An I/O-efficient Distance Oracle for Evolving Real-World Graphs *

Deepak Ajwani [†*], Ulrich Meyer [‡±] and David Veith [§±]

[*]Bell Laboratories Ireland
[±]Institut für Informatik, Goethe-Universität Frankfurt

Abstract

Computing shortest path distance is a fundamental primitive in many graph applications. On graphs that do not fit in the main memory of the computing device, computing such distances requires *hours* to *months* even with the best I/O-efficient shortest path implementations. For applications requiring many such shortest path distances, one would ideally like to preprocess the input graph into a space-efficient data structure I/O-efficiently, such that the distance queries can be answered with a small additive distortion using only $O(1)$ I/Os. Furthermore, in a batch setting, one would like to answer $O(n)$ such distance queries in $\tilde{O}(n/B)$ I/Os. In this paper, we focus on engineering an I/O-efficient distance oracle for large graphs that model real-world interactions. Our engineered oracle (i) preprocesses graphs with multi-billion edges in less than an hour using a single core of a typical PC, (ii) answers online shortest path queries in milliseconds using a SSD, (iii) answers batched shortest path queries using HDDs with an average time per query of a few microseconds, (iv) results in a highly accurate shortest path estimate and (v) uses space linear in the number of nodes.

Our implementation creates small oracle labels (i.e., they can still be kept in internal memory for rather large graphs) but also efficiently handles the case when both the graph and these labels have to reside on external storage. Dynamic settings where new edges are continuously inserted into the graph are efficiently supported, too.

[*]Partially supported by the DFG grant ME 2088/3-1, and by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

[†]deepak.ajwani@alcatel-lucent.com
[‡]umeyer@cs.uni-frankfurt.de
[§]dveith@cs.uni-frankfurt.de

1 Introduction

Many search engines, social networking sites, e-commerce platforms and other businesses regularly model their real world data in the form of graphs. The features extracted from these graphs are used to offer various services (e.g., ranking web-pages, people you may know service [46, 29], product recommendation, churn prediction etc.) to users. Some of these services are in direct response to user queries, while others are recommendation services (e.g., a friend/professional you may want to connect on an online social network). While the direct user interactions obviously require low latency computations, recommendations should be computed very fast, too, since a delayed recommendation is likely to be less effective. Computing the shortest path distance between a node pair is a fundamental primitive in many graph computations. Furthermore, the distance between nodes is also one of the basic features in learning systems, such as those used for product recommendations and churn prediction. Thus, a fast way of answering distance queries in graphs is of fundamental importance for a large number of applications. What makes this problem very challenging is that (i) the distance queries have to be answered extremely fast (in microseconds), (ii) the graphs are often very large, consisting of billions of nodes and edges, and (iii) the graphs are evolving as new links are continously being inserted in these graphs.

To address the first issue of answering the distance queries extremely fast, we pre-process the graph into a space-efficient data-structure (allowing for some approximation). In the literature, such data-structures are referred to as distance oracles. As for the second issue of the graph size, i.e., to be able to deal with graphs of billion edges and more, and still be able to answer distance queries in microseconds, data-centers, computing clusters, and supercomputers can be used. However, there is a considerable expense and environmental impact associated with procuring and maintaining such systems. It is also non-trivial to run graph algorithms on such massive parallel systems efficiently. They often result in a poor speed-up due to their irregular access structure and communication requirements. Furthermore, owing to privacy concerns, companies may not want to use a cluster provided on a cloud. Ideally, we would like to be able to maintain such a distance oracle with an inexpensive and standard desktop PC with minimal additions (such as a few extra disks). The main difficulty in achieving this is that graph computations on disk often involve many random accesses and thus, suffer from a heavy I/O-bottleneck. The third issue of dealing with edge-insertions makes this even more challenging.

In this paper, we take up this challenge and engineer a distance oracle that can I/O-efficiently pre-process large graphs (with many billions of nodes and edges) in a few hours on a typical desktop PC. Our distance oracle is extremely space-efficient and it stores less than 40 bytes per node for each BFS-tree. Even this may be be too big for the main memory if in addition to the number of edges, the number of nodes in the graph is also very high. Nonetheless, we show that even if the oracle itself is stored on the disk, we can still support highly accurate distance queries in micro- to milli-seconds. This may look like an impossible goal because an I/O from a traditional hard disk drive (HDD) takes a few milliseconds and if the distance oracle is stored on a HDD, we do need to do quite a few I/Os. We solve this problem in two different ways: First, we show that by choosing a smaller block size on solid-state disks (SSDs) and carefully minimizing the constant factors in our query time, we can get the online query time in milliseconds. Secondly, we show that by using a few HDDs in parallel, we can support batched queries on HDDs in amortized time of microseconds.

Furthermore, our distance oracle can handle edge insertions in a batched manner. We show that our amortized time per update is in milliseconds for a typical PC with a small number of parallel HDDs. Thus, we can insert hundreds of edges per second and still maintain a highly accurate distance oracle (supporting microsecond queries).

An important component in our distance oracle is the I/O-efficient computation of LCA queries for small height trees. Our engineering of this component can be of independent interest for other applications as well.

Computation model Theoretical results on I/O-efficient algorithms are based on the commonly accepted external-memory (EM) model by Aggarwal and Vitter [2]. This model assumes a two level memory hierarchy with fast internal memory having a capacity to store M data items (e.g., vertices or edges of a graph) and a slow disk of conceptually infinite size. In an I/O operation, one block of data, which can store B consecutive data items, is transferred between the disk and internal memory. The measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \Theta(N/B)$. The number of I/Os required to sort N items is $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$. For graphs with n nodes and m edges, we face the fully-external case if both n amd m exceed M whereas in the semi-external case we have $n \leq M$.

Outline In Section 2, we discuss related work on distance oracles. Section 3 describes our I/O-efficient oracle. Section 4 presents our experimental framework and Section 5 shows the efficacy of our distance oracle on large

real-world graphs.

2 Related Work

There is a rich body of literature on distance oracles. In this section, we categorize this work based on whether the approach works for internal or external memory, what graph classes it is designed for and whether it is a theoretical work or has a carefully engineered practical implementation. Note that our focus is on practical I/O-efficient distance oracles for graphs capturing the real-world interactions, such as online social networks. These graphs are known to have small diameter, power-law degree distribution and many other specific properties. In the remainder of this paper, we refer to these graphs as real-world graphs.

2.1 Theoretical work on distance oracles

The seminal work of Thorup and Zwick [44] described a distance oracle that gives $2k-1$ approximation with $O(k)$ query time, $O(kn^{1+1/k})$ space and $O(kmn^{1/k})$ preprocessing time on a weighted undirected graph with n nodes and m edges, for any integer $k \geq 2$. The preprocessing time and the query time of this distance oracle were subsequently improved (e.g. [16, 15, 14, 33]), but the space versus approximation factor trade-off has remained almost the same. Recently, Sommer, Verbin and Yu [42] showed that a k-approximate distance oracle preprocessed in time t must occupy $n^{1+\Omega(1/tk)}$ space. Also, Patrascu and Roditty [38] have proved various lower bounds under plausible conjectures for the space versus worst-case approximation factor trade-off. Such lower bounds suggest that it is unlikely that a distance oracle can result in a significantly better trade-off for general weighted graphs. This limitation on general graphs have led the study of distance oracles for specialized graph classes. There are some better theoretical results known for special classes of graphs, such as for planar graphs [43], geometric (Euclidean) graphs that are t-spanners (for $t > 1$) [28], graphs with doubling dimension [12, 30], but they are not known to have any practical implementation suited for real-world graphs.

2.2 Distance oracles on road networks

Distance oracles for road networks have received a considerable attention owing to their applications in navigation systems. Many approaches are known (e.g., [13, 23, 24, 40]) for supporting exact or close to exact shortest path queries with efficient preprocessing and little space. For a recent survey on routing in road networks (including a detailed comparison of various algorithms for road networks, with different tradeoffs between preprocessing and query times, as well as space usage), we refer the readers to [22].

These distance oracles crucially rely on many specific characteristic properties of road networks such as the existence of small natural cuts, a grid-like structure, highway hierarchies, etc. On other graph classes such as those arising from online social networks, these solutions are not known to produce equally good approximations. Also, the road networks are typically not very big (a few tens of millions of vertices for a whole continent), so most algorithms and systems work with the assumption that the graph fits in the main memory.

2.3 Internal memory distance oracles for social networks

In recent years, many distance oracles have been developed for real-world graphs. These techniques ([1, 47, 45, 39]), which involve storing a few bytes of labels with each node, claim to provide very high accuracy in estimating shortest path distances on real-world graphs, particularly, online social networks. However, the preprocessing in these oracles becomes extremely slow if the graph is so big that it does not fit in the main memory.

2.4 Exact distance oracles

Very recently, considerable progress has been made in designing and engineering exact distance oracles [21, 7]. These distance oracles return the exact shortest path distance for any query. Some of these techniques even work well in a dynamic setting with evolving graphs [8]. However, for social network graphs, these techniques often result in large label sizes and so, the distance oracle requires a very large storage space. In fact, in discussions [37] with some authors of [21], they essentially excluded that our larger test instances could be processed by their RXL approach using standard PC RAM sizes and quite likely still on their big server with 384 GB RAM. We are also not aware of any I/O-efficient preprocessing for these techniques.

2.5 External memory graph traversal

To deal with graphs that do not fit in the main memory, many I/O-efficient traversal algorithms have been designed (e.g., [32]) and engineered [5, 3, 35, 6] in the last two decades. However, for graphs with billions of edges, the computation of shortest paths from a single source using these techniques still requires many hours and thus, these techniques are not suited for applications that require a large number of shortest path computations. Nonetheless, they are useful tools for the preprocessing stage in our distance oracle. Dealing with many sources at the same time has been explored in the context of all-pairs shortest-paths [10, 19] and betweenness centrality [9]. The theoretical gains of better block utilization, however, are diminished by the huge (and practically infeasible) overall I/O and/or space consumption when considering these problems in external memory.

2.6 External memory distance oracles

While there is a considerable research done on distance oracles in the RAM model and there are many results for external memory graph traversal, there is very little work on distance oracles with I/O-efficient pre-processing. This is surprising given that the seminal work by Thorup and Zwick [44] already mentioned an I/O-efficient pre-processing data structure. In fact, this first idea was already adapted by Sarma et al [41] for a distributed implementation.

Derungs et al. [26] present a theoretical approach to an I/O-efficient distance oracle. However, their preprocessing stage involves creation of a dense graph, that can have many more edges than the original graphs. Although this is later sparsified using a graph spanner algorithm, the storage and retrieval of this intermediate graph can be very expensive. For the graph sizes that we are interested in, the storage space of such an intermediate graph can also be prohibitive. Furthermore, their queries involve doing a few levels of BFS around the two query nodes in the original graph. We are interested in oracles that can get us query time in micro- to milli-seconds even with disk-based graph storage and this work is unlikely to yield such query times.

Zhu et al. [48] have explored an I/O-efficient exact oracle for single-source-all-destination queries (as opposed to our problem of point-to-point shortest path queries). Their approach is based on graph contraction. However, they implicity rely on certain special graph properties for their preprocessing technique to be efficient and seem to restrict their experiments on the semi-external case. Beyond these restrictions, their preprocessing can suffer from prohibitive slowdown. For example, on our test instance sk-2005 (described in Section 4.2) the proprocessing of Zhu et al.'s approach did not finish even after more than 200 hours (on a machine with 30 HDDs in parallel). From our traces it seems that the contraction does not manage to remove sufficiently many vertices per phase in order to counterbalance the large number of newly inserted shortcut edges for this kind of instances.

3 Engineering an I/O-efficient Distance oracle

In this section, we consider the design of a distance oracle that will allow for micro-to-milli second queries even when the graph does not fit in the main memory. To attain this query time, we aim for a query complexity of $O(\log n)$ I/Os with small block sizes on SSDs for online queries and $O(\log(n)/B)$ amortized I/Os for batched queries on HDDs. By itself, this is not difficult as one can store a matrix of all-pair shortest path and answer all queries in $O(1)$ I/Os online and $O(1/B)$ amortized I/Os for batch queries. However for graphs with many billions of nodes, this will require Exabytes of space and a huge preprocessing time. On the other hand, we want to obtain near-linear (in terms of the number of nodes) space for our distance oracle and a preprocessing time of a few hours. Since a single computation of BFS can take hours on graphs of such sizes, our precomputation shouldn't require more I/Os than a few BFS computations. Furthermore, the distance estimate that we return should be highly accurate. As described in Section 2, it is unlikely that we can attain these goals with worst-case approximation guarantees on general graphs (because of the various lower bounds that are known for this problem). Thus, we focus on techniques that provide a good estimate of shortest path distances on real-world graphs in practice.

In the past, distance oracles in this category (e.g., [41, 39]) were mainly based on carefully selecting a set of landmarks and storing the distance from these landmarks for each node. When answering a query for distance between two nodes x and y, the minimum sum of distances from x and y to a landmark is returned. However compared to these techniques, better accuracy can be obtained with the same number of landmarks if rather than just storing the distances from the landmark, the whole BFS tree could be encoded instead. Clearly, the unique distance between nodes x and y in a tree T rooted at a landmark r is no greater than $d(x, r) + d(y, r)$ as returned

by the landmark based schemes and is no less than the distance $d(x, y)$ in the input graph G, thereby resulting in better distance estimates. Note that this can increase the space requirement of the distance oracle, but in Section 3.2 and Section 3.3, we show that we can encode the distances in a tree with a constant number of bytes per node and still answer the online distance queries with $O(1)$ I/O per tree. This is done by leveraging the fact that the BFS trees of real-world graphs have small height. In Section 5.3, we also show that the resultant increase in preprocessing time is also acceptable.

Thus, the basic idea of our approach is to compute a set of BFS trees from high-degree nodes and then answer the distance queries by returning the minimum distance found between the nodes in the different trees. In [4], it was shown that this simple approach provides as good an accuracy as more complex distance oracle techniques (such as those based on embedding of graphs [47] in R^{10} using a Hyperboloid model with a small curvature) on real-world graphs. Next, we present the details of how we implement this oracle I/O-efficiently.

3.1 Multiple BFS computations

To compute the BFS I/O-efficiently, we use the Ajwani et al's [3] implementation of the Munagala and Ranade's BFS algorithm [36] (MR_BFS). A single MR_BFS run requires $O(\min\{d \cdot \mathrm{sort}(m), n + \mathrm{sort}(m)\})$ I/Os for a graph with diameter d and is thus, particularly efficient for small diameter graphs. The algorithm computes the BFS tree level by level. Given the nodes in level i, it computes the nodes $L(i+1)$ in level $i+1$ by first scanning all the edges of the graph to compute the neighbors $N(i)$ of nodes in level i. This set of neighbors $N(i)$ is then scanned in parallel with nodes in level $L(i)$ and $L(i-1)$ to compute $L(i+1) := N(i)/\{L(i) \cup L(i-1)\}$. To compute multiple runs from t different source nodes, we scan the graph jointly to compute the sets $\bigcup N(i)$. Although this does not affect the asymptotic complexity of $O(t \cdot d \cdot \mathrm{sort}(m))$, it reduces the constant factors in the read I/Os.
If the graph size is such that we can keep one bit per node in the main memory, our implementation automatically switches to a semi-external MR_BFS implementation. Compared to a fully external BFS, it saves some important constant factors helping us to reduce the preprocessing time by a multiplicative factor of 2-3 in this setting.

3.2 Queries

We support two different queries - an online query using SSDs and a batched query using parallel HDDs. For the online query, we also consider two variants to solve it.

The first variant is to find the nodes u and v in each BFS tree T and then traverse up in the tree to the root r of T. The common part between the two paths is then removed to get the unique path between u and v in the tree T. The length of the shortest such paths among all trees, is then returned as the approximated distance. In the RAM model, this variant performs faster than many complex approaches with $O(1)$ query time simply because the height of our BFS trees is very small (bounded by the small graph diameter) and the constant factors hidden in the other approaches were high. However in the disk setting, the path traversal up the tree incurs $\Omega(h)$ I/Os for a tree with height h. In Section 5, we show that even when using a small block size with a solid state disk (SSD) on a carefully configured machine, the query time of this variant for a graph sk-2005 (with 1.8 billion edges) was around 120 milliseconds using 20 trees. This is still quite slow for the applications we are targetting – both for applications where there is a low latency requirement and for applications that require a large number of such queries. For million such queries, the runtime will be around 1.4 day.

Thus, it makes sense to consider ways to encode the trees such that the distance queries can be answered with $O(1)$ I/Os for each tree, resulting in $O(t)$ I/Os overall, where t is the number of BFS trees. Note that the distance $d(u, r)$ for each node in the BFS tree is already computed during the MR_BFS computation and can be stored with the node. The distance between two nodes u and v in T can be determined as $d(u, r) + d(v, r) - 2 * d(lca(u, v), r)$, where $lca(u, v)$ is the least common ancestor of u and v in T. Since, $d(u, r)$ and $d(v, r)$ are already stored, we only need to keep some additional label such that $lca(u, v)$ can be computed in $O(1)$ I/Os. In Section 3.3, we describe how we can I/O-efficiently compute such a label that requires little space.

To reduce the I/Os even further, we store all the t distance-to-roots and the labels for efficient LCA computation contiguously on the disk for each node. Thus, to answer the query in this second variant, we first perform an SSD I/O (with a smaller block size) to obtain $d_{T_i}(u, r_i)$ and $l_{T_i}(u)$ for $1 \leq i \leq t$ and another I/O to determine all $d_{T_i}(v, r_i)$ and $l_{T_i}(v)$. Then, we compute the LCA of u and v in each tree T_i using the labels $l_{T_i}(u)$ and $l_{T_i}(v)$. However, we still need to perform $O(t)$ I/Os to determine $d_{T_i}(lca_{T_i}(u, v), r)$ that is stored with the node $lca_{T_i}(u, v)$. Once, these values are computed, we can compute the length of the unique path between u and v in each tree and

return the minimum among them. In Section 5, we show that this variant is significantly faster than the previous variant without increasing the space requirement of the oracle significantly.

For the batched queries on HDDs, we answer Q queries using $O(sort(|Q|) + scan(n \cdot t))$ I/Os. For $|Q| = n$, this results in an amortized $O((\log_{M/B} n + t)/B)$ I/Os per query. To answer the batched queries, we first sort the query end-vertices by node ids and scan this sorted set together with a tree to collect the distances and the labels of the query end-points. The query set is then sorted back such that the two end-points of the query are together. LCAs for all queries are then computed with one scan and in another scan with the tree, the distances of the LCAs from the root are determined and subtracted to answer the queries.

3.3 I/O-efficient encoding for LCA queries

Shortest paths in a tree can be exactly encoded using $O(\log^2 n)$ bits [27] per node and $O(1)$ query. On the other hand, storing only the parent node label with each node (i.e., $O(\log n)$ bits) suffices to answer distance queries, although the query complexity in this case is $O(h)$, where h is the height of the tree. As this involves repeatedly following the parent pointers from two nodes to the root in the query, the $O(h)$ RAM query translates to $O(h)$ I/Os in the external memory.

We engineer a middle way that allow us to have the best of both extremes: an amortized label size of $O(\log n+h)$ bits per node, while ensuring a query complexity of $O(\frac{h}{\log n})$. Since our trees are BFS trees, their height is bounded by the diameter of the input graph. Thus, for small diameter real-world graphs, $h = O(\log n)$ for our trees and we achieve $O(\log n)$ bits per node (or $O(1)$ machine-word per node) encoding that still gives $O(1)$ query time. The total size of encoding a single tree is thus $O(n)$ words.

Complete Binary Tree We first observe that for complete binary trees, merely keeping the inorder number ($O(\log n)$ bits) with each node allows us to answer distance queries with $O(1)$ instructions. This is done by first computing the least common ancestor z for the query node pair x, y. The distance $d(x, y)$ is then given by $d(x, r) + d(y, r) - 2d(z, r)$, where r is the root of the tree. Since the inorder numbering captures the path from root to a given node, the least common ancestor can be computed easily. We first identify the leftmost bit where the inorder numberings of x and y differ, pad it up with 0s on the right and the common bits of x and y on the left. The distance of a node x from the root r is computed as the position of the last one bit in the inorder number of x minus one.

Generalization to a Tree of Arbitrary Degree To generalize the above approach to a tree T with arbitrary degree, we first transform $T(V, E)$ into a weighted binary (not necessarily complete) tree $T'(V', E')$ such that $V \subseteq V'$ and for any two nodes $u, v \in V$, $d_T(u, v) = d_{T'}(u, v)$. For a node $u \in V$ with degree k greater than 2, let $u_1, u_2, \ldots, u_k$ be its children in T (i.e., $\{u, u_1\} \in E, \{u, u_2\} \in E$, ..., $\{u, u_k\} \in E$; cf. Figure 1). We construct the binary tree T' by introducing some new nodes u'_1, u'_2, ..., u'_{k-2}. To add these new nodes, we first compute the number of descendants of a node (including itself) in a bottom-up way. Then, we use the Huffman coding [31] procedure with the number of descendants as weights to add the new nodes. Specifically, we start with a set S of the k children $u_1, u_2, \ldots, u_k$. Let u_i and u_j be the nodes with the two smallest weights in S. We then introduce a new node u'_l as a parent of u_i and u_j in T', remove u_i and u_j from S and add u'_l into S. The weight of u'_l is the sum of weights of u_i and u_j. The procedure is repeated till only two nodes remain in S and at this stage, we add these two nodes as children of u (cf. Figure 2). The edges connecting the new nodes u'_1, u'_2, ..., u'_{k-2} to their parents have length 0, while the edges connecting the nodes $u_1, u_2, \ldots, u_k$ to their parents in T' have length 1. This process is repeated independently for all nodes with degree greater than 2.

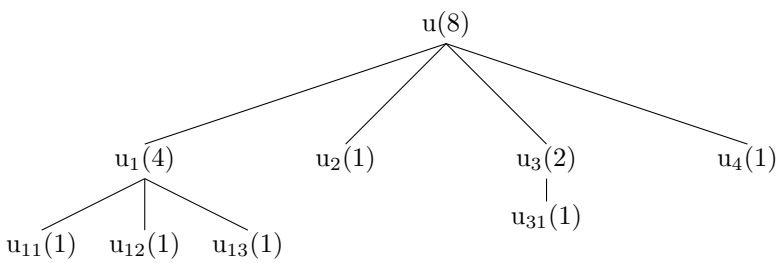

Figure 1: Node u in tree T has a degree of four. The values in brackets are the number of their descendants.

In appendix A, we show that the space required by our distance encoding technique is $O(n)$ words (or $O(n \log n)$

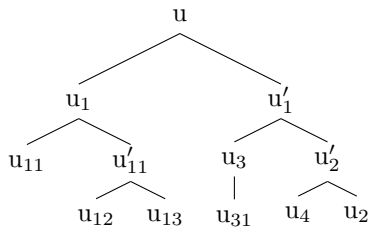

Figure 2: Part of the tree T' corresponding to Figure 1

bits), where $n := |V|$ for trees with height $O(\log n)$. In our experiments, we found that 64 bits per node was always enough to store the LCA labels of all the BFS trees on all the graphs that we considered (Section 4.2). To compute the LCA labels efficiently in external memory, we sort the nodes by their distance from the root in decreasing order and consider the nodes in the tree from the leaves to the root. Using time-forward processing [18], we compute the number of descendants for each node (including itself). Then, we sort the nodes in increasing distance from the root. Using time-forward processing with this order, we now compute the labels of all nodes. The label of the root node is assigned to be 2^{63}. For a parent with k children, the $\log k$ bits after the last one in the parent's label are updated according to Huffman encoding with the number of descendants as the weight of a node.

3.4 An Alternative distance oracle

We also consider an alternative distance oracle. This approach consists of computing a hierarchy of BFS forests. At the top level or 0th level, we have a BFS tree from a random source covering all the nodes of the graph. At the ith level, we have 2^i BFS trees from random sources and we grow the BFS trees from these sources in parallel, till all nodes are contained in some BFS tree. Ties are broken arbitrarily.

Note that each node needs to keep its distance from $O(\log(n))$ tree roots – one tree from each level, as it is covered by exactly one tree in each level. In addition, it needs to keep a constant number of bytes to store its LCA label in the corresponding tree and some index to mark the hierarchy level of the tree (as some roots may be common in different levels of hierarchy). Thus, the space required per node is $O(\log(n))$ bytes per node.

The preprocessing can be done similar to the randomized preprocessing of the I/O-efficient BFS algorithm of Mehlhorn and Meyer [32], where BFS from different source nodes is grown in parallel till all nodes are covered.

To answer a query between two nodes x and y, we consider the $\log(n)$ trees in which the two nodes belong and identify the common trees. Then from among the common trees, we compute the minimum distance between them using LCA labels, similar to the way described in Section 3.2 and Section 3.3 and return this as the estimated distance. To do this efficiently, we also store a STL map (in internal memory) from (hierarchy_level, root) to the corresponding arrays with the distance and LCA labels.

Note that Sarma et al. [41] had proved a bound of $2c-1$ approximation for the corresponding landmark based scheme, if the hierarchy was computed $\tilde{\Theta}(n^{1/c})$ times (with independent random source set) and then the distance was returned as the minimum among them. As shown before, the distance returned by taking the minimum BFS tree distance is always more accurate than the minimum sum of distances to landmarks and as such, this approach has at least as good worst-case bounds as that of Sarma et al. As a corollary, we get that with $\tilde{\Theta}(1)$ such BFS-tree hierarchies, we get a worst-case approximation ratio of $2 * \log(n) - 1$ for distances on arbitrary graphs.

To save on the space requirement, we only consider one such hierarchy. Furthermore, we only use enough levels such that in the last level there are $O(n/C)$ roots for a large constant C. This has the additional advantage that our STL map that points to the array of LCA labels fits internally (assuming $n/C \leq M$; otherwise an external memory STXXL map can be used instead). As we show later, this is already enough to achieve small distortion on real-world graphs in practice. However, we show in Section 5 that the resultant accuracy-space trade-off from this approach is slightly worse than the BFS from high-degree-nodes technique outlined earlier.

3.5 Supporting edge insertion

We also support insertion of edges in our distance oracle. To update the BFS trees we use Beckmann et al's implementation [17] based on Meyer's dynamic BFS algorithm [34]. When a new edge $\{u, v\}$ is inserted into the

graph, we first query the distance oracle to find the distance between u and v. If the distance is above a certain threshold, we update the BFS trees (and the corresponding encodings). Otherwise, we simply keep it in a buffer for batched updates later. When the buffer gets full, we recompute all the BFS trees and their encodings from scratch.

3.6 Extension to weighted graphs

It is easy to extend this implementation to weighted graphs. We start by computing single source shortest path trees from high degree nodes, rather than the BFS trees. The semi-external SSSP implementation [35] can be used for this purpose. While computing the shortest path trees, we also compute the distance of each node from the root in the tree. The encoding of the trees for supporting efficient LCA queries remains exactly the same.

However, note that in our experimental analysis, we only focus on undirected and unweighted graphs with a small diameter d.

4 Experiments

In this section, we describe our experimental set up to evaluate the efficiency of our oracle.

4.1 Configuration

Our external-memory distance oracle implementation relies on the STXXL library 1.3.1 [25]. We performed our experiments on a machine A with a AMD FX(tm)-4170 Quad-Core Processor processor @ 4.2GHz, 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) @ 4.1GHz, 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 only require a small fraction of the main memory offered by A and B. Our experience with larger cache usage by the STXXL is that it usually slows down the experiments because the overhead of maintaining a larger cache for each data structure is larger than the little constant factor decreased in the total number of I/Os.
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.

4.2 Graph classes

To evaluate the preprocessing time, online and batched query time as well as space, we consider some large real-world and synthetic graphs. The real-world graph sk-2005 has around 50 million nodes, about 1.8 billion edges and is based on a web-crawl of the Slovakian internet in 2005 (refer to http://law.di.unimi.it/webdata/sk-2005). This graph was also used by Crescenzi et al. [20] and has a known diameter of 40.
The real-world graph com-friendster has around 65 million nodes and about 1.8 billion edges and is based on the user data of the old Friendster network. The diameter is 32 and the graph data can be found on the data base SNAP (https://snap.stanford.edu/data/com-Friendster.html).
We generated the preferential attachment graph price-100mio using a graph generator by Atwood et al. [11] with 100 million nodes for comparison reasons. It took 4.5 days to generate a graph of this size and almost 20 GB main memory. However, the generated graph is a pseudo-tree with 100 million edges. Therefore we have added a circle of length n to price-100mio to preserve the preferential attachment structure but increase the complexity. It has a diameter of 14 (without the circle: 58).
The synthetic graph graph_2b_8b_d30 has 2.1 billion nodes and 8.4 billion edges. The diameter of this graph is 30. It has a regular structure with around 71 million nodes per level. Only the first level has one vertex. The vertex labels are permuted.
To evaluate the accuracy of our distance oracle we have solved all pairs shortest path on a set of small graphs (ca-AstroPh, dblp, Facebook NY, hyperGrid, p2p-Gnutella31, Facebook SantaBabara, web-BerkStan) with up to a million nodes. Their diameter is between 5 and 23. Details about these graphs can be found in [4].

5 Results

In this section, we show the efficacy of our I/O-efficient distance oracle.

5.1 Comparing the accuracy of the distance oracles

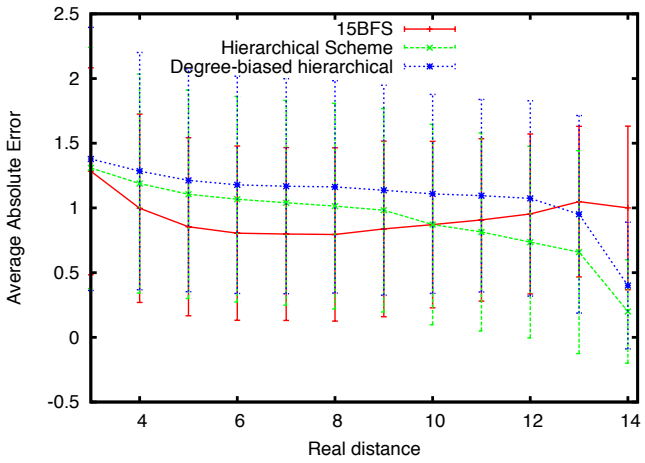

Figure 3: Average absolute error of the two distance oracle techniques on Facebook SantaBarbara graph.

Figure 3 shows the accuracy of three different distance oracles on the Facebook Santabarbara graph. Our first variant uses 15 BFS trees from high degree nodes, our second variant uses the hierarchical family of BFS trees from random source nodes and the third variant uses a hierarchical family of BFS trees from high degree nodes. We first observe that all the three variants are fairly accurate. The average absolute error, defined as the average difference between the estimated and the actual distance (the estimated distance is always higher than the actual distance) over all node pairs with a given real distance, is always smaller than 1.5 for all three approaches. The relative comparison shows that the first variant is more accurate for node pairs with small distances in the graph while the second variant is better for larger distance node pairs. Interestingly, the hierarchical BFS tree family from high degree nodes always performed poorly compared to the hierarchical version with random sources. On the other hand, the BFS from high degree nodes are significantly more accurate than BFS from random source nodes [4]. In the rest of the section, we focus on the distance oracle using the BFS trees from high-degree nodes as many applications like people-you-may-know are primarily interested in accurate answers of node pairs with small real distances.

5.2 Accuracy of the high-degree BFS oracle

Computing all pair shortest path distances on sk-2005 requires a very large running time and prohibitive space. Thus, to evaluate the accuracy of the resultant shortest paths on this graph, we sample 60,000 random source nodes and query the distance of all nodes from these 60,000 nodes. We then use a 40-node cluster of LOEWE CSC to compute the exact distances for these query nodes over two days. We found that for this graph, more than 80% of the queries have been answered correct. For the remaining queries, we observed that the error from the real distance was very small.

For the smaller graphs, we analyzed the queries over all node pairs. We found that the accuracy of our result was dependent on the structure of the graph. On complex graphs such as DBLP, we only got 41% correct answers. However, even for this graph, 48% of the node pairs had an error of exactly 1. Thus for 89% of the queries, our oracle returned an estimate that was within an error of 1.

We refer the readers to figures 4 and 5 for more details on the accuracy for the different smaller graphs using high degree nodes as sources. For random sources the shape of the curves is similar but the accuracy of the queriesis a bit worse than with high degree sources. Note that our accuracy results are in line with those observed in [4].

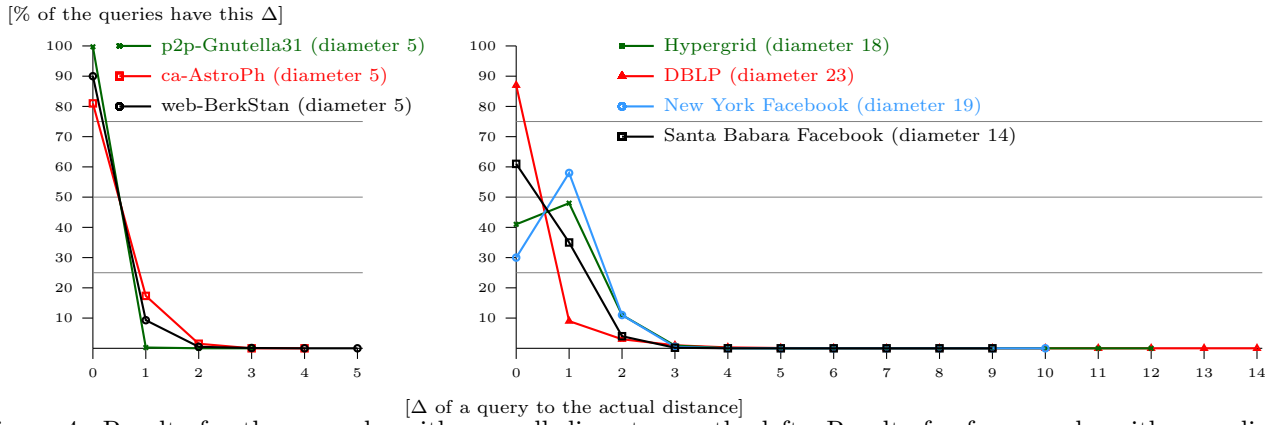

Figure 4: Results for three graphs with a small diameter on the left. Results for four graphs with a medium diameter on the right. For the sake of clarity only the results with high degree sources are presented.

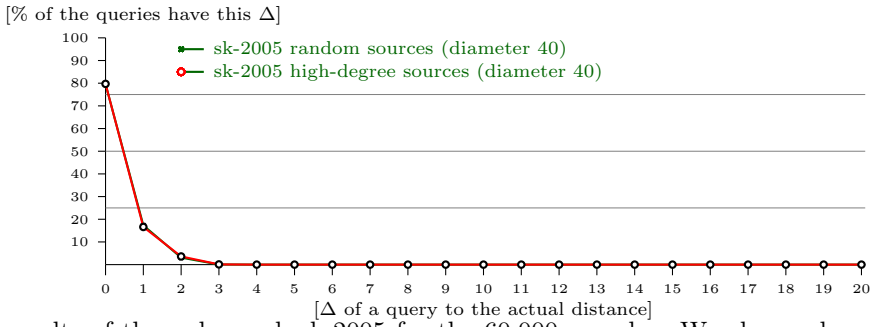

Figure 5: Detailed results of the web graph sk-2005 for the 60,000 samples. We observed an accuracy of almost 80% with both, random and high degree vertices as sources.

5.3 Preprocessing time and space requirements

For sk-2005 on machine A, the total time for the preprocessing of our oracle (with the semi-external BFS variant) is just around 2.7 hours with 20 BFS trees. It took less than 5 minutes to compute a BFS and less than 4 minutes to encode the trees. Since the preprocessing happens only once in the static case it is as good as can be hoped. The variant of preprocessing based on a fully external MR_BFS takes around 20 minutes per tree. The result 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. The preprocessing time of price-100mio was similar to sk-2005

On the graph com-friendster, it takes 3.8 hours to compute the preprocessing with 20 BFS trees. The timing results on machine B are similar to them on A. On 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 with sk-2005 is between 8.7 GB for 5 BFS trees and 173.6 GB for 100 trees. For 20 BFS trees the space consumption 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.

5.4 Online queries with SSDs and batched queries with HDDs

Online queries are a common application of distance oracles. A user might want to know how close an item is related to another item in a network in real time. Therefore we evaluated single queries for sk-2005 on different machines. As described in Section 3.2, we need a constant number of I/Os, namely $O(t)$ where t is the number of BFS trees. On common HDDs the query time is about a second for 20 BFS trees on sk-2005. If we use a similar block size of 512 KB per disk, we get the same result on SSDs. However, we found that by decreasing the block

Queries	average time per query on machine A	average time per query on machine B
$\tilde{}2^{27}$	26,078 μs	16,274 μs
$\tilde{}2^{28}$	24.562 μs	15,571 μs
$\tilde{}2^{29}$	23.926 μs	15,626 μs
$\tilde{}2^{30}$	23.616 μs	15,759 μs
$\tilde{}2^{31}$	23.709 μs	16,144 μs

Table 1: Average query time for a batched query on machines A and B for different number of queries on sk-2005.

size on SSDs to 16 KB respectively 8 KB, we can reach 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 get 6.7 ms.
Batched queries work for applications without real time requirements such as those in some learning systems. By batching $\theta(n)$ queries, we achieve a query time in the range of microseconds for sk-2005. For different query times with 20 BFS trees on sk-2005, we refer the readers to Table 1.
For price-100mio, we observed that the query time only differs by a small factor as compared to sk-2005.

5.5 Update time for edge insertion

For the experiments to determine the update time, we first compute a spanning tree of the real-world graph and use this to initialize the distance oracle. We then insert the remaining edges in a random order. 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 BFS trees or it is batched for a future update, and on the size of the buffer that we keep for the update. These two parameters allow us to navigate 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 in milliseconds per edge. Our preliminary experiments suggested that the resultant accuracy loss owing to batching such a large number of random edges is still a small additive factor, particularly when the graph has a large number of edges already. On the other hand, updating a BFS tree with Beckmann's implementation of dynamic BFS can take many minutes, as already shown in [17]. Thus to keep our average update time small, which is a requirement for some applications, we can keep the threshold fairly high (relative to the graph diameter) and the buffers fairly large. Note that most nodes in real-world graphs lie in a tightly connected core and thus, most node pairs have very small distance between them as compared to the graph diameter. Thus, when considering edges in a random order, a large number of these edges get batched and do not require instant update.

6 Conclusion

We presented a distance oracle for real-world graphs with an I/O-efficient preprocessing of a few hours on graphs with many billions of nodes and edges. It provides distance estimates that are exact for a large number of node pairs and only have a very small additive error on the remaining node pairs. Our oracle requires less than half a KB of label with each node. Even when such an oracle is too big to be stored on an external storage, we show that we can still support online queries on SSDs in milliseconds and batched queries on HDDs in a few microseconds per query for graphs with many billions of nodes and edges. Furthermore, we show that our oracle can be I/O-efficiently updated whenever new edges are inserted in the graph.

Acknowledgments

We want to thank Daniel Delling, Renato Werneck, Thomas Pajor, Xiaokui Xiao and Zhu DiWen for providing the results respectively source code from their distance oracles for comparison.

References

[1] Abraham, I., Balakrishnan, M., Kuhn, F., Malkhi, D., Ramasubramanian, V., and Talwar, K. Reconstructing approximate tree metrics. In *Proc. of PODC* (New York, NY, USA, 2007), ACM, pp. 43–52.

[2] Aggarwal, A., and Vitter, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM, 31(9)* (1988), 1116–1127.

[3] Ajwani, D., Dementiev, R., and Meyer, U. A computational study of external memory bfs algorithms. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2006), pp. 601–610.

[4] Ajwani, D., Kennedy, W. S., Sala, A., and Saniee, I. A geometric oracle for distance approximation in large real-world graphs.

[5] Ajwani, D., and Meyer, U. Design and engineering of external memory traversal algorithms for general graphs. In *Algorithmics of Large and Complex Networks* (2009), vol. 5515 of *LNCS*, Springer, pp. 1–33.

[6] Ajwani, D., Meyer, U., and Veith, D. I/O-efficient hierarchical diameter approximation. In *ESA* (2012), pp. 72–83.

[7] Akiba, T. Pruned labeling algorithms: fast, exact, dynamic, simple and general indexing scheme for shortest-path queries. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume* (2014), pp. 1339–1340.

[8] Akiba, T., Iwata, Y., and Yoshida, Y. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014* (2014), pp. 237–248.

[9] Arge, L., Goodrich, M. T., and van Walderveen, F. Computing betweenness centrality in external memory. In *Big Data, 2013 IEEE International Conference on* (2013), IEEE, pp. 368–375.

[10] Arge, L., Meyer, U., and Toma, L. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)* (2004), vol. 3142 of *LNCS*, Springer, pp. 146–157.

[11] Atwood, J., Ribeiro, B. F., and Towsley, D. Efficient network generation under general preferential attachment. *CoRR abs/1403.4521* (2014).

[12] Bartal, Y., Gottlieb, L., Kopelowitz, T., Lewenstein, M., and Roditty, L. Fast, precise and dynamic distance queries. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011* (2011), pp. 840–853.

[13] Bast, H., Funke, S., and Matijevic, D. Ultrafast shortest-path queries via transit nodes. In *The shortest path problem : ninth DIMACS implemenation challenge* (2009), C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds., vol. 74 of *DIMACS Series on Disrecte Mathematics and Theoretical Computer Science*, AMS, pp. 175–192.

[14] Baswana, S., Gaur, A., Sen, S., and Upadhyay, J. Distance oracles for unweighted graphs: Breaking the quadratic barrier with constant additive error. In *35th International Colloquium on Automata, Languages and Programming (ICALP)* (2008), vol. 5125 of *Lecture Notes in Computer Science*, Springer, pp. 609–621.

[15] Baswana, S., and Kavitha, T. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *47th Annual IEEE Symposium on Foundations of Computer Science FOCS* (2006), IEEE Computer Society, pp. 591–602.

[16] Baswana, S., and Sen, S. Approximate distance oracles for unweighted graphs in expected $o(n^2)$ time. *ACM Transactions on Algorithms 2*, 4 (2006), 557–577.

[17] Beckmann, A., Meyer, U., and Veith, D. 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* (2013), pp. 121–132.

[18] Chiang, Y. J., Goodrich, M., Grove, E., Tamassia, R., Vengroff, D., and Vitter, J. External memory graph algorithms. *In Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (1995), 139–149.

[19] Chowdury, R., and Ramachandran, V. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *Proc. 16th SODA* (2005), ACM-SIAM, pp. 735–744.

[20] Crescenzi, P., Grossi, R., Imbrenda, C., Lanzi, L., and Marino, A. Finding the diameter in real-world graphs – experimentally turning a lower bound into an upper bound. In *Proc. 18th ESA* (2010), vol. 6346 of *LNCS*, Springer, pp. 302–313.

[21] Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. Robust distance queries on massive networks. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings* (2014), pp. 321–333.

[22] Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. Robust exact distance queries on massive networks. Tech. Rep. MSR-TR-2014-12, July 2014.

[23] Delling, D., Goldberg, A. V., Razenshteyn, I., and Werneck, R. F. F. Graph partitioning with natural cuts. In *25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (2011), IEEE, pp. 1135–1146.

[24] Delling, D., Goldberg, A. V., and Werneck, R. F. F. Shortest paths in road networks: From practice to theory and back. *Information Technology 53*, 6 (2011), 294–301.

[25] Dementiev, R., and Sanders, P. Asynchronous parallel disk sorting. In *Proc. 15th SPAA* (2003), ACM, pp. 138–148.

[26] Derungs, J., Jacob, R., and Widmayer, P. Approximate shortest paths guided by a small index. *Algorithmica 57*, 4 (2010), 668–688.

[27] Gavoille, C., Peleg, D., Pérennes, S., and Raz, R. Distance labeling in graphs. *J. Algorithms 53*, 1 (2004), 85–112.

[28] Gudmundsson, J., Levcopoulos, C., Narasimhan, G., and Smid, M. H. M. Approximate distance oracles for geometric spanners. *ACM Transactions on Algorithms 4*, 1 (2008).

[29] Gupta, P., Goel, A., Lin, J., Sharma, A., Wang, D., and Zadeh, R. Wtf: The who to follow service at twitter. In *Proceedings of the 22Nd International Conference on World Wide Web* (2013), WWW '13, pp. 505–514.

[30] Har-Peled, S., and Mendel, M. Fast construction of nets in low-dimensional metrics and their applications. *SIAM J. Comput. 35*, 5 (2006), 1148–1184.

[31] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers 40*, 9 (September 1952), 1098–1101.

[32] Mehlhorn, K., and Meyer, U. External-memory Breadth-First Search with sublinear I/O. In *Proc. 10th ESA* (2002), vol. 2461 of *LNCS*, Springer, pp. 723–735.

[33] Mendel, M., and Naor, A. Ramsey partitions and proximity data structures. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (2006), IEEE Computer Society, pp. 109–118.

[34] Meyer, U. On dynamic Breadth-First Search in external-memory. In *25th Annual Symposium on Theoretical Aspects of Computer Science (STACS)* (2008), pp. 551–560.

[35] Meyer, U., and Osipov, V. Design and implementation of a practical I/O-efficient shortest paths algorithm. In *Proceedings of the annual conference on Algorithm Engineering and Experiments (ALENEX)* (2009), SIAM, pp. 85–96.

[36] Munagala, K., and Ranade, A. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA)* (1999), ACM-SIAM, pp. 687–694.

[37] Pajor, T., and Werneck, R. Personal communications, June - September 2014.

[38] Patrascu, M., and Roditty, L. Distance oracles beyond the thorup-zwick bound. In *51th Annual IEEE Symposium on Foundations of Computer Science, (FOCS)* (2010), IEEE Computer Society, pp. 815–823.

[39] Qiao, M., Cheng, H., and Yu, J. X. Querying shortest path distance with bounded errors in large graphs. In *23rd International Conference on Scientific and Statistical Database Management (SSDBM)* (2011), vol. 6809 of *Lecture Notes in Computer Science*, Springer, pp. 255–273.

[40] Sanders, P., and Schultes, D. Engineering highway hierarchies. *ACM Journal of Experimental Algorithmics 17*, 1 (2012).

[41] Sarna, A. D., Gollapudi, S., Najork, M., and Panigrahy, R. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the Third International Conference on Web Search and Web Data Mining (WSDM)* (2010), pp. 401–410.

[42] Sommer, C., Verbin, E., and Yu, W. Distance oracles for sparse graphs. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA* (2009), pp. 703–712.

[43] Thorup, M. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM 51*, 6 (2004), 993–1024.

[44] Thorup, M., and Zwick, U. Approximate distance oracles. *Journal of the ACM (JACM) 52*, 1 (2005), 1–24.

[45] Tretyakov, K., Armas-Cervantes, A., García-Bañuelos, L., Vilo, J., and Dumas, M. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proceedings of the 20th ACM Conference on Information and Knowledge Managemen (CIKM)* (2011), pp. 1785–1794.

[46] Ugander, J., and Backstrom, L. Balanced label propagation for partitioning massive graphs. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013* (2013), pp. 507–516.

[47] Zhao, X., Sala, A., Zheng, H., and Zhao, B. Y. Efficient shortest paths on massive social graphs, October 2011. (Invited Paper) Proceedings of 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom).

[48] Zhu, A. D., Xiao, X., Wang, S., and Lin, W. Efficient single-source shortest path and distance queries on large graphs. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2013)* (2013), ACM, pp. 998–1006.

A Proofs for our LCA labels

Lemma 1. *For any two nodes u, v in T, $d_T(u,v) = d_{T'}(u,v)$*

Proof. The proof follows by a simple induction on the number of nodes in T with degree greater than 2, that 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 $d_T(u,v) = d_{T'}(u,v)$. Lets assume the lemma 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, v and let $P'_{k-1}(u,v)$ be the path between u and v in T'_{k-1} of length $d_{T'_{k-1}}(u,v)$. By induction hypothesis we have $d_T(u,v) = d_{T'_{k-1}}(u,v)$ and we want to show that $d_T(u,v) = d_{T'_k}(u,v)$. There are three cases:

1. $P'_{k-1}(u,v)$ does not pass through $w, w_1, \ldots, w_k$. In this case, the transformation of node w does not affect this path and $P'_k(u,v) = P'_{k-1}(u,v)$ and therefore, $d_{T'_k}(u,v) = d_{T'_{k-1}}(u,v) = d_T(u,v)$

2. $P'_{k-1}(u,v)$ contains exactly one edge from the edges $\{w,w_1\}$, $\{w,w_2\}$, ..., $\{w,w_k\}$. Let's call this edge $\{w,w_i\}$ and lets assume w.l.o.g. that there is a path $P'_{k-1}(u,w)$ between u and w as well as a path $P'_{k-1}(w_i,v)$ between w_i and v without including this edge. With the transformation, there is a path between w and w_i of distance 1. Thus, there is a path $P'_k(u,v)$ in T'_k consisting of $P'_{k-1}(u,w)$ followed by the path between w and w_i and then $P'_{k-1}(w_i,v)$. The total length of this path is $d_{T'_{k-1}}(u,w) + 1 + d_{T'_{k-1}}(w_i,v) = d_{T'_{k-1}}(u,v)$. Since in a tree, there is only one simple path between any two nodes ($P'_k(u,v)$ of length $d_{T'_{k-1}}(u,v)$), $d_{T'_k}(u,v) = d_{T'_{k-1}}(u,v) = d_T(u,v)$.

3. $P'_{k-1}(u,v)$ consists of two edges from $\{w,w_1\}$, $\{w,w_2\}$, ..., $\{w,w_k\}$. Let's call these edges $\{w,w_i\}$ and $\{w,w_j\}$ and lets assume w.l.o.g. that there is a path $P'_{k-1}(u,w_i)$ between u and w_i as well as a path $P'_{k-1}(w_j,v)$ between w_j and v without including w. With the transformation, there is a path between w_i and w_j of distance 2 through their least common ancestor (which may not be w) in T'_k. Note that this is because the edges $\{w^1_l, w_i\}$ and $\{w^1_m, w_j\}$ introduced in the transformation have both length 1. Thus, there is a path $P'_k(u,v)$ in T'_k consisting of $P'_{k-1}(u,w_i)$ followed by the path between w_i and w_j and then $P'_{k-1}(w_j,v)$. The total length of this path is $d_{T'_{k-1}}(u,w_i) + 2 + d_{T'_{k-1}}(w_j,v) = d_{T'_{k-1}}(u,v)$. Since in a tree, there is only one simple path between any two nodes ($P'_k(u,v)$ of length $d_{T'_{k-1}}(u,v)$), $d_{T'_k}(u,v) = d_{T'_{k-1}}(u,v) = d_T(u,v)$.

□

Thus, 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 2. *A node 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 from u to r in T and let $w_k, w_{k-1}, \ldots, w_1$ be the weights of nodes $u_k, u_{k-1}, \ldots, u_1$ respectively. An edge $\{u_i, u_{i+1}\}$ in this path in T gets replaced by a path of at most $\lceil \log \frac{w_i}{w_{i+1}} \rceil$ hops in T'. Thus, the number of hops in the corresponding path in T' is at most $\sum_{i=1}^{k-1} \lceil \log \frac{w_i}{w_{i+1}} \rceil \leq l(u) + \sum_{i=1}^{k-1} \log \frac{w_i}{w_{i+1}} = l(u) + \log \frac{w_1}{w_k}$. Clearly, $\frac{w_1}{w_k} \leq n$ as the number of descendant at any node is at least 1 (as it includes itself) and at most n. Thus, the node u is at most $l(u) + \log n$ hops away from the root in T'. □

Corollary 1. *The height h' of T' is at most $h + \log n$*

Proof. Since by Lemma 2, all nodes are at most $l(u) + \log n \leq h + \log n$ hops away from the root in T', the height of T' is at most $h + \log n$. □

Theorem 1. *Given two nodes u, v in T, $d_T(u,v)$ can be computed in $O(1 + \frac{h}{\log n})$ instructions*

Proof. Given two nodes u, v in T, we show how to compute $d_{T'}(u, v)$ in $O(1)$ instructions. Since by Lemma 1, $d_T(u, v) = d_{T'}(u, v)$, this completes the proof.

We first compute $IN[u]$ and $IN[v]$ to get the inorder numbering of u, v in T'. The inorder number of the least common ancestor of u and v in T' is $IN[lca(u, v)] = 2^i$, where $i = \lfloor \log_2[IN[u] xor IN[v]] \rfloor$. We can then compute $d_{T'}(r, lca(u, v)) = DIST[IN[lca(u, v)]]$. The distance $d_{T'}(u, v)$ is then computed as $d_{T'}(r, u) + d_{T'}(r, v) - 2d_{T'}(r, lca(u, v))$. Since the inorder numbers are at most $\log n + h$ bits long, computing the least common ancestor takes $O(\frac{h}{\log n})$ instructions, which asymptotically dominates the query time. □

Theorem 2. *The space required by our distance encoding technique is $O(n)$ words (or $O(n \log n)$ bits), where $n := |V|$ for trees with height $O(\log n)$.*

Proof. For trees with height $O(\log n)$, we have shown that keeping the inorder number requires $O(\log n)$ bit, i.e., $O(1)$ word. The data structure consists of two hash-maps that contain as many elements (inorder numbers, distances and node indices) as the number of nodes in T'. Thus, we only need to show that the number of nodes n' in T' is linear in the number of nodes in T, i.e., $O(n)$.

This can be shown by a simple charging argument. For a node u with $k > 2$ children, we introduce $\lceil k/2 \rceil + \lceil k/4 \rceil + \ldots + 1 \leq k + \log_2 k \leq 2k$ new nodes in T'. These new nodes can be charged (2 each) to the k children of u in T. It is easy to see that each node in T gets charged at most once, when its parent is transformed. Thus, the total charge is at most $2n$ and the total number of new nodes introduced in T' is at most $2n$. The total number of nodes in T' is, thus, at most $3n$ (including the n nodes common to T). □