Erschienen in: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) - GRADES-NDA'19. - New York : ACM Press, 2019. - 7
https://dx.doi.org/10.1145/3327964.3328496

Experiences with Implementing Landmark Embedding in Neo4j

Manuel Hotz
University of Konstanz
Konstanz, Germany
manuel.hotz@uni.kn

Theodoros Chondrogiannis
University of Konstanz
Konstanz, Germany
theodoros.chondrogiannis@uni.kn

Leonard Wörteler
University of Konstanz
Konstanz, Germany
leonard.woerteler@uni.kn

Michael Grossniklaus
University of Konstanz
Konstanz, Germany
michael.grossniklaus@uni.kn

ABSTRACT

Reachability, distance, and shortest path queries are fundamental operations in the field of graph data management with various applications in research and industry. However, while various preprocessing-based methods have been proposed to optimize the computation of such queries, the integration of existing methods into graph database management systems and processing frameworks has been limited. In this paper, we present an implementation of a static graph index that employs *landmark embedding* for Neo4j, to enable the index-based computation of reachability, distance, and shortest path queries on the database. We explore different strategies for selecting landmarks and different schemes for storing the precomputed landmark distances. To evaluate the efficiency of each landmark selection strategy and each storage scheme, we conduct an experimental evaluation using four real-world network datasets. We measure the preprocessing cost, the query processing time, and the accuracy of the distance estimation of different configurations of our index structure.

KEYWORDS

Landmark-based Indexing, Shortest Paths, Graph Databases

1 INTRODUCTION

Large-scale data analysis increasingly focuses on the relationships between entities and the networks that are formed. Various application domains such as route planning on transportation networks, social network analysis, and web marketing require the detection and the analysis of such relationships over very large volumes of graph data. As a result, various graph DBMS such as Neo4j[1], DEX/Sparksee [17], and OrientDB[2], and processing frameworks such as Apache Giraph[3] and GraphX[4] have been developed.

A key component of graph data analysis that provides useful insight in a number of different domains is the detection and analysis of paths with certain characteristics. Path analysis is used on road networks for navigation, on communication networks for identifying crucial connections in a network, and on social network graphs to identify relationships and interactions between users. Towards the development of efficient solutions for computing path analytics, three important queries that have been studied extensively are reachability, distance and shortest path queries. These queries are not only directly useful in a number of different scenarios, e.g., route planning, but they can also be used as building blocks for more complex queries, e.g., k-Nearest Neighbor queries, itinerary planning queries, etc.

To achieve the scalable computation of reachability, distance and shortest path queries, various preprocessing-based methods have been proposed [2, 23, 24, 30]. Such methods precompute information incurring a relatively high yet one-time cost, and utilize the precomputed information to process queries efficiently. However, despite the fact that the aforementioned queries are some of the most important graph-specific operations, existing graph DBMS do not provide indexing structures to efficiently process such queries. While existing preprocessing-based methods support the computation of queries in main memory, to the best of our knowledge, only one preprocessing-based method has been adapted and implemented into a graph DBMS [21].

In this paper, we present an implementation of a static graph index for Neo4j that employs *landmark embedding*, a cost-efficient preprocessing scheme for approximate distance computation. We adapt existing algorithms to enable the index-based processing of reachability, distance and shortest path queries. We explore different strategies for selecting landmarks and different schemes for storing the precomputed landmark distances. To evaluate the efficiency of our static index structure, we conduct an experimental evaluation using four real-world network datasets. For each landmark selection strategy and each storage scheme, we measure the

[1] https://neo4j.com
[2] https://orientdb.com
[3] https://giraph.apache.org
[4] https://spark.apache.org/graphx/

preprocessing cost, the query processing time for reachability, distance and shortest path queries, and the accuracy of the estimation for approximate distance queries.

The rest of the paper is organized as follows: Section 2 overviews the related work. Section 3 introduces basic definitions along with key notation that is used throughout the paper. Section 4 overviews the preprocessing required for landmark embedding, different landmark selection strategies, and the queries we investigate. In Section 5, we elaborate on the technical details of the implementation of our static, landmark-based index for the Neo4j graph DBMS. In Section 6, we present the results of our experimental evaluation. Finally, Section 7 concludes the paper and points to future work.

2 RELATED WORK

Approximate Distance Queries. Landmark embedding has been studied extensively in the context of processing approximate distance queries on various types of graphs, e.g., communication networks [25] and social networks [28]. Towards the efficient processing of such queries, various approaches have been proposed. Potamias *et al.* [18] conduct an extensive analysis to measure the effect of various landmark selection strategies on approximate distance computation. Gubichev *et al.* [11] propose a scheme for both distance and shortest path estimation that employs landmarks to construct path-sketches, i.e., structures that maintain the complete path between every node and the selected landmarks. Tretyakov *et al.* [26] propose a storage scheme for landmark distances that optimizes the computation of approximate distances and supports dynamic updates on the graph as well. Qiao *et al.* [20] propose a query-dependent local landmark scheme which finds a landmark close to both the source and the target node, thus improving the distance estimation accuracy, without having to consider all landmarks. Qi *et al.* [19] study the landmark embedding for approximate distance computation on billion-node graphs. More recently, Bonchi *et al.* [1] extended landmark embedding to process approximate distance queries on edge labeled graphs.

Distance and Shortest Path Queries. The classical solution for processing distance and shortest path queries is Dijkstra's algorithm [7]. Despite being an elegant solution, Dijkstra's algorithm does not scale for large graphs. Therefore, to achieve scalability, various preprocessing-based methods for shortest path queries have been proposed [23]. Towards this end, Goldberg *et al.* [8] proposed the ALT algorithm, a popular goal-directed preprocessing-based method that employs landmark embedding in combination with the triangle inequality to compute lower bounds. Then an A^* *search* [12] is employed that uses the landmark-based lower bounds as heuristics to prioritize the expansion of nodes that are closer to the target during the computation of the shortest path. As we employ the ALT algorithm in this work for processing distance and shortest path queries, we elaborate on it more in Section 4.

Reachability Queries. More recently, landmark embedding has been used to process *label-constraint reachability queries* on labeled graphs. Valstar *et al.* [27] construct a landmark-based approach that precomputes the reachability of each landmark to all nodes of the graph for multiple label constraints, and conduct a label-respecting breadth-first search, while exploiting the precomputed information to terminate the search early whenever possible.

3 PRELIMINARIES

Let $G = (N, E)$ be a *directed weighted* graph with a set of nodes N and a set of edges $E \subseteq N \times N$. Each edge $(n_i, n_j) \in E$ is assigned a *positive* weight $w(n_i, n_j)$, which captures the cost of moving from node n_i to node n_j. A (simple) path $p(s \rightarrow t)$ from a source node s to a target node t is a *connected* and *cycle-free* sequence of nodes $\langle s, n_1, \ldots, t \rangle$. The length $\ell(p)$ of a path p is the sum of the weights of all contained edges, i.e.,

$$\ell(p) = \sum_{\forall (n_i, n_j) \in p} w(n_i, n_j). \tag{1}$$

The shortest path $p_{sp}(s \rightarrow t)$ is the path with the lowest length among all paths that connect nodes s and t. The *distance* of two nodes is defined as the length of the shortest path between them, i.e., $d(s, t) = \ell(p_{sp}(s \rightarrow t))$.

As we have already mentioned, Dijkstra's algorithm is the classical solution to find the shortest path while A^* *search* improves upon Dijkstra's algorithm by employing a heuristic function to prioritize the traversal of nodes that are closer to the target. With Dijkstra's algorithm, which traverses the graph expanding nodes in increasing order of their distances from the source, A^* *search* expands nodes considering both the distance from the source and their estimated distance to the target. The estimated cost $\hat{f}(n)$ that is used to determine the order in which the nodes are expanded is

$$\hat{f}(n) = d(s, n) + \hat{h}(n, t) \tag{2}$$

and $\hat{h}(n)$ is an estimate for the distance from n to t. To guarantee the correctness of the result, $\hat{h}(n, t)$ must always be a lower bound for the distance from n to t, i.e., $\hat{h}(n, t) \leq d(n, t)$. Note that if $\hat{h}(\cdot, \cdot)$ is always 0, then A^* *search* emulates Dijkstra's algorithm.

4 LANDMARK-BASED QUERY PROCESSING

In this section, we review *landmark embedding*, a cost-efficient approach for computing upper and lower bounds, and we show how to employ landmarks to process approximate distance, distance, shortest path, and reachability queries. Due to the fact that landmark-based lower bounds are network-specific, i.e., they carry some information w.r.t. the structure of the graph, they are expected to be much tighter than heuristics such as the Euclidean distance.

4.1 Landmark-based Bounds

During the *preprocessing* phase, a small set of k nodes of G is chosen as landmarks L. Then, for each of the selected landmarks $l \in L$ we construct the tree from l to all the other nodes of the graph and the reverse shortest path tree to l from all the other nodes of the graph. As a result, all distances $d(n, l)$ from each landmark $l \in L$ to a node n of the graph, and all distances $d(l, n)$ from a node n to each landmark $l \in L$ are precomputed.

Having precomputed all $d(n, l)$ and $d(l, n)$ distances, by using the triangle inequality, we are able to obtain both upper and lower bounds for the distance from s to t.

Given a graph $G = (N, E)$, a source node $s \in N$, a target node $t \in N$ and a landmark node $l \in N$, from the triangle inequality we

have:

$$d(s,t) \leq d(s,l) + d(l,t) \tag{3}$$

$$d(s,l) \leq d(s,t) + d(t,l) \Rightarrow d(s,l) - d(t,l) \leq d(s,t) \tag{4}$$

$$d(l,t) \leq d(l,s) + d(s,t) \Rightarrow d(l,t) - d(l,s) \leq d(s,t) \tag{5}$$

which provide both an upper and lower bound for the distance from s to t.

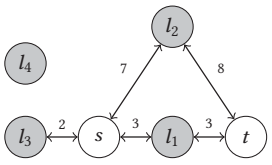
Naturally, obtaining tighter bounds, i.e., upper and lower bounds that are close to the actual distance from s to t, is preferable. Considering only a single landmark may not result in very tight bounds. For instance, let a landmark l lie far away from both s and t while its distance from/to s is almost equal to its distance from/to t. In such a case, the value of the upper bound is going to be very high, while the value of the lower bound will be very close to zero. Hence, instead of choosing a single landmark, it is preferable to choose a set of landmarks L to maximize the chances of obtaining tight bounds. Therefore, following from Equation 3, the upper bound for the distance $d(s,t)$ obtained using a set of landmarks L is

$$\overline{d}(s,t) = min_{\forall l \in L}\{d(s,l) + d(l,t)\} \tag{6}$$

while, following from Equations 4 and 5, the lower bound for the distance $d(s,t)$ obtained using a set of landmarks L is

$$\underline{d}(s,t) = max_{\forall l \in L}\{d(s,l) - d(t,l), d(l,t) - d(l,s)\} \tag{7}$$

Figure 1 illustrates how upper and lower bounds are computed using a set of landmarks L. Landmark l_1 gives the tightest upper bound of 6, as it lies on the shortest path between s and t, landmark l_3 gives the tightest lower bound of 6, since it lies "behind" s, and landmark l_2 gives a bad upper and lower bound for $d(s,t)$. Landmark l_4 demonstrates the distances stored for a landmark that is connected to neither s nor t.



	lower	upper
l_1	0	6
l_2	1	15
l_3	6	10
l_4	0	∞

Figure 1: Example of upper and lower bound computation using a set of 4 landmarks (tightest bounds in a frame).

As we observe from our previous example, the position of the landmarks in relation to the source and the target nodes affects heavily the quality of the upper and lower bounds we obtain.

4.2 Landmark Selection Strategies

In order to select the set of landmarks L, different strategies have been proposed [8, 9, 18, 26]. We concentrate on strategies that show promising results, i.e., low distance approximation error, while requiring relatively low preprocessing time.

Random. The k landmarks are selected uniformly at random from the set of all nodes in the graph.

Degree. The k nodes with the highest degree are selected as landmarks (ties are resolved arbitrarily).

Farthest. The greedy strategy proposed by Goldberg *et al.* [8] picks as the next landmark, the most distant node to the previously selected landmarks. The first landmark is selected at random.

BestCover. Potamias *et al.* [18] define the *coverage* of landmarks w.r.t. a pair of nodes (s,t), i.e., L *covers* (s,t), if $L \cap p_{sp}(s \rightarrow t) \neq \emptyset$. Tretyakov *et al.* [26] propose a greedy strategy to approximate the optimal landmarks set w.r.t. coverage, since computing the optimal set is NP-hard [18]. A set M of node pairs is sampled and the shortest paths SP_M that connect each pair of nodes are computed. Landmarks are selected in an iterative fashion. At each round, a node is chosen as the next landmark such that it covers the most shortest paths in SP_M. After the landmark is selected, the set of covered shortest paths is then removed from SP_M. This process continues until k nodes have been selected as landmarks.

Avoid. The main idea behind this method, proposed by Goldberg *et al.* [9], is to extend a set of landmarks while minimizing errors in parts of the graph that are not well served. Starting from a randomly selected landmark l, i.e., $L = \{l\}$, we select a node n at random such that $n \notin L$ and we compute its shortest path tree. A score is computed recursively for each node of the tree based on the estimation errors of the nodes of its subtree. Then, we pick the subtree of the root which has the highest score, we traverse the tree expanding each child with the highest score, until a leaf node is reached. That leaf node is selected as the next landmark. The process is repeated until k nodes have been selected as landmarks.

4.2.1 Adapting Strategies for Directed Graphs. In a directed graph, there are potentially two different distances between a pair of nodes—one in each direction. Some of the landmark selection strategies presented above were either explicitly or implicitly proposed in the context of undirected graphs, i.e., Degree (w.r.t. in- and out-degree), Farthest and Avoid. For example, the notion of "farthest away", e.g., as optimized by the Farthest and Avoid strategies, is ambiguous in a directed graph. Hence, some of the selection strategies have to be adapted for this use case. Since there are two distances for each pair of node and landmark, the Farthest strategy leaves two choices. If we search for the "nearest landmark", the two distances have to be consolidated right away in order to be comparable. In contrast, if a different landmark is allowed to be "nearest" in each direction, the consolidation can be postponed until nodes are compared. The Avoid strategy has the option to only calculate approximation errors in one direction, i.e., by using only outgoing edges, relying on a fortunate initialization for the shortest path tree for each landmark selection. However, this can lead to a bad sample. For example, in a tree-like graph, shortest path subtrees often only contain a small portion of all nodes. Instead, two spanning trees—one for outgoing and one for incoming edges—can be combined, again yielding two distances per node. For both Random and BestCover no changes have to be made, since neither of them use the notion of distance between nodes. In both Farthest and Avoid we maintain two distances and take the average before comparison, in Degree we use the total degree.

4.3 Query Processing

Approximate Distance Queries. Following from Equations 6 and 7, the result of an approximate distance query from s to t can be any value in the interval $[\underline{d}(s,t), \overline{d}(s,t)]$, including the upper and lower bounds. For instance, Potamias *et al.* [18] return the upper bound as the result for an approximate distance query, while Goldberg *et al.* [8] employ the lower bound in their ALT algorithm.

Distance and Shortest Path Queries. To process distance and shortest path queries we employ the ALT algorithm [8, 9], a goal-directed method that employs the lower bound of Equation 7 as a heuristic for A^* *search* [12], i.e., $\hat{h}(n,t) = \underline{d}(n,t)$. Note that the computation of a distance query $d(s,t)$ is essentially equivalent to the computation of the shortest path $p_{sp}(s \rightarrow t)$. In fact, to compute the shortest path $p_{sp}(s \rightarrow t)$ the ALT algorithm first computes the distance from s to t and then backtracks its steps to retrieve the list of nodes that form the shortest path.

Reachability Queries. Existing works for reachability query processing using landmarks store connectivity information instead of distances [27]. To enable the processing of reachability queries while maintaining the capability to process distance and shortest path queries, we express the connectivity information using distance values, i.e., $reach(s,t) \Leftrightarrow d(s,t) \neq \infty$. Hence, if a node is not connected to a landmark l and/or vice-versa, then we store $d(s,l) = \infty$ and/or $d(l,s) = \infty$, respectively.

To compute a reachability query $reach(s,t)$, we do not necessarily need to compute the shortest path between nodes s and t. As long as there exists a path from s to some landmark $l \in L$ and a path from l to t, then $reach(s,t) = \text{true}$. However, if such a landmark does not exist, that does not necessarily mean that t is not reachable from s. For the connectivity between s and t w.r.t. a given landmark $l \in L$ we have:

$$reach_l(s,t) = \begin{cases} \text{false}, & \text{if } (reach(l,s) \wedge \neg reach(l,t)) \\ & \quad \vee (reach(t,l) \wedge \neg reach(s,l)), \\ \text{true}, & \text{if } reach(s,l) \wedge reach(l,t), \\ \text{unknown}, & \text{otherwise.} \end{cases} \tag{8}$$

Following from Equation 8, the result of a reachability query $reach(s,t)$ can be computed using a set of landmarks L as

$$reach(s,t) = \bigvee_{l \in L} reach_l(s,t) \tag{9}$$

Note that only if $reach(s,t) = \text{unknown}$, we need to run a graph traversal algorithm to determine whether there is a path connecting s and t. This can be the case, e.g., if the graph is disconnected, and s and t lie in a weakly-connected component of the graph that has not been assigned any landmarks.

5 IMPLEMENTATION IN NEO4J

In this section, we detail how the concepts of the previous section are transferred to the graph database Neo4j. We discuss the technical challenges and the decisions we made to implement the landmark selection strategies of Section 4.2 and the ALT algorithm along with the required access methods for query processing. Since we want to eschew Cypher parsing and execution overhead, our implementation does not use Cypher queries internally. Instead, we implemented all required operations for constructing the landmark-based index and for query processing using only direct API calls.

5.1 Index Construction

The index construction consists of three phases: landmark *selection*, landmark distance *computation*, and landmark distance *storage*.

5.1.1 Landmark Selection. For selecting a set of k landmarks from the graph we implement the selection strategies of Section 4.2. For the implementation of the Random strategy we use reservoir sampling. For the Degree selection strategy, we retrieve the k nodes with the maximum total degree from the database. The retrieved nodes are maintained in a max-heap with its capacity limited to k nodes.

The Farthest strategy tries to choose landmarks which are far away from previously chosen landmarks. Starting from a randomly selected landmark, we execute Dijkstra's algorithm twice from the selected landmark to all the nodes of the graph, once by traversing outgoing edges and once by traversing incoming edges in reverse. We repeat this process until k landmarks are chosen, but instead of choosing a subsequent landmark randomly, we choose the node with the largest distance to its nearest previously selected landmark.

The BestCover selection strategy is a greedy strategy optimizing the coverage criterion for the chosen landmarks. More specifically, we uniformly sample a large number of node pairs and compute the respective shortest paths. For each node that lies on some of the computed shortest paths, we count the number of shortest paths that the node lies on. The node that lies on the most shortest paths is selected as the first landmark. Then, we update the set of the computed shortest paths by removing the paths that cross the node that has been selected as landmark, and we repeat the process until k landmarks are chosen.

The Avoid selection strategy augments a set of chosen landmarks iteratively by minimizing approximation errors in a greedy fashion. Again, starting from a randomly selected landmark l, we compute the distances from and to all the nodes of the graph using Dijkstra's algorithm. At each iteration, we chose another random node n and compute the shortest path tree T to all other nodes of the graph. For each node $n' \in N \setminus \{l, n\}$ we retrieve both the exact distance and the approximate distance from n that is computed using all previously selected landmarks. Next, for each node n' we sum the approximation errors of n' and all its descendants in T, unless one of the descendants is a landmark in which case the cumulative approximation error is set to zero. Then, we obtain the subtree of T that is rooted at the node with the maximum cumulative approximation error, and we execute a *depth-first search* from the root expanding every time the child with the highest cumulative approximation error among its siblings, until a leaf node is found; that leaf node is selected as the next landmark. This process is repeated until k landmarks are determined.

5.1.2 Landmark Distance Computation. After the selection of the set of landmarks L, the second phase is the computation of two sets of landmark distances, i.e., distances *from* each landmark to each node, and distances from each node *to* each landmark. To compute these distances, we execute Dijkstra's algorithm twice from each selected landmark to all the nodes of the graph, once by

traversing outgoing edges and once by traversing incoming edges in reverse. This results in $2 \cdot |L|$ executions of Dijkstra's algorithm. Note that if the landmark selection strategy we employ requires the computation of the same distances, i.e., FARTHEST and AVOID, then the distances computed during the landmark selection phase can be retained and do not have to be computed from scratch.

5.1.3 Index Storage Schemes. To store the landmark distances, we implement three different storage options, each of which uses a different mechanism provided by the Neo4j platform.

In-Memory Distance Storage. The first storage option we implement involves the creation and maintenance of the distances between all landmarks and nodes in main memory. We store all the distances of nodes to and from landmarks in a three-dimensional array of size $|N| \cdot |L| \cdot 2$. The advantage of such a storage scheme is that it eliminates disk access. Hence, lookup operations are expected to be fast since they take place in main memory. However, the drawback of this storage scheme is that once the database server is restarted, landmark distances have to be either recomputed from scratch, or recovered from external memory, possibly creating a very long start-up time.

Distances as Node Properties. The second storage scheme stores landmark distances as node properties. Each node of the graph maintains two arrays, one for the distances *to* all landmarks and one *from* all landmarks, stored as properties. This storage scheme promises good data locality, since for each node all distances to landmarks are stored close by. Neo4j stores these distance arrays in a single file, i.e., the `propertystore.db.arrays` file where all array-valued properties are stored.

Distances as Relationship Properties. The last storage scheme is motivated by the observation that, conceptually, our precomputed landmark distances connect landmarks and nodes with each other. Relationships are a natural way to model those connections and store their distances. Therefore, we create two relationships between each landmark and all the other nodes of the graph, and we store the distances between landmarks and nodes as properties of the created relationships. Obviously, if a landmark is not reachable from a given node or vice-versa, then the respective relationship is not created. In Neo4j, the data of this storage scheme is stored in two separate files, i.e., the `relationshipstore.db` file where all relationships are stored, and the `propertystore.db` file where scalar properties are stored.

5.2 Query Processing in Neo4j

Approximate distance computation between pairs of nodes is the main operation of our index structure and is an essential part of the computation of distance, shortest path, and reachability queries. Since each of the three storage schemes described in Section 5.1.3 provides a different access method to distance information, we first explain how the index is employed to determine the approximate distances between two nodes for each of these storage schemes. Afterwards, we show how these approximations are applied for distance, shortest path, and reachability query processing.

5.2.1 Approximate Distance. To compute the approximate distance between two nodes s and t, the index employs a different access method to compute upper and lower bounds based on the storage scheme that is currently is use.

In-Memory Distance Storage. Conceptually, the in-memory storage scheme performs a join on landmark distances for s and t. In our implementation, we retrieve landmark distances for nodes s and t by direct array access on the precomputed three-dimensional array. To calculate the lower bound $\underline{d}(s,t)$, we apply Equation 7 on the retrieved distances, while to calculate the upper bound $\overline{d}(s,t)$, we apply Equation 6. Note that since the lookup tables store ∞ for distances to non-reachable nodes, we have to adapt the equations to return at minimum 0 to uphold the requirement of non-negative estimations.

Distances as Node Properties. When storing landmark distances as node properties, calculating the approximate distance involves the retrieval of distances to/from landmarks for nodes s and t from their respective node properties. Then, similar to the in-memory storage scheme, we use Equation 7 and Equation 6 to calculate the lower and upper bound, respectively. Again, we have to account for infinite distances for unreachable nodes.

Distances as Relationship Properties. When landmark distances are stored as relationship properties on directed edges between landmarks and nodes, the approximate distance between two nodes s and t can be concisely expressed as a Cypher query. As an example, let `L` be the label for a landmark, `L_REL` the type for precomputed relationships between landmarks and nodes, and `dist` the weight property on these relationships. The following query computes Equation 7 and returns the lower bound for the distance between two nodes s and t using all landmarks in the graph:

```
MATCH (s)-[rsL:L_REL]->(l:L), (l:L)-[rLs:L_REL]->(s),
      (t)-[rtL:L_REL]->(l:L), (l:L)-[rLt:L_REL]->(t)
WHERE s.name = 's' AND t.name = 't'
UNWIND [rsL.dist - rtL.dist, rLt.dist - rLs.dist] as est
RETURN max(est) as tightestLower
```

The upper bound is computed in a similar fashion by switching the UNWIND and RETURN expressions to match Equation 6. We implement this Cypher query in Java as a hash join to provide an efficient implementation and eschew query execution overhead. In the build phase of the hash join, landmark distances are retrieved from incoming and outgoing landmark relationships of s and l. Then in the probe phase, landmark distances for incoming and outgoing landmark relationships between l and t are used to calculate the lower or the upper bound.

5.2.2 Distance and Shortest Paths. To process distance and shortest path queries, we employ the ALT algorithm [8]. We implemented an A^* *search* in Neo4j using direct API calls and abstract over the heuristic function such that we are able to plug in estimators for our different storage schemes easily. As mentioned before, for the purposes of this work we do not differentiate between shortest path and distance queries, since as soon as a distance is computed, the shortest path can be reconstructed by backtracking or by storing it along the way. In our implementation, we store the shortest path in each intermediate result for low latency answers. The shortest path is computed using our A^* *search* along with the lower bounds obtained using our our index as the heuristic.

Table 1: Datasets used in the experiments. For the non-road datasets, we retained only the largest (strongly) connected component, indicated by *(S)CC*.

Dataset	Direction	Edge Weights	Component	Network Type	$\|N\|$	$\|E\|$	Diameter	DB Size
DBLP [29]	Undirected	None	CC	Social	$317K$	$1M$	21	39 MiB
Web-Google [16]	Directed	None	SCC	Web	$435K$	$3.4M$	21	118 MiB
CAL [6]	Directed	Travel Distance	All	Road	$1.9M$	$4.7M$	$2.3 \cdot 10^3$	360 MiB
Wiki-Talk [15]	Directed	None	SCC	Communication	$111K$	$1.4M$	9	50 MiB

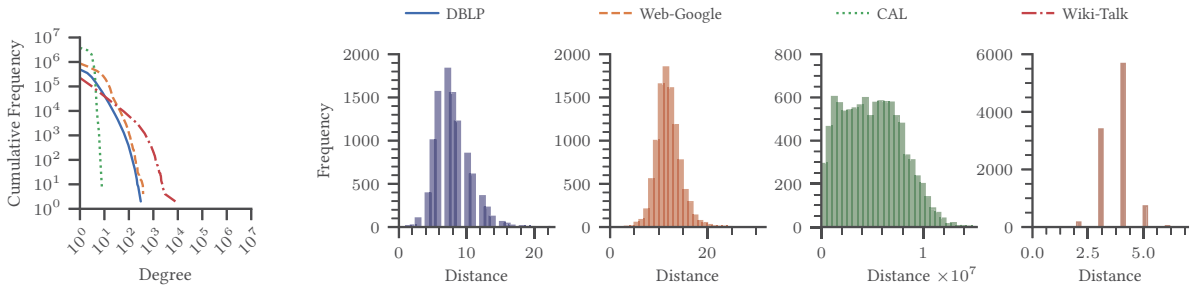


Figure 2: Degree and distance distribution in evaluation datasets. Distances are sampled from 10000 random node pairs.

5.2.3 Reachability. Lastly, to process a reachability query for two nodes s and t, approximate distances for s and t to and from landmarks are retrieved from the index. Based on Equation 8, the answer is computed over all landmarks. If after applying this calculation for each landmark the answer is "unknown", a *breadth-first search* is executed to determine whether there exists a path from s to t.

6 EXPERIMENTAL EVALUATION

In this section, we present the result of our experimental evaluation where we measure the index construction time, the space overhead, and the running time for processing shortest path queries for each of the storage schemes presented in Section 5.1.3. We also report the approximation error that each landmark selection strategy yields on approximate distance queries. Our tests run on a server-grade machine with two AMD EPYC 7351 16-Core processors, 512GiB 2666MHz DDR4 memory, Intel 660P NVMe SSD, running GNU/Linux 4.15.0-46, OpenJDK 11, and Neo4j 3.5.0.

6.1 Datasets

We run the experiments using a diverse set of real-world datasets.

DBLP This dataset represents a co-authorship network, extracted from DBLP. An undirected edge exists between two authors if they published at least one paper together.

Web-Google A hyperlink network provided by Google, in which nodes represent websites and directed edges hyperlinks from one website to another.

CAL The directed road network of California and Nevada. Edges are weighted by travel distance.

Wiki-Talk The nodes in this dataset represent Wikipedia users. A directed edge is present if a user edited another user's talk page at least once.

Table 1 lists the structural characteristics of the networks we use, while Figure 2 illustrates the degree and distance distributions of each network. Note that, if the network is undirected, we only store one edge, since Neo4j's traversal incurs no performance cost when traversing edges in reverse direction.

6.2 Indexing Cost

The first set of experiments measure the index construction time and the space overhead. Recall that the index construction consists of three phases: landmark selection, landmark distance computation, and landmark distance storage. As the time required to compute landmark distances is invariant over the storage scheme and depends solely on the number of landmarks, i.e., requires $2 \cdot |L|$ runs of Dijkstra's algorithm, we report the time and storage requirements for the first and the last phase only.

Landmark Selection Time. Figure 3 shows the landmark selection time for each strategy on all datasets for varying number of landmarks. We observe in all four datasets that the selection time using the Random and BestCover selection strategies is almost constant with increasing number of landmarks, while for the Degree selection strategy we observe only a slight increase with the number of landmarks. That is to be expected as the Random and Degree strategies do not execute any graph traversals to determine the landmarks, while the BestCover strategies executes a constant number of graph traversals, regardless of the number of landmarks. In contrast, the selection time using the Farthest and Avoid selection strategies increases with the number of landmarks, as both strategies need to performs a number of graph traversals proportional to the number of landmarks.

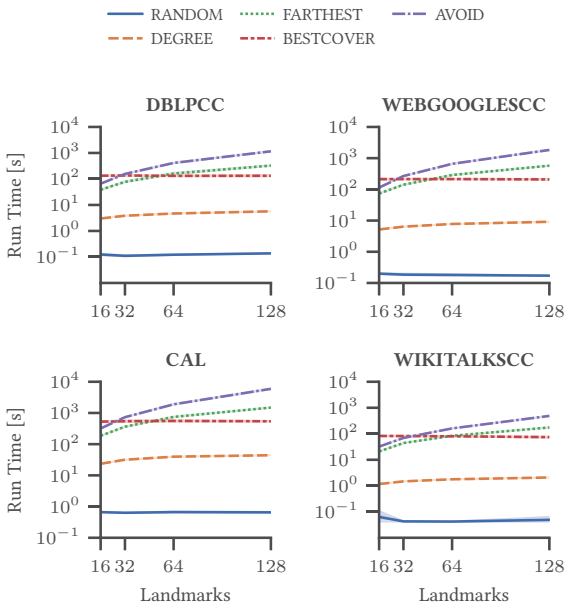


Figure 3: Preprocessing time for landmark selection.

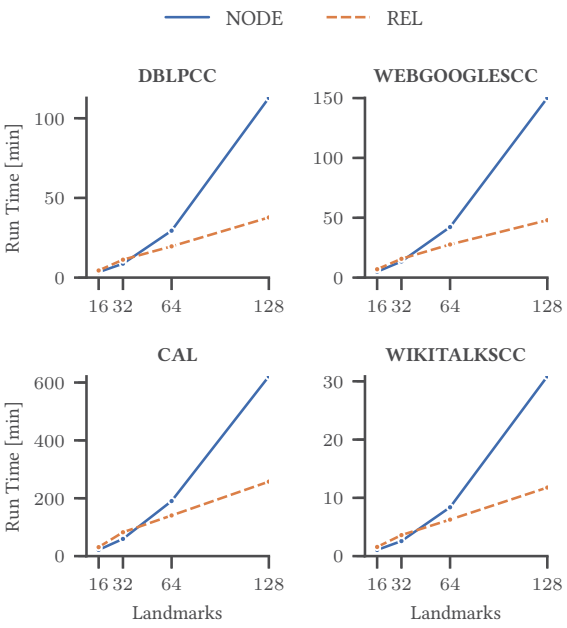


Figure 4: Preprocessing time for landmark distance storage.

Landmark Distance Storage. Figure 4 reports on the storage time for the two disk-based storage schemes. We observe that the storage time required to store landmark distances as node properties increases heavily with the number of landmarks, while the storage time required to store landmark distances as relationship properties increases linearly. This difference in behavior is due to the fact that for updating distances stored as node properties we iterate repeatedly over all the nodes of the graph. For each node we load the entire array of landmark distances from the associated storage file of Neo4j, update the array by setting the new distance, and write the table back to disk. On the contrary, for updating distances stored as relationship properties, we simply append the new relationship and relationship property to the associated storage files of Neo4j.

Figure 5 reports on the storage overhead of our two disk-based storage schemes. We observe that the storage overhead incurred by storing landmark distances as relationship properties is consistently much more that the storage overhead incurred by storing landmark distances as node properties/arrays. For storing distances as node properties, we simply add more information, i.e., the array that stores the landmark distances, on the already allocated space by Neo4j for each node. Hence, the storage overhead is low. On the contrary, to store distances as relationship properties, we need to create new relationships and properties. As a result, Neo4j needs to store additional information w.r.t. the new relationship, thus requiring significantly more space.

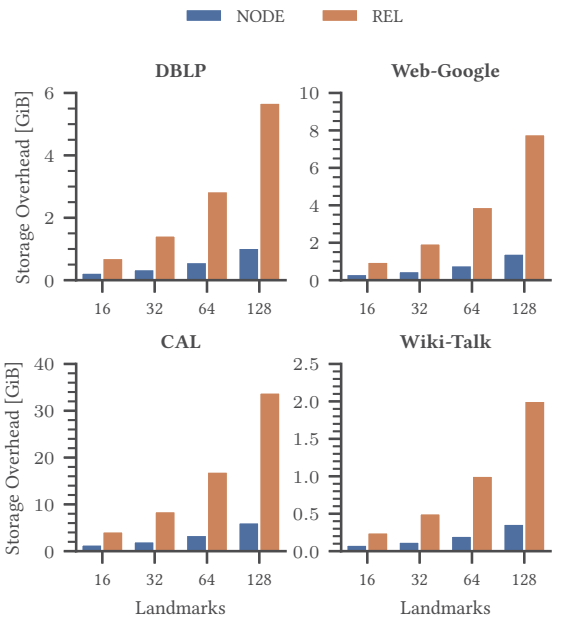


Figure 5: Landmark distance storage overhead.

6.3 Distance Approximation Error

Table 2 reports the average distance approximation error (%) for the lower bound, i.e., $avg(|\underline{d} - d|/d)$, over 1,000 random node pairs for each implemented landmark selection strategy using 16 and 128 landmarks. For the RANDOM strategy, we report the median of 9 runs to mitigate the effect of bad initialization. For the CAL road network graph, we observe that the approximation error of all landmark selection strategies is very low, i.e., varies between 1.5% and 6.5%. For both 16 and 128 landmarks, the AVOID strategy demonstrates the lowest distance approximation error. For all other

Table 2: Percentage of average approximation error for the lower bound using different selection strategies (abbreviated by their initials).

Dataset	$\|\mathcal{L}\|$	R	D	F	B	A
DBLP	16	56.65	58.03	55.82	56.69	55.59
Web-Google	16	62.44	63.49	63.49	62.11	61.18
CAL	16	4.37	6.36	3.37	5.86	2.67
Wiki-Talk	16	67.20	66.81	67.5	67.92	66.91
DBLP	128	40.16	39.93	44.38	41.41	40.03
Web-Google	128	44.23	43.39	46.54	44.67	41.53
CAL	128	1.52	3.49	1.54	1.70	1.26
Wiki-Talk	128	56.73	56.49	56.82	57.26	55.53

datasets, all strategies demonstrate a very high distance approximation error varying between 40% and 70%, which confirms the results found in previous work [1, 20]. In contrast to the road network, the diameter of all other networks is much smaller, thus the distances are very short as well. Hence, even a small absolute error in the distance approximation results in a large approximation error relative to the exact distance. This explains the discrepancy between the distance approximation error on CAL and the non road network datasets. In all cases, the Avoid selection strategy demonstrates the lowest average distance approximation error, with the exception of the DBLP network for 128 landmarks where the Degree selection strategy is slightly better.

6.4 Shortest Path Query Processing

Figure 6 shows ALT's speedup over Dijkstra's algorithm over 500 randomly chosen queries using our three storage schemes, varying the number of landmarks. The execution time of Dijkstra's algorithm is depicted as a horizontal line at 1.0. In all four networks, the speedup using the in-memory scheme is the largest. As the in-memory storage scheme requires no disk access to determine lower bounds, the computation cost is dominated by the disk accesses required to traverse the graph; this cost is the same for all storage schemes. The scheme that stores distances as node properties achieves less speedup than the in-memory storage, but greater speedup than the scheme that stores distances as relationship properties. By storing distances as node properties, for every distance lookup we need to access the disk once to retrieve the array that stores the landmark distances. The scheme that stores distances as relationship properties achieves no speedup. In fact, this scheme makes ALT slower than Dijkstra's algorithm. Since landmark distances are stored in different relationships, each distance lookup needs to access the disk several times to retrieve these relationships and the respective properties where distances are stored.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented an implementation of a graph index that employs *landmark embedding* in Neo4j. We implemented five landmark selection strategies and three different storage schemes using the primitives and the core API provided by Neo4j. We presented access methods to enable the processing of approximate distance,

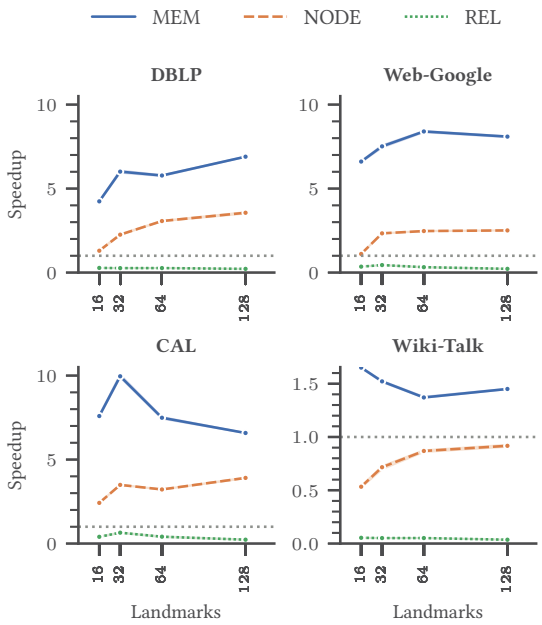


Figure 6: Speedup of ALT versus Dijkstra's algorithm for each storage scheme.

reachability, distance, and shortest path queries using our index. In our experimental evaluation, we used four diverse real-world datasets to evaluate the performance of our index implementation w.r.t. approximation errors, preprocessing cost, and shortest path query processing. We also presented experiments to demonstrate how each storage scheme affects each implemented access method. Apart from validating existing work, our results show that storing landmark distances as node property arrays is much more efficient than storing them as relationship properties. Furthermore, our results indicate that the accuracy of the distance estimation is affected by the type of the input network.

In the future, we plan to investigate how to balance between in-memory and disk-based storage for our index structure. Furthermore, following previous work of our group on query processing on large graphs [3–5, 10, 14, 22] and Cypher query optimization [13], we plan to investigate which characteristics of a given graph affect the performance of our index, and how our index should be configured based on these characteristics. We also plan to extend our landmark-based index to support dynamic updates on the underlying graph and query processing under label constraints, a key functionality of modern graph DBMS. Our long term goal is to implement and evaluate a variety of indexing structures on graph DBMS, and then investigate formal methods to analyze an input graph and select the optimal index structure and configuration automatically.

ACKNOWLEDGMENTS

This work is supported by Grant No. GR 4497/2 of the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. 2014. Distance oracles in edge-labeled graphs. In *Proc. EDBT*.

[2] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying graphs.* Morgan & Claypool Publishers.

[3] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. 2017. Exact and Approximate Algorithms for Finding k-Shortest Paths with Limited Overlap. In *Proc. EDBT*. 414–425.

[4] Theodoros Chondrogiannis and Johann Gamper. 2014. Exploring Graph Partitioning for Shortest Path Queries on Road Networks. In *Proc. GI-Workshop Grundlagen von Datenbanken*. 71–76.

[5] Theodoros Chondrogiannis and Johann Gamper. 2016. ParDiSP: A Partition-Based Framework for Distance and Shortest Path Queries on Road Networks. In *Proc. MDM*. IEEE, 242–251.

[6] Camil Demetrescu, Andrew Goldberg, and David Johnson. 2006. 9th DIMACS Implementation Challenge: Shortest Paths. http://users.diag.uniroma1.it/challenge9/. Accessed: 07.03.2019.

[7] E.W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (1959), 269–271.

[8] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A* search meets graph theory. In *Proc. SODA*. ACM-SIAM, 156–165.

[9] Andrew V Goldberg and Renato Fonseca F Werneck. 2005. Computing Point-to-Point Shortest Paths from External Memory. In *Proc. ALENEX Workshop*. 26–40.

[10] Michael Grossniklaus, Stefania Leone, and Tilmann Zäschke. 2013. *Towards a benchmark for graph data management and processing*. Technical Report.

[11] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *Proc. CIKM*. ACM, 499–508.

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[13] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *Proc. EDBT/ICDT Workshops*.

[14] Jürgen Hölsch, Tobias Schmidt, and Michael Grossniklaus. 2017. On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database. In *Proc. EDBT/ICDT Workshops*.

[15] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proc. SIGCHI*. ACM, 1361–1370.

[16] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[17] N. Martínez-Bazan, S. Gómez-Villamor, and F. Escalé-Claveras. 2011. DEX: A high-performance graph database management system. In *Proc. ICDE Workshops*. IEEE, 124–127.

[18] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *Proc. CIKM*. ACM, 867–876.

[19] Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. 2013. Toward a Distance Oracle for Billion-node Graphs. *PVLDB* 7, 1 (2013), 61–72.

[20] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. 2014. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Trans. on Knowledge and Data Engineering* 26, 1, 55–68.

[21] Peter Rutgers, Claudio Martella, Spyros Voulgaris, and Peter Boncz. 2016. Powerful and Efficient Bulk Shortest-path Queries: Cypher Language Extension & Giraph Implementation. In *Proc. GRADES Workshop*. ACM, 6:1–6:7.

[22] Dimitris Sacharidis, Panagiotis Bouros, and Theodoros Chondrogiannis. 2017. Finding The Most Preferred Path. In *Proc. SIGSPATIAL*. ACM, 5:1–5:10.

[23] Christian Sommer. 2014. Shortest-path Queries in Static Networks. *Comput. Surveys* 46, 4 (2014), 1–31.

[24] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2017. Reachability querying: can it be even faster? *IEEE Trans. on Knowledge and Data Engineering* 29, 3 (2017), 683–697.

[25] Liying Tang and Mark Crovella. 2003. Virtual Landmarks for the Internet. In *Proc. SIGCOMM*. ACM, 143–152.

[26] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proc. CIKM*. ACM, 1785–1794.

[27] Lucien D J Valstar, George H L Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proc. SIGMOD*. ACM, 345–358.

[28] Monique V. Vieira, Bruno M. Fonseca, Rodrigo Damazio, Paulo B. Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. 2007. Efficient Search Ranking in Social Networks. In *Proc. CIKM*. ACM, 563–572.

[29] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[30] Jeffrey Xu Yu and Jiefeng Cheng. 2010. *Graph Reachability Queries: A Survey*. Springer US, 181–215.