair of vertices $(s, t)$ using upper and lower bounds based on the following triangle inequality observations.

Let $d(s, t)$ be the length of the shortest path from $s$ to $t$. Then, for any other vertex $u \in V$ the following upper and lower bounds exist:
Figure 2.1 exemplifies the above bounds for sample vertices $s$, $t$ and $u$.  

$$d(s, t) \leq d(s, u) + d(u, t) \quad (2.3)$$

and

$$d(s, t) \geq |d(s, u) - d(u, t)| \quad (2.4)$$

Figure 2.1: Illustration of the upper (left) and lower (right) bounds for landmark-based pairwise distance estimation. Re-drawn based on the original paper.

In the pre-computation step, a set of landmarks $\mathcal{L} = \{\ell_1, \ell_2, ..., \ell_k\}$ where $\mathcal{L} \subset V$ are sampled and the single source shortest path problem (SSSP) is solved for each $\ell_i \in \mathcal{L}$. This is achieved by doing a BFS traversal of the graph for unweighted graphs or running Dijkstra’s Algorithm for the weighted case. The actual paths from the landmarks are ignored and only pairwise distances are stored in memory. By the above inequalities, the true distance between any two vertices $s$ and $t$ has the lower and upper bounds $L$ and $U$ such that:

$$L \leq d(s, t) \leq U \quad (2.5)$$

where

$$L = \max_{\ell_i \in \mathcal{L}} |d(s, \ell_i) - d(\ell_i, t)| \quad (2.6)$$

$$U = \min_{\ell_j \in \mathcal{L}} \{d(s, \ell_j) + d(\ell_j, t)\} \quad (2.7)$$

Based on experimental evaluation, Potamias et al. recommend simply using $U$ as the
estimated distance between any arbitrary pair of vertices. That is, the approximate distance \( d'(s, t) \) is given by

\[
d'(s, t) = \min_{\ell_i \in L} \{ d(s, \ell_i) + d(\ell_i, t) \}
\]  

(2.8)

This algorithm takes \( O(k) \) time to approximate the distance between a pair of vertices and requires \( O(nk) \) space for the pre-computation data. A limitation of this approach is that it provides a simple scalar estimation on the distance, without a path. In addition, although this method meets the objective of estimating pairwise distances quickly, more accurate results can be obtained by leveraging the pre-computed paths from each landmark.

An important contribution of Potamias’ work is the analysis for selecting the best landmarks. They show that finding the minimum number of landmarks to obtain the true distance between all pairs of vertices in a graph is an \textbf{NP-hard} problem.

### 2.2.2 Distance Approximation with Network Coordinates

Zhao et al. \cite{89} propose a distance estimation system in large graphs by an approach similar to network coordinate (NC) systems \cite{20, 22, 69}. They implement a graph coordinate system by mapping vertices in a graph to an n-dimensional Euclidean space to compute pairwise distances in constant time.

In network coordinate systems, pairwise distances between Internet hosts are measured by Internet PING. Although fast, this method leads to further challenges since such sensitive measurements can violate the triangle inequality in the network. For example, \( t_{\text{ping}}(A, B) \) can be greater than \( t_{\text{ping}}(A, C) + t_{\text{ping}}(C, B) \) since PING times are affected by network latency, which is not always correlated with the actual physical locations of the Internet hosts.

In their paper describing graph coordinate systems, Zhao et al. suggest using landmarks as fixed points in an n-dimensional Euclidean space. Initially, \( k \) landmarks are chosen and
shortest paths from each landmark to the rest of the graph are computed. Next, these landmarks are mapped to the n-dimensional space where errors between virtual and measured distances (i.e. graph coordinates and number of hops) are minimized using a non-linear optimization algorithm such as Simplex Downhill [71]. Once the landmark coordinates are fixed in the n-dimensional space, they are used to calibrate coordinate values for the rest of the vertices in the graph. Given a vertex \( v \) and \( k \) landmarks \( \mathcal{L} = \{l_1, \ldots, l_k\} \) whose coordinates are fixed, the goal is to minimize the error between the Euclidean distance from \( v \) to \( \mathcal{L} \) based on the measured shortest path lengths \( \{d(v, \ell_1), \ldots, d(v, \ell_k)\} \) for all \( v \in V \).

Although this method is suitable for fast approximation of pairwise distances, it has two disadvantages. Graph coordinate systems are reported to be error sensitive as pairwise distances in large networks with small diameter are all integers (hop counts), and usually clustered across a small number of possible values with small variance. As a result, estimation errors in a graph coordinate system can be quickly rounded up, which makes the optimization process more challenging as the graph gets bigger. The proposed system is only evaluated on graphs with 200K to 400K vertices. Second and more importantly, once the vertices are mapped to a virtual coordinate space, all path information is lost. The graph coordinate system can quickly approximate pairwise distances, however, further information regarding how to travel from one vertex to another cannot be determined. In applications of graph mining such as betweenness centrality, obtaining paths is a key requirement.

### 2.2.3 Path Concatenation Methods

It is often critical to find the actual shortest paths between arbitrary pairs of vertices in large graphs. For example, computing the route between two locations in a large road network requires finding one or more shortest paths. In bioinformatics, finding the shortest path forms a basis for predicting the secondary structure of an RNA molecule from its sequence [49].

For relatively small networks where approximation is not necessary, exact algorithms
used to compute the length of the shortest path can easily be extended to include the path itself in the result. In the SSSP problem, Dijkstra’s algorithm can be modified so that each vertex contains a pointer to its predecessor, and the shortest path from a vertex to the source can be traced simply by following the pointers. The Floyd-Warshall algorithm can similarly be modified to find all pairs shortest paths. For large networks however, while a number of methods have been proposed to approximate the distance between pairs of vertices, a path sequence is often disregarded.

To address this issue, Gubichev et al. [44] suggest the Sketch algorithm which extends the landmark-based methods via path concatenation. In addition to distance, the actual shortest paths \( \pi^*(\ell, v) \) \( \forall \ell \in \mathcal{L}, v \in V \) are stored as part of the pre-computation data.

Consider the simplest case with a single landmark \( \ell \) in a connected graph \( G \). Given two vertices \( s \) and \( t \), there exists at least two shortest paths from \( \ell \), namely \( \pi^*(\ell, s) \) and \( \pi^*(\ell, t) \). Using these, a path from \( s \) to \( t \) can be constructed as follows:

\[
\pi(s, t) = \pi^*(s, \ell) \cdot \pi^*(\ell, t)
\]  

(2.9)

Since there are multiple landmarks in practice, the shortest path obtained by using any of the landmarks is returned as the final approximation result.

\[
\pi'(s, t) = \min_{\ell_i \in \mathcal{L}} \{ \pi^*(s, \ell_i) \cdot \pi^*(\ell_i, t) \}
\]  

(2.10)

Note that \( \pi^*(s, \ell) \) is obtained by reversing \( \pi^*(\ell, s) \) if \( G \) is undirected. Otherwise, the pre-computation step is also run on \( G' \), the graph built by reversing all edges of \( G \).

Figures 2.2 and 2.3 show the operation of the Sketch algorithm on a sample graph with eight nodes. There is a single landmark \( \ell = 0 \) and shortest paths from vertices 6 and 7 are highlighted with green and yellow arrows in Figure 2.2b. Observe that a new path from 6 to
7 is constructed passing through the landmark. In this case, the approximate shortest path \( \pi'(6, 7) \) has length six although it is possible to improve the initially estimated path.

Two optimization techniques are described for improvement.

- A naïve cycle elimination algorithm removes all cycles in a given path of length \( l \) in \( O(l^2) \) time to reduce the length of the concatenated path. For example, consider an initially approximated path \( \pi'(s, t) = (s, u_1, u_2, \ell_j, u_3, u_1, t) \) of length six. This path contains a cycle of four hops, \((u_1, u_2, \ell_j, u_3)\), and it can be removed to optimize the length of the shortest path. In a graph with diameter \( D \), the worst-case time complexity of the suggested cycle elimination algorithms is \( O(D^2) \).

- The short-cutting optimization progressively examines the out-going edges of each vertex in the first half of the path and checks if there is a direct connection to any vertex on the other half. For example, consider an initially approximated path from \( s \) to \( t \) through landmark \( \ell_j \): \( \pi'(s, t) = (s, u_1, u_2, \ell_j, u_3, u_4, t) \). The short-cutting method first expands the out-going edges of \( s \) and checks if any of the vertices succeeding \( s \) occurs in the second half of the path which includes vertices from \( \ell_j \) to \( t \). If there exists
such a neighboring vertex, say $u_3$, then the path can be optimized by substituting the edge $(s, u_3)$ with the longer path $(s, u_1, u_2, \ell_j, u_3)$. This neighbor expansion process is performed until all vertices up to $\ell_j$ are examined for short-cutting edges and the shortest path that can be obtained is returned as the final result.

Given $k$ landmarks for the pre-computation step, the worst-case space complexity of the Sketch algorithm is $O(nkD)$ since the shortest path between a landmark and arbitrary vertex cannot be longer than $D$.

Although this algorithm can approximate the shortest paths with low error, it is not trivial to implement in parallel and the serial implementation does not scale. In the original paper, the largest graph evaluated experimentally contains only 3M vertices and it takes about 11 hours to perform all pre-computations on a multi-core server with 48 GB of memory. In addition, for the 3M data set, the space required to store the pre-computed results is reported to be 7.4 GB and this data is stored in memory to provide fast shortest path approximations. According to the experiment results, the Sketch algorithm can take up to 2.5 seconds to respond to a single distance estimation query. The performance of the system under heavy load with multiple queries arriving at the same time is not studied.

When working with large datasets such as social networks with billions of vertices and edges, storing the entire pre-computation data in memory may be infeasible due to lack of space. The short-cutting algorithm further requires random access to the edge set of the graph. A parallel all pairs shortest path implementation would involve multiple threads or processes working simultaneously. Storing all available data in the memory of a central server for random access would introduce a severe performance bottleneck in such an application scenario.

Figure 2.3 demonstrates the short-cutting optimization on the initially estimated path of the sample graph in the previous figure. This is achieved by expanding the neighbors of each vertex on the estimated path to find a short-cutting edge. When neighbors of 4 are expanded, the edge $(4,5)$ is observed to improve the approximation quality. Final result after
removing the extra edges from the initial path are illustrated in Figure 2.3b. Note that the short-cutting method requires random access to the entire graph structure and is not feasible for arbitrarily large graphs.

Tretyakov et al. [85] describe the Landmark-BFS algorithm which addresses the storage requirement problem using a basic path compression method. They suggest keeping a single parent pointer for each vertex and landmark pair \((\ell, v)\), rather than storing the entire shortest path sequence between \(\ell\) and \(v\) as a list. By tracing the parent pointers for a given pair, they re-construct the shortest path when needed. This approach reduces the total memory requirement from \(O(nDk)\) to \(O(nk)\) since paths between vertices and landmarks are no longer stored.

Tretyakov suggests two algorithms to improve upon the existing path concatenation and short-cutting methods.

- **Distance-LCA**

  Given two arbitrary vertices \(s, t\) and a single landmark \(\ell\), consider the shortest path
Figure 2.4: Finding the lowest common ancestor on a shortest path tree. Notice that this is equivalent to detecting and removing the cycle \([0, 1, 3, 2]\) from the initially estimated path.

Tree where \(\ell\) is the root and \(s\) and \(t\) are the leaves. The \textbf{Distance-LCA} algorithm finds the lowest common ancestor (LCA) of \(s\) and \(t\) in this tree, which can be done in linear time with the path length. Note that in practice, this method is equivalent to \textbf{path concatenation} and \textbf{cycle elimination} methods combined in the \textbf{Sketch} algorithm, but reduces the worst-case complexity of approximating the shortest path for a given pair of vertices using a single landmark from \(O(D^2)\) to \(O(D)\). \textbf{Figure 2.4} illustrates the lowest common ancestor method on a sample graph.

- **Landmark-BFS**

The \textbf{Landmark-BFS} algorithm is designed to improve the accuracy. Given a set of landmarks \(L = \{\ell_1, ..., \ell_k\}\), this method also pre-computes the shortest paths from each landmark to the rest of the graph. However, when approximating the distance between two arbitrary vertices \(s\) and \(t\), an induced sub-graph is constructed by taking the union of all pre-computed shortest paths between \(s, t\) and \(L\). Then a BFS search (or Dijkstra's Algorithm if \(G\) is weighted) is invoked on that sub-graph to find the
shortest possible path.

More formally, given all shortest paths between $s, t \in V$ and $\forall \ell_i \in \mathcal{L}$, an induced sub-graph $G_{\mathcal{L}}(s, t)$ is constructed by taking a union of all vertices and edges as follows:

$$G_{\mathcal{L}}(s, t) = \bigcup_{\ell_i \in \mathcal{L}} \pi^*(s, \ell_i) \cdot \pi^*(\ell_i, t) \quad (2.11)$$

An approximate shortest path $\pi'(s, t)$ is obtained by running BFS (or Dijkstra’s Algorithm if $G$ is weighted) on $G_{\mathcal{L}}(s, t)$. The short-cutting technique used in the Sketch algorithm is also adapted by Landmark-BFS to improve the initial estimate.

Figure 2.5 illustrates the operation of Landmark-BFS on a sample graph with two landmarks and eight vertices. Solid arrows show shortest paths from $\ell_1 = 0$ whereas dashed arrows correspond to shortest paths from $\ell_2 = 5$. Consider the problem of finding a shortest path estimate between vertices 3 and 8. Using the edges from both landmarks, an approximate path $\pi'(3, 8)$ of length four can be obtained.

For a given pair of vertices, the size of the induced graph is $O(kD)$ and the estimation algorithm runs in at most $O(k^2D^2)$ time. Observe that the main advantage of Landmark-BFS over the Sketch algorithm is that it leverages more data from multiple landmarks at the same time and this yields results with higher accuracy because an edge in one shortest path tree may act as a short-cut in another.

The largest graph used to evaluate the Landmark-BFS algorithm is a Skype network with 454M vertices and 3B edges. Landmark data is stored in memory using a server with 32 x 2.2 Ghz Quadcore CPU’s and 256 GB of RAM. The average time to compute a shortest path between two arbitrary vertices using 100 landmarks is reported as 16 ms. In a denser Twitter network with 40M vertices and 1.2B edges, however, the average estimation time increases to 889 ms for a single pair using 100 landmarks. The
Figure 2.5: **Landmark-BFS** with vertices 0 and 5 selected as the landmarks. The approximate path $\pi'(3, 8)$ has length four. This figure is redrawn based on an illustration from the original paper.

The authors explain this slow down by the presence of vertices with very high degree in this network. The **short-cutting** optimization introduces a considerable overhead in dense graphs with high average degree per vertex since each neighbor expansion query requests all outgoing edges of a vertex.

### 2.2.4 Limitations of Path Concatenation Methods

In summary, existing path concatenation methods mainly focus on fast approximation of the shortest path between an arbitrary pair of vertices and assume pre-computation data is small enough to fit in memory. The shared memory central server approach has inherent scalability limitations. The **short-cutting** optimization makes a significant contribution to
accuracy. However, the main drawback of this technique is its requirement to randomly access the neighbors of any vertex in the graph. The Sketch algorithm is only studied on graphs with a few million vertices and implemented using high-end servers. It has a large storage complexity which cannot scale to billion-sized graphs. Although the Landmark-BFS algorithm reduces the storage complexity and provides faster estimates of the actual shortest paths, it is still not suitable for computing all pairs shortest paths (APSP) on an entire, or even a smaller subset of a large graph due to the same assumption. Using a larger server with hundreds of cores and Terabytes of memory is not cost effective. Also, the performance of these methods under heavy load with many distance estimation requests arriving in parallel has not been evaluated. Recent studies show that applications running on multi-core architectures tend to be limited by off-chip bandwidth due to shared resources in the memory hierarchy [25, 63]. Consequently, approximating shortest paths in billion-scale graphs remains a computationally expensive problem that requires careful parallel design.

Another shortcoming of the existing landmark-based methods is the limited information obtained during the pre-computation steps. Given a landmark $\ell$ and a vertex $v$, both algorithms compute a single shortest path $\pi(\ell, v)$ and either store it in memory or re-construct it using parent pointers when needed. However, especially in large graphs with high average degree per vertex, there is often more than one shortest path between a landmark and a vertex. The choice of the shortest path can be critical for approximation quality. In chapter 3, we present an extension to the Landmark-BFS algorithm that leverages more pre-computation data while eliminating the need to store everything in main memory. This allows massive parallelism as multiple compute nodes can process different subset of vertices independently without relying on a central server.

## 2.3 Graph Centrality: Definitions and Approximations

Although several different definitions exist for centrality, the common goal is to measure the structural importance and prominence of a vertex based on its connectivity inside the
network [10]. It is often the case that there is a strong correlation between the centrality of a vertex and how easily it can be reached from anywhere in the network or how much traffic flows through it when reaching other vertices. This section will provide the definitions and background in three widely known centrality measures and discuss their applicability to large graphs.

For the rest of the discussion, given a graph $G$, $A$ represents its adjacency matrix where entry $a_{ij}$ is 1 if there is an edge from $v_i$ to $v_j$, or 0 otherwise. Weighted graphs can be represented similarly by the edge weight matrix $W$, where $w_{ij}$ equals to the weight of the edge from $v_i$ to $v_j$, or 0 if they are not connected. Note that for unweighted graphs, $A$ and $W$ are the same.

2.3.1 Degree Based Approaches

The most basic and well known centrality measure is degree centrality, which measures the number of out-going edges of a given vertex. Formally, the degree centrality of a vertex $v_i$ is given by

$$c(v_i) = \sum_{v_j \in V} a_{ij}$$ (2.12)

Another definition for degree centrality is the number of vertices that can be reached from a given vertex in one hop. Sade [78] gives a broader definition that extends to hops of length $k$, called $k$-path centrality which counts all paths of length $k$ or less starting from a vertex. Borgatti et. al [10] suggest several variations of this metric by choosing different restrictions on the structure of the paths. One of them is the geodesic $k$-path centrality which only considers shortest paths of length $k$ or less that originate from a given vertex. This definition can be made even stricter by counting only the edge disjoint paths. Two paths
are \textbf{edge disjoint} if and only if they share no common edges. Ford and Fulkerson \cite{33} state that the number of \textbf{edge disjoint} paths between two vertices equals to the minimum number of edges that must be removed to make them disconnected, and thus can be seen as a measurement for network vulnerability. Intuitively, if the average \textbf{edge disjoint} \textit{k-path centrality} is small, the network communication or information diffusion in that graph is considered to be more vulnerable to edge removals.

A similar construct exists for \textbf{vertex disjoint} paths which counts the number of \textbf{vertex disjoint} paths of length \(k\) or less between two vertices. As the name suggests, \textbf{vertex disjoint} paths do not share any vertices except for the two ends of the path. Menger \cite{70} shows that the number of \textbf{vertex disjoint paths} linking two vertices in a graph equals to the minimum number of vertices that must be deleted to disconnect them.

\section{2.3.2 Closeness Based Approaches}

Degree based centrality measures only consider the number of vertices that can be reached from a given vertex, without considering the path length. More detailed measures of centrality consider the lengths of the walks from a given vertex to the rest of the graph, taking edge weights into account. Freeman \cite{36} introduces closeness centrality as the total geodesic distance from a vertex to the rest of the graph. This is formally given as

\[ c(v) = \sum_{u \in V} d(u, v) \quad (2.13) \]

where \(d(u, v)\) is a single entry in the distance matrix \(D\), representing the geodesic distance from \(u\) to \(v\). Observe that closeness centrality is an inverse measure of centrality as larger values indicate a less central vertex. Beauchamp \cite{6} modified the above definition to fix this naming inconsistency by taking the reciprocal of the above and multiplying it by the number of vertices minus one such that
\[ c(v) = \frac{|V| - 1}{\sum_{u \in V} d(u, v)} \quad (2.14) \]

It should also be noted that closeness centrality generally applies to connected graphs. In a disconnected setting, closeness centrality should be considered separately for each connected component as otherwise it would either be \( \infty \) (2.13) or undefined (2.14). Some studies on closeness centrality suggest considering multiple paths between a pair of vertices \( u \) and \( v \), rather than just the shortest path. Friedkin [39] introduces the concept of immediate effects centrality, defined by the reciprocal of the average distance between two vertices which considers all paths. However, calculating such a metric is computationally more expensive as it requires finding all possible paths between all pairs in a graph.

2.3.3 Flow Based Approaches

Degree and closeness based centrality measures only consider the cases where a walk starts or ends at a vertex, but they do not take intermediate vertices that are part of these walks into consideration. For example, a vertex with a low degree and closeness centrality can occur on a relatively critical path of the network, acting as a bridge between several pairs. In such a setting, that vertex should be considered effective for information diffusion in the network, and thus, have high centrality.

Freeman describes the betweenness centrality as the number of times a vertex occurs in any shortest path between any pair of vertices in the graph [35]. A more formal expression can be given by defining \( \sigma_{st} \) as the number of geodesic paths from \( s \) to \( t \), and \( \sigma_{svt} \) as the number of such paths that \( v \) lies on. The betweenness centrality of \( v \) is then defined as

\[ b(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{svt}}{\sigma_{st}} \quad (2.15) \]
Borgatti [9] also suggests a variation of betweenness centrality which considers walks of any length between a pair of vertices, rather than just the geodesic paths. However, since computing that metric is expensive, they only present results based on simulations.

Another variation is $k$ betweenness centrality, which considers all paths of length at most $k$ hops. The intuition behind this approach is the assumption that long paths are not frequently used and should be discarded when computing the betweenness of a vertex. Friedkin [39] studied a specific version of $k$ betweenness centrality for $k = 2$. It is also possible to restrict the paths considered based on certain criteria. For example, Freeman [38] also defines flow betweenness centrality which only considers edge disjoint paths between pairs of vertices.

2.3.4 Centrality in Large Graphs

The complexity of computing a centrality metric depends on the definition of the measure, and can be quiet simple or prohibitively expensive for large graphs. Computing degree based measures are relatively easier since they do not require deeper link analysis of the graph. For example, degree centrality is simply equal to the number of edges adjacent to a vertex. However, more elaborate measures regarding closeness and flow based approaches necessitate running more complex graph algorithms, finding shortest paths and summing node or edge occurrences which may be time consuming for graphs even with a few million edges.

Closeness

Exact computation of Freeman’s closeness centrality [36] requires running an all pairs shortest path algorithm which has $O(n^3)$ run time and $O(n^2)$ storage complexity. For billion scale networks, this is practically impossible and approximation is unavoidable.

Eppstein and Wang [29] give an approximation algorithm for closeness centrality in weighted graphs, which runs in $O\left(\frac{\log n}{\epsilon^2}(n \log n + m)\right)$ with an additive error of at most $\epsilon\Delta$ where $\epsilon$ is a constant and $\Delta$ is the diameter of the graph. They suggest estimating the
closeness centrality by picking \( k \) samples and computing the shortest paths from each sample to the rest of the graph. The centrality of a particular vertex \( u \) is then estimated by

\[
c(u) = \frac{k}{n \sum_{i=1}^{k} d(v_i, u)}
\]

(2.16)

Bader and Madduri [4] study a parallel version of the Eppstein-Wang algorithm on high-end shared memory symmetric multiprocessor and multi-threaded architectures. In particular, they use a 40-processor Cray MTA-2 [14] with 160 GB shared memory, performing experiments with networks of size up to three million nodes. However, their implementations assume random memory access. Larger networks potentially having hundreds of millions of nodes can reduce the memory subsystem performance, as sparse graph analysis tends to have low degrees temporal and spatial locality in their memory access patterns and exhibit large memory footprint [66].

Kang et al. [52] define a new metric called effective closeness centrality to estimate closeness centrality in unweighted graphs with MapReduce. Their method estimates the number of unique vertices that can be reached from a given vertex within \( r \) hops using a probabilistic bit vector that can represent \( n \) unique nodes with \( O(\log n) \) space complexity [31]. Given a graph with diameter \( D \), they estimate closeness centrality in \( D \) iterations where each iteration is a separate MapReduce job. In their experimental evaluations, Kang et al. observe that nodes with higher degrees have higher effective closeness centrality values indicating they can reach other nodes in the graph within a small number of hops. In contrast, nodes with low degrees have varying effective closeness centrality values suggesting this metric could be an indicator for low degree nodes in large networks. This approximation algorithm is also important because it is the first centrality approximation method in large-scale that leverages MapReduce as the underlying platform and shows promising scalability figures.
On the other hand, Kang’s effective closeness centrality metric does not generalize to weighted graphs due to the limitations of the probabilistic bit vector they use. Weighted interpretations are quite common in social networks and can provide deeper information compared to an unweighted setting. For example in Facebook, the number of messages exchanged between two parties can be used to represent the importance of their relationship and transferred as an edge weight to the underlying social graph. A similar observation holds for the number of papers co-authored between pairs of authors in the DBLP network.

**Betweenness**

Exact computation of Freeman’s betweenness centrality [37] is even harder than closeness because it requires computing all shortest paths between all pairs of vertices in the graph. Brandes [11] presents a method that can compute the exact betweenness centrality of a graph with $n$ nodes and $m$ edges in $O(nm)$ and $O(nm + n^2 \log n)$ time on unweighted and weighted graphs respectively.

Bader, Madduri and Ediger et al. [4, 66, 27] present the first parallel implementations of betweenness centrality designed for the massively multi-threaded Cray systems [14, 59]. Their algorithm approximates betweenness centrality by computing the shortest paths from $k$ sample vertices to the rest of the graph, and accumulating the number vertex occurrences in these shortest paths by a parallel lock-free implementation of Brandes’ [11] algorithm.

Ediger et al.’s work [27] provides a good example of approximating betweenness centrality by sampling a small subset of the graph and utilizing parallel processing although their approach has some drawbacks in terms of accuracy and scalability. When comparing the exact betweenness centrality results with approximations in small graphs, they suggest using $10–25\%$ of the total number of vertices in the graph as the sample size for accurate estimations. However, storing everything in memory does not allow sampling the recommended number of vertices in larger networks due to increased storage cost. In their largest experiments with 1 TB shared memory, they approximate betweenness centrality in a graph
with 61M vertices using 256 samples and report that memory capacity can be a bounding factor. Madduri et al. also discuss that the scalability of this approach on larger systems is questionable and may necessitate structural changes in the algorithms and data structures used [66].

Kang introduces a new flow based centrality measure called LineRank and implements it using MapReduce [52]. The main idea of LineRank is to compute the importance of a vertex by summing up the importance score of its edges. The input graph $G$ is transferred to a line graph $L(G)$, where edges in $G$ are converted to vertices in $L(G)$, and new edges in $L(G)$ are defined by a transitive relation between the vertices of $G$. The objective is to compute the stationary distribution of a random walk that starts in an arbitrary vertex in $L(G)$, and moves to another vertex with probability $p$ or stays at the same vertex with probability $(1 - p)$, which is noticeably similar to the PageRank algorithm [12]. Based on experiments with real world graphs having up to 24M vertices, Kang reports that vertices with high degree have varying LineRank scores and can be distinguished easily via this measure.

Although being a flow based measure that can be computed in parallel, LineRank is not a direct replacement for betweenness centrality. It lacks the intuitive notion of contribution to shortest paths and simply considers the whole flow in the graph to assign a probabilistic score to each edge. How the LineRank of a vertex compares to its betweenness centrality and how these two metrics relate to each other are not studied in the paper.

In summary, computing the closeness and betweenness centrality metrics in large graphs is a challenging problem. Parallel implementations in massively multi-threaded shared-memory systems are not scalable to large graphs due to lack of storage and limitations with the memory bus. Furthermore, solutions that use special and expensive hardware are not widely available. Some of the alternative centrality measures that can be computed on cheap hardware platforms do not generalize to weighted graphs and they are not direct replacements to the closeness and betweenness centrality metrics. Also, the scalability of these algorithms have only been tested on artificial graphs. In chapter 4, we describe parallel approximation
algorithms for both metrics and study their accuracy and scalability on real-world graphs that are up to 10 times bigger using only 20 compute nodes.

2.3.5 Large Scale Graph Processing

Several frameworks have been proposed to solve computing problems that concern large graphs. Single-computer graph libraries such as BGL [80], Stanford GraphBase [58] and NetworkX [45] are limited in their ability to handle large graphs due to both storage and computation requirements. The CGMgraph library [16] is built on MPI and provides users a set of functions for graph algorithms. In particular, it contains methods for computing an Euler Tour of a forest, detecting whether a given graph is bipartite, and finding the connected components of a graph. Parallel BGL [43] is another framework built on MPI that implements a set of graph algorithms including parallel versions of Dijkstra’s shortest path and betweenness centrality. However, Parallel BGL follows a distributed or shared memory approach where the entire graph structure is assumed to fit in memory. In addition, it does not address fault tolerance which is common in commodity clusters and cloud computing environments. Both CGMgraph and Parallel BGL are procedural object-oriented programming libraries written in C++.

Pregel [68] is a functional programming based computational model offered by Google. It aims to solve large-scale graph problems in a commodity cluster environment and addresses fault tolerance automatically via checkpointing. Pregel is designed after the "think like a vertex" idea and the basic unit of computation is called a superstep. In superstep $S$, a vertex receives a set of messages from its incoming edges that were produced in superstep $S - 1$, performs a local computation, and produces a set of messages for its neighbors to be received in superstep $S + 1$. The graph data including the vertices, edges and the computation state is stored in distributed memory by the help of a deterministic partitioning function. The vertex-centric design makes the programming model similar to MapReduce since the fundamental idea is to partition the graph, process each vertex independently, and send/receive messages.
between supersteps. Unfortunately, there is currently no publicly available implementation of Pregel, although the graph algorithms that are presented in this dissertation are all based on MapReduce with a vertex centric approach. More specifically, each record is a \(<\text{key}, \text{value}>\) pair where the key is a vertex id and value is a message destined at the corresponding vertex. The reducers group all messages targeting a given vertex and process them in batches. Thus, all graph algorithms described in this dissertation can easily be adapted to Pregel’s vertex-centric programming model if such an implementation becomes available for researchers in the future.

MapReduce proved useful in handling large data and performing complex data analytics in cluster environments although it has been criticized for yielding suboptimal performance for iterative algorithms, forcing the users make overly simplifying assumptions [64]. This problem mainly originates from the strict functional programming model that enforces the result of the reduce output go into the file system. In an iterative setting, several map and reduce functions are invoked repeatedly, and a mapper has to read the output of the previous reduce from the file system, which decreases the overall application performance. There is also no notion of a computation state. Everything between two iterations must be serialized to disk and each iteration is regarded as a new program invocation by the system. These restrictions on the programming model have been addressed by specifically designed MapReduce frameworks that allow long running tasks, program state that can be shared between iterations, and include support for prioritized iterations [28, 88, 13]. Such optimized frameworks showed that MapReduce can be flexible for iterative computations and the amount of data shuffled between successive iterations can be reduced for noticeable performance gains. In the area of graph mining, MapReduce has been used for detecting connected components, enumerating triangles and rectangles, diameter estimation and finding sub-graphs of high connectivity [53, 18]. These studies suggest that graph mining frameworks built on MapReduce scale well on cheap hardware. MapReduce implementations of existing algorithms may also exhibit suboptimal performance due to poor design which may result in excess amounts
of data being shipped from mappers to reducers at each iteration over the input dataset. Developers often assume the parallel framework automatically handles all data management issues efficiently and do not focus on optimizing the algorithms. Poor MapReduce adaptation of existing algorithms often generate duplicate records and distribute them among the cluster to create an embarrassingly parallel structure where compute nodes can work independently on different parts of the input. However, data inflation may outrun the gains obtained by parallelism and introduce a scalability bottleneck for the algorithm. In chapters 3 and 4 of this dissertation, we give examples to such naïve parallel implementations and describe novel optimization techniques to address these challenges.

2.3.6 Summary

Existing graph algorithms for distance estimation and computing centrality are either single-computer implementations or they target distributed shared-memory platforms that essentially assume the input graph can fit in memory.

Previous distance estimation algorithms mainly focus on fast approximation of pairwise distances using pre-computation data. These algorithms keep the pre-computation results in the memory of a central high-end server which is limited in scalability, especially when there is demand for a large number of pairwise distance estimation queries arriving in parallel. It is possible to increase the approximation quality using more pre-computation data and eliminate the memory requirements using batch processing methods and efficient data partitioning techniques. Such parallel distance estimation algorithms can allow fast computation of pairwise distances within a subset of a graph in parallel, which is not possible with the algorithms that are currently available.

Computing node centrality concentrates on measuring the importance of a node in a graph. Two common centrality measures are closeness and betweenness centrality. Exact computations of these metrics are prohibitively expensive in large graphs. They require solving the all pairs shortest path problem, or even more complicated tasks such as counting
all walks of a given length. Approximation algorithms for centrality fall in one of the two
categories: The first class of algorithms assume the graph is small enough to fit in the
memory of a single node or use expensive shared memory hardware that is not widely and
cheaply available. The second class of algorithms introduce new centrality metrics that
can be computed in parallel on commodity hardware. However, the suggested metrics are
not direct replacements to closeness and betweenness centrality and some of them do not
generalize to weighted graphs.

There has been some research effort on mining large graphs using MapReduce that
presents satisfactory performance and scalability results. Although MapReduce enforces
a functional programming style and can be inefficient due to file system overheads and the
lack of a program state, these issues have been addressed by recent studies that describe ef-
ficient MapReduce frameworks for iterative applications. It is possible to further optimize a
MapReduce algorithm by careful design. In general, minimizing the amount of intermediate
data shuffled over the network from mappers to reducers can yield noticeable performance
gains.
CHAPTER 3
METHODS FOR EXACT AND APPROXIMATE DISTANCE COMPUTATION IN LARGE GRAPHS

3.1 Overview

The next two chapters contain detailed descriptions of the graph algorithms and their distributed communication cost analysis. Figure 3.1 shows a roadmap of the methods presented and the relationships between them. We start with an exact algorithm to compute the single source shortest paths in large graphs. Using this as the basis, we describe a new algorithm to approximate the distance between arbitrary pairs of vertices. In chapter 4, we explain how to parallelize the distance estimation algorithm on shared-nothing architectures. Finally, we present efficient techniques to approximate closeness and betweenness centrality in large graphs, which use the parallel distance estimation algorithm as the main building block.

Figure 3.1: Roadmap of the large scale graph algorithms.
3.2 Datasets

The characteristics of the datasets used for experimental evaluation of the methods described in this dissertation are summarized in Table 3.1. All graphs are considered as undirected to increase the data size and work on one strongly connected component, which otherwise would not be possible.

<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>4K</td>
<td>176K</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>7K</td>
<td>214K</td>
</tr>
<tr>
<td>Enron</td>
<td>36K</td>
<td>423K</td>
</tr>
<tr>
<td>Twitter</td>
<td>41M</td>
<td>2.4B</td>
</tr>
<tr>
<td>WWW-2002</td>
<td>720M</td>
<td>12.8B</td>
</tr>
</tbody>
</table>

The first three graphs are medium-sized networks from the Stanford Large Network Dataset Collection [60] and used to compare the true and estimated values for distance and centrality. We intentionally chose medium sized networks to be able to compute the true distance and centrality values by running the NetworkX Python library [46] on a single desktop computer.

- **Facebook**: Contains friendship relations between network members.

- **Wikipedia**: Voting network between contributors. There is an edge from user \(i\) to \(j\) if one of them voted on the other.

- **Enron**: A graph composed of individuals who exchanged at least one email within a corporate network.

The last two datasets are used for large-scale performance and scalability experiments.

- **Twitter**: A subset of the social network from a snapshot in 2008 with over 41 million nodes. There is an edge from user \(i\) to \(j\) if one of them follows the other.
- **WWW**: Contains billions of hyperlinks from a web crawl in 2002 by the Yahoo! Altavista search engine.

Figure 3.2 shows the long-tail degree distribution of the WWW and Twitter datasets plotted on logarithmic scale. Notice that as the degree becomes larger, the frequency gets smaller following a power law distribution. In section 3.3, we present three novel optimization techniques that exploit these characteristics to speed up the computations.

![Power Law Degree Distribution](a) WWW Degree Distribution

![Power Law Degree Distribution](b) Twitter Degree Distribution

Figure 3.2: Power law degree distribution of the large graphs used in the experiments
3.3 Parallel-SSSP

We begin with an efficient MapReduce algorithm for solving the single source shortest path problem, which provides the basis for the landmark-based distance estimation algorithms. Given \( \ell \) as the source (landmark), the traditional definition of this problem asks for finding the shortest path \( \pi^*(\ell, v) \) for all \( v \in V \). Although there are often multiple shortest paths between a pair of vertices, practical applications typically return the first one as the answer and neglect the rest. We slightly modify the problem definition to discover \( \Pi^*(\ell, v) \), a set of shortest paths for each \((\ell, v)\) pair. More formally

\[
\Pi^*(\ell, v) = \{ \pi(\ell, v) : |\pi(\ell, v)| = d(\ell, v) \} \quad (3.1)
\]

Instead of a shortest path tree rooted at the landmark, we are interested in generating a shortest path sub-graph where there are many ways to reach a vertex. Note that the output of this computation is larger, but it is stored on the distributed file system (DFS) rather than memory. A sample graph is given in Figure 3.3.

![Sample graph](image)

**Figure 3.3**: a) Sample undirected graph with no edge weights and nine vertices. b) The sub-graph for vertex 7 which contains the output of SSSP for the pair (0,7). Three shortest paths are highlighted with arrows: \( \{0,1,3,7\} \), \( \{0,1,4,7\} \) and \( \{0,2,5,7\} \).
Representing Vertices

For consistency, the same binary graph file format is used for all algorithms described in this dissertation. Figure 3.4 shows the logical representation of a vertex record. Each field is associated with a name and type. The neighbors and sub-graph fields can be arbitrarily long and they are compressed using delta encoding when serialized on disk.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>long</td>
</tr>
<tr>
<td>source</td>
<td>long</td>
</tr>
<tr>
<td>degree</td>
<td>int</td>
</tr>
<tr>
<td>distance</td>
<td>float</td>
</tr>
<tr>
<td>active</td>
<td>bool</td>
</tr>
<tr>
<td>[neighbors]</td>
<td>list</td>
</tr>
<tr>
<td>[sub-graph]</td>
<td>list[list]</td>
</tr>
</tbody>
</table>

Figure 3.4: Logical representation of a vertex record.

In the beginning, the distance and active fields of $\ell$ are set to 0 and true respectively. All other vertices have their distance fields equal to $\infty$, and active fields set to false. When the algorithm terminates, the sub-graph field of a vertex $v$ contains one or more shortest paths from $\ell$ to $v$ combined in a sub-graph as shown in Figure 3.3b, and the distance field is set to $d(\ell, v)$.

All vertex id’s are represented by variable length long integers to reduce data size. In addition, delta encoding [21] is used to compress neighbor lists. For each vertex, the neighbors are sorted in ascending order and only the difference between each consecutive pair is stored. To re-construct, a sequential addition operation is applied from the first to the last element of the compressed list. Note that the differences are stored using variable length integers so smaller values occupy less physical space. Sample encoded and decoded representations of a neighbor list are given in Figure 3.5.

The same compression scheme is also applied to the sub-graph field. Each sub-graph is converted to the adjacency list representation and for each vertex, the neighbors attached to it are compressed separately using delta encoding.
Algorithm Overview

The Parallel-SSSP algorithm is a breadth-first search. The idea is to propagate the minimum distance from the landmark to the rest of the graph. Let $d(v, \ell)^{(k)}$ be the distance between $v$ and $\ell$ at iteration $k$. Initially, $d(\ell, \ell)^{(0)} = 0$ and all other distances are set to $\infty$. In an unweighted graph, $d(v, \ell)^{(k)}$ can be computed iteratively as follows:

$$d(v, \ell)^{(k)} = \min \left\{ d(v, \ell)^{(k-1)}, \min_{u \in \mathcal{N}(v)} \left\{ d(u, \ell)^{(k-1)} \right\} + 1 \right\}$$ (3.2)

where $\mathcal{N}(v)$ contains the incoming neighbors of $v$. This computation can be expressed as a sequence of MapReduce jobs using a vertex-centric approach.

- **Initial Conditions**
  $$d(\ell, \ell) = 0 \text{ and } \ell \text{ is marked as active.}$$
  $$\forall v \in V \land v \neq \ell, d(v, \ell) = \infty \text{ and } v \text{ is marked as inactive.}$$

- **MAP**
  
  Each active vertex adds 1 to its current distance, sends it to its out-neighbors and becomes inactive.

- **REDUCE**
  
  Each vertex iterates over the distance messages received from its in-neighbors and finds the minimum. If the minimum is smaller than the current distance, the current
distance is updated and the vertex is marked as active so that it can propagate the new minimum to its out-neighbors when the next iteration starts.

This cycle continues until all vertices become inactive, which indicates that the algorithm converged. In a connected graph, all vertices contain a distance value that is less than $\infty$ after convergence.

The iteration steps of this algorithm for the sample graph above is shown in Figures 3.6 and 3.7. Active vertices are colored red while inactive vertices are colored grey. Once a vertex switches from active to inactive, it becomes black. Observe that initially, only the landmark ($\ell = 0$) is active and its distance $d(\ell, \ell)$ is set to 0. After sending the distance and path information to its out-neighbors, it switches to inactive state and becomes black. In the next iteration, vertices 1 and 2 become active and repeat the same steps. At the end of the second iteration, their distances $d(0, 1)$ and $d(0, 2)$ are both set to 1 and the paths fields contain the corresponding paths, [0, 1] and [0, 2]. The algorithm converges after four iterations when all vertices are discovered. It is worth noting that vertices 7 and 8 contain multiple paths from the landmark. That is, there is more than one way to reach these vertices from $\ell = 0$. Vertex 7 receives three different paths - ([0, 1, 3], [0, 2, 4], [0, 5, 7]) - from its parents. Similarly, vertex 8 receives three different paths - ([0, 1, 4], [0, 2, 5], [0, 2, 6]) - from its parents. In the actual implementation, these paths are merged into an induced sub-graph as shown in Figure 3.3b and further compressed using the delta encoding technique.

Note that for the sample graph, there are at most three shortest paths between a (landmark, vertex) pair and all of them are stored as part of the output for demonstration purposes. However in large graphs, there may be too many shortest paths between a given landmark and vertex. Storing all shortest paths requires excessive amounts of disk space and should be avoided for performance issues. In section 3.3.2, we describe path sampling algorithms to address this problem and analyze their storage and performance costs in detail.
The pseudo-code for the MapReduce implementation of Parallel-SSSP is given in Algorithm 1.

MAP

The MAP function sends two different types of output to the reducers. Inactive vertices only send the vertex record (lines 14-15). Messages sent from an active vertex $v$ to its out-neighbors contain the shortest paths along with the distance (lines 4-5). After an active vertex sends its distance and paths, it is marked as inactive and written to the final output location on DFS, rather than being sent to reducers (line 12). This is a special IO optimization for unweighted graphs. In an unweighted setting, all shortest paths from $\ell$ to $v$ are discovered during the $i$'th iteration where $d(\ell, v) = i$. There is no need to process this vertex again in the subsequent iterations since any additional path found from $\ell$ to $v$ will be sub-optimal. Thus, $v$ can be removed from the input for the rest of the computation which is achieved by writing it to the final output location on DFS. We call this scheme Incremental Graph Serialization (IGS).

Figure 3.8 shows the operation of IGS for the sample graph when $\ell = 0$. Discovered vertices at a given iteration are highlighted in bold. They are serialized on DFS to construct the final output incrementally. The input graph for each iteration is smaller than the preceding one.

REDUCE

The REDUCE function saves the vertex record in line 23. It also maintains the minimum length paths and the minimum distance received so far (lines 25-30). Between lines 33 and 40, the record is updated if a smaller distance is found. Note that for unweighted graphs, this happens at most once for each vertex. The first time a vertex $v$ is discovered (i.e., $d(\ell, v)$ is set to $i$ for some $i < \infty$) all shortest paths from $\ell$ to $v$ are found.

In billion-scale graphs, there may be thousands to millions of shortest paths between
Figure 3.6: Operation of Parallel-SSSP on a sample graph with 8 vertices with $\ell = 0$. Iterations 0 and 1.
Figure 3.7: Operation of Parallel-SSSP on a sample graph with $\ell = 0$. Iterations 2 and 3.
Algorithm 1 Parallel-SSSP

1: **Input:** Graph $G=(V,E)$ with $\ell \in \mathcal{L}$ specified as the source.

2: **function** MAP(v.id, v)
3: 3: if v.active then
4: 4: d ← v.distance + 1
5: 5: msg ← (d, v.paths)
6: 6: for all u ∈ v.out-neighbors do
7: 7: if ¬ DISCOVERED(u.id) then
8: 8: 8: EMIT(u.id, msg)  \text{▷ distance}
9: 9: end if
10: 10: end for
11: 11: v.active ← False
12: 12: DFS.WRITE(v)  \text{▷ final output}
13: 13: else
14: 14: msg ← v
15: 15: EMIT(v.id, msg)  \text{▷ vertex record}
16: 16: end if
17: 17: end function

18: **function** REDUCE(v.id, [msg$_1$,...,msg$_k$])
19: 19: $d_{\text{min}} \leftarrow \infty$
20: 20: $\Pi_{\text{min}} \leftarrow \emptyset$  \text{▷ set of min length paths}
21: 21: for all m ∈ [msg$_1$,...,msg$_k$] do
22: 22: if ISMETADATA(m) then
23: 23: v ← m
24: 24: else
25: 25: if m.distance < $d_{\text{min}}$ then
26: 26: $d_{\text{min}} \leftarrow m$.distance
27: 27: $\Pi_{\text{min}} \leftarrow m$.paths
28: 28: else if m.distance = $d_{\text{min}}$ then
29: 29: $\Pi_{\text{min}} \leftarrow \Pi_{\text{min}} \cup m$.paths
30: 30: end if
31: 31: end if
32: 32: end for
33: 33: if $d_{\text{min}} < v$.distance then  \text{▷ update record}
34: 34: v.distance ← $d_{\text{min}}$
35: 35: APPEND($\Pi_{\text{min}}$, v)
36: 36: v.paths ← $\Pi_{\text{min}}$
37: 37: SAMPLE(v.paths, MAX_PATHS)
38: 38: v.active ← True
39: 39: DISCOVERED(v.id) ← True
40: 40: end if
41: 41: if v.degree < $\delta$ then  \text{▷ pass to next MAP}
42: 42: EMIT(v.id, v)
43: 43: end if
44: 44: DFS.WRITE(v)  \text{▷ send to fat vertex pool}
45: 45: end if
46: 46: end function
Figure 3.8: Incremental Graph Serialization for the sample graph when $\ell = 0$.

<table>
<thead>
<tr>
<th>iteration</th>
<th>input graph</th>
<th>final output on DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1,2,3,4,5,6,7,8</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1,2,3,4,5,6,7,8</td>
<td>0,1,2</td>
</tr>
<tr>
<td>2</td>
<td>3,4,5,6,7,8</td>
<td>0,1,2,3,4,5,6</td>
</tr>
<tr>
<td>3</td>
<td>7,8</td>
<td>0,1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>

(a landmark,vertex) pair. Storing and using millions of paths for estimating pairwise distances is impractical and should be avoided for performance reasons. In line 37, paths are sampled randomly and at most $\text{MAX PATHS}$ of them are stored as the pre-computation data. HOBA allows users to control $\text{MAX PATHS}$ according to the size of the graph. For all experiments presented in this dissertation, $\text{MAX PATHS}$ is set to 250. The sampling algorithm is explained in more detail in section 3.3.2.

Active/Inactive Vertex Labelling

Labelling vertices as active or inactive is a critical step for scalability. A previously described MapReduce algorithm by Lin [62] for solving SSSP does not adapt the labelling technique. With that approach, each vertex sends its distance and paths from the landmark to its out-neighbors at each step. Given a graph with diameter $D$, this results in $O(DE)$ messages being passed from the mappers to the reducers until convergence. In large graphs, $E$ can be considerably larger than $V$ and the number of redundant messages quickly becomes a limiting factor for the algorithm performance. Once a vertex sends it distance from the landmark to its neighbors, it should stop sending the same message repeatedly at each iteration since it has no effect on the neighbors. More specifically, a vertex should only notify the neighbors when its distance from the landmark node is updated to a smaller value.

Figure 3.9 shows a redundant message passed from B to C in step 0 of the algorithm.
Figure 3.9: Redundant messages are colored red whereas required ones are green. Notice the redundant messages sent from B to C in iteration 0, and A to B in iteration 1. The current distance of each vertex is displayed on the table in the right hand side.

Since B’s distance is $\infty$, there is no need to pass it to its out-neighbors. At the end of step 0, B’s distance is updated to 1 based on the new message received from A. In step 1, B sends its new distance to C, while A sends the same distance to B again, which in this case has no effect on B.

The redundant message traffic is avoided by labelling nodes as active or inactive. An active vertex is the one that has new information that could potentially affect its out-neighbors. For example, A is an active vertex in step 0, and B is an active vertex in step 1. When the distance information of a vertex is updated to a smaller value, it is labelled as active. Once a vertex passes its current distance and paths to the out-neighbors, it switches back to the inactive (passive) state.

Figure 3.10 demonstrates how active vertices pass distance information to their neighbors only when necessary. Notice that considering a MapReduce implementation, these messages...
Figure 3.10: Only active vertices send a message to their neighbors. A is active in iteration 0, and B is active in iteration 1. Redundant messages are avoided by the active/inactive labelling scheme.

are sent from the mappers to the reducers through the network and sorted before the reduce operation starts. Thus, minimizing the amount of intermediate data can result in a significant performance gain.

The active/inactive labelling method is in effect similar to the **vote for halt** feature of Pregel [68]. In Pregel, an individual vertex can **vote for halt** if it has no more work to do. The computation terminates when all the vertices vote for halt. A vertex that has voted for halt is analogous to an **inactive** vertex in the **Parallel-SSSP** algorithm. When all vertices are passive, there is no more information to pass along, and thus, the computation terminates. In Hadoop, we can keep track of the number of passive and active nodes using global counters. When a vertex becomes active, it increments a global counter value, which is aggregated by the MapReduce runtime (Hadoop) at the end of the job. If the value of the counter is zero, the iterations terminate implying convergence.
Selective Push using Bit Vectors

Most MapReduce adaptations of existing graph algorithms exhibit sub-optimal performance due to redundant data being sent from mappers to reducers multiple times. A fundamental reason for this inefficiency is the **vertex-centric** approach that lacks a global application state. A message is pushed from a vertex to its neighbors whether the neighbors already have the same information or not. This results in redundant data being generated and shuffled over the network from mappers to reducers at each iteration, which often causes a significant performance decrease.

The data inflation is addressed partially by labelling vertices as **active** or **inactive**. The second redundancy occurs when a vertex sends a message to one of its neighbors which already has a smaller distance value. Consider vertex 4 from the sample graph in Figure 3.3a. The distance values of its neighbors when vertex 4 is active are given in Figure 3.11. Notice that, since $d(0,1) < d(0,4)$ there is no need to send a distance message to vertex 1.

Observe that 7 and 8 are not discovered yet so a distance message should be sent to both. However, 1 was discovered previously since it has a direct link to 0. A distance message from 4 to 1 will have no contribution to the final results and should be avoided.

![Figure 3.11: a) Vertex 4 when it is active, along with its neighbors. b) The corresponding bit vector for the sample graph. Only those fields that correspond to the vertices above are highlighted.](image-url)
Such redundant messages can be suppressed using a global bit vector that is accessible from all compute nodes. The bit vector technique is reminiscent of coloring vertices as white, grey or black in serial implementations of BFS or Dijkstra’s algorithm. Let $\delta$ be an $n$-bit vector where $n = |V|$. In unweighted graphs, $\text{Parallel-SSSP}$ maintains the following condition for $\delta$:

$$
\delta(i) = \begin{cases} 
1, & \text{if } d(\ell, i) < \infty, \\
0, & \text{otherwise}
\end{cases}
$$

Initially only $\delta(\ell) = 1$ and the rest of the bit vector is set to 0. In the beginning of each iteration, compute nodes read the latest bit vector from DFS and store it in local memory. Before sending a message to an out-neighbor, an active vertex first checks whether it was previously discovered by querying its local copy of the bit vector (line 7). When a new vertex is discovered, the corresponding location of the local bit vector is set to 1 (line 39). At the end of each iteration, all bit vectors are merged into a single one by a bitwise-OR operation to maintain the global state for $\delta$, which is then stored on DFS. It is worth noting that the bit vector optimization does not require any modifications to the MapReduce programming model and it can be implemented in common MapReduce frameworks such as Hadoop. The coordination is achieved by merging the local copy of all bit vectors at the end of each iteration and re-distributing the newest copy to all compute nodes through the distributed file system.

An unweighted graph with 100 million vertices can be represented using a 12.5 MB bit vector, which easily fits in the memory of a single compute node. For weighted graphs, it is necessary to store the actual distance information inside the vector so a slight modification is required. Depending on the size of the weights, a few bits or multiple bytes may be reserved for each vertex.
Parallel Bit Vector Merge Algorithm

A challenging part of Selective-Push is to update the global bit vector by merging the local copies altered by each reducer efficiently. This operation must be carried out at the end of each iteration to ensure the global bit vector represents the current state of the graph at that point of the computation.

After the REDUCE phase, each reducer dumps its own altered local copy of the bit vector to a pre-determined location on the distributed file system (DFS). A naïve approach to merge all bit vectors into a single one would be to spawn a single process which would read all bit vectors on DFS one by one and perform a bitwise-OR operation to construct the final updated vector. However, this scheme can have multiple drawbacks depending on how it is implemented.

- In most MapReduce frameworks including Hadoop, there is a controller application that runs on the user’s local machine and submits MapReduce jobs to the cluster. When a MapReduce job finishes or fails, the control returns back to the local application. If the naïve scheme is adapted, the single process that reads and merges all bit vectors can be invoked from the local machine of the user after the completion of each MapReduce job which corresponds to another iteration of Parallel-SSSP. However, in typical deployments, the user application does not share the same network interconnect with the compute cluster. There is usually a slower link between the user and the cluster, and transferring large files from the cluster to the user application can take longer compared to moving them between cluster nodes. If a single application running on a local user machine is to merge all bit vectors, it has to access DFS and download all available data. Such an implementation is limited by the capacity of the network interconnect that links the user to the cluster machines, and it may not yield the desired throughput.

- Another way to implement the naïve approach would be to spawn the single process
using a wrapper MapReduce job to execute it in one of the cluster nodes. That is, a single mapper with no reducers can perform the bitwise-OR operations by accessing DFS and reading all vectors. Although this solves the limited network bandwidth problem, it would still not be efficient when there is a large number of bit vectors to be merged into a single one. Since there is a single application that goes through each vector one by one, the merge operation may take a long time to complete. This approach also reduces the cluster utilization since there is only one compute node that does all the computational work while the rest of the cluster nodes only act as data stores, serving DFS requests.

The above idea can be improved to merge bit vectors in parallel using a MapReduce job. Instead of a single node, multiple cluster nodes can merge different bit vectors in parallel and repeat the same process until a single bit vector is produced. We start by describing a 2-level bit vector merge process and later generalize it to $k$-levels.

2-level Parallel Merge

Let $R$ be the number of reducers in $\text{Parallel-SSSP}$ that produce $R$ altered bit vectors at each iteration. The goal of 2-level parallel merge is to combine $R$ vectors into one using a single MapReduce job. In the first level, assume $M$ map tasks are spawned. Each mapper combines $\frac{R}{M}$ bit vectors into one and sends it to the only reduce task available. The single reducer then iterates over all $M$ vectors received from the mappers, constructs the final global vector by bitwise-OR and writes the result on DFS.

The optimum number of map tasks can be determined by analyzing the total run time to merge all bit vectors. Map tasks run in parallel and each mapper merges $\frac{R}{M}$ bit vectors. Next, the single reducer combines the remaining $M$ vectors. To find the optimum number of map tasks, the following function should be minimized:
\[ f(M) = \frac{R}{M} + M \]  

(3.3)

Note that \( f(M) \) represents the time to merge \( R \) vectors into a single vector using \( M \) map tasks. Minimizing \( f(M) \) corresponds to minimizing the number of bit vectors merged *serially*.

Taking the derivative yields:

\[ f'(M) = \frac{\partial}{\partial M} \left\{ \frac{R}{M} + M \right\} = 1 - \frac{R^2}{M} \]  

(3.4)

Since \( M \) must be an integer, setting \( M = \lceil \sqrt{R} \rceil \) yields the desired result. The total run time of 2-level Parallel Merge is \( 2\sqrt{R} \) and the time complexity of the algorithm is \( O(\lceil R^{\frac{3}{2}} \rceil) \).

Figure 3.12 shows an illustration of the 2-level Parallel Merge for \( R = 15 \) bit vectors. The optimum number of map tasks is four, and all mappers but the last one merge four vectors each. Finally, the reducer combines all intermediate results and produces the global bit vector. Note that each level takes \( t \) amount time, where \( t \) is the time to merge four bit vectors into one.

**k-level Parallel Merge**

For very large values of \( R \), a single MapReduce job with two levels will not be efficient. In this case, the same idea can be extended to multiple levels. Given \( k \) as the number of levels to merge all bit vectors, the method described in the 2-level Parallel Merge algorithm can be extended as follows:

At each level, a map task merges \( \sqrt[k]{R} \) bit vectors locally and dumps the intermediate output back to DFS. The number of reducers should be set to 0 for all MapReduce jobs except for the last one. The output of the \( i \)'th job will be the input to the \( i+1 \)'th job.
Figure 3.12: Illustration of the 2-level Parallel Merge algorithm. There are 15 vectors and four map tasks. The first three map tasks merge four vectors each and the last one merges three vectors. The single reducer combines all four vectors received from the map tasks and produces the final answer. Note that both levels take $t$ amount of time, in this case $t$ is the time to merge four vectors.
The final MapReduce job will basically be identical to the 2-level Parallel Merge with a single reducer. Given $k$ levels, the total time complexity of this algorithm is $O(k[R^1_1])$.

### 3.3.1 Load Balancing High-Degree Vertices

The work done inside a map task is dominated by the total number of distance messages generated (lines 6-10). The more edges processed inside a map task, the longer it takes to finish. The skewed degree distribution of large real-world graphs results in disproportionate map task run-times. While most map tasks finish within minutes, a small percentage of those that process vertices with very high degrees can take over an hour to complete. This is an undesired behaviour in MapReduce because the REDUCE phase cannot start until all map tasks finish. The cluster utilization drops significantly towards the end of the MAP step as the majority of the compute nodes stay idle, waiting for a small fraction of the map tasks to complete.

We address this problem by processing high-degree vertices simultaneously in all map tasks rather than assigning each vertex to a random map task as usual. A vertex is identified as a fat vertex if its degree is greater than a threshold value $\delta$. For the large graph datasets considered in this paper, we set $\delta$ to 50000. If a fat vertex is identified during the REDUCE step, it is sent to a special directory called the fat vertex pool on DFS (lines 41-45). Each reduce task creates a separate file under the fat vertex pool for serializing high-degree vertices it identifies.

Before the next MAP step begins, all map tasks execute \texttt{PROCESS-FAT-VERTICES} during the task setup. Algorithm 2 shows the pseudo-code for this function. Input is the path to the fat vertex pool on DFS and the number of map tasks spawned for the job. Each task pulls all fat vertices discovered in the REDUCE step of the previous iteration from the DFS in randomized order (lines 4-10). The randomization prevents file system swamping. If all tasks read the fat vertices in the same order, data nodes serving the initial DFS requests may not be able to keep up with the large number of queries and the file system may become
Algorithm 2 PROCESS-FAT-VERTICES

1: **Input**: Pool: DFS directory that contains fat vertex files
2: M: Number of map tasks

3: **function** PROCESS-ALL(Pool, M)
4: file_list ← DFS.READ(Pool)
5: RANDOMIZE(file_list)
6: for all file ∈ file_list do
7:     for all v in file do
8:         PROCESS-ONE(v)
9:     end for
10: end for
11: **end function**

12: **function** PROCESS-ONE(v, M)
13:     task_id ← RUNTIME.TASKID \(\triangleright\) get current task id
14:     d ← v.distance+1
15:     msg ← (d, v.paths)
16:     for all u ∈ v.out-neighbors do
17:         if u.id mod M = task_id then \(\triangleright\) range partitioning
18:             if ¬ DISCOVERED(u.id) then
19:                 Emit(u.id, msg) \(\triangleright\) distance
20:         end if
21:     end for
22:     if v.id mod M = task_id then \(\triangleright\) vertex record
23:         msg ← v
24:         Emit(v.id, msg)
25:     end if
26: end if
27: **end function**

unstable. Randomizing the file list helps balancing the load on DFS and results in higher IO throughput. The **PROCESS-ONE** function is structurally similar to the **MAP** operation of Algorithm 1 described earlier. In addition, it involves a range partitioning mechanism for load balancing. When a **fat vertex** is processed, its out-neighbors are divided into disjoint subsets of almost equal size using a partitioning function.

Assume there are \(m\) map tasks for the job and let \(\mathcal{N}(v)\) be the out-neighbor set of \(v\). The subset of neighbors that fall in the scope of map task \(M_i\) are determined by the partitioning
Figure 3.13: Sample range partitioning on a vertex with degree eight. There are four map tasks and all of them pull the same vertex record from the pool on DFS. Each map task processes a disjoint subset of neighbors where subsets are determined by a partitioning function. Note that a vertex with degree eight is far from being a fat-vertex in practice. Furthermore, there are usually hundreds of map tasks spawned for a MapReduce job. The sample vertex with degree eight and four map tasks are chosen to demonstrate the concept on a micro scale. In practice, the fat vertex pool contains thousands of vertices with degrees ranging from tens of thousands to millions.

Function $\phi$ such that

$$\phi(M_i, v) = \{ u : u \in \mathcal{N}(v), \ u \mod m = i \}$$

The range partitioning mechanism is demonstrated on a sample vertex in Figure 3.13. Each task is responsible from sending a distance message to its own range of out-neighbors for a given fat vertex. This ensures that the computational load introduced by the high degree is distributed evenly among all tasks and solves the straggling process problem. Finally, the vertex record is also sent only once, based on a similar modulo operation (lines 23-26).
This avoids generating multiple copies of the same vertex record in all map tasks. After the MAP phase completes, the fat vertex pool is cleared for the next iteration.

3.3.2 Path Sampling

Although Parallel-SSSP explores all shortest paths between each landmark and vertex pair $(\ell, v)$, it does not store all of these paths as the final output. There are two performance problems with storing all shortest paths for all $(\ell, v)$ pairs:

- The algorithm proceeds by sending shortest paths from a vertex to its neighbors. In each iteration, paths are sent from mappers to reducers over the network. In addition, an external global sort is applied to the map output before passing it to the reducers. Larger map outputs often result in excessive sorting and shuffling times and decrease the application performance. In billion-sized graphs, there may be millions of shortest paths between some $(\ell, v)$ pairs. Thus, storing, sorting and shuffling all shortest paths should be avoided for large networks.

- The REDUCE function of Parallel-SSSP saves the minimum length shortest paths in memory until all messages sent to a particular vertex are read (lines 21-32). The final list of shortest paths are stored in an array like data structure denoted by $\Pi_{min}$ as seen in the pseudo code. In billion-scale networks, $\Pi_{min}$ can be arbitrarily large and sometimes contain millions of shortest paths. Storing too many shortest paths can result in out-of-memory errors and terminate the application unexpectedly.

To avoid these problems, the number of shortest paths sent from a parent vertex $v$ to its neighbors $u$ is limited by a threshold. We use $\tau$ to denote the maximum number of shortest paths (i.e., MAX_PATHS) stored for each $(\ell, v)$ pair. A vertex with degree $d$ receives $O(d\tau)$ shortest paths during the REDUCE step and only a fraction of those paths are stored after the sampling algorithm is applied.
There are several methods to sample shortest paths for a given vertex and landmark pair. We explain each technique with their advantages and disadvantages and finally present the weighted random walk sampling technique used for the implementation of Parallel-SSSP. For all sampling techniques described in this section, we assume all shortest paths from the landmark to the target vertex are combined into an induced sub-graph $G_P$ as illustrated for the sample graph in Figure 3.3b. The goal of path sampling is to traverse $G_P$ efficiently and uniformly sample $\tau$ paths. For simplicity, we limit the discussion to unweighted graphs.

**Breadth First Sampling**

A straightforward approach for sampling is to perform a breadth first traversal (BFS) of the induced sub-graph $G_P$ to enumerate the paths and sample $\tau$ of them. In regular BFS, vertices are stored in a first in first out (FIFO) queue and removed one by one. To enumerate the shortest paths, all discovered paths between the landmark $\ell$ and any vertex $u$ in $G_P$ should also be stored in the queue. That is, when the neighbors of an arbitrary vertex $u$ are being expanded, all shortest paths between the $(\ell, u)$ pair should be passed to the neighbors of $u$. This is illustrated in Figure 3.14. The information required to be stored in memory for each vertex is shown in the table on the right. For example, there are three shortest paths between $\ell$ and $t$: $\{[\ell, v, t], [\ell, q, t], [\ell, s, t]\}$. These paths are passed onto the neighbors of $t$ when they are expanded. Observe that both $y$ and $z$ contain all shortest paths between $\ell$ and $t$. As a result, the breadth first approach is memory intensive and can cause out-of-memory errors when $G_P$ is large and dense.

**Depth First Sampling**

The extensive memory requirements of breadth first sampling can be addressed by a depth first traversal of $G_P$. We can think of the enumerated paths as a long input stream of size $|s|$. Using reservoir sampling [87], $\tau$ paths can be sampled uniformly by choosing each path with a probability of $\frac{1}{|s|}$. This technique requires $O(D\tau)$ space in memory since a shortest
path can contain at most $D$ vertices where $D$ is the graph diameter. However, the depth first approach can be quite slow especially when $G_P$ is large. In depth first traversal, the same shortest paths are explored repeatedly. This is illustrated in Figure 3.15. To visit the three different paths \{[\ell, ..., v], [\ell, ..., q], [\ell, ..., s]\}, the path through $\ell$ to $u$ is explored three times. Repeated path exploration results in slow application run time and can take extremely long when the number of shortest paths are in the order of millions. Consequently, the depth first traversal of $G_P$ will not be efficient in large networks although it solves the potential memory overflow problem.
Uniform Random Walk Sampling

Another alternative for path sampling is to perform random walks on the induced sub-graph $G_P$. Each random walk starts from the landmark $\ell$ and ends at the target vertex $u$. To accomplish this, a breadth first traversal from $\ell$ to $u$ is applied to the induced sub-graph and each vertex is coupled with a list of successors that can be followed to reach the target vertex, $u$. Essentially, the successor list of a vertex consists of a subset of its neighbors. Note that, building successor lists is required to guarantee optimal length random walks. Otherwise, a random walk may take longer than the shortest path length or even contain cycles.

In order to sample $\tau$ paths, $\tau$ random walks from $\ell$ to $u$ are done and duplicates are
A naïve duplicate removal algorithm that compares all paths takes $O(D\tau^2)$ time. For large values of $\tau$, a sort and linear sweep can be done to complete in $O(D\tau \log \tau)$ time. More advanced hashing algorithms with $O(1)$ amortized cost for checking duplicates can also be used for better performance.

Figure 3.16: Uniform random sampling on a sample graph.

In a completely random walk, a random successor is chosen uniformly from the successor list of a vertex. That is, given a vertex with $|s|$ successors, one of them is chosen with probability $\frac{1}{|s|}$. This approach has a drawback in the number of unique paths that can be sampled when potential successors have uneven number of shortest paths to the target vertex. Figure 3.16 illustrates this problem on a sample graph. There are 1000 shortest
paths between $u$ and $v$, and only one shortest path between $u$ and $q$. Assume the objective is to sample 100 shortest paths from $\ell$ to $u$. The random walk starts from $\ell$ which has two successors. If a successor is chosen randomly with $\frac{1}{2}$ for $\tau = 100$ times, $q$ will be chosen approximately 50 times and all paths from $q$ to $u$ will be identical. Thus, 49 of these paths will be removed at the end. Even if the random walks from $v$ to $u$ produce a perfectly distinct set of 50 shortest paths, the algorithm will sample 51 unique shortest paths from $\ell$ to $u$ at the end. Thus, although it is fast and memory efficient, uniform random sampling may reduce the desired sample size to a fraction of $\tau$.

Figure 3.17: Weighted random sampling on a sample graph.
Weighted Random Walk Sampling

The uniform random walk approach can be tweaked by adding weights to the probability of choosing successors. The basic idea is illustrated in Figure 3.17. There are 200 shortest paths between \((q, u)\), 100 shortest paths between \((t, u)\) and 500 shortest paths between \((v, u)\). The probability of choosing a particular successor \(y\) is proportional to the number of shortest paths between \(y\) and \(u\). In the figure, the probabilities for choosing \(q\), \(t\) and \(v\) are \(\frac{2}{5}\), \(\frac{1}{5}\) and \(\frac{5}{8}\) respectively. The weighting scheme avoids selecting poor successors and yields more diverse shortest paths as the outcome. Note that, the number of shortest paths passing through each vertex in \(G_P\) can be calculated during the initial breadth first traversal for building successor lists.

3.3.3 Complexity in MapReduce

We use the complexity measures introduced for MapReduce algorithms by Afrati et al. [2, 3]. The complexity of a MapReduce algorithm has two main components. Communication cost is the amount of data transmitted from mappers to reducers over the network. Computation cost is the total work done inside the MAP and REDUCE functions. In Parallel-SSSP, there is a one-to-one correspondence between the communication and computation costs so we only analyze the former.

In an unweighted connected graph, each vertex becomes active exactly once. For each active vertex, at most one distance message is sent to all neighbors. In practice, the bit vector optimization suppresses most edges after the first few iterations. Thus, the total number of distance messages sent from mappers to reducers is \(O(m)\).

For a given iteration, sending one record per vertex from mappers to reducers yields \(O(n)\) messages. Although the algorithm requires at most \(D\) iterations to converge, the majority (90%) of the vertices are generally discovered and removed from the input in less than 10 iterations due to small-world and shrinking diameter phenomena in large real-world graphs [77, 61]. As a result, the majority of the communication takes place within the first \(D'\)
iterations where $D'$ is the effective diameter.

Therefore, the communication and computation cost of \textit{Parallel-SSSP} for unweighted graphs is $O(Dn + m) \approx O(D'n + m)$. Note that in scale-free networks $D'$ is small. Empirical studies on many real-world graphs typically suggest values between 4 and 7 while $D$ is observed to decrease further with increasing graph size [15]. For comparison, a previously described MapReduce algorithm by Lin [62] for solving \textit{SSSP} runs in $\Theta(Dn + Dm)$ time. This algorithm sends messages from all nodes to their neighbors at each iteration until convergence without avoiding any redundancy. The novelty and main advantage of \textit{Parallel-SSSP} comes from the Selective Push and IGS optimizations. In section 3.5 we show that these techniques speed-up the computation five to seven times in massive graphs.

3.3.4 Extending Parallel-SSSP to Weighted Graphs

As covered in Section 3.3, in an unweighted connected setting each vertex becomes active exactly once. When a vertex $v$ is discovered, i.e., its distance $d(\ell, v)$ from the landmark $\ell$ is updated to a value less than $\infty$, the optimal length is found and there is no need to explore this vertex again. Thus, a simple breadth first traversal of the graph starting from the landmark node is sufficient to solve \textit{SSSP} considering a serial implementation.

However in weighted graphs, the distance of a vertex can be updated to shorter values as the algorithm makes progress. This is why Dijkstra’s Algorithm uses a priority queue and orders vertices based on their values in a serial implementation. The distance of any vertex in the priority queue can be updated until it is removed from the queue, in which case the optimal solution is found [19].

In \textit{Parallel-SSSP} there is no priority queue and there is no way of making sure whether a distance value is optimal or not. A vertex can be activated multiple times and its distance from the landmark can potentially be updated to shorter values at any given iteration. Consequently, \textit{Incremental Graph Serialization} (IGS) cannot be used in weighted graphs. In addition, equation 3.3 should be revised to include edge weights as follows:
\[ d(v, \ell)^{(k)} = \min \left\{ d(v, \ell)^{(k-1)}, \min_{u \in \mathcal{N}(v)} \left\{ d(u, \ell)^{(k-1)} \right\} + w(v, u) \right\} \]  

(3.6)

where \( w(v, u) \) is the weight of the edge between \( v \) and \( u \). Initial conditions and the REDUCE steps remain identical and the MAP function is revised to include two minor modifications to tackle the edge weights.

- Each active vertex \( v \) adds the edge weight \( w(v, u) \) instead of 1 to its current distance when sending a distance message to an out-neighbor \( u \).

- The \([0, 1]\) bit-vector for the Selective Push optimization described in Section 3.3 cannot be used as is for weighted graphs. Instead of a single bit, the exact distance information for each vertex should be kept in memory. Depending on the size of edge weights, a few bits or multiple bytes per vertex may be required. That is, the bit vector is replaced by an in memory distance vector for weighted graphs.

Note that the memory requirements of a distance vector will be higher than a bit vector although it is still possible to scale the algorithm for billion-sized networks. A compute node on a commodity cluster typically contains 1 to 4 GB of memory. If each vertex requires 1 to 8 bytes of storage depending on edge weights, a distance-vector can handle graphs with 125 million to 4 billion vertices. Further compression and edge weight normalization schemes can be applied to better utilize main memory.

### 3.4 Distance Estimation with PathCrawler

Shortest paths discovered using Parallel-SSSP form the basis of distance estimation in large networks. In this section, we describe the PathCrawler algorithm which is an extension of the Landmark-BFS method. The main idea is to use the extra pre-computation data to increase the accuracy of the results.
Consider vertices 7 and 8 from the sample graph in Figure 3.3a when $\ell = 0$. There are three shortest paths between each pair $(0,7)$ and $(0,8)$. The true distance between 7 and 8 can be calculated by using either $\{0,1,4,7\} - \{0,1,4,8\}$ or $\{0,2,5,7\} - \{0,2,5,8\}$ at the same time. The shortest paths from $\ell = 0$ to vertices 7 and 8 are given in Figure 3.18. Possible shortest path combinations that yield the true distance for $d(7,8)$ are presented in Figure 3.19. Note that out of nine possible path combinations, only two of them result in the true distance. If only one shortest path for each (landmark, vertex) pair is stored, the probability of finding the exact answer is $\frac{2}{9}$.

![Figure 3.18: Shortest paths from the landmark ($\ell = 0$) to vertices 7 and 8. Note that Parallel-SSSP can compute multiple shortest paths for each (landmark, vertex) pair and the results are stored on the distributed file system. In this example, there are three shortest paths between each of the pairs $(0,7)$ and $(0,8)$.](image)

With PathCrawler, we increase the odds of finding the true distance by leveraging all shortest paths available from the pre-computation step. The algorithm essentially crawls the entire set of paths through the landmarks and obtains the shortest possible path from source to destination. The implementation is given in Algorithm 3. Input is a pair of vertices $\{s,t\} \in V$ and shortest paths from/to the set of landmarks $L$. All shortest paths are merged in lines 4-12 to create an induced sub-graph $G_P(s,t)$. That is:
Figure 3.19: Combination of shortest paths that yield the true distance between the pair 0,7 in the sample graph. The first set of paths is \([\{0, 1, 4, 7\} \text{ and } \{0, 1, 4, 8\}\)] and the second set of paths is \([\{0, 2, 5, 7\} \text{ and } \{0, 2, 5, 8\}\)] All other combinations yield sub-optimal answers.

\[
G_P(s, t) = \bigcup_{\ell_i \in \mathcal{L}} \Pi^*(s, \ell_i) \cdot \Pi^*(\ell_i, t) 
\]  

(3.7)

In addition, there is an optional 2-hop guarantee optimization\(^1\) which adds all neighbors of \(s\) and \(t\) to \(G_P(s, t)\). Finally, Dijkstra's algorithm (or BFS for unweighted graphs) is called as a sub-routine to find an approximate shortest path.

The PathCrawler algorithm deals with the potentially large induced sub-graphs by limiting the number of shortest paths between a vertex and a landmark, as discussed previously. Let \(\tau\) be the maximum number of shortest paths stored for each (landmark, vertex) pair, i.e. MAX_PATHS. The length of a shortest path has an upper bound of \(D\), the graph diameter. The size of \(G_P\) is bounded above by \(O(kD\tau)\) where \(k\) is the number of landmarks and the worst-case run time complexity of the algorithm is \(O(k^2D^2\tau^2)\). In practice, the size of

\(^1\) Recommended when \(m = O(n)\)
the induced sub-graph is quite smaller than $O(kD\tau)$ because multiple shortest paths from several landmarks highly overlap.

### Algorithm 3 PathCrawler

1: **Input:** $\{s,t\} \in V, \mathcal{L}$: Set of landmarks
2: $\Pi^*(s, \ell)$ and $\Pi^*(\ell, t) \forall \ell \in \mathcal{L}$
3: **function** PATHCRAWLER($s,t$)
4: $P \leftarrow \emptyset$
5: for all $\ell \in L$ do
6: for all $\pi \in \Pi^*(s, \ell)$ do
7: $P \leftarrow P \cup \pi$ \hspace{1cm} $\triangleright$ paths from $s$ to $\ell$
8: end for
9: for all $\pi \in \Pi^*(\ell, t)$ do
10: $P \leftarrow P \cup \pi$ \hspace{1cm} $\triangleright$ paths from $\ell$ to $t$
11: end for
12: end for
13: for all $u \in s$.out-neighbors do \hspace{1cm} $\triangleright$ optional
14: $e \leftarrow (s, u)$
15: $P \leftarrow P \cup e$
16: end for
17: for all $u \in t$.in-neighbors do \hspace{1cm} $\triangleright$ optional
18: $e \leftarrow (u, t)$
19: $P \leftarrow P \cup e$
20: end for
21: Let $G_P$ be the sub-graph of $G$ induced by $P$
22: $\pi^* \leftarrow$ Dijkstra($G_P$, $s$, $t$)
23: return $\pi^*$
24: end function

Unlike previous distance estimation methods which rely on a fast central server to store the pre-computation results, the PathCrawler algorithm is designed to work on shared-nothing architectures where input is stored on and read from a distributed file system and partitioned efficiently among a large number of compute nodes. PathCrawler does not use any in-memory optimization techniques such as the short-cutting methods adapted by the Sketch and Landmark-BFS algorithms. Instead, it makes use of much larger pre-computation data composed of multiple shortest paths between each (landmark, vertex) pair and uses this information to obtain better results. In chapter 4, we present a parallel implementation of this algorithm to solve the all pairs shortest path problem (APSP) for a set of vertices.
3.5 Experimental Evaluation

3.5.1 Cluster Setup

The large scale experiments were run on a 20 node Hadoop cluster with 8GB of memory and 4 physical cores at 3.4GHz in each machine. The main software stack includes Ubuntu 12.04 along with Java 1.6 64-bit and Hadoop 1.1.2.

3.5.2 Accuracy

We designed two different experiments to measure the accuracy of distance estimation.

Medium Sized Networks

The first class of experiments are run on the medium sized datasets where the correct distances are calculated using the NetworkX library. Figures 3.20, 3.21 and 3.22 show three heat maps that compare the true and estimated distance in medium sized networks. Darker squares correspond to denser areas of the pairwise distance distribution. For the pre-computation step, we used five landmarks with the highest degree in each network and stored up to 250 shortest paths between each (landmark, vertex) pair. For each graph, we calculated the exact and approximate shortest path lengths among half a million pairs. Exact shortest paths are computed using the NetworkX Python software package [45] on a single desktop computer. Approximate shortest paths are computed in parallel by running the PathCrawler algorithm on the cluster. We skip the details of the parallel implementation in this section since chapter 4 contains an in-depth discussion of the topic with alternative approaches and their cost analyses.
The average error rate in distance estimation is computed by the following formula:

$$\epsilon = \frac{\sum d_{approx}}{\sum d_{exact}} - 1 \quad (3.8)$$

This metric is also called the \textit{stretch} of the approximation. The Facebook graph resulted in the lowest stretch of 0.0002 followed by 0.0018 and 0.0030 in the Enron and Wikipedia graphs. We found the true shortest path length nine out of ten times in all datasets.

Figure 3.20: Distance estimation accuracy for Facebook. Median pairwise distance is 4 hops. The Facebook dataset produced almost perfect results.
Figure 3.21: Distance estimation accuracy for Enron. Median pairwise distance is 4 hops. Slight approximation errors are visible around the (3,4) and (4,5) regions.

Figure 3.22: Distance estimation accuracy for Wikipedia. Median pairwise distance is 3 hops. A noticeable approximation error is visible around the (3,4) region.
Large Networks

The second class of experiments are run on the large WWW and Facebook datasets. A distributed experiment is designed to measure the accuracy using a sampling technique. The accuracy of large scale distance estimation with increasing number of landmarks is reported in Figure 3.23. For this experiment, we chose 10 random vertices referred as the oracles and computed the length of the shortest path from each oracle to the rest of the graph using Parallel-SSSP. Then, we sampled 5 million vertices and approximated their distance to the oracles and reported the error. Notice that even in the WWW graph which contains 700 million nodes and 12 billion edges, we were able to calculate the shortest paths with 7% average error using only 10 landmarks. In both graphs, we selected the highest degree vertices as the landmarks.

![Figure 3.23: Error Rate vs. Number of Landmarks](image)

3.5.3 Performance and Scalability

Selective Push

We evaluated the efficiency of the Selective Push optimization described in section 3.3 by running the Parallel-SSSP algorithm on the Twitter and WWW datasets. We measured the
number of distance messages sent from mappers to reducers with and without the bit vector optimization and plotted the results in Figure 3.24. Observe that the bit vector dramatically reduces the number of messages sent from mappers to reducers when the number of messages are over 100 million, resulting in an order of magnitude improvement. The Selective Push + IGS combination reduced the run-time from 146 minutes to 19 minutes in the Twitter graph, and 581 minutes to 115 minutes in the WWW graph for a single landmark. For this experiment, we set $\text{MAX\_PATHS}$ to one. That is, we only stored one shortest path between each (landmark, vertex) pair. For larger values of $\text{MAX\_PATHS}$, we observed that the run time difference between the optimized and unoptimized versions of Parallel-SSSP tends to increase even further. This is an expected consequence because more shortest paths yield larger messages which increase the intermediate data size. In summary, the Selective Push + IGS combination is a unique component of the Parallel-SSSP algorithm which allows scaling to massive graphs with billions of edges.

Figure 3.24: Number of distance messages with and without the bit-vector for the first 15 iterations of Parallel-SSSP. Note that the WWW graph converges in approximately 60 iterations.
Effects of the Small-World Phenomena

In Parallel-SSSP, the time spent on a particular iteration varies significantly based on the number of shortest paths computed. For example, the first iteration runs relatively fast because only the neighbors of the source vertex are discovered. The second one takes longer because all vertices that are two hops further from the source are identified. This increase continues exponentially until the majority of the vertices are discovered. Due to the small-world and shrinking diameter phenomena in large graphs [77, 61], the effective diameter is small and over 90% of the vertices are discovered rapidly. In practice, there are usually two or three booming iterations where a large portion of the shortest paths are computed. When a high-degree vertex is chosen as the source, a booming effect is often observed within two or three hops around it. This behavior can be examined in Figure 3.24. There is a steep increase in the number of messages generated during the second and third iterations for the Twitter and WWW graphs respectively. The number of messages begins to go down in both graphs soon after reaching the peak, as most of the shortest paths are already computed at that point. The overall run time of Parallel-SSSP is dominated by the first $D'$ iterations where $D'$ is the effective diameter.

It may take longer to observe the booming effect when the algorithm is started from a random low-degree vertex which is loosely connected to the rest of the graph. Figure 3.25 shows the total number of reachable vertices from a random source within increasing number of hops. Notice that a rapid jump starts with the fifth hop in the WWW graph. Almost all vertices are reachable by the tenth hop in both graphs except for the long and sparse chain-like sequences.

Figure 3.26 shows the percentage of records sent from mappers to reducers when a random vertex is chosen as the landmark for Parallel-SSSP. Another interpretation of this figure is the decrease of the problem size over time. As we explained the IGS technique in section 3.3, recently discovered vertices are sent to the final output location on DFS and discarded from the remaining iterations when the graph is unweighted (line 12 of Algorithm 1).
Consequently, the records sent from mappers to reducers correspond to those vertices that have not been discovered yet. As the figure suggests, large percentage rates are observed within the first few iterations and the algorithm starts converging rapidly afterwards. The input size reduces to a small fraction of the initial value after reaching the effective diameter. Notice that figures 3.25 and 3.26 are reverse-shaped plots emphasizing on two different measures that are inversely proportional. Both figures indicate that the majority of the computation and communication takes place within the first $D'$ iterations. The figures are plotted by sampling a set of nodes from both graphs and averaging the results to represent
a single random vertex.

Performance Optimizations in Parallel-SSSP

The efficiency of Parallel-SSSP comes from three novel optimization techniques:

- **Incremental Graph Serialization**: Sends vertices to the final output location on DFS soon after they are discovered - section 3.3

- **Selective Push**: Uses bit-vectors to suppress redundant messages sent from mappers to reducers - section 3.3

- **Fat Vertex Optimization**: Handles high-degree vertices exclusively to avoid the straggling map task problem - section 3.3.1

Figures 3.27 and 3.28 show the combined effect of these optimizations on the WWW and Twitter datasets. We compare the optimized algorithm with a regular version which does not contain any of the optimizations above. We plot the run time distribution of the map tasks for one of the long-running iterations that dominate the total run time in both versions of the algorithm. Both figures contain 400 map tasks sorted in ascending order of execution time.

The straggling map tasks are easily observable in Figures 3.27a and 3.28a with the long tail rising steeply towards the end. The longest running map task for the WWW graph takes about 1750 seconds to complete whereas the shortest one runs in approximately 500 seconds with the regular algorithm which does not use any of the optimization techniques. The gap is larger for the Twitter graph with the slowest and fastest map tasks taking 1300 and 6250 seconds respectively. The main reason for the increasing tail is the presence of very few high-degree vertices in both graphs due to the skewed degree distribution. The more edges a map task processes, the longer it takes to complete. Map tasks assigned to process the highest degree vertices fall behind and result in under-utilization of the compute cluster. A large number of idle CPU cores hold onto the few busy ones without doing any useful work.
Figure 3.27: Map task run times for iteration #4 over the WWW graph. Median task completion times are 954 and 340 seconds for the regular and optimized versions respectively.

until the REDUCE phase starts. The under-utilization can be observed more clearly in Figure 3.28a. The first 200 tasks finish in approximately 2000 seconds whereas the second half takes over 6000 seconds to complete. If there were 400 CPU cores to run all tasks simultaneously, half of the cluster would stay idle for over an hour.
The optimized algorithm removes the long tail at the end and balances individual task execution times. Load balancing helps increasing the cluster utilization and decreasing the total run time dramatically. The fastest and slowest map tasks take 290 and 414 seconds for the WWW graph. The gap is smaller for the Twitter network with task run times ranging from 56 to 128 seconds. The overall effect of the optimization techniques for a single iteration

Figure 3.28: Map task run times for iteration #3 over the Twitter graph. Median task completion times are 2183 and 78 seconds for the regular and optimized versions respectively.
is observable in Figures 3.27b and 3.28b. There is a noticeable gap between the regular and optimized versions with median task run times going down from 954 to 340 seconds for the WWW graph and 2183 to 78 seconds for the Twitter graph.

To understand why the optimized algorithm yields better speed-up for the Twitter dataset, we look at the individual contribution of each technique. Figure 3.24 shows that Selective Push kicks in starting iterations #6 and #3 for the WWW and Twitter datasets respectively. The difference between the number of messages generated with and without the bit-vector is negligible in iteration #4 of the WWW graph. However, there is over an order of magnitude difference between the two quantities in iteration #3 of the Twitter dataset. As a result, Selective Push yields considerable speed-up on the Twitter dataset whereas it is not as effective on the WWW graph for the particular iterations considered. The three times speed-up in iteration #4 of the WWW graph mainly results from handling high-degree vertices efficiently and avoiding straggling map tasks. The Twitter dataset benefits from all optimizations during iteration #3 so a much higher speed-up is observed.
3.6 Summary

We presented Parallel-SSSP, a MapReduce algorithm for solving the single source shortest path problem. The objective is to compute and store multiple shortest paths between the source vertex and all other vertices in the graph. The output of this computation can be considerably large for billion-scale networks and it is stored on the distributed file system. We utilize basic graph compression techniques to reduce the size of each record and limit the number of shortest paths stored for each pair using a fast path sampling algorithm based on weighted random walks.

In large networks, the effective diameter is often small so the majority of the graph can be reached within a few hops from a random vertex. This is called the small word phenomenon and it may have negative effects on distributed graph algorithms. For example, there is usually a rapid increase in the number of messages exchanged between different processes due to the small diameter. Thus, handling large graphs efficiently using a vertex-centric distributed framework such as MapReduce is a challenging problem.

A critical performance bottleneck for most MapReduce algorithms is the intermediate data size. Most MapReduce adaptations of existing graph algorithms exhibit sub-optimal performance due to redundant intermediate data being generated and sent over the network. We suggest three optimization techniques to decrease the intermediate data size for Parallel-SSSP. The active/passive vertex labelling technique avoids visiting all edges of the graph at each iteration. The Selective-Push optimization uses a global bit vector to summarize the graph state and avoids sending redundant messages from a vertex to its neighbors. In unweighted graphs, the final output is constructed incrementally by serializing the recently visited vertices on the distributed file system in each iteration. This scheme is called IGS - Incremental Graph Serialization and it avoids reading and writing the entire graph structure at every iteration of Parallel-SSSP.

Another common performance issue in large networks is the skewed degree distribution. A small fraction of the graph vertices have millions of neighbors. In a cluster environment,
computers that are responsible for processing high-degree vertices take longer to complete. This is called the straggling task problem. We describe a special optimization technique to handle such vertices exclusively using a special partitioning function. This optimization helps distributing the extra load evenly among all cluster nodes and balances task execution times.

The Path Crawler algorithm is described to approximate shortest paths in large batches. Unlike previous landmark-based methods that are designed for fast approximation of a small number of queries, our approach focuses on approximating billions of shortest paths in parallel. We eliminate the scalability bottlenecks of existing algorithms that assume the entire pre-computation data as well as the input graph can fit in memory. Our technique uses more pre-computation data and involves efficient storage and partitioning techniques which allow each compute node to work on a different set of vertex pairs independently. In the next chapter, we describe methods for parallelizing the Path Crawler algorithm on commodity clusters.
CHAPTER 4
PARALLEL SHORTEST PATHS AND APPROXIMATING CENTRALITY

4.1 Overview

In this chapter, we discuss efficient parallelization schemes for the PathCrawler algorithm and analyze the communication cost of alternative parallel approaches. Next, we use the parallel distance estimation algorithm for computing the closeness and betweenness centrality metrics in parallel. We show how these algorithms scale with increasing input and cluster sizes and finally present interesting results from large real-world datasets.

4.2 Parallel All Pairs Shortest Paths

The PathCrawler algorithm can be run in parallel to find an approximate solution for APSP. We assume the graph is connected and undirected so \( d(i, j) = d(j, i) \) \( \forall (i, j) \in V \). Note that regardless of the implementation efficiency, the lower bound for processing \( \binom{n}{2} \) pairs of vertices is \( \Theta(n^2) \). In this section, we present parallelization techniques and explain how to minimize the communication cost between cluster nodes which has a significant effect on performance.

The input is the result of running Parallel-SSSP for all elements of \( L \). Each vertex is coupled with a compact output record consisting of shortest paths from the landmarks. Let \( r_i \) be the size of the record associated with vertex \( i \). The total input size is given by \( T(n) = \sum_{i=1}^{n} r_i \). For simplicity, we assume \( T \) grows as \( O(n) \). That is, \( |T(n)| \leq |n| c \) for some \( c \in \mathbb{R}_{>0} \) so the total input size of Parallel-APSP is \( O(n) \).

The number of reduce tasks required for a MapReduce job is \( R \). In general, the rule of thumb is to choose \( R \) large enough to ensure each compute node executes one or more reducers to increase the amount of parallelism. In a balanced setting, each reducer is expected to
process $O(n^2)$ vertices. Finally, a reducer is assumed to have enough physical memory to buffer $\mu$ vertex records on average where $\mu < n$.

### 4.2.1 Naïve Parallel Implementation

A simple parallel algorithm for solving APSP would be to generate all pairs of vertices $(i, j)$ for $0 < i < j \leq n$ in mappers, and send them to reducers. Each vertex must be replicated $n - 1$ times inside the map function and sent over the network. The communication cost of this algorithm is $O(n^2)$. Figure 4.1 shows the replication factor and record distribution among the reducers for a sample graph. Blue squares correspond to vertex pairs sent from mappers to reducers. Each vertex is replicated five times and the pairs are distributed among three reducers. In general, a graph with $n$ vertices results in an intermediate output with $\binom{n}{2}$ pairs.

![Figure 4.1: Map output for a small graph with six vertices.](image)

In a cluster environment, the interconnect speed is limited and network bandwidth is shared by compute nodes. Transferring a record over the network usually takes longer than processing it. In addition, most MapReduce implementations use the local disk as the default storage medium for intermediate records. Map outputs and reduce inputs are temporarily stored on disk and gradually fed into main memory during computation. This approach is consistent with the initial design premise of MapReduce and similar data intensive frame-
works where data size exceeds the total cluster memory. There is a performance penalty for shipping intermediate records over the network and moving them from disk into memory. Therefore, minimizing the intermediate data size can considerably improve the throughput.

4.2.2 Bucket-APSP

The Bucket-APSP algorithm is designed to avoid replicating each vertex \( n - 1 \) times. The idea is to group vertices in buckets and process a pair of buckets inside the reduce function. Each vertex is assigned to exactly one bucket. Reducers then compute an approximate shortest path between all pairs that contain one vertex from each bucket. Let \( b \) denote the number of buckets for the set \( \{ B_0, ..., B_{b-1} \} \). Each bucket contains \( \frac{n}{b} \) vertices. A pair of buckets is labelled by a unique key \((B_i, B_j)\) such that \( 0 \leq i \leq j < b \). Each bucket is paired with every other \( b \) buckets including itself. The total number of unique keys is \( \binom{b+1}{2} = O(b^2) \).

A reducer responsible from the bucket pair with key \((B_i, B_j)\) stores all vertices from each bucket in main memory and computes the approximate shortest paths for each vertex pair \((u, v)\) such that \( u \in B_i \) and \( v \in B_j \). Figure 4.2 shows a sample graph with six vertices and three buckets where each bucket contains two vertices. There are three buckets and six reducers. Each bucket contains two vertices. Blue squares correspond to pairs of buckets processed by the same reduce task. Observe that each vertex is replicated only three times and the communication cost is lower than the naïve implementation. Reducers \( R_1 \), \( R_3 \) and \( R_5 \) only contain a single bucket. Such reducers are called mono-reducers and they compute all shortest paths within a single bucket.

The Bucket-APSP algorithm uses a fast data structure called the BucketMap. It is a hash map that associates a bucket with a list of bucket pairs that it participates in. That is, given a bucket \( B_i \), \( \text{BucketMap}[B_i] \) returns an ordered list of bucket pairs \( B_xB_y \) such that \((0 \leq x \leq y < b)\) and \((x = i \lor y = i)\). For example, \( \text{BucketMap}[B_1] \) would return the ordered list \( \{B_0B_1, B_1B_1, B_1B_2\} \) for the sample graph in Figure 4.2.

Algorithm 4 shows the pseudo-code for Bucket-APSP. Building and storing BucketMap
Figure 4.2: Map output of Bucket-APSP for the same graph in the previous figure.

takes $O(b^2)$ time and space (lines 3-10). The input to MAP is the compact binary record that contains the list of shortest paths from a vertex $v$ to the set of landmarks $L$ computed by Parallel-SSSP. In line 12, $v$ is assigned to one of the $b$ buckets identified by $b_v$ and a copy of $v$ is sent to the corresponding bucket pairs. The REDUCE function operates on a pair of buckets $B_i B_j$ and all vertices that are assigned to $B_i$ or $B_j$. Once the input vertices are saved in memory (lines 20-27), a complete bipartite matching is done between the two buckets and approximate shortest paths are computed for each vertex pair via ALLPAIRS. Note that when $i = j$, only a single bucket is processed inside the REDUCE and such reducers are called mono-reducers.

Bucket-APSP has a lower replication factor than the naïve implementation. Each vertex goes exactly to one bucket. Since each bucket is replicated $b$ times, total communication cost is $O(bn)$. Although a smaller $b$ value indicates less communication, it should be chosen carefully to balance the amount of parallelism. During the reduce function, two buckets are stored in memory containing $\frac{2n}{b}$ vertices total. The maximum number of vertices that a reduce task can buffer is $\mu$. Thus, $b = \frac{2n}{\mu}$ results in the lowest communication cost. The value of $b$ can be calculated easily based on the average record size and the amount of physical
Algorithm 4 Bucket-APSP

1: Input: Graph $G=(V,E)$ with shortest paths from/to $L$.

2: BucketMap ← $\emptyset$

3: function setup($b$) $\triangleright$ generate bucket pairs
4: $\text{for } i = 0 \rightarrow b - 1 \text{ do}$
5: $\text{for } j = i \rightarrow b - 1 \text{ do}$
6: $\text{APPEND(BucketMap}[i], B_{i}B_{j})$
7: $\text{APPEND(BucketMap}[j], B_{i}B_{j})$
8: $\text{end for}$
9: $\text{end for}$
10: end function

11: function map(v.id, v) $\triangleright$ find the bucket $v$ belongs with
12: $b_v = v.id \mod b$
13: $\text{for all } B_{i}B_{j} \in \text{BucketMap}[b_v] \text{ do}$
14: $\text{EMIT(B}_{i}B_{j}, v)$
15: $\text{end for}$
16: end function

17: function reduce($B_{i}B_{j}, [v_1, ..., v_k]$)$\triangleright$ save in memory
18: $B_{i} \leftarrow \emptyset$
19: $B_{j} \leftarrow \emptyset$
20: $\text{for all } v \in [v_1, ..., v_k] \text{ do}$
21: $\text{bucket} = v.id \mod b$
22: $\text{if } \text{bucket} = i \text{ then}$
23: $\text{APPEND(B}_{i}, v)$
24: $\text{else}$
25: $\text{APPEND(B}_{j}, v)$
26: $\text{end if}$
27: $\text{end for}$
28: $\text{if } \text{Size}(B_{i}) > 0 \text{ and } \text{Size}(B_{j}) > 0 \text{ then}$ $\triangleright$ compute all pairs
29: $\text{ALLPAIRS}(B_{i},B_{j})$
30: $\text{else}$
31: $\text{if } \text{Size}(B_{i}) > 0 \text{ then}$
32: $\text{ALLPAIRS}(B_{i},B_{i})$ $\triangleright$ mono reducer
33: $\text{else}$
34: $\text{ALLPAIRS}(B_{j},B_{j})$ $\triangleright$ mono reducer
35: $\text{end if}$
36: $\text{end if}$
37: end function

38: function ALLPAIRS($B_{x},B_{y}$)$\triangleright$
39: $\text{for all } v_x \in B_{x} \text{ do}$
40: $\text{for all } v_y \in B_{y} \text{ do}$
41: $\text{sp} \leftarrow \text{PathCrawler}(v_x, v_y)$
42: $\text{EMIT(v}_{x}v_{y}, sp)$
43: $\text{end for}$
44: $\text{end for}$
45: end function
memory in a compute node. However, as $b$ decreases, the number of bucket pairs also goes down. A MapReduce job has $R$ reduce tasks and for maximum parallelism, none of them should be left idle. Ideally, the condition $(\frac{b+1}{2})/R \geq 1$ should be satisfied to ensure that compute nodes have enough work to stay busy.

**Generalization to Disjoint Subset of Vertices**

Shortest paths between two disjoint subsets of vertices $V_1$ and $V_2$ can also be computed with $\text{Bucket-APSP}$. Copies of input vertices should partitioned into two sets of buckets $S_1$ and $S_2$, where $S_1$ contains the buckets for $V_1$ and $S_2$ contains the buckets for $V_2$. Then, each reducer can process a pair of buckets $B_iB_j$ such that $B_i \in S_1$ and $B_j \in S_2$. The same idea can be generalized to $k$ disjoint subsets for computing pairwise distances between vertex clusters or communities with a single pass of the input data.

### 4.3 Approximating Centrality

Computing the closeness and betweenness centrality metrics both require solving APSP. Although $\text{Bucket-APSP}$ reduces the communication cost of the naïve algorithm by a factor of $O(\frac{b}{n})$, the quadratic computational cost of computing all pairwise distances makes application to large graphs infeasible. Instead of solving the exact APSP, we approximate these metrics in large graphs by adapting a sampling technique.

#### 4.3.1 Approximate Parallel Closeness Centrality

Given a sample set of vertices $S \subset V$, we estimate all shortest paths from $S$ to $V$ using the $\text{PathCrawler}$ algorithm. The approximate closeness centrality of a vertex is then defined as the inverse of its average estimated distance to the sample dataset. Formally,
\[ c'(v) = \frac{|S|}{\sum_{v \in V} d'(v, u)} \] (4.1)

The parallel implementation of closeness centrality is fairly straightforward. The input is the same as \texttt{Parallel-APSP} and a sample dataset \( S \). We assume the sample dataset can be buffered in local memory. That is, \( |S| < \mu \). The \texttt{MAP} function simply reads the input from DFS and partitions it among \( R \) reducers. Each reducer gets \( O\left(\frac{n}{R}\right) \) vertices as its own share of input. Reducers also read \( S \) from DFS and buffer it in local memory. Therefore, the communication cost of this algorithm is \( O(n + R|S|) \). The \texttt{REDUCE} function computes the approximate shortest paths between \( S \) and \( V_j \) for \( 0 \leq j < R \) in parallel using the \texttt{PathCrawler} algorithm.

The \( |S| < \mu \) assumption is not a requirement and larger sample sizes can be handled by partitioning them and processing each partition as a separate MapReduce job. Alternatively, a similar grouping technique described in \texttt{Bucket-APSP} can be applied to run the entire computation in a single pass. In our experiments with large graphs, all sample sets were small enough to easily fit in memory.

Using this method, we can answer queries such as ”Who are the most influential politicians of USA in Twitter?” or ”What are the most popular news websites in the Middle East?” measuring influence or popularity by closeness centrality. Note that it is often good practice to limit the total input size by region or category depending on context to achieve higher throughput and better performance.

4.3.2 Approximate Parallel Betweenness Centrality

Betweenness centrality is approximated similarly. Instead of counting how many times a vertex occurs in all shortest paths, we sample a large set of vertices \( S \in V \) and compute shortest paths for each pair in \( S \) using \texttt{Bucket-APSP}. Formally,
\begin{equation}
\begin{aligned}
b(v) &= \sum_{s \neq v \neq t, S \subseteq V} \frac{\sigma_{svt}}{\sigma_{st}}, \quad s, t \in S \quad \text{and} \quad v \in V \\
\end{aligned}
\end{equation}

To increase accuracy, we slightly modify \textbf{PathCrawler} to return multiple approximate shortest paths between each \((s, t)\) pair, rather than just one. Finally, we count how many times each \(v \in V\) occurs in the obtained shortest paths with another MapReduce job.

**Suggesting a Sample Size**

Determining the right sample size is important to make fast and accurate approximations. We use a geometric progressive sampling technique [74] to calculate the right sample size when approximating centrality. Let \(n_0\) be the initial sample size and \(a\) be a constant. A schedule \(S_g\) of increasing sample sizes is defined as

\begin{equation}
S_g = a^i n_0 = \{n_0, a.n_0, a^2.n_0, ..., a^k n_0\} 
\end{equation}

At each step, the sample size is increased and approximate closeness or betweenness centralities are computed. Then vertices are sorted and ranked based on their centrality values. This process continues until there is minimum change in vertex centrality rankings between two consecutive iterations. The change in rankings is determined by computing Spearman’s rank correlation coefficient [82].

Rankings are usually more sensitive towards the lower end (less central part) of the distribution and considering the ranks of the entire input set may require a large number of iterations. For fast convergence on large datasets, we compute the rank correlation among the top \(K\) vertices where \(K\) is a user supplied parameter generally ranging from a few hundreds to thousands. Typically, highest ranked vertices are less sensitive to changes and
this allows the algorithm to converge much faster. Another alternative for rapid convergence
is to compute the fraction of overlapping vertices between two consecutive iterations. That
is, the algorithm stops if the most central $K$ vertices from the last two iterations overlap by
more than $x\%$ where $x$ is a user defined threshold.

4.4 Experimental Evaluation

4.4.1 Accuracy

Exact vs. Approximate Closeness

We first measured the accuracy of the closeness centrality algorithm in medium sized net-
works. The comparison of the true and approximate closeness centrality values are displayed
in Figures 4.3, 4.5 and 4.4. The near-linear correlation suggests that closeness centrality can
be approximated highly accurately in large graphs if pairwise distances are calculated with
small error. Approximated closeness centrality values can replace the originals in large scale
machine learning tasks such as a feature vector in a prediction algorithm.

![Figure 4.3: Correlation coefficient for Facebook: 0.9997.](image)
The accuracy experiments also reveal that the parallel closeness centrality algorithm can identify outliers. That is, vertices with very high or very low closeness centrality values can be identified easily. For example, in the Facebook dataset, there are a few vertices with low centrality scores around the (0.17 - 0.25) region that stand out from the average.
Similarly, the vertex with the largest centrality score is identified in the upper right corner. The Wikipedia dataset yielded similar outliers. There are a few vertices with very high closeness centrality scores around the (0.45 - 0.50) region. The vertex with the lowest closeness centrality value is also visible in the bottom left corner. These results indicate that our parallel closeness centrality approximation algorithm can be used to detect individual members of a large network with very high or low impact potentials. For example, detecting and removing vertices with top closeness centrality scores can reduce the speed of information dissemination in the network. An alternative application area can be advertising. Online viral advertisements can target members with the highest closeness centrality values with the intuition that a picture or a video can spread faster through such individuals.

Overlap of Exact and Approximate Vertices

We show the percentage overlap of the $X$ most central nodes between the true and estimated results where $X$ varies from 100 to 1000 in Figures 4.6 and 4.7. For example, the top 100 vertices with the highest closeness centrality values in the Facebook graph are identified with 96% overlap. As $X$ increases, the overlap ratio stays mostly above 80% in all datasets. Results from this experiment suggest that we can accurately answer social network analysis queries such as ”Who are the top 500 most popular Twitter users?” or ”What are the top 100 most central web sites with media content on the Internet?”.

The approximation algorithms are sensitive to individual rankings of each vertex. For example, we cannot accurately answer ”What is the vertex with the 100th highest betweenness centrality score?” but we can detect the top 100 vertices having the highest betweenness centrality scores with a very high overlap ratio. Finding the exact ranking of each vertex is a harder problem and the experiment results indicate that individual rankings are quite sensitive to the smallest approximation errors. In practice, exact ranking are not used commonly. Indeed, most network scientists are often interested in detecting the most important vertices in large graphs to understand how their presence and connectivity affect the data.
Figure 4.6: Betweenness Centrality: Overlap of the most central vertices in the true and estimated results.

Figure 4.7: Closeness Centrality: Overlap of the most central vertices in the true and estimated results.
dispersion in the network.

In Tables 4.1, 4.2 and 4.3, we present the exact number of vertices that overlap within the given [0,X] range for all three datasets.

Table 4.1: Facebook Centrality Overlap

<table>
<thead>
<tr>
<th>Range</th>
<th>Closeness</th>
<th>Betweenness</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0-100]</td>
<td>95</td>
<td>73</td>
</tr>
<tr>
<td>[0-200]</td>
<td>172</td>
<td>157</td>
</tr>
<tr>
<td>[0-300]</td>
<td>231</td>
<td>257</td>
</tr>
<tr>
<td>[0-400]</td>
<td>308</td>
<td>345</td>
</tr>
<tr>
<td>[0-500]</td>
<td>407</td>
<td>431</td>
</tr>
<tr>
<td>[0-600]</td>
<td>516</td>
<td>507</td>
</tr>
<tr>
<td>[0-700]</td>
<td>618</td>
<td>569</td>
</tr>
<tr>
<td>[0-800]</td>
<td>695</td>
<td>638</td>
</tr>
<tr>
<td>[0-900]</td>
<td>810</td>
<td>700</td>
</tr>
<tr>
<td>[0-1000]</td>
<td>965</td>
<td>789</td>
</tr>
</tbody>
</table>

Table 4.2: Enron Centrality Overlap

<table>
<thead>
<tr>
<th>Range</th>
<th>Closeness</th>
<th>Betweenness</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0-100]</td>
<td>88</td>
<td>76</td>
</tr>
<tr>
<td>[0-200]</td>
<td>182</td>
<td>155</td>
</tr>
<tr>
<td>[0-300]</td>
<td>278</td>
<td>258</td>
</tr>
<tr>
<td>[0-400]</td>
<td>367</td>
<td>338</td>
</tr>
<tr>
<td>[0-500]</td>
<td>459</td>
<td>415</td>
</tr>
<tr>
<td>[0-600]</td>
<td>540</td>
<td>489</td>
</tr>
<tr>
<td>[0-700]</td>
<td>630</td>
<td>569</td>
</tr>
<tr>
<td>[0-800]</td>
<td>713</td>
<td>652</td>
</tr>
<tr>
<td>[0-900]</td>
<td>809</td>
<td>740</td>
</tr>
<tr>
<td>[0-1000]</td>
<td>895</td>
<td>838</td>
</tr>
</tbody>
</table>
Table 4.3: Wikipedia Centrality Overlap

<table>
<thead>
<tr>
<th>Range</th>
<th>Closeness</th>
<th>Betweenness</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0-100]</td>
<td>71</td>
<td>80</td>
</tr>
<tr>
<td>[0-200]</td>
<td>158</td>
<td>168</td>
</tr>
<tr>
<td>[0-300]</td>
<td>242</td>
<td>239</td>
</tr>
<tr>
<td>[0-400]</td>
<td>340</td>
<td>332</td>
</tr>
<tr>
<td>[0-500]</td>
<td>423</td>
<td>408</td>
</tr>
<tr>
<td>[0-600]</td>
<td>526</td>
<td>502</td>
</tr>
<tr>
<td>[0-700]</td>
<td>625</td>
<td>584</td>
</tr>
<tr>
<td>[0-800]</td>
<td>732</td>
<td>679</td>
</tr>
<tr>
<td>[0-900]</td>
<td>840</td>
<td>774</td>
</tr>
<tr>
<td>[0-1000]</td>
<td>938</td>
<td>877</td>
</tr>
</tbody>
</table>

4.4.2 Performance and Scalability

We studied the performance and scalability characteristics of the centrality approximation algorithms with two experiments. In the first experiment, we fixed the number of compute nodes to 20 and increased data size to understand how the system scales with larger datasets. In the second one, we fixed the data size and increased the number of compute nodes from four to twenty in order to measure the effects of increasing cluster size on run time.

Increasing Input Size

Figures 4.8 and 4.9 shows how the parallel closeness and betweenness centrality algorithms scale with increasing input size. For betweenness centrality, we sampled 30000 vertices and computed up to 250 shortest paths between each pair in the largest setting. This resulted in approximately 450 million pairs of vertices. We calculated a maximum of 63 and 51 billion shortest paths in the WWW and Twitter graphs respectively. For closeness centrality, we sampled the top one million vertices with the highest degree and ranked them based on the approximated values. In the largest setting, we calculated the approximate distance between one billion vertex pairs in 11.5 hours. Observe that both algorithms scale almost linearly with increasing input.
Figure 4.8: Betweenness Centrality in billion-scale: Change in run time with increasing input size.

Figure 4.9: Closeness Centrality in billion-scale: Change in run time with increasing input size.
We measured an average system throughput of 17600 pairs/second for betweenness and 20300 pairs/second for closeness centrality which includes the overhead to start a MapReduce job and initialize all tasks. Note that the betweenness centrality algorithm has lower throughput because for a given pair of vertices, it computes up to 250 approximate shortest paths whereas the closeness centrality algorithm only computes a scalar distance estimate for each pair.

Increasing Cluster Size

The scalability of the closeness and betweenness centrality algorithms with increasing number of compute nodes is shown in Figures 4.10 and 4.11. Higher number of hardware resources results in increased system throughput and faster application run time with linear scalability. The experiment results indicate that on large production clusters with thousands of commodity machines, the proposed centrality algorithms will be able to handle billion-scale graphs within minutes. On the other hand, the suggested algorithms are also successful at processing massive graphs on small clusters at the expense of more computation time. Observe that for betweenness centrality, we were able to calculate approximate shortest paths between 50 million pairs of vertices using only four nodes in less than 4 hours. Similarly for closeness centrality, the approximate distance between 100 million pairs of vertices were calculated in about 7.5 hours on a four node cluster.

In comparison with the existing landmark-based methods, the parallel distance estimation algorithms described in this dissertation do not rely on the main memory and the pre-computation data is stored entirely on disk. Consequently, a large number of pairwise distance approximations can be carried out even on a small cluster with a few compute nodes given enough time. In addition, the single node performance of PathCrawler is significantly faster than previous methods. The average distance estimation time for a pair of vertices inside a single computer is under 8 milliseconds for both input graphs. For comparison, the Landmark-BFS algorithm is reported to estimate the distance between a pair of vertices on
Figure 4.10: Betweenness Centrality in billion-scale: Change in run time with increasing hardware resources.

Figure 4.11: Closeness Centrality in billion-scale: Change in run time with increasing hardware resources.
a comparable Twitter graph in 889 milliseconds. Thus, our approach yields over two orders of magnitude improvement for a single compute node. The BucketAPSP algorithm runs PathCrawler in parallel where each compute node can work independently on different pairs. These characteristics of our methods allow handling arbitrarily large datasets efficiently on big production clusters. Finally, we report the top 10 vertices with the highest betweenness and closeness centralities from the Twitter and WWW datasets in Tables 4.4 and 4.5.

Table 4.4: Closeness Centrality

<table>
<thead>
<tr>
<th>Twitter</th>
<th>WWW-2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarackObama</td>
<td>google.com</td>
</tr>
<tr>
<td>Ashton Kutcher</td>
<td>yahoo.com</td>
</tr>
<tr>
<td>Ellen DeGeneres</td>
<td>dmoz.org</td>
</tr>
<tr>
<td>CNN Breaking News</td>
<td>rambler.ru</td>
</tr>
<tr>
<td>Oprah</td>
<td>hypermail.org</td>
</tr>
<tr>
<td>Ryan Seacrest</td>
<td>macromedia.com</td>
</tr>
<tr>
<td>Britney Spears</td>
<td>refdesk.com</td>
</tr>
<tr>
<td>SHAQ</td>
<td>wunderground.com</td>
</tr>
<tr>
<td>Twitter</td>
<td>commondreams.org</td>
</tr>
<tr>
<td>@shravandude</td>
<td>blogger.com</td>
</tr>
</tbody>
</table>

Table 4.5: Betweenness Centrality

<table>
<thead>
<tr>
<th>Twitter</th>
<th>WWW-2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ashton Kutcher</td>
<td>google.com</td>
</tr>
<tr>
<td>CNN Breaking News</td>
<td>yahoo.com</td>
</tr>
<tr>
<td>Barack Obama</td>
<td>dmoz.org</td>
</tr>
<tr>
<td>Britney Spears</td>
<td>rambler.ru</td>
</tr>
<tr>
<td>Ellen DeGeneres</td>
<td>hypermail.org</td>
</tr>
<tr>
<td>Twitter</td>
<td>dmoz.org</td>
</tr>
<tr>
<td>SHAQ</td>
<td>careerbank.com</td>
</tr>
<tr>
<td>Oprah</td>
<td>adobe.com</td>
</tr>
<tr>
<td>Ryan Seacrest</td>
<td>sourceforge.net</td>
</tr>
<tr>
<td>Lance Armstrong</td>
<td>phpbb.com</td>
</tr>
</tbody>
</table>
In Bucket-APSP, the number of buckets should be chosen carefully to maximize cluster utilization and minimize the run time. The communication cost of the algorithm is $O(bn)$ where $b$ is the number of buckets. Intuitively, as $b$ decreases the run time of the algorithm is expected to go down since less intermediate data is generated. However, if the total number of unique bucket pairs is less than the number of available CPU cores, the cluster utilization goes down. That is, in general $\binom{b+1}{2}$ should be a multiple of $R$ where $R$ is the total number of reduce tasks. This ensures that each reduce task receives multiple bucket pairs as input. Increasing $b$ can have a positive effect on the run time performance since it results in higher record granularity i.e., more even distribution of vertex pairs among the reduce tasks. On the other hand, very large $b$ values can quickly inflate the intermediate data and reduce the performance. Among the possible values that $b$ can take, there is usually a sweet spot which maximizes cluster utilization without excess amounts of network communication and disk writes.

![Figure 4.12: Number of buckets vs. run time and intermediate data in Bucket-APSP](image)

We plot the change in run time and the amount of intermediate data with varying number of buckets while approximating the distance between 10000 vertices. Figure 4.12 shows the
results for 100 million vertex pairs. As expected, the size of the intermediate data increases with larger $b$ values although the run time varies depending on the cluster utilization. Observe that increasing $b$ from 10 to 80 results in a noticeable performance improvement. This is because more CPU cores are utilized and the total work is distributed evenly among the reduce tasks. As the number of buckets goes beyond 200, generating and shuffling excessive amounts of intermediate records (bucket pairs) starts to dominate the total execution time of the algorithm and affects the performance negatively.
4.5 Summary

We described alternative implementations for running Path Crawler in parallel to solve the all pairs shortest path problem on commodity clusters. A naïve approach simply generates all pairs and distributes them among the compute nodes. In a cluster environment, the network interconnect is shared by all computers and the total bandwidth is limited. Generating and sending all pairs through the network is not efficient and this approach cannot handle billion-sized networks.

The Bucket-APSP algorithm is designed to tackle this problem by grouping vertices in buckets and approximating shortest paths between all possible bucket pairs. This method has a lower replication factor and generates much smaller intermediate data. The number of buckets is a critical parameter for performance and it can be chosen based on the number of computers in the clusters as well as the size of the input graph.

The Path Crawler algorithm is further used to approximate the closeness and betweenness centrality metrics in large networks. Due to the quadratic lower bound of the all pairs shortest path problem, we propose sampling-based methods to approximate these metrics for very large graphs. Given a sample set of size \( S < n \), the basic principle is to reduce the number of approximated pairs from \( O(n^2) \) to \( O(S^2) \) for betweenness and \( O(nS) \) for closeness centrality. We suggest progressive sampling methods to determine the appropriate sample size which can vary depending on the characteristics of the input graph.

Experiments show that our centrality approximation algorithms scale linearly with the number of vertex pairs. Furthermore, adding more computers to the cluster results in a linear increase in the system throughput. On a 20-node compute cluster, we were able to calculate approximate closeness and betweenness centrality metrics in real-world graphs with billions of edges.
CHAPTER 5
CLOSING REMARKS

5.0.1 Future Research Directions

The optimization techniques described for Parallel-SSSP are generalizable to other distributed graph algorithms and frameworks that are vertex-centric. The fat-vertex optimization can be used on any large graph that exhibits skewed degree distribution where the performance is affected by a small number of high-degree vertices. This is a common problem in graph processing as the degree distribution of many real world graphs follow power laws. The Selective Push optimization can be used to target a particular subset of vertices with higher priority for fast convergence of iterative graph algorithms. Recent examples to such algorithms include computing the incremental PageRank, identifying connected components and graph-based label propagation for recommendation systems using prioritized iterative computations [88].

Parallel distance estimation has many applications in graph mining. The neighborhood function \( N(h) \) of a graph is defined as the number of vertex pairs that are within \( h \) hops of each other [30]. The definition is extended further for individual nodes and subgraphs [72]. The individual neighborhood function \( IN(u, h) \) of a vertex \( u \) is defined as the number of nodes that are reachable from \( u \) in at most \( h \) hops. For small graphs, \( N(h) \) can be computed by summing the individual neighborhood function over all \( u \in V \). Given two subsets of graph vertices \( V_1 \) and \( V_2 \), the generalized neighborhood function \( N(h, V_1, V_2) \) gives the number of pairs between \( V_1 \) and \( V_2 \) that are within distance \( h \) or less. Approximate neighborhood functions have been used previously for measuring vertex importance, detecting subgraph similarity and clustering subgraphs in large networks. Our methods can be used to compute exact or approximate values for these functions in billion-scale graphs. Exact answers to \( IN(u, h) \) can be calculated efficiently with Parallel-SSSP without the requirement to store the entire graph structure in cluster memory. Fast approximations to
$IN(u, h)$ can be provided by running **Path-Crawler** in parallel, eliminating the need for a complete graph traversal from arbitrary $u$. The **neighborhood function** $N(h)$ can be approximated by sampling a set of vertices, summing the exact or approximate individual neighborhood functions for each element of the sample set and extrapolating the results. Finally, the **Bucket-APSP** algorithm can be used to estimate the **generalized neighborhood function** by computing all pairwise distances between $V_1$ and $V_2$.

Centrality has several use cases for measuring the structural importance of a vertex. It can be used for identifying critical hubs in a road network, detecting vulnerabilities in router networks or finding the most influential people in social networks. Another application area of centrality is graph clustering and community detection. Communities are dense subgraphs that are loosely connected with each other. Previously, closeness and betweenness centralities have been used as building blocks of the k-medoids and Girvan-Newman algorithms for clustering medium sized networks [55, 40, 75]. The parallel algorithms we presented for approximating centrality can be incorporated into similar techniques to cluster massive graphs using cheap commodity hardware.
5.0.2 Conclusions

We have motivated, described, and evaluated scalable and efficient algorithms for approximating distance and centrality on shared nothing architectures. The algorithms proposed in this dissertation are implemented and packaged together as a Java library called Hoba. Hoba runs on top of Hadoop and can handle graphs with hundreds of millions of nodes and billions of edges using a small cluster.

To the best of our knowledge, there is no work on batch computation of shortest paths in large graphs. We described methods for optimizing the single source shortest path and all pairs shortest path problems in MapReduce, and suggested a novel parallel algorithm for approximating shortest paths in shared nothing architectures. We used these algorithms to approximate closeness and betweenness centrality metrics and identified the vertices with top centrality scores in the largest publicly available real-world graphs. All algorithms described in this dissertation can run on weighted and directed graphs with minor modifications. Further research direction includes approximating neighborhood functions in massive networks and graph clustering using the proposed methods for distance and centrality estimation.

In a single compute node, we can approximate the distance between a pair of vertices over 100 times faster than existing methods that rely on a central server with large amounts of memory. The biggest contribution of the suggested distance estimation algorithm is scalability which allows massive parallelism. Each compute node can work independently on a different subset of the input to avoid potential bottlenecks that would normally be inevitable using the previously suggested centralized approaches. Furthermore, our methods are designed to scale horizontally with increasing cluster resources. Experiment results show that the system throughput increases linearly with the number of computers. On a small cluster with 20 compute nodes, we were able to find 63 billion approximate shortest paths in six hours.

The current version of Hoba is open source and it is compatible with the Amazon Elastic MapReduce service. It allows any network scientist to mine billion-scale graphs for modest
costs. Our methods also avoid the overhead of using high-end servers that are expensive and hard to maintain. We encourage using the suggested algorithms and utilizing the library for social network analysis and web mining.
REFERENCES


[67] Marc Maier, Matthew Rattigan, and David Jensen. Indexing network structure with shortest-path trees. ACM Transactions on Knowledge Discovery from Data, 5(3), 2011.


