Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Approximating Distributed Graph Algorithms

Michael Schramm

Course of Study: Informatik

Examiner: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Supervisor: Dr. rer. nat. Sukanya Bhowmik

Commenced: October 3, 2018

Completed: April 3, 2019

Abstract

The interest in the ability of processing data that has an underlying graph structure has grown in the recent past. This has led to the development of many distributed graph processing systems. But these existing graph processing systems take several minutes or even hours to execute popular graph algorithms. The amount of data is also growing fast, for example, the world wide web or social graphs. This leads to the question: do we always need to know the exact answer for a large graph?

In other fields like big data analytics, approximation gained interest in recent time, the user can decide about the accuracy of the result and if the user accepts a less accurate result the calculations could be speed up. Also, for distributed event based systems, such as publish/subscribe, and stream processing systems approximation techniques exists. For distributed graph processing exists only a few approaches that provide approximation techniques. Most of these approaches focus on sparsification of the graph or approximation of the vertex function itself. But the bottleneck in distributed graph processing arises mainly from the message passing between vertices.

This thesis, investigates message dropping for the Page Rank algorithm. Two ways of message dropping are investigated, individual dropping of messages based on message properties and dropping of all messages from selected edges (edge sampling). The dropping aims to reduce the runtime, while minimizing the error. Both approaches are tested with different properties. A detailed analysis of the results of both approaches and the different properties is presented. The evaluation is conducted on three real world graphs. The error metrics used for the evaluation are also described in this thesis.

Contents

1 Introduction **15**
1.1 Thesis Organization 17

2 Background **19**
2.1 Distributed Graph Processing 19
2.2 Algorithms 19

3 Related Work and Problem Statement **23**
3.1 Graph Processing Systems 23
3.2 Approximation Analytics 24
3.3 Problem Statement 26

4 System Model and Preliminaries **29**
4.1 Partitioning 29
4.2 GrapH 30
4.3 Page Rank Implementation 32

5 Approximation Algorithms **35**
5.1 Edge-Removing-Approach 35
5.2 Value-Based-Approaches 40

6 Error Metrics **45**
6.1 Page Rank 45
6.2 Single Source Shortest Path 46
6.3 Communities 46
6.4 Graph Coloring 47

7 Analysis and Results **49**
7.1 Experimental Setup 49
7.2 The Data 49
7.3 Analysis and Results 50

8 Conclusion and Outlook **59**
8.1 Outlook 59

Bibliography **61**

List of Figures

2.1 Structure of GrapH after partitioning . . . 20

3.1 Source and target vertex . . . 26

4.1 Execution pipeline . . . 29
4.2 Properties of the eu-2015-tpd graph . . . 30
4.3 Shaded vertices are ghosts and slaves respectivley [GLG+12] . . . 31
4.4 Local sum and overall sum at the example Page Rank (PR) . . . 31
4.5 Example of the pulling of the messages . . . 32
4.6 Messages between the master and its slaves . . . 33

5.1 Example graph edge removing . . . 37
5.2 Intervals edge removing . . . 38
5.3 Example graph value-based . . . 41
5.4 Intervals value-based . . . 42

7.1 Comparison out-properties . . . 51
7.2 Comparison with and without vertex-update . . . 52
7.3 Comparison of inOut-property with other approaches . . . 53
7.4 Comparison of the edge-removing-approaches . . . 54
7.5 Comparison of outIn-property against the value-based-approach . . . 55
7.6 Speedup . . . 56

List of Tables

7.1 Properties of the used graphs 50

List of Algorithms

5.1 Edge removing approximation algorithms 39
5.2 Value-based approximation algorithms 43

List of Abbreviations

CDF cumulative distribution function. 37

GAP Graph Analytics by Proximation. 16

GAS Gather Apply Scatter. 23

HDFS Hadoop Distributed File System. 24

HDRF High Degree Replicated First. 29

PR Page Rank. 7

RDD Resilient Distributed Datasets. 25

SSSP Single Source Shortest Path. 16

TCAM Ternary Content Addressable Memory. 15

TPD top private domains. 49

UDF User Defined Function. 25

1 Introduction

In the recent past one could observe a growing interest in graph-based complex-data processing. Many sources support this fact, for example a ranking among databases revealed that graph databases have grown in popularity by over 500% in the last few years alone [And15]. In 2000, the web graph already had 2.1 billion vertices and 15 billion edges [LP01]. The internet has grown in the last years, the number of web pages listed by google rose to 64 billion web pages [Kun19]. This development continues.

Research in genome sequencing tries to solve the genome assembly problem by constructing, simplifying, and traversing the de Brujin graph of the read sequence [ZB08]. The graphs reach sizes of 4^k vertices where k is at least 20. There are more examples which show the interest in graph data and also in handling big amounts of data like simulation grids, Bayesian networks, and social graphs [MCF+11][PNS14] [SHK14]

The growing use of graph-structured data and the rising amount of data for many applications fueled the need to scale out graph processing to multiple machines. Data scientists seek for two major goals: The first is to reduce the runtime of graph processing from days and hours down to minutes and the second is to be able to process graphs with up to trillions of vertices and edges. Therefore two big challenges arise. One is the increasing runtime complexity of many graph algorithms, for example, graph clustering, information propagation, and subgraph matching [SWW+12]. The second is that the growth of the data is faster than the increase of computation power. Graphs grow faster than our capacity to process them [Mit16]. That is the reason why the ability to process large graphs will get more important in the next years.

The interest in processing big graphs with billions of vertices and trillions of edges leads to different approaches for distributed graph processing frameworks [GLG+12; GXD+14; MAB+10; MTMR18]. There are very different implementations. Two representive distributed graph processing systems are Pregel [MAB+10] and GraphLab [LBG+12]. Pregel is a bulk synchronous message passing approach which runs all vertex-programs in parallel in a sequence of so called supersteps. GraphLab is an asynchronous distributed shared memory system. The vertex-programs have shared access to a distributed graph with data stored on every vertex and edge.

Many new applications and use-cases like the Internet of Things (IoT) or social graph analytics on hypergraphs [MMB+18] [KAKS99] have the requirement of timeliness of the analysis results. This is even a must for actionable analytics [19]. The existing distributed graph processing frameworks provide exact answers to the given algorithms. This results in long runtimes up to several minutes or even hours to provide an answer even for relatively small graphs. Many applications could benefit from a rough answer. For example, if the application needs to find patterns, it's often good enough to know roughly how often a pattern occurred. The existing graph processing systems, mentioned above, don't support approximated results.

In other fields like big data analytics, approximation gained interest in recent time [AMP+13] [AHR+14] [GBNN15], the user can decide about the accuracy of the result and if the user accepts a less accurate result the calculations could be speed up. BlinkDB [AMP+13] uses stratified sampling to generate samples and then chooses samples to satisfy the query budget. GRASS [AHR+14] mitigates stragglers in approximation jobs, by starting copies for slow tasks. Many approximation proposals base on the key to use only a small dataset. Some of these approximations kill tasks selectively in order to achieve the desired latency bound. But non of these approaches consider complex iterative workloads like distributed graph processing. Also, for distributed event based systems, such as publish/subscribe [BTG+18] [BTGR16] [BTBR17] approximation techniques exists. One of these approaches adapts filters in a way that the traffic slightly impacts quality of result in order to perform hardware filtering to achieve line-rate forwarding of events, but saves Ternary Content Addressable Memory (TCAM) entries. However hardware filtering has it's own limitations [TKBR14] like table size, fixed header fields, etc. and this demands approximation techniques. Also in stream processing systems approximation approaches exists [QCB+17] [TÇZ07] [TÇZ+03]. For example, the queries are distributed on multiple servers. If one server is overloaded, excessive load has to be shedded to hold the latency bound. The shedding decisions must be well coordinated because of the dependencies between the queries. In [TÇZ+03] they focus on dropping with probabilistic behavior, naming them Random Drop and Window Drop. Random Drop discards individual tuples based on a drop probability, while Window Drop does so for whole windows. The probability is calculated with linear programs based on statistics and metadata of the queries. These calculations are done while the system runs. If a node detect an upcoming overload, it contacts its parent nodes and these nodes drop load with the precalculated shedding plan.

In one of the first papers about approximating distributed graph processing [IPV+18], they use the recent advancements in spectral sparsification theory [ST08] to reduce the graph size significantly. The decision to keep an edge between a and b is done by a probability function which is adaptable with a parameter. They built a system called Graph Analytics by Proximation (GAP) with these techniques including a machine learning mechanism to choose the best sparsification parameter to gain a user given runtime- or accuracy-bound. They want to achieve this by building a model with a set of standard graph algorithms running on a set of representative graphs. For new workloads they want to analyze the properties of the given algorithm and graph and decide on the best sparsification parameter with the existing model. The first results show that approximation and speedup on distributed graph systems is possible.

This thesis brings these approaches into the context of distributed graph processing. There are many options to approximate distributed graph algorithms. Vertex skipping means that some vertices don't calculate the vertex function whereas edge sampling means that edges will be removed from the graph before calculating the algorithm. Lastly, messages dropping is discarding messages from one vertex to another. It is possible to define all these techniques as message dropping. Vertex skipping defined as message dropping denotes dropping all messages from and to this vertex. The goal is to find properties we can use for groups of algorithms to approximate them. The user can define a dropping rate or a latency bound and the system calculates which approximation techniques can fulfill these requirements and will produce the best result.

This thesis investigates message dropping in two ways, individual dropping of messages based on message properties and dropping all messages from selected edges (edge sampling). The dropping aims to reduce the runtime, while minimize the error. We identify error metrics for different algorithms, such as PR, Single Source Shortest Path (SSSP), communities and graph coloring, to

evaluate the results and compare the different approximation approaches. We look into different properties of the graphs and implement approximation based on edge sampling as well as two versions using message dropping based on message properties. In edge sampling we decide at the beginning of the algorithm on the properties of the graph which edges are suitable to remove for the algorithm. Message dropping based on message properties means we drop the number of messages given by the dropping rate in every step of the calculation. We implement these approximations in GrapH, a distributed graph processing system and evaluate these approaches with the identified error metrics and measure the runtime to show the speedup of the approximations. We run the evaluation on three different real world graphs, with sizes from 5 million up to 170 million edges.

1.1 Thesis Organization

The remaining part of the thesis is organized as follows:

Chapter 2 provides the background that is necessary to understand distributed graph processing systems. It introduces the concepts of distributed graph processing systems and algorithms that can be calculated by them.

Chapter 3 presents some of the existing research work relevant to the subject of study of this thesis. It particularly gives a brief overview of existing distributed graph processing systems and existing approximation techniques. This chapter also provides a formal specification of the problem statement.

Chapter 4 gives a description of the distributed graph processing system used in this thesis. It also describes the used partitioning technique to partition the graph onto the workers and provides a description of the implementation of the PR-implementation in GrapH.

In chapter 5 the algorithms that have been realized in this thesis have been described in details. This chapter describes the two different approaches and the properties used of these approaches.

Chapter 6 describes error metrics for different algorithms that can be calculated with distributed graph processing systems. These metrics calculate the error of an approximation against the original implementation. That can be used to evaluate the approximation algorithms.

Chapter 7 provides the results and the runtime measurements of the different approximation approaches. It also introduces the test environment used in this thesis and an analysis of the results.

Finally chapter 8 concludes this thesis with a brief summary of the work conducted. It also gives an outlook on possible future work.

2 Background

The growing interest to run algorithms on graph data, for example, web-graphs or social graphs as well as graphs in bioinformatics leads to a development of distributed graph processing systems, which can calculate algorithms on big graphs. This chapter aims to describe what distributed graph processing means and how it works. Moreover it introduces some algorithms which are often used in distributed graph processing.

2.1 Distributed Graph Processing

The interest in processing big graphs with billion of vertices and trillion of edges led to the development of distributed graph processing systems. Very different implementations exist, but all of them partitioning the data or the workload to calculate the algorithms on clusters or similar systems in parallel. A distributed graph processing system gets a graph $G = \{V, E\}$ and a vertex-program P, which is executed in parallel on each vertex $v \in V$ and can interact with neighboring vertices, if an edge exists. In contrast to more general message passing models, distributed graph processing systems sent messages over the edges only. In this thesis we use GrapH [MTMR18], a distributed graph processing system for in-memory data analytics on graph-structured data, which uses the vetrix-centric, iterative computation model also used by PowerGraph [GLG+12]. It will be described in more detail in chapter 4.2.

Figure 2.1 shows such a distributed graph processing system. The graph is partitioned into subgraphs and every worker calculates the vertex function on the vertices in its subgraph. In this example, the partitioning principle is vertex-cut, this means that the edges are distributed and there could be more than one instance of a vertex. For every vertex a master exists and the workers exchange messages to update the data of the instances of a vertex and for synchronization of the workers during the phases. The other option and more details about partitioning are described in Section 4.1. This model is used by GrapH and many other graph processing systems.

2.2 Algorithms

The distributed graph processing systems can execute many different algorithms on graphs. This thesis concentrates on PR, but describes some more algorithms, which are implemented in the GrapH-system and often used in practice.

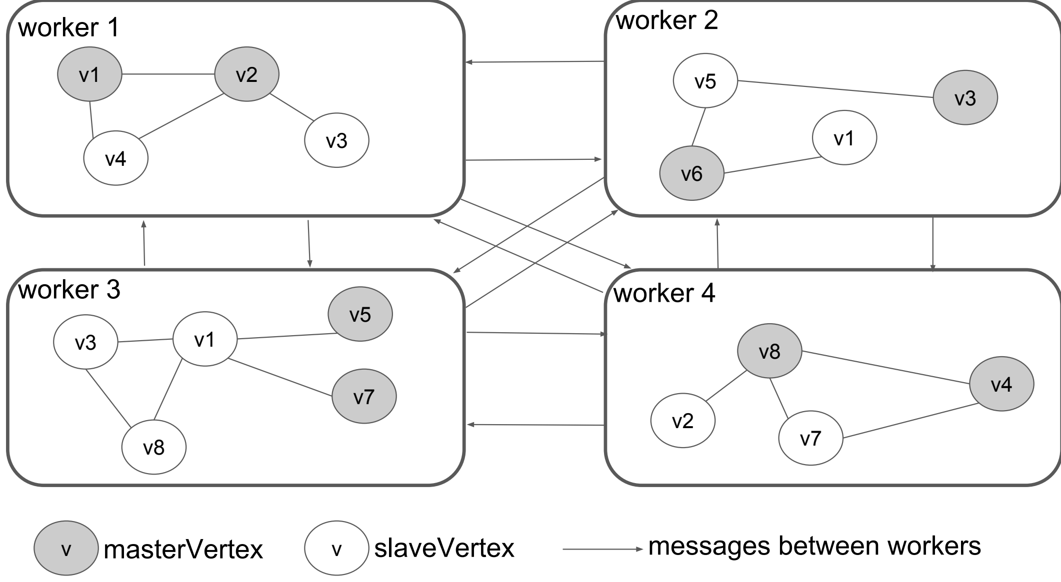


Figure 2.1: Structure of GrapH after partitioning

2.2.1 Page Rank

The PR algorithm was introduced by Sergey Brin and Lawrence Page [BP98] to calculate the importance of a web page. Thereby every website gets a rank describing the probability of a random surfer being on this website. The PR calculation is given in equation 2.1, A is a web page, d is the damping factor and $C(T)$ is the out-degree of a neighbor which has an link to the web page A. The damping factor d can be set between 0 and 1. This number represents the probability that the random surfer requests a new web page randomly. In general and also for our calculations the damping factor is set to 0.85.

$$PR(A) = (1 - d) + d(\frac{PR(T1)}{C(T1)} \dots \frac{PR(Tn)}{C(Tn)}) \tag{2.1}$$

2.2.2 Single Source Shortest Path

SSSP is a graph algorithm which calculates all shortest paths from one source vertex to every other vertex in the graph.

$$sssp(v) = \forall v_i \in V \setminus \{v\}\text{distance}(v, v_i) \tag{2.2}$$

$$distance(v_0, v_t) = \forall < v_0, v_1 \dots v_t > \min(\sum_{i=1}^{t} w(v_{i-1}, v_i)) \tag{2.3}$$

If there exists no path between v and v_i the distance is ∞. The goal of the approximation is to keep the approximated distance as similar as possible to the original distance. The approximated distance, by removing edges or deleting some update messages, can only be greater than the original distance because no shortcuts are added.

2.2.3 Communities

There are many different definitions for communities on graphs. In this section we want to describe two of these definitions, strong communities and semi-clustering.

Strong Communities

One definition of communities is called strong communities and is described in [RCC+04]. The algorithm from [RAK07] describes a similar definition. The basic quantity for the definition of strong communities in [RCC+04] is k_i. k_i is the degree of a node i, which in terms of the adjacency matrix $A_{i,j}$ of the network G is

$$k_i = \sum_j A_{i,j} \tag{2.4}$$

An adjacency matrix describes the topology of the network. The axes of the matrix consists of all nodes of the network. In an unweighted and undirected network an entry of two nodes will be 1 if there consists a direct edge between these two edges and 0 otherwise.

If we consider a subgraph $V \subset G$ to which node i belongs, we can split the total degree in two contributions:

$$k_i(V) = k_i^{IN}(V) + k_i^{OUT}(V) \tag{2.5}$$

$$k_i^{IN}(V) = \sum_{j \in V} A_{i,j} \tag{2.6}$$

$$k_i^{OUT}(V) = \sum_{j \notin V} A_{i,j} \tag{2.7}$$

With these definitions we can define strong communities as:

$$k_i^{IN}(V) \geq k_i^{OUT}(V) \qquad \forall i \in V \tag{2.8}$$

At least two subgraphs must fulfill the definition because splitting a network randomly into a very large part and a part with only a few nodes leads to the effect that the very large part almost always fulfills the definitions of strong communities [RCC+04]. As a result a subgraph $V \subset G$ is a strong community, if every node in the subgraph have more or equal edges to nodes in the subgraph than to nodes outside of V. With the difference that in [RAK07] the inside-degree will be really greater than the outside-degree:

$$k_i^{IN}(V) > k_i^{OUT}(V) \qquad \forall i \in V \tag{2.9}$$

Semi Clustering

Another clustering version, semi-clustering [MAB+10], arises in social graphs. Edges may have weights to represent the interactions' frequency or strength. With the difference that in strong communities a vertex could be part of more than one community/cluster. The input is a weighted,

undirected graph and its output is at most C_{max} semi-clusters, each containing at most V_{max} vertices, where C_{max} and V_{max} are user-specified parameters. A semi-cluster c is assigned a score:

$$S_c = \frac{I_c - f_B B_c}{\frac{V_c(V_c-1)}{2}} \tag{2.10}$$

I_c is the sum of the weights of all internal edges, B_c is the sum of the weights of all boundary edges, V_c is the number of vertices in the semi-cluster, and f_B, the boundary edge score factor, is a user-specified parameter, usually between 0 and 1. To not receive artificially high scores for large clusters the score is normalized. The exact calculation of the semi-clusters with this score in some iterations can be read in [MAB+10].

2.2.4 Graph Coloring

The aim of graph coloring is to color every vertex of a graph $G = (V, E)$ in a way that no two adjacent vertices get the same color. If at most k colors are used the coloring is named a k-coloring [HKKN04]. We want to find the smallest possible k for a certain graph. This could be described with a function $f : V \to C \subseteq \mathbb{N}_0$. The elements of C are the colors. The function is valid if the following condition is fulfilled:

$$\forall v \in V : \forall w \in \Gamma(V) : f(v) \neq f(w) \tag{2.11}$$

where $\Gamma(V)$ is the number of the neighbors of v.

3 Related Work and Problem Statement

This work is related to distributed graph processing systems and approximate analytics systems. The first part of this chapter describes which graph processing systems exists and explains the differences between them. The second part describes the related work in approximation analytics. In the third part the problem will be described formally.

3.1 Graph Processing Systems

There exists a tremendous number of distributed graph processing systems in the literature [BG11; GLG+12; GXD+14; KBG12; MAB+10; MTMR18; RBMZ15; WXDG13] describe distributed graph processing on static graphs, [CHK+12; CLS12; HML+14; MMI+13; MMMS15] focus on graph analytics on evolving graphs. There are examples for asynchronous graph processing as well as synchronous. Interaction between vertices is implemented using, for example, shared state or message passing, but all of them search for the exact solution of the given problem and none of them have approximated analytics implemented yet. Three representative distributed graph processing systems are Pregel, GraphLab and PowerGraph.

Pregel [MAB+10], a bulk synchronous message passing approach which runs all vertex-programs in parallel in a sequence of so called supersteps. In a superstep each vertex processes all messages arrived in the superstep before and send messages to its neighbors for the next superstep. The supersteps constitute barriers because no vertex starts with the calculation of the next superstep until all vertices have ended the calculation of the actual superstep. This synchronicity ensures that Pregel programs are inherently free of deadlocks and data races common in asynchronous systems. Because the graphs are distributed over the workers one should be able to balance the load so that the synchronicity doesn't add excessive latency. The program stops if all vertices have voted to stop. The calculation of the vertex functions is done by so called combiners which are user defined functions to merge the arrived messages of a vertex.

GraphLab [LBG+12] is an asynchronous distributed shared memory system, where the vertex-programs have shared access to a distributed graph with data stored on every vertex and edge. Every vertex-program can access the data of itself, of its neighboring vertices, and the adjacent edges. Vertex-programs can also start programs on neighboring vertices. To ensure serializability, neighboring vertex-programs can not run simultaneously. This is a great benefit in contrast to other asynchronous frameworks which do not ensure serializability or provide only basic mechanisms to recover from data races. GraphLab supports a broad range of consistency settings, allowing a program to choose the level of consistency needed for correctness.

PowerGraph [GLG+12] combines the best features of Pregel and GraphLab. It takes the data-graph and shared-memory view of GraphLab and borrows the commutative, associativ gather concept. In [GLG+12] they introduce the Gather Apply Scatter (GAS) concept and show that Pregel and

GraphLab working with this principle, but in very different ways. GAS consists of three phases. In the gather phase every active vertex (vertices can be active or passive) collects information about adjacent vertices and edges and calculates a generalized sum (the concrete function depends on the algorithm). Copies of a vertex can exist on different workers. In the apply phase the master of a vertex calculates the new value with the sums of the gathering phase. The scatter phase updates the adjacent edges of the vertex and activates the vertices for the next superstep.

PowerGraph is able to run asynchronously or synchronously and can emulate Pregel programs as well as GraphLab programs. It is optimized to process power law graphs. In power law graphs many vertices have a small degree and only a few have really high degrees. Power law graph are hard to partition in a way that the workload is distributed evenly. PowerGraphs introduces the balanced p-way vertex-cut to solve this problem.

These three graph processing frameworks are powerful to calculate iterative graph algorithms, like PR and can execute big graphs up to billions of vertices and edges. With the given restriction on vertex-centric algorithms on a single static graph, these systems gain a performance benfit in contrast to data-parallel systems such as Hadoop MapReduce. But this makes it difficult for these systems to express operations often found in graph analytics pipelines. For example, constructing a graph from external resources, modifying the graph structure, or merging two graphs. The data-parallel systems are well suited for those problems but the calculation of iterative graph algorithms leads to complex joins and exessive data movement. To avoid these problems existing graph analytics pipelines use external storage systems such as Hadoop Distributed File System (HDFS), but the resulting APIs are tailored to specific tasks and aren't easy to use for end-users to implement own algorithms. These problems are addressed by GraphX [GXD+14], a distributed graph computation framework which unifies graph-parallel and data-parallel computation in a single system. GraphX allows to view the same data as graph data and as tables without data movement or duplication. It allows data-parallel operators, such as map, reduce, filter, and join, as well as some graph-parallel operators.

3.2 Approximation Analytics

Approximation analytics gained a lot of attention in recent time. For example, in the big data analytics many new approaches emerged recently. BlinkDB [AMP+13] uses stratified sampling to generate samples and then chooses samples to satisfy the query budget. GRASS [AHR+14] mitigates stragglers in approximation jobs. The dominant technique to mitigate stragglers is to launch speculative copies for the slower tasks, where a speculative copy is simply a duplicate of the original task. This technique was used succsesfully in datacenters of Facebook and Microsoft. GRASS adopted them for approximation analytics. ApproxHadoop [GBNN15] enables approximation in MapReduce Systems. MapReduce is a computing model designed for processing large data sets on server clusters. To approximate MapReduce jobs they considered three strategies, input data sampling, task dropping and user-defined approximation strategies for the algorithm to calculate itself.

Also, for distributed event based systems, such as publish/subscribe [BTG+18] [BTGR16] [BTBR17] approximation techniques exists. One of these approaches adapts filters in a way that the traffic slightly impacting quality of result in order to perform hardware filtering to achieve line-rate forwarding of events, but saves TCAM entries. However hardware filtering has it's own limitations

[TKBR14] how table size, fixed header fields, etc. and this demands approximation techniques. Also in stream processing systems approximation approaches exist [QCB+17] [TÇZ07] [TÇZ+03]. For example, in distributed stream processing systems, the queries are distributed on multiple servers. If one server is overloaded, excessive load has to be shedded to hold the latency bound. The shedding decisions must be well coordinated because of the dependencies between the queries. In [TÇZ+03] they focus on dropping with probabilistic behavior, they named them Random Drop and Window Drop. Random Drop discards individual tuples based on a drop probability, while Window Drop does so for whole windows. The probability is calculated with linear programs based on statistics and metadata of the queries. These calculations are done while the system runs. If a node detects an upcoming overload, it contacts its parent nodes and these nodes drop load with the precalculated shedding plan.

Another interesting approach aims on time-evolving graphs. GraphTau [ILDS16], a graph processing system for time-evolving graphs that sort of approximates while calculating the algorithms. It can change the graph during calculation. This is only possible for algorithms resilient to graph changes, for example PR. GraphTau uses Resilient Distributed Datasets (RDD)s one of the main abstractions provided by Apache Spark, a data-parallel computation engine that supports general DAG computations on Spark Streaming, the streaming component in Spark. The approximation tasks are placed in the Pause-Shift-Resume phase. If a new edge or vertex should be added to the graph, GraphTau stops the actual calculations, shift the meta-data (i.e. actual PR values) to the so called new snapshot (the graph with the new element) and then resume the calculations. Studies have shown that the results of this approximations are within a reasonable error compared to the calculation from scratch on the actual graph.

There exists another work about approximation of graph computing, which approximates the User Defined Function (UDF) [SY14], the function implemented from the end-user on a graph processing system to calculate a algorithm, for example, PR. The approach can be used with many different graph-processing systems, it's independent from the graph problem because it approximates the UDF and is able to approximate algorithms with an exact answer as well as further approximate algorithms, with a approximate answer. The approach is able to approximate any UDF, there are no fix approximation algorithms implemented. The approximation is done automatically from the system. They use five different techniques for this purpose: Sampling, Memorization, Task Skipping, Interpolation and System Function Replacement. Sampling means the UDF is not calculated with all messages, but with a portion of it, for example, 20%. The result is calculated by estimating the result with all messages. With the 20% and a sum function the algorithm multiplies the result from the 20% of the messages with the factor 5. The second technique, Memorization, keeps results from previous executions if the input in the next superstep is identical or similar to the previous results. Task Skipping is a similar technique, but the input isn't checked. A portion of the vertices is chosen randomly, these vertices ignore the input from its neighbors and use the old result from the previous iteration. If this technique works it often works on vertices with small degree, because decisions of them are often not important to the result of the full graph. Interpolation is used, if the UDF is a non-linear numerical function. Sometimes it's possible to approximate such a function with a combination of linear functions. System Function Replacement means, using of an optimized system built-in mathematical functions, for example, for sin(). The described algorithm samples the graph and tests the different techniques in terms of error and time cost to find a good approximation for the whole graph.

In one of the first approaches that aims general approximating distributed graph processing [IPV+18], they use the recent advancements in spectral sparsification theory [ST08] to reduce the graph size significantly. The sparsifier decides to keep an edge between a and b with a probability function

$$\frac{d_{AVG} \times s}{\min(d_a^o, d_b^i)} \tag{3.1}$$

where d_{AVG} is the average degree of all vertices in the graph, d_a^o is the out-degree of vertex a, d_b^i is the in-degree of vertex b, and s is a parameter to control the level of sparsification. The function drops edges from vertices with high degree and keeps edges from vertices with low degree. With this function and a machine learning mechanism to choose the best sparsification parameter to gain a user given speedup- or accuracy-bound, they built a system called GAP. The goal they want to achieve is to build a model with a set of standard graph algorithms running on a set of representive graphs and learn which sparsification parameter fit the best. For new workloads they want to analyze the properties of the given algorithm and graph, and decide on the best sparsification parameter with the existing model. The first results show that approximation and speedup on distributed graph systems is possible.

3.3 Problem Statement

A graph G is a set of vertices V and a set of edges E. An edge is a 3-tuple consisting of a source vertex, a target vertex, and an edge weight. An edge is a connection from one vertex to another and the edges in our web-graphs are directed. An edge lead from its source vertex to its target vertex (see figure 3.1). In web graphs the weight can be defined as 1, since clicking a link is one hop to the next web page.

Figure 3.1: Source and target vertex

The GrapH2 framework used in this thesis gets an edge list as input. An edge list is a sequence of edges in the graph. Represented by the source vertex ID followed by the ID of the target vertex. The vertices result from these IDs. Generally the data sets are ordered in some way like breadth first arising from the web crawler obtaining the data set.

The vertices exchange messages to calculate the algorithms. Through these messages the neighbors of a vertex get new information and calculate new own values with the vertex function. The vertex function is the user-defined function executed by every vertex to calculate the algorithm on all vertices. This is done for some iterations until the termination condition is reached and the algorithm stops. But these messages are the bottleneck of distributed graph processing [IPV+18] and there are many applications that could benefit from approximated but faster calculations.

Because these messages are the bottleneck it is an interesting approach to examine dropping of messages and the resulting errors and runtime improvements. The goal is to enable the user to decide how accurate the result has to be or how much time are given for the calculations. This is always a trade-off between accuracy of the result and runtime improvements. The optimization goal for algorithms is that the approximated PR-values should be as close as possible to the PRs calculated without approximation. This can be expressed with the following equation:

$$\text{optimization goal error} = \min\left(\sum_{v \in V} |PR_n(v) - PR_a(v)|\right) \tag{3.2}$$

where PR_n is the „normal" PR and PR_a is a approximated PR. The algorithms aim to minimize the error for a given dropping rate.

Because the PR-algorithm is a ranking algorithm it's also an important goal to keep the ordering of the ranking as near as possible to the original ranking. Therefore, we want to minimize the number of the pairwise inversions:

$$\text{optimization goal ranking} = \min\left(\sum_{(i,j):i>j} [\sigma(i) < \sigma(j)]\right) \tag{3.3}$$

where σ is the permutation of the approximated solution. And if the order between two elements in approximated result is different to the original solution we increment the counter. The goal is to minimize this value. That is a different goal than the first because it's possible that two approximated results have the same error, but in one solution the error is evenly distributed and the ranking is the same and in the other some values have a high error and other a low value and therefore a different ranking.

4 System Model and Preliminaries

This chapter describes the system used to implement the approximations. It starts with a description of the preparation of the used real-world graphs to run our experiments. An important issue before running graph processing on distributed workers is partitioning the data. This thesis uses an algorithm that is well suited for power law graphs like web-graph data. It follows a description of used distributed graph processing system GraphH which uses the GAS principle introduced by PowerGraph. Picture 4.1 gives an overview of the different steps. In an academic paper [Gep18], the pipeline was implemented in a script. We adapted the script and used it for our experiments.

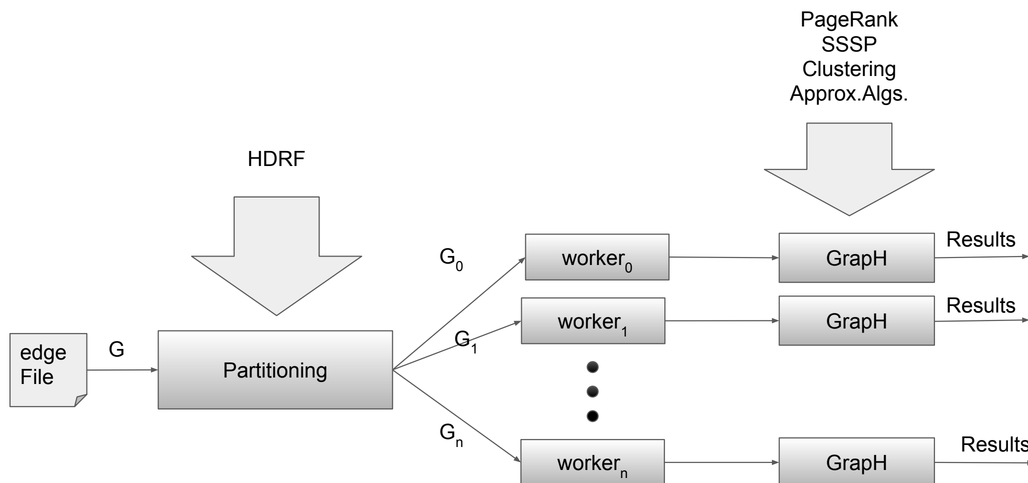
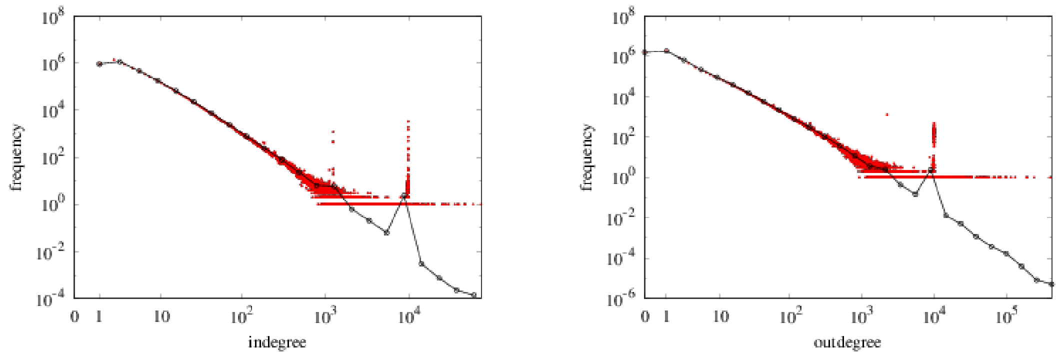


Figure 4.1: Execution pipeline

4.1 Partitioning

To compute distributed algorithms, the graph has to be partitioned into the right number of subgraphs. At the beginning, the graph-data is in different formats. The Berkley-Standford webgraph [LLDM09] and the Google web graph [LK14] are available as arc files that means every edge is represented as a line with the ids of the nodes the edge connects. The data of the third graph the eu-2015-tpd graph [BMSV14] is available in compressed form [BRSV11] and can be decompressed with the WebGraph framework [BV04] to an arc file of the graph. The partitioner needs the arc file format. For the partitioning, we used High Degree Replicated First (HDRF)[PQD+15], a stream based



(a) In-degree distribution of eu-2015-tpd graph **(b)** Out-degree distribution of eu-2015-tpd graph

Figure 4.2: Properties of the eu-2015-tpd graph

partitioner for power-law-graphs because real world graphs from social networking or the web show a power-law distribution. Power-law means most vertices have relatively few connections while some vertices have many [FFF99]. This holds for the graphs we used, too. Figure 4.2 shows the in- and out-degree frequency of the vertices of the graph (red dots). The additional line is the fibonacci binning which is recommend in [Vig13] to prove power-law-distributions.

The HDRF algorithm tries to hold strongly connected components with low degree vertices on one partition and vertices with high degree on many partitions. As mentioned before, the algorithm works stream based. Therefore the algorithm doesn't preprocess the input and only uses information from already partitioned edges. But even with this restriction it reaches better results on power-law graphs than offline algorithms like Ginger [CSCC15] and METIS [KK95] in terms of replication factor and standard deviation.

4.2 GraphH

There are some distributed graph processing systems, such as Pregel, PowerGraph, GraphX and GraphH [GLG+12; GXD+14; MAB+10; MTMR18], which we could use to develop and test approximation techniques. We decided to use GraphH because it is easy to adapt and there are no optimizations for the distributed algorithms implemented yet. GraphH is a distributed graph processing system for in-memory data analytics on graph-structured data and uses the vetrix-centric, iterative computation model also used by PowerGraph [GLG+12]. In each iteration all active vertices execute the user defined function. An iteration is called superstep like in Pregel and PowerGraph. The computation occurs synchronized, so every active vertex ends the phase before the system starts the next phase. The vertex function consists of three phases (Gather, Apply, Scatter) and works on user defined vertex data. In the gather-phase the active vertices aggregate the data from its neighbors to a gathered sum. During the apply-phase the vertices change their local data. The scatter-phase activates vertices for the next iteration.

To parallelize the execution, the graph has to distribute onto multiple workers. There are two options to partition the graph, edge-cut and vertex-cut [GLG+12]. Edge-cut distributes the vertices onto the workers, while vertex-cut distributes the edges to the partitions, so there can be the same vertex on different workers, one as master and the others as slaves. With edge-cut, there will be

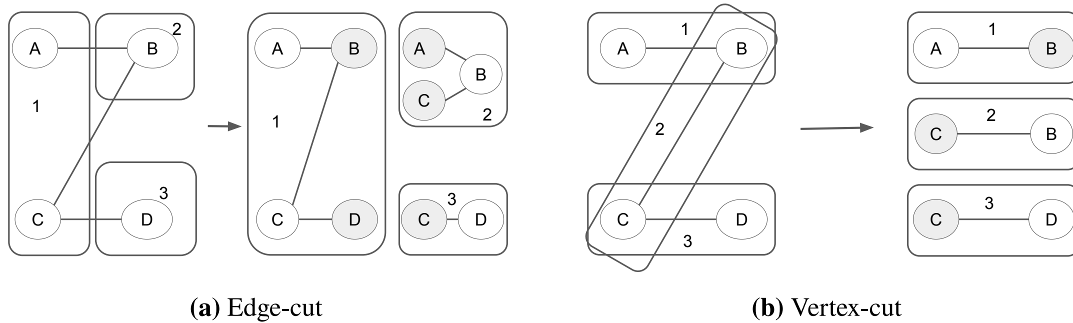


Figure 4.3: Shaded vertices are ghosts and slaves respectivley [GLG+12]

either real edges between the workers or so called ghosts, vertices which the vertices of the worker has neighbors to (see figure 4.3). With vertex-cut, communication between the workers will be communication from master to slaves of a vertex and vice versa. All workers with a replica of a vertex u are in the replica set R_u. Vertex-cut has superior partitioning properties for real world graphs, like Twitter or Facebook [GLG+12].

As a consequence, the inter-partition traffic is traffic between the replicas of a vertex or messages to keep the workers synchronized. For every vertex a master exists. It starts the distributed computation and keeps the data consistent. In the gather-phase an active master sends a RequestGather message to get the local sums of the associated slaves. The slaves of this vertex answer with a ResultGather message which contains the local sum of this slave. With the local sums, the master calculates the new value, for example, the new PR-value. This process is illustrated in figure 4.4 using the example of the PR-algorithm. The grey vertex in worker 2 is the master vertex. Also, the local sum and the overall sum, which is in this case the PR, are shown. An example for the calculation of the local sum is shown in worker 3.

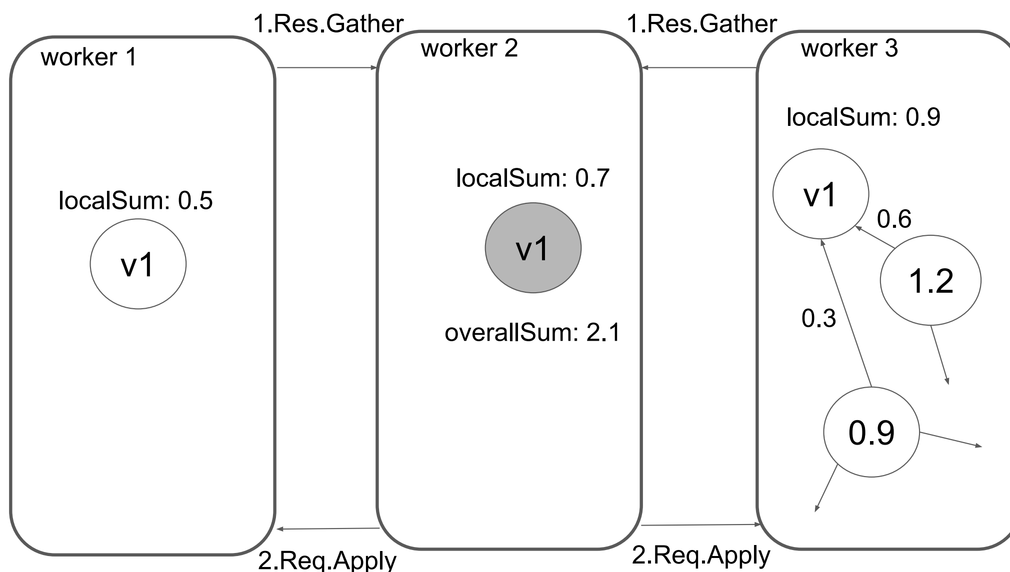


Figure 4.4: Local sum and overall sum at the example PR

At this point, it's also important to note that in GrapH the values to calculate the local sum are pulled. This means the vertex which wants to calculate its local sum gets the data of the neighboring vertex and calculates the message value. This is shown in figure 4.5. That is a remarkable difference to Pregel where these messages are actually sent. For the approximations we consider this also as messages and the dropping takes place before the gathering phase.

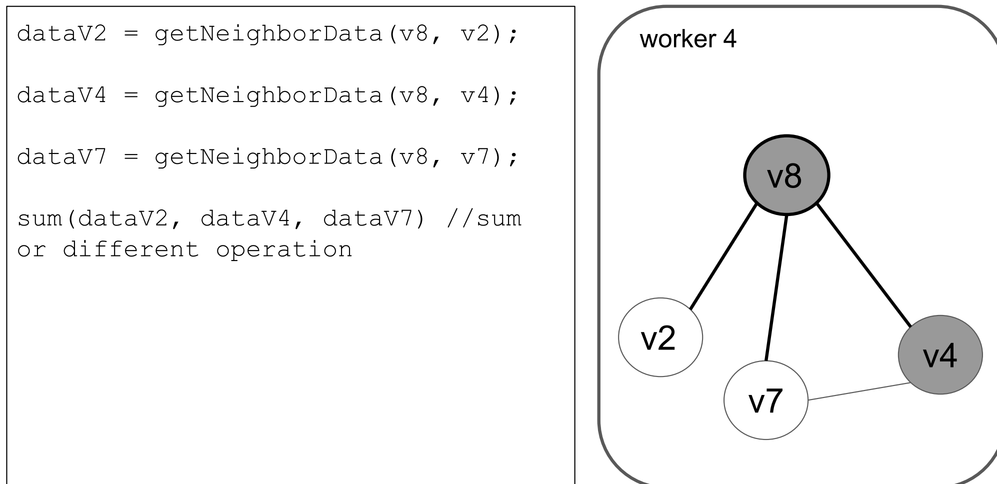


Figure 4.5: Example of the pulling of the messages

In the apply-phase the master updates the vertex data of the slaves to the new value. It sends the data and some additional information which can be used in a mode not used in this thesis to repartition the graph during calculation. The slaves answer with an acknowledgement. During the scatter phase (in the GrapH-framework named signal) master and slaves exchange signal messages to set neighbors active or not. This could be used to set only relevant vertices active if, for example, the algorithm has only effects on a part of the graph. In our implementation all vertices set active for the next iteration. Also, the algorithm will stop, if no vertices are active any more, so in our implementation no vertex is set active in the 20th superstep and the algorithm stops. Figure 4.6 shows the whole process with all message-types. Between the phases the workers exchange sync-messages to know when they can start the next phase. The single messages are not illustrated in the picture because they are not between master and slaves, but between the workers itself and not relevant for the algorithms.

4.3 Page Rank Implementation

As described above the GrapH framework based on the GAS principle and the algorithms are implemented in these three phases. This section describes the implementation of PR in the GrapH framework. The first important information GrapH needs for an algorithm is whether the gathering is done on in-edges, out-edges or both. In the case of PR, the gathering is done on in-edges, because we sum the values from the in-edges up. In the gathering phase the value of all in-edges of a vertex

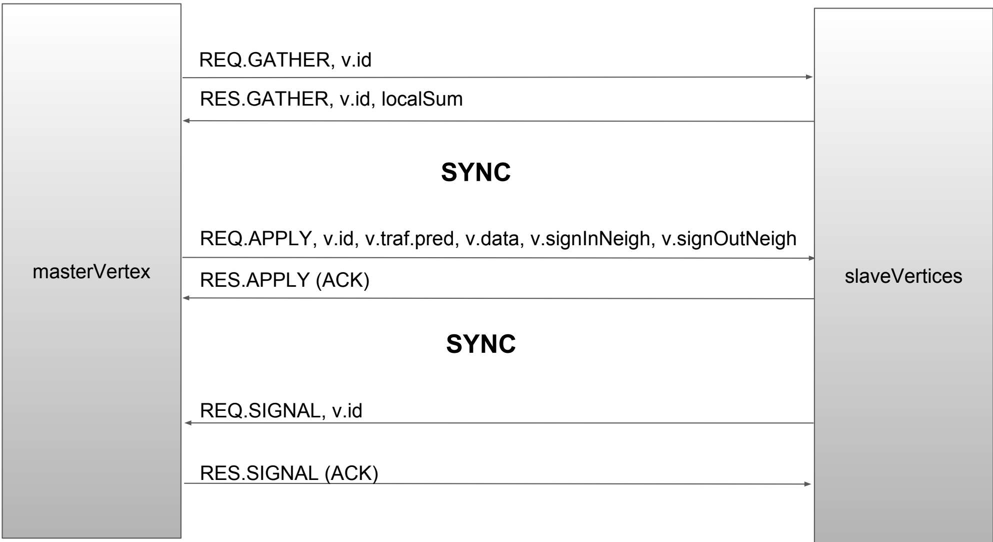


Figure 4.6: Messages between the master and its slaves

is calculated. These values are summed up with the sum-function, which is called by the gathering method of the main-class. At this point in the main-class the approximation is taking place. The master of a vertex sums up the sums from the mirrors and himself, calculates the new PR-value (in the applyScatter-function) and sends messages with the new PR to the mirrors.

5 Approximation Algorithms

This chapter provides an insight of the techniques used to approximate the PR-algorithm. There are many options to approximate distributed graph algorithms. Vertex skipping means that some vertices don't calculate the vertex function whereas edge sampling means that edges will be removed from the graph before calculating the algorithm. Lastly, messages dropping is discarding messages from one vertex to another. It is possible to define all these techniques as message dropping, so vertex skipping defined as message dropping means that all messages from and to this vertex will be dropped. Edge sampling, as message dropping, is dropping all messages over the removed edges.

The approximation work is done in the *localGather* method of the GrapH-framework. This method determines the in-edges of every vertex of the subgraph. The subgraph contains all edges which are on a worker and the vertices they connect. So at this point, the algorithm goes through all edges of the subgraph which have a target on an active vertex. For all these edges the framework calls the gather-method described in 4.3 and goes on to the apply- and scatter-method. The reason why we drop at this time of the calculation is that we save the time of all three phases and it's the time at which we see all values over the edges.

This thesis investigates edge sampling and message dropping by value on the PR-algorithm. The dropping rate θ is specified by the end-user. Messages are sent (or pulled, see chapter 4.2) between the vertices. The vertices gather the information from their neighbors and calculate their new PR-values. The dropping aims to reduce the runtime, while minimizing the error. The number of messages which are dropped can be set by the end-user with the dropping rate. The dropping rate is the percentage of messages that will be dropped during the computation of the algorithm. The two approaches are substantially different and will be described in the next sections. With both approaches we tried different properties for the algorithm.

5.1 Edge-Removing-Approach

The first approach removes edges from the graph permanently. This takes place at the beginning of the algorithm. The goal is to find properties of the edges which have an influence on the result. The properties are used to give every edge a priority. The edge removing algorithm decides based on the value of the property which edges are removed. The algorithm must delete the given percentage of messages. This percentage is the dropping rate given to the program by the end-user. To be able to do that, the algorithm has to calculate a threshold. There are some steps to do that. At first, gather the values of the used property for every edge, then calculate the threshold, and as a third step, delete all edges with a value smaller or greater (depends on the property) than the threshold.

To reach the goal to minimize the error by a given fixed dropping rate θ, the approximation algorithm needs to rate the edges from unimportant to important to be able to remove the unimportant $\theta\%$ of the edges. To rate the edges, the algorithm must use properties of the edges. To describe these

properties, we use the definitions from the problem statement of source and target vertex. In this thesis different properties were tested, these properties are described in the next part. The following description of the algorithm will use the out-property to simplify the description.

5.1.1 Out

The decision of removing is made by the number of out-edges of the source vertex of an edge.

$$\text{property} = \text{source.out-degree} \tag{5.1}$$

The reason is that this determines the portion of the PR-value sent over the edge. So, it follows that edges from vertex with high out-degree should be deleted first because these edges transport a smaller portion of the PR from their source vertex compared to other edges. To verify this, we also tried to delete edges with small out-degree at their source vertex first. In the chapter 7 this will be evaluated.

5.1.2 OutIn

This approach multiplies the out-degree of a source vertex with the in-degree of the target vertex of an edge.

$$\text{property} = \text{source.out-degree} * \text{target.in-degree} \tag{5.2}$$

As said for the out-property the out-degree determines the portion of the PR-value sent over the edge. The in-degree factor based on the idea that nodes with many in-edges suffer less from removing some of these edges. This means, with this property the goal is to remove edges first, which send a small portion of the information from their source vertices to target vertices with many in-edges and thus suffer less than vertices with less in-edges. The worst case example would be a vertex with one in-edge. That vertex would lose all information and the PR-value of this vertex would probably be totally wrong (besides its correct PR-value is 1). Edges with a high value for this property will be deleted first.

5.1.3 InOut

This approach divides the in-degree through the out-degree of the source vertex of an edge.

$$\text{property} = \frac{\text{source.in-degree}}{\text{source.out-degree}} \tag{5.3}$$

There is a correlation between PR and the in-degree of the vertex. For vertices with a high in-degree a high PR-value is more likely. Therefore the property takes that into account. The in-degree is divided by the out-degree of the vertex because this determines the portion of the PR-value sent over the edge and the idea is that this is a good estimation on how great the values over the edges are. Edges with a small property value will be deleted first.

5.1.4 Algorithm

These properties are used in the algorithm to rank the edges. In the following, for simplification, the text will describe the algorithm on the example of the out-property and remove edges with high out-degree at its source vertex first. The goal is to delete θ% of the edges, so the algorithm has to find the θ% of edges with the highest out-degree at the source vertex. The exact solution would be to sort the values and calculate the precise value to drop the given θ% of edges. But this approach would be too time consuming because of the sorting of many values. The implemented approach calculates a threshold. For this purpose a distribution of the out-degrees is needed to be able to calculate the value that marks the point where θ% of the out-degrees are higher than this threshold.

To calculate the threshold, the algorithm uses a cumulative distribution function (CDF), a function that specifies the distribution of values, in this case the out-degrees. In the first step the value range of the property is divided into intervals and the algorithm counts how many values there are in the respective interval. This is shown on an example. In the graph 5.1 the out-degree property is written for every edge. The graph has ten edges in total. The value range in this example is 1-3 and

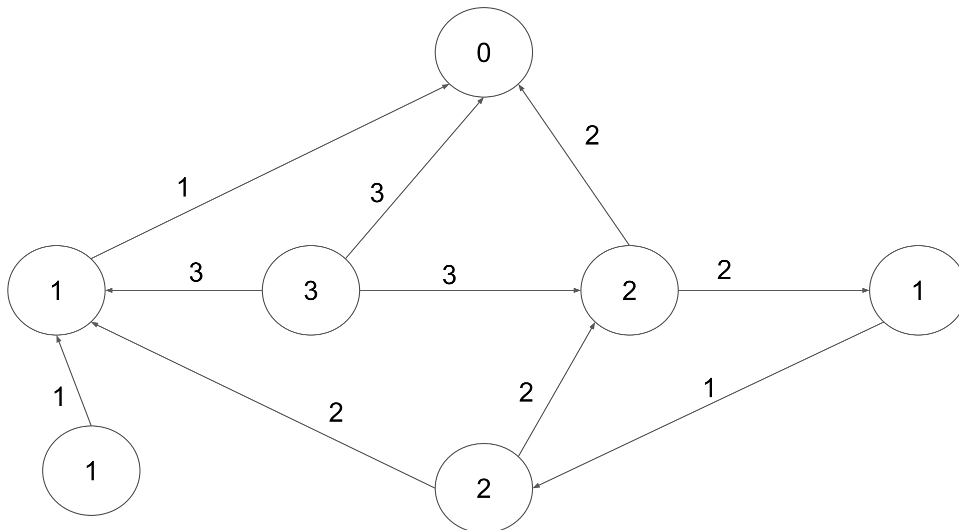


Figure 5.1: Example graph edge removing

the related intervals are shown in picture 5.2. The value of the property is the range of this interval, in the case of discrete numbers it's this one value. The algorithm checks the property for every edge and increments the counter of the interval. In the second step of the algorithm these intervals are used to calculate the threshold. For example, if the θ is 30%, the algorithm starts with the bin with the highest interval and calculates the percentage of the edges in this interval, 30% in the example. The 30% percent of this bin are greater or equal than the θ, thus the threshold in the example is 3. The first two steps together are the CDF. In the third step all values greater or equal than the threshold are deleted. In the example, the three edges with the property 3 will be removed.

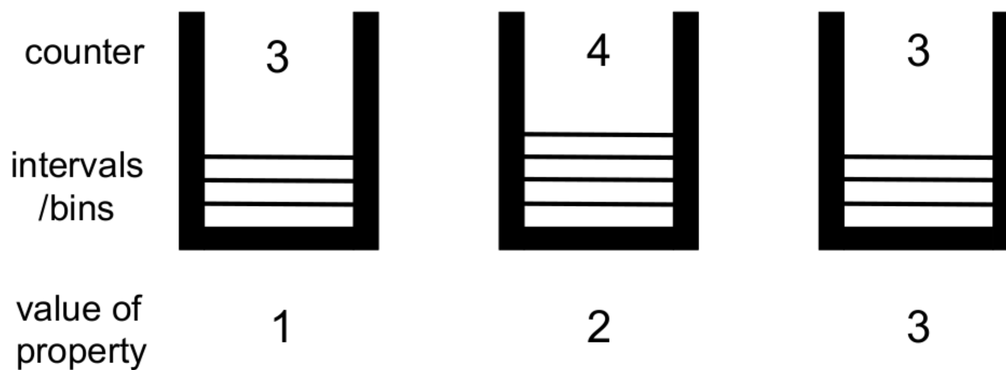


Figure 5.2: Intervals edge removing

Algorithm 5.1 shows the implementation of these steps in pseudocode. The *gatherDistribution*-method is used to count the number of values of the respective interval (line 1) and the *removeEdges*-method to calculate the threshold and removing of the edges (line 11). To gather the values and construct the CDF an array is used. The size of the array is the number of intervals needed. This size is different for different properties and graphs. Every element of the array is initialized with zero (not showed in Algorithm 5.1). Something which wasn't described yet is the factor needed for the properties except the out-property. Sometimes it is needed to multiply the value of the property with a factor to distribute the values to the full array (line 2). For the out-property the factor is 1. For the InOut-approach the number of intervals has to be greater than the factor because there will be many values greater than 1. The values are greater than 1 for the edges from source-vertices with greater in- than out-degree. Another detail in implementation not described yet is the handling of outliers. These are edges with a really high value of the property. There are only a few of them and it wouldn't be a good solution to choose more intervals because this would cost processing time. The outliers are added to the last interval (line 3-5) and because the number is clearly under 10% these values will be deleted at first or last (depends on the property).

When the distribution is gathered, the threshold based on the given θ can be calculated, which is done once. The procedure *removeEdges* is called for every edge of the graph, so a flag is used to do the calculation only once. The calculation itself is done by a for-loop that iterates over the array and sums up the elements of the array until the desired percentage is reached. Then the threshold is calculated and used to remove the edges from the graph. Sometimes it's not possible to set the threshold accurately. There are cases where too many edges would be removed. For example, if more than 10% of the edges have the exact same value for a property. The algorithm calculates a correction for the calculated dropping rate. For example, if θ is 20%, but the number of deleted edges will be 40% the correction choose randomly 50% of these 40% to remove the desired 20%. If the number of deleted messages is corrected, the correction is chosen 100% (line 28).

It would be possible to sample the graph before starting the algorithm. To be able to compare both approaches the dropping is done in the second superstep. There are two options after removing the edges. The first option is to only remove the edges but to not update the vertex data, so only the messages over these edges are dropped. The other option is to update the vertex data (in- and

Algorithm 5.1 Edge removing approximation algorithms

```
1:  procedure gatherDistribution(propValue,dropRate)
2:      pos = propValue ∗ factor
3:      if i > NROFINTERVALS then
4:          distributionArr[NROFINTERVALS] ← distributionArr[NROFINTERVALs] + 1
5:          count ← count + 1
6:      else
7:          distributionArr[pos] ← distributionArr[pos] + 1
8:          count ← count + 1
9:      end if
10: end procedure
11: procedure removeEdges(propValue,dropRate,edgeID)
12:     if firstTime = true then
13:         flag ← true
14:         dropPoint ← 0
15:         for i ← 0; i < NROFINTERVALS; i ← i + 1 do
16:             if dropPoint ∗ 100 < dropRate then
17:                 dropPoint ← dropPoint + distributionArr[i]/count
18:             else
19:                 threshold ← i − 1                    // Or (1/NROFINTERVALS)*(j-1)
20:                 break
21:             end if
22:         end for
23:         firstTime ← false
24:     end if
25:     random ← RandomNum(0, 100)
26:     if propValue ≥ threshold then
27:         if random ≤ 1/((dropPoint ∗ 100)/dropRate) ∗ 100 then
28:             RemoveEdge(edgeID)
29:         end if
30:         Gather(propValue)
31:     end if
32:     Gather(propValue)
33: end procedure
```

out-degree of a vertex saved for every vertex). This example illustrates the case, if the actual PR-value of a vertex is 1 and this vertex has four out-edges, the value sent over these edges is 0.25. If one of the out-edges is deleted by the algorithm without vertex data update, 0.25 is sent over the remaining three edges. With vertex data update the values are 0.33. All algorithms are evaluated without vertex data update, but the thesis also describes the effects of the vertex data update.

5.1.5 Random

One last edge-removing-approach is still missing in the descriptions. This approach doesn't use properties of the graph, it deletes edges randomly. For every edge the algorithm chooses a random number between 0 and 100. If the number is smaller than the dropping rate, we delete this edge. This approach is only for comparison with the other algorithms.

5.2 Value-Based-Approaches

The second approach drops messages individually based on message properties. The goal is to delete the messages with the smallest influence on the result of the PR-calculation. The priority of a message is rated by the property. Like for the edge-removing-approach, the algorithm must delete a given percentage of supersteps. But this percentage is deleted every superstep. The steps are the same as for edge removing, the message values must be gathered, a threshold must be calculated and the messages must be dropped. But in contrast to the edge-removing-approach these steps are executed every superstep.

To reach the goal to minimize error by a given fixed dropping rate θ, the algorithm needs to rate messages on their importance, to be able to remove the unimportant $\theta\%$ of the messages every superstep. To rate the messages the algorithm needs to use properties of the messages. In contrast to the edge-removing-approach, the algorithm can use the actual value of the message as well as graph properties. The next part will describe the used properties and why these properties were chosen to examine them.

5.2.1 Value-Based

The property used in this approach is the message value itself, so it's the actual PR-value divided by the out-degree of the edge, both of the source vertex.

$$\text{property} = \frac{\text{source.pageRank}}{\text{source.out-degree}} \tag{5.4}$$

The presumption of this approach is that messages with smaller values have less influence on the result than such with greater values. The number of intervals and the factor differ between the graphs.

5.2.2 Ratio-Value-Based

This approach works the same way as the value-based-approach, it adds another property to the calculation. The value used in the value-based-approach is divided by the PR of the target vertex.

$$\text{property} = \frac{\frac{\text{source.pageRank}}{\text{source.out-degree}}}{\text{target.pageRank}} \tag{5.5}$$

The idea takes into account how big the influence of this message is on this certain vertex. For example, if two nodes have an in-edge from the same vertex the values of the messages to these vertices will be the same. But the message could have a different importance for them, because if this edge is the only in-edge of a vertex the importance will be much higher than for a node with 100 in-edges.

5.2.3 Algorithm

These properties are used in the algorithm to rank the messages. In the following, for simplification, the text will describe the algorithm on the example of the value-based-property. The goal is to delete θ% of the messages every superstep and minimize the error. The algorithm needs to find the θ% smallest message values and drop them. The sorting of the message by their value isn't even possible in this approach because the algorithm has to decide online if a message is dropped or not. This means, it processes every message only one time and has to decide whether to keep or drop it in that moment. If it would be possible, it would also be too time consuming due to the sorting. The implemented approach uses the distribution of the values of the superstep before and calculates a threshold.

To calculate the threshold the algorithm uses also a CDF. In the first step, the value range is divided into intervals and the algorithm counts how many values are in the respective interval. The intervals can not be discrete values because the value-based-approach produces continuous values. Figure 5.3 shows an example in which the values in the vertices are the actual PR of the vertices (all 1 in the first superstep). The values at the edges are their properties. The algorithm checks every edge and

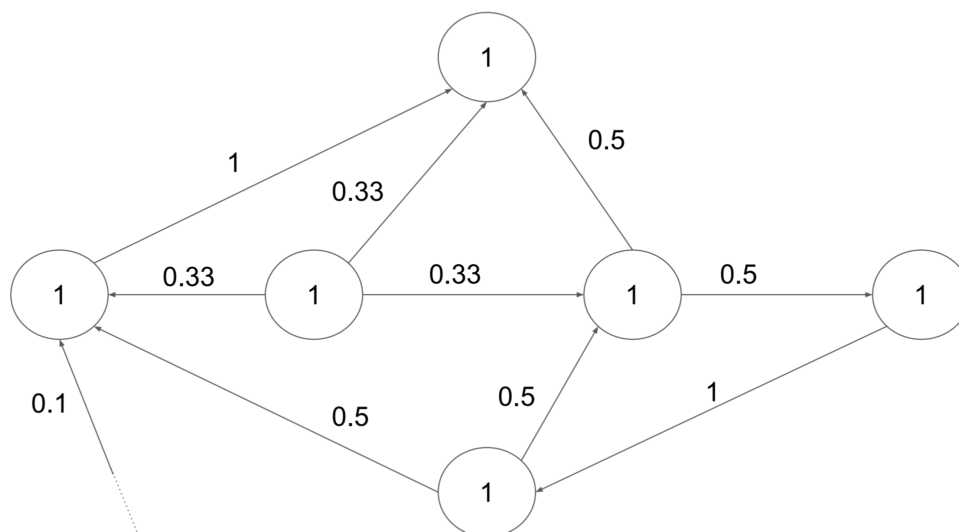


Figure 5.3: Example graph value-based

increments the counter of the respective interval. The result after the calculation with the ten edges of the graph in Figure 5.3 is shown in Figure 5.4. Always in the next superstep this distribution is used of the algorithm to calculate the threshold. For example, if θ is 40%, the algorithm starts with

the bin with the smallest interval and calculate the percentage of the edges in this interval, 10% in the example. These 10% are smaller than θ, so the algorithm calculates the percentage with the next interval. The 40% of both intervals are greater or equal than θ, thus the threshold in the example is the greater border of this interval, 0.4. The first two steps together are the CDF. In the third step all values greater or equal than the threshold are deleted. In the example the four edges with the smallest value will be deleted. In contrast to the edge-removing-approach this is done during the whole algorithm. In the first superstep, only gathering is possible, but from the second superstep on, the new distribution is calculated as well as with the last distribution the threshold is calculated and the messages are dropped.

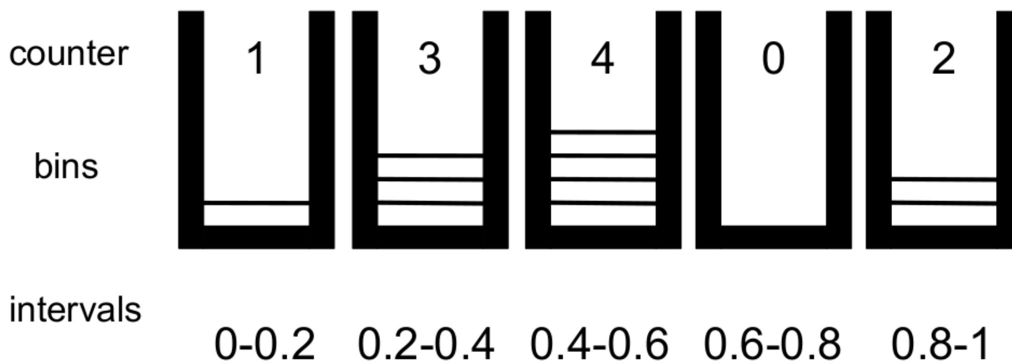


Figure 5.4: Intervals value-based

Algorithm 5.2 shows the implementation of the algorithm in pseudocode. The principle is the same, but there are some differences. The most important difference is that the constructing of the distribution is done from the first superstep on, while the threshold calculation is done from the second superstep (line 2, 11). As described, the calculations are done every superstep. As a consequence every element of the array must be reset to 0. This is done with the for-loop that is used for the threshold calculation, too (line 22). The random correction is the same as for the edge-removing-approach (line 28). Another difference to the edge-removing-approach is the deletion. It would be too time consuming to update the subgraph, because the update would only be used one time and then the next superstep must start on the original graph again. So, this approach skips these messages. This means for all messages that should be deleted the gathering function isn't called (line 29).

The potential of runtime saving is smaller than with the other approach, but this approach could delete messages based on their real value and so it may be suited to approximate better. Another benefit is that the dropping rate could be changed over the supersteps, because for example, the messages in the first supersteps are more important than the messages at later supersteps. The effects of the greater flexibility aren't investigated in this thesis.

Algorithm 5.2 Value-based approximation algorithms

```
 1: procedure generalApprox(propValue,superstep,dropRate)
 2:     if superstep >= 1 then
 3:         pos = propValue ∗ factor
 4:         if i > NROFINTERVALS then
 5:             distributionArr[NROFINTERVALS] ← distributionArr[NROFINTERVALS] + 1
 6:             count ← count + 1
 7:         else
 8:             distributionArr[pos] ← distributionArr[pos] + 1
 9:             count ← count + 1
10:         end if
11:         if superstep >= 2 then
12:             if firstTime = true then
13:                 flag ← true
14:                 dropPoint ← 0
15:                 for i ← 0; i < NROFINTERVALS; i ← i + 1 do
16:                     if dropPoint ∗ 100 < dropRate then
17:                         dropPoint ← dropPoint + distributionArr[i]/count
18:                     else if flag = true then
19:                         threshold ← (1/NROFINTERVALS) ∗ (j − 1)
20:                         flag ← false
21:                     end if
22:                     distributionArr[i] ← 0
23:                 end for
24:                 firstTime ← false
25:             end if
26:             random ← RandomNum(0, 100)
27:             if propValue ≥ threshold then
28:                 if random ≤ 1/((dropPoint ∗ 100)/dropRate) ∗ 100 then
29:                     ⇒ Don't process message
30:                 end if
31:                 Gather(propValue)
32:             end if
33:             Gather(propValue)
34:         end if
35:     end if
36: end procedure
```

6 Error Metrics

Minimizing error is the main objective of the proposed algorithms. While investigating error metrics we identified those for some algorithms. These metrics can be used to evaluate approximation results for the different algorithms.

6.1 Page Rank

This thesis describes metrics for the PR-algorithm. These metrics will be used in the evaluation to compare the different approaches and properties. As described in the problem statement one of the optimization goals is that the approximated PR-values should be as close as possible to the PRs calculated without approximation. Therefore the percentage error of every PR-value in the result is calculated:

$$\text{percVec} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \forall v \in V \left| \frac{PR_o(v) - PR_a(v)}{PR_o(v)} \right| * 100 \tag{6.1}$$

where PR_o is the „original“ PR and PR_a is a approximated PR and n is the number of vertices in the graph. The result is an vector with the error for every vertex. For evaluation, average and maximum of the vector are calculated.

The approximated ranking should be as similar as possible in terms of ordering of the web pages to the ranking calculated with the original PR-algorithm. There is a well-known metric used to calculate the distance between two permutations (in our case different rankings) $\sigma, \tau \in S_n$: the Spearman's footrule distance $F(\sigma, \tau)$ $[n] = \{1, \ldots, n\}$ is the elements in the universe (web pages) and S_n the permutations on $[n]$, then is $\sigma(i)$ the rank of element i. The formal definition is:

The spearman's footrule distance is given by

$$F(\sigma) = \sum_i (|i - \sigma(i)|) \tag{6.2}$$

this measures the total element-wise displacement from the original result.

The metric is 0 if the rankings are equal and rises the more dissimilar the rankings are. The best approximation algorithm is the algorithm that has the smallest ranking value.

6.2 Single Source Shortest Path

Similar to the PR result, the metric must take into account the length of the original path in comparison to the approximated path. So again it makes sense to calculate the percentage error of every result value, the path lengths of every vertex to the source vertex.

$$\text{percVec} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \forall v \in V | \frac{sssp_o(v) - sssp_a(v)}{sssp_o(v)} | * 100 \tag{6.3}$$

Where $sssp_o$ is the original PR, $sssp_a$ is a approximated PR, and n is the number of vertices in the graph. The result is a vector with the error for every path. For evaluation,the average, median, maximum and standard deviation are calculated.

When the graph becomes partitioned (several unconnected components) while removing edges, the vertices of the other component are unreachable. The rate of paths not available anymore is called failure rate. Since an unreachable path would be handled as distance ∞, a separate handling is needed.

6.3 Communities

The first presented metric is feasible for both presented community versions. The goal is to find the same communities with the approximated versions of the algorithm as the original algorithm. In [RAK07] a way is described to determine the percentage of nodes classified in the same group in two different solutions of strong communities. For that purpose we build a matrix M, where $M_{i,j}$ is the number of nodes chosen for community i in one solution and community j in the other solution. Then we can calculate

$$f_{same} = \frac{1}{2}(\sum_i max_j\{M_{ij}\} + \sum_j max_i\{M_{ij}\})\frac{100}{n} \tag{6.4}$$

This metric is often used for example in [New04] and [WH04], but has the drawback that sometimes several correct solutions exist instead of one. Another problem is the missing sensitivity to the seriousness of errors. For example, when few nodes from several different communities in one solution are fused together as a single community in another solution, the value of f_{same} does not change much.

6.3.1 Strong Communities

For strong communities there are more sensitive metrics to such differences between solutions, for example, the Jaccard's index [Mil89]. It counts the pairs that are classified in the same community in both solutions (a), as well as the pairs that are in the same community in the first solution and in

different communities in the second solution (b) and vice versa (c). Then the Jaccard's index is defined as:

$$Jac = \frac{a}{a+b+c} \tag{6.5}$$

The equation reaches values between 0 and 1, with higher values indicating stronger similarity between the two solutions.

Modularity [New06] measures strength of division of a network into communities (modules, clusters). Measures takes values from range $< -1, 1 >$. Values close to 1 indicate a strong community structure. When $Q = 0$ the community division is not better than random.

$$Q = \sum_{i=1}^{k} (e_{ii} - a_1^2) \tag{6.6}$$

Where k is number of communities, e_{ii} is the number of edges that has both ends in community i and a_i is the number of edges with one end in community i.

6.4 Graph Coloring

For graph coloring two things have to be taken into account. At first a good approximation should make as few errors as possible. An error would be two neighbors with the same color. So, the first metric is the number of the errors. It's important to note that an error will be counted twice if we check every vertex for same colored neighbors.

The second metric counts the number of the additionally colors needed by the approximated solution. The question of which metric is more important depends on the purpose of the approximation. For many applications it will be more important that the result is without error.

In this thesis we specifically examined the error metrics for PR as the purposed algorithms are specified for PR

7 Analysis and Results

This chapter describes the experimental setup used for the experiments, the used data sets, and the results of the experiments. The results are evaluated with the error metrics described in chapter 6 and also the runtimes of the different algorithms are evaluated.

7.1 Experimental Setup

The experiments run on a testsystem with 4 workers on a cluster with 4 machines connected to each other. Every single machine has 32 GB of main memory and an Intel Xeon W2145 CPU with 8 cores. The CPU base frequency is 3.7GHz and its turbo frequency is 4.5GHz. The operating system on these machines is ubuntu 18.04. The sampling and evaluation of the results were also done on this cluster.

To reduce the impact from randomness in some of the approximation algorithms and the shared hardware usage, each algorithm was run five times. The metrics were calculated for every single run and the means of the metrics are used for the evaluation. The metrics are the ranking metric and the percentage error metric described in chapter 6. The ranking metric measures the total element-wise displacement from the original result. The error metric calculates the percentage error of every vertex. In this evaluation the mean and the maximum of this error is used.

7.2 The Data

This thesis uses three real-world graphs with different sizes to evaluate the approximation algorithms. The preprocessing to get the needed format for all graphs is described in section 4.1. An overview of the graphs is given in table 7.1

In the Berkley-Standford webgraph [LLDM09], nodes represent pages from berkely.edu and stanford.edu domains and directed edges represent hyperlinks between them. The graph was crawled in 2002. It consists of almost 700,000 nodes and over 7.5 million edges.

The Google web graph [LK14], was released by Google 2002 as a part of the Google Programming Contest. It is made of almost 900,000 nodes and over 5 million edges.

The third graph we used for our analysis is a graph of the top private domains (TPD) of the eu-2015 graph. The eu-2015 graph is a large snapshot of domains in the EU taken in 2015 by BUbiNG [BMSV14] starting from the site http://europa.eu/. Mostly the TPD is the public suffix of an URL. An example for a public suffix is youtube.com for all websites from the domain youtube. But there are examples, like blogspot.com where people can host their own sites at websites like foo.blogspot.com. The public suffix would be blogspot.com but the TPD of this is foo.blogspot.com.

This is meaningfull for the purpose of this thesis because all these domains are independent websites. The graph is published on the website of the WebGraph project [BRSV19]. The files are online in a compressed form [BRSV11] and could be decompressed with the WebGraph framework [BV04] to an arc file of the graph (every edge is represented as a line with the ids of the nodes the edge connects). The graph consists of 6.7 million nodes and 170.1 million edges.

Graphname	# Nodes in millions	# Edges in millions
Web-Berk-Stan	0.7	7.5
Google-2002	0.9	5
eu-2015-tpd	6.7	170.1

Table 7.1: Properties of the used graphs

7.3 Analysis and Results

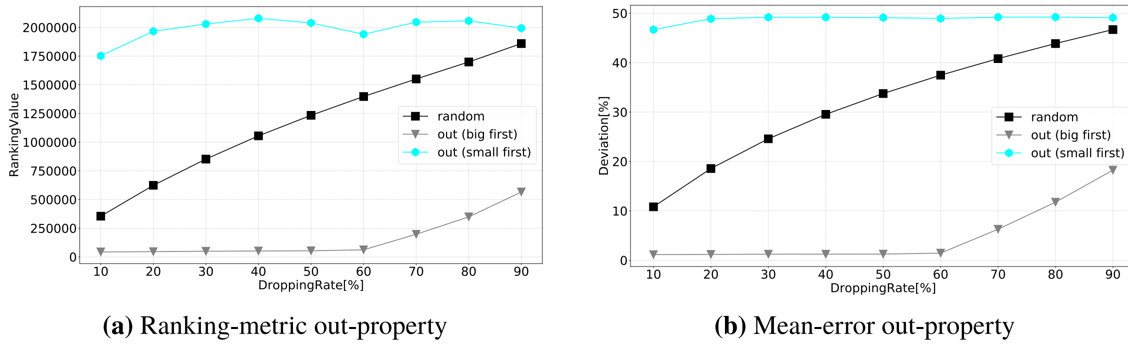
This chapter will describe and evaluate the results. First the edge-removing-approaches will be evaluated, then the value-based-approaches and as a third part both approaches will be compared. For the evaluation, the described metrics will be used and the thesis will also describe the runtime speedups with the approaches.

7.3.1 Edge-Removing-Approaches

At first we will describe the results of the out-property for the two different strategies. As said in the description of the properties, two strategies are implemented. The first strategie deletes edges with low out-degree at their source vertex first and the second deletes edges with high out-degree at their source vertex first. Then the effect of the update of the vertex data will be evaluated. The difference between these two approaches is, as described in the algorithm section, the update of the in- and out-degree information. These are the values, the framework uses to calculate the message values. Lastly an interesting observation with the inOut-property will be described.

Out-Property

This part will describe the results of the different out-approaches. The intuition before the experiments was that it is better to delete edges with high out-degree at their source vertex, first. Figure 7.1 shows the results of the eu-2015-tdp graph. It shows the ranking-metric (7.1a) as well as the mean-error (7.1b) of the approximated PR-values. The mean error is given as a percentage and the x-axis is the dropping rate in percentage. In addition to the result of both approaches the random-approach is plotted to show the comparison to this approach, too. As we can see the result is clear. To delete the edges with low out-degree at their source vertex first leads to high error and a bad ranking. Already with a dropping rate of 10% the mean error is over 46%. The out-property with deleting high out-degree edges first shows good results in comparison to the low out-degree



(a) Ranking-metric out-property

(b) Mean-error out-property

Figure 7.1: Comparison out-properties

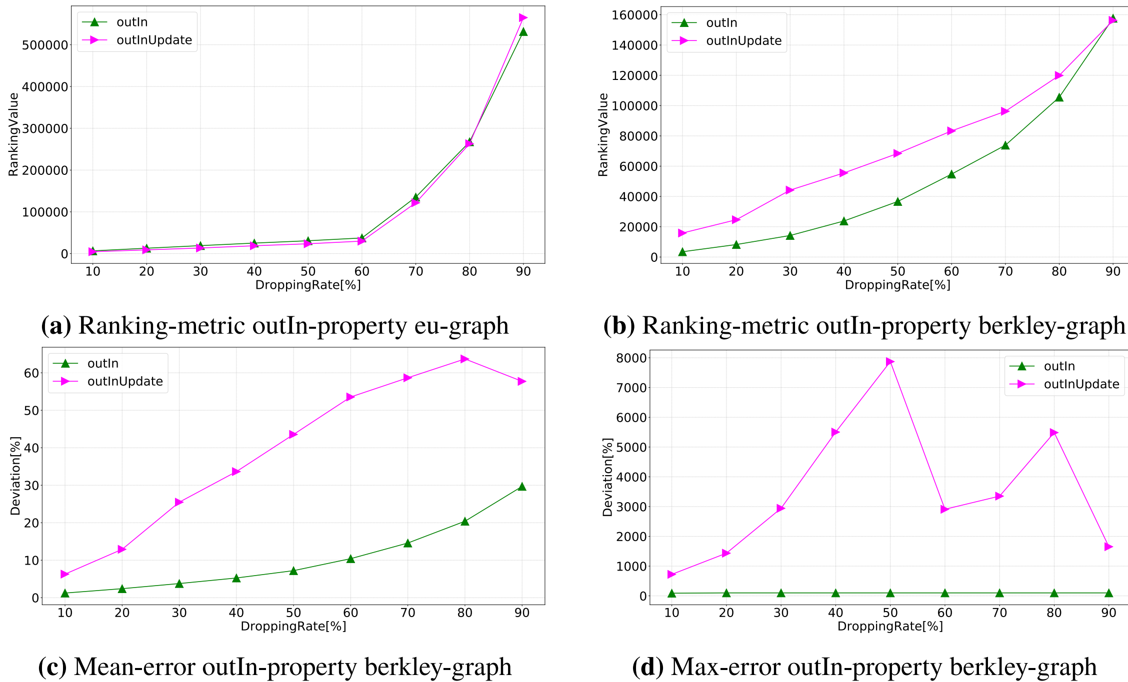
version and also in comparison to the random-approach. While the error for the random-approach is over 10% by 10% deleted edges and rising, the high out-property keeps the mean-error around 1% up to 60% dropping rate. The same holds for the ranking metric. At this point we can conclude the out-property with deleting edges with high out-degree at their source vertex shows promising results. For this comparison the results of the other graphs are the same. Later this approach will be compared against the other properties.

OutIn-Property

This section will describe the results for the outIn-property. The outIn-property was implemented in two versions. The difference between these two versions is, as described in the algorithm section 5, the update of the in- and out-degree information. These are the values, the framework uses to calculate the message values. The outIn-property multiplies the out-degree of the source vertex with the in-degree of the target vertex of the edge. The out-degree determines the portion of the PR-value sent over the edge. The in-degree factor based on the idea that nodes with many in-edges suffer less from removing some of these edges. Due to the fact that this property shows promising results in comparison to the other properties (this will described later), this property was chosen to test the vertex-update-approach.

Figure 7.2 shows some of the results. Figure 7.2a and figure 7.2b show the ranking results for the eu- and the berkley-graph. The version with the update of the vertex data shows slightly better results for the eu-graph from 10% to 80% dropping rate, but the results are very similar. This also holds for the google-graph (not showed with a figure). For the berkley-graph the results are different. The results of the version with update are clearly worse than the results for the version without update.

For all graphs the version with update shows higher mean- and max-error. Figure 7.2c and figure 7.2d show the mean- and the max-error of the two versions of the berkley-graph. The first noticeable observation is the max-error of the version with update. The max-error is the maximum error of all vertices in percent. For all other approaches the error can't become greater than 100%. It reaches 100%, if the approximated value is 0 in contrast to the original value. It's not possible that an approximated result has a higher value than in the original result. That's different in the version with vertex-data-update, because a vertex can get a bigger part of the PR-values of its neighbors. This leads to high maximum error up to 8000%. This means one PR-value is 80 times higher than in the original solution. Also the mean-error of the version with update is higher than the version without



(a) Ranking-metric outIn-property eu-graph

(b) Ranking-metric outIn-property berkley-graph

(c) Mean-error outIn-property berkley-graph

(d) Max-error outIn-property berkley-graph

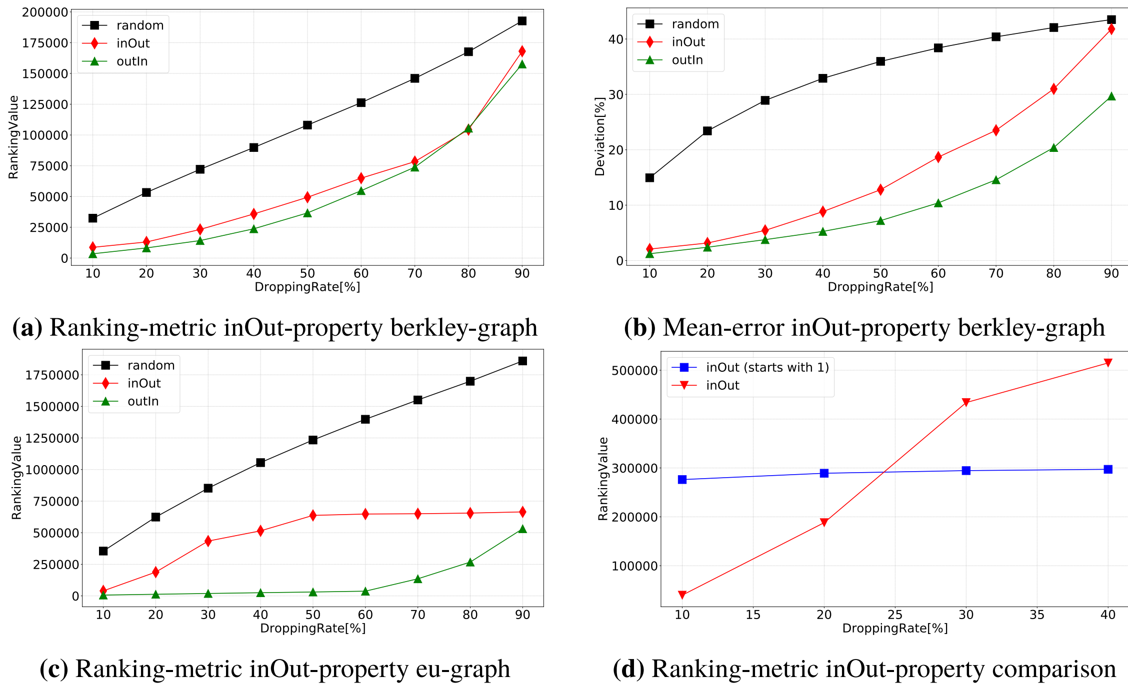
Figure 7.2: Comparison with and without vertex-update

vertex-data-update. So, we can conclude that the vertex update leads to higher error, but this has a surprisingly small effect on the ranking. The results with the ranking-metric vary from graph to graph. All together the vertex-update shows only small benefits in terms of ranking on some graphs, but much higher error. The results of the vertex-update-version are hard to estimate. The reason is that this approach could lead to a totally different distribution of the PR-values. For example, if many edges of a vertex with high PR are deleted, the remaining neighbors get much higher values. We can conclude the vertex-update doesn't improve the results of the approximations.

InOut-Property

The inOut-property divides the in-degree through the out-degree of the source vertex of an edge. For vertices with a high in-degree, a high PR-value is more likely. Edges with a small property value will be deleted first. The inOut-property shows on all graphs higher mean error than the outIn-property without vertex-update. Figure 7.3a and figure 7.3b show the results on the berkley-graph. The inOut-property shows better results, than the random-approach, but especially in terms of error the results are not as good as the results of the outIn-property. In terms of ranking, the results are closer to the outIn-property.

This also holds for the eu-graph, but on this graph the property performs worse (7.3c). On the eu-graph the property shows an interesting effect. The error rises relative fast up to 50% dropping rate and from 50% to 90%, the ranking-metric as well as the error keeps nearly constant. The property value of the additionally deleted edges from 60% on is around 1. This means these are edges with only a few more in-edges than out-edges. The eu-graph has many of those edges. It seems to be a good idea to delete these edges first because they lead to almost no additional error.



(a) Ranking-metric inOut-property berkley-graph

(b) Mean-error inOut-property berkley-graph

(c) Ranking-metric inOut-property eu-graph

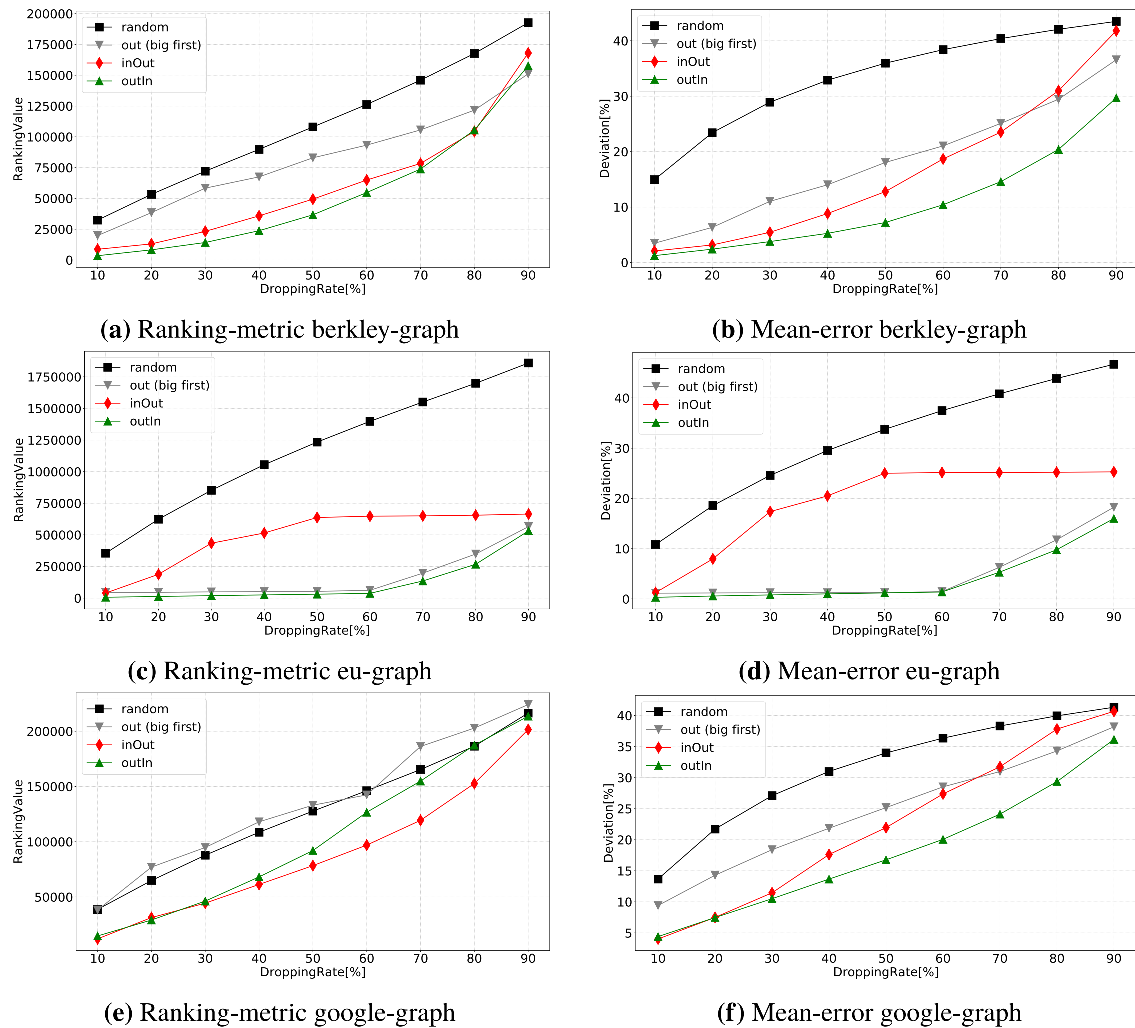
(d) Ranking-metric inOut-property comparison

Figure 7.3: Comparison of inOut-property with other approaches

The result, if the algorithm deletes these edges first, is shown in figure 7.3d. The blue line is the new result and the red line the original version until 40% dropping rate. As we can see in the beginning these edges lead to higher error, but these higher error almost stays constant. It's not a good solution to delete these edges first.

Comparison Edge-Removing-Approaches

The last parts described the three properties and their versions in detail, however the question is which of these approaches performs best overall graphs? As described, the out-property with deleting small values first performs badly in terms of error and ranking. The vertex-update version leads to higher error and no clear benefit in terms of ranking. Figure 7.4 shows the ranking-metric-results and the mean-error of the best version of each property. The out-property shows in no graph the best results. For the google graph it shows worse results than the random-approach in terms of ranking (7.4e). The inOut-property performs better than the two other approaches on the google graph in terms of ranking (7.4e), but it is worse than the outIn-property in terms of error on the same graph (7.4f). But this approach is much worse than the other approaches on the eu-graph (7.4c and 7.4d). Over all three graphs the outIn-property shows the best results. It's on all graphs the best in terms of error and only on the google-graph, with dropping rates over 50% worse than the inOut-property. The results show that the additional use of the in-degree of the target vertex is a great improvement against the out-property, because the results are always better. The outIn-property is also preferable to the inOut-property, because the inOut-property shows a bad performance on the eu-graph and performs also worse on the berkley-graph (7.4a and 7.4b). One reason why the outIn-property is the most stable approach is that it is using a property of the source vertex and the target vertex. All other approaches delete all out-edges of a vertex before deleting edges starting from another



(a) Ranking-metric berkley-graph **(b)** Mean-error berkley-graph

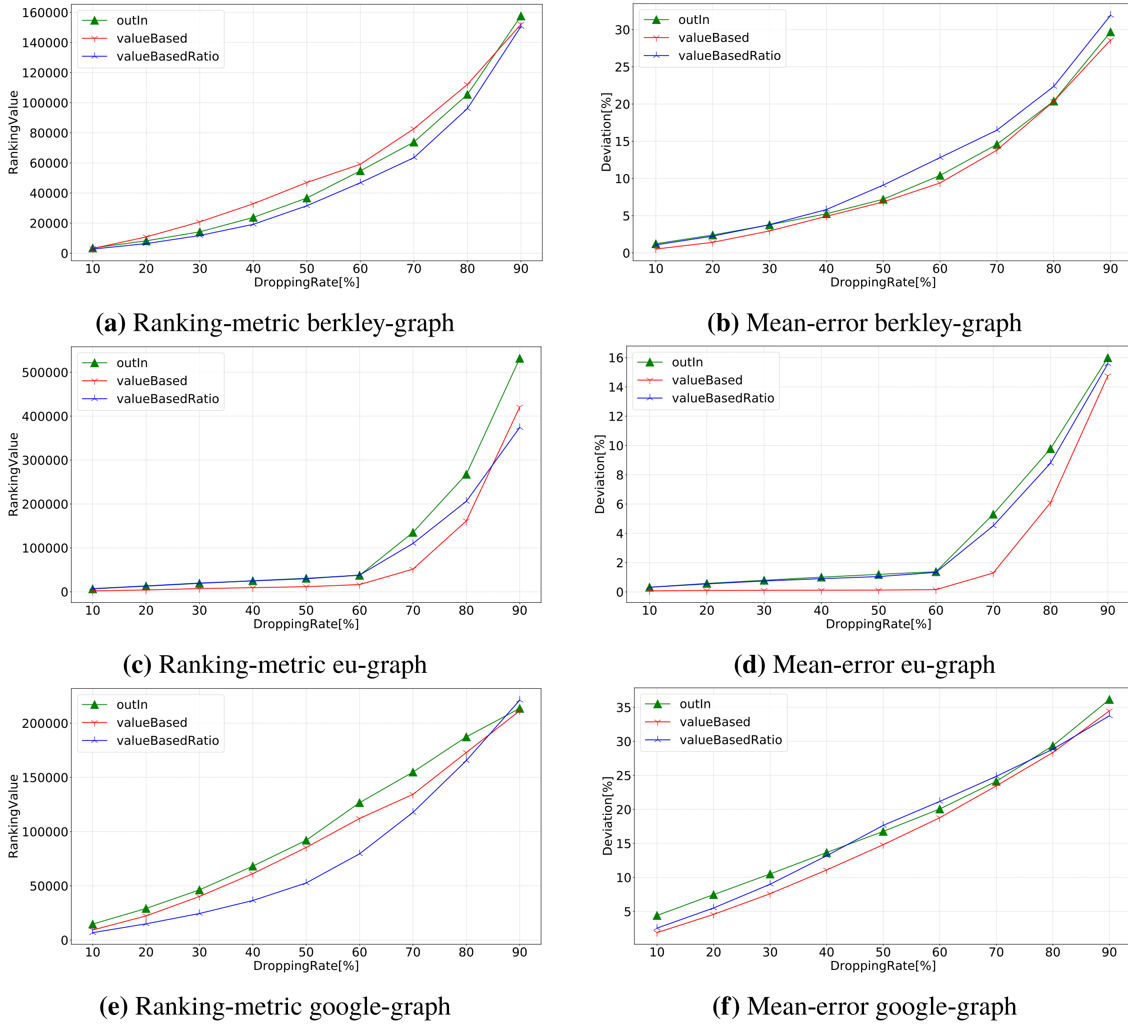
(c) Ranking-metric eu-graph **(d)** Mean-error eu-graph

(e) Ranking-metric google-graph **(f)** Mean-error google-graph

Figure 7.4: Comparison of the edge-removing-approaches

vertex because only the properties of one vertex decide about the edge property. The error is distributed over the neighbors because all edges of a vertex are connected to different vertices, but as higher the dropping rate as more vertices can't send the information about their PR-value. With the outIn-property less vertices can send no information, only a part of the information from more vertices gets lost.

Also the results on the different graphs differ in terms of improvement to the random-approach. The reason is the density of the different graphs. On the google-graph for each vertex there are about five edges, on the berkley-graph for each vertex there are ten edges, and on the eu-graph for each vertex there are 25 edges. So, a conclusion is that the approximations work better on denser graphs. That's a positive result, because the eu-graph is a newer crawl of the world-wide-web and there exists even more dense graphs. These graphs are especially interesting for approximation because they are even greater than the eu-graph and calculations on them would benefit from approximations.



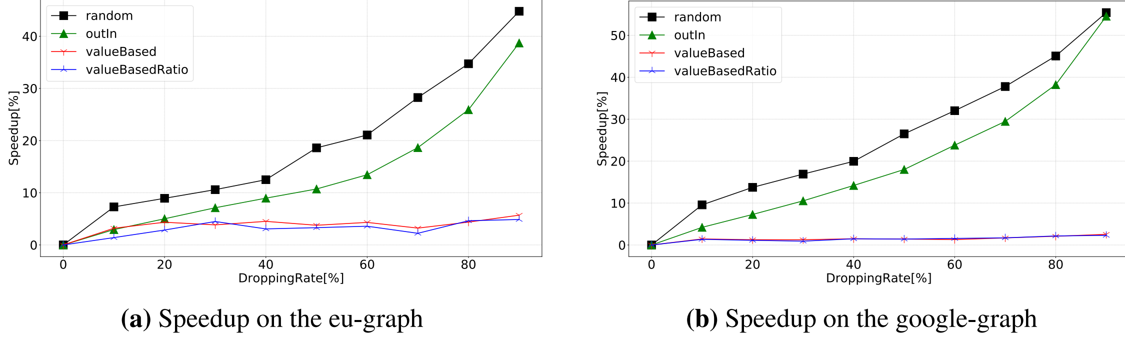
(a) Ranking-metric berkley-graph
(b) Mean-error berkley-graph
(c) Ranking-metric eu-graph
(d) Mean-error eu-graph
(e) Ranking-metric google-graph
(f) Mean-error google-graph

Figure 7.5: Comparison of outIn-property against the value-based-approach

7.3.2 Value-Based-Approaches

This section will describe the value-based-approaches. In contrast to edge-removing-approaches, these approaches decide in every superstep based on the message value which messages are dropped. Therefore, messages over different edges could be deleted every superstep. The first version of the value-based-approach decides on the messages value itself which messages should be dropped. The second version takes also the actual PR-value of the target vertex into account. This should delete the messages with the smallest impact at the target vertices first.

Figure 7.5 shows the ranking-metric results and the mean error of all three graphs. The random-approach isn't plotted for the value-based-approaches because the results a clearly better than the random-approach. For comparison the outIn-approach is plotted. Both value-based-approaches lead to very similar results on all graphs. The non ratio-version leads to better results in terms of error (7.5b, 7.5d, 7.5f). The ratio version performs better, in terms of the ranking metric on the google- (7.5e) and berkley-graph (7.5a), but a little worse on the eu-graph (7.5c). All in all none of the two versions is clearly better.



(a) Speedup on the eu-graph

(b) Speedup on the google-graph

Figure 7.6: Speedup

7.3.3 Runtimes

Figure 7.6 shows the speedup of the outIn-, random-, and both value-based-approaches. The speedup in the graph is the percentage, the approximating approaches are faster than the original version of the PR-implementation. The formal description is:

$$\text{speedup} = \begin{pmatrix} \theta_{10} \\ \theta_{20} \\ \vdots \\ \theta_{90} \end{pmatrix} = \forall\theta\left(\frac{\text{runtime}_o - \text{runtime}_a}{\text{runtime}_o}\right) * 100 \tag{7.1}$$

where runtime_o is the runtime of the original PR-implementation and runtime_a is the runtime of the approximated approach with the respective dropping rate θ.

The figure 7.6 shows the speedup on the eu-graph (7.6a) and the google-graph (7.6b). The results on the berkley-graph are similar. The speedup is showed for the outIn-approach, both value-based-approaches, and for comparison the random approach is shown as well. The edge-removing-approaches show a speedup up to 50% on the google-graph for 90% dropping rate. This means the calculation needs half of the time of the original implementation. On the eu-graph the maximal speedup of the outIn-approach is around 40%. The speedup could be increased by removing the edges in a preprocessing step. This is a benefit of this approach. The value-based-approaches show only a few percent speedup. On the eu-graph it's around 6% with 90% dropping rate. The reason why the edge-remove-approaches perform better in terms of speedup is that the edges could be removed finally from the graph. This saves main memory as well as the algorithm doesn't need to iterate over the deleted edges after the deletion. With the value-based-approaches the GrapH implementation needs to iterate through all edges once every superstep and can only save the time needed for the GAS-phases. In the case of PR the vertex function is not complex, it only calculates the sum of the message values. Therefore, the value-based-approaches don't save a lot of time due to the fact that the calculation of the threshold also takes some time.

7.3.4 Comparison

The comparison of the different properties for the edge-removing-approaches is done in Section 7.3.1. Now the best of these properties, the outIn-property, is compared to the value-based-approaches. Figure 7.5 shows the results of these three approaches on all graphs. The outIn-approach shows slightly better results than the value-based-ratio-approach in terms of error on the berkley- (7.5b) and the google-graph (7.5f). On the eu-graph the results are very similar to the value-based-ratio-approach (7.5d). Also for the ranking-metric results the outIn-approach shows almost as good results as the value-based-approaches.

In terms of speedup the outIn-approach performs clearly better than the value-based-approaches. The reasons are described above. The surprising result is that the outIn-approach shows similar results as the value-based-approaches in terms of error and ranking-metric. It is highly likely that there don't exist much better approximation algorithms, at least in terms of error, if we only consider message dropping and no other estimations. The reason is that the value-based-property deletes the messages with the lowest value and the value-based-ratio-property deletes the messages with the lowest impact at the target vertex first. As a result there couldn't be an approach that leads to lower error than these two approaches. It could be possible to keep the ranking in a better order (as the inOut-property did on the google-graph (7.4e)), but it's also unlikely that a clearly better property exists. This leads to the conclusion that the outIn-approach is a very good approximation property that only uses information of the graph which are known before the algorithm starts and that message dropping every superstep isn't useful for the PR-algorithm.

8 Conclusion and Outlook

This thesis presented approximation algorithms for distributed graph processing systems. From the numerous options to approximate distributed graph processing systems this thesis concentrated on message dropping. It investigated message dropping in two ways, individual dropping of messages based on message properties and dropping all message from selected edges (edge sampling). The dropping aims to reduce runtime, while minimizing the error. Different properties of the graph were examined and different versions of approximation based on edge sampling as well as message dropping based on message properties were implemented. Edge sampling means the decision which edges will be removed suitable for the algorithm is made at the beginning of the algorithm on the properties of the graph. Message dropping based on message properties means that the number of messages dropped is given by the dropping rate in every step of the calculation. The thesis investigated the PR-algorithm. The PR-algorithm calculates the importance of web pages. The approximations were implemented in GrapH, a distributed graph processing system.

The implemented approximation algorithms were discussed in detail along with an analysis of the latter. Different properties were tested, the graph properties for the edge removing approach and the message properties for the message dropping approach. Graph properties are, for example, the out-degree of the source vertex of an edge and an example for a message property is the value of the message itself. Experiments were conducted on a cluster with four machines. The evaluations were done on three real world graphs from 5 million up to 170 million edges to identify the best properties and investigate the speedup of the different approaches. The experiments showed that the edge removing approach is capable to achieve as good results as the message dropping approach. This is a surprising result because the message dropping is the best possible approximation in terms of error if we only consider message dropping and no further estimations. Further estimations would be the approximation of the vertex function itself or to try to compensate the dropped messages. The edge removing approach showed speedups up to 50%. So, the approximated version calculates the algorithm in half the time of the original implementation. The experiments lead to the conclusion that the edge removing approach is able to reach very good approximation results although it only uses information of the graph known before the algorithm starts.

8.1 Outlook

This thesis showed limitations of the speedup with approaches that delete single messages. There exist several possible reasons for this limitation. The first reason which was already mentioned is the simple vertex function of the PR-algorithm. A second possible reason is the system model of the GrapH-framework. The framework iterates in every superstep through all edges of the active vertices and that's more time consuming than the calculation of the sum for every edge. A last reason is the calculation of the threshold in every superstep which also takes time.

The first reason leads to the future work of investigating algorithms with a more complex vertex function. If the processing of a message at a vertex is more time consuming, approaches that delete single messages save more time. An interesting question is if it will be possible to decide with an efficient algorithm which messages should be deleted. This decision shouldn't take more time than the processing itself. For more complex algorithms this is more likely to be achieved.

Investigating different graph processing systems with different data-models will provide answers to the question if a different processing of the data provides better possibilities to approximate algorithms. In this context the investigation of edge-cut partitioning could be interesting. Vertex-cut showed superior properties for partitioning real world graphs but maybe edge-cut compensates these drawbacks with superior properties for the purpose of approximation through message dropping. Systems already exist that are able to work with vertex-cut as well as edge-cut partitioned data. These systems provide the possibility to compare both partitioning strategies directly against each other. A better suited data model for approximation is also a possible solution for the third mentioned reason. The approximation algorithm could benefit from a data processing specifically designed for the purpose of approximation through message dropping.

Furthermore, interesting future work would be investigating more algorithms. For example, the additionally described algorithms as well as triangle counting or subgraph isomorphism. Finally this should lead to a system that decides on the properties of the algorithm and of the graph which approximation techniques are suitable in this case. Also more approximation techniques are to evaluate. It is also possible to skip vertices. In terms of message dropping all messages of that vertex would be dropped. With vertex skipping it would also be possible to delete the vertices at the beginning of the algorithm finally from the graph or to choose the vertices which should be skipped in every superstep. The question would be if the findings in this thesis also hold for vertex-skipping approaches. An improvement for all approaches could be to try to compensate the dropped messages in a way. For example, use values from the superstep before again or estimate the dropped values.

Bibliography

[19] *Graph Data Usecases*. https://neo4j.com/resources/. 2019 (cit. on p. 15).

[AHR+14] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, M. Yu. "{GRASS}: Trimming Stragglers in Approximation Analytics". In: *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 2014, pp. 289–302 (cit. on pp. 16, 24).

[AMP+13] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, I. Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 29–42 (cit. on pp. 16, 24).

[And15] P. Andlinger. *Graph DBMS increased their popularity by 500 percent within the last 2 years*. http://db-engines.com/en/blog_post/43. Mar. 2015 (cit. on p. 15).

[BG11] A. Buluç, J. R. Gilbert. "The Combinatorial BLAS: Design, implementation, and applications". In: *The International Journal of High Performance Computing Applications* 25.4 (2011), pp. 496–509 (cit. on p. 23).

[BMSV14] P. Boldi, A. Marino, M. Santini, S. Vigna. "BUbiNG: Massive Crawling for the Masses". In: *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228 (cit. on pp. 29, 49).

[BP98] S. Brin, L. Page. "The anatomy of a large-scale hypertextual web search engine". In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117 (cit. on p. 20).

[BRSV11] P. Boldi, M. Rosa, M. Santini, S. Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th international conference on World Wide Web*. Ed. by S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, R. Kumar. ACM Press, 2011, pp. 587–596 (cit. on pp. 29, 50).

[BRSV19] P. Boldi, M. Rosa, M. Santini, S. Vigna. *WebGraph compressed datasets*. http://law.di.unimi.it/datasets.php. Mar. 2019 (cit. on p. 50).

[BTBR17] S. Bhowmik, M. A. Tariq, A. Balogh, K. Rothermel. "Addressing TCAM limitations of software-defined networks for content-based routing". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, pp. 100–111 (cit. on pp. 16, 24).

[BTG+18] S. Bhowmik, M. A. Tariq, J. Grunert, D. Srinivasan, K. Rothermel. "Expressive Content-Based Routing in Software-Defined Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 29.11 (2018), pp. 2460–2477 (cit. on pp. 16, 24).

[BTGR16] S. Bhowmik, M. A. Tariq, J. Grunert, K. Rothermel. “Bandwidth-efficient content-based routing on software-defined networks”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM. 2016, pp. 137–144 (cit. on pp. 16, 24).

[BV04] P. Boldi, S. Vigna. “The WebGraph Framework I: Compression Techniques”. In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601 (cit. on pp. 29, 50).

[CHK+12] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, E. Chen. “Kineograph: taking the pulse of a fast-changing and connected world”. In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012, pp. 85–98 (cit. on p. 23).

[CLS12] Z. Cai, D. Logothetis, G. Siganos. “Facilitating real-time graph mining”. In: *Proceedings of the fourth international workshop on Cloud data management*. ACM. 2012, pp. 1–8 (cit. on p. 23).

[CSCC15] R. Chen, J. Shi, Y. Chen, H. Chen. “Powerlyra: Differentiated graph computation and partitioning on skewed graphs”. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 1 (cit. on p. 30).

[FFF99] M. Faloutsos, P. Faloutsos, C. Faloutsos. “On power-law relationships of the internet topology”. In: *ACM SIGCOMM computer communication review*. Vol. 29. 4. ACM. 1999, pp. 251–262 (cit. on p. 30).

[GBNN15] I. Goiri, R. Bianchini, S. Nagarakatte, T. D. Nguyen. “Approxhadoop: Bringing approximations to mapreduce frameworks”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 1. ACM. 2015, pp. 383–397 (cit. on pp. 16, 24).

[Gep18] H. Geppert. “Evaluating Approximate Graph Processing with Graph Sampling”. In: *Fachstudie*. 2018, p. 8 (cit. on p. 29).

[GLG+12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. “Powergraph: distributed graph-parallel computation on natural graphs.” In: *OSDI*. Vol. 12. 1. 2012, p. 2 (cit. on pp. 15, 19, 23, 30, 31).

[GXD+14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, I. Stoica. “GraphX: Graph Processing in a Distributed Dataflow Framework.” In: *OSDI*. Vol. 14. 2014, pp. 599–613 (cit. on pp. 15, 23, 24, 30).

[HKKN04] J. Hansen, M. Kubale, Ł. Kuszner, A. Nadolski. “Distributed largest-first algorithm for graph coloring”. In: *European Conference on Parallel Processing*. Springer. 2004, pp. 804–811 (cit. on p. 22).

[HML+14] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, E. Chen. “Chronos: a graph engine for temporal graph analysis”. In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 1 (cit. on p. 23).

[ILDS16] A. P. Iyer, L. E. Li, T. Das, I. Stoica. “Time-evolving graph processing at scale”. In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM. 2016, p. 5 (cit. on p. 25).

[IPV+18] A. P. Iyer, A. Panda, S. Venkataraman, M. Chowdhury, A. Akella, S. Shenker, I. Stoica. "Bridging the GAP: towards approximate graph analytics". In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM. 2018, p. 10 (cit. on pp. 16, 26).

[KAKS99] G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar. "Multilevel hypergraph partitioning: applications in VLSI domain". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79 (cit. on p. 15).

[KBG12] A. Kyrola, G. Blelloch, C. Guestrin. "GraphChi: Large-Scale Graph Computation on Just a {PC}". In: *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, pp. 31–46 (cit. on p. 23).

[KK95] G. Karypis, V. Kumar. "METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0". In: (1995) (cit. on p. 30).

[Kun19] M. de Kunder. *The size of the World Wide Web (The Internet)*. http://www.worldwidewebsize.com. Mar. 2019 (cit. on p. 15).

[LBG+12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein. "Distributed GraphLab: a framework for machine learning and data mining in the cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727 (cit. on pp. 15, 23).

[LK14] J. Leskovec, A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on pp. 29, 49).

[LLDM09] J. Leskovec, K. J. Lang, A. Dasgupta, M. W. Mahoney. "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters". In: *Internet Mathematics* 6.1 (2009), pp. 29–123 (cit. on pp. 29, 49).

[LP01] M. Levene, A. Poulovassilis. "Web dynamics". In: *Software Focus* 2.2 (2001), pp. 60–67. DOI: 10.1002/swf.30. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/swf.30. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/swf.30 (cit. on p. 15).

[MAB+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146 (cit. on pp. 15, 21–23, 30).

[MCF+11] S. Ma, Y. Cao, W. Fan, J. Huai, T. Wo. "Capturing topology in graph pattern matching". In: *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 310–321 (cit. on p. 15).

[Mil89] G. W. Milligan. "A Study of the Beta-Flexible Clustering Method". In: *Multivariate Behavioral Research* 24.2 (1989). PMID: 26755277, pp. 163–176. DOI: 10.1207/s15327906mbr2402_2. eprint: https://doi.org/10.1207/s15327906mbr2402_2. URL: https://doi.org/10.1207/s15327906mbr2402_2 (cit. on p. 46).

[Mit16] S. Mittal. "A survey of techniques for approximate computing". In: *ACM Computing Surveys (CSUR)* 48.4 (2016), p. 62 (cit. on p. 15).

[MMB+18] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, K. Rothermel. "HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 458–467 (cit. on p. 15).

[MMI+13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455 (cit. on p. 23).

[MMMS15] P. Macko, V. J. Marathe, D. W. Margo, M. I. Seltzer. "Llama: Efficient graph analytics using large multiversioned arrays". In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 363–374 (cit. on p. 23).

[MTMR18] C. Mayer, M. A. Tariq, R. Mayer, K. Rothermel. "GrapH: Traffic-Aware Graph Processing". In: *IEEE Transactions on Parallel and Distributed Systems* 29.6 (2018), pp. 1289–1302 (cit. on pp. 15, 19, 23, 30).

[New04] M. E. Newman. "Fast algorithm for detecting community structure in networks". In: *Physical review E* 69.6 (2004), p. 066133 (cit. on p. 46).

[New06] M. E. Newman. "Modularity and community structure in networks". In: *Proceedings of the national academy of sciences* 103.23 (2006), pp. 8577–8582 (cit. on p. 47).

[PNS14] A. Pascale, M. Nicoli, U. Spagnolini. "Cooperative bayesian estimation of vehicular traffic in large-scale networks". In: *IEEE Transactions on Intelligent Transportation Systems* 15.5 (2014), pp. 2074–2088 (cit. on p. 15).

[PQD+15] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, G. Iacoboni. "Hdrf: Stream-based partitioning for power-law graphs". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM. 2015, pp. 243–252 (cit. on p. 29).

[QCB+17] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, T. Strufe. "StreamApprox: approximate computing for stream analytics". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM. 2017, pp. 185–197 (cit. on pp. 16, 25).

[RAK07] U. N. Raghavan, R. Albert, S. Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Physical review E* 76.3 (2007), p. 036106 (cit. on pp. 21, 46).

[RBMZ15] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel. "Chaos: Scale-out graph processing from secondary storage". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 410–424 (cit. on p. 23).

[RCC+04] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, D. Parisi. "Defining and identifying communities in networks". In: *Proceedings of the National Academy of Sciences* 101.9 (2004), pp. 2658–2663 (cit. on p. 21).

[SHK14] T. Suzumura, C. Houngkaew, H. Kanezashi. "Towards billion-scale social simulations". In: *Proceedings of the Winter Simulation Conference 2014*. IEEE. 2014, pp. 781–792 (cit. on p. 15).

[ST08] D. A. Spielman, S.-H. Teng. *Spectral sparsification of graphs. CoRR, abs/0808.4134, 2008*. 2008 (cit. on pp. 16, 26).

[SWW+12] Z. Sun, H. Wang, H. Wang, B. Shao, J. Li. “Efficient subgraph matching on billion node graphs”. In: *Proceedings of the VLDB Endowment* 5.9 (2012), pp. 788–799 (cit. on p. 15).

[SY14] Z. Shang, J. X. Yu. “Auto-approximation of graph computing”. In: *Proceedings of the VLDB Endowment* 7.14 (2014), pp. 1833–1844 (cit. on p. 25).

[TÇZ+03] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. “Load shedding in a data stream manager”. In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment. 2003, pp. 309–320 (cit. on pp. 16, 25).

[TÇZ07] N. Tatbul, U. Çetintemel, S. Zdonik. “Staying fit: Efficient load shedding techniques for distributed stream processing”. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 159–170 (cit. on pp. 16, 25).

[TKBR14] M. A. Tariq, B. Koldehofe, S. Bhowmik, K. Rothermel. “PLEROMA: A SDN-based high performance publish/subscribe middleware”. In: *Proceedings of the 15th International Middleware Conference*. ACM. 2014, pp. 217–228 (cit. on pp. 16, 25).

[Vig13] S. Vigna. “Fibonacci binning”. In: *arXiv preprint arXiv:1312.3749* (2013) (cit. on p. 30).

[WF94] S. Wasserman, K. Faust. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press, 1994.

[WH04] F. Wu, B. A. Huberman. “Finding communities in linear time: a physics approach”. In: *The European Physical Journal B* 38.2 (2004), pp. 331–338 (cit. on p. 46).

[WXDG13] G. Wang, W. Xie, A. J. Demers, J. Gehrke. “Asynchronous Large-Scale Graph Processing Made Easy.” In: *CIDR*. Vol. 13. 2013, pp. 3–6 (cit. on p. 23).

[ZB08] D. R. Zerbino, E. Birney. “Velvet: algorithms for de novo short read assembly using de Bruijn graphs”. In: *Genome research* 18.5 (2008), pp. 821–829 (cit. on p. 15).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature