Fast and Accurate Estimation of Shortest Paths in Large Graphs

Andrey Gubichev Srikanta Bedathur Stephan Seufert* Gerhard Weikum
Max-Planck Institute for Informatics
Saarbrücken, Germany
gubichev@gmail.com, {bedathur, sseufert, weikum}@mpi-inf.mpg.de

ABSTRACT

Computing shortest paths between two given nodes is a fundamental operation over graphs, but known to be nontrivial over large disk-resident instances of graph data. While a number of techniques exist for answering reachability queries and approximating node distances efficiently, determining actual shortest paths (i. e. the sequence of nodes involved) is often neglected. However, in applications arising in massive online social networks, biological networks, and knowledge graphs it is often essential to find out many, if not all, shortest paths between two given nodes.

In this paper, we address this problem and present a scalable sketch-based index structure that not only supports estimation of node distances, but also computes corresponding shortest paths themselves. Generating the actual path information allows for further improvements to the estimation accuracy of distances (and paths), leading to near-exact shortest-path approximations in real world graphs.

We evaluate our techniques – implemented within a fully functional RDF graph database system – over large real-world social and biological networks of sizes ranging from tens of thousand to millions of nodes and edges. Experiments on several datasets show that we can achieve query response times providing several orders of magnitude speedup over traditional path computations while keeping the estimation errors between 0% and 1% on average.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and Networks*; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

Graph Databases, Shortest Paths, Social Networks

*corresponding author

CIKM'10, October 26–30, 2010, Toronto, Ontario, Canada.

1. INTRODUCTION

Graphs are routinely used in the modern digital world in a number of settings, such as online social networks (like LinkedIn, Facebook, MySpace), synthesized entity-relationships in large-scale knowledge repositories [1, 25], biological interaction models [10, 12, 13], transportation networks [20], the massive hyperlink graph between documents of the World Wide Web, XML data, and many more.

Due to the ever increasing size of the graphs of interest, many seemingly straightforward operations become challenging. In this paper, we turn our attention to the computation of shortest paths between any two nodes in the graph, a problem with long algorithmic history. This operation forms a building block for many mining tasks, and is also an increasingly important application in itself over instances such as online social networks. Consider the following two application scenarios for shortest path queries:

- **Social networks** such as LinkedIn enable professional networking among individuals. A person interested in reaching out to the hiring manager of his favourite future employer would seek a shortest path to reach that person, starting from his friends.
- **Biological (metabolic) networks** (such as the Biochemical Network Database [13]) model the complex chemical processes within an organism. A biologist may be interested in identifying biotransformation paths between two target metabolites to help in designing experiments in the wet lab.

Similar applications arise for almost every instance of graph data, which includes the commonly studied problem of finding a shortest route between two points in road networks [28]. In many cases, the graphs of interest comprise millions of nodes and edges. Thus, for scalability reasons, each of the shortest path query instances has to be answered as fast as possible while minimizing the consumption of resources such as memory and processor cycles.

1.1 Problem Difficulty

What makes the shortest path computation particularly hard on large graphs? Dijkstra's algorithm, the classical technique to compute the shortest path between two nodes in a graph has the asymptotic runtime complexity of $\mathcal{O}\big(m + n\log(n)\big)$, where n is the number of nodes and m is the number of edges [5]. On one of the benchmark datasets that we use in this paper – the Orkut social network comprising about 3 million nodes and 220 million edges – a straighforward implementation of Dijkstra's algorithm takes more

than 500 seconds on average. This is way too slow for most applications. The reason for this is that Dijkstra's algorithm has to construct and maintain shortest paths to all nodes in the graph whose distance to the source node is smaller than the distance from the source node to the destination node. Consequently, the memory consumption of Dijkstra's algorithm is very high, requiring to maintain a number of 2.5 million nodes in the heap for the Orkut dataset, which is prohibitively expensive for simultaneous execution of many queries.

The naïve alternative of precomputing all-pairs shortest paths distances and maintaining them on disk for quick lookups is practically infeasible, requiring $\mathcal{O}(n^2)$ space. For general graphs, it has been shown that constant query time for exact shortest path distance queries is only achievable with super-linear space requirements during preprocessing [26]. Relaxing the exactness requirement, a number of *distance oracles* have appeared which aim at providing a highly accurate estimate of the node distances [4, 22, 24, 26, 29]. The main application for these approaches occur in *geographic information systems* (GIS) or, specifically, in route planning over transportation networks. Techniques developed in this domain exploit in a crucial way special properties of transportation networks such as their near planarity, low node-degree and the presence of a hierarchy based on the importance of roads [2, 3]. These properties also help in devising faster variants of Dijkstra's online algorithm by combining it with A*-search and other goal-oriented pruning strategies [7].

On the other hand, the social networks that we consider in this work do not exhibit the same properties as road networks. It is well known that social networks contain many high degree nodes, are nowhere close to planar, and typically have no hierarchical structures that can be exploited for improving shortest path queries. Potamias et al. [19] make the important observation that even a 2-approximation, which is considered highly competitive for general graphs, is insufficient in the case of social networks as the distances are already very small. Further, none of the prior work has explicitly addressed the problem of providing the shortest paths themselves, not just the node distances, as we do in this paper.

We also briefly mention here that database research has considered queries over different forms of graph databases apart from road networks, such as XML graphs and biological networks [8, 9, 23, 27]. The focus, however, has been primarily on answering *reachability queries*, not in computing the shortest path distance or the shortest path itself.

1.2 Contributions

We build upon the recently proposed sketch-based framework [6], which in turn is based on the classical landmark-based approach used in distance oracles [26]. The goal is to precompute and store a $\mathcal{O}(n)$ sized *sketch* of the graph so that any distance query can be answered approximately but with high accuracy. The prior work considered maintaining only the distance information as part of the sketch. However, we observe that for most real-world graph data (apart from road networks), like social networks, the path lengths are small enough to be considered almost constant [17], and thus propose to store the complete path information (i. e. the information about constituent nodes and edges) as part of the sketch. Based on the availability of such *path-sketches* (as opposed to the distance-sketches in [6]) we develop a set of lightweight algorithms that can approximate shortest paths between any two nodes with at most 1% error. Additionally, we can also generate a set of paths (i. e. the identity of nodes in the path) that correspond to the estimated shortest path distance with no additional overhead – a feature hitherto not available.

In summary, contributions made in this paper are as follows:

1. We introduce *path-sketches*, which can be effectively used in large graphs with small diameters (e.g., almost all online social networks). The path-sketches maintain the complete path information between every node and a selected set of *landmark nodes*, computed as part of a graph preprocessing step.
2. We develop a set of lightweight, yet highly effective, techniques that use path-sketches to significantly improve the quality of shortest path estimations.
3. Along with estimates of shortest path distances, we show how to generate corresponding instances of shortest paths themselves with no computational overhead – a feature not found so far in most distance estimating techniques. In fact, our algorithm generates a queue of paths in increasing order of their lengths, an important need in many applications over social and biological networks.
4. Finally, we implement all the proposed methods in a fully functional large-scale graph processing engine, RDF-3X[18], and evaluate against a number of real-world large-scale graphs.

In most of the datasets we used, the estimates are almost always exact, and generate a number of alternative shortest paths between a given pair nodes.

1.3 Outline

The remainder of this paper is organized as follows: In the next section we explain the previously proposed distance oracle algorithm devised by Das Sarma et al. [6] (in the following referred to as the *sketch* algorithm) and show how simple yet powerful modifications to it yield shortest path estimates of higher quality. Subsequently, we describe a new algorithm that uses the data obtained in the precomputation step of the sketch algorithm and returns paths whose accuracy beats the previous approaches by an order of magnitude at the expense of only a marginal time overhead. In section 3 we describe the implementation details of our algorithms within the graph database system RDF-3X. Section 4 comprises a comprehensive experimental evaluation that shows the practicability of our algorithms both in terms of query response time as well as in the approximation quality of the returned results. We conclude the paper with a review of related work and finally identify possible future research directions.

2. ALGORITHM DESCRIPTION

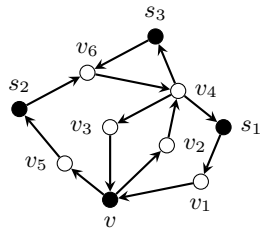
In this section we explain our algorithms for shortest path approximation in detail.

2.1 Preliminaries

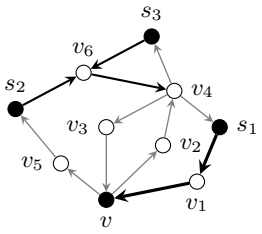
Let $G = (V, E)$ denote a directed graph with vertex set V and edge set E.

Paths and Distances. A path p of length $l \in \mathbb{N}$ in the

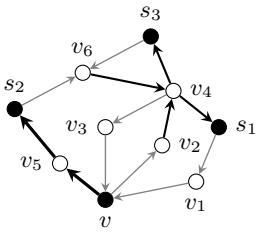
Figure 1: Precomputation Example



(a) Vertex v and seed set $S = \{s_1, s_2, s_3\}$



(b) BFS from S in forward direction reaches v



(c) BFS from S in backward direction reaches v

Direction	Landmark	Path
$\leftarrow$	s_1	(s_1, v_1, v)
$\rightarrow$	s_2	(v, v_5, s_2)
$\vdots$	$\vdots$	$\vdots$

(d) Resulting entries in Sketch(v)

graph is an ordered sequence of $l+1$ vertices, such that there exists, for every vertex in the sequence, an edge to its subsequent vertex, except the last one:

$$p = (v_1, v_2, \ldots, v_{l+1}) \quad \text{with} \quad v_i \in V, 1 \leq i \leq l+1, \tag{1}$$

$$(v_i, v_{i+1}) \in E,\ 1 \leq i < l. \tag{2}$$

For a node $v \in V$ of the graph we denote by $\mathcal{S}(v)$ the set of the successors of v in G, that is the set of vertices $w \in V$ with $(v, w) \in E$. Thus, we can express requirement (2) equivalently as $v_{i+1} \in \mathcal{S}(v_i),\ 1 \leq i < l$.
We write $|p| = l$ to denote the length of the path p. For vertices $u, v \in V$, let $\mathcal{P}(u, v)$ be the set of all paths that start in u and end in v. The distance from u to v, denoted by $\text{dist}(u, v)$, is the number of edges in the shortest such path – or infinity if v is not reachable from u:

$$\text{dist}(u, v) := \begin{cases} \arg\min_{p \in \mathcal{P}(u,v)} |p| & \text{if } \mathcal{P}(u, v) \neq \emptyset, \\ \infty & \text{else.} \end{cases} \tag{3}$$

Path Approximation. Given two vertices $u, v \in V$, let p denote a shortest path (note that there could be many) from u to v, that is, a path starting in u and ending in v with length $|p| = \text{dist}(u, v)$. Furthermore, let q be an arbitrary path from u to v. By regarding q as an approximation of the shortest path p, we can define the approximation error of this path as

$$\text{error}(q) := \frac{|q| - |p|}{|p|} = \frac{|q| - \text{dist}(u, v)}{\text{dist}(u, v)} \in [0, \infty]. \tag{4}$$

Path Concatenation. Let $p = (u_1, u_2, \ldots, u_{l_1}, u_{l_1+1})$ and $q = (v_1, v_2, \ldots, v_{l_2+1})$ denote paths of lenghts l_1 and l_2 respectively. Suppose $u_{l_1+1} = v_1$, that is, the last node in path p equals the first node in path q. Then, we can create new path, denoted by $p \circ q$, of length $l_1 + l_2$ by concatenating the paths p and q:

$$\begin{aligned} p \circ q &= (u_1, u_2, \ldots, u_{l_1}, u_{l_1+1}) \circ (v_1, v_2, \ldots, v_{l_2+1}) \\ &:= (u_1, \ldots, u_{l_1}, v_1, \ldots, v_{l-2+1}). \end{aligned} \tag{5}$$

2.2 Sketch Algorithm

The sketch algorithm [6] approximates the shortest path distance between two given nodes in general graphs using a landmark-based approach. In order to answer a distance query for a pair of nodes (s, d) in real time, the algorithm employs a two-staged approach: a precomputation step to generate sketches (distances from all vertices to so-called landmark nodes) beforehand, and an approximation step that uses this precomputed data to provide a very fast approximation of the node distance at query time. It works by combining the two distances $\text{dist}(s, l)$, $\text{dist}(l, d)$ of the query nodes to/from a selected landmark node l into the approximated distance

$$\tilde{d}(s, d) \leq \text{dist}(s, l) + \text{dist}(l, d).$$

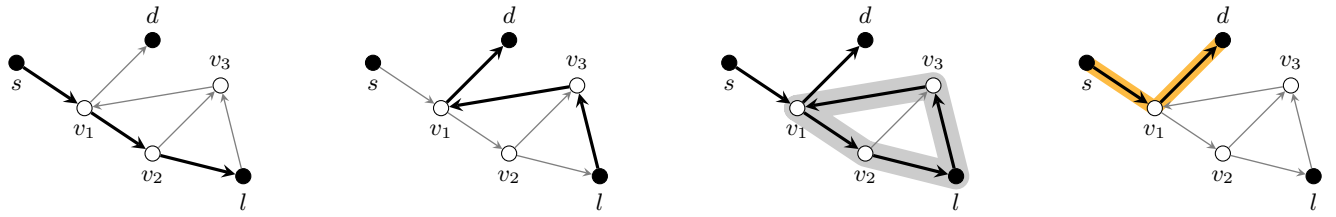
Therefore, in the original paper [6], the authors suggest to store for every node v the distances $\text{dist}(v, \cdot)$ and $\text{dist}(\cdot, v)$ from (to) the node to (from) certain landmark nodes as the result of the precomputation. This set of node-landmark distances is called *sketch* of a node.
Instead of keeping just the distances, we modify the precomputation step to store the distances along with the actual paths. The diameters of social networks are usually small [17], so the paths are expected to be relatively short. Therefore, the storage overhead of maintaining full path information as part of the sketch is not substantial – our experiments show that it is no more than twice the sketch with only distance information. Obviously, we do not incur any additional computational overhead during the precomputation step, since we require no more information than generated by the breadth-first search and reverse breadth-first search steps of sketch computation.
Also note that while the original algorithm returns an estimate of the distance between the query nodes, our algorithm returns more than just one approximate path between the query nodes, namely a queue of such paths (sorted in ascending order by path length). By providing many candidate paths, this modification could prove useful in scenarios where – for example – constraints on certain nodes/edges must be satisfied.
In this section we explain these two building blocks of the (modified) sketch algorithm in detail:

2.2.1 Precomputation

The precomputation step involves sampling some sets of nodes, computing for every node in the graph a shortest path to and from a member of this set and storing the thus obtained set of paths on external memory. These paths will be used in the approximation step later. The preprocessing, illustrated in Figure 1, works as follows:

1. **Seed Set Sampling**
   Let $r := \lfloor \log(n) \rfloor$ where $n = |V|$. We uniformly sample $r+1$ sets of nodes (called seed sets) of sizes $1, 2, 2^2, \ldots, 2^r$ respectively. The selected sets are denoted by $S_0, S_1, \ldots, S_r$.

Figure 2: Cycle Elimination Example



(a) Path from s to l **(b)** Path from l to d **(c)** Concatenated path from s to d through l **(d)** Final path from s to d after cycle elimination

2. **Shortest Path Computation**
   For each of the sampled seed sets S_i and every node $v \in V$ we compute a shortest path $p_{S_i \to v}$ that connects any member of the seed set to v (note that there could be more than one such path). We use breadth-first expansion from S_i and build the complete shortest path tree. For every node v, we thus obtain the closest seed node, denoted by l_1. Likewise, we compute a shortest path $p_{v \to S_i}$ that connects v to the seed set, using breadth-first expansion from S_i on reversed edges, terminating at the first seed node, denoted by l_2. The nodes l_1, l_2 are called *landmarks* of S_i for the vertex v.

This precomputation routine – that is, seed set sampling and shortest path computation – is repeated k times, thereby generating for each vertex $v \in V$ a number of $2rk$ landmarks and paths (at costs of the same number of BFS expansions from the seed sets). Note that the set of selected landmarks might be different for every node. The data (sketch) gathered for node v, consisting of the $2rk$ landmarks and paths, is denoted by $\mathsf{Sketch}(v)$. As a result of the precomputation step, we store the sketches of all nodes on disk.

2.2.2 *Shortest Path Approximation*

In the second stage, the algorithm receives a pair of nodes, (s, d), as input. The goal is to compute, in real time, a path $p_{s \to d}$ from s to d that provides a good approximation of the shortest path, that is, a path with small error$(p_{s \to d})$. The sketch algorithm, presented in Algorithm 1, performs the following steps to generate such a path:

1. Load the sketches of nodes s and d from disk
2. Let L be the set of common landmarks in the sketches. Note that for undirected graphs we can guarantee (for nodes in the same component) that there exists at least one such landmark, because the sketches of both s and d contain a path from (respectively to) the only member of the singleton seed set S_0. In directed graphs this guarantee can only be given for nodes contained in the same strongly connected component.
3. For each common landmark $l \in L$, let $p_{s \to l}$ be the path from s to l and $p_{l \to d}$ the path from l to d. Construct (by concatenation) the path $p_{s \to d} := p_{s \to l} \circ p_{l \to d}$ from s to d through l, and add it to a priority queue (sorted in ascending order by path length).
4. Return the priority queue of paths obtained in step 3.

Algorithm 1: SKETCH(s, d)

Input: $s, d \in V$
Result: Q – priority queue of paths from s to d, ordered by path length

```
1 begin
2     Load sketches Sketch(s), Sketch(d) from disk
3     L ← common landmarks of Sketch(s) and Sketch(d)
4     foreach l ∈ L do
5         p ← path from s to d through l
6         Add p to queue Q
7     return Q
```

2.3 Improving the Accuracy

In this section we explain our modifications to the original sketch algorithm to obtain better approximations. As a first step, we eliminate cycles in the paths found by the sketch algorithm and as a second improvement we try exploit existing shortcuts within the paths. While these two modifications to the original algorithm are simple, they provide considerable improvements in terms of the approximation quality, as we will show in the experimental evaluation.

2.3.1 *Cycle Elimination*

The paths returned by Algorithm 1 approximate the shortest path for the two query nodes and thus might be suboptimal, i. e. longer than the true shortest path. Some of the returned paths can however be easily improved, because they contain cycles. Consider the example shown in Figure 2: Suppose $l \in V$ is a common landmark for the nodes $s, d \in V$ and the sketches $\mathsf{Sketch}(s)$ and $\mathsf{Sketch}(d)$ contain the paths $p_{s \to l} = (s, v_1, v_2, l)$ and $p_{l \to d} = (l, v_3, v_1, d)$ respectively. Then, the queue Q returned by Algorithm 1 contains the path

$$p_{s \to l} \circ p_{l \to d} = (s, v_1, v_2, l, v_3, v_1, d).$$

Obviously, we can obtain a shorter path by removing the cycle (v_1, v_2, l, v_3, v_1), thus obtaining the path (s, v_1, d). The modified sketch algorithm, named SKETCHCE, is described in Algorithm 2.

For a path of length l, our naïve cycle elimination routine performs at most $\mathcal{O}(l^2)$ node comparisons. In theory, we could make use of more advanced cycle elimination/detection approaches [11]. However, the diameters and thus the shortest paths in social networks are usually bounded by a small constant [17], thus we can assume constant time complexity for the cycle elimination routine on a single path. Furthermore, the number of paths in Q is bounded by the choice

Algorithm 2: SketchCE(s, d)

Input: $s, d \in V$
Result: Q – priority queue of paths from s to d, ordered by path length

```
1  begin
2      Q ← Sketch(s, d)
3      foreach p = (p_1, p_2, ..., p_l) ∈ Q do
4          for i = 1 to l − 1 do
5              for j = 0 to l − i − 1 do
6                  if p_i = p_{l−j} then
7                      Q ← Q ∪ {(p_1, ..., p_i, p_{l−j+1}, ..., p_l)}
8                      break                          ▷ continue in line 3
9      return Q
```

Algorithm 3: SketchCESC(s, d)

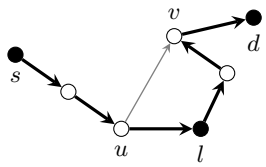
Input: $s, d \in V$
Result: Q – priority queue of paths from s to d, ordered by path length

```
1  begin
2      Q ← SketchCE(s, d)
3      foreach p = (p_1, p_2, ..., p_{i−1}, p_i, p_{i+1}, ..., p_l) ∈ Q do
                                          ▷ p_i denotes the landmark in path p
4          for j = 1 to i − 1 do
5              S ← set of successors of p_j
6              for k = 0 to l − i + 1 do
7                  if p_{l−k} ∈ S then
8                      Q ← Q ∪ {(p_1, ..., p_j, p_{l−k}, ..., p_l)}
9                      break                          ▷ continue in line 3
10     return Q
```

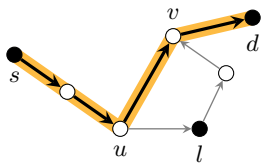
of precomputation rounds, k, and the number of seed sets, r. We eventually obtain the upper bound $|Q| \leq 2rk$ for the queue size because for each vertex we store two paths for every seed set (forward and backward paths). With the standard choice $r = \log(n)$ this leads to a time overhead of $\mathcal{O}\big(k \log(n)\big)$ for the cycle elimination enhancement. We can keep the queue Q in main memory, as a result the increase in running time with respect to the basic algorithm (Algorithm 1) is negligible (as we will show in the experimental evaluation).

2.3.2 *Shortcutting*

The second modification we propose is path shortcutting: Suppose the queue Q returned by the algorithm contains a path $p_{s \to l \to d}$ from s to d via a landmark l. Two nodes u, v in the path might actually have a closer connection than the one contained in the respective subpath of $p_{s \to l \to d}$. Consider the example depicted below: While the nodes u and v are connected by a subpath of $p_{s \to l \to d}$ of length 3, the original graph contains the edge (u, v). We can then easily substitute this subpath by the single edge (u, v). Note that in all cases that allow for this shortcutting optimization, the landmark l will be located on the subpath from u to v.



(a) Path from s to d. The original graph contains the arc (u, v)



(b) Path from s to d after shortcutting

In order to find out whether any two nodes in a given path $p_{s \to l \to d} = p_{s \to l} \circ p_{l \to d}$ are neighbors, we start at the first vertex, s, and load the list $\mathcal{S}(s)$ of its successors in the original graph. Then we check if any of the successors of s is contained in the path $p_{l \to d}$ from the landmark l to the destination vertex d. If this is the case, we can substitute the subpath from s to this node by the single edge. If no successor of s is contained in the path from l to d, we proceed to the second node in the path $p_{s \to l}$, load the set of its successors and repeat the procedure. We can terminate this routine if we either

- find a shortcut or
- arrive at the last vertex of the path $p_{s \to l}$, the landmark node l. In this case, the original path $p_{s \to l \to d}$ cannot be improved by shortcutting, because the path $p_{l \to d}$ from l to d is already guaranteed to be a shortest path (obtained using BFS in the preprocessing step).

The complete algorithm (cycle elimination + shortcutting), called SketchCESC, is depicted in Algorithm 3.

2.4 Tree Algorithm

In this section we describe our third contribution, a new algorithm for shortest path approximation that also utilizes the precomputed sketches.

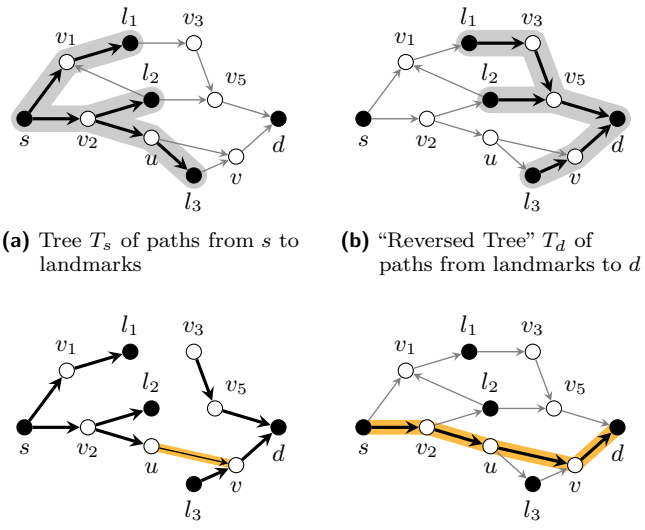
The sketch $\mathsf{Sketch}(v)$ of a node v contains two sets of paths: (1) the set of paths connecting v to landmarks (called *forward-directed paths*) and (2) the set of paths connecting landmarks to v (called *backward-directed paths*). In the undirected setting, both sets would correspond to trees having landmarks as leaves and v as a root. Every inner node of each tree corresponds to a vertex contained in one of the paths in the sketch. In the directed setting, only the forward-directed part of the sketch (from v to landmarks) forms a tree, while the backward-directed paths yield a tree with "reversed edges" (see Figures 3a+b).

Our new algorithm, named TreeSketch, takes the two query nodes s, d as input, loads the sketches $\mathsf{Sketch}(s), \mathsf{Sketch}(d)$ from disk and constructs the tree T_s, rooted at s, that contains all the forward paths stored in $\mathsf{Sketch}(s)$. Likewise, the "reversed tree" T_d, rooted at d, containing all the backward directed paths from the landmarks to d is being created from $\mathsf{Sketch}(d)$. Then, the algorithm starts two breadth-first-expansion on the trees simultaneously: $\text{BFS}(T_s, s)$ from s in T_s and $\text{RBFS}(T_d, d)$ (BFS on reversed edges) from d in T_d. At any point of time during execution, let $\mathcal{V}_{\text{Bfs}}$ and $\mathcal{V}_{\text{RBfs}}$ denote the sets of visited nodes during the respective BFS runs.

For every vertex $u \in \mathcal{V}_{\text{Bfs}}$ encountered during $\text{BFS}(T_s, s)$, the algorithm loads the list $\mathcal{S}(v)$ of its successors in the original graph. Then, it checks whether any of the vertices discovered during $\text{RBFS}(T_d, d)$ is contained in the list $\mathcal{S}(u)$. If such a vertex $v \in \mathcal{S}(u) \cap \mathcal{V}_{\text{RBfs}}$ exists (see Figure 3c), we have found a path p from s to d, given by

$$p = p_{s \to u} \circ (u, v) \circ p_{v \to d},$$

Figure 3: Tree Algorithm Example



(a) Tree T_s of paths from s to landmarks

(b) "Reversed Tree" T_d of paths from landmarks to d

(c) Arc discovered in line 13 of algorithm TreeSketch

(d) Obtained path from s to d

where $p_{s\to u}$ and $p_{v\to d}$ denote the paths from s to u in T_s and from v to d in T_d respectively (Figure 3d). An equivalent procedure is executed for every vertex encountered during reverse BFS from d.

We continue this procedure of BFS expansions and successor list checks, adding paths discovered along the way to the queue Q. Let l_{shortest} denote the length of the shortest path in Q. The algorithm terminates if there is no further chance to find a path that is shorter than the current shortest path in Q. This is the case when the sum of depths of both BFS runs exceeds l_{shortest}.

The complete algorithm is depicted in Algorithm 4.

Algorithm 4: TreeSketch(s, d)

Input: $s, d \in V$
Result: Q – priority queue of paths from s to d, ordered by path length

1 **begin**
2 Load Sketch(s), Sketch(d) from disk
3 $T_s \leftarrow$ tree of paths from s ▷ taken from Sketch(s)
4 $T_d \leftarrow$ tree of paths to d ▷ taken from Sketch(d)
5 $Q \leftarrow \emptyset$
6 $l_{\text{shortest}} \leftarrow \infty$
7 $\mathcal{V}_{\text{Bfs}} \leftarrow \emptyset$
8 $\mathcal{V}_{\text{RBfs}} \leftarrow \emptyset$
9 **foreach** $u \in \text{Bfs}(T_s, s)$ **and** $v \in \text{RBfs}(T_d, d)$ **do**
10 $\mathcal{V}_{\text{Bfs}} \leftarrow \mathcal{V}_{\text{Bfs}} \cup \{u\}$
11 $p_{v\to d} \leftarrow$ path from v to d in T_d
12 **foreach** $x \in \mathcal{V}_{\text{Bfs}}$ **do** ▷ iteration in order of visits
13 **if** $v \in \mathcal{S}(x)$ **then** ▷ $\mathcal{S}(x)$ is set of successors of x in G
14 $p \leftarrow p_{s\to x} \circ (x, v) \circ p_{v\to d}$
15 $Q \leftarrow Q \cup \{p\}$
16 $l_{\text{shortest}} \leftarrow \min\{l_{\text{shortest}}, |p|\}$
17 $\mathcal{V}_{\text{RBfs}} \leftarrow \mathcal{V}_{\text{RBfs}} \cup \{v\}$
18 $p_{s\to u} \leftarrow$ path from s to u in T_s
19 **foreach** $x \in \mathcal{V}_{\text{RBfs}}$ **do** ▷ iteration in order of visits
20 **if** $x \in \mathcal{S}(u)$ **then** ▷ $\mathcal{S}(u)$ is set of successors of $u \in G$
21 $p \leftarrow p_{s\to u} \circ (u, x) \circ p_{x\to d}$
22 $Q \leftarrow Q \cup \{p\}$
23 $l_{\text{shortest}} \leftarrow \min\{l_{\text{shortest}}, |p|\}$
24 **if** $\text{dist}(s, u) + \text{dist}(v, d) \geq l_{\text{shortest}}$ **then break**
25 **return** Q

3. IMPLEMENTATION

We implemented all methods – Dijkstra's online shortest-path algorithms as well as the sketch-based techniques – within RDF-3X [18], a recently proposed high-performance database system for storing and querying large RDF graph repositories. Before presenting the implementation details, we briefly give a background on RDF-3X, focusing on its ability to store large-graphs highly compressed and, at the same time, easy to query from. Note that we omit details of query processing and optimization strategies for queries within RDF-3X, as they are not relevant in our setting.

3.1 RDF-3X: RDF Graph Processor

RDF-3X is a fully functional, high-performance storage engine and query processor designed particularly for storing – as the name suggests – RDF and querying using SPARQL. It maintains the whole graph essentially as a *huge triples table*, in contrast to the recently favored property-table approach. Thus, an edge $e = (u, v)$ is represented as a triple $\langle u, e, v\rangle$, or in RDF terminology as $\langle Subject(S), Predicate(P), Object(O)\rangle$. In the first step, each of these values are mapped to a simple integer, and this mapping is maintained in a global *Dictionary*. Then, all six possible permutations of S, P and O are maintained in six separate clustered B$^+$-tree indexes. In other words, an index exists for each of (SPO, SOP, OSP, OPS, PSO, POS) orderings. Each index can be significantly compressed by the use of *delta-coding* of triples, a generalization of similar compression used in inverted list indexes to id triples. Further, RDF-3X also builds 6 more indexes in order to support analytic queries efficiently. Counterintuitively, the overall database size containing 12 different clustered B$^+$-tree indexes with all the compressions mentioned above works out to be smaller than the original data in triple format.

In our implementation, we store graphs in RDF-3X edge-wise with each edge represented as a triple $\langle s, e, t\rangle$. We do not restrict RDF-3X from building all the 12 indexes automatically, although we do not use all of them in this work – in fact, we exploit only SPO and OPS ordered indexes.

Dijkstra's Algorithm. Implementing Dijkstra's algorithm over RDF-3X basically involves opening a scan on the SPO index to determine, for each node visited during the execution of the algorithm, all the successor nodes. Note that we get reverse Dijkstra's algorithm, needed during computation of sketches, essentially for "free" by simply opening the scan on the OPS index, and letting the algorithm run. The priority-queue required during Dijkstra's algorithm is maintained in memory using an implementation available in GNU-C++ STL.

3.2 Sketch Implementation

For simplicity, we store the sketches also as RDF triples in a separate database under RDF-3X. Since sketches (both

distance- and path-sketches) are associated with oriented paths, we format them as follows:

$$\langle v_i \rangle \langle [\texttt{to}|\texttt{from}] \rangle \langle l_{ij} : d_{ij} \rangle$$

where, v_i is the id of the source node, `to` and `from` are string literals indicating the orientation of the path, l_{ij} is the landmark for node v_i from the seed-set S_j, and d_{ij} is the corresponding shortest distance to the landmark. Similarly, path-sketches are also stored as triples of the form:

$$\langle v_i \rangle \langle [\texttt{to}|\texttt{from}] \rangle \langle l_{ij} : p_{ij} \rangle$$

where p_{ij} refers to the sequence of node-ids in the path between the node v_i and the landmark node l_{ij}.
It should be noted again that RDF-3X builds 12 indexes over this triple database of sketches, but we require only one of these indexes for our shortest path estimation algorithms, namely, the B^+-tree over the permutation SPO. Thus, the disk space consumption we provide later in experiments section can be further reduced significantly, although the relative sizes of sketches remains the same. In all our experiments we used the default setting of $k = 2$ precomputation rounds.

4. EXPERIMENTAL EVALUATION

In this section we provide an experimental evaluation of our algorithms. First, we give an overview of the datasets used. Afterwards, in subsection 4.2, we describe the generation of test instances used in the subsequent evaluation. The approximation quality of the different approaches is assessed in subsection 4.3.1, query running time measurements are carried out in subsection 4.3.2. In addition, we briefly consider the question of path diversity in subsection 4.3.3. We conclude with subsection 4.3.4, where we quantify the space and time requirements for sketch precomputation.

4.1 Datasets

We prove the practicability of our approach by a number of experiments on the following real-world networks:

Slashdot Zoo — a network of users of the technology-news website Slashdot, introduced in 2002. In this network, users can tag each other via friend and foe links. The data was crawled in 2008 [16].

Google Webgraph — a fraction of the webgraph released by Google for the Google Programming Contest in 2002. This network has the largest diameter of the datasets we consider [16].

YouTube — a 2007 crawl of the social network consisting of roughly 1 million users of the video-sharing community YouTube [17],

Flickr — a social network of about 1.7 million users of the photo-sharing website Flickr, crawled in 2007 [17],

WikiTalk — network of Wikipedia members commenting on each other's Talk pages. An edge exists from user a to user b if a has commented on b's user page. This network is not very well connected, only about 5% of the users belong to the largest strongly connected component [14],

Twitter — parts of the social network of the microblogging community Twitter, crawled in 2009 [21], and

Table 1: Used Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\overline{\delta}$	$\lvert\mathcal{S}\rvert/\lvert V\rvert$	$d_{0.9}$
Slashdot	77,360	905,468	23.4	90.9 %	5.59
Google	875,713	5,105,039	11.7	49.6 %	9.02
YouTube	1,138,499	4,945,382	8.7	44.7 %	7.14
Flickr	1,715,255	22,613,981	26.4	69.5 %	7.32
WikiTalk	2,394,385	5,021,410	4.2	4.6 %	4.98
Twitter	2,408,534	48,776,888	40.5	57.5 %	5.52
Orkut	3,072,441	223,534,301	145.5	97.5 %	5.70

Datasets with no. of vertices $|V|$, no. of edges $|E|$, average degree $\overline{\delta}$, percentage of nodes in the largest strongly connected component $\mathcal{S}$ and effective diameter $d_{0.9}$ [15].

Orkut — the "pure" social network Orkut, containing more than 3 million users. This network exhibits a very high average node degree [17].

The networks and their properties are listed in table 1.

4.2 Methodology

In order to evaluate the approximation quality and running times of our algorithms, we use a set of test triples of the form

$$\big(x, y, \mathrm{dist}(x, y)\big), \tag{6}$$

consisting of a pair of nodes $x, y \in V$ and the actual distance $\mathrm{dist}(x, y)$ (length of shortest path) of these nodes in the graph. We generate these triples by uniformly sampling one hundred vertices and computing shortest path trees (forward and backward direction) for each vertex, using Dijkstra's algorithm [5]. As output we obtain for every sampled vertex v one tree connecting the vertex to every other reachable node and one "reversed" tree, connecting every node for which a path to the sampled vertex exists to v.
As a result we obtain a set of triples of the structure shown in equation (6). Then, we group these triples into categories corresponding to the distance $\mathrm{dist}(x, y)$. From every such category, we sample at most 50 triples (tests) as our test set. The actual number of tests varies for every network because both the number of groups as well as the number of contained triples might be different.

4.3 Results

4.3.1 Approximation Quality

In order to assess the approximation quality of the paths generated by the different algorithms, we run for every triple $\big(s, d, \mathrm{dist}(s, d)\big)$ contained in the test set a shortest path query for all 4 proposed algorithms: Sketch, SketchCE, SketchCESC, and TreeSketch. For every shortest path query (s, d), we compare the length l_{shortest} of the shortest path $p_{s\to d}$ in the returned queue with the true node distance $\mathrm{dist}(s, d)$ specified in the test triple. Then, we obtain the approximation error, $\mathrm{error}(p_{s\to d})$, for the path:

$$\mathrm{error}(p_{s\to d}) = \frac{l_{\mathsf{shortest}} - \mathrm{dist}(s, d)}{\mathrm{dist}(s, d)} \in [0, \infty].$$

For every algorithm we record the average approximation error over all the test triples. The obtained error values are provided in table 2 and plotted in Figure 4, using a logarithmic scale to display the error values.
As these results clearly demonstrate, the approximation quality of the algorithms we propose in this paper turns out to be superior to the previously proposed method (denoted as

Figure 4: Average Approximation Error error(·)

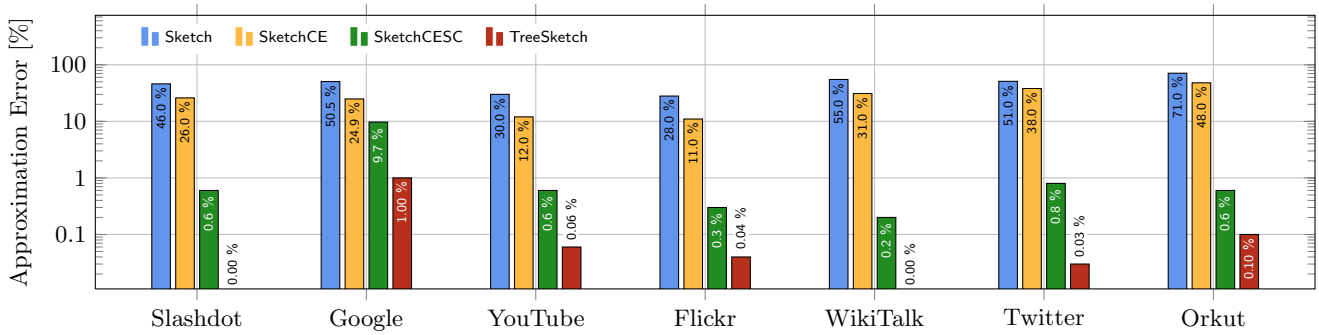


Table 2: Approximation Quality and Running Times

	Average Approximation Error					Average Running Times				
Dataset	Tests	Sketch	SketchCE	SketchCESC	TreeSketch	Sketch	SketchCE	SketchCESC	TreeSketch	Dijkstra (Queue)
Slashdot	910	46.0 %	26.0 %	0.6 %	0.00 %	140 ms	140 ms	198 ms	193 ms	4 s (46K)
Google	3,526	50.5 %	24.9 %	9.7 %	1.00 %	932 ms	932 ms	1,270 ms	1,339 ms	35 s (157K)
YouTube	758	30.0 %	12.0 %	0.6 %	0.06 %	872 ms	872 ms	1,282 ms	1,318 ms	48 s (380K)
Flickr	934	28.0 %	11.0 %	0.3 %	0.04 %	1,217 ms	1,217 ms	2,177 ms	1,951 ms	73 s (696K)
WikiTalk	780	55.0 %	31.0 %	0.2 %	0.00 %	703 ms	703 ms	1,400 ms	1,680 ms	101 s (2M)
Twitter	897	51.0 %	38.0 %	0.8 %	0.03 %	1,932 ms	1,932 ms	3,900 ms	4,000 ms	119 s (1.1M)
Orkut	385	71.0 %	48.0 %	0.6 %	0.10 %	1,090 ms	1,090 ms	2,595 ms	2,751 ms	503 s (2.5M)

SKETCH). For all datasets under consideration, we are able to return a shortest path for the query nodes with an average estimation error of 1% in the worst case, while providing exact solutions in almost all cases for the Slashdot and WikiTalk networks (using the TREESKETCH algorithm). Compared to the paths returned by the basic SKETCH algorithm of [6], for several datasets we are able to provide two orders of magnitude improvement in approximation quality using SKETCHCESC and TREESKETCH. The simple cycle elimination enhancement also leads to a considerable decrease of estimation errors to the order of close to 1.7-2 factors for the datasets under consideration.

4.3.2 *Query Execution Time*

The second important evaluation category we are assessing is query execution time. We compare the results of our methods, averaged over the test triples, to the average running time of Dijkstra's algorithm. The results are listed in table 2, a semilogarithmic plot of the query execution times is depicted in Figure 5.

Observe that, using algorithm SKETCHCESC, we are able to answer shortest path queries with excellent accuracy within on average 190 milliseconds for the smallest dataset (Slashdot) to 4 seconds in the large Twitter dataset. Using algorithm TREESKETCH we can provide even better path estimations at an negligible additional time overhead, providing the results between one and two orders of magnitude faster than the classical Dijkstra algorithm. Note that for the Slashdot and Wikitalk dataset, the query is executed extremely fast while achieving an approximation error of 0% for almost all of the test triples.

All query execution measurements have been carried out on an out-of-the-box laptop with a 2.0 GHz Intel Core 2 Duo processor and 4 GB of RAM, running Ubuntu Linux 10.04. Note that we filtered the previously generated test triples in such a way, that no node of the graph appears in more than one test triple. This way, we try to eliminate caching effects of the RDF-3X database system.

4.3.3 *Path Diversity*

In many application settings it is not only important to quickly generate an accurate approximation of the shortest path, but also crucial to return as many candidate paths as possible. Our algorithms are designed in such a way that this goal can be satisfied.

They return an ordered queue of paths which can – for example – be used to filter out candidates based on some user-specified constraints. The average number of generated shortest paths is given in table 4.

Table 4: Number of paths obtained

	Slashdot	Google	YouTube	Flickr	WikiTalk	Twitter	Orkut
SketchCESC	15.0	2.4	19.3	33.3	18.6	45.5	9.5
TreeSketch	31.5	3.1	40.8	55.6	50.7	92.0	29.6

The number of candidate paths created by TREESKETCH is always greater than the number of paths generated by the other variants. For the Twitter dataset, we able to generate 92 paths on average, more than twice as much as provided by our SKETCHCESC algorithm.

4.3.4 *Preprocessing*

Finally, we evaluate the space and time requirements for the preprocessing step (the sketch computation).

Space Requirements. We evaluate the space consumption of the sketches by comparing their size against the size of the orginal database and the sketches computed by the original sketch algorithm [6], that store just the distances. See Table 3 and Figure 6 for a detailed overview over the necessary disk space for the different datasets. The path sketches

Figure 5: Query Execution Times

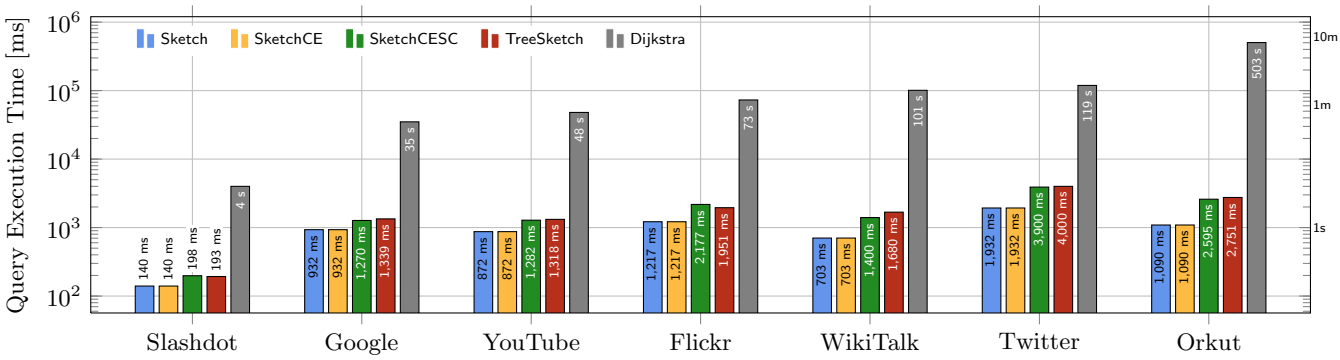


Figure 6: Sketch Space Consumption for $k = 2$

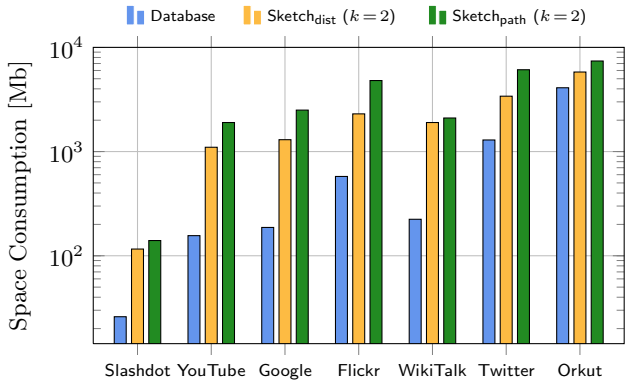


Table 3: Sketch Space and Time Consumption for $k = 2$

Dataset	Database	$\mathsf{Sketch}_{\mathsf{dist}}$	$\mathsf{Sketch}_{\mathsf{path}}$	t_{precomp} $[s]$
Slashdot	26 Mb	112 Mb	139 Mb	267.2
Google	0.15 Gb	1.1 Gb	1.9 Gb	4,156.6
YouTube	0.19 Gb	1.3 Gb	2.5 Gb	3,159.0
Flickr	0.57 Gb	2.3 Gb	4.4 Gb	2,223.4
WikiTalk	0.22 Gb	1.9 Gb	2.1 Gb	5,430.0
Twitter	1.30 Gb	3.4 Gb	6.1 Gb	12,806.1
Orkut	5.70 Gb	5.8 Gb	7.4 Gb	36,486.2

surpass the original database size by a factor of at most 10 (with the exception of the Google dataset). Compared to the original distance sketches, storing the actual paths instead of just the distances incurs a comparatively low additional cost, materializing in a factor of 2 in the worst and 1.1 in the best case.

Precomputation Time. We measure the running time of the preprocessing stage for all datasets. Table 4 provides an overview over the required times for $k = 2$ preprocessing iterations. Note that the required time increases linearly in k. For small datasets like Slashdot, the sketches can be obtained within five minutes, while the largest dataset (Orkut) requires about 11 hours of preprocessing.
The time measurements were conducted on Dell PowerEdge M610 servers, each of which has two Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, and runs Debian GNU/Linux with SMP Kernel 2.6.29.3.1 as an operating system.

5. CONCLUSIONS AND OUTLOOK

In this paper, we have presented fast and accurate algorithms for the approximation of shortest paths, building upon a previously proposed algorithm for distance estimation using precomputed sketches. We describe how simple, yet powerful, modifications to the original algorithm together with an entirely new algorithm operating on the same precomputed data yield paths with excellent quality while attaining query execution times that beat the classical Dijkstra shortest path algorithm by up to two orders of magnitude. In addition, we are able to generate not only one but many candidate paths, as our experimental evaluation on a number of real world datasets underpins.
Our algorithms have been implemented within the recently proposed high-performance RDF storage and retrieval system, RDF-3X. To evaluate the quality and speed of our approaches, we have conducted a large number of experiments that prove the practicability of our algorithms in terms of speed, quality, and diversity.
A promising future research direction in this area is the consideration of user-provided constraints (e. g. on the type or label of the nodes and edge) that have to be satisfied by the returned paths.

Acknowledgements

This work is supported by DFG (German Research Foundation) within the priority research program 1355 “Scalable Visual Analytics”.

6. REFERENCES

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC 2007 + ASWC 2007: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference*, Lecture Notes in Computer Science 4825. Springer, 2007.

[2] H. Bast. Car or Public Transport – Two Worlds. Lecture Notes in Computer Science 5760/2009, pages 355–367. Springer, 2009.

[3] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *ALENEX'07: Proceedings of the 2007 SIAM*

Workshop on Algorithm Engineering and Experiments. SIAM, 2007.

[4] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-Hop-Labels. In *SODA'2002: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 937–946, 2002.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[6] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A Sketch-Based Distance Oracle for Web-Scale Graphs. In *WSDM'10: Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*, pages 401–410, New York, NY, USA, 2010. ACM.

[7] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *ALENEX'06: Proceedings of the 2006 SIAM Workshop on Algorithm Engineering and Experiments*. SIAM, 2006.

[8] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: A High-Compression Indexing Scheme for Reachability Query. In *SIGMOD'09: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 813–826. ACM, 2009.

[9] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently Answering Reachability Queries on Very Large Directed Graphs. In *SIGMOD'08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 595–608. ACM, 2008.

[10] M. Kanehisa, S. Goto, M. Hattori, K. F. Aoki-Kinoshita, M. Itoh, S. Kawashima, T. Katayama, M. Araki, and M. Hirakawa. From Genomics to Chemical Genomics: New Developments in KEGG. *Nucleic Acids Research*, 34 (Database Issue):354–357, 2006.

[11] D. E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, 1981.

[12] J. Köhler, J. Baumbach, J. Taubert, M. Specht, A. Skusa, A. Rüegg, C. J. Rawlings, P. Verrier, and S. Philippi. Graph-Based Analysis and Visualization of Experimental Results with ONDEX. *Bioinformatics*, 22(11), 2006.

[13] J. Küntzer, C. Backes, T. Blum, A. Gerasch, M. Kaufmann, O. Kohlbacher, and H.-P. Lenhof. BNDB - The Biochemical Network Database. *BMC Bioinformatics*, 8, 2007.

[14] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting Positive and Negative Links in Online Social Networks. In *WWW'10: Proceedings of the 19th International World Wide Web Conference*, pages 641–650. ACM, 2010.

[15] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *KDD'2005: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 177–187. ACM, 2005.

[16] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *arXiv:0810.1355v1*, October 2008.

[17] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *SIGCOMM'07: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, pages 29–42. ACM, 2007.

[18] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal – International Journal on Very Large Data Bases*, 19(1):91–113, 2010.

[19] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast Shortest Path Distance Estimation in Large Networks. In *CIKM'09: Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 867–876. ACM, 2009.

[20] R. C. Prim. Shortest Connection Networks and some Generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

[21] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling Online Social Networks without Pains. In *NetDB'09: 5th International Workshop on Networking Meets Databases*, 2009.

[22] J. Sankaranarayanan and H. Samet. Distance Oracles for Spatial Networks. In *ICDE'09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 652–663. IEEE Computer Society, 2009.

[23] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *EDBT'04: Proceedings of the 9th International Conference on Extending Database Technology*, Lecture Notes in Computer Science 2992, pages 237–255, 2004.

[24] C. Sommer, E. Verbin, and W. Yu. Distance Oracles for Sparse Graphs. In *FOCS'09: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 703–712, 2009.

[25] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *WWW'07: Proceedings of the 16th International World Wide Web Conference*, pages 697–706. ACM, 2007.

[26] M. Thorup and U. Zwick. Approximate Distance Oracles. In *STOC'01: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 183–192. ACM, 2001.

[27] S. Trißl and U. Leser. Fast and Practical Indexing and Querying of Very Large Graphs. In *SIGMOD'07: Proceedings of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*, pages 845–856. ACM, 2007.

[28] F. B. Zhan and C. E. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transportation Science*, 32(1):65–73, 1998.

[29] U. Zwick. Exact and Approximate Distances in Graphs – A Survey. In *ESA'01: Procceding of the 9th Annual European Symposium on Algorithms*, Lecture Notes in Computer Science 2161/2001. Springer, 2001.