GRAPH ALGORITHMS FOR APPROXIMATE AND DYNAMIC SETTINGS IN THE EXTERNAL-MEMORY MODEL

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Goethe-Universität
in Frankfurt am Main

von
DAVID VEITH
aus Offenbach am Main

Frankfurt 2019
(D 30)

Danksagung

Ich bin dankbar, dass ich während meiner Promotion ein wissenschaftliches Abenteuer begehen und zugleich eine wertvolle Erfahrung sammeln durfte. Am Lehrstuhl für Algorithm Engineering in Frankfurt am Main durfte ich mit den höchsten Kreisen der deutschen und internationalen Wissenschaft in Berührung kommen und gleichzeitig mein Verständnis für andere Kulturen rund um den Globus vertiefen. Dies wäre ohne die Mitwirkung vieler Kollegen, Freunde und meiner Familie nicht möglich gewesen.

Mein Dank gilt meinem Doktorvater Prof. Dr. Ulrich Meyer. Er hat mir die Chance eröffnet, Spitzenforscher in Deutschland und Europa besser kennenzulernen. Ich durfte bedeutende Konferenzen und Forschungsevents auf drei Kontinenten besuchen, sowie regelmäßig in Dänemark am Zentrum für große Datenmengen bei MADALGO in Aarhus forschen und so viele Kontakte knüpfen. Uli Meyer hat mich immer unterstützt, angeleitet und durch immer neue Herausforderungen in Forschung, Koordination und Lehre wachsen lassen.

Ein ganz besonderer Dank gilt Dr. Deepak Ajwani. Deepak hat sich während meiner wissenschaftlichen Karriere häufig als Co-Autor und Mentor erwiesen und mich durch seine ruhige, erfahrene Art und seinen Ideenreichtum dazu gebracht, immer nach höchsten wissenschaftlichen Standards zu streben.

Weiterhin möchte ich meinen ehemaligen Kollegen danken. Sie waren alle wunderbare Menschen, die den noch recht jungen Lehrstuhl innerhalb von etwas mehr als zehn Jahren zu einer Arbeitsgruppe geformt haben, in der ich mich überaus wohlgefühlt habe. Ich möchte mich bei folgenden Personen bedanken: Andreas Beckmann, Mahyar Behdju, Julia Hoeboer, Annamaria Kovacs, Arlene Kühn, Kasimir Kuliberda, Gabriel Moruz, Andrei Negoescu, Manuel Penschuck, Alexander Schickedanz und Volker Weichert. Ein besonderer Dank geht auch an Claudia Heinemann. Ich hatte das Privileg sie als lebenslustige, gebildete und nachdenkliche Person kennenzulernen. Sie hat meine Sicht auf die Welt und mögliche Lebensweisen nachhaltig geprägt.

Weiterhin möchte ich meinen anderen Kollegen an der Universität danken, die meinen Alltag und Mittagspausen bereichert haben. Das wären meine alten Kollegen von der GDV (u.a. Jörg Demmer, Thorsten Gattinger, Patrick Sacher, Daniel Schiffner, Sarah Voß-Nakkour, David Weiß, Alexander Wolodkin und Prof. Dr. Detlef Krömker), sowie die Mitglieder des Lehrstuhls THI - Bert Besser, Mario Holldack, Hannes Seiwert und Matthias Poloczek, sowie Prof. Georg Schnitger. Von ihm habe ich gelernt, wie man zugleich zutiefst menschlich als auch fordernd und fördernd sein kann, um alles aus einem Menschen herauszuholen (ich denke u.a. an eine Ansprache nach einer Prüfung, der ich beisitzen durfte). Weiterhin danke ich den gute Seelen des Prüfungsamts für Informatik, Diana Firnges und Sylvia Trouillet, für die Zusammenarbeit in der Studienfachberatung sowie im Prüfungsausschuss. Für die Zusammenarbeit in der Studienfachberatung gebührt auch Ronja Düffel ein gesonderter Dank. Schließlich danke ich noch Claudia Schwarz, die im Dekanat so einige Kohlen aus dem Feuer geholt und uns ebenso wie Jutta Nadland während Frau Heinemanns Erkrankung unterstützt hat. Ich danke all den anderen wunderbaren Menschen, die im Laufe meiner vielen Jahre an der Universität Frankfurt kennenlernen durfte.

Abschließend möchte ich mich ganz besonders bei meiner Frau Nadine und meiner gesamten Familie für ihre Unterstützung und ihre Liebe bedanken.

Abstract

In this thesis we examine huge data sets in the area of graph algorithms from a theoretical point of view, combined with implementations in C++ and experimental evaluations. A graph $G = (V, E)$ is an ordered pair with n vertices (sometimes also called nodes) in the set V and m edges in the set E. Graphs can be used to model objects as vertices and edges as connections between the objects. One example is the constellation Ursa Major, where the stars are the vertices and the edges connect the stars to the figure that can be seen in the star atlas. Due to their structural diversity, graphs can have different interesting properties. In this thesis we focus on the following main topics: diameter approximation, dynamic breadth-first search, and distance oracles. We use methods of algorithm engineering [San09] combined with the external-memory model [AV87] for theoretical considerations. Graphs are a common model to process connection based information like routing on streets or communication in networks, and customer behavior in online shops. Web based data is growing by several terabytes every day. Thus, classic graph algorithms designed for the RAM model (a fast processor plus a fast main memory) perform poorly compared to algorithms that are explicitly designed for big data in the external-memory model.

In 1987, Aggarwal and Vitter introduced the external-memory model [AV87]. Simplifying the hardware of a computer and its memory hierarchy into a model containing a CPU, a fast main memory of limited size M, and an external memory with communication in blocks of size B, this model allows us to abstract the cost of slow input/output operations (or short "I/Os"). The challenge is to reorganize data in a way that a block access to the external-memory device provides $\Omega(B)$ useful data elements that can be processed in the near future before the fetched data is displaced from the main memory again. A simple example is to read a text file and count the number of appearances of a specific word. Each time a new block is fetched, it can be immediately determined if the word appears or starts in the received block and there is no need to load this specific block again.
Around the year 2000, researchers started to implement the library STXXL, written in C++, which has been designed to supply commonly known C++ standard template

library (short: STL) packages like sorting [DKS08]. Using STXXL, the developer is able to use the sorting function for big data as easy as in the STL. The complexity of the library implementation in the external-memory model, which uses techniques to parallelize and interleave operations on external-memory and CPU, is hidden for the developer. We use the STXXL for implementations of our external-memory graph algorithms.

Several overview papers have been published on efficient graph algorithms in the external-memory model quite early [CGG+95, MSS03]. Nevertheless there are still some famous open external-memory graph problems, for which we lack efficient algorithms such as depth-first search. Similaryly, a factor 2 approximation of the diameter can be computed in linear time in main memory. However, in external memory – even after years of research – no algorithm with an equivalent number of I/Os, namely $\Theta((n+m)/B)$ I/Os, has been found in the last decades for depth-first search. Based on the work of Meyer [Mey08b], we present two variants of external-memory implementations, and a few improvements on them, in order to approximate the diameter of a graph I/O-efficiently. Our experimental evaluation demonstrates that these algorithms are viable and suitable as preprocessing steps for more complex algorithms that can be improved by knowing an approximation for the diameter of the given graph. The I/O-complexity of our first implementation is $\mathcal{O}(k \cdot \text{scan}(n+m) + \text{sort}(n+m) + \text{MST}(n,m) + \sqrt{\frac{n \cdot m}{k \cdot B}} \cdot \log_2\left(\frac{n}{k}\right))$ I/Os with an approximation error of $k \cdot \log(n)$. With $\text{scan}(n+m) = \mathcal{O}(\frac{n+m}{B})$ I/Os we express the number of I/Os to entirely read a linear file of size $n+m$, with $\text{sort}(n+m) = \mathcal{O}(\frac{n+m}{B} \cdot \log_{M/B}\left(\frac{n+m}{B}\right))$ I/Os the number of I/Os to sort a file of size $n+m$, and with $\text{MST}(n,m)$ the I/O-complexity to compute the minimum spanning tree (MST) of a graph with n vertices and m edges. The I/O-complexity for the currently best randomized EM algorithm to determine an MST is $\mathcal{O}(\text{sort}(n+m))$. Using recursion we were able to design an external-memory algorithm that can approximate the diameter of a graph using $\mathcal{O}(k \cdot \text{scan}(n+m) + \text{sort}(n+m) + \text{MST}(n,m))$ I/Os with an approximation error of $k^{4/3-\varepsilon}$ under the side condition that the recursively shrunken graph fits into the main memory which predefines the valid range for values of k. One of our contributions is a mechanism to automatically determine a value for k in the recursive step with high probability.

Subsequently we look at dynamic graph algorithms in external memory, which are applied to process data that changes over time. We consider the breadth-first search (BFS) algorithm on sparse graphs as an example, where we can improve the I/O-complexity from the static to the dynamic case. This is different from main memory, where the dynamic algorithm in general provides no guaranteed improvement over the time complexity. Our implementation for dynamic breadth-first search is based on previous work by Meyer [Mey08a]. We focus on the case that in each step an edge is inserted into the graph. While the currently best static EM-BFS algorithm has an I/O-complexity of $\mathcal{O}(n/\sqrt{B}+\text{sort}(n))$ I/Os on sparse graphs for one computation, the dynamic EM-BFS algorithm achieves an amortized I/O-complexity of $\mathcal{O}(n \cdot (\frac{n}{B^{2/3}}+\text{sort}(n) \cdot \log{(B)}))$ I/Os for n updates.
In addition to the implementation of dynamic BFS in external memory, we introduce a new clustering which is essential for the practically efficient preprocessing and the dynamic updates. The clustering provides clusters of uniform size each and dynamically changes the cluster sizes within one scan.
During the experimental evaluation, our dynamic EM-BFS implementation shows in many scenarios, even on more challenging edge insertion sequences, that it saves a significant amount of the I/O-volume to update the BFS tree compared to the static EM-BFS implementation.

Finally, we provide a real-time external-memory algorithm that is able to answer distance queries for arbitrary pairs of vertices with sufficient accuracy on real-world graphs with a diameter bounded by $\mathcal{O}(\log{(n)})$, once the algorithm has precomputed a larger data structure. We can answer online distance queries in milliseconds on SSDs (solid-state drives) and amortized even in microseconds on HDDs (hard disk drive), if the queries are answered in a batched way.

Contents

List of Figures

List of Tables

1 Introduction

Entities in our daily life such as street networks, transportation services, logistics or even relationships between people can be expressed by graphs. In the graph model we can use edges for several purposes to model properties of entities such as distances, connections or relationships. Science and industry in particular have been using graph algorithms for several decades in order to solve problems on big data and gain insights about relationships between entities from the real world.
Seen from the field of discrete mathematics, graph theory is a well researched topic. There exist various methods and algorithms to solve problems or provide services based on results from the graph theory since many decades. Many graph algorithms scale well on growing data sets - for example in linear time on graph traversal algorithms. Some of them are introduced in Chapter 2. However, when the input size grows to the point when the data set becomes too large to fit into the main memory of a single compute unit, the situation changes and these algorithms - designed for the RAM model - usually start to perform poorly. There are several options to deal with this issue. One possibility is to divide the data into several subsets, such that each subset fits into the main memory of a single machine. In a computing cluster these subsets are sent to an appropriate number of machines and partial results are computed separately. After the local computations have been finished, the partial results are gathered and combined to a solution by one or more machines. This scenario is especially part of the research areas distributed computing [Lyn96, TvS07] and high-performance computing [HW10, Eij15]. The concept of distributed computation is powerful if the partial results can be trivially combined. The challenge for distributed computation is that in general the combination of the results is complex and limited memory constrains increase the complexity. One example is parallel merge sort [Col86], where splitters are computed for both sequences that are parallelly merged.

Another common technique to speed up data processing is GPGPU (General-Purpose Computing on Graphics Processing Unit) that came up in 2001. At that time graphics hardware become more complex and with growing demands on shading the number of floating point operations that can be processed per second increased a lot. Today, graphics hardware with GFLOPS (10^9 operations per second) or even TGFLOPS (10^{12} operations per second) are available for the consumer market. In 2005, the first meaningful GPGPU implementation of LU factorization [GGHM05, DWL+12] which solves dense linear systems was published. In GPGPU a single instruction can cause several computation steps in parallel on one or more highly parallelized processing units (SIMD: Single Instruction Multiple Data). Algorithms that are designed for single instructions on single data - which is usually the case - might not scale well on GPGPUs and therefore we often have to process huge data sets in a different way in order to solve the underlying problem. There is a relationship between parallel and external-memory algorithms. If we achieve a well-performing algorithm in the external-memory model we can construct a well-performing parallel algorithm by a simulation. Refer to Dehne et al. [DDH03] for an example of the construction of an external-memory algorithm through a simulation on a parallel algorithm.

In industry, daily growing data sets resulted in more expensive machines with growing main memory. One example of this development is the HANA (High Performance Analytic Appliance) database architecture. Färber et al. [FML+12] provide a short introduction to the architecture of the model. The HANA architecture is based on a large main memory and compression of the stored data in order to keep as much data of the working set as possible in main memory. HANA is designed to be used in the daily work on business processes of a company. The working sets usually include only a narrow part of the data like a week or a month and a smaller footprint respectively summary of older data. However, if it is needed to store growing data sets on external-memory devices for special purposes like machine learning or optimization, due to the lack of free main memory, we still need efficient external-memory algorithms. In addition, such an architecture cannot be used on embedded systems as in production facilities or even on smart phones due to its large hardware requirements and energy consumption.

In this thesis we demonstrate that we are able to design algorithms on small hardware requirements even for real-time systems by the capability of the model we use for algorithm design.

1.1 History of the external-memory model

In theoretical computer science we utilize computation models to express space and time requirements of algorithms we design. Two very popular examples every computer scientist should be familiar with are the Turing machine [Tur38] by Alan Turing and the RAM model, as defined by Goldstine and von Neumann in 1947 [GvN47]. Both models are still very popular and have their advantages, as we can examine the complexity class of a problem $\mathbb{P}$ on a Turing machine or implement an algorithm - if $\mathbb{P}$ is a computable problem. The RAM model is used to determine time complexity on today's computers. However, the RAM model leaves a gap to the performance of realistic machines. While the RAM model assumes one computation step for each instruction and a fast main memory of unlimited size, in real computers fast memory is small and expensive whereas slower memory is larger and much cheaper. To close the performance gap in the memory hierarchy, the external-memory computation model has been introduced by Aggarwal and Vitter in 1987 [AV87].

The external-memory model gives a detailed description of the performance of memory accesses in the memory hierarchy of modern computers. The model consists of two levels of the memory hierarchy [1]: the main memory and the external memory of any computer. Access time to data that is stored in the main memory is reasonably fast (nanoseconds) compared to the execution time of an instruction on the CPU[2], whereas it is orders of magnitude slower to access a data element on a classic hard drive with mechanical disks, in both worst and average case. Thus, we aim to reduce the communication steps between these two stages of the memory hierarchy while we design algorithms with the external-memory model. The data elements are stored on the external-memory devices in blocks of fixed size. Even if only a single data element is requested from the external-memory device, the whole block is sent. A block transfer is called an **I/O** (input/output) operation. Since the introduction of the external-memory model, computer scientists were able to research efficient algorithms for several well-known graph problems in external memory as it was already the case for the corresponding

[1]Refer to section 2.7 for further details.

[2]On modern machines, 40 to 80 clock cycles are need to access a single data element in the main memory at random.

main-memory algorithms. Refer to [CGG$^+$95, KS96, AM09] to get an overview and an introduction into existing external-memory graph algorithms and ongoing research. However, there are still examples such as depth-first search (short **DFS**), where so far we have not been successful in developing an I/O-efficient algorithm, expect the trivial one with one I/O per data request which is not feasible in practice. Two heuristic DFS reference implementations for huge input graphs should be mentioned: an implementation by Sibeyn et al. and another implementation by Zhang et al. [SAM02, ZYQS15].

1.2 Research questions

In this thesis we deal with several research questions based on three algorithms. These algorithms have in common that they deal with graphs in an external-memory setting. The first algorithm is designed only for static data sets, whereas the others are designed for dynamic data sets. The third algorithm even deals with the demands of a real-time application. Our algorithms have the following research questions in common:

- How does the algorithmic performance depend on graph parameters like the diameter?
- Which graph classes enforce worst-case approximation bounds?
- Are there possibilities to improve our I/O performance respectively computation time?

Our first algorithm, which is introduced in Chapter 3, approximates the diameter of a given graph. We show that there are various possibilities to approximate the diameter of a graph and implement an algorithm by Meyer [Mey08b] and an extension of it. We published these two results at PASA 2012 and ESA 2012, respectively. Important research questions for this topic are:

- The original description of the diameter approximation left some details open. Does the idea perform well in practice, and how much effort does it take to fill in these details?

- Is the approximation error in practice good enough to use the heuristic as a preprocessing method?
- Is our algorithm competitive compared to existing solutions [AMO06, Bru07] for diameter approximation in external memory?

The next algorithm deals with dynamic breadth-first search that has been introduced by Meyer in 2008 [Mey08a]. We implemented a first prototype for ESA 2013 regarding the following research questions:

- Is it worth to use a dynamic algorithm compared to the static one?
- How much effort is needed to compute the dynamic clustering? And how much effort is done to change the size of the clusters, if needed?
- On which graph classes and in which update cases does dynamic breadth-first search perform well?

Our third algorithm deals with an external-memory distance oracle for real-world graph data. We demonstrate that we are now able to create data structures for applications that can be used in web services and other areas in real time. For such applications we have to answer queries in a few milliseconds. To achieve this, we deal with the following research questions:

- How can we use properties of real-world data to design a real-time application?
- Which hardware and parameters do we need to achieve our aim?
- How feasible is our trade-off between time and space?
- How close are the answers by the distance oracle to the actual distances and are they close enough for applications?

1.3 Structure of the thesis

This thesis aims to give an overview of our work on several graph algorithms, their history, current state of the art and possible ongoing work. We explain how we gained new insights into the behavior of algorithms on different graph classes and several algorithmic tools while we dealt with performance issues. We evaluate the performance of dynamic algorithms in external memory. We create and experimentally analyze an implementation for an external-memory algorithm for dynamic breadth-first search [Mey08a, BMV13]. Furthermore, we go one step further and focus on new scenarios where it is necessary to be able to answer all possible combinations of distance queries for two arbitrary nodes u and v in a graph instead of having a fixed starting point u and only the degree of freedom in the selection of an arbitrary target node v. Thereby we demonstrate that we are now able to develop external-memory approximation algorithms that can be used by companies to provide services which can be executed in real time. We conclude with a chapter about ongoing and future work.

2 Basics

2.1 Graph

Definition 2.1. *A graph $G = (V, E)$ is an ordered pair of two finite sets $V \neq \varnothing$ and E, where V represents the set of vertices and E the set of edges.*

The vertices are used to model entities as for example stars in a constellation. We describe vertices by their label: for example the vertex v_i is usually described by a label $i \in \mathbb{N}$. In a constellation the position of each star is an important property as well as in street networks and many other applications. It is possible to add further properties to vertices such as the vertex type. One important vertex type is the start vertex. This property is usually not fixed by the input graph and we have to determine which vertex or in some cases which and how many vertices we select as start vertices.

An edge $e \in E$ represents a connection between two or more[3] vertices in G. We focus on edges that connect two vertices. In the example of a constellation an edge is the connection between two stars. Edges can be directed or undirected. In the case of no direction, the edge between two vertices u and v can be used in both directions. More precisely, the set of undirected edges E is defined as follows: $E \subseteq \{\{u, v\} | u, v \in V \wedge u \neq v\}$. We exclude self-loops in our definition. If self-loops are allowed, the edge $\{u, u\}$ is a valid member of E, if $u \in V$.
In the case of directed edges, the first vertex of an edge represents the source and the second one the target vertex of an edge. The set of directed edges E is defined as $E \subseteq \{(u, v) | u, v \in V \wedge u \neq v\}$. A connection from u to v does not imply a connection from v to u.
Edges can be weighted by a function $c(e) : E \to \mathbb{R}$. An unweighted graph can be seen as a weighted graph with $\forall e \in E : c(e) \to 1$. Important application of edge weights are street or communication networks.

Annotation 2.2. *The cardinality $|V|$ of V is denoted by n (number of vertices) and the cardinality $|E|$ of E by m (number of edges).*

[3]If an edge connects more than two vertices, the graph is called a hypergraph [Ber89]

Definition 2.3. *The density of a graph $G = (V, E)$ is defined as the ratio between the actual number of edges m in the graph G and the largest possible number of unique edges $n \cdot (n-1)/2$ in an undirected graph that defines a complete graph K_n. A graph K_n is also called a **clique** of size n. We call a graph G **sparse**, if $m = \mathcal{O}(n)$ and **dense**, if $m = \Theta(n^2)$.*

In this thesis, graphs are stored in the adjacency list representation. We have a list L_u for each vertex $u \in V$ where each list L_u stores the an entry v for each edge $(u, v) \in E$. If the graph is undirected, the edge $\{u, v\} \in E$ results in an entry v in the adjacency list of u and vice versa. If the graph has weighted edges $(u, v, c(e))$, a pair of two values $(v, c(e))$ is stored in the adjacency list.
The graph model is quite simple to understand and very powerful. The graph model is used to describe various entities from the real world. Simple examples are subway, road, internet or social media networks. In the example of subway networks, the vertices model the stations and the edges[4] model the offered connections between these stations. In some flight networks, the edges between different points in the atmosphere can be used to model the fuel consumption, the average speed of the plane and therefore the duration of a flight, possible alternative routes (e.g. to redirect the airplane in the case of bad weather) or the height of a plane to use so called jetstreams that are usually stable in the atmosphere. However, a flight network is a very complex example, e. g. due to time dependency of edge weights.

2.2 Properties of graphs

Graphs have various properties we are potentially interested in. For an input graph at first only the sizes of n and m are known. However, if other properties can be quickly determined for the input graph, this knowledge can be used to select an algorithm that is tailored for the graph class the input graph belongs to. Examples for graph classes are trees, cliques or web graphs.
The diameter (Definition 2.5) of a graph is an example for such a property that potentially has a strong impact on the performance of an algorithm.
In main memory, we can approximate the diameter of a graph using linear time

[4] A single edge is used if we only model the tracks and multi-edges are applied if we model the different lines between stations.

graph traversal algorithms like breadth-first search (BFS) within a factor of two. However, in external memory, if the diameter is known, the appropriate external-memory breadth-first search algorithm [AMO06] could be selected. This results in a chicken-and-egg problem. It makes no sense to compute BFS for diameter approximation in external-memory to improve the performance of BFS itself. We would be forced to select the BFS algorithm with best worst case guarantees on every input graph in the first step to choose a better performing algorithm in the second iteration. We discuss this mutual dependence in Chapter 3 and engineer a fast approximation algorithm that allows us to avoid this self-dependence.
Another very important property is the degree $deg(v)$ of a vertex $v \in V$. Usually, we are interested in the average degree or the maximum degree of all vertices $v \in V$. The degree information is useful to design mechanisms to improve the approximation quality of heuristic algorithms. We show examples of such mechanisms, namely in the diameter approximation or our distance oracle computation (refer to the Chapters 3 and 5).

2.3 Breadth-first search

Breadth-first search (**BFS**) is one of the most fundamental and oldest graph traversal algorithms [Moo59, LS10, CLRS07]. BFS explores an unweighted graph $G = (V, E)$ from a start vertex $s \in V$ level-wise. The vertex s belongs to level L_0. The neighbors $N(0)$ of s are assigned to level L_1. The neighbors of level L_1, which have not been visited while a previous level has been constructed, are assigned to level L_2 $(N(1))$ and so forth. A level L_i is called the i-th BFS-level.
The data structure to implement BFS is a queue with the operations *enqueue*() and *dequeue*(). A bit array with random access is used to check whether a vertex has been visited. A sketch of the implementation is provided in Algorithm 1.

Each vertex v is marked as visited in line 12 and thus stored in the queue Q only once.When a vertex is fetched from the queue, all outgoing edges are read once for this vertex. If all vertices are reachable from the start vertex s, $\mathcal{O}(m)$ outgoing edges are read and n vertices are marked as visited. This results in a linear run time, we formulate in Lemma 2.4.

Algorithm 1 Breadth-first search

```
procedure BFS(Graph G, vertex s)
    Queue Q
    Array visited[n,false]
    Array level[n,-1]
    Q.enqueue(s)
    visited[s] ← true
    level[s] ← 0
    while !Q.empty() do
        vertex u ← Q.dequeue()
        vertices adjacent_vertices ← u.get_neighbors()
        for v : adjacent_vertices do
            if !v.visited then
                visited[v] ← true
                level[v] ← level[u]+1
                Q.enqueue(v)
            end if
        end for
    end while
end procedure
```

Lemma 2.4. *For an unweighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ the time complexity of* ***BFS*** *is $\mathcal{O}(n+m)$.*

A very space efficient linear time BFS algorithm by Hagerup requires $n \cdot \log_2(3) + \mathcal{O}((\log(n))^2)$ Bits [Hag18].

2.4 The single-source and the all-pairs shortest path problem

The single-source shortest path algorithm in a directed and weighted graph $G = (V, E)$ (**SSSP**) defines the task of finding a shortest path from the start vertex s to all other reachable vertices $v \in V$. Dijkstra's algorithm [Dij59] solves the SSSP within $\mathcal{O}((n+m) \cdot \log(n))$ computation steps for positive[5] edge weights

[5]Johnson modified Dijkstra's algorithm in 1973 so that it is able to deal with negative edge weights, if the graph has no directed cycles of negative length [Joh73].

$c(e) \in \mathbb{R}^+$ using a heap [Vui78][6] as the underlying priority queue. With later developed Fibonacci heaps the time complexity of Dijkstra's algorithm has been reduced to $\mathcal{O}(n \cdot \log(n) + m)$ [FT87]. Today's improvements of the SSSP complexity target the average case and tend to a linear complexity in average case [Mey03]. Thorup [Tho07] proved that with sophisticated priority queues SSSP can be implemented with an amortized time complexity of $(\mathcal{O}(n \cdot \sqrt{\log(n)}^{1+\varepsilon} + m))$. Elmasry et al. introduced a priority queue with reduced space consumption on bit level in $\mathcal{O}((m + n \cdot \log n + n \cdot \left(\frac{\log n}{s(n)}\right)^2)$ time [EHK15]. However, several side conditions for the input and output structure are needed to implement this space efficient approach. In this thesis we focus on a simple implementation based on the STL priority queue in C++ which solves SSSP in $\mathcal{O}((n+m) \cdot \log(n))$ time, which is feasible for our purposes.

With the SSSP-tree can answer queries from a fixed source s to an arbitrary target vertex. However, many applications require to answer queries between two arbitrary vertices. To answer such queries time-efficiently, the computation of a shortest path between all pairs of vertices is needed. The extension of SSSP is called the all-pairs shortest problem (**APSP**).
Two classic algorithms solve this problem: the Floyd–Warshall [Flo62] and the Bellman–Ford algorithm [Bel58]. Floyd–Warshall uses dynamic programming[7] and has a time complexity of $\mathcal{O}(n^3)$. Bellman–Ford was actually designed to solve the SSSP and is able to deal with edges of negative length. In addition, the algorithm can detect cycles of negative length and can be computed in $\mathcal{O}(n \cdot m)$ time. In total the Bellman-Ford has a time complexity of $\mathcal{O}(n^2 \cdot m)$ to solve APSP and therefore performs well on sparse graphs. The Floyd–Warshall algorithm is the better choice for dense input graphs. In the case of sparse graphs with non-negative edge weights, APSP can be solved by using Dijkstra's algorithm from every vertex in $\mathcal{O}(n \cdot m + n^2 \log(n)) = \mathcal{O}(n^2 \log(n))$ time. Beside these algorithms there are several improvements like solving APSP in $\mathcal{O}(n^2)$ on graphs with random edge weights from $[0, 1]$ with high probability [PSSZ10]. An overview of the currently known bounds for several APSP variants are given in a discussion of the Shoshan-Zwick

[6]The author refers to J.W. Williams and R.W. Floyd as originators of the heap and refers to the Art of Computer Programming [Knu73].

[7]Richard Bellman is seen as one of the originators of dynamic programming [Bel54].

algorithm by Eirinakis et al. [EWS17].

Several graph properties which we are interested in, can be determined by solving the single-source or the all-pairs shortest path problem for a graph. Some well-known examples are betweenness centrality (**BC**), where the value of each vertex v gives information on how many shortest paths in the graph over all shortest paths v participates [Fre77, Bra01, BMS15], closeness centrality (**CC**), as a measurement how close a vertex is to all other vertices [Bav50, Sab66], or the diameter of a graph. While each vertex in a graph might have a distinct value for BC or CC, the diameter is a global property of a graph.

Definition 2.5. *The diameter of a graph is defined as the length of the longest path among the set of all shortest paths in a graph G. If the graph is not connected, we only consider the diameter of the largest connected component in G [CGI+10].*

2.5 Graph partitioning

Divide & Conquer is a common technique to deal with algorithmic problems by recursion on growing inputs. On graphs, we usually divide the set V into smaller disjoint subsets V_i for $1 \leq i \leq k$. We call such a collection of k subsets a partition of V. Simple examples for partitions are: every vertex has its own partition, vertices are arbitrarily grouped into partitions at random, or vertices with a distance smaller than a specific value are grouped together. The last calculation regulation potentially produces conflicts in the partitioning algorithm. Vertices can be grouped into different partitions and the algorithm has to find a partition, where the number of vertices in each cluster is balanced, the centers of the clusters are well distributed and as many information of the graph structure as possible is represented by the clustering. In this thesis, we use various advanced methods to partition a graph (refer to sections 3.7 and 4.3 for examples) to work on smaller representations of a graph or load locally connected graph components as working sets.

2.6 Distance oracle

For many applications such as web services with real-time answer requirements for their users, an algorithm that solves APSP is not viable for reasonable large inputs. There are two main reasons for this. The first reason is that the data used by such web services usually stems from a dynamic setting. Hence, static precomputed answers for user requests have no value due to many changes of the data over time. Current work on dynamic APSP algorithms is currently just breaking the $\mathcal{O}(n \cdot m)$ bound [HKN16] which is still not feasible for real-time applications. The second reason is that for sparse graph data sets in the size of hundreds of gigabytes or even terabytes, it is often too expensive to store precomputed distance tables due to their quadratic size in the input size.

Hence, a new data structure has been developed by Thorup and Zwick in 2001 [TZ01, TZ05] – the distance oracle. It should be mentioned that a similar idea for objects in data bases and a distance function on these objects has been developed in 1998 by Yang et al. [YZW+98]. However, the approach by Yang et al. was not designed for graph algorithms. The publication of Thorup and Zwick led to more than one hundred subsequent publications on distance oracles and applications for graphs in the community. This is another example for the versatility of the graph model.
A distance oracle for a graph $G = (V, E)$ aims at two competing targets. The first target is to answer distance queries as fast as possible (usually we aim $\mathcal{O}(1)$ or at most $\mathcal{O}(\log(|V|))$ in practice for real-time applications), the second target is to require less space and the third target is to achieve a high approximation quality. Thorup's and Zwick's distance oracle is able to give a $(2k-1)$–approximation of all queries with a query time of $\mathcal{O}(k)$, using $\mathcal{O}(k \cdot n^{1+1/k})$ space and $\mathcal{O}(k \cdot m \cdot n^{1/k})$ preprocessing time on undirected and weighted graphs for any integer $k \geq 2$.
Since the introduction of distance oracles, some improvements have been found for unweighted graphs by Baswana et al. [BGSU08] respectively for weighted graphs by Pătrasçu and Roditty [PR10]. An important bound, shown by Sommer, Verbin and Yu [CSTW12], is that a k-approximate distance oracle needs at most $\mathcal{O}(n^{1+1/tk})$ space, where t is the time for preprocessing. Distance oracles have been implemented using various design patterns. They always have in common that there is a trade-off between the query time, the space requirements per vertex/object and the quality

of the output. Many newly developed distance oracles focus on design patterns and trade-offs based on their applications as for time-dependent networks [KZ16] or planar graphs [CDW17].

2.7 The external-memory model

The external-memory model (short **EM model**) has been introduced by Aggarwal and Vitter in 1987 [AV87] to design algorithms that can perform well on the communication between the main-memory and the external-memory level of the memory hierarchy. The cache-oblivious model, introduced in 1999 by Frigo et al. [FLPR99, FLPR12], is designed to perform well in each stage of the memory hierarchy. In this thesis we focus on the external-memory model, regarding only the communication costs between the RAM and the external-memory devices.

For simplicity in the EM model some assumptions of the RAM model are reused: the access of the CPU to a single data element in the RAM is as fast as one computation step and costs $O(1)$ time. However, the size of the RAM is now limited by M data elements, whereas in the RAM model the entire data set is initially stored in an external-memory. While a random access to the main memory is quite cheap, the external-memory device behaves differently. If an access to a single data element of constant size to the external-memory device is issued, the cost of this access is denoted by 1 I/O (input/output operation). The time to accomplish one random access on the external-memory device is between 10 to 100 μs for solid state drives and 6 to 10 ms for hard drives. In contrast, the time to perform one random access in main memory is between 40 and 100 ns. Hence, an I/O is several thousands to 100,000 times slower than one single access on RAM. On the other hand, if an external-memory device transfers data to the main memory, not only one element is transferred but a block with B data elements. Hence, external-memory devices are organized in blocks of size B to support this transfer pattern. With the block pattern for data organization on the external-memory device the vendors of such devices try to close the gap between the time to physically reach the position where the data is stored and the time to read the data when the data stream from the position is read. While accessing a block is very expensive (on hard drives in the range milliseconds), reading a block of consecutive data elements consumes only a

few nanoseconds per element. Another possibility to speed up the data transfer rate is to combine several devices as a RAID (Redundant Array of Independent Disks).
Abstracting all this information about the memory hierarchy and the hardware architecture, the external-memory model is defined by the following parameters:

- N is the size of the input. Initially, it is stored as a file on the external memory in consecutive, aligned blocks.
- M is the size of the main memory. The model assumes $N \gg M$.
- B is the size of a block of consecutive elements that are read or wrote in one I/O. $M/B \geq 2$ but usually the main memory can store a few hundred to a few thousand blocks simultaneously depending on the block size.
- D is the number of disks. For theoretical analysis we assume $D = 1$. In the later experiments we usually use 4 to 6 disks.

The two main operations in the external-memory model are scanning (reading/writing the data in blocks) and sorting data. We denote the consumed I/Os of scanning by $\text{scan}(N) = \Theta(\frac{N}{B})$ I/Os and of sorting by $\text{sort}(N) = \Theta(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$ I/Os. In practice external-memory sorting is slower by a constant factor (between 2 and 20) compared to scanning.

The external-memory model aims at two major targets. One target is to make external-memory algorithms comparable and the other target is to design algorithms in a way that they use as few I/Os as possible and still have a feasible computation time. We usually observe that external-memory algorithms are I/O-bound. That means that the CPU has time windows where it has to wait for data transfers from the external-memory device before the CPU can continue the computation. There are still a few cases where parts of an I/O-efficient implemented algorithm is compute bound, e.g. if many operations are done on each data element in a data stream. Pipelining operations, where for example the data stream is read, transformed, removed or sorted in consecutive computation steps, are one common example for compute bound implementation parts [BDS09]. Our libraries and implementations for algorithms are designed to balance the workload between the CPU and the external-memory devices with methods as interleaving – e. g. the computation starts when the first block is loaded from the disk instead of waiting the operation to be finished.

2.8 Minimum spanning tree computation in external memory

Definition 2.6. *A spanning tree $T = (V, E')$ of an undirected and connected graph $G = (V, E)$ is an acyclic subset $E' \subseteq E$, with $|E'| = n{-}1$.*

Deduced from Definition 2.6, a *minimum* spanning tree $T = (V, E')$ of an undirected, connected and *weighted*[8] graph $G = (V, E)$ is a spanning tree where the sum $C(T) = \Sigma_{e \in E'} c(e)$ is as low as possible.
Minimum spanning trees (MSTs) are well-understood for the RAM model. Early MST algorithms have been developed in 1926 by Borůvka [Bor26] and in 1930 by Jarník [Jar30]. The two basic algorithms which are usually taught in the first year theoretical computer science lectures are based on Prim's algorithm [Pri57] from 1957 and Kruskal's algorithm [Kru56] from 1956. Prim actually rediscovered Jarník's algorithm. Kruskal's algorithm is based on the union-find data structure. The MST algorithm by Prim has a time complexity of $\Theta(n \log(n) + m)$ using Fibonacci Heaps and Kruskal's algorithm has a time complexity of $\Omega(m \log(m))$. For external memory no efficient union-find data structure is known. An I/O-efficient batched version has been presented by Agarwal et al. in 2010 [AAY10]. A previously known sequence of N union and find operations can be executed with $\mathcal{O}(\text{sort}(N))$ I/Os, if the sets of each union operation are disjointed. Therefore, a commonly used implementation of external-memory MST algorithm modifies Prim's algorithm using the fact that we know how to implement I/O-efficient priority queues [BK98, BCFM00, WY14, ELY16]. Two special modifications are applied to Prim's algorithm in order to convert it into a feasible version for external memory. The first modification is that we store edges as ordered node pairs $< u, v >$ with

[8]For an arbitrary spanning tree it does not matter whether the graph is weighted or not.

their edge weight as the key in the priority queue instead of vertices with their smallest seen distance to the already constructed tree. With this modification we can easily check whether a vertex has already been visited by checking if the next entry on the priority queue concerns the same edge, e.g. the entry $< u, v >$ followed by the entry $< v, u >$. For unweighted graphs we ensure this by adding unique random edge weights to each edge or deterministically by numbering the edges in a fixed order (e. g. lexicographically). The second modification is that we reduce the data size by Borůvka steps [Bor26], which we introduce in Definition 2.7.

Definition 2.7. *Given a weighted input graph $G = (V, E)$ for the MST computation, a* ***Borůvka step*** *is a method to partition the vertices such that the vertices in a partition can be represented by a new vertex. Each partition has at least 2 vertices and therefore the output of the step has at most $|V'| = \frac{|V|}{2}$ vertices left.*

In order to partition the vertices for a **Borůvka step**, the edge with the smallest edge weight of each vertex is marked as being part of the resulting MST. In each step so called pseudo-trees with one cycle of length 2 each are constructed out of the marked edges. The cycles are removed and the remaining trees are a valid subset of the solution. The vertices of each subset are coalesced into a single super node. With a logarithmic number of such steps number of remaining super nodes can be reduced so that the computation can be finished with internal-memory algorithms.
Combining the modified priority queue with randomization and super Borůvka steps that guarantee less steps in total, we can achieve an I/O-complexity of $\mathcal{O}(\text{sort}(n+m))$ I/Os [ABW98, ABW02, KKT95][9]. An I/O-efficient implementation was published by Dementiev et al. in 2004 [DSSS04].

The external-memory MST algorithm is motivated by its versatile applications in the preprocessing of various external-memory algorithms and its usability within an approximation scheme, e.g., for the diameter of a graph (refer to Chapter 3). Usually, the computation of the MST is one of the first steps in a chain of preprocessing steps to reorganize data.

[9]It is an interesting open problem whether we can achieve this I/O-complexity for deterministic algorithms, too.

Lemma 2.8. *The external-memory minimum spanning tree computation can be done within $\mathcal{O}(sort(n{+}m))$ I/Os using randomization.*

2.9 Euler-Tour in external memory

The Euler-Tour is one of the first application for which graphs have been used. In 1736, the mathematician Leonhard Euler created graphs to solve the "Seven Bridges of Königsberg" problem [Eul36]. The problem was to find a tour through the city of Königsberg using each of the seven bridges exactly once. Euler proved that for the given topological layout of the seven bridges there exists no valid tour under the given constraints.
In this thesis we consider Euler-Tours on trees. Each edge can be used in both directions only once and therefore there always exists an Euler-Tour. To construct an unordered Euler-Tour, a cyclic order has to be fixed for each vertex (ingoing edge, outgoing edge, ingoing edge,...). In external memory an unordered Euler-Tour can be computed in $\mathcal{O}(\text{scan}(N))$ I/Os on trees [CGG+95]. The list ranking algorithm can be used to sort the tour data to a sequence of consecutive elements within $\mathcal{O}(\text{sort}(N))$ I/Os.

2.10 List ranking in external memory

Definition 2.9. *Given an unordered pointer-based linked list with elements of the type $(data, nextElement)$ and the position of the head, the task of the* ***list ranking*** *problem is to compute the rank of each list element. The rank of a list element is its position in the list. The head has rank* 1*, the next element rank* 2 *and so forth.*

The list ranking problem [CGG+95, Sib03, JLS14] can be solved within $\mathcal{O}(N)$ computation steps in main memory. Beginning at the head of the list in each step, the algorithm jumps to the address of the successor element e and memorizes the rank of e by the rank of the predecessor $p(e)$ plus one. In external memory this simple algorithm is not feasible due to $\Omega(N)$ memory accesses and therefore $\Omega(N)$ I/Os.
However, there exist several algorithms that can solve the list ranking problem

in $\mathcal{O}(\text{sort}(N))$ I/Os [MSS03, chap. 3.6]. One algorithm, derived from parallel computing, is based on the usage of an independent set computation:

Definition 2.10. *An independent set of a graph $G = (V, E)$ is a subset V' of V where no two vertices $u, w \in V'$ have an edge $e = \{u, w\}$ in common. Thus, if a vertex v is part of the independent set, no vertex u with $\{u, v\} \in E$ can be part of the set.*

For general graphs the computation of an independent set of maximum size is $\mathcal{NP}$-complete, whereas the computation of a maximal independent set is not. A maximal independent set means that the solution is not expandable, while the maximum independent set means that the size $|V'|$ of V' is maximum under all existing maximal independent sets of a graph.

The list ranking algorithm in external memory is designed as follows: first, an independent set L_1 on the list elements is computed using time-forward processing [CGG+95, section 4]. The elements in L_1 are removed from the list L and the resulting smaller list L' is recursively shrunken until the shrunken list has a size of $\min\{M, \frac{|L|}{B}\}$. When the list is shrunken we memorize each deleted element e by adding its weight to the weight of its successor $s(e)$ in order to compute the correct ranks of the remaining elements. One independent set computation for a list and the removal of its elements can be done within $\mathcal{O}(\text{scan})$ I/Os.
The total I/O-complexity of list ranking is $T(N) = T(2 \cdot N/3) + \mathcal{O}(\text{sort}(N)) = \mathcal{O}(\text{sort}(N))$ I/Os. The complexity is derived from the size of the maximal independent set on a list, which is at least $\frac{N}{3}$.

Lemma 2.11. *The external-memory list ranking problem can be solved recursively within $\mathcal{O}(\text{sort}(N))$ I/Os using independent sets in order to shrink the list.*

2.11 Breadth-first search in external memory

While in main memory breadth-first search has linear time complexity, there is no external-memory BFS algorithm known with an I/O-complexity better than $\mathcal{O}(\frac{n}{\sqrt{B}} + \text{sort}(n))$ I/Os for sparse graphs, which is slower than the MST computation [MM02].

In order to compute a BFS tree from a vertex v, two issues have to be addressed by the algorithm which many cause the I/Os.

1. Check whether a vertex from the queue has already been visited. One I/O per vertex would lead to $\Omega(n)$ I/Os and therefore this is not feasible.

2. For every unvisited vertex from the queue: load its adjacency list to process it. This is actually a bigger problem for sparse graphs, because in general only a small fraction of the block that has to be loaded can be used in the next computation steps, if at all.

In 1999 Munagala and Ranade published a BFS algorithm [MR99] for undirected graphs, which deals with the issue to recognize visited vertices and remove related entries from the working set efficiently. In the following we call this algorithm **MR_BFS**. The core idea of the external-memory algorithm is based on the observation that when a new BFS-level $L(t)$ is computed, only $N(L(t-1))$, the set of neighbors of the vertices in the already determined level $L(t-1)$ are potential candidates. Edges that connect a vertex $u \in L(t-1)$ to another vertex $v \in L(t-1)$ and edges that connect a vertex $u \in L(t-1)$ to a vertex $v \in L(t-2)$ have to be ignored for the construction of $L(t)$. Therefore, $L(t) = \{N(L(t-1)) - \text{duplicates}\} \setminus \{L(t-1) \cup L(t-2)\}$[10]. In order to compute $L(t)$ we have to maintain three sorted files in external memory and scan them in parallel while we compute $L(t)$[11]. The two files of $L(t-1)$ and $L(t)$ can be reused to compute the BFS-level $L(t+1)$. This causes $\mathcal{O}(\text{sort}(n+m))$ I/Os over all BFS-levels. Due to the issue to load adjacency lists, we additionally have $\mathcal{O}(n+m/B)$ I/Os to fetch the adjacency lists from the external memory at random. Hence, the total I/O-complexity of MR_BFS is $\mathcal{O}(n+\text{sort}(n+m))$ I/Os instead of $\mathcal{O}(n+m)$ I/Os for the trivial version. There are graph classes where MR_BFS is I/O-efficient in terms of sorting complexity. This for example is true, if the diameter of the graph is a constant k. For a bounded number of at most d

[10]With the term duplicates we refer to multiple appearance of the same vertex by several edges from the previous level from different vertices. Only one entry per vertex is permitted

[11]It is possible to merge these files into one file to avoid $\Omega(n)$ I/Os for writing these files or even keep small sets in main memory.

BFS-levels we obtain an I/O-complexity of $\mathcal{O}(d \cdot \text{scan}(m) + \text{sort}(n+m))$ I/Os. Since many real-world graphs have a very small diameter, MR_BFS performs well as a preprocessing step. In Chapter 5 we use this property of MR_BFS to construct a distance oracle, especially designed for real-world data with a small diameter.

For general graphs and real-world data sets, the bound of $\mathcal{O}(n + \text{sort}(n+m))$ I/Os for MR_BFS is not feasible for sparse graphs due to the worst-case $\Omega(n)$ I/Os by fetching adjacency lists at random. In 2002 Mehlhorn and Meyer published an external-memory BFS algorithm that deals with the second issue [MM02]. MR_BFS is still used as a subroutine to deal with the first issue. In addition, some new preprocessing is done on the graph in order to compute a partition of the vertices, so that it is potentially enough to load the adjacency lists of the grouped vertices only once into the main memory block by block. Then we can reallocate the main memory fast to replace outdated entries by adjacency lists that are likely to be used for the BFS computation in the near future.
The idea of Mehlhorn's and Meyer's external-memory BFS algorithm (short **MM_BFS**) is to use an extra data structure that maintains the adjacency lists of a graph with some knowledge about the distance of the vertices to each other in their cluster. The MM_BFS clustering of the adjacency lists of a graph G works as follows: first, a spanning tree $T = (V, E')$ of $G = (V, E)$ is computed. Next, an Euler-Tour is computed on T using list ranking to order the tour. Finally, we go through the Euler-Tour and split it into smaller clusters of size μ. Only the first occurrence of each vertex is kept in a cluster and empty clusters are deleted. Hence, there is a unique assignment of each vertex $v \in V$ to exactly one cluster $C(v)$. The distance of any vertex $v \in C(v)$ to any other vertex $u \in C(v)$ is at most μ.
If during the BFS computation the adjacency list of vertex $v \in C(v)$ is requested, the whole cluster $C(v)$ is loaded into the so-called hotpool $\mathcal{H}$, instead of a single entry. The exact value for the parameter μ and its impact on the I/O-complexity is discussed in the following paragraph.

For an arbitrary pair of vertices $u, v \in V$ their distance in the graph is upper-bounded by their distance in the corresponding MST T. In general $dist_T(u, v) \geq dist_G(u, v)$ is not a very tight bound. For example, the distance between two

vertices u and v can be 1 in the original graph G, but $\Omega(n)$ in the corresponding MST T. Hence, we gain no guarantees for the BFS tree by the looking at the MST. Nevertheless, if to vertices u and v have at most distance μ in the MST, their BFS-levels differ by at most μ. Thus, as previously described we partition vertices using the Euler-Tour. Ideally, the value μ would be $\Omega(B)$ so that we need $\mathcal{O}(1/B)$ I/Os amortized to load the adjacency list of vertex v.
On the other hand, if we load a cluster $C(v)$ into the hotpool, in the worst case this cluster stays for $\Theta(\mu)$ steps in the hotpool. If this happens for too many clusters during the BFS computation at the same time, we have to scan $\mathcal{O}(N/B)$ the clusters $\Theta(B)$ times which results in an I/O-complexity of $\Theta(N/B \cdot B)$ I/Os. Refer to Figure 24 for an example graph, where this run time behavior is enforced. In order to avoid this unbalanced performance between fast cluster loading and many scans of the hotpool file, we scan in each round, we have to find a reasonable trade-off between the cluster size μ and the number of scans of the hotpool on average. Mehlhorn and Meyer derived that μ has to be in the order of $\mathcal{O}(\sqrt{B})$ so that the scanning of the hotpool for each level-computation is not too expensive. This trade-off results into a more expensive loading of the adjacency lists, respectively in clusters of smaller diameter. It is an open problem if the two divergent aims[12] can be modified to improve the I/O-complexity. Altogether, MM_BFS improves the worst case I/O-complexity of external-memory BFS by a factor of $\Omega(\sqrt{B})$ while dealing with the issue to load adjacency lists I/O-efficiently. The original clustering for MM_BFS has been improved to keep the first or last occurrence of each vertex at random. This increased the expected workload of each cluster (number of vertices in a cluster) to at least $\frac{\mu}{8}$ [Mey08a]. In Chapter 4 we introduce a clustering that goes one step further: it is dynamic in the cluster size with a guaranteed high number of vertices in each cluster. It is used for the computation of dynamic BFS. We conclude this section with Lemma 2.12.

Lemma 2.12. *External-memory breadth-first search on undirected graphs can be solved within $\mathcal{O}(d \cdot scan(m)+sort(n+m))$ I/Os for small diameter graphs and for general undirected graphs consuming $\mathcal{O}(\frac{n}{\sqrt{B}}+sort(n+m)+MST(n,m))$ I/Os on sparse graphs.*

[12] To be as cheap as possible to load adjacency lists with $\mathcal{O}(1)$ I/Os while the hotpool has to maintain as few elements for each level computation as possible.

2.12 SSSP in external memory

In 2006 Meyer and Zeh [MZ06] provided an algorithm that solves the SSSP problem in external memory using $\mathcal{O}(\sqrt{\frac{(n \cdot m)}{B}} \cdot \log(n) + \text{MST}(n,m))$ I/Os on undirected graphs. Hence, on sparse graphs $\mathcal{O}(\frac{n}{\sqrt{B}} \cdot \log(n) + \text{MST}(n,m))$ I/Os are needed, which is only a factor of $\mathcal{O}(\log(n))$ slower than MM_BFS. However, it is claimed by Meyer and Osipov that this algorithm has too many complicated details to be implemented efficiently and would suffer of high constants [MO09]. They performed an experimental study to engineer an semi-external-memory implementation for SSSP based on STXXL, using ideas from previous work such as the hotpool structure to keep edges that are needed in the near future in internal memory, or bounded edge weights to maintain the queues for the sorters in the pipeline with less overhead. The available implementation is only semi-external due to one bit per vertex which is kept in the main memory.

2.13 STXXL

The standard template library for extra large data sets (short STXXL)[13] is a library written in C++. This library models the standard template library (short STL) for C++ in the external-memory setting. STL provides some basic data structures and algorithms such as queues, stacks, vectors or sorting. STXXL provides a similar interface, but in addition STXXL adds some underlying implementation layers to manage the I/Os for the developer, implicitly organizes the block layout and provides a pipelining interface to avoid I/Os by redundant operations [BDS09]. The first version of STXXL was released in 2003 by Dementiev and Sanders [DKS05, DKS08]. Since 2016, the STXXL is in a review process and has been modernized for C++14 and C++17. In this thesis we use version 1.3.1, released in March 2011. We additionally use some bug fixes of the developer version (queue package, some minor development output bug fixes). These bug fixes are part of the release 1.4 which was releases in December 2013.
The current version 1.4.1 was released in October 2014.

[13]refer to stxxl.org for further implementation details

2.14 Some details about graph classes and data sets used

In our experimental evaluations in the following chapters the same kind of graph data sets appear several times. These sets have diverging properties, which is used to evaluate the performance of the different implementations and make the performance results for the algorithms comparable to each other. As previously described for the external-memory Breadth-first search, the performance of an algorithm often depends on the graph structure. Hence, the evaluation of an implementation has to be done carefully, concerning the used graph classes.

- Graphs with a diameter of $\Theta(\log(n))$. Graphs in this diameter class are typically real-world data sets such as web graphs or social media graphs. Random graphs are another very important graph class that have a shape with logarithmic graph diameter with high probability.
- Graphs with a diameter of $\Theta(\sqrt{n})$. Examples for graphs in this diameter class are grids of other regular or non-regular structures with paths of length $\Theta(\sqrt{n})$ as stars or trees.
- Graphs with a diameter of $\Theta(n)$. Graphs in this class tend to be lists or structures with long paths and smaller hub regions in between.

In addition to these graph classes, we use worst case graphs, if we are able to construct such instances. These worst case graphs are tailored to provoke as bad performance as possible. Here is a small overview of our test data sets.

sk-2005: The sk-2005 graph[14] belongs to the class of real-world data sets with a small diameter in the range of $\mathcal{O}(\log(n))$. It has a diameter of 40 and is a web crawl of the Slovakian sk-domain. The UbiCrawler has been used to crawl the data for this graph [BCSV04]. This real-world data set has a huge core region with an even smaller diameter and some satellites and lists with small clusters in the outer rim. This data set has been used by several scientists in their experimental evaluations while working on graph algorithms. Hence, it is very likely that the sk-2005 graph will be used in the future again.

[14] http://law.di.unimi.it/webdata/sk-2005/

$\sqrt{n}$-level graph: This class is constructed by having a root vertex in the first level and in each following $\Theta(\sqrt{n})$ levels $\Theta(\sqrt{n})$ vertices each. For the last level less vertices might be generated to fulfill the user input for the parameter n. Our implementation ensures that the graph is connected by connecting each vertex in the new level L_i to on vertex on the previous level L_{i-1} at random. The edges left to achieve the desired number of edges respectively the mean vertex degree are created at random by inserting random edges from level L_{i-1} to level L_{i-1}. An example of the structure is given in Figure 1. This graph structure was used by Ajwani in his thesis [Ajw08].
$\Theta(n)$-level graph: These graphs have a large range diameter of $\Theta(n)$ and a regular structure. The same generator as for $\sqrt{n}$-level graphs is used. Therefore these graphs have a structure as shown in Figure 1 but only a few vertices on each level.

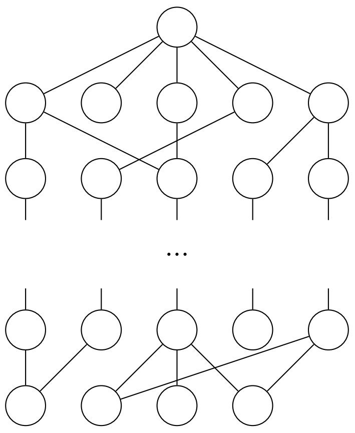


Figure 1: An example for a k-level graph [Vei12].

Table 1 enlists some important data sets that are commonly used in our experiments.

graph name	diameter	n	m	avg. $deg(v)$	file size [GB]
sk-2005	40	$5.06 \cdot 10^7$	$1.81 \cdot 10^9$	35.7	27.0
$\sqrt{n}$-level graph	16,385	$2.68 \cdot 10^8$	$1.13 \cdot 10^9$	4.2	33.6
$\sqrt{n}$-level graph 2	46,342	$2.15 \cdot 10^9$	$8.59 \cdot 10^9$	4.0	128.0
$\Theta(n)$-level graph	67,108,864	$2.68 \cdot 10^8$	$9.04 \cdot 10^8$	3.4	26.9
$\Theta(n)$-level graph 2	536,870,913	$2.15 \cdot 10^9$	$5.62 \cdot 10^9$	2.6	83.8

Table 1: Some statistics on our graph data sets we use since 2011.

3 Diameter Approximation

Preliminaries. Some parts of this chapter have already been partially published in a master thesis [Vei12] and two papers at PASA 2012 [ABMV12] and ESA 2012 [AMV12].

3.1 Introduction

We introduced the diameter d as a global graph property (refer to Section 2.2) defined by the maximum length among all shortest paths in a graph $G = (V, E)$. By knowing an approximation of the diameter of a graph we can choose the best suited graph algorithm among under all available options. The breadth-first search problem, we introduced in Section 2.11, is one example we are interested in. For a reasonable small diameter bounded by $d = \mathcal{O}(\log(n))$, we select the MR_BFS algorithm and for a larger diameter the MM_BFS algorithm which performs better on graphs with a larger diameter. This might have a huge impact on the computation time in practice. In the following we specify some side conditions under which the diameter of G is computed. These side conditions have already been used in well-known work on the diameter computation and approximation of graphs as in the publication by Crescenzi et al. [CGI+10]: If the graph $G = (V, E)$ is not connected we compute the diameter of the largest connected component C_1 under all connected components of G. We assume that C_1 is substantially larger than the other components. In many data sets we encountered a huge connected component, some singletons and a few connected components with less than 100 vertices. If needed, the diameter of smaller graph components can be computed independently. It is assumed that the source s is part of C_1.
Obviously, an easy way to approximate the diameter d of a graph in main memory is the computation of a BFS tree T_{BFS}. The height h of T_{BFS} bounds the diameter of an unweighted and undirected graph G within at most $2 \cdot d$. The height h of T_{BFS} can never be smaller than $\lceil \frac{d}{2} \rceil$. We prove this well-know fact in Lemma 3.1.

Lemma 3.1. *The diameter d of an undirected, unweighted graph $G = (V, E)$ can be bounded through one BFS computation from source $s \in V$ by $h \leq d \leq 2 \cdot h$ where h is the height of the computed BFS tree.*

Proof. We first show that $h \leq d$: Let the path from u to v be a path that defines the diameter of an unweighted and undirected graph. If the source s equals u respectively v, the resulting height h of the BFS tree matches exactly the diameter. Otherwise, if s is in any other position on the path, h is at least $d - min\{dist(s,u), dist(s,v)\}$ and by the definition of the diameter $h \leq d$ because the path from u to v is the longest among all shortest paths. If s is exactly in the middle of the path h is at least $\lceil \frac{d}{2} \rceil$. Refer to Figure 2 for an example.

If s is not on the path between u and v, we have to include the following considerations: the length $l(p_1) = d$ of the path p_1 between the two vertices u and v is the longest among all shortest paths in the graph G. The length of the paths p_2 from s to u and p_3 from s to v is bounded by d. Furthermore, $l(p_2) + l(p_3) > d$. This brings us to the second inequality, $d \leq 2 \cdot h$: From the first inequality it is known that h is at least $\lceil \frac{d}{2} \rceil$ and at most d. Hence, $2 \cdot h \geq 2 \cdot \lceil \frac{d}{2} \rceil \geq \lceil \frac{2 \cdot d}{2} \rceil \geq d$ and $2 \cdot h \leq 2 \cdot d$. □

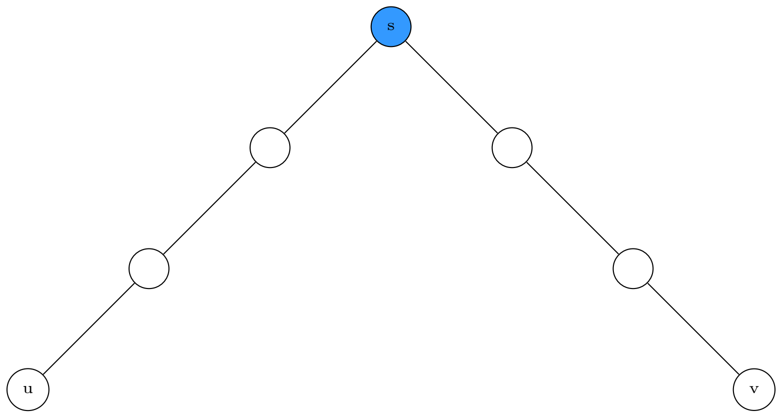


Figure 2: An example of a simple tree, where the BFS tree from the source s is exactly half of the diameter.

For the main-memory scenario BFS is a feasible diameter approximation algorithm with its linear time complexity in the size of the graph and the reasonable approximation factor of at most 2. In external memory, however, we have to select the EM-BFS implementation with the best worst-case I/O-complexity (refer to Section 2.11) even if another implementation needs less I/Os for the specific input.

Therefore, we want another algorithm to approximate the diameter in external-memory that has an I/O-complexity less than $\Omega(\frac{n}{\sqrt{B}})$ on sparse graphs. In 2008, Meyer published a scheme for an external-memory diameter approximation that is not based on the computation of BFS [Mey08b]. The core idea is the concept of parallel cluster growing. We evaluated this approach in 2012 [Vei12, ABMV12]. However, the performance of the parallel cluster growing still depends on the shape of the graph. Hence, we improved the parallel cluster growing to a recursive version [AMV12]. In this thesis we present the results in detail and evaluate the development since that time. In 2013 we tried to further improve our approximation guarantees in a diploma thesis by Timmer [Tim13]. However, only in a few cases the weight sensitive recursive cluster growing produced slightly better results compared to the first weight-oblivious recursive version, while significantly increasing the approximation error in many other cases. So, we focus on the version from 2012 which does not include the weights of the shrunken graph into the clustering.
Beyond the original purpose of the parallel cluster growing, the author of this thesis also used the core of the parallel cluster growing implementation for a lean minimum spanning (MST) tree implementation. This MST implementation is competitive to the reference MST implementation by Schultes in 2003 [DSSS04] but, is less complex and easier to understand and even helped to improve the reference implementation. This showed that parallel cluster growing is a versatile and efficient method that is worth to be considered for further applications.

3.2 Some theoretical background

Füredi and Kim proved a bound on the number of distinct graphs with n labeled vertices and a diameter $3 \leq d \leq n - c_1 \cdot \log(n)$ for any constant $c_1 > 0$ in 2012 [FK13]. The number of graphs for fixed values for n and d regarding the side conditions is upper bounded by

$$(1+o(1))\frac{d-2}{2}n_{(d-1)}3^{n-d+1}2^{\binom{n-d+1}{2}}.$$

Note that $n_{(d-1)} = n \cdot (n-1) \cdot \cdots \cdot (n-d+2)$. However, for reasonable large values of n and d, there exist too many different graphs so that a hash table or another

kind of lookup table is not a viable option to match input graphs to their exact diameter. In general, the time complexity to compute the exact diameter of an arbitrary unweighted graph is $\mathcal{O}(n \cdot m)$ [CDK02] and $\mathcal{O}(M(n) \cdot \log(n))$ for an arbitrary weighted graph, where $M(n) = n^{2.373}$ is the complexity of the matrix multiplication [Sei92, Wil12].
So, instead of exact diameter computation it is a viable option to use an approximation algorithm. In the literature, there exist various techniques to approximate the diameter that are based on one or more BFS computations. Corneil, Dragan and Köhler [CDK02] showed in 2002 that specialized versions of BFS can approximate the diameter with a result not worse than $d - \lfloor k/2 \rfloor$, where k is the size of the largest cycle in the given graph G.

In 2009, Magnien, Latapy and Habib gave an overview on techniques to determine a lower, respectively upper bound of the diameter [MLH08]. They introduced the double sweep lower bound technique (**dslb**), based on two BFS computations, as a well-performing algorithm with linear time complexity. For trees, lists and some other graph structures dslb returns the exact diameter [CDHP01, Han73]. In 2010, Crescenzi et al. published the fringe algorithm that can compute an upper bound of the diameter on many additional graph classes using an a priori unknown number of BFS computations [CGI+10]. In 2015 Crescenzi et al. published an improved algorithm that is able to determine a reasonable lower and upper bound of the diameter with only ten to a few hundred BFS computations on real-world graphs [BCH+15]. The authors stated that they were surprised by the good performance of their algorithm and that the most important open question was (and still is) why this approach performs that well.
Nevertheless, all these diameter approximation algorithms use BFS as an underlying function and their performance relies on the linear time of BFS in main memory. In external-memory, however, current BFS algorithms are much slower than their MST counterpart. Thus, we aim to achieve a better performance for diameter approximation with algorithms that avoid external-memory BFS.
The I/O-complexity of MST is viable, but the diameter approximation of an MST even of the complete graph K_n is in expectation $\mathcal{O}(\sqrt{n})$ [RS67]. Therefore, MST is no good candidate to replace BFS. With such an approximation error we would

fail to choose the proper algorithm for external-memory BFS on sparse graphs. MM_BFS would be selected instead of MR_BFS although MR_BFS could solve the problem within few sorting steps which is the best we can hope for. Hence, we need a heuristic that performs as close to the sorting bound in external memory as possible and has a much better expected approximation factor than the MST even on a complete graph.

3.3 Graph classes

In the following sections we discuss several heuristics that can be used to approximate the diameter in external memory. To evaluate the performance of the different approaches, we selected a set of graphs as follows: a real-world graph with small diameter of 40, instances of a synthetic graph class with diameter of $\Theta(\sqrt{n})$, instances of another synthetic graph class with diameter of $\Theta(n)$, and two graph classes that enforce worst-case behavior for our parallel and recursive cluster growing implementations. We present details about our worst-case graphs later in this chapter (refer to Sections 3.8 and 3.11). Details about the other graphs can be found in Section 2.14.

3.4 Hardware configuration

While initially working on this topic in 2011 and 2012 we used two kinds of hardware architectures in order to perform our experiments [ABMV12, AMV12]. The first architecture was designed for external-memory experiments and the second architecture was used for internal-memory experiments concerning some techniques that are too expensive to be computed in an external-memory setting. The following description is taken from our original paper, published at PASA 2012 [ABMV12]:

1. To determine the behavior of different techniques in an external-memory setting, we used a machine with an Intel dual core E6750 processor @ 2.66 GHz, 4 GB internal memory (around 3.5 GB free), 4 hard disks with 500 GB each as external memory for STXXL, and a separate disk for the operating system, application and storing data, log files etc. The operation system was Debian GNU/Linux amd64 'wheezy' (testing) with kernel 3.0. The programs were

compiled with GCC 4.4 in C++0x mode using optimization level 3.

2. For running the heuristics in internal memory, we used a machine (part of the HPC cluster at Goethe University) with four quad-core AMD Opteron^{TM} processors 8384 @ 2.7 GHz (only one core was used) and 64 GB internal memory. The only purpose of the internal-memory experiments is to determine the approximation quality.

3.5 External-memory BFS with DSLB

As a base line for our approximation algorithm we ran experiments with external-memory BFS on various data sets. We extended the MM_BFS implementation by Ajwani et al. from 2006 [AMO06] in order to use the double sweep lower bound technique. The results are presented in Table 2.

graph (diameter)	time [h]	approx. diameter	size [GB]
sk-2005 (40)	5.27	39	27.0
$\sqrt{n}$-level graph (16,385)	10.64	16,385	33.6
$\sqrt{n}$-level graph 2 (46,342)	56.6	46,342	128.0
$\Theta(n)$-level graph (67,108,864)	4.75	67,108,864	26.9
$\Theta(n)$-level graph 2 (536,870,913)	27.8	536,870,913	83.8
worst_PAR_APPROX (2,440,341)	1.66	2,440,341	8.0
worst_2step (8,111)	3.1	8,111	31.9

Table 2: Results from external-memory MM_BFS experiments with all data sets. Note that the diameter is easily determined for tree structures like the worst case graph classes, and for synthetic data with large diameters.

As discussed in Section 3.2, BFS can be a well-performing technique to approximate or – in some cases – even compute the diameter [BCH+15]. The results from Table 2 show that BFS combined with the double sweep lower bound performs well and reasonable fast. The performance of MR_BFS is of interest on real-world data sets like the sk-2005 graph, where the computation of BFS needs only a couple of minutes (about five to twenty minutes on a typical desktop depending on the type of external storage). Therefore, we use MR_BFS as a preprocessing routine for our distance oracle (refer to Chapter 5).

3.6 Diameter approximation based on MST heuristics

In the worst-case the diameter of a minimum spanning tree is a bad predictor to select the most suited EM-BFS implementation (refer to Section 3.2). On the other hand, the randomized computation of an external-memory spanning tree can be done in sorting complexity which is quite cheap compared to the currently known I/O-complexity of external-memory BFS in general. Furthermore, the computation of BFS-levels on any tree T can also be done in sorting complexity by computing an Euler-Tour on T first and then use list ranking to determine the distance of the vertices to the source. Knowing the BFS-level of each vertex in an arbitrary spanning tree T of a graph G, the following algorithm can be used to improve $T = (V, E')$ to another spanning tree T' with less height. First, we check for each vertex $u \in V$ in a batched way, whether there exists an edge $e = \{u, v\} \in E$ with $e \notin E'$, where the difference between the height-levels of u and v in the spanning tree is at least 2. Assume that the height-level of v is smaller than the height-levels of u. In that case, we replace the edge from the parent of u to u in T by the edge e, which refines the tree and improves its diameter approximation quality. The I/O-complexity of this strategy is $\mathcal{O}(\text{sort}(n+m))$ I/Os in each iteration with the implementation of Brudaru [Bru07]. If we head for a total I/O-complexity of $\mathcal{O}(\text{sort}(n+m) \cdot \log(n))$, up to $\mathcal{O}(\log(n))$ iterations are a feasible number of refinement phases. In the following sections we refer to the implementation of Brudaru by the shortcut **SPAN**. Our experimentally results for SPAN are presented in Table 3.

graph (diameter with BFS)	time [h]	approx. diameter	time BFS [h]
sk-2005 (39)	7.65	60	5.27
$\sqrt{n}$-level graph (16,385)	7.74	46,262	10.64
$\Theta(n)$-level graph (67,108,864)	4.81	86,488,096	4.75
worst_PAR_APPROX (2,440,341)	3.34	3,982,472	3.1

Table 3: Diameters approximated by SPAN. For the $\sqrt{n}$-level graph SPAN was faster than MM_BFS by 27%.

3.7 Parallel cluster growing

In order to outperform external-memory BFS for the external-memory diameter approximation, Meyer introduced two algorithms in 2008 [Mey08b]. Both algorithms are based on a parameter trade-off between the I/O-complexity and the approximation quality. Both algorithms shrink the graph $G = (V, E)$ during the preprocessing to a smaller graph $G' = (V', E')$ with $|V'| = \Theta(|V|/k)$ for a parameter k. A vertex $v \in V$ is part of the shrunken graph with probability $1/k$.

We first analyze the Euler-Tour based algorithm, which is derived from the clustering of MM_BFS (refer to Section 2.11). The aim is to reduce the number of vertices from $|V|$ to $\Theta|V|/k)$ deterministically. We first compute an MST within $\mathcal{O}((1+\log\log(B \cdot n/m)) \cdot \text{sort}(n+m))$ I/Os [ABT04] and subsequently we compute an Euler-Tour and a list ranking with $\mathcal{O}(\text{sort}(n))$ I/Os. The sorted tour is chopped into chunks of size $\mu = max\{1, \sqrt{\frac{n \cdot B}{n+m}}\}$. Only the first occurrence of each vertex v in a chunk is kept, while the duplicates of v in all other chunks are removed. Each of the $\mathcal{O}(n/\mu)$ non-empty chunks forms a cluster. The distance between two vertices u and v in such a cluster is bounded by μ in the graph by construction. We now reorganize the graph into a smaller representation in the following way: each cluster is transformed into a single new vertex: we merge the adjacency lists of all vertices in the cluster and thereby potentially reduce the number of edges, too. After the transformation we obtain a shrunken graph G' with $\mathcal{O}(n/\mu)$ vertices. The number of edges is reduced by one for every edge between two vertices in the same cluster and for every duplicate edge between the same pair of clusters. On the shrunken graph G' we run a deterministic MM_BFS which differs from the random variant by the described preprocessing. The height of the resulting BFS tree is the approximation of the diameter.
In order to determine the approximation bound, the size of the chunks is denoted by k with $1 < k < \Theta(B)$. We now recap the influence of the transformation on the graph which has been proven by Meyer [Mey08b].

Lemma 3.2. *Let $p_{u,v}$ be the shortest path with length $d(u, v)$ between the two vertices u and v in the unweighted and undirected graph G. Furthermore, let*

$d'_k(u,v)$ be the length of the path in the shrunken graph G'_k. The following inequality is true: $\frac{d(u,v)}{k} \leq d'_k(u,v) \leq d(u,v)$.

Proof. Let $\langle P = u = w_1, \ldots, w_l = v \rangle$ be a shortest path in G. Being a shortest path P has no cycle. Thus, $w_i \neq w_j \; \forall 1 \leq i < j \leq l$. With the clustering at most k different vertices are mapped to the same cluster $C(\cdot)$. The vertices of the shortest path P are mapped to their related clusters. Thus, $P'_k = \langle C(u), \ldots, C(v) \rangle$. Any two vertices v_i and v_j that are mapped to the same cluster $C(\cdot)$ create a cycle that shrinks the length of P'_k by at most $d(v_i, v_j) < k$. In total the distance of the two endpoints of P is at least $\lfloor d_G(u,v)/k \rfloor \leq d_{G'}(C(u), C(v))$. On the other hand the length of the path is not increased. Each vertex on the path P is mapped to one cluster. The original length is obtained if each vertex is mapped to a different cluster. Otherwise, at least one cycle is produced and the length of P'_k is decreased. It holds $d_{G'}(C(u), C(v)) \leq d_G(u,v)$. □

Following Lemma 3.2 the diameter d of the original graph G is reduced to $\lfloor d/k \rfloor \leq d_{G'} \leq d$ for the reduced graph G'. With a single external-memory BFS run on G', rooted at an arbitrary vertex v, the diameter is bounded by $\lfloor d/(2 \cdot k) \rfloor \leq d_{BFS} \leq d$. The approximation error is then bounded by $\mathcal{O}(k)$. In total we achieve an I/O-complexity for MM_BFS of $\mathcal{O}(\sqrt{\frac{n/k \cdot (n/k+m)}{B}} + \text{sort}(n/k+m) + \text{MST}(n/k, m))$ I/Os. With the preprocessing bounded by sorting complexity in the original graph size, we obtain in total an I/O-complexity of $\mathcal{O}(n/\sqrt{k \cdot B} + \text{sort}(n) + \text{MST}(n/k, m))$ I/Os for unweighted, undirected and sparse graphs with $m = \mathcal{O}(n)$.
The approximation error of $\mathcal{O}(k)$ is in general too large to select a suitable BFS implementation for external memory if the value of k is close to the value of B. On the other hand, if k is too small, we are still close to the I/O-complexity of MM_BFS. Thus we have a look at the second approach: the first algorithm does not propagate information to G' about the distance between the nodes within the clusters. With the following approach we preserve some information about the distance between the clusters and thereby reduce the approximation error. To obtain a weighted graph, each cluster C_i is created from a center point, well refer to as *master vertex*. After selecting $\mathcal{O}(n/k)$ master vertices at random and $O(n/k)$ deterministically by an Euler-Tour to ensure a distance of at most k between the master vertices, local BFS runs are computed from each master vertex as a source

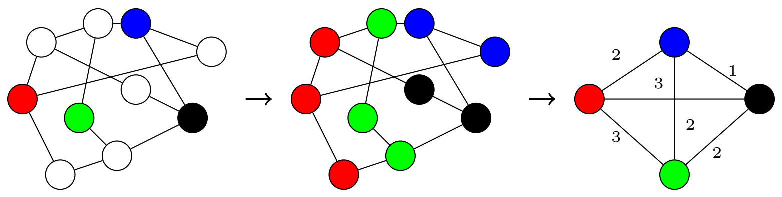


Figure 3: An example for the parallel cluster growing with 4 master vertices.

in parallel. In iteration step j we assign newly reachable vertices to exactly one cluster, which means that the distance of the newly assigned vertices to at least one master is exactly j and the distance to no master is smaller than j. Assume that two clusters C_a and C_b hit an unassigned vertex u at in the same iteration j. Then vertex u is assigned to C_a or C_b at random. Now, the two clusters meet at u. Hence, we store the cluster-connecting edge $\langle C_a, C_b, 2 \cdot j \rangle$ in the temporary edge set E'_t for the shrunken graph G'. Whenever a cluster C_e hits a vertex u' in a later iteration j' and u' is already assigned to another cluster C_d with distance j'', a new cluster connecting edge $\langle C_d, C_e, j'+j'' \rangle$ is added to E'_t. When all vertices are assigned to a cluster, we sort and scan E'_t and keep only the lightest entry for each cluster-connecting edge and obtain the edge set E' for G'. A small example for the *parallel cluster growing* as new preprocessing is depicted in Figure 3. Lemma 3.3 states the I/O-complexity of the parallel cluster growing.

Lemma 3.3. *The parallel cluster growing has an I/O-complexity of* $\mathcal{O}(k \cdot \mathit{scan}(n+m) + \mathit{sort}(n+m))$ *I/Os.*

Proof. We select $\Theta(n/k)$ master vertices deterministically on the Euler-Tour with at most distance of k between the masters on the tour and the same amount of master vertices at random. With the deterministically chosen masters we ensure that the distance of each non-master vertex to any master is at most $k-1$. With $k-1$ iterations the parallel cluster growing assigns each non-master to a cluster. Thus, the distance between the masters of two clusters is at most $2 \cdot k-1$.
In each iteration the adjacency lists are scanned once, what produces $\mathcal{O}(k \cdot \text{scan}(n+m))$ I/Os in total. In order to assign the vertices to exactly one clus-

ter, we have to sort the output of each iteration, which results in sort(n+m)) I/Os in total. □

We have to solve a single-source shortest path problem instead of the cheaper BFS computation in order to obtain the shortest paths on the shrunken graph. On the other hand, the expected approximation error is reduced to $\mathcal{O}(\sqrt{k} \cdot \log(k))$ [Mey08b]. In the original publication it is actually proven that the expected approximation for single paths is $\mathcal{O}(\sqrt{k})$. However, for graph structures with many paths with length in the range of the diameter it can happen that some resulting compressed paths in G' overshoot the expectation value by at most a multiplicative log-factor with high probability. We recapitulate our approximation bound for one path in Lemma 3.4.

Lemma 3.4. *When a graph is shrunken by the parallel clustering growing approach, the expected multiplicative error for one path is $\mathcal{O}(\sqrt{k})$ with high probability.*

Proof. We distinguish two cases: in the first case, the diameter d is at most $2 \cdot \sqrt{k}$ and in the second case the diameter is larger than $2 \cdot \sqrt{k}$.

1. Due to the diameter of at most $2 \cdot \sqrt{k}$, the edge weights between the clusters are in the interval $1, \ldots, \min\{4 \cdot \sqrt{k}, 2 \cdot k - 1\}$ and each path contains at most $2 \cdot \sqrt{k}$ edges. In Lemma 3.2 we already argued that the length of a shortest path P is maximally increased if every vertex on P belongs a different cluster and the edge weights are $4 \cdot \sqrt{k}$ each. This results in paths of weight $4 \cdot \sqrt{k} \cdot d \leq 8 \cdot k$ and therefore, the approximation error is bounded by $\mathcal{O}(\sqrt{k})$.

2. If the diameter is larger than $2 \cdot \sqrt{k}$, we consider a shortest path $P = \langle u = w_0, \ldots, w_{p'} \rangle = v$ with length of the diameter d and split it into sub-paths p'_i of length between $\sqrt{k}$ and $2 \cdot \sqrt{k}$ edges. Each path is considered separately. W.l.o.g. let x be the master vertex which reaches the vertex w_x on the sub-path p'_i earliest. By construction x has a distance of at most k to the reached vertex on p'_i. We denote the iteration in which x reaches the first vertex on p'_i by $t \leq k$. There are two sub cases. The master x can capture all vertices on the reached sub-path within the next $p' < 2 \cdot \sqrt{k}$ steps. Or, there is at least one other master vertex y which also captures another vertex $w_y \neq w_x$ on the

same sub-path between the steps t and $t+p'$. The number of such different master vertices, also capturing vertices on p'_i, is limited by the number of non-master vertices on the sub-path and by the number of non-masters in the neighborhood of the sub-path. Let us quantify the impact of these capturing master vertices on the resulting path length: when x reaches w_x in step t, a weight of size t is added to the path weight. When another master vertex y captures a vertex on p'_i, the paths P_x and P_y from the masters x and y do not have any vertex in common on p'_i. Therefore, the number of different master vertices that can capture a vertex on p'_i is limited by the amount of non-masters on the path and the number non-master vertices in the neighborhood of the path p'_i with distance less than t to p'_i. We accommodate the non-masters with distance less than t by A_t. At most $\min\{A_t/t, p'+1\}$ different master vertices reach the sub-path within the rounds $t, \ldots, t+p'$. Thus, in the worst-case each of the $\min\{A_t/t, p'+1\}$ different master vertices can increase the length by an edge weight of at most $2 \cdot (t+p')+1$. If we sum up this influence, we gain in total a new path weight of $\mathcal{O}(\min\{A_t/t, p'+1\} \cdot (t+p'))$. With $\min\{A_t/t, p'+1\} \cdot (t+p') \leq A_t+(p')^2 \leq A_t+4 \cdot k$ and the uniform and independent selection of a vertex as a master vertex with probability $1/k$, it is easy to see that $E[A_t] \leq k$ with high probability. Thus, the expected detour and resulting length of the sub-path is bounded from above by $\mathcal{O}(k)$. Now, the sub-path is increased by a multiplicative factor of at most $\mathcal{O}(\sqrt{k})$. In order to combine the results of the sub-paths, we use the linearity of expectation: we have $\Theta(d/\sqrt{k})$ sub-paths of length $\Theta(\sqrt{k})$ each. For each sub-path we have already proven that the extra detour by the path growing from the masters in parallel is bounded by $\mathcal{O}(k)$. Thus, the total detour for the complete path is $\mathcal{O}(k \cdot d/\sqrt{k} = \sqrt{k} \cdot d)$.

□

The parallel cluster growing produces a weighted graph. We use the semi-external SSSP implementation by Meyer and Osipov [MO09] and achieve an I/O-complexity of $\mathcal{O}(k \cdot \text{scan}(n+m)+\text{sort}(n+m)+\text{MST}(n, m)+\sqrt{\frac{n \cdot m}{k \cdot B}} \cdot \log_2(\frac{n}{k}))$ I/Os with $\text{MST}(n, m)) = \mathcal{O}(\text{sort}(n+m))$ using the random MST implementation. A full external version is possible using the SSSP algorithm by Meyer and Zeh [MZ03, MZ06]. In our implementation, we omit the deterministic master vertices to decrease our preprocessing time. We refer to the implementation of the parallel cluster growing algorithm by **PAR_APPROX**.

3.8 A graph class to enforce worst-case performance

In our later experimental evaluation, we challenge the parallel cluster growing diameter approximation. In order to achieve its worst-case approximation performance, we constructed a graph class to enforce bad results [Vei12, ABMV12, AMV12]. The weakness of the parallel cluster growing is the construction of the cluster connecting edges which define the shrunken representation of the graph. Even a single vertex on a direct path between two master vertices can be replaced by a larger weight. Refer to Figure 4 for an example how a third master vertex can increase the distance between two master vertices which are connected by a direct path in the original graph G.

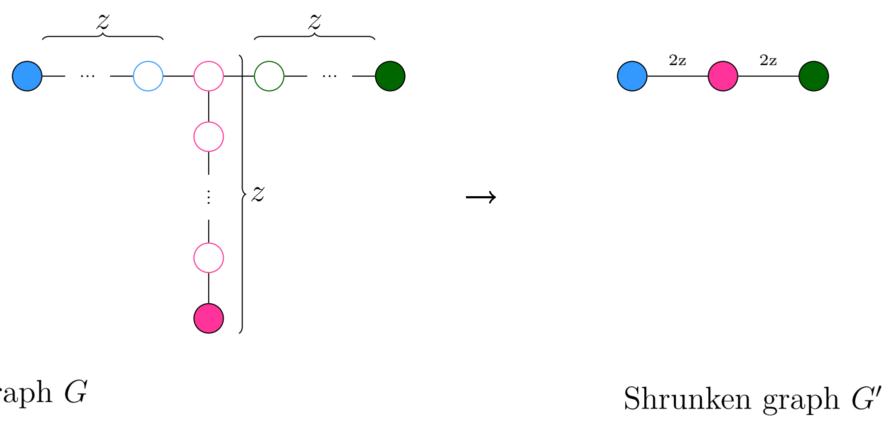


Figure 4: A worst case example for parallel cluster growing with three masters: the distance from the blue to the green cluster is increased from $2 \cdot z$ to $4 \cdot z$ through the purple cluster.

In order to create a worst-case graph class, we used the observation that clusters interfere with each other and blow up shorter paths by growing side paths into them which results in larger edge weights in the shrunken graph. Our construction starts with a long path, that essentially defines the diameter. We add side chains to this path to create a potential to blow up this path. In order to concentrate the randomly chosen master vertices at the endpoints of these side chains, we add fans at their endpoints. Each fan attracts many master vertices at endpoints of side chains. In order to avoid that many master vertices selected on the chains, the length k_1 of the side chains must not overshoot the length of the parameter k. More precisely, we chose $k_1 = k^{1/2-\epsilon}$. The size of the fans is $k_2 = k \cdot \log(n)$ so that is very likely that at least one master vertex is selected in the fan. The construction of a single long path with side chains can still fail if randomly selected master vertices cut off many side chains. Therefore, we split the long path into smaller segments with buffers in between. A segment with side chains has a path length of $k_3 = \sqrt{k}$ to ensure that $k_3 \cdot k_1 < k$. Then, the size of the buffers is $\Theta(k_1 + k_3)$. With these buffers many of the segments will succeed and increase their diameter from $k_3 + \mathcal{O}(k_1)$ to $k_3 \cdot k_1$, which results in an approximation error of $k^{1/2-\epsilon}$. For an example of such an segment, refer to Figure 5.

3.9 Experimental evaluation of PAR_APPROX

We evaluate our **PAR_APPROX** implementation empirically on different graph classes to derive its performance in practice. We aim to find a reasonable ratio between the amount of masters and vertices in the graph such that the performance of the diameter approximation is feasible on as many different graph classes as possible. In Figure 6 we have sketched the program flow of our **PAR_APPROX** implementation: we first select the master vertices, then we shrink the graph. While we shrink the graph, we memorize a correcting value c_f for the diameter. With the correcting value c_f we try to cover the case that parts of the diameter defining path(s) may be not covered by edges in the shrunken graph. The value of c_f is the maximum over all distances of the masters to the non-master vertices. Without that correcting value we might underestimate the diameter of lists and other longer structures. Note, that the expected value of c_f decreases with a growing amount of master vertices relative to the number of vertices.
After the reduction of the graph, it is checked whether the shrunken graph fits into

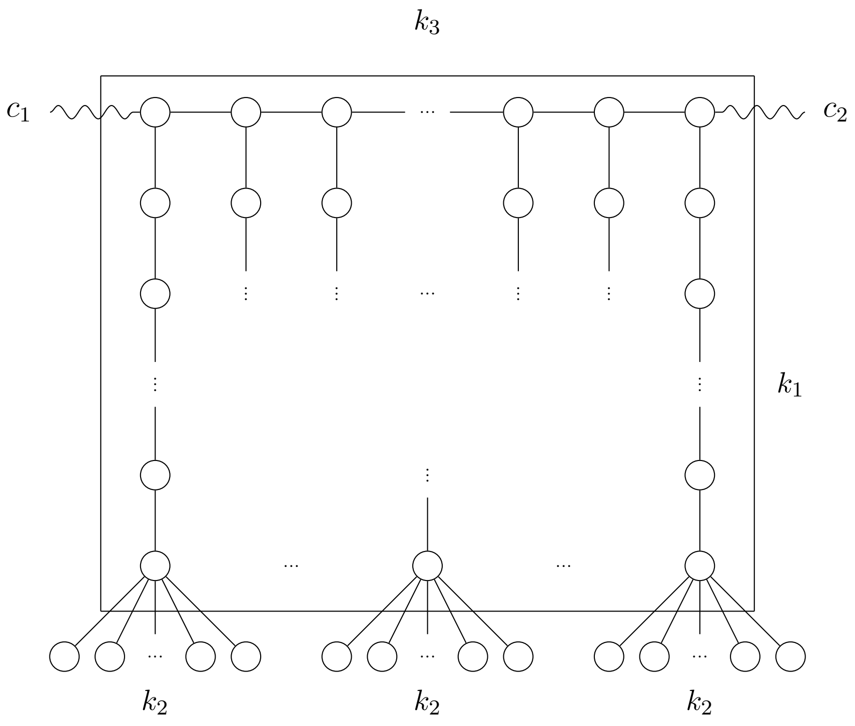
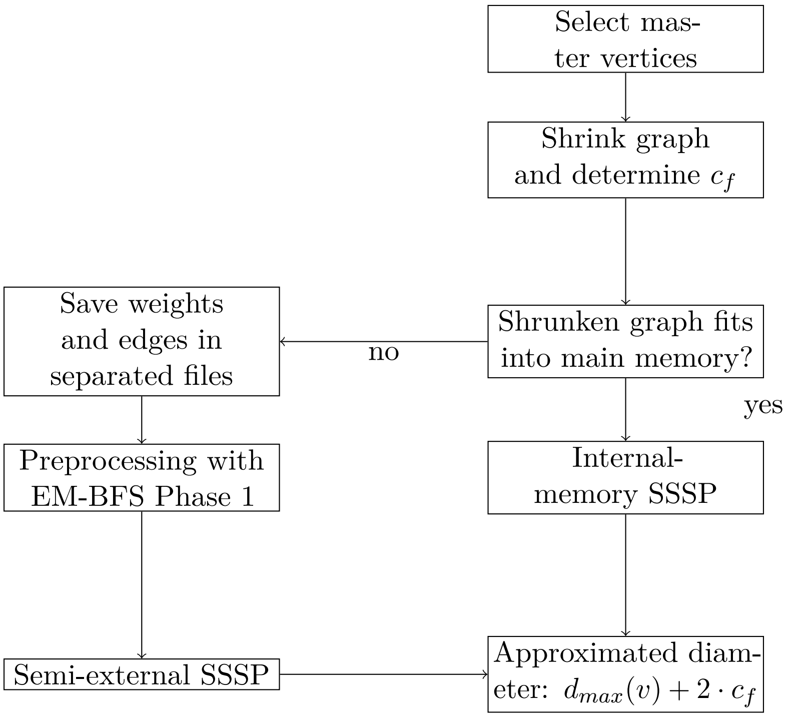


Figure 5: A sketch of a segment of the worst-case graph class for **PAR_APPROX** [ABMV12]. Many segments can be connected at the endpoints of the horizontal chain of length k_3 with buffers of size $\Theta(k_1 + k_3)$ in between.



Select master vertices
Shrink graph and determine c_f
Shrunken graph fits into main memory?
no
Save weights and edges in separated files
yes
Preprocessing with EM-BFS Phase 1
Internal-memory SSSP
Semi-external SSSP
Approximated diameter: $d_{max}(v) + 2 \cdot c_f$

Figure 6: Flowchart of our PAR_APPROX implementation [Vei12].

the main memory. If this is the case, an internal-memory SSSP implementation is run and the double sweep lower bound trick is used. The first source of SSSP is chosen at random and we improve the result in the second run by the double sweep lower bound technique.
If the shrunken graph G' does not fit into the main memory M, we have to store our data in two separated files, compute the MM_BFS clustering on G' – MST, Euler-Tour and list ranking – and then execute the semi-external memory SSSP implementation by Meyer and Osipov [MO09] (we assume, that $\frac{n}{k} < M$ which is feasible).

In order to run hundreds of experiments within a small time window on PAR_APPROX, we created an internal-memory simulation of it. We used this version to research the approximation quality on different ratios for masters and non-masters on our web graph sk-2005, our synthetic graphs $\sqrt{n}$-level graph and $\Theta(n)$-level graph with about 2^{28} vertices and worst_PAR_APPROX, our instance of the worst case graph

class with almost 2^{28} vertices, $k_1 = 10$ and $k_2 = 100$ for a single segment with $k_3 = 2,440,322$.
It is easy to see that the parameter k should be in a range that $|V'| = \mathcal{O}(n/k) < M$, if we want to use the semi-external SSSP. On the other hand, the less vertices the shrunken graph G' has, the more likely G' fits into main memory such that we can use internal-memory SSSP to evaluate G' in a few seconds without the need of further I/Os. Thus, we started experiments with different values for k on each graph class. Here are some highlights of the experimental results on the internal-memory simulation of the diameter approximation, we used to improve our external-memory version: for a larger k in the range of 2^{18}, respectively small number of 2^{10} master vertices, we have encountered 20 iterations for the parallel cluster growing on our web graph and a comparable small value of 69 iterations on the $\sqrt{n}$-level graph but also $228,249$ iterations on the $\Theta(n)$-level graph and $7,626$ iterations on our worst case graph instance. While less than 100 iterations are reasonable fast, $228,249$ iterations are definitely not viable. With a small k in the range of 64, respectively 2^{22} master vertices, the parallel cluster growing needed not more than 121 iterations on any graph. In Figure 7 we depicted the approximation ratio for various numbers of master vertices on the graphs in external memory and in Figure 8 the running time for the experiments in external memory is given. For $k = 1,024$ three graphs performed acceptable in time and approximation quality. However, the $\Theta(n)$-level graph was too slow with 41.60 hours and 1698 iterations (2^{18} master vertices). MM_BFS outperform PAR_APPROX by a factor of 8 in this case. Therefore, we only stated that a small value for k between 64 and 256 is potentially feasible.

We concluded our work on PAR_APPROX in the published study [ABMV12] with: "Our experiments have shown that the parametrized diameter approximation method is in fact faster than plain external-memory BFS and typically produces much better approximation bounds than the theory predicts. Nevertheless, it turns out that it is currently not suited as a section guide between different BFS approaches: as soon as the condensed graph does not fit into main memory, the overhead to run the semi-external memory SSSP is not worth the subsequent savings of a carefully chosen BFS approach." [ABMV12]

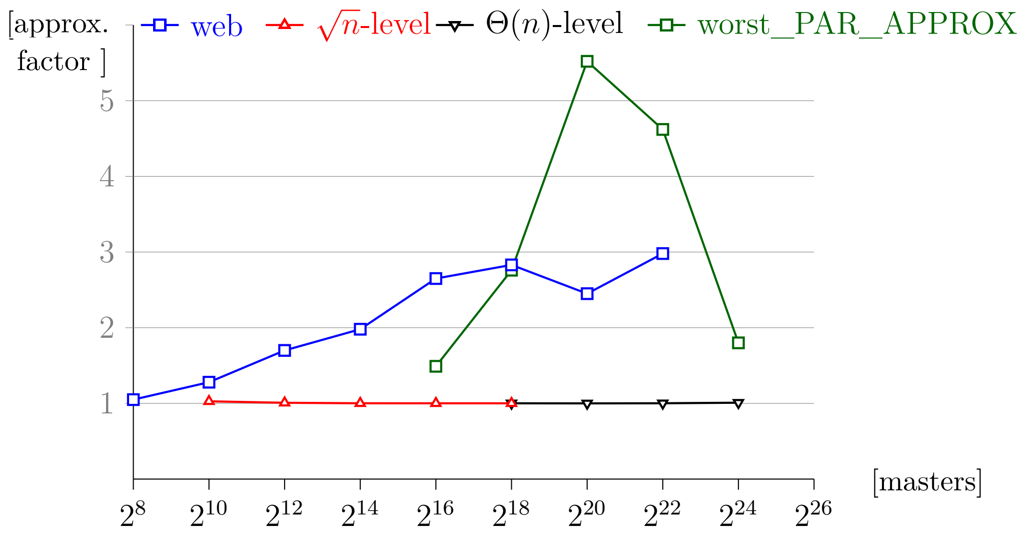


Figure 7: The approximation ratio of PAR_APPROX for different numbers of master vertices $\mathcal{O}(n/k)$, with $n = 2^{28}$. Only for the web graph, $n < 2^{26}$. Note that these are results for our external-memory implementation. The increasing approximation error for the web graph can be explained by more detours that are added to the edge weights in the shrunken graph due to the small diameter of the hub region.

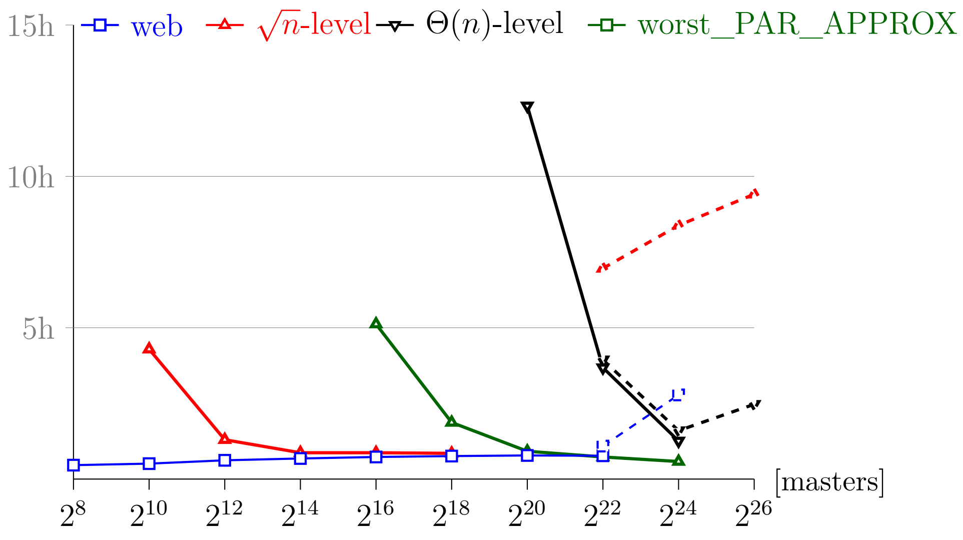


Figure 8: The execution time of PAR_APPROX depends on the number of master vertices. Note that these are results for our external-memory implementation. We skipped the value of 41.60 hours for the $\Theta(n)$-level graph with 2^{18} master vertices in this figure for a better readability. The dashed lines are from experiments we did with semi-external SSSP for the shrunken graph.

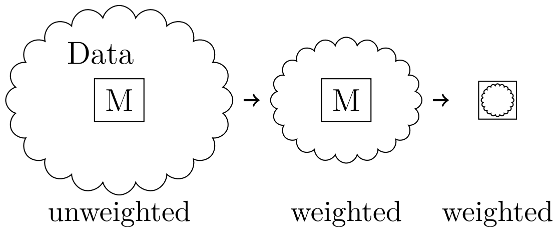


Figure 9: This figure depicts the scheme of the memory consumption we encountered after the first and the second cluster growing step with our recursive approach in many cases.

PAR_APPROX works well on graphs with small diameter, which is the case on many real-world graphs, and on structures as trees or lists. At this point we decided to tackle the challenge of extensive computation times by a recursive clustering growing. Initially, we start with a small value for k in order to keep the number of iterations small, and shrink the shrunken graph again – if needed. The next section describes our modifications towards a recursive version in more detail.

3.10 Recursive parallel cluster growing

With the implementation of **PAR_APPROX** we demonstrated that on some graph classes we can outperform MM_BFS to approximate the diameter of the graph in terms of the execution time. However, on graph classes as the $\sqrt{n}$-level graph we cannot achieve this goal. Therefore, we implemented a recursive version of **PAR_APPROX**, we call **REC_APPROX**.
The main idea behind the recursion is to start with a parallel cluster growing iteration with a large amount of master vertices in expectation.[15] Then we measure some properties of the shrunken graph and conclude parameters to shrink the graph recursively such that it fits into main memory, if this is not already the case (refer to Figure 9 for better imagination of the memory consumption).
In order to derive the value of k for the next shrinking step, respectively the number of master vertices selected on G', we include the number of vertices in the original graph n_0, the number of edges with m_0 in the original graph and the number of

[15]In our implementation we have a default value of $k = 16$, and thus n' is $n/16$ masters in expectation.

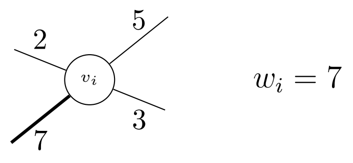


Figure 10: For each vertex we select the biggest among all its edge weights as its corresponding w_i value for our adaptive rule.

vertices in the shrunken graph n_1 as well as the number of edges m_1 in the shrunken graph. We developed an adaptive rule, such that we prefer vertices with larger edge weights to adjacent vertices as master vertices and reduce the size of the re-shrunken graph G'' to something that fits into the main memory. Hence, we keep for each vertex $v_i \in V'$ the value w_i, where w_i is the maximum under all edges adjacent to v_i. Refer to Figure 10 for an example. The value w_{min} represents the smallest value among all w_i and $S_w = \sum_i w_i$ is the sum over all w_i.
The probability for a vertex v_i to be selected as master vertex is now given by $p_i = \alpha \cdot \frac{(w_i - w_{min}) \cdot n_2}{S_w - (n_1 \cdot w_{min})}$, where $\alpha = \min \left\{1 - \left(\frac{m_1}{m_0} - \frac{n_1}{n_0}\right), 1\right\}$ and $n_2 = n_1 \cdot (m_2/m_1)$. The value of m_2 is defined by the main memory, we reserved for the edges of the shrunken graph G'' in main memory. We assume, that $((w_i - w_{min}) \cdot n_2) < (S_w - (n_1 \cdot w_{min}))$, which is the case if enough edge weights in G' are greater than w_{min}, and furthermore we assume that $n_1 \gg n_2$ and $m_1 \gg m_2$.

In 2013, Thorsten Timmer worked on a modification of **REC_APPROX** with additional queues to sort the integer edge weights such that for example an outgoing edge with weight 5 waits for 4 rounds before it is considered for cluster growing. It turned out that this idea only slightly improves the diameter approximation quality on some graphs [Tim13] while it increases the running time by a significant factor. So we stick to the idea of ignoring the weights for the cluster growing, while we count them for the cluster edge weights for the resulting shrunken graph. In Section 3.12 we present two minor improvements, we added to the recursive implementation in order to further improve the approximation quality.

3.11 Approximation error of REC_APPROX

While we have proven an upper bound for the approximation error of $\mathcal{O}(\sqrt{k})$ for simple paths using **PAR_APPROX**, we did not attempt to prove such an upper bound for the approximation error of our recursive approach. So, we designed a graph pattern according to the worst case graph pattern of **PAR_APPROX** for two REC_APPROX with two iterations. For this new pattern we have proven a lower bound for the approximation error of $\Omega(k^{\frac{4}{3}-\epsilon})$ for **REC_APPROX**, when two cluster growing steps suffice to shrink the graph such that it fits into main memory [Vei12, ABMV12] and no improvement for the selection of master vertices is used (refer to Section 3.12).
We reuse the basic pattern of a long path with side chains that will grow into this path with high probability. In order to retain this behavior we extended the side chains by larger structures, we describe as follows:

- **x-fan (Figure 11)**: the root v can be connected to the left and to the right, e.g. to be part of a longer path. The root v has x vertices as children, where these x vertices are only connected to v. In total this element has $x+1$ vertices with $x = \Omega(k \cdot \log(n))$.

- **x-double-fan (Figure 12)**: the root v can be connected to the left and to the right, e.g. to be part of a longer path. The root v has x children $v_1 \dots v_x$. Each of these children v_i connects to another child w_i and each w_i in turn has x further children $z_{i,j}$ with $j \in [1, x]$. The x-double-fan comprises x^2+x+1 vertices in total and $x = \Omega(k \cdot \log(n))$.

- **side-chain block (Figure 13)**: a side-chain block is a list of length $2 \cdot k^{1/3}+1$. The vertex in the center of the list is denoted as u. The center u itself is the root of another path with $k^{5/3-\varepsilon}$ vertices and at the end of this path is an x-double-fan. The other $2 \cdot k^{1/3}$ vertices on the list, except the center u, are the root of an x-fan each. The side-chain block has $2 \cdot k^{1/3} \cdot (x+1)+k^{5/3-\varepsilon}+x^2+x+1$ vertices in total.

- **basic block (Figure 14)**: a basic block is assembled by $k^{1/3}$ side-chain blocks. In addition, a list of $k^{2/3}$ x-fans to the left and to the right is connected. At the endpoints to the left and to the right there is an x-double-fan as a buffer.

- **$\Omega(x)$-fan (Figure 15)**: the $\Omega(x)$-fan is the remaining structure of the x-double-fan during the parallel cluster growing. Although the structure of this subgraph is randomly determined, it is similar to an x-fan with high probability.

Our graph is now constructed by connecting y basic blocks to a long path, where each segment grows independently. The diameter is then bounded by $\mathcal{O}(y \cdot k^{2/3} + k^{5/3-\varepsilon})$. If $y > k$ then the first term dominates. We chose x in the order of $\Theta(k \cdot \log(n))$. In the first round a vertex is selected as a master vertex with probability $\frac{1}{k}$. After the first round of shrinking, the shape of the basic block is transformed into some list with several $\Omega(x)$-fans with high probability. The chain from the side-chain block will be a list with a length in the range of $k^{2/3-\varepsilon} \pm k^{1/3}$ with high probability. Refer to Lemma 3.5 for a proof. A possible shape of a shrunken basic block is depicted in Figure 16.

Lemma 3.5. *The length of the list in a shrunken side-chain block remains in the interval of $k^{2/3-\varepsilon} \pm k^{1/3}$ with probability $1 - 2 \cdot e^{-\frac{k^{\varepsilon}}{2}}$.*

Proof. Each vertex v_i in the list of the side-chain block in G is represented by the random variable X_i. If v_i is selected as a master vertex, $X_i = 1$ and else $X_i = 0$. We denote δ as the deviation of the expected list length $\mu = k^{2/3-\varepsilon}$. We want to bound the deviation of the list length by $\pm k^{1/3}$.

We first bound the deviation to the right: the Chernoff bound (refer to [Gon07] for details) is then $P\left[\sum_{i=1}^{n} X_i \geq (1+\delta) \cdot \mu\right] \leq e^{-\frac{\min\{\delta,\delta^2\} \cdot \mu}{3}}$. For the additive deviation we set $(1+\delta) \cdot \mu = \mu + k^{1/3}$ and therefore, $\delta = \frac{k^{1/3}}{\mu}$.

$$e^{\frac{-\min\left\{\delta,\delta^2\right\}\cdot\mu}{3}} = e^{\frac{-\min\left\{\frac{k^{1/3}}{\mu},\frac{k^{2/3}}{\mu^2}\right\}\cdot\mu}{3}} = e^{\frac{-\min\left\{k^{1/3},\frac{k^{2/3}}{\mu}\right\}}{3}} = e^{\frac{-\min\left\{k^{1/3},\frac{k^{2/3}}{k^{2/3-\varepsilon}}\right\}}{3}} = e^{\frac{-\min\left\{k^{1/3},k^{\varepsilon}\right\}}{3}} = e^{\frac{-k^{\varepsilon}}{3}}, \text{ for } 0 < \varepsilon \leq k^{1/3}.$$

The Chernoff bound for deviation to the left is $P\left[\sum_{i=1}^{n} X_i \leq (1-\delta) \cdot \mu\right] \leq e^{-\frac{\delta^2}{2} \cdot \mu}$. Using the same technique to transform the multiplicative bound to its subtractive

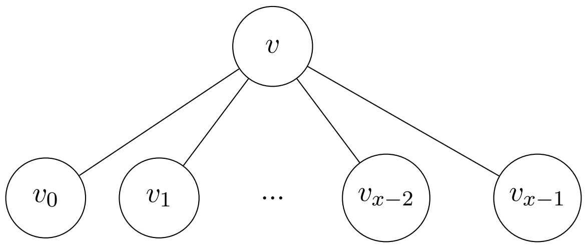


Figure 11: The x-fan [Vei12].

form, we again get $\delta = \frac{k^{1/3}}{\mu}$.

$e^{-\frac{\delta^2}{2}\cdot\mu} = e^{-\frac{k^{2/3}}{2\cdot\mu^2}\cdot\mu} = e^{-\frac{k^{2/3}}{2\cdot\mu}} = e^{-\frac{k^{2/3}}{2k^{2/3-\varepsilon}}} = e^{-\frac{k^{\varepsilon}}{2}}$.
We condense the bounds to the left and to the right by twice the bound to the left and the complementary probability is our probability $1-2\cdot e^{-\frac{k^{\varepsilon}}{2}}$ that the length of the chain in the shrunken graph remains in the interval $k^{2/3-\varepsilon} \pm k^{1/3}$. $\square$

The second iteration of recursive cluster growing lets the side chains grow into the main chain of the side-chain blocks. A possible shape for the graph after the second iteration is provided in Figure 17. The diameter of each basic block can be increased from $\Omega(k^{2/3})$ to $\Omega(k^{1/3}\cdot k^{5/3-\varepsilon}) = \Omega(k^{2-\varepsilon})$. So the expected diameter of G'' is $\Omega(y\cdot k^{2-\varepsilon})$ which causes an expected approximation error of at least $k^{4/3-\varepsilon}$.

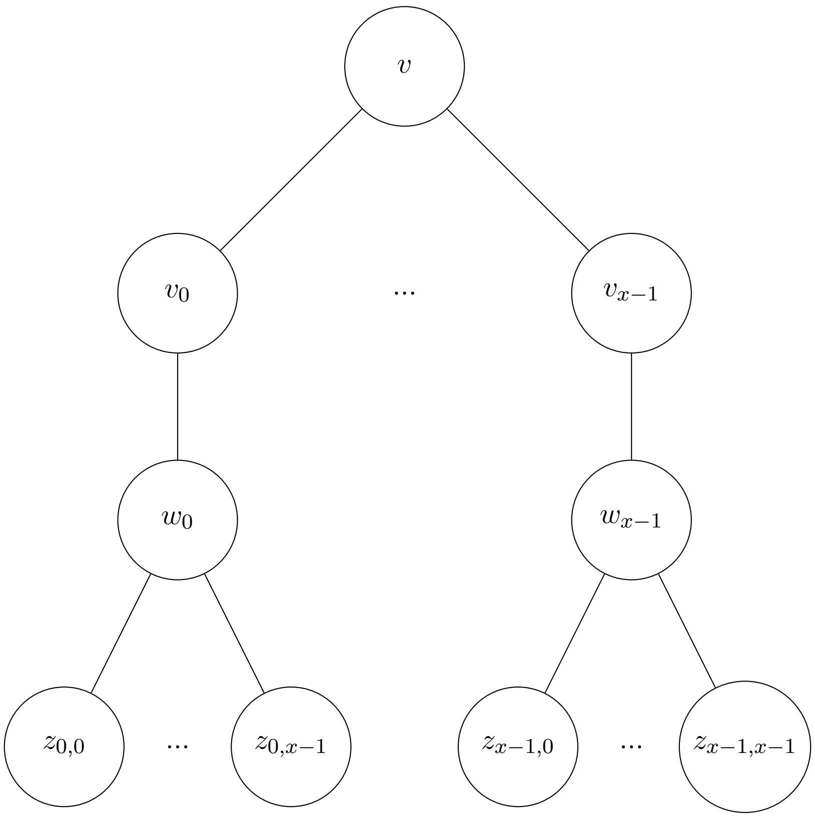


Figure 12: The x-double-fan [Vei12].

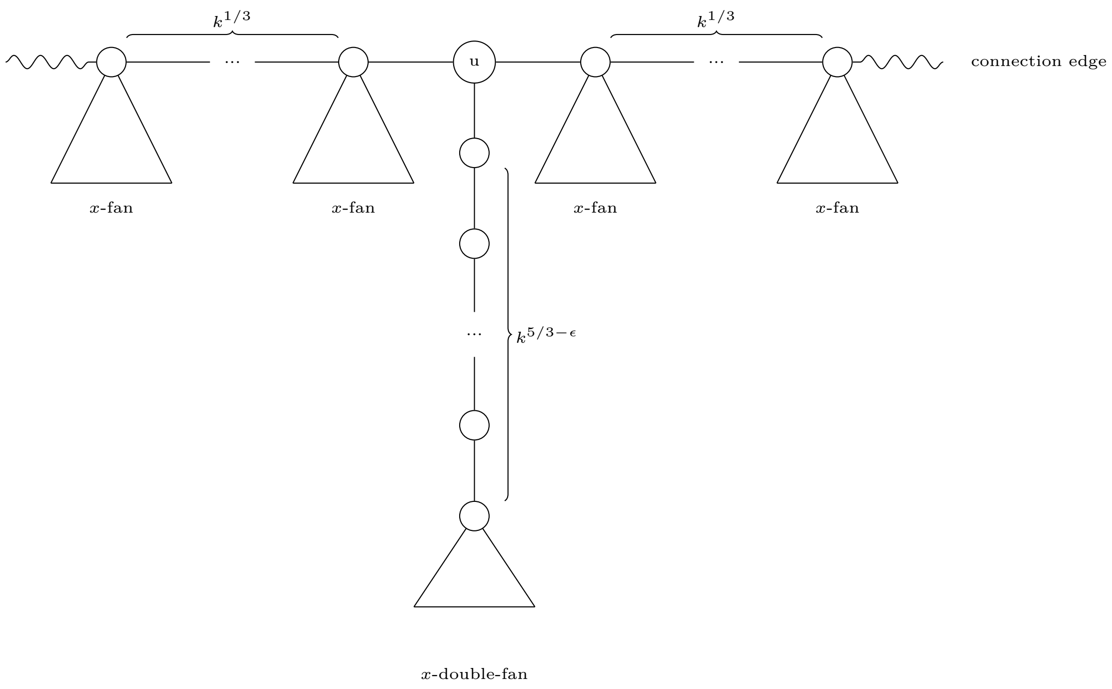


Figure 13: A side-chain block as a subelement we need to generate our worst-case graph class [Vei12].

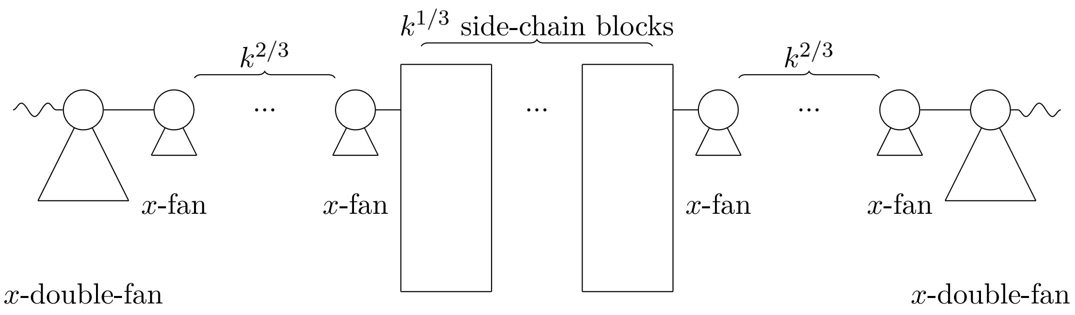


Figure 14: The basic block as an element for our worst-case graph class for REC_APPROX [Vei12].

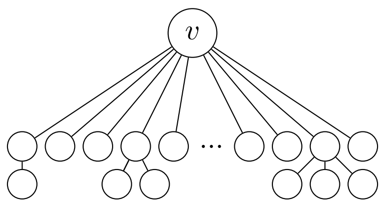


Figure 15: A possible shape of the $\Omega(x)$-fan after the first cluster growing. Note that the shape of the $\Omega(x)$-fan is random [Vei12].

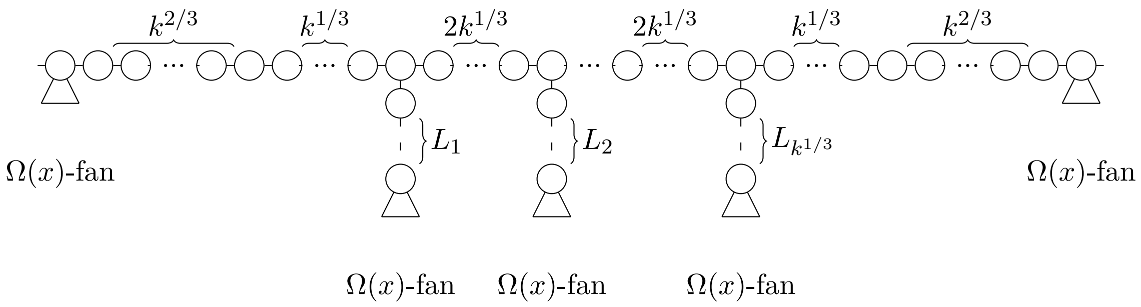


Figure 16: A possible shape of a basic block after the first cluster growing step[Vei12].

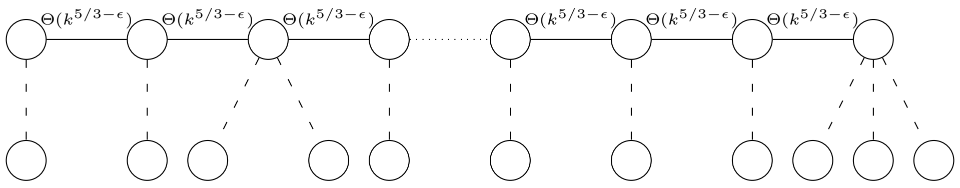

$\Theta(k^{5/3-\epsilon})$
$\Theta(k^{5/3-\epsilon})$
$\Theta(k^{5/3-\epsilon})$
$\Theta(k^{5/3-\epsilon})$
$\Theta(k^{5/3-\epsilon})$
$\Theta(k^{5/3-\epsilon})$

Figure 17: A possible shape for the graph after the second clustering. The largest path is determined by the horizontal baseline [Vei12].

3.12 Two improvements of REC_APPROX: tie breaking and cluster centering

In order to improve the approximation quality of REC_APPROX in the context of weighted shrunken graphs, we added two tricks. We improved the tie breaking rule by assigning non-master vertices in case of conflict to the master vertex with minimum distance. Note, that such a tie breaking rule is not needed for PAR_APPROX, which operates on unweighteed graphs, because two master vertices can only compete for one vertex in the same iteration. For an example of the tie breaking rule in a recursive step of REC_APPROX refer to Figure 18.

The second improvement aims to reduce the influence of clusters that grow into longer paths. As mentioned in the constructions of our worst-case graph classes we encounter bad performance by side chains with huge fans in which most of the master vertices emerge with high probability (refer to Figures 5 and 13). In order to reduce the impact of such structures in a graph, we move the master of each cluster to its center after the clustering has been finished. This reduces the edge weights between the clusters by a constant factor in some cases. An example is given in Figure 19.

The I/O-costs for moving the masters to the center of their clusters are bounded by an additional sorting step. Each cluster is loaded into main memory, the center is approximated using internal-memory SSSP from the master and a second SSSP run from one vertex with farthest distance to the master, similarly to the double

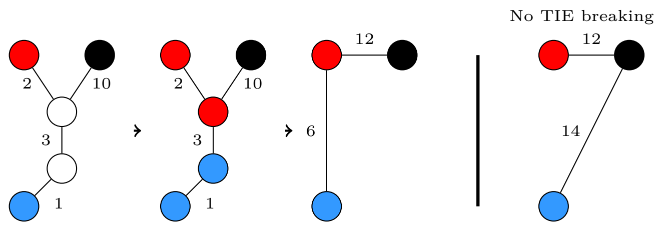


Figure 18: With active tie breaking, the algorithm assigns a non-clustered vertex to the cluster with smallest distance to the master.

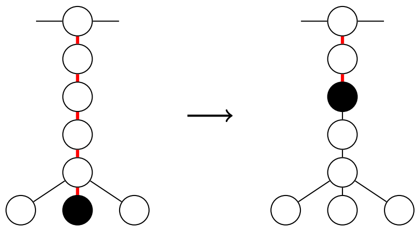


Figure 19: After the clustering is computed, the master vertices are moved to the center of their cluster in order to reduce the possible impact on paths they might interfere with.

sweep lower bound technique (refer to Section 3.2). On the middle of the longest path in the tree of the second SSSP-tree we select a vertex as the new master of the cluster.

3.13 Experimental evaluation of REC_APPROX

We executed experiments on four graph instances: our web graph sk-2005, an instance of the $\sqrt{n}$-level graph with a diameter of 46,342 and a size of 128 GB, an instance of our $\Theta(n)$-level graph with a diameter of 536,870,913 and a size of 83.8 GB and an instance of our worst-case graph class for REC_APPROX: worst_2step with a diameter of 8,111 and a size of 31.9 GB. While the web graph has about 51 million vertices, the other graphs have more than 2.1 billion vertices.

In Table 4 we give an overview of the computation time that REC_APPROX achieved on our graphs. REC_APPROX outperforms the execution time of external-memory MM_BFS on every graph. In addition, the approximation quality of REC_APPROX is still reasonable and close to the quality we achieved with PAR_APPROX.
In order to check the performance of our improvements on REC_APPROX, we analyzed how the approximation error grows with k. We generated a set of worst-case graph instances for REC_APPROX with 2^{28} vertices and a doubled value for the graph parameter k. Note, that the size of the x-fans is $x = \lceil k \cdot \log_2 n \rceil$ (refer to Section 3.11). The diameter approximation error is provided in Figure 20 for an increasing number of master vertices (probability for each vertex to be selected

	HIER-time	**ratio**	BFS-time (single)	ratio
sk-2005	**0.6 h**	3.3	5.6 h (3.2 h)	$\sim$ 1.0
$\sqrt{n}$-level graph 2	**9.0 h**	$\sim$ 1.0	56.6 h (37.3 h)	1.0
$\Theta(n)$-level graph 2	**3.6 h**	$\sim$ 1.0	27.8 h (19.0 h)	1.0
worst_2step	**1.4 h**	3.1	15.3 h (11.6 h)	1.0

Table 4: Performance of REC_APPROX on various graph classes [AMV12]. Note that the diameter approximation ratio is provided including the improved tie breaking rule and the master movement routine. The BFS-time includes the double sweep lower bound heuristic.

	Exact	Basic	Adaptive	Tie break	Move	All
sk-2005	40	185	196	173	182	168
ratio		4.625	4.900	4.325	4.550	4.200
$\sqrt{n}$-level graph	16385	16594	16416	16604	16597	16408
ratio		1.013	1.002	1.013	1.013	1.001
$\Theta(n)$-level graph	67108864	67212222	67131347	67212123	67174264	67131036
ratio		1.002	1.000	1.002	1.001	1.000
worst_2step_k32	3867	138893	38643	137087	17321	13613
ratio		35.918	9.993	35.450	4.479	3.520

Table 5: The approximation quality of our improvements on REC_APPROX on the four graph classes [AMV12] compared to other configurations. Note, that the $\sqrt{n}$- and $\Theta(n)$-level graphs are the smaller versions due to the time for a run and the missing impact of the improvements on these graph classes. The graph worst_2step_k32 is another instance of our worst-case graph class with $|V| = 2^{28}$, $k = 32$ and thus a size of $x = 896$ for the fans (refer to Section 3.11).

as master vertex). In Figure 21 we depicted the distribution of the distance of the vertices to their masters for the graph worst_2step_k32 with and without our improvements. With our improvements the vertices have a smaller distance to their master on average.

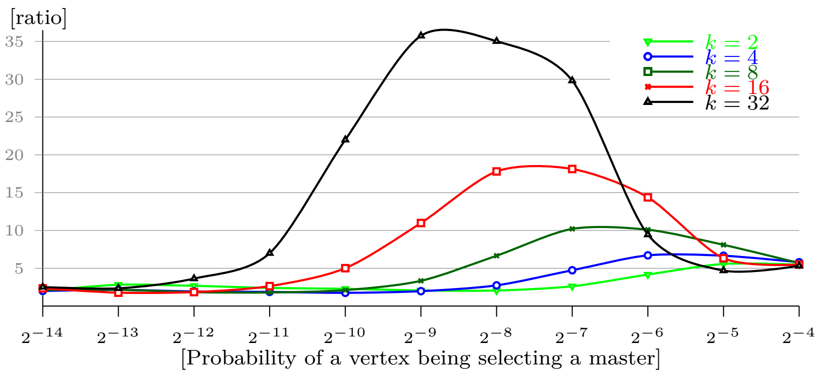


Figure 20: Approximation error for several instances of the REC_APPROX worst case graph class with $|V| = 2^{28}$, $\varepsilon = 0.025$ and growing values for k [AMV12].

3.14 Conclusion

We have developed an external-memory diameter approximation method, which can handle huge data sets in appropriate time with feasible approximation quality and has only little hardware requirements.

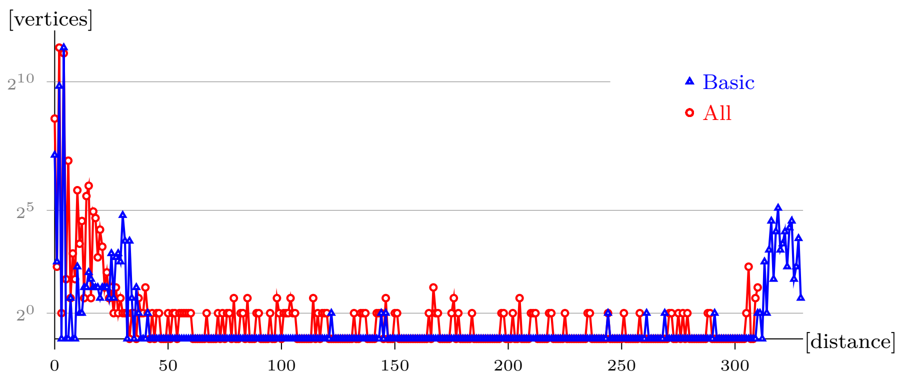


Figure 21: The distance distribution of the vertices to their masters on worst_2step_k32 after the second cluster growing - in the basic approach with no improvements and a second line with all extensions [AMV12].

4 Dynamic Breadth-First Search

Preliminaries Some parts of this chapter have already been published at ESA 2013 [BMV13].

4.1 Introduction

Initially we give a definition of the external-memory Dynamic Breadth-First Search problem:

Definition 4.1. *The Dynamic Breadth-First Search problem (short **D-BFS**) deals with the following issue: given a graph $G = (V, E)$ with undirected and unweighted edges and a BFS tree T for G from a fixed source $s \in V$, the graph G is transformed into a new graph G' by an edge insertion or an edge deletion. The task is now to compute the BFS tree T' for source s on G', on the base of T in a way that needs less I/Os than by a recalculation T' from scratch.*

The idea to dynamically update the BFS tree after a single update plays a minor role in internal memory. This is due to the fact that the worst-case RAM time complexity is the same for the dynamic as for the static case. We discuss this in Section 4.2 in more detail. However, the situation changes for the I/O-complexity in the external-memory setting. In 2008 Meyer [Mey08a] published a dynamic BFS algorithm which is able to transform the BFS tree T of the original input graph G into the BFS tree T' for the updated graph G' with asymptotically less I/Os than the static MM_BFS, which is the currently best external-memory BFS algorithm for general undirected graphs (see Section 2.11). An upper bound of $\mathcal{O}(n/B^{2/3}+\text{sort}(n)\cdot\log(n))$ I/Os on sparse graphs with high probability per update was proven for n consecutive edge insertions [Mey08a]. In the following, we examine how a practical implementation of dynamic BFS behaves. In the work of Meyer, no statements about the constant factors as well as the behavior on individual updates are included. We research for which graphs the dynamic algorithm provides a speed up even on single updates and in which cases we have to compute many or batched updates to gain a benefit from the dynamic implementation compared to the static one.

In order to implement the dynamic BFS algorithm, we had to implement a new clustering algorithm for the preprocessing to improve the clustering used in the original MM_BFS publication (refer to Section 4.3 for further details). In addition, we need a dynamic reorganization of the clustering. The spadework for the new clustering was done by Beckmann and Meyer within a first draft in 2009. A first prototype was implemented by Asmaa Edres in her master thesis in 2012 in C++ using the external-memory C++ library STXXL [Edr12]. In order to improve the performance we implemented a streamed version of the clustering and adapted the interfaces to our implementation project that resulted in our ESA 2013 publication about D-BFS. We rewrote the implementation with several optimizations in order to reduce our main memory consumption whenever possible, avoiding unnecessary context switches. Besides the streaming operations we used many ideas from time-forward processing [CGG+95]. Details about the reimplementation are given in Section 4.3.
We mainly focus on edge insertions in this thesis. In 2016 Fabian Knöller wrote a bachelor thesis about the case of edge deletions [Knö16]. Knöller analyzes in more detail the various cases an implementation has to regard during the deletion of an edge. Furthermore, a scheme for the implementation of edge deletion is designed and a prototype is evaluated. Details can be found in Section 4.5.

4.2 Dynamic BFS: the algorithm

Figure 22 depicts an instance of a graph class where each update requires $\Omega(n)$ BFS-level updates if the edge updates are inserted as alternating shortcuts. In the figure the dashed lines are edge insertions from s to u_1,u_2,u_3,u_4,u_5, and u_6 in this fixed order. With each of theses updates the distance from the start vertex S to each vertex in X respectively in Y is shrunken. The respective distances from s to all vertices in Z are decreased by one in each update step. If the size of the subsets $|X|$, $|Y|$ and $|Z|$ in the graph satisfy $\Omega(n)$, we enforce $\Omega(n)$ BFS-level updates. In this scenario a dynamic BFS does not obtain any cost saving in the computation time. Therefore we concentrate on the reduction of the I/O-complexity. We can try to avoid I/Os by using the knowledge of the graph structure we have from the previous BFS results. We write this down in Observation 4.2.

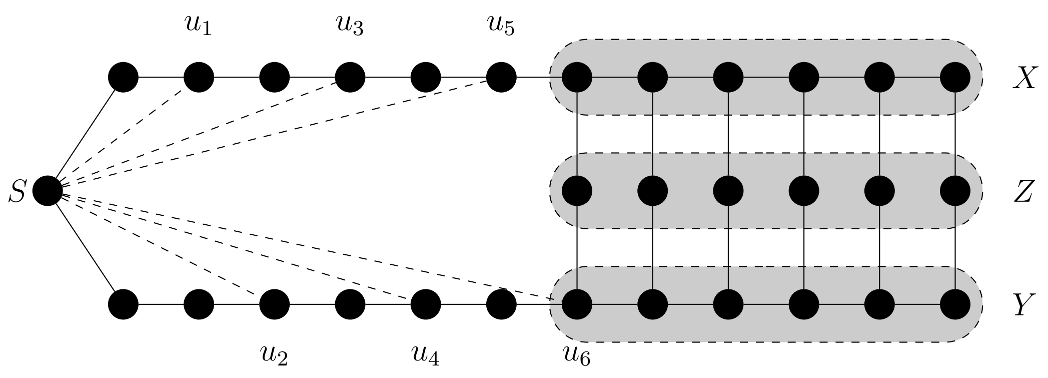


Figure 22: A graph with two lines and a block element of three sets as an example for an instance of a graph class where each update requires $\Omega(n)$ BFS-level updates if the edge updates are inserted as alternating shortcuts.

Observation 4.2. *The number of potential changes of BFS-levels in a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ and a fixed start vertex s for one edge insertion is bounded by $\Omega(n)$. Refer to Figure 22 for an example.*

Both, algorithm and implementation of dynamic BFS are based on modifications of the MM_BFS reference implementation by Ajwani et al. [AMO06]. During the computation of the BFS tree, the MM_BFS implementation feeds several clusters of adjacency lists into its hotpool H. The hotpool keeps track of the graph data of components that are reached in the near future. The near future is defined by the maximal diameter of each cluster, called μ. In Section 2.11 we argued that the optimal trade-off for the value of μ on sparse graphs is $\sqrt{B}$. In the dynamic setting we use the knowledge about the structure (BFS-levels) of the vertices in a cluster of the previous BFS result.

Notation 4.3. *In a sequence of n edge insertions $e_1, \ldots, e_n$ we denote the insertion of the i-th edge (u, v) to the graph by $G_i(d_i)$. The variable d_i denotes the depth of the BFS tree in the graph G_i. The value $d_i(v)$ denotes the BFS-level of vertex $v \in V$ in G_i. Note, that the value of d_i respectively all $d_i(v)$ are unknown before* *the dynamic BFS is calculated.* $G_0(d_0)$ *represents the original graph* $G = (V, E)$ *without any modifications.*

As mentioned in the introduction of this chapter we mainly focus on edge insertions in this thesis. If we deal with edge deletions, it is explicitly mentioned in the corresponding paragraph.

Observation 4.4. *To update the BFS tree of* $G_i(d_i)$ *after an edge insertion into* $G_{i-1}(d_{i-1})$*, we can simply keep a data window of size* $d_{i-1}(v^*) - d_i(v^*) = \mathcal{O}(n)$ *BFS-levels in the hotpool, where the vertex* v^* *is the vertex with the largest level change in the updated BFS tree. We use the parameter* $\alpha > 1$ *to estimate the data window size.*

If we follow the idea presented in Observation 4.4, we can construct a dynamic BFS algorithm in external memory quite simple. However, as already explained in Notation 4.3, the correct value for $d_i(v^*)$ is unknown in the beginning and thus the same holds for the data window size. Furthermore, $d_{i-1}(v^*) - d_i(v^*)$ is bounded by $\mathcal{O}(n)$. We can derive two main targets the dynamic implementation has to fulfill:

1 We have to keep track of the consumed I/Os. If we consume too many I/Os, α was too small and we have to adapt the data window size.

2 If we increase the estimator for α, we have to adapt our clustering to the new alpha within at most sorting complexity.

We deal with the first target by assigning a small starting value to α for the cluster size and so the levels we look ahead. We fix the size of our clusters to be in the range of $\alpha = 2^1, 2^2, 2^3, \ldots, \sqrt{B}$. Clusters of size larger than $\sqrt{B}$ contradict the observation of MM_BFS that cluster diameters greater than $\sqrt{B}$ violate our trade-off between the cluster size and the long residence time of clusters in the hotpool (refer to Section 2.11 for details). On the other hand, clusters of a too small size are not efficient due to the overhead by maintaining the cluster while it has only a few elements. In theory, we can use the same parameter α for prefetching levels into a hotpool and the cluster size. In practice, we split α into two values, namely α_1 and α_2, in order to separately maintain the growing cluster size and the levels we load into the hotpool. We refer to Section 4.7 for details about the values for α we used for our experiments.

For performance reasons, we have to keep track of the random I/Os, caused by cluster fetches into the hotpool. We allow up to $\alpha \cdot n/B$ random I/Os. If the number of allowed random I/Os is exceeded, the original algorithm by Meyer [Mey08a] stops, doubles the value of α and restarts the computation from scratch. In our implementation, we continue the calculation with the increased α, reinitialize the hotpools from the current level and keep the already computed results.

Later, we extended our implementation to split α into two distinct parameters α_1 and α_2. The number of simultaneously levels which are kept in the static hotpool H is denoted by α_1, whereby the diameter of the clusters that are fetched into the dynamic hotpool HC is limited by α_2. This is helpful in order to keep the number if random I/Os small by having larger clusters. For the insertion of a new edge into the graph, we determined the first level in which changes can happen so that we start prefetching our α BFS from this level (refer to Lemma 4.5).

Lemma 4.5. *Using the notation of Definition 4.1, let $G = (V, E)$ and T be given as well as a new edge $\{v_1, v_2\}$, i.e. $G' = (V, E \cup \{v_1, v_2\})$. For each vertex $v \in V$ let $d(v)$ denote the BFS-level of v in T and $d'(v)$ its BFS-level in T'. Let $l_1 := d(v_1)$ and $l_2 := d(v_2)$ be the BFS-levels of the vertices v_1 and v_2 in T, and w.l.o.g. be $l_1 \leq l_2$. Furthermore, let l_0 be the smallest BFS-level in T, at which a vertex v_0 exists with $d'(v_0) < d(v_0) = l_0$. We can calculate the BFS-level l_0 by $l_0 = l_1 + \lceil \frac{l_2 - l_1}{2} \rceil + 1$.*

Proof. Let v be a vertex with BFS-level $d(v) \leq l_2$ such that $d'(v) < d(v)$. Then there exist non-negative integers x, y, such that $d(v) = l_1 + x = l_2 - y$. Since the BFS-level of v is decreased after inserting the new edge, the length of any shortest path P between v_2 and v in G is at least y (otherwise we would get $d(v_2) < l_2$ by combining P with the shortest path between s and v in G) and therefore it holds that

$$d(v) = l_1 + x > d'(v) \geq l_1 + 1 + y \;\Leftrightarrow\; x - y > 1 \;\Leftrightarrow\; x - y \geq 2$$

Moreover, since $y \geq 0$, it follows that $x \geq 2$ and we obtain $l_2 \geq l_2 - y = l_1 + x \geq l_1 + 2$ as a necessary condition for the existence of such a vertex v. However, the condition $l_2 - l_1 \geq 2$ is also sufficient because it holds that $d'(v_2) = l_1 + 1 < l_2$.

For the rest of the proof let us assume that $l_2 - l_1 \geq 2$. Now let v_0 be a vertex with BFS-level $d(v_0) = l_0$ and $d'(v_0) < d(v_0)$ (clearly, at least one such vertex exists, namely v_2 or one of its predecessors in T), then there exist non-negative integers x_0, y_0, such that $l_0 = l_1 + x_0 = l_2 - y_0$ (and therefore $x_0 - y_0 \geq 2$ as shown above) and we have

$$\begin{aligned} l_0 \leq d(v) \ &\Leftrightarrow\ l_1 + x_0 \leq l_1 + x \ \Leftrightarrow\ x_0 \leq x \\ l_0 \leq d(v) \ &\Leftrightarrow\ l_2 - y_0 \leq l_2 - y \ \Leftrightarrow\ y_0 \geq y \end{aligned}$$

for any vertex v with $d'(v) < d(v)$ and associated integers x, y as above. Using these inequalities we obtain $2 \leq x_0 - y_0 \leq x - y$, and by combining this result with the definition of l_0, we can deduce that either $x_0 - y_0 = 2$ (if $l_2 - l_1$ is even) or $x_0 - y_0 = 3$ (if $l_2 - l_1$ is odd) holds (otherwise we could simply replace v_0 by one of its predecessors in T in contradiction to the definition of l_0).

Case 1: $x_0 - y_0 = 2$
On the one hand we have the equation $l_0 = l_1{+}x_0$, and on the other we have the equation $l_0 = l_2{-}y_0 = l_2{-}x_0{+}2$. By adding these two equations we obtain

$$\begin{aligned} & 2 \cdot l_0 = l_1 + x_0 + l_2 - x_0 + 2 = l_1 + l_2 + 2 \\ \Leftrightarrow \qquad & l_0 = \frac{l_1 + l_2}{2} + 1 = l_1 + \frac{l_2 - l_1}{2} + 1, \end{aligned}$$

which is the desired result.

Case 2: $x_0 - y_0 = 3$
Analogously, we have the equations $l_0 = l_1{+}x_0$ and $l_0 = l_2{-}y_0 = l_2{-}x_0{+}3$. Again, by adding these two equations we obtain the desired result

$$\begin{aligned} & 2 \cdot l_0 = l_1 + x_0 + l_2 - x_0 + 3 = l_1 + l_2 + 3 \\ \Leftrightarrow \qquad & l_0 = \frac{l_1 + l_2}{2} + \frac{3}{2} = l_1 + \left\lceil \frac{l_2 - l_1}{2} \right\rceil + 1, \end{aligned}$$

where the last equality holds because $l_2 - l_1$ is odd in this case, as mentioned above. □

Alternatively we can prove the Lemma by using some properties of the BFS tree T as follows:

Proof. For each vertex $v \in V$ let $P(v) = (w_0(v), \ldots, w_{d(v)}(v))$ denote the (unique) path of length $d(v)$ from the source s to v in the BFS tree T (i.e. $w_0(v) = s$, $w_{d(v)}(v) = v$, $w_i(v) \neq w_j(v)$ for $i \neq j$, and $\{w_i(v), w_{i+1}(v)\} \in E$ for all i). Observe that for each vertex v the path $P(v)$ is a shortest path in G, and that each neighbor u of v in G has a BFS-level $d(v){-}1 \leq d(u) \leq d(v){+}1$ in T. Furthermore, after inserting the edge $\{v_1, v_2\}$, the path $P(v_1)$ still remains a shortest path in G' (otherwise there would exist a path from s to v_2 of length at most $d(v_1){-}1$ in G contradicting $l_1 \leq l_2$) and therefore it holds that $d'(v_1) = l_1$.

On one hand, we still have the path $P(v_2)$ of length l_2 in G', which is a shortest path in G, and on the other we can now extend the shortest path $P(v_1)$ in G' with the edge $\{v_1, v_2\}$ and obtain the path $P'(v_2) := (s, \ldots, v_1, v_2)$ of length $l_1{+}1$. Hence, at least one of these two paths is a shortest path in G' which allows us to determine $d'(v_2)$: if $P(v_2)$ is a shortest path in G', then we have $d'(v_2) = l_2 \leq l_1{+}1$ (and thus $T' = T$), else we have $d'(v_2) = l_1{+}1 < l_2$ and therefore $l_1 \leq l_2{-}2$ as a necessary and sufficient condition for the existence of a BFS-level l_0 (and thus $T' \neq T$).

Now in order to calculate l_0 let us assume that $l_1 \leq l_2{-}2$. As we have seen above, in this case we have $d'(v_2) = l_1{+}1 < l_2$ and therefore $l_1{+}1 \leq d'(u) \leq l_1{+}2$ for each neighbor u of v_2 (since we have $l_1{+}1 \leq l_2{-}1 \leq d(u)$ and we can extend the path $P'(v_2)$ by the edge $\{v_2, u\}$ to obtain a path of length $l_1{+}2$ in G'). However, as we want to calculate l_0, we are only interested in vertices v with BFS-levels $l_1 < d(v) < l_2$, therefore it suffices to observe the vertices along the path $P(v_2)$ (in reverse order) between the BFS-levels l_2 and l_1. For each such vertex $w_{l_2-i}(v_2)$ the shortest path $P(w_{l_2-i}(v_2))$ in G is of length $l_2{-}i$, while the path $P'(w_{l_2-i}(v_2)) := (s, \ldots, v_1, v_2, w_{l_2-1}(v_2), \ldots, w_{l_2-i}(v_2))$ in G' (an extension of $P'(v_2)$) is of length $l_1{+}1{+}i$. Thus $d'(w_{l_2-i}(v_2)) < d(w_{l_2-i}(v_2))$ holds if and only if

$$l_1{+}1{+}i < l_2{-}i \quad \Leftrightarrow \quad i < \frac{l_2 - l_1 - 1}{2}.$$

Since l_0 is the smallest of these BFS-levels, we obtain

$$l_0 = \left\lfloor l_2 - \frac{l_2 - l_1 - 1}{2} \right\rfloor + 1 = \left\lfloor \frac{l_2 + l_1 + 1}{2} \right\rfloor + 1 = \left\lfloor l_1 + \frac{l_2 - l_1 + 1}{2} \right\rfloor + 1$$
$$= l_1 + \left\lfloor \frac{l_2 - l_1 + 1}{2} \right\rfloor + 1 = l_1 + \left\lceil \frac{l_2 - l_1}{2} \right\rceil + 1$$

which completes the proof. □

4.3 Level-aligned Hierarchical Clustering

The high level idea for our hierarchical clustering for undirected graphs is rather easy: we renumber each vertex with a new bit representation $\langle b_r, \ldots, b_{q+1}, b_q, \ldots, b_1 \rangle$ that is interpreted as a combination of prefix $\langle b_r, \ldots, b_{q+1} \rangle$ and suffix $\langle b_q, \ldots, b_1 \rangle$. Different prefixes denote different clusters, and for a concrete prefix (cluster) its suffixes denote vertices within this cluster. Depending on the choice of q we get the whole spectrum between few larger clusters (q big) or many small clusters (q small). In particular we would like the following to hold:
For any $1 \leq \mu = 2^q \leq \sqrt{B}$, (1) there are $\lceil n/\mu \rceil$ clusters, (2) each cluster comprises μ vertices (one cluster may have less vertices), and (3) for any two vertices u and v belonging to the same cluster, their distance in G is $\mathcal{O}(\mu)$. In order to make this work the new vertex numbers have to be carefully chosen. Additionally, a look-up table is built that allows to find the sequence of disk blocks for adjacency lists of the vertices associated with a concrete cluster using $\mathcal{O}(1)$ I/Os.
In order to group close-by vertices into clusters (such that an appropriate renumbering can take place) we start with an arbitrary spanning tree T_s^0 rooted at source vertex s. Then we work in $p = \left\lceil \log_2 \sqrt{B} \right\rceil$ phases, each of which transforms the current tree T_s^j into a new tree T_s^{j+1} having $\lceil |T_s^j|/2 \rceil$ vertices. (The external-memory BFS algorithms considered here only use clusters up to a size of $\sqrt{B}$ vertices so this construction is stopped after p phases. The hierarchical clustering approach itself imposes no limitations and can be applied for up to $\lceil \log n \rceil$ phases for other applications.) The tree shrinking is done using time-forward processing [Arg03, CGG+95] from the leaves towards the root (for example by using negated BFS numbers for the time order of the vertices in T_s^j). Consider the leaves $v_1, \ldots, v_k$ with highest

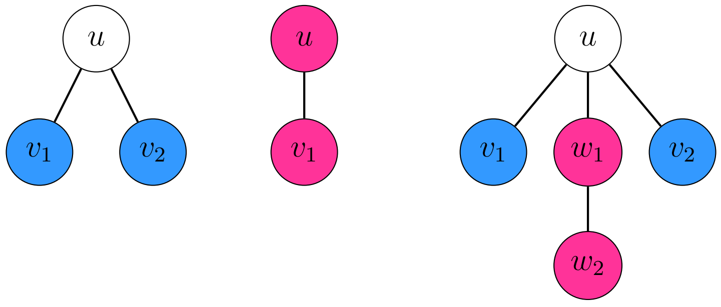


Figure 23: Three cases of merging vertices into pairs during the clustering. Note, that each vertex can represent a set of vertices that have been clustered in previous iterations. Case 1: merge two siblings. Case 2: merge a parent with its only child. Case 3: an example for a more complex merge with a combination of case one and two.

BFS numbers, what is equal to the smallest time indexes, and common parent vertex u in T_s^j. If k is even, then v_1 and v_2 form a cluster (and hence a vertex in T_s^{j+1}), v_3 and v_4 are combined, v_5 and v_6, etc. (*sibling-merge*, see Figure 23 on the left). In case k is odd, v_1 is combined with u (*parent-merge*, Figure 23 in the middle) and (if $k \geq 3$) v_2 with v_3, v_4 with v_5, etc. Merged vertices are removed from T_s^j and therefore each vertex is a leaf at the time it is reached by TFP, e.g. node w_1 shown in Figure 23 on the right was already merged with vertex w_2 and removed, so it is no longer present at the time v_1 and v_2 get processed. Thus, each vertex of T_s^{j+1} is created out of exactly two vertices from T_s^j, except for the root which may only consist of the root from T_s^j. Note that the original graph vertices kept in a cluster are not necessarily direct neighbors but they do have short paths connecting them in the original graph. The following lemma makes this more formal:

Lemma 4.6. *The vertices of T_s^j form clusters in the original graph having size* $\text{size}^{(j)} = 2^j$ *(excluding the root vertex which may be smaller), maximum depth* $\overline{\text{depth}}^{(j)} = 2^j - 1$*, and maximum diameter* $\overline{\text{diam}}^{(j)} = 2^{j+1} - 2$.

Proof. By induction (obvious for $\text{size}^{(j)}$). The clusters defined by T_s^0 consist of exactly one vertex each and satisfy $\overline{\text{diam}}^{(0)} = 0$ and $\overline{\text{depth}}^{(0)} = 0$. For $j > 0$, three ways of merging vertex v_1 have to be considered (with a sibling $(\{v_1, v_2\})$, the parent $(\{v_1, u\})$ or not merged at all $(\{v_1\})$), resulting in

$$\begin{aligned}
\overline{\text{depth}}^{(j)} &= \max \left\{ \begin{array}{l} \overline{\text{depth}}^{(j)}(\{v_1, v_2\}), \\ \overline{\text{depth}}^{(j)}(\{v_1, u\}), \\ \overline{\text{depth}}^{(j)}(\{v_1\}) \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} \max \left\{ \overline{\text{depth}}^{(j-1)}(v_1), \overline{\text{depth}}^{(j-1)}(v_2) \right\}, \\ \overline{\text{depth}}^{(j-1)}(v_1) + 1 + \overline{\text{depth}}^{(j-1)}(u), \\ \overline{\text{depth}}^{(j-1)}(v_1) \end{array} \right\} \\
&= 2 \cdot \overline{\text{depth}}^{(j-1)} + 1 = 2 \cdot (2^{j-1} - 1) + 1 = 2^j - 1
\end{aligned}$$

$$\begin{aligned}
\overline{\text{diam}}^{(j)} &= \max \left\{ \begin{array}{l} \overline{\text{diam}}^{(j)}(\{v_1, v_2\}), \\ \overline{\text{diam}}^{(j)}(\{v_1, u\}), \\ \overline{\text{diam}}^{(j)}(\{v_1\}) \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} \overline{\text{depth}}^{(j-1)}(v_1) + 1 + \overline{\text{diam}}^{(j-1)}(u) + 1 + \overline{\text{depth}}^{(j-1)}(v_2), \\ \overline{\text{depth}}^{(j-1)}(v_1) + 1 + \overline{\text{diam}}^{(j-1)}(u), \\ \overline{\text{diam}}^{(j-1)}(v_1) \end{array} \right\} \\
&= 2 \cdot \overline{\text{depth}}^{(j-1)} + \overline{\text{diam}}^{(j-1)} + 2 = 2 \cdot (2^{j-1} - 1) + \overline{\text{diam}}^{(j-1)} + 2 \\
&= 2^j + \overline{\text{diam}}^{(j-1)} = 2^j + 2^j - 2 = 2^{j+1} - 2
\end{aligned}$$

Note that while $\overline{\text{diam}}^{(j)}$ and $\overline{\text{depth}}^{(j)}$ denote the *maximum* diameter and depth possible for a cluster with 2^j vertices the actual values may be much smaller. $\square$

For each $1 \leq \mu = 2^q \leq \sqrt{B}$, the hierarchical approach produces a clustering with $\Theta(n/\mu)$ clusters containing $\Theta(\mu)$ vertices (excluding the root cluster) and a diameter of $\mathcal{O}(\mu)$.

Details on the construction of T_s^{j+1}. Two types of messages (*connect*(ID) and *merged*(ID)) are sent during the time-forward processing. When two vertices u and v are combined, the vertex u visited first sends the ID of the new vertex in T_s^{j+1} to v in a *merged*() message. The *connect*() messages are used to generate edges of T_s^{j+1} using the new IDs. The *merged*() messages are sorted, so that a vertex v (if any has been sent) can read it, before the *connect*() messages of that vertex, so checking whether the current minimum entry in the priority queue has received such a message can be done in $\mathcal{O}(1)$ I/Os.

Renumbering the vertices. The p phases of contracting the spanning tree each contribute one bit of the new vertex number $\langle b_r, \ldots, b_{p+1}, b_p, \ldots, b_1 \rangle$. The construction of T_s^{j+1} defines the bit b_{j+1} for the vertices from T_s^j to be 0 for the left and 1 for the right child in case of a sibling-merge, to be 0 for the parent and 1 for the child in case of a parent-merge, and to be 1 for the root vertex (unless it was already merged with another vertex). After p contraction phases T_s^p with $c = \lceil n/2^p \rceil$ vertices/clusters remains. The remaining $(r-p)$ bits are assigned by computing BFS numbers (starting with $(2^{\lceil \log_2 c \rceil} - c)$ at the root) for the vertices of T_s^p. These BFS numbers are inserted (in their binary representation) as the bits $\langle b_r, \ldots, b_{p+1} \rangle$ of the new labels for the vertices of T_s^p.
In order to efficiently combine the label bits from different phases and propagate them to the vertices of G again time-forward processing can be applied. The trees $T_s^p, \ldots, T_s^0$ are revisited in that order and vertices of T_s^j can be processed e.g. in BFS order and each vertex v of T_s^j combines the label bits received from T_s^{j+1} with the bit assigned during the construction phase and then sends messages with its partial label to the vertices in T_s^{j-1} it comprises. The resulting new vertex numbering of G covers the integers from $[n'-n, n')$ where $n' = 2^{\lceil \log_2 n \rceil}$. There is a single gap $[0, n'-n)$ (unless n is a power of two) that can be easily excluded from storage by applying appropriate offsets when allocating and accessing arrays. Thereafter the new labeling has to be propagated to adjacency lists of G and the adjacency lists have to be reordered using $\mathcal{O}(\text{sort}(n+m))$ I/Os.
We assume that the adjacency lists are stored as an adjacency array sorted by vertex numbers, i.e. two arrays, the first with vertex information, e.g. offsets into the edge information; the second with edge information, e.g. destination vertices. Hence, there is no need for an extra index structure to retrieve any cluster in $\mathcal{O}(1+\frac{x}{B})$ I/Os, where x is the number of edges in that particular cluster. This is possible because all vertices of a cluster are numbered contiguously (for all values of $1 \leq \mu = 2^q \leq \sqrt{B}$) and the numbers of the first and last vertex in a cluster can be computed directly from the cluster number (and cluster size μ).

Lemma 4.7. *For an undirected connected graph G with n vertices, m edges, and a given spanning tree T_s the Hierarchical Bottom-Up Clustering for all cluster sizes $1 \le \mu = 2^q \le n$ can be computed using $\mathcal{O}(sort(n))$ I/Os for constructing the new vertex labeling and $\mathcal{O}(sort(n+m))$ I/Os for the rearrangement of the adjacency lists of G.*

4.4 The I/O-complexity of dynamic BFS for the incremental algorithm

For sparse graphs, the static external-memory BFS algorithm MM_BFS has an I/O-complexity of $\mathcal{O}(\frac{n}{\sqrt{B}}+\text{sort}(n+m)+\text{MST}(n,m))$ I/Os (refer to Section 2.11). We aim to reduce the first term of $\mathcal{O}(\frac{n}{\sqrt{B}})$ I/Os.

The dynamic BFS algorithm consists of a preprocessing, where we compute a balanced clustering, which can be efficiently reorganized and a computation phase, in which we update the BFS tree to deal after an edge update. Meyer proved in 2008 [Mey08a] that for $\Theta(n)$ edge insertions on sparse graphs, even if we start with a list graph, the amortized expected number of I/Os is bounded by $\mathcal{O}(\frac{n}{B^{2/3}}+\text{sort}(n)\cdot\log(B))$ I/Os per update. This is deduced from the potential number of changes of the distances of the vertices to the source s of the BFS tree the over all updates. The proof of the I/O-complexity of dynamic BFS distinguishes two major cases for edge insertions, in order to count the I/Os over all updates.

Case I deals with edge insertions, where a single vertex v or a whole sub component C_v of the graph G, which includes v, is connected to the component C_s, which includes the source s of the BFS tree. The two components C_s and C_v are merged to the new component C'_s. While an added singleton is trivial (the level of v is the level of u plus one), the BFS-level of vertices in a larger sub component can be computed by traversing C_v with MR_BFS. With MR_BFS the complexity of a merge is bounded by $\mathcal{O}(|C_v|+\text{sort}(|C_v|))$ I/Os for computing the BFS-levels of C_v and $\mathcal{O}(\text{sort}(|C_v|)+\text{scan}(n))$ I/Os to update the data structure that contains the BFS tree. Obviously, this case can occur at most $n-1$ times until all vertices are part of the component that belongs to s. We summarize the I/O-complexity of this case in Lemma 4.8.

Lemma 4.8. *The graph $G = (V, E)$ contains at least two disjoint components. Let C_s be the connected component that contains the root of the BFS tree. Let C_v be the connected component that contains v and $C_v \neq C_s$, the I/O-complexity of an edge insertion $\{u, v\}$ ($u \in C_s$) is bounded by $\mathcal{O}(|C_v|+sort(|C_v|)+scan(n))$ I/Os for sparse graphs.*

Case II deals with edge insertions, where both vertices of the new edge are already member of the component C_s. We potentially update the BFS-levels of up to $\Omega(n)$ vertices as already demonstrated within the example depicted in Figure 22. The algorithm prefetches $\alpha = 32 \cdot 2^j$ BFS-levels into the static hotpool H, where α is a parameter of the algorithm with a size based on the j-th attempt. During computation, the the number of random I/Os is counted, that are needed to fetch clusters into the second hotpool HC. If too many random I/Os are encountered, we increase the value for α by increasing the attempt j and continue the dynamic BFS computation as long as the value for α is smaller than $B^{1/2}$. In that case we do not count the random I/Os anymore, because we ran in the case that the dynamic BFS turned into the static version (MM_BFS).
Note, if $\Delta d_i = d_{i-1}(v) - d_i(v) \leq \alpha$, we can prefetch α BFS-levels in the static hotpool H and compute the updated BFS tree within $\mathcal{O}(\alpha \cdot \frac{n}{B})$ I/Os. The timers to memorize the age of loaded data are set to $t_j = \Theta(2^j)$ for the dynamic hotpool HC, in order to avoid that some clusters stay in this hotpool during the update, if cluster information is loaded but not consumed. This sometimes happens if vertices are in a cluster and loaded by an unstructured access but close to the border of the cluster and the prefetching window of size α such that the information is prefetched and read from the static hotpool and therefore not read and eliminated from the dynamic hotpool.
We recapitulate the proof of Meyer [Mey08a] over n updates on the next few pages which leads to an amortized I/O-complexity of $\mathcal{O}(\frac{n}{B^{2/3}}+\text{sort}(n) \cdot \log(B))$ I/Os over n updates on sparse graphs.

First, we prove that if the dynamic BFS algorithm can complete an update within the j-th attempt, with $j \geq 1$, the I/O-complexity for this update is bounded by $\mathcal{O}(2^j \cdot n/B + \text{sort}(n) \cdot \log(B))$ I/Os. The initial value for α is $\alpha := 32 \cdot 2^j$. The clustering starts with chunks of size $\alpha/4 = 8 \cdot 2^j = \mu$. We allow at most $\Theta(n/\alpha)$ random cluster fetches during the successful j-th attempt. Thus, we only count the costs for the last attempt because the I/Os for previous attempts are bounded by the last attempt (refer to Lemma 4.10). Beforehand, we shortly prove that we have less than $\log(B)$ attempts (Lemma 4.9) to successfully finish dynamic BFS.

Lemma 4.9. *Dynamic BFS never needs more than $\mathcal{O}(\log(B))$ attempts to update the BFS tree.*

Proof. The parameter j is used to derive the size of the clusters in the j-th attempt. Starting with $j = 1$ in the first attempt, we double j each time, too many random I/Os are observed. Doubling j after each attempt means that after $\log(B)$ attempts the cluster size is in the range of B. However, we already stop the dynamic algorithm, as soon as we reach a cluster size larger than $\sqrt{B}$ and switch to MM_BFS. So, we need $O(\log(B))$ attempts for each update. □

Lemma 4.10. *Assume that the dynamic BFS computation on a sparse connected graph succeeds during the j-th attempt with $1 \leq j < \log(B)$. Then the algorithm performed $\mathcal{O}(2^j \cdot n/B + \text{sort}(n) \cdot \log(B))$ I/Os.*

Proof. For each attempt, we compute a new clustering within
$\mathcal{O}(\text{sort}(n) \cdot \log(B))$ I/Os (refer to Section 4.3). The recomputation of the clustering can be done within $\mathcal{O}(\text{sort}(n))$ I/Os. So, in total $\mathcal{O}(\text{sort}(n) \cdot \log(B) + j \cdot \text{sort}(n)) = \mathcal{O}(\text{sort}(n) \cdot \log(B))$ I/Os suffice for the preprocessing in the worst case, since we ensure that $j < \log(B)$.
The dynamic BFS computation consumes $\mathcal{O}(\text{sort}(n))$ I/Os for the static hotpool H. For the random loading of clusters we allow at most $2^k \cdot n/B$ I/Os in the k-th attempt. In total, we achieve an I/O-complexity of $\mathcal{O}(\sum_{k \leq j} 2^k \cdot n/B + \text{sort}(n)) = \mathcal{O}(2^j \cdot n/B + \text{sort}(n) \cdot \log(B))$ I/Os. □

For single updates it can happen, that MM_BFS *is* cheaper than the dynamic BFS. We need an argument that dynamic BFS is cheaper over $\Theta(n)$ updates.

To prove the expected I/O-complexity over $\Theta(n)$ updates, we first look at the less common scenario that we insert an edge which connects the component C_s to another component. The original paper [Mey08a] states that $\mathcal{O}(n \cdot \text{sort}(n) \cdot \log(B))$ I/Os are needed to compute up to n updates, in which a new component is added to C_s. This comes from the pessimistic recomputation of connected components in each update step. We argue that we can compute these updates with less I/Os, if we maintain a data structure that stores the information in which connected component each vertex is.

Lemma 4.11. *At most $n{-}1$ updates, called Type A updates, will add a subgraph to the component C_s until all vertices $v \in V$ have a BFS-level $d_{n-1}(v) < n$. These $n{-}1$ updates on sparse graphs cause $\mathcal{O}(n \cdot sort(n))$ I/Os in total.*

Proof. It is easy to see that at most $n{-}1$ updates of type A can occur. The graph has $|V| = n$ vertices and if we exclude our starting component C_s that contains at least s, we can have at most $n{-}1$ non-empty components of size at least one. Each update is of the following pattern: insert a new edge $\{u, v\}$, with $u \in C_s$ and $v \in C_{nc}$, where C_{nc} is a component that was not reachable from s prior to this update. The **first update** of type A needs a computation of the connected components of G. After each further update of type A we can update the connected components within an additional sorting step plus the parallel scanning of the added vertices in a temporary vector and the vector that stores the connected component information.
The computation of the BFS-levels of the new component can be done by MR_BFS in $\mathcal{O}(|C_{nc}|{+}\text{sort}(n))$ I/Os. Altogether we achieve $\mathcal{O}(n \cdot \text{sort}(n))$ I/Os for connected components and MR_BFS over the up to $\Theta(n)$ updates of type A. □

Now we analyze the I/O-complexity of the type B updates: an edge is inserted between two vertices that are member of the connected component C_s. We use our strategy to work with growing cluster sizes in each attempt.

Lemma 4.12. $\Theta(n)$ *updates on unweighted and undirected sparse graphs inside a connected component C_s (type B updates) require*
$\mathcal{O}(n \cdot (\frac{n}{B^{2/3}}{+}sort(n) \cdot \log(B)))$ *I/Os with the level-aligned hierarchical clustering.*

Proof. We add a new edge $\{u, v\} \notin E_{i-1}$ to E_i. We assume w.l.o.g. that $d(u) \leq d(v)$. Furthermore let $\Delta d_i(v) = d_{i-1}(v) - d_i(v)$. We have clusters of size $\mu = \alpha/4 = 8 \cdot 2^j$. If the insertion of the new edge causes at least one random I/O for a cluster fetch, while operating with advance value $\alpha_k = 32 \cdot 2^k$ in the k-th attempt, it holds that $\Delta d_i(v) > \alpha_k$. For all vertices v' in the cluster $C(v)$ containing v it holds that $\Delta d_i(v') = d_{i-1}(v') - d_i(v') =$
$\underbrace{d_{i-1}(v') - d_{i-1}(v)}_{\geq -\mu} + \underbrace{d_{i-1}(v) - d_i(v)}_{> \alpha_k} + \underbrace{d_i(v) - d_i(v')}_{\geq -\mu} > \alpha_k - 2\mu \geq \alpha_k/2.$
The last statement expresses that the distance of each vertex in the cluster $C(v)$ to s has been decreased by at least $\alpha_k/2$ through the update.

If we succeed in the j-th attempt, we have fetched at most $\mathcal{O}(\alpha_j \cdot n/B)$ times a cluster into the dynamic hotpool HC. So, we have fetched $\Theta(\alpha_{j-1} \cdot n/B = 2^{j+4} \cdot n/B)$ clusters in the previous attempt $j-1$ to the dynamic hotpool with $\Theta((2^{j+4} \cdot n/B) \cdot (\alpha/4)) = \Theta(2^{2 \cdot j+3} \cdot n/B) = \gamma$ vertices. For these γ vertices we know that their distances are decreased by at least $\Delta d_i(\cdot) \geq \alpha_{j-1}/2 = 2^{j+3}$.
With the decreased distances of these γ vertices, we reduced the potential over the whole graph of the case that we need at least j attempts during a later update again. To express this more formally, we summarize the potential of the distance reduction by D_i for the i-th update by $D_i = \sum_{v \in V_s} d_i(v)$ and $\Delta D_i = |D_{i-1} - D_i|$. For the i-th update we can now formalize that if this update succeeds within the j-th attempt, we have a change of the potential of distance decrements in the next update by $\Delta D_i \geq 2^{3 \cdot j+5} \cdot n/B := Y_j$ with a probability of at least $\frac{1}{2}$. We call such an event **a large j-yield** with yield-factor Y_j.

Let us consider two updates i' and i'' that both succeed after the same number of attempts j. The two updates are computed independently and both have a large yield-factor with probability of at least $\frac{1}{2}$. Concerning several independent updates, we can show, using Chernoff bounds [HR90], that out of $k \geq 16 \cdot c \cdot \ln(n)$ updates, with a constant $c > 0$, at least $k/4$ updates cause a large yield event with high probability, namely $1 - \frac{1}{n^c}$.
For a sequence of $z = \Theta(n)$ edge insertions it holds that $n^2 > D_0 \geq D_1 \geq \cdots \geq D_{k-1} \geq D_z > 0$. For a large j-yield event in the i-th update it holds that $\Delta D_i \geq Y_j$.

Thus, in the worst case there can be at most $\frac{n^2}{Y_j} = \frac{n^2}{2^{3 \cdot j+5} \cdot n/B} = \frac{n \cdot B}{2^{3 \cdot j+5}}$ of such j-yield events during such an update sequence. The reason is that we cannot reduce the distance of any vertex anymore because we already have build a clique and the distance of any pair of disjoint vertices is 1 and no smaller distance is possible on unweighted graphs.

In order to continue our previous results, we have to distinguish between two types of updates again. Updates, where the value for α never becomes larger than $\alpha'_j < B^{1/3}$ (Type B1) and thus need not more than j' attempts and those updates where $B^{1/3} \leq \alpha \leq B^{1/2}$ (Type B2). For updates of type B1 we know by Lemma 4.10 that at most $\mathcal{O}(2^{j'} \cdot n/B + \text{sort}(n) \cdot \log(B))$ I/Os are needed. With $2^{j'} = B^{1/3}$ we have an I/O-complexity of $\mathcal{O}(\frac{n}{B^{2/3}} + \text{sort}(n) \cdot \log(B))$ I/Os for each type B1 update. Each update in a sequence of $\Theta(n)$ updates can be of type B1. For the type B2 update with high probability there are at most $\mathcal{O}(n \cdot B/2^{3 \cdot j'})$ updates that succeed with at most j' attempts each. As previously discussed, at least $k/4$ updates produce a large j-yield with high probability. Hence, with high probability, we need at most $k_j = 4 \cdot n \cdot B/2^{3 \cdot j+5}$ updates that succeed within the j-th attempt in order to achieve the desired number of j-large yield updates.

Altogether with Boole's inequality the total amount of I/Os caused by the type B2 updates is bounded by $\mathcal{O}\left(\left(\sum_{g \geq 0} \frac{n \cdot B}{(B^{1/3} \cdot 2^g)^3} \cdot \frac{B^{1/3} \cdot 2^g \cdot n}{B}\right) + n \cdot \text{sort}(n) \cdot \log(B)\right) = \mathcal{O}(n \cdot (\frac{n}{B^{2/3}} + \text{sort}(n) \cdot \log(B)))$ I/Os with high probability.

□

The analysis, which was originally given by Meyer in 2008 [Mey08a], is independent of the structure of the graph. It does not matter, if the diameter is small, huge or the BFS-levels contain only $\mathcal{O}(1)$ vertices each. In addition, the original analysis assumed a randomized clustering.

4.5 Dynamic BFS with edge deletions

The original work on dynamic BFS states for the decremental version of dynamic BFS that the complexity over $\Theta(n)$ updates is the same as for the incremental version with the following arguments: if we delete an edge, we can only increase the distance reduction potential D_i. For the incremental version D_i can be only decreased. Therefore, the arguments of the proof in Section 4.4 slightly change

but the bound stays the same. The potential D_i grows over time but is bounded in total by $\mathcal{O}(n^2)$. We still have the same mechanisms for increasing values for α, counting random I/Os and the same two hotpools for static and dynamic data. In cooperation with Fabina Knöller we worked on a prototype of an implementation for the edge deletion on dynamic BFS in external memory [Knö16]. Together, we described the possible impacts of an edge deletion $\{u, v\} \in E_i$ during the i-th update. W.l.o.g. let $l(v) \geq l(u)$.
We distinguish the following cases for the deleted edge $\{u, v\}$:

- Case I: The vertices u and v belong to the same BFS-level. This case has no impact on the BFS tree.

- Case II: The vertex v has no other edge and is now an isolated vertex. The BFS-level of v is set to n to mark it unreachable.

- Case III: The vertex v has another edge into the previous level. This can be checked with $\mathcal{O}(1 + (x + y)/B)$ I/Os, where x is the length of the adjacency list of v and y the number of vertices in the previous level.

- Case IV: The vertices v and w are in the same BFS-level, and v has no remaining edge to the previous level. Thus, the BFS-level of v is increased by one. Then, the children of v in the BFS tree have to be checked if their level is increased by one, too. If we have a data structure that stores the adjacency lists in sorted order (by the BFS-levels), we can do this as follows: we start a local MR_BFS from v and only check for vertices below, if they have another connection to the upper level. We can scan the levels downwards with a small value for α and determine vertices that are only connected to s by a path going through v. The total I/O-complexity is sort(n) I/Os.

- Case V: The vertex v is not isolated but none of its edges connect v to another vertex in the previous or at least in the same BFS-level. We distinguish two sub cases: in case Va the deletion of the edge $\{u, v\}$ isolates a whole sub component of the graph. Case Vb deals with the search for the shortest replacement path that connects v to s now. The BFS-level of v is increased by at least 2. Sub case Va can be covered by computing the connected components with $\mathcal{O}(\text{sort}(n+m) \cdot \log(B)$ I/Os. Sub case Vb needs a total reordering of the BFS tree by dynamic BFS. We can start the dynamic BFS from the level $l(u)$.

The cases I to IV are more likely in settings like social networks or web graphs due to the small diameter and large connectivity between the vertices. Case V is more likely for small clusters in the outer regions or on road networks (e.g. when a road to the highway is closed for a while, a detour of several kilometers and sometimes through a few smaller cities is not uncommon).
The prototype implementation by Knöller showed in experiments that the deletion of edges is only a constant factor slower than the edge insertion in practice due to the overhead when we have to deal with heavy updates as described in case Vb. We tried to avoid some update overhead by excluding vertices and regions that will not be affected. Knöller's experiments showed that dynamic BFS is slower by a factor between 1.5 and 30 for edge deletion compared to edge insertion.

4.6 Configuration and test data

Our external-memory dynamic BFS implementation relies on the STXXL library in version 1.3.1 [DKS08, BDS09] and highly uses pipelining. For our static EM-BFS results we used the MM_BFS implementation from Ajwani et al. [AMO06] which also applies STXXL functions. We performed our experiments on a machine with an Intel dual core E6750 processor @ 2.66 GHz, 4 GB main memory (3.5 GB free), 3 hard disks with 500 GB each as external memory for STXXL, and a separate disk for the operating system, graph data, log files etc. The operating system was Debian GNU/Linux amd64 'wheezy' (testing) with kernel 3.2. We compiled with GCC 4.7.2 in C++11 mode using optimization level 3.
We use four different graphs to evaluate the performance of dynamic BFS. As for the diameter approximation the web graph sk-2005 with a diameter of 40, the two synthetic graphs $\sqrt{n}$-level and $\Theta(n)$-level with diameters of $\Theta(\sqrt{n})$ and $\Theta(n)$ respectively. Further details about these three graphs are given in Section 2.14. The fourth graph is an instance of a tree pattern $T = (k, L, \mu)$ that is designed to elicit poor performance on static MM_BFS: k parallel lists of the same length

$L = l \cdot \mu - 1$[16] are combined into a tree by attaching the lists to the same root vertex. Such a tree has $n = k \cdot L + 1$ vertices. Now, the deterministic Euler-Tour clustering will divide this graph in a way such that on each chain $\mathcal{O}(l)$ clusters of length μ are produced. The root is part of the first cluster and therefore an offset of two is encountered whenever the Euler-Tour enters a new chain. As a consequence, for the static BFS computation of the first level, k different clusters have to be loaded and kept in the hotpool. Based on this offset, for the computation of each following level $\Theta(k/\mu)$ clusters have to be loaded into the hotpool which results in the same amount of I/Os. This sums up to $\Theta(k/\mu \cdot L) = \Theta((n/\mu) = \Theta(n/\sqrt{B})$ I/Os. A schematic of the tree pattern and the offset of the clusters is depicted in Figure 24. The implementation of MM_BFS is tuned to recognize such cases and try to keep adjacency lists cached in the main memory for structures that seem to be list like. However, for reasonable large k this attempt fails due to the too small main memory and therefore MM_BFS cannot benefit from the caching heuristic as it does for a single or even a few parallel lists. The level-aligned hierarchical clustering benefits from two advantages that avoid this bad performance: there is no offset in the clusters and clusters in the same level in a tree are stored with consecutive cluster IDs. Therefore, k clusters are loaded at the same time within $\mathcal{O}((k \cdot \mu)/B)$ I/Os each μ levels. This results in a total I/O-complexity of $\mathcal{O}(\text{scan}(n))$ over the L levels to load the clusters. We summarize these observations in Lemma 4.13.

Lemma 4.13. *With the tree pattern $T = (k, L, \mu)$ we require $\Omega(n/\sqrt{B})$ I/Os for loading the clusters into the hotpool while executing MM_BFS with the deterministic Euler-Tour based clustering and $\mathcal{O}(scan(n))$ I/Os with the level-aligned hierarchical clustering.*

4.7 Experimental results of the first study in 2013

In some of our experiments we inserted new edges (v_1, v_2) into the graph, where we set $v_1 = s$ (the source of the BFS tree) and select the other vertex v_2 from BFS-levels $0.1 \cdot d$, $0.2 \cdot d$, ..., d where d denotes the height of the BFS tree. The experiments were executed independently. For each inserted edge the initial BFS

[16]Note that the cluster size $\mu = \sqrt{B}$ (see Section 2.11)

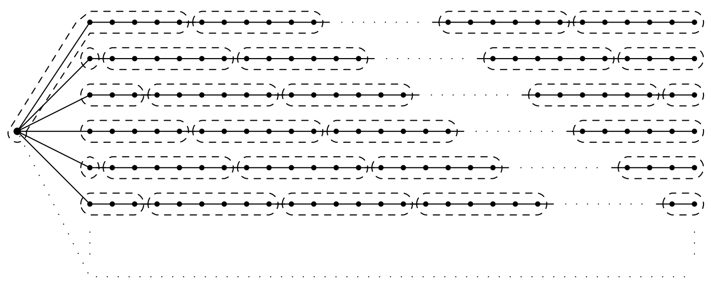


Figure 24: Our cl_n2_29 graph has the same shape as the graph in this picture but with 1048576 lists of length 511 each. In this picture the result of an Euler tour based clustering with $\mu = 6$ is shown as it is used in MM_BFS.

tree / graph was the same. We intentionally fixed one vertex of the inserted edge as the source vertex s in order to increase the number of vertices whose BFS-levels have to be updated. Two vertices far away from the source might have a small distance to each other and then usually only a small fraction of the tree has to be reassigned to new BFS-levels in data sets. We measured the time for dynamic BFS plus the time to write the result and the number of vertices whose BFS-levels have been updated. Experiments during the implementation showed that for a small initial cluster size α_2, e. g. $\alpha_2 = 64$, the value of α_2 is algorithmically never increased. This leads to a high number of random I/Os, because a too small fraction of each read block is loaded into the hotpool during a cluster fetch. Thus we set $\alpha_2 = 1024$ which causes only a small amount of random cluster fetches and yields in a better performance. For smaller α_2 our performance was slightly better for each graph but not for the $\sqrt{n}$-level graph. The number of elements in the static hotpool H is influenced by the value of α_1. We set the initial $\alpha_1 = 4$ in order to avoid too many sequential I/Os for repeatingly reading the same contents of the hotpool for each level computation. Table 6 contains the time for computing static EM-BFS for each graph divided into the preprocessing and the BFS computation. The cl_n2_29 graph stands out with a slow BFS computation while the preprocessing is almost as fast as the preprocessing for the other graph classes. Table 7 contains the time for the hierarchical clustering and the time that is needed to reorganize the adjacency lists (add cluster information to edges, sort them, ...). The hierarchical clustering is slower than the preprocessing of the static BFS because logarithmically many phases, each of which containing Euler Tour and list-ranking computations, have to be done instead of one.

	sk-2005	$\sqrt{n}$-level	$\Theta(n)$-level	cl_n2_29 graph
Preprocessing	0.91	1.35	1.19	1.29
BFS computation	2.41	3.26	1.36	$>$ 17

Table 6: Running time (in hours) of static EM-BFS with source 0.

	sk-2005	$\sqrt{n}$-level	$\Theta(n)$-level	cl_n2_29 graph
Compute hierarchical level-aligned clustering	0.39	1.64	1.35	3.01
Reorganization of adjacency lists	1.38	0.94	0.84	0.54

Table 7: Running time (in hours) of level-aligned hierarchical clustering.

Figure 25 depicts the performance of our dynamic BFS computing during the updates of the BFS tree. A huge amount of the vertices of the web graph sk-2005 are in a hub region and thus close to the source vertex of the BFS tree. Vertices with a larger distance to the source seem to have a list-like connecting path to the source which affects only a few vertices to be updated. In our experiments the worst performance is observed on the $\sqrt{n}$-level graph. Due to a huge amount of vertices in each of the $\sqrt{n}$ levels, we encounter many BFS-level updates for an inserted edge so that we are slower than static BFS in some cases.

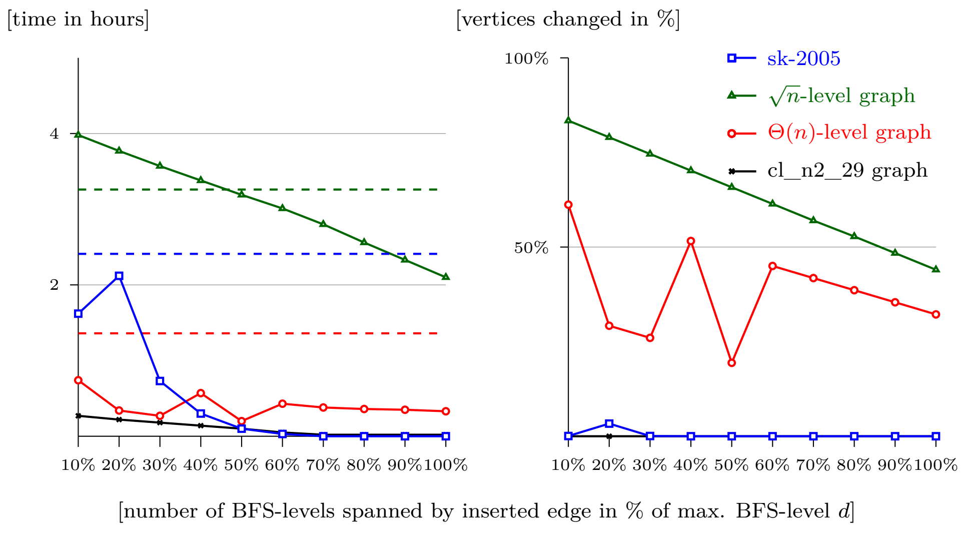


Figure 25: Results of dynamic BFS. The time of each static BFS is plotted in as a dashed line in the left plot for sk-2005 (2.41 h), $\sqrt{n}$-level graph (3.26 h) and $\Theta(n)$-level graph (1.36 h). The static BFS time of cl_n2_29 graph was not drawn because it is too huge with 17 hours to fit into the plot.

Results of our experiments with cl_n2_29 graph: as expected the hotpool of static MM_BFS with the Euler-tour based clustering had to go external and read/wrote Terabytes of data (input data set size: 8 GB) and needed more than 17 hours to compute the BFS tree. In contrast, each update during the dynamic BFS computation needed at most 0.23 hours.

Our results using hard disks were viable due to the choice of a comparatively large value for α_2. In experiments on a similar machine using solid state drives we were able to improve our results. We outperform the static BFS for each graph class in each test scenario by using a smaller value for $\alpha_2 = 256$. For our $\sqrt{n}$-level graph we were able to outperform static BFS by a factor of at least 1.14. This is explained by the fact that for smaller α_2 the CPU work is much smaller but the I/O-time increases.

4.8 Some detailed analysis of real-world data

Almost five years after the original experiments with our implementation of dynamic BFS the graph community designs more and more algorithms that depend on the properties of real-world data. So does a colleague of us – Manuel Penschuck. In 2017 he published a graph generator that is able to generate random hyperbolic graph data in linear time [Pen17]. Graphs generated by this generator have the feature that their graph properties are close to real-world data with controllable edge distribution. We generated a data set with the parameters $n = 28$, degree $= 10$ and $\gamma = 3.0$.

The largest connected component has 258.3 million vertices and 1.3 billion edges. The height of the BFS tree for source 0 is 80. This data set is denoted by *hyp_n_28_d_10_g_3*. We used the sk-2005 web graph to compare our synthetic graph to an actual real-world data set (recall that sk-2005 has about 50 million vertices and 1.8 billion edges; refer to Section 2.14 for further details about the web graph).

We ran experiments on a machine with Ubuntu 16.04.1, Kernel version 4.13.0-36-generic and compiled with gcc 5.4.1. The hardware of the machine was an Intel i5-4590 CPU @ 3.30GHz with 32 GB main memory and six SSDs of 512 GB each.

	SBFS_TC[s]	SBFS_TL[s]	DBFS_TC[s]
hyp_n_28_d_10_g_3	1979.81	1315.74	3360.03
sk-2005	1337.8	1200.41	3102.37

Table 8: The times to compute the Euler-Tour based clustering (SBFS_TC) and the BFS-levels (SBFS_TL) for source 0. Furthermore the time to computer the level-algined hierarchical clustering for dynamic BFS (DBFS_TC).

We computed an external-memory BFS followed by 100 dynamic BFS runs with random edge insertions. In Table 8 the times for the static BFS computation and the clustering for the static as for the dynamic BFS are given. The results for the 100 runs were grouped by the delta of the BFS-levels between the two vertices u and v of the inserted edge $\{u, v\}$. In the Tables 9 and 10 it can be seen that at least 33% of the total execution time can be saved compared to the static BFS.

Δ	# cases	avg. time[s]	min[s]	max[s]	avg. speed up to static BFS
0	23	0.01	0	0.02	99.99 %
1	29	0.01	0.01	0.01	99.99 %
2	20	453.15	115.83	819.07	65.56 %
3	12	510.15	22.71	1007.71	61.23 %
4	5	746.57	616.28	1132.8	43.26 %
5	3	263.23	66.3	602.71	79.99 %
6	2	92.56	62.35	122.76	92.97 %
7	1	105.9	105.9	105.9	91.95 %
9	1	105.2	105.2	105.2	92.00 %
10	1	57.48	57.48	57.48	95.63 %
13	1	65.73	65.73	65.73	95.00 %
16	1	46.12	46.12	46.12	96.49 %
25	1	23.94	23.94	23.94	98.18 %

Table 9: Dynamic BFS performance on hyp_n_28_d_10_g_3 over 100 experiments with random edge insertions.

Δ	# cases	avg. time[s]	min[s]	max[s]	avg. speed up to static BFS
0	25	0.01	0.01	0.02	99.99 %
1	33	0.01	0.01	0.02	99.99 %
2	26	612.64	50.4	1190.5	48.96 %
3	9	800.02	173.79	1110.29	33.35 %
4	5	805.92	10.45	1217.04	32.86 %
5	2	86.8	14.61	158.99	92.77 %

Table 10: Dynamic BFS performance on sk-2005 over 100 experiments with random edge insertions.

The results of these experiments demonstrate that dynamic BFS is viable for current application in real-world data with small diameter and huge hub regions. In the hub regions, we can store updates in a buffer to evaluate them later, due to their small impact. We use this idea for the dynamic part of our distance oracle (refer to Section 5.9).

4.9 Conclusion

We implemented dynamic BFS in external memory and researched in more detail a dynamic and level-aligned hierarchical clustering that is versatile for further purposes. Our experiments demonstrate that dynamic algorithms can be an feasible option in external-memory settings. We tuned the hotpool by splitting it into two separate structures for the structured and unstructured external-memory accesses. During our experiments, we achieved a viable speed-up of at least 33% on real-world data.

5 Distance Oracle on Real-World Data Sets

Preliminaries Some parts of this chapter have already been published at ALENEX 2015 [AMV15].

5.1 Introduction

When we have a look at the short history of the external-memory model and the derived graph algorithms, breakthroughs in the provided applications were focused on adapting algorithms we have known since the 1950s such as BFS, DFS or SSSP. It took many years for BFS to tune two implementations so that they cover all possible input graphs in a reasonable I/O-complexity. With our dynamic BFS we did the first step on a journey to a library of external-memory algorithms that provide results on queries or updates faster in order to be used in an online application. Unfortunately, the proposed solution is still too slow for most real-time applications.

The data sets of applications like social media or scientific research grow each day by several hundred gigabytes and more. Algorithms that run for weeks or even a few days to answer a single query are not viable in these settings. As mentioned in Chapter 1 the industry tries to solve this issue by using in-memory databases on machines with an extra-large main memory. In addition, companies as Facebook Inc. developed an improved version of the programming language PHP to make it ready for big data applications on servers with tools as filtering, sorting and parallel and distributed computation on huge data sets. Refer to Ottoni [Ott18] for some information about the underlying compiler architecture. To be competitive to the development that larger machines and clusters are used if the main memory becomes to small for an application, we demonstrate on a single desktop computer that with a design pattern based on the external-memory model, it is possible to implement real-time applications that can handle a updates and queries of a dynamic shape. Based on the work of Ajwani et al. on geometric distance oracles [AKSS14], we focus on a graph algorithm application with the following constraints:

- Huge (undirected, unweighted) graphs containing real-world patterns.

- Distance queries for any pair of vertices $(u, v) \in V$ with $u \neq v$.
- A query time in milliseconds on single queries and a query time in microseconds on batched queries is desired.
- Limited availability of advanced the hardware – for example a standard desktop computer.

In order to deal with these constraints, we need a data structure that can be initialized efficiently and needs only a small amount of memory for each data element. We call this data structure a distance oracle (refer to Section 2.6).

5.2 The idea behind our distance oracle

We target to answer distance queries on a real-world graph such as a social network or a web graph using a constant number of I/Os. On a single BFS tree for an arbitrary root vertex r, we can give an approximate answer for a distance query between u and v with two I/Os – one I/O to get $dist(u)$ and one I/O to get $dist(v)$. The approximated distance is then $dist(u)+dist(v)$. However, we might have overestimated the distance between u and v because we might have ignored shared subpaths on the paths of from u respectively v to the root r. If the two vertices have a lowest common ancestor $lca(u, v) \neq r$ (refer to Definition 5.1), we can improve our answer by the subtraction of $dist(lca(u, v))$ two times from our naive result and obtain the exact distance of u and v restricted to the BFS tree (refer to Lemma 5.2). We introduce the concept of lowest common ancestors in more detail in Section 5.3.

Definition 5.1. *The lowest common ancestor of two vertices u and v, with $u \neq v$, in a tree T, rooted by an arbitrary root r, is given by the first shared vertex $lca(u, v)$ on the paths from u respectively v to r. If u and v share no vertex but on their paths to the root, $lca(u, v) = r$. Another notable case is that u is a descendant of v, so that $lca(u, v) = u$*

Lemma 5.2. *The exact distance between two vertices u and v in a BFS tree T with root r is determined by $dist(r, u)+dist(r, v)-2 \cdot dist(r, lca(u, v))$ whereby the function $dist(r, x)$ returns the distance from the root r to a vertex $x \in V$ in T and*

$lca(u, v)$ returns the lowest common ancestor of two vertices u and v in the BFS tree.

If we can provide an implementation that gives an answer for the $lca(u, v)$ function with a constant number of I/Os, we achieve our target to give an answer in real time. However, the approximation quality using a single BFS tree might be arbitrarily bad - even if we use the lowest common ancestor to improve our computations. A worst-case example is, that for two vertices u and v with farthest distance to the root r the lowest common ancestor $lca(u, v) = r$ but there is a connecting edge between u and v, which is not part of the tree. The returned answer is $\mathcal{O}(n)$ but in fact, the distance is only 1.
Hence, in order to achieve a better approximation ratio on average, we compute a set of BFS trees from different root vertices that shall cover as many shortest paths in the graph as possible. Two new questions come up: how many trees should we use and how do we ensure that the different roots cover as many shortest paths as possible.
We use observations of previous work on geometric distance oracles by Ajwani et al. [AKSS14], to answer these questions. They dealt with the same issues and found out that there is a strong relationship between the degree $deg(v)$ of a vertex v and its betweenness centrality in real-world graphs as stated in Observation 5.3. The betweenness centrality $bc(v)$ of a vertex v is a measurement for the ratio of the number of shortest paths among all existing shortest paths in G that go through v. A simple example to explain the meaning of the betweenness centrality: on a list graph, an endpoint has no influence on any shortest path between any two other vertices in the list. On the other hand, every shortest path p_i starting at an arbitrary vertex in the left part of the list to an arbitrary vertex in the right part of the list goes through the center vertex c of this list graph. Therefore, c has a high betweenness centrality value in this list graph, while the endpoint has not. Thus, a vertex with a high betweenness centrality is suitable as a root of a BFS tree in a distance oracle because it covers many shortest paths. A typical distribution of the betweenness centrality and the vertex degree in real-world graphs is depicted in Figure 26.

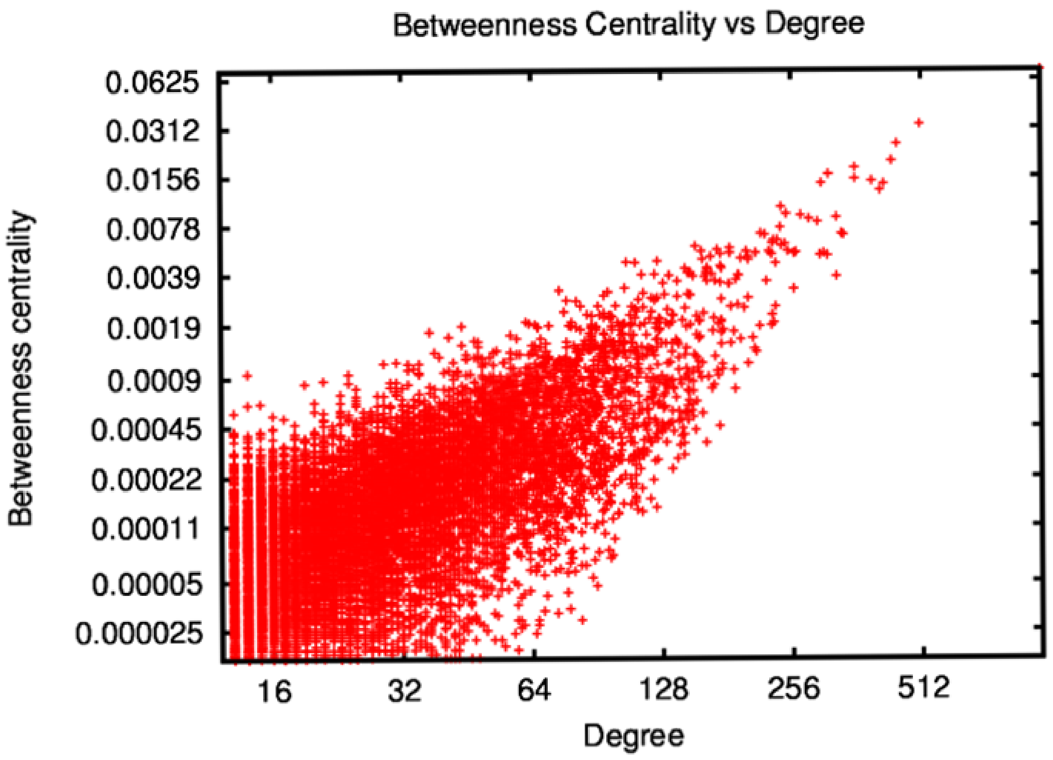


Figure 26: The relationship between the betweenness centrality and the degree of a vertex in a social network based on a part of the Santa Barbara Facebook graph [AKSS14].

Observation 5.3. *In real-world graphs, there is a strong relationship between the degree of a vertex and its betweenness centrality value. Usually, in social networks and web graphs there are a few vertices that have a high degree (hub nodes) and these vertices also have a high betweenness centrality value. This observation is supported by the work of Ajwani et al. [AKSS14] where they experimentally evaluated this relationship on various real-world data sets from science and industry and is known as power law distribution. The relationship between the structure of many real-world networks and the power law was shown by Price with an statistical approximation of the beta function [Pri76] and is used as power-law distribution for many graph applications [JJKP19, Blo17, VdS16].*

In experiments by Ajwani et al. it turned out that $\mathcal{O}(\log(n))$ BFS trees result into a good approximation ratio compared to the needed memory [AKSS14]. In this thesis, we adopt this number of BFS trees and evaluate the quality of the given answers based on a distance oracle with 20 BFS trees on various input graphs. We compute 20 BFS trees $T_1, \dots, T_{20}$ rooted at the twenty vertices with highest degree in G to empirically cover as many shortest paths as possible. In order to answer a distance query $dist(u, v)$ we compute the distance between u and v in each of these trees as described in Lemma 5.2 and return the minimum result over all trees.

5.3 I/O-efficient LCA computation

In order to answer real-time distance queries in time, we need an efficient data structure that can provide the result of the $lca(u, v)$ function using not more than $\mathcal{O}(1)$ I/Os. The height of the BFS tree is bounded by the diameter of the graph. Real-world data sets as social or web networks usually have a diameter asymptotically bounded by $\mathcal{O}(\log(n))$. A common ancestor of two vertices in a tree means that these two (or potentially even more) vertices share one or more nodes on their paths form the root r to these two vertices in a tree (refer to Definition 5.1). We can encode the path of a vertex v to the root by symbols (e. g. bit strings): compare the encoded paths of the two vertices and extract the common suffix on the encoded paths. The last common symbol of the suffixes of two vertices in their encoded paths is the last common ancestor on their paths. In the following we refer to the encoded path of each vertex as the **label** of vertex v. Previous work provided labels of size $\mathcal{O}(\log(n)$ [Fis09, AHL14] for fast LCA computation. However, these sophisticated approaches need a complex memory management. Labels might be of different size, a large set of symbols is needed and, depending on the labeling algorithm, we have to group vertices into clusters to ensure a shorter label size. This overhead to label the vertices is efficiently manageable for internal-memory algorithms, but in external memory the overhead provokes a larger amount of I/Os if the labels can be arbitraryly large[17]. Therefore, as in the work of Harel and Tarjan [HT84], we focus on binary labels. We bound the label size for our implementation by 64 bit. Our labels shall be compared by an exclusive-OR (XOR)

[17]We discussed a label size of $O(n)$ for general graphs during the implementation, but the STXXL external-memory dictionary of our approach has to be replaced by a data structure that maintains labels of different sizes in a hierarchical way.

operator in order to achieve a fast LCA computation. We transform our general BFS tree into a binary tree, but we have to keep the path- and distance information. We do this by adding dummy vertices to reorganize vertices with more than two children such that we have at most two children for each vertex in our binary tree representation. Dummy vertices have no influence on the distance computation nor can they be part of an input of a distance query – they are just virtual vertices. In the following we construct an algorithm to transform a BFS tree into a corresponding binary tree with a bounded height of $\mathcal{O}(\log(n)+h)$, where $n = |V|$ and h is the height of the BFS tree. For a graph with a diameter bounded by $\mathcal{O}(\log(n))$, the BFS tree height is $h = \mathcal{O}(\log(n))$ and therefore our labels are of size $\mathcal{O}(\log(n))$.

In order to transform a BFS tree T into a corresponding binary tree T' that represents the original tree in a binary structure, we implement a top-down approach from the root to the leaves. We add dummy vertices such that a vertex v with more than two children has only two children left. In order to restrict the height of the transformed tree, we compute the number of descendants using time-forward processing and keep vertices with more more descendants closer to their parent vertex. Refer to Lemma 5.4 for the pattern of the construction. The number of dummy vertices is bounded by $\mathcal{O}(n)$ as stated in Lemma 5.5.

Lemma 5.4. *For a vertex v with x children and $x \geq 3$, $x-2$ dummy vertices suffice to construct a binary representation of v and its children.*

Proof. We prove the lemma by induction.
Base cases: Let $x = 3$: we name the three children of vertex v, sorted decreasingly by the number of their descendants: c_1, c_2, and c_3. One dummy vertex d_1 suffices to rearrange the children into a binary representation of v and its children. The child c_1 has the largest amount of descendants in our sorted sequence, hence c_1 remains as a child of v and we add the dummy vertex d_1 as the second child of v. Now, the vertices c_2 and c_3 are assigned to d_1 as its children. No vertex has more than two children and we have a valid binary representation of v and its children. Let $x = 4$: we name the four children c_1, c_2, c_3, and c_4. Two dummy vertices d_1 and d_2 suffice to rearrange the children. There are two possible transformations: if the number of descendants of the child c_1 is greater than the sum of the numbers

of descendants of the other three children, c_1 remains a direct child of v and d_1 becomes the other direct child of v. Then the dummy d_1 has the dummy vertex d_2 assigned to it as a child and c_2 as the other. Finally, the remaining two children c_3 and c_4 are assigned to d_2. Otherwise, both dummy vertices become children of v and the four previous children c_1 to c_4 of v are assigned to the two dummy vertices.

Induction step with $x \to x{+}1$: We use the observation that it is never the case that a dummy vertex has only one child. Otherwise such a dummy vertex can be removed and replaced by its own child. For x children, that are sorted by the number of their descendants, we have already added $x{-}2$ dummy vertices. So we select the dummy vertex d_{x-2} and add a new dummy vertex d_{x-1} as a child of d_{x-2}. We move the original second child from d_{x-2} to d_{x-1} and add the new vertex $x{+}1$ to d_{x-1}, too. We now have $x{-}1$ dummy vertices for $x{+}1$ vertices. $\square$

If we adapt Lemma 5.4 from a single vertex to the whole tree, this results in Lemma 5.5.

Lemma 5.5. *In total, if we transform a BFS tree T with n vertices into its corresponding binary representation T', the number of added dummy vertices is bounded by $\mathcal{O}(n)$.*

We used the same argumentation as in the Huffman coding [Huf52, vL76] to construct and prove the above lemmas. Note, that the dummy vertices can be seen as virtual vertices only. The distance between a vertex v and its children is still counted as one, therefore we implicitly added edges with a weight $w_i = 0$ to the binary tree representation. The edges of the BFS tree have weight $w_i = 1$. In Lemma 5.6, which has been already published in 2015 [AMV15], we proved in that the distances between two vertices in the transformed tree remains the same as in the original BFS tree.

Lemma 5.6. *For any two vertices u, v in our BFS tree T, it holds that $d_T(u,v) = d_{T'}(u,v)$. Thus, we can use the binary representation T' of T to construct a distance oracle.*

Proof. The lemma follows by a simple induction on the number of nodes in T with degree greater than 2 which we have to transform. In the case there are no such nodes, T' is the same as T (with all edges having length 1) and therefore $d_T(u,v) = d_{T'}(u,v)$. Now let us assume the statement to be true for the tree T'_{k-1} after $k-1$ high degree node transformations (induction hypothesis), and we prove it for T'_k formed after k transformations. Consider a node pair u, w, let $P'_{k-1}(u,w)$ be the path between u and w in T'_{k-1} of length $d_{T'_{k-1}}(u,w)$, and let v be the next node to be transformed having ℓ children $v_1, \ldots, v_\ell$. By induction hypothesis we have $d_T(u,w) = d_{T'_{k-1}}(u,w)$ and we want to show that $d_T(u,w) = d_{T'_k}(u,w)$. There are three cases:

1. $P'_{k-1}(u,w)$ does not pass through $v, v_1, \ldots, v_\ell$. In this case, the transformation of node v does not affect this path and $P'_k(u,w) = P'_{k-1}(u,w)$ and therefore, $d_{T'_k}(u,w) = d_{T'_{k-1}}(u,w) = d_T(u,w)$.

2. $P'_{k-1}(u,w)$ contains exactly one edge from the edges $\{v, v_1\}$, $\{v, v_2\}$, $\ldots$, $\{v, v_\ell\}$. Let us call this edge $\{v, v_i\}$ and let us assume w.l.o.g. that there is a path $P'_{k-1}(u,v)$ between u and v as well as a path $P'_{k-1}(v_i,w)$ between v_i and w without including this edge. After the transformation there is a path between v and v_i of distance 1. Thus, there is a path $P'_k(u,w)$ in T'_k consisting of $P'_{k-1}(u,v)$ followed by the path between v and v_i and then $P'_{k-1}(v_i,w)$. The total length of this path is $d_{T'_{k-1}}(u,v)+1+d_{T'_{k-1}}(v_i,w) = d_{T'_{k-1}}(u,w)$. Since in a tree there is only one simple path between any two nodes ($P'_k(u,w)$ of length $d_{T'_{k-1}}(u,w)$), it holds that $d_{T'_k}(u,w) = d_{T'_{k-1}}(u,w) = d_T(u,w)$.

3. $P'_{k-1}(u,w)$ consists of two edges from $\{v, v_1\}$, $\{v, v_2\}$, $\ldots$, $\{v, v_\ell\}$. Let us call these edges $\{v, v_i\}$ and $\{v, v_j\}$ and let us assume w.l.o.g. that there is a path $P'_{k-1}(u,v_i)$ between u and v_i as well as a path $P'_{k-1}(v_j,w)$ between v_j and w without including v. After the transformation there is a path between v_i and v_j of distance 2 through their least common ancestor (which may not be v) in T'_k. Note that this is because the edges $\{v_l^1, v_i\}$ and $\{v_m^1, v_j\}$ introduced in the transformation have both length 1. Thus, there is a path $P'_k(u,w)$ in T'_k consisting of $P'_{k-1}(u,v_i)$ followed by the path between v_i and v_j and then $P'_{k-1}(v_j,w)$. The total length of this path is $d_{T'_{k-1}}(u,v_i)+2+d_{T'_{k-1}}(v_j,w)$ $= d_{T'_{k-1}}(u,w)$. Again, since in a tree there is only one simple path between any two nodes ($P'_k(u,w)$ of length $d_{T'_{k-1}}(u,w)$), it holds that $d_{T'_k}(u,w) = d_{T'_{k-1}}(u,w) = d_T(u,w)$.

□

The dictionary for the LCA detection stores dummy vertices with the BFS-level of v. With the knowledge that the distances between the vertices are the same in the BFS tree T as in its corresponding binary representation T', we can construct labels that can be used to determine the LCA of two vertices fast, based on the path information in T'. The construction of the LCA labels follows the idea of the in-order traversal [Mor79]. In a binary tree the in-order traversal visits the left child first, continues with the root and visits the right child last (left root right). This traversal is done recursively for each child.
While the labels are constructed, we use data words of fixed size. We start with the *root* and assign the label 2^{63} in binary representation. If we go to the left from a vertex u to a vertex v, the label l_v is created out of the label l_u of the vertex u by shifting the last 1 in the binary representation one position to the right. A simple example: $l_u = 1010$ and thus $l_v = 1001$.
If we go to the right from u to a vertex w while we traverse T', we add a 1 after the the last occurrence of a 1 in the binary representation of the label l_u to construct the label l_w for w. A simple example: $l_u = 1010$ and thus $l_w = 1011$. Figure 27 gives an example for a small tree.

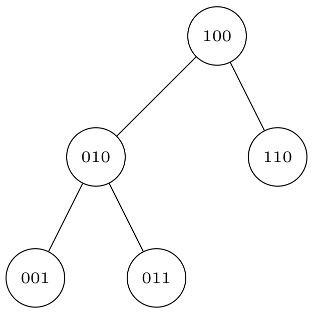


Figure 27: An example of a small binary tree that is encoded with our LCA label scheme.

In a general binary tree with height h', we need h' bits for each label to encode an in-order matchable numbering pattern. From the transformation T to T' we know that $h' \geq h$. Now, we prove that $h' \leq \log(n)+h$ so that the size of our vertex labels for LCA computation can be bounded by $\mathcal{O}(\log(n)+h)$. We prove that any vertex u with distance $dist(r,u)$ in T with $dist(r,u) \leq h$ has a distance of at most $dist(r,u) + \log(n)$ in T'. The following lemma was published in our ALENEX 2015 publication [AMV15] in slightly different structure and notation with the following remarks:

"We can encode distances in T by encoding distances in T'. The distances in tree T' are encoded using two hash-maps: IN that maps a node $u \in V'$ to its in-order numbering in T' and DIST that maps an in-order numbering to the distance of the corresponding node from the root in T'. In practice, we can reduce the space further by keeping the IN hash-map only for nodes in V (rather than V').

The inorder numbering requires h' bits, where h' is the height of T'. Next, we show that $h' \leq \log n + h$, thereby proving that the in-order number of a node can be stored in $O(\log n + h)$ bits. Consider a node u with degree greater than 2. Let the weight of its children $u_1, u_2, \ldots, u_k$ be $w_1, w_2, \ldots, w_k$. It follows from the optimality of Huffman coding that the node u_i is placed at most $\lceil \log \frac{\sum_j w_j}{w_i} \rceil$ hops away from the node u. Since in our case, the weight w of node u is greater than $\sum_j w_j$, it follows that each child u_i is at most $\lceil \log \frac{w}{w_i} \rceil$ hops away from u".

Lemma 5.7. *A BFS tree T with root r can be transformed into a binary tree T' with the same root r such that the height of T' is bounded by $\mathcal{O}(\log(n) + h)$, whereby $n = |V|$ and h is the diameter of the original graph G. In other words: a vertex u at distance $l(u)$ from the root in T is at most $l(u) + \log n$ hops away from the root in T'.*

Proof. Let $u = u_k, u_{k-1}, u_{k-2}, \ldots, u_1 = r$ be the path p from an arbitrary vertex u to the root r in T. Furthermore let $w_k, w_{k-1}, \ldots, w_1$ be the weights of the corresponding vertices $u_k, u_{k-1}, \ldots, u_1$ on the same path p respectively. An edge $\{u_i, u_{i+1}\}$ in this path p in T gets replaced by a path p' of at most $\lceil \log \frac{w_i}{w_{i+1}} \rceil$ hops inserted by the dummy vertices added to T'. The bound for the number of hops follows from the optimality of Huffman coding [Huf52, vL76]. The vertex u_i is placed at most $\lceil \log \frac{\sum_j w_j}{w_i} \rceil$ hops away from the node u. Since in our case, the weight w of vertex u is greater than $\sum_j w_j$, it follows that each child u_i is at most $\lceil \log \frac{w}{w_i} \rceil$ hops away from u.
Thus, the number of hops in the corresponding path p in T' is at most $\sum_{i=1}^{k-1} \lceil \log \frac{w_i}{w_{i+1}} \rceil \leq dist(u,r) + \sum_{i=1}^{k-1} \log \frac{w_i}{w_{i+1}} = dist(u,r) + \log \frac{w_1}{w_k}$. Clearly, $\frac{w_1}{w_k} \leq n$ as the number of

descendant at any vertex is at least 1 (as it includes itself) and at most n. Thus, the vertex u is at most $dist(u, r) + \log(n)$ hops away from the root r in T'.
Since we know that $dist(u, r)$ is bounded by h for any vertex $u \in V$, and we know for each of these vertices that their distance to the root is at most $dist(u, r) + \log(n)$ in T', the height of T' is bounded by at most $h + \log(n)$. $\square$

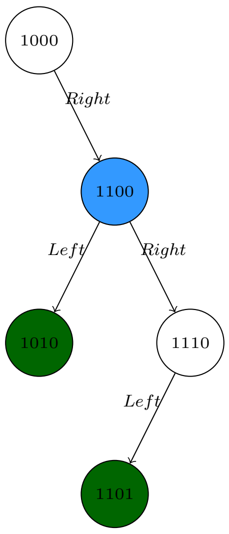


Figure 28: The distance between the two green marked vertices is requested. The answer is constructed by the sum of their BFS-levels minus twice the distance of the LCA to the root. The LCA (1100) is determined by the XOR operation on the labels to find the first position, in which the labels of the vertices 1010 and 1101 differ. All bits until this position are copied, the position, in which the labels differ is 1 and all other bits to the right are 0. This results in the distance $2{+}3 - 2 \cdot 1 = 3$.

The label $LCA(u, v)$ of the LCA of two vertices u and v is computed as follows: all labels have the same length. So, the new label is initialized as the label of the root (a one followed only by zeros). The labels of u and v are read in parallel from the left to the right. As long as the bits are equal, the value at the current position is written into the same position of the label to be constructed. We stop at the first position, where the labels differ. This position is kept as 1 an all following bits are zero. Note, that if one vertex is part of the left sub-tree and the other is one of the

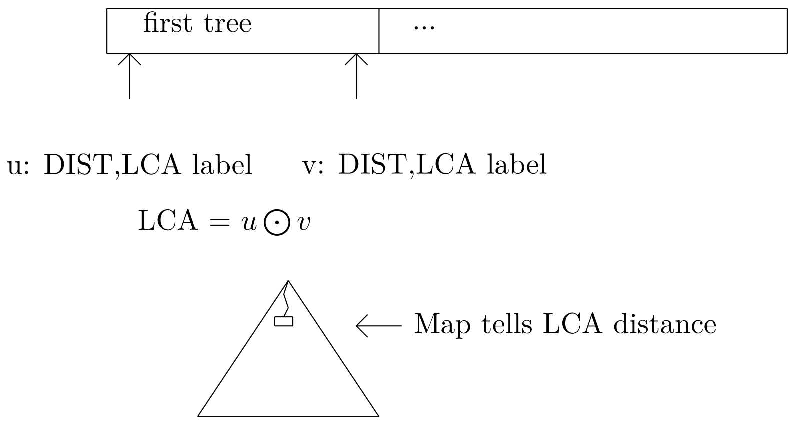


Figure 29: For two vertices u and v we get their BFS-Levels and their LCA labels from an array We combine their labels by bit operations to get the label of the LCA. Then we request the distance of the LCA from an external-memory map.

right sub-tree, e.g. the label of u is 0111 and the label of v is 1010, the label of the root 1000 is the result.
In Figure 28 an example is depicted, where u and v have the labels 1010 respectively 1101. The two labels are equal in the first position. Therefore the label $LCA(u, v) = 1100$. Note, that the LCA can be a dummy vertex. To return the BFS-level of any vertex, including dummy vertices, we have to store the BFS-levels into a dictionary, where the binary label of each vertex is the key and the BFS-level is the value. The dummy vertices store the BFS-level of the non-dummy above.

So far, we have described how we efficiently encode the BFS tree in an equivalent binary tree representation with bounded height and label size. In our implementation we create an array that stores the distance and the corresponding LCA label for each vertex $v \in V$ for each BFS tree $T_1, \ldots, T_{20}$. So, using a constant number of I/Os, we can extract all information about two arbitrary vertices u and v: their BFS-levels and binary labels for LCA computation in all trees. Then we compute the related set of LCA labels without any I/O and extract the BFS-level of each

LCA in the associated tree by a look up into our **external-memory dictionary**, which is provided by the STXXL, using $O(\log(n))$ I/Os per request. Therefore, in total we achieve an I/O-complexity of $O(\log(n))$ I/Os for a single query.
We achieve a query time of $O(\log(n)/B)$ I/Os for batched queries, if we collect $\Theta(n)$ queries and compute their results in a batched way by several scanning and sorting steps. Observe that we have computed all LCA labels afterward. In order to avoid random accesses to the STXXL dictionary implementation, we sort the labels and scan the dictionary array. Figure 29 depicts the query accesses to the data structures for one BFS tree.

5.4 Data sets

In order to evaluate the quality of our distance oracle we used a set of smaller graphs that represent different real-world network instances such as web graphs, p2p network data, a snapshot from the astro physics collaboration network and the DBLP co-author network. These graphs can be found in the SNAP database[18]. These graphs (ca-AstroPh, dblp, Facebook NY, hyperGrid, p2p-Gnutella31, Facebook Santa Barbara and web-BerkStan) have a few thousand up to a million vertices and diameters between 5 and 23. Details about these graphs can be found in the previous work on geometric distance oracles by Ajwani et al. [AKSS14]. For these data sets we solved APSP and ran our distance oracle for all possible queries. We compared the results to obtain a baseline of the quality we are able to achieve on real-world data sets in realistic use cases. Furthermore, we used the LOEWE-CSC super computer to compute a baseline on one of our larger web graphs, namely sk-2005, which is definitely too large to solve APSP for all possible queries in a realistic time setting (refer to Section 2.14 for more details about this graph). We use two more graphs in our external-memory implementation: com-friendster and twitter-2009. The com-friendster network has around 65 million vertices and 1.8 billion edges. The diameter is 32, the graph is a snapshot of the old Friendster network and it is listed in the SNAP data base. The twitter-2009 graph is an anonymized crawl of followers in August 2009, has about 52.2 million vertices, 1.6 billion edges and a diameter of 18 [GGL+13]. To demonstrate that our approach

[18] `https://snap.stanford.edu`

can deal with huge data sets we additionally generated a graph similar to our $\Theta(\sqrt{n})$-diameter graphs (refer to section 2.14) with 2.1 billion vertices, 8.4 billion edges and a diameter of 30. We refer to it by graph_2b_8b_d30 later this chapter.

5.5 Configuration of our machines

We implemented our external-memory distance oracle with C++11 and the external-memory library STXXL in version 1.3.1 [DKS08]. We performed our experiments on a machine A with an AMD FX(tm)-4170 Quad-Core Processor processor @ 4.2 GHz, 16 GB main memory (15 GB free), 4 hard disks with 1 TB each as external memory for STXXL, and a separate disk for the operating system, graph data, log files etc. Machine B has a AMD A10-6800K APU with Radeon(tm) @ 4.1 GHz, 32 GB main memory (31 GB free), 6 solid state drives with 512 GB each as external memory for STXXL, and a separate disk for the other data. The operating system on A and B was Debian GNU/Linux amd64 with kernel 3.14-2. We compiled on A and B with GCC 4.9.1 in C++11 mode using optimization level 3.
We also used 40 nodes of the LOEWE-CSC (http://csc.uni-frankfurt.de) with 128 GB main memory per node for the evaluation of the accuracy with a Linux Red Hat 2.6.32 kernel and a GCC 4.4.5 in C++0x mode using optimization level 3.

5.6 Experimental evaluation of the approximation quality

To experimentally evaluate the approximation quality of our distance oracle we solve exact APSP on our set of small real-world data sets, we described in Section 5.4, and compare the distance for every vertex pair to the result an internal-memory version of our distance oracle would provide. It turned out that on the set of real-world graphs we achieved a high accuracy rate, covering more than 80% of the queries with an error of at most 1 (refer to Figures 30 and 31).
For the DBLP graph we achieved only in 41% of the queries a matching distance result. A larger amount of queries, namely 48%, have an error of 1 and 11% of the queries have an error larger than 1.
Figure 32 depicts the results of the computation of 60,000 BFS trees in main memory from distinct random sources on the web graph sk-2005, which is equal to 1.2% of the vertices. We compared the results to our internal-memory distance

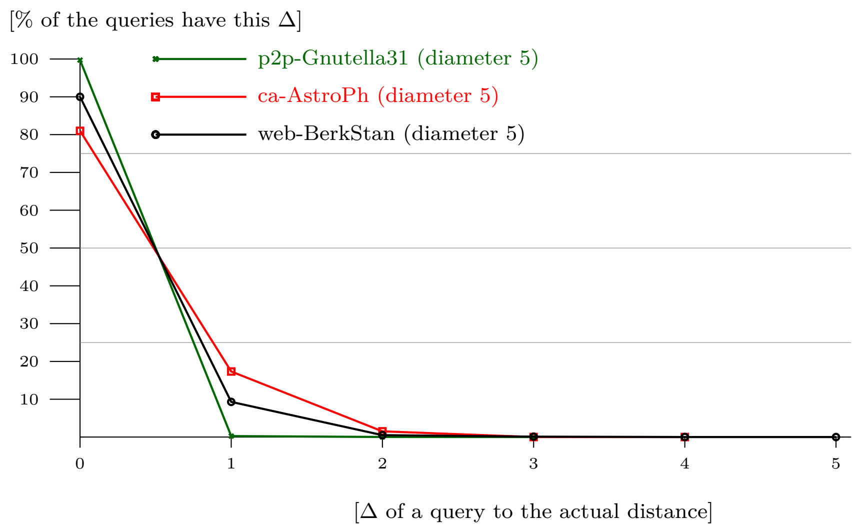


Figure 30: Results of our distance oracle with 20 BFS trees compared to the APSP for three graphs with a small diameter.

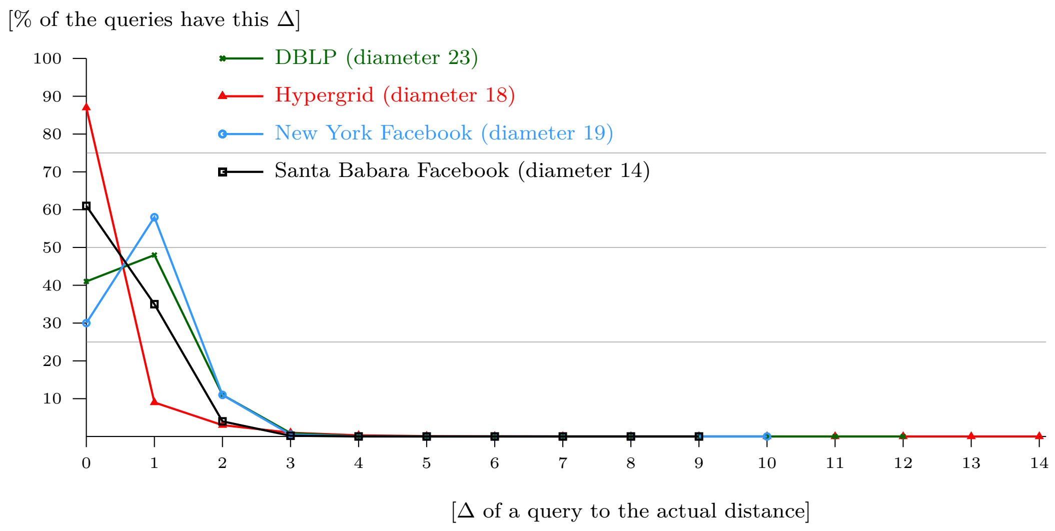


Figure 31: Results of our distance oracle with 20 BFS trees compared to the APSP for four graphs with a medium diameter.

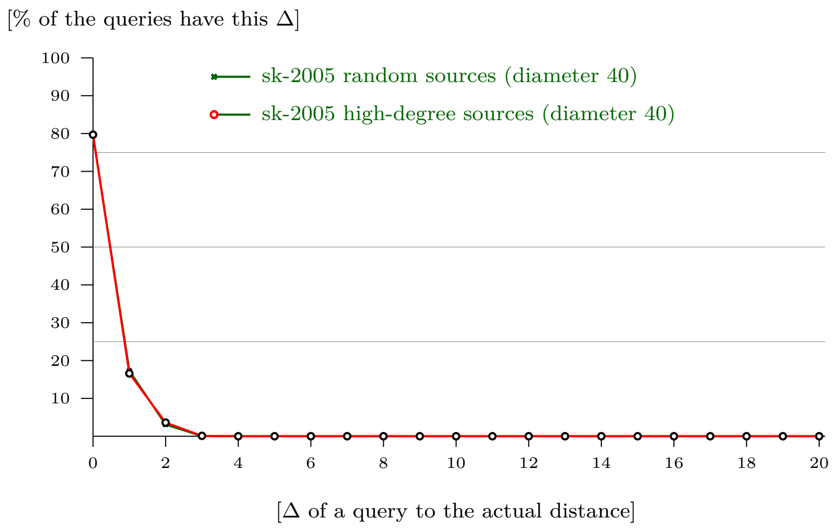


Figure 32: Detailed results of the web graph sk-2005 for 60,000 BFS computations from random sources. We observed an accuracy of almost 80% with both, random and high degree vertices as sources for the BFS trees of our distance oracle.

oracle simulation We needed almost 100,000 compute hours on our university's cluster computer Loewe-CSC[19]. Even though we could not test all n^2 distance pairs, the trendline shows that the answers of the distance oracle seem to provide the same quality as for the smaller graphs.

5.7 Preprocessing time

The preprocessing of our distance oracle involves the computation of the 20 BFS trees, the computation of the LCA labels for each BFS tree and the setup of the data structures like the external-memory map for each BFS tree. For the sk-2005 web graph the total time for the preprocessing of our distance oracle (with the semi-external MR_BFS variant) is 2.71 hours with 20 BFS trees on machine A. It took less than five minutes to compute one BFS tree with a tuned MR_BFS implementation with a switch to go semi-external and less than four minutes to encode such a tree using time forward processing. With a fully external MR_BFS

[19]https://csc.uni-frankfurt.de/wiki/doku.php?id=public:start

it takes around 20 minutes per tree. The resulting data is stored on disk and can be easily reused. Analyzing the preprocessing time for sk-2005 in more detail, we found that it takes around 6.8 hours with 50 BFS trees and around 14.1 hours with 100-BFS tress with the semi-external variant on machine A.
On the twitter-2009 graph the preprocessing time was almost the same as for sk-2005 with 2.69 hours in total with 20 BFS trees on machine A. On the graph com-Friendster, it tool 3.8 hours to compute the preprocessing with 20 BFS trees. On our synthetic graph graph_2b_8b_d30 the computation of one BFS tree took 2.1 hours on machine A and 1.4 hours on machine B. The computation of the labels took 2.1 hours on machine A and 0.8 hours on machine B.
The size of the preprocessing result of sk-2005, stored in a file, is between 8.7 GB for 5 BFS trees and 173.6 GB for 100 trees. For 20 BFS trees the required space on disk is 34.7 GB. For graph_2b_8b_d30 the file size for one BFS tree is about 56 GB. A file for 20 BFS trees would have a size of 1.1 TB and it would need about 3.5 days on machine A and 1.9 days on machine B to compute it.

5.8 Online queries on SSDs and batched queries on HDDs

Online distance queries are a common application of distance oracles, especially in the web. A user might want to know how close an item is related to another one in a network. Therefore, we evaluated randomized online distance queries on the web graph sk-2005 on machine A and B. We need a constant number of I/Os per tree for one query. With four HDDs in parallel on machine A, the query time is about a second for 20 BFS trees on sk-2005. The result on SSDs is similar if the same block size of 512 KB per disk is set. However, we found that by decreasing the block size on SSDs to 8-16 KB, we can achieve a query time of a few milliseconds for the same scenario. For 16 KB we get an average query time of 8.1 ms and for 8 KB, we even achieved a query time of 6.7 ms. An access to a single memory cell on a flash memory is quite fast but reading too many cells in a row wastes time to read a big block without any value for this scenario. On the other hand, the block size should not be too small so that the overhead to maintain the STXXL data structures and fetching blocks from the SSD do not slow down the performance for batched queries.

Batched distance queries work for applications without real-time requirements such as those in some learning systems which are used in many applications today in order to adapt interfaces to their users. By batching $\Theta(n)$ queries, we achieved a query time in the range of 15 to 26 microseconds per query for sk-2005, depending on the architecture. For various batch sizes on the web graph sk-2005, we refer the readers to Table 11.

For the twitter-2009 graph, we observed that the query time only differs by a small factor as compared to sk-2005.

Number of queries	Avg time $[A]$	Avg time $[B]$
~2^{27}	26,078 μs	16,274 μs
~2^{28}	24.562 μs	15,571 μs
~2^{29}	23.926 μs	15,626 μs
~2^{30}	23.616 μs	15,759 μs
~2^{31}	23.709 μs	16,144 μs

Table 11: Average query time for a query on machine A with HDDs and on machine B with SSDs for different natch sizes on sk-2005.

5.9 Dynamic setting

Our implementation supports edges updates in the following pattern: when a new edge (u, v) is requested to be added to the graph, we determine the distance $dist(u, v)$ between u and v in the graph without this edge. If $dist(u, v)$ is smaller or equal to a threshold, which can be defined by the user, we insert the new edge into a buffer and our distance oracle contains outdated data. Edge insertions with a larger distance $dist(u, v)$ directly force to a recomputation of our distance oracle. While the BFS trees can be updated as in previous dynamic BFS implementations, the LCA encoding is recomputed from scratch. If the buffer is full we recompute the distance oracle from scratch for all updates. A chart of the program flow is provided in Figure 33.

We found that the average update time for an edge insertion depends on the threshold that determines whether an edge is incorporated immediately into the recomputation of BFS trees or whether it is batched for a future update, and also on the size of the buffer that we keep for the update. These two parameters allow us to control the update time vs. accuracy trade-off. For the extreme scenario where we batch a million or more edges, we can easily get the amortized update time down to milliseconds per edge. Our preliminary experiments suggested that for random edge insertions the resulting accuracy loss, is still a small additive factor, particularly when the initial graph already has a large number of edges. On the other hand, updating a BFS tree with our dynamic BFS implementation for a single edge can take many minutes. Similarly, updating the LCA encoding of a tree from scratch takes a few minutes on our real-world graphs. Thus, in order to keep our average update time small, which is a requirement for some applications, we choose the threshold fairly high (relative to the graph diameter) and the buffers fairly large. Note that most vertices in real-world graphs lie in a tightly connected core (hub region) and thus, most vertex pairs have a very small distance between them as compared to the graph diameter. So, many edge insertions do not require an immediate update.

5.10 Conclusion and open problems

We demonstrated that we are able to implement a real-time data structure for external-memory graph algorithms with our distance oracle. The preprocessing of our implementation needs only a few hours on our data sets. We have chosen data sets of typical size. The presented data sets have been used in several publications over the past few years. Even larger data sets like our synthetic data set graph_2b_8b_d30, which is more complex than the typical real-world data sets, is processed within a few days. We store less than a kilobyte per vertex for 20 BFS trees in our oracle. On SSDs, we can answer online queries in milliseconds due to our efficient data structures and constant I/O-complexity. On both architectures, SSDs and HDDs we can answer batched queries in microseconds per query for a reasonable large number of queries. In addition, our distance oracle can be updated I/O-efficiently for edge insertions by either buffering updates with less impact or processing new edges immediately if their impact is too high.

Observe that the idea of small LCA labels, designed for a specific use case, is used in various algorithms nowadays to achieve fast answers for queries in several

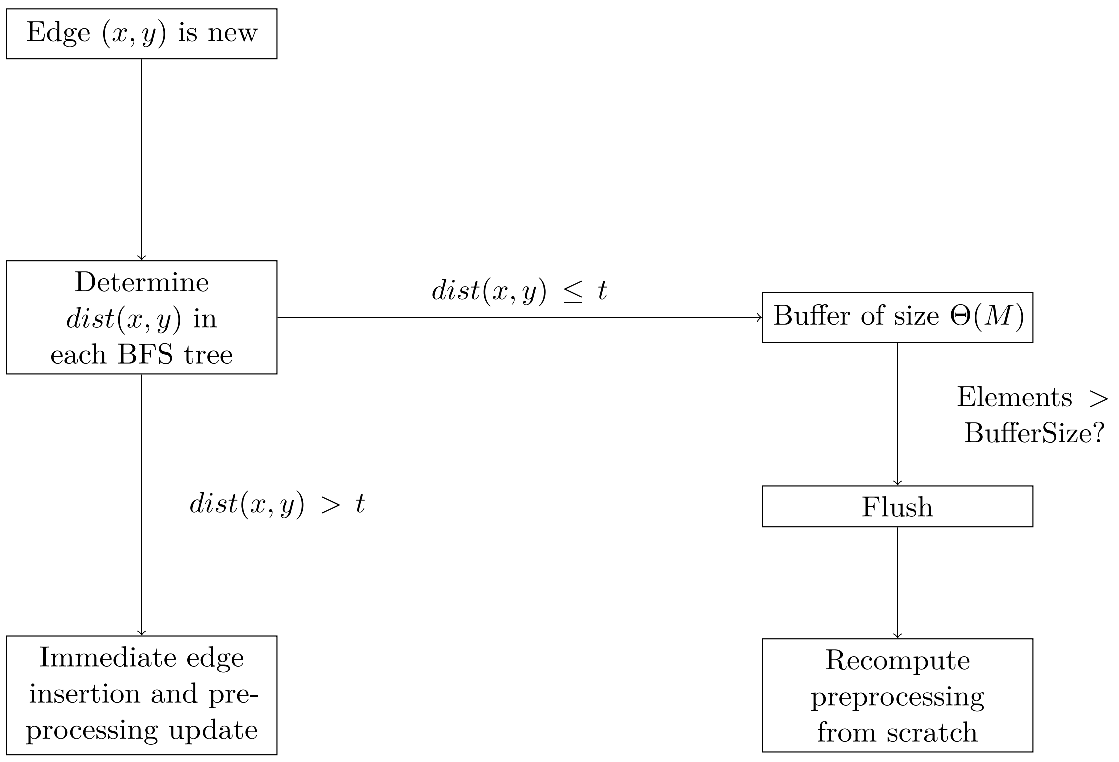

Edge (x, y) is new
Determine $dist(x, y)$ in each BFS tree
$dist(x, y) \leq t$
Buffer of size $\Theta(M)$
Elements > BufferSize?
$dist(x, y) > t$
Flush
Immediate edge insertion and preprocessing update
Recompute preprocessing from scratch

Figure 33: Chart of the program flow in our distance oracle if we allow edge insertions.

applications. One example is given by Barros et al. who answer queries on XML data streams [BLMdS16]. There is still potential for further applications that use simple label schemes to develop algorithms that can answer queries in real-time.

6 Conclusion and future work

We presented some results of external-memory algorithms that have been implemented and improved by methods from algorithm engineering. In Chapter 3, we presented techniques to approximate the diameter of huge graphs faster than with BFS, with a feasible approximation quality. During the work with different synthetic and real-world data sets, we learned that real-world data sets usually have structures as hub regions, that make it possible to use less complex implementations which are faster in the average case and easier to understand. For PAR_APPROX (refer to Section 3.7), we concluded in our PASA 2012 publication [ABMV12] as follows: "Our experiments have shown that the parametrized diameter approximation method is in fact faster than plain external-memory BFS and typically produces much better approximation bounds than the theory predicts. Nevertheless, it turns out that it is currently not suited as a section guide between different BFS approaches: as soon as the condensed graph does not fit into main memory, the overhead to run the semi-external memory SSSP is not worth the subsequent savings of a carefully chosen BFS approach. [...] Hierarchical clustering seems to be the natural choice but as the condensed graphs become weighted already after the first round, the parallel cluster growing of the next rounds needs to appropriately handle these weights, too. Currently, this step relies on the fact that the edges are unweighted".

We tackled this problem by our recursive extension REC_APPROX, where we included improvements as master movement, tie breaking and adaptive master vertex selection for the recursive steps (refer to Section 3.10 for details). With the recursive extension, we could process significantly larger external-memory graphs than before while keeping approximation ratio and computation time reasonable [AMV12]. As an interesting open question we stated the we have not looked at directed graphs. For directed graphs, the computation of BFS is more complex because the idea behind the algorithm of MR_BFS does not work anymore. MR_BFS looks two BFS-levels back to determine, if a vertex v has been visited or is part of the new level. The same is true for MM_BFS, which uses MR_BFS as a subroutine. We have done some work in the field of directed BFS together with

Rafael Franzke [Fra16] in 2016 but we have not solved the problem for efficient directed diameter approximation yet.

In Chapter 4, we provided the first dynamic external-memory implementation of a graph algorithm. We have experimentally shown that even for single updates on the BFS tree, a significant speed up can be seen between the dynamic and the static BFS tree computation. On real-world data with small diameter of $\log(n)$ many updates only take a few seconds up to a few minutes instead of hours or even days. Only in some edge cases, respectively on synthetic data, we were as slow as a computation from scratch or even slightly slower. Furthermore, we introduced a new deterministic level-aligned hierarchical clustering.

In Chapter 5, we presented our external-memory distance oracle. We used a set of BFS trees with root vertices selected by their vertex degrees and therefore, in many cases, on vertices in hub regions which cover many shortest paths. Previous work on geometric distance oracles supporting this section strategy was done by Ajwani et al. in 2014 [AKSS14]. We used vertex labels to improve the distance computation in BFS trees. In combination with real-world data with small diameters, we were able to store distance oracles with only a few kilobytes per vertex and query times of milliseconds for a single query on machines that use SSDs and even microseconds for batched queries on both kinds of hardware devices - HDDs and SSDs.

References

[AAY10] Pankaj K. Agarwal, Lars Arge, and Ke Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Trans. Algorithms*, 7(1):11:1–11:21, 2010.

[ABMV12] Deepak Ajwani, Andreas Beckmann, Ulrich Meyer, and David Veith. I/O-efficient approximation of graph diameters by parallel cluster growing - a first experimental study. In *ARCS 2012 Workshops, 28. Februar - 2. März 2012, München, Germany*, pages 493–504, 2012.

[ABT04] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004.

[ABW98] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. In *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings*, pages 332–343, 1998.

[ABW02] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[AHL14] Stephen Alstrup, Esben Bistrup Halvorsen, and Kasper Green Larsen. Near-optimal labeling schemes for nearest common ancestors. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 972–982, 2014.

[Ajw08] Deepak Ajwani. *Traversing large graphs in realistic setting.* PhD thesis, Saarland University, 2008.

[AKSS14] Deepak Ajwani, W. Sean Kennedy, Alessandra Sala, and Iraj Saniee. A geometric distance oracle for large real-world graphs. *CoRR*, abs/1404.5002, 2014.

[AM09] Deepak Ajwani and Ulrich Meyer. Design and Engineering of External Memory Traversal Algorithms for General Graphs. In *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*, pages 1–33, 2009.

[AMO06] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Breadth first search on massive graphs. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*, pages 291–308, 2006.

[AMV12] Deepak Ajwani, Ulrich Meyer, and David Veith. I/O-efficient hierarchical diameter approximation. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 72–83, 2012.

[AMV15] Deepak Ajwani, Ulrich Meyer, and David Veith. An I/O-efficient distance oracle for evolving real-world graphs. In *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 159–172, 2015.

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[AV87] Alok Aggarwal and Jeffrey Scott Vitter. The I/O complexity of sorting and related problems (extended abstract). In *Automata, Languages and Programming, 14th International Colloquium, ICALP87, Karlsruhe, Germany, July 13-17, 1987, Proceedings*, pages 467–478, 1987.

[Bav50] Alex Bavelas. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America*, 22(6):725–730, 1950.

[BCFM00] Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics*, 5:17, 2000.

[BCH$^+$15] Michele Borassi, Pierluigi Crescenzi, Michel Habib, Walter A. Kosters, Andrea Marino, and Frank W. Takes. Fast diameter and radius bfs-based computation in (weakly connected) real-world graphs: With an application to the six degrees of separation games. *Theor. Comput. Sci.*, 586:59–80, 2015.

[BCSV04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: a scalable fully distributed web crawler. *Softw., Pract. Exper.*, 34(8):711–726, 2004.

[BDS09] Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building a parallel pipelined external memory algorithm library. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–10, 2009.

[Bel54] Richard Bellman. Some applications of the theory of dynamic programming - A review. *Operations Research*, 2(3):275–288, 1954.

[Bel58] Richard Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[Ber89] Claude Berge. *Hypergraphs: combinatorics of finite sets*. North-Holland mathematical library, 1989.

[BGSU08] Surender Baswana, Akshay Gaur, Sandeep Sen, and Jayant Upadhyay. Distance oracles for unweighted graphs: Breaking the quadratic barrier with constant additive error. In *35th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5125 of *Lecture Notes in Computer Science*, pages 609–621. Springer, 2008.

[BK98] Gerth Stølting Brodal and Jyrki Katajainen. Worst-case external-memory priority queues. In *Algorithm Theory - SWAT '98, 6th Scandinavian Workshop on Algorithm Theory, Stockholm, Sweden, July, 8-10, 1998, Proceedings*, pages 107–118, 1998.

[BLMdS16] Evandrino G. Barros, Alberto H. F. Laender, Mirella M. Moro, and Altigran Soares da Silva. LCA-based algorithms for efficiently processing multiple keyword queries over XML streams. *Data Knowl. Eng.*, 103:1–18, 2016.

[Blo17] Mindaugas Bloznelis. Degree-degree distribution in a power law random intersection graph with clustering. *Internet Mathematics*, 2017, 2017.

[BMS15] Elisabetta Bergamini, Henning Meyerhenke, and Christian Staudt. Approximating betweenness centrality in large evolving networks. In *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 133–146, 2015.

[BMV13] Andreas Beckmann, Ulrich Meyer, and David Veith. An implementation of I/O-efficient dynamic breadth-first search using level-aligned hierarchical clustering. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 121–132, 2013.

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně*, 3:37 –58, 1926.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[Bru07] Irina Ioana Brudaru. Heuristics for average diameter approximation with external memory algorithms. Master's thesis, Universität des Saarlandes, October 2007.

[CDHP01] Derek G. Corneil, Feodor F. Dragan, Michel Habib, and Christophe Paul. Diameter determination on restricted graph families. *Discrete Applied Mathematics*, 113(2-3):143–166, 2001.

[CDK02] Derek G. Corneil, Feodor F. Dragan, and Ekkehard Köhler. On the power of BFS to determine a graphs diameter. In *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Cancun, Mexico, April 3-6, 2002, Proceedings*, pages 209–223, 2002.

[CDW17] Vincent Cohen-Addad, Søren Dahlgaard, and Christian Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. *CoRR*, abs/1702.03259, 2017.

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California.*, pages 139–149, 1995.

[CGI$^+$10] Pierluigi Crescenzi, Roberto Grossi, Claudio Imbrenda, Leonardo Lanzi, and Andrea Marino. Finding the diameter in real-world graphs - experimentally turning a lower bound into an upper bound. In *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, pages 302–313, 2010.

[CLRS07] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag GmbH, München, Deutschland, 2007.

[Col86] Richard Cole. Parallel merge sort. In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 511–516, 1986.

[CSTW12] Wei Chen, Christian Sommer, Shang-Hua Teng, and Yajun Wang. A compact routing scheme and approximate distance oracle for power-law graphs. *ACM Transactions on Algorithms*, 9(1):4, 2012.

[DDH03] Frank K. H. A. Dehne, Wolfgang Dittrich, and David A. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36(2):97–122, 2003.

[Dij59] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Nummerische Mathematik 1*, pages 269–271, 1959.

[DKS05] Roman Dementiev, Lutz Kettner, and Peter Sanders. : Standard template library for XXL data sets. In *Algorithms - ESA 2005, 13th*

Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings, pages 640–651, 2005.

[DKS08] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Softw., Pract. Exper.*, 38(6):589–637, 2008.

[DSSS04] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, pages 195–208, 2004.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory D. Peterson, and Jack Dongarra. From CUDA to Opencl: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[Edr12] Asmaa Edres. Deterministisches hierarchisches Clustering als Vorbereitungsphase des dynamischen BFS. Master's thesis, Goethe–Universität Frankfurt am Main, July 2012.

[EHK15] Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient basic graph algorithms. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, pages 288–301, 2015.

[Eij15] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. lulu.com, 2015.

[ELY16] Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. *CoRR*, abs/1611.00911, 2016.

[Eul36] Leonhard Euler. Solvtio problematis ad Geometriam sitvs pertinentis. http://math.dartmouth.edu/~euler/docs/originals/E053.pdf, 1736. [Online; accessed 21-March-2017].

[EWS17] Pavlos Eirinakis, Matthew D. Williamson, and K. Subramani. On the Shoshan-Zwick algorithm for the all-pairs shortest path problem. *J. Graph Algorithms Appl.*, 21(2):177–181, 2017.

[Fis09] Johannes Fischer. Short labels for lowest common ancestors in trees. In *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, pages 752–763, 2009.

[FK13] Zoltán Füredi and Younjin Kim. The structure of the typical graphs of given diameter. *Discrete Mathematics*, 313(2):155–163, 2013.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298, 1999.

[FLPR12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, 2012.

[FML$^+$12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[Fra16] Rafael Franzke. An implementation of external-memory breadth-first-search on directed graphs. Master's thesis, Goethe–Universität Frankfurt am Main, March 2016.

[Fre77] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[GGHM05] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, page 3, 2005.

[GGL$^+$13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 505–514, 2013.

[Gon07] Teofilo F. Gonzalez, editor. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.

[GvN47] Herman H. Goldstine and John von Neumann. Planning and coding of the problems for an electronic computing instrument, 1947.

[Hag18] Torben Hagerup. Fast breadth-first search in still less space. *CoRR*, abs/1812.10950, 2018.

[Han73] Gabriel Y. Handler. Minimax location of a facility in an undirected tree graph. *Transportation Science*, 7(3):287–293, August 1973.

[HKN16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the $O(mn)$ barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016.

[HR90] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[Huf52] David Albert Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1098–1101, 1952.

[HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* Chapman & Hall/CRC Computational Science, 2010.

[Jar30] Vojtěch Jarník. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně*, 6:57 –63, 1930.

[JJKP19] Yong-Yeon Jo, Myung-Hwan Jang, Sang-Wook Kim, and Sunju Park. Realgraph: A graph engine leveraging the power-law distribution of real-world graphs. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 807–817, 2019.

[JLS14] Riko Jacob, Tobias Lieber, and Nodari Sitchinava. On the complexity of list ranking in the parallel external memory model. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, pages 384–395, 2014.

[Joh73] Donald B. Johnson. A note on Dijkstra's shortest path algorithm. *J. ACM*, 20(3):385–388, 1973.

[KKT95] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[Knö16] Fabian Knöller. Eine Implementierung von Kantenlöschung für dynamische Breitensuche im Externspeicher. Master's thesis, Goethe–Universität Frankfurt am Main, April 2016.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[KS96] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996.*, pages 169–176, 1996.

[KZ16] Spyros C. Kontogiannis and Christos D. Zaroliagis. Distance oracles for time-dependent networks. *Algorithmica*, 74(4):1404–1434, 2016.

[LS10] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 303–314, 2010.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[Mey03] Ulrich Meyer. Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. *J. Algorithms*, 48(1):91–134, 2003.

[Mey08a] Ulrich Meyer. On dynamic breadth-first search in external-memory. In *STACS 2008, 25th Annual Symposium on Theoretical Aspects of Computer Science, Bordeaux, France, February 21-23, 2008, Proceedings*, pages 551–560, 2008.

[Mey08b] Ulrich Meyer. On trade-offs in external-memory diameter-approximation. In *Algorithm Theory - SWAT 2008, 11th Scandinavian Workshop on Algorithm Theory, Gothenburg, Sweden, July 2-4, 2008, Proceedings*, pages 426–436, 2008.

[MLH08] Clémence Magnien, Matthieu Latapy, and Michel Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[MM02] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear I/O. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, pages 723–735, 2002.

[MO09] Ulrich Meyer and Vitaly Osipov. Design and implementation of a practical I/O-efficient shortest paths algorithm. In *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments, ALENEX 2009, New York, New York, USA, January 3, 2009*, pages 85–96, 2009.

[Moo59] Edward F Moore. *The shortest path through a maze.* Bell Telephone System., 1959.

[Mor79] Joseph M. Morris. Traversing binary trees simply and cheaply. *Inf. Process. Lett.*, 9(5):197–200, 1979.

[MR99] Kamesh Munagala and Abhiram G. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA.*, pages 687–694, 1999.

[MSS03] Ulrich Meyer, Peter Sanders, and Jop Sibeyn. *Algorithms for memory hierarchies: advanced lectures*, volume 2625 of *LNCS*. Springer, 2003.

[MZ03] Ulrich Meyer and Norbert Zeh. I/O-efficient undirected shortest paths. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 434–445, 2003.

[MZ06] Ulrich Meyer and Norbert Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Algorithms - ESA 2006, 14th Annual*

European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings, pages 540–551, 2006.

[Ott18] Guilherme Ottoni. HHVM JIT: a profile-guided, region-based compiler for PHP and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 151–165, 2018.

[Pen17] Manuel Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, pages 26:1–26:21, 2017.

[PR10] Mihai Pătrasçu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. In *51th Annual IEEE Symposium on Foundations of Computer Science, (FOCS)*, pages 815–823. IEEE Computer Society, 2010.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *THE BELL SYSTEM TECHNICAL JOURNAL*, pages 1389–1401, November 1957.

[Pri76] Derek De Solla Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science*, pages 292–306, 1976.

[PSSZ10] Yuval Peres, Dmitry Sotnikov, Benny Sudakov, and Uri Zwick. All-pairs shortest paths in $\mathcal{O}(n^2)$ time with high probability. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 663–672, 2010.

[RS67] Alfréd Rényi and George Szekeres. On the height of trees. *Journal of the Australian Mathematical Society*, 7:497–507, 1967.

[Sab66] Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[SAM02] Jop F. Sibeyn, James Abello, and Ulrich Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, Winnipeg, Manitoba, Canada, August 11-13, 2002*, pages 282–292, 2002.

[San09] Peter Sanders. Algorithm engineering - an attempt at a definition. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, pages 321–340, 2009.

[Sei92] Raimund Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 745–749, 1992.

[Sib03] Jop F. Sibeyn. List-ranking on interconnection networks. *Inf. Comput.*, 181(2):75–87, 2003.

[Tho07] Mikkel Thorup. Equivalence between priority queues and sorting. *J. ACM*, 54(6):28, 2007.

[Tim13] Thorsten Timmer. I/O-effiziente Durchmesser-Approximierung auf gewichteten Graphen. Master's thesis, Goethe Universität Frankfurt am Main, December 2013.

[Tur38] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1938.

[TvS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms (2. ed.).* Pearson Education, 2007.

[TZ01] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 183–192, 2001.

[TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[VdS16] André Luís Vignatti and Murilo Vicente Gonçalves da Silva. Minimum vertex cover in generalized random graphs with power law degree distribution. *Theor. Comput. Sci.*, 647:101–111, 2016.

[Vei12] David Veith. Implementation of an External-Memory Diameter Approximation. Master's thesis, Goethe Universität Frankfurt am Main, February 2012.

[vL76] Jan van Leeuwen. On the construction of huffman trees. In *Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, UK, July 20-23, 1976*, pages 382–410, 1976.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, 1978.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898, 2012.

[WY14] Zhewei Wei and Ke Yi. Equivalence between priority queues and sorting in external memory. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 830–841, 2014.

[YZW$^+$98] Yanling Yang, Kaizhong Zhang, Xiong Wang, Jason Tsong-Li Wang, and Dennis E. Shasha. An approximate oracle for distance in metric spaces. In *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, pages 104–117, 1998.

[ZYQS15] Zhiwei Zhang, Jeffrey Xu Yu, Lu Qin, and Zechao Shang. Divide & conquer: I/O efficient depth-first search. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 445–458, 2015.

Graph algorithms for approximate and dynamic settings in the external-memory model

von

David Veith

Graphalgorithmen. Graphen sind seit langer Zeit ein äußerst beliebtes mathematisches Modell, um Beziehungen in Daten zwischen den einzelnen Datenpunkten zu modellieren. Schon 1736 entwickelte der Schweizer Mathematiker Leonhard Euler Graphen zur Lösung eines praktischen Problems bestehend aus der Suche nach einer Rundreise durch eine Stadt über sieben Brücken, wobei Euler anhand seiner Konstruktion zeigen konnte, dass im geforderten Fall keine Lösung existiert. Ein Graph G wird über seine Knotenmenge V und seine Kantenmenge E modelliert. In Eulers Fall waren die Inseln der Stadt die Knoten und die Brücken die Kanten. Er konnte zeigen, dass keine Rundreise existiert, bei keine Brücke zweimal verwendet werden muss, indem er sich die Knotengrade angeschaut hat. Es konnte nur einen Weg geben, wenn die Inseln so oft besucht werden, wie sie verlassen werden oder aber, wenn es eine Insel gibt, in der die Reise startet und eine Insel, in der die Reise endet. Daraus wurde die Bedingung, dass ein Eulerweg existiert, wenn alle Konten einen geraden Knotengrad haben oder genau zwei Knoten einen ungeraden Knotengrad besitzen. Die Benutzung von Graphen setzte sich aufgrund der Einfachheit und der vielseitigen Verwendbarkeit durch. Auf Graphen können Eigenschaften wie Distanzen zwischen Knoten, der Durchmesser eines gegebenen Graphs G (das Maximum unter allen kürzesten Distanzen) oder auch Beziehungen zwischen Knoten als Gemeinschaft (community detection) effizient bestimmt werden. Viele elementare Graphalgorithmen gelten seit Jahrzehnten in

Forschung und Lehre als fester Bestandteil der Werkzeugpalette zur Konstruktion effizienter Algorithmen. Anwendungsgebiete finden sich in maschinellem Lernen, dem Betreiben von Echtzeitsystemen zur Planung von Aufgaben (topologische Sortierung), zur Planung von Infrastruktur oder zur Navigation.

Externspeichermodell. Schneller Speicher ist nach wie vor ein knappes und teures Gut. Während Desktopcomputer inzwischen problemlos 16 bis 32 Gigabyte Hauptspeicher vorweisen können, haben Smartphones typischerweise 2 oder 3 Gigabyte, teure Geräte bis zu 8 Gigabyte (Stand 2019). Nach Abzug aller Arbeitsdaten reicht der Platz selten für mehr als ein bis zwei Applikationen gleichzeitig. Dementsprechend werden Arbeitsdaten auf einem langsameren, aber persistenten, Speicher wie SD-Karten abgelegt. Während der Arbeitsspeicher etwa 100 Nanosekunden für eine Datenabfrage benötigt, sind es bei der SD-Karte Millisekunden. Dieser Unterschied von mehreren Größenordnungen führt dazu, dass Smartphones scheinbar stillstehen, wenn Daten ständig zwischen der SD-Karte und dem Arbeitsspeicher ausgetauscht werden. Die Kommunikationslatenz wird in solchen langsameren Speichern durch eine hohe Transferrate von Datenblöcken der Größe B pro Kommunikationsschritt wieder wettgemacht.
1987 veröffentlichten Aggarwal und Vitter [AV87] ein Modell, welches dieses Verhalten im Externspeichermodell formalisiert.
Das Externspeichermodell beschreibt einen Computer durch einen Zentralprozessor (CPU), einen schnellen, aber in der Größe auf M begrenzten Hauptspeicher sowie einen Externspeicher unbeschränkter Größe mit hoher Latenz, auf dem die Eingabe der Länge N gespeichert wird, wobei $N \gg M$, sowie eine Kommunikation in Blöcken der Größe B. Ein Kommunikationsvorgang zwischen dem Hauptspeicher und dem Externspeicher wird als ein I/O (input/output) bezeichnet.
Ziel eines Algorithmus im Externspeichermodell ist es, die Anzahl der I/Os zur Lösung eines Problems für eine Eingabe von N Datenelementen zu minimieren. Dabei wird jedoch darauf geachtet, dass die Minimierung nicht durch eine überproportionale Last auf der CPU ersetzt wird. Zwei elementare Routinen werden im Externspeichermodell regelmäßig zum Implementieren von Algorithmen genutzt. Zum einen das Lesen einer konsekutiven Eingabe von N Elementen, wofür $\mathcal{O}(N/B)$ I/Os benötigt werden, zum anderen das Sortieren von N Daten,

welches mit $\mathcal{O}(N/B \cdot \log_{M/B}(N/B))$ I/Os durchgeführt werden kann, wobei der logarithmische Term in der Praxis Werte zwischen 2 und 5 erreicht.

Externspeicheralgorithmen auf Graphen. Breitensuche und Tiefensuche sind zwei gängige Algorithmen, um Graphen zu traversieren. Die Breitensuche wird in dieser Arbeit als vielseitige Basismethode eingesetzt, da mit einer Breitensuche die kürzeste Distanz (Anzahl der Zwischenknoten) eines jeden erreichbaren Knotens $v \in V \setminus \{s\}$ von einem Startknoten $s \in V$ bestimmt werden kann. 1999 wurde von Munagala und Ranade [MR99] der Externspeicheralgorithmus MR_BFS zur Breitensuche veröffentlicht, welche die triviale I/O-Komplexität von $\mathcal{O}(n+m)$ I/Os auf $\mathcal{O}(n + sort(n + m))$ I/Os senken konnte. Für Graphen mit einem Durchmesser d kann MR_BFS sogar mit $\mathcal{O}(d \cdot scan(m) + sort(n + m))$ I/Os analysiert werden, was für $d < B$ zu einer besseren Schranke führt.
Erst 2002 wurde mit MM_BFS ein für alle Graphdurmesser effizienter Algorithmus veröffentlicht, wobei dieser mit einer worst-case-I/O-Komplexität von $\Omega(N/\sqrt{B})$ I/Os immer noch um einen Faktor von fast $\Omega(\sqrt{B})$ teurer als die Sortierkomplexität ist [MM02]. Diese Lücke konnte bis heute nicht geschlossen werden und hat Auswirkungen auf die Verwendung von Breitensuche als Subroutine für komplexere Algorithmen.

Ergebnisse in dieser Arbeit. In dieser Arbeit werden drei Themenkomplexe aus dem Bereich der Externspeicheralgorithmen näher beleuchtet: Approximationsalgorithmen, dynamische Algorithmen und Echtzeitanfragen.
Das Thema Approximationsalgorithmen wird sowohl im Kapitel 3 als auch im Kapitel 5 behandelt. In Kapitel 3 wird ein Algorithmus vorgestellt, welcher den Durchmesser eines Graphen heuristisch bestimmt. Im RAM-Modell ist eine modifizierte Breitensuche selbst ein günstiger und äußerst genauer Algorithmus. Dies ändert sich im Externspeicher. Dort ist die Hauptspeicher-Breitensuche durch die $\mathcal{O}(n + m)$ unstrukturierten Zugriffe auf den externen Speicher zu teuer. 2008 wurde von Meyer ein Verfahren zu effizienten Approximation des Graphdurchmessers im Externspeicher gezeigt, welches $\mathcal{O}(k \cdot scan(n + m) + sort(n + m) + \sqrt{\frac{n \cdot m}{k \cdot B}} \cdot \log_2(k) + MST(n, m))$ I/Os bei einem multiplikativen Approximationsfehler von $\mathcal{O}(\sqrt{k} \cdot \log(k))$ [Mey08b] benötigt.

Die Implementierung, welche in dieser Arbeit vorgestellt wird, konnte in vielen praktischen Fällen die Anzahl an I/Os durch Rekursion auf $\mathcal{O}(k \cdot scan(n + m) + sort(n + m) + MST(n, m))$ I/Os reduzieren. Dabei wurden verschiedene Techniken untersucht, um die Auswahl der Startpunkte (Masterknoten) zum rekursiven Schrumpfen des Graphen so wählen zu können, dass der Fehler möglichst klein bleibt. Weiterhin wurde eine adaptive Regel eingeführt, um nur so viele Masterknoten zu wählen, dass der geschrumpfte Graph nach möglichst wenigen Rekursionsaufrufen in den Hauptspeicher passt. Es wird gezeigt, dass die untere Schranke für den worst-case-Fehler dabei auf $\Omega(k^{\frac{4}{3}-\epsilon})$ mit hoher Wahrscheinlichkeit steigt. Die experimentelle Auswertung zeigt jedoch, dass in der Praxis häufig deutlich bessere Ergebnisse erzielt werden.

In Kapitel 4 wird ein Algorithmus vorgestellt, welcher, nach dem Einfügen einer neuen Kante in einen Graphen, den zugehörigen Baum der Breitensuche unter Verwendung von $\mathcal{O}(n \cdot (\frac{n}{B^{2/3}} + sort(n) \cdot \log(B)))$ I/Os mit hoher Wahrscheinlichkeit aktualisiert. Dies ist für hinreichend große B schneller als die statische Neuberechnung. Zur Umsetzung des Algorithmus wurde eine neue deterministische Partitionsmethode entwickelt, bei der die Größe der Cluster balanciert und effizient veränderbar ist. Hierfür wird ein Dendrogramm des Graphen auf einer geeigneten Baumrepräsentation, wie beispielsweise Spannbaum, berechnet. Dadurch hat jeder Knoten ein Label, welches aufgrund seiner Lage innerhalb der Baumrepräsentation berechnet worden ist. Folglich kann mittels schneller Bit-Operationen das Label um niederwertige Stellen gekürzt werden, um Cluster der Größe $\mu = 2^i$ zu berechnen, wobei der Clusterdurchmesser auf μ beschränkt ist, was für die I/O-Komplexität gewährleistet sein muss, da der Trade-off aus MM_BFS zwischen Cluster- und Hotpoolgröße genutzt wird. In der experimentellen Auswertung wird gezeigt, dass die Performanz von dynamischer Breitensuche sowohl auf synthetischen als auch auf realen Daten oftmals schneller ist als eine statische Neuberechnung des Baums der Breitensuche. Selbst wenn dies nicht der Falls ist, so sind wir nur um kleine, konstante Faktoren langsamer als die statische Implementierung von MM_BFS.
Schließlich wird in Kapitel 5 ein Approximationsalgorithmus vorgestellt, welcher sowohl dynamische Komponenten beinhaltet als auch die Eigenschaft besitzt, Anfragen in Echtzeit zu beantworten. Um die Echtzeitfähigkeit zu erreichen,

darf eine Anfrage nur $\mathcal{O}(1)$ I/Os hervorrufen. Im Szenario dieser Arbeit wurden Anfragen zu Distanzen zwischen zwei beliebigen Knoten u und v auf realen Graphdaten mittels eines Distanzorakels beantwortet. Es wird eine Implementierung sowohl für mechanische Festplatten als auch für SSDs vorgestellt, wobei kontinuierliche Anfragen im Onlineszenario von SSDs in Millisekunden gelöst werden können, während ein großer Block von Anfragen auf beiden Architekturen in Mikrosekunden pro Anfrage amortisiert gelöst werden kann. Die experimentelle Auswertung zeigt, dass die Anfragen typischerweise nur einen geringen Fehler enthalten, wodurch das Distanzorakel praktische Relevanz erhält.

Curriculum vitae

Name	David Veith
Geburt	08.11.1985 in Offenbach am Main
Nationalität	deutsch

Schule

2002 – 2005	Abitur am Albert Einstein Gymnasium, Maintal

Zivildienst

2005 – 2006	Haus der Jugend in Frankfurt am Main

Studium

10/2006 – 09/2009	Bachelor Informatik an der Goethe-Universität Frankfurt Abschlussnote: sehr gut (1,5) Bachelorarbeit: TeamVision – visualisiertes Team- und Projektmanagement Betreuer Bachelorarbeit: Prof. Dr. Detlef Krömker
10/2009 – 02/2012	Master Informatik an der Goethe-Universität Frankfurt Abschlussnote: mit Auszeichnung (1,2) Masterarbeit: Implementation of an External-Memory Diameter Approximation Betreuer Masterarbeit: Prof. Dr. Ulrich Meyer

Promotionsstudium

04/2012 – 03/2017	Wissenschaftlicher Mitarbeiter an der Goethe-Universität Professur für Algorithm Engineering, Prof. Dr. Ulrich Meyer

Publikationen

PASA 2012	I/O-efficient approximation of graph diameters by parallel cluster growing – a first experimental study. Co-Autoren: Deepak Ajwani, Andreas Beckmann und Ulrich Meyer.
ESA 2012	I/O-efficient Hierarchical Diameter Approximation Co-Autoren: Deepak Ajwani und Ulrich Meyer.
ESA 2013	An Implementation of I/O-efficient Dynamic Breadth-First Search Using Level-Aligned Hierarchical Clustering Co-Autoren: Andreas Beckmann und Ulrich Meyer.
ALENEX 2015	An I/O-efficient Distance Oracle for Evolving Real-World Graphs Co-Autoren: Deepak Ajwani und Ulrich Meyer.

Erfahrungen in der Lehre

Als Hilfswissenschaftler begleitend zum Masterstudium

SoSe 2011	Betriebssysteme (Prof. Dr. Rüdiger Brause)
SoSe 2011	Datenstrukturen (Prof. Dr. Ulrich Meyer)
WiSe 2011 – 2012	Adaptive Systeme 1 und 2 (Prof. Dr. Rüdiger Brause)
WiSe 2011 – 2012	Algorithmentheorie (Prof. Dr. Ulrich Meyer)

Organisation und Betreuung von Übungsbetrieben als wissenschaftlicher Mitarbeiter

WiSe 2012 – 2013	Algorithm Engineering 1
SoSe 2013	Algorithm Engineering 2
WiSe 2013 – 2014	Theoretische Informatik I: Algorithmentheorie
SoSe 2014	Praktikum: Experimentelle Algorithmik
WiSe 2014 – 2015	Theoretische Informatik I: Algorithmentheorie
SoSe 2015	Effiziente Algorithmen
WiSe 2015 – 2016	Theoretische Informatik I: Algorithmentheorie
SoSe 2016	Praktikum: Experimentelle Algorithmik
WiSe 2016 – 2017	Theoretische Informatik I: Algorithmentheorie

Außerdem Betreuung und Organisation von diversen Seminaren sowie Seminar- und Abschlussarbeitenarbeiten in den Jahren 2012 – 2017

Erfahrungen in der Selbstverwaltung

2012 – 2017	Studienfachberater für Informatik
2016 – 2017	Mitglied des Prüfungsausschuss für Informatik

Frankfurt am Main, 20. April 2020