Label Constrained Shortest Path Estimation on Large Graphs

Ankita Likhyani

IIIT-D-MTech-CS-DE-13
July 8, 2013

Indraprastha Institute of Information Technology
New Delhi

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Data Engineering

Keywords: Graph Databases, Shortest-Path, Label-constraints

Certificate

This is to certify that the thesis titled **Label Constrained Shortest Path Estimation on Large Graphs** submitted by **Ankita Likhyani** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by her under my guidance and supervision in the Information Management and Data Analytics group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Srikanta Bedathur
Indraprastha Institute of Information Technology, New Delhi

Abstract

In applications arising in massive on-line social networks, biological networks, and knowledge graphs it is often required to find shortest length path between two given nodes. Recent results have addressed the problem of computing either exact or good approximate shortest path distances efficiently. Some of these techniques also return the path corresponding to the estimated shortest path distance fast.

Many of the real-world graphs are edge-labeled graphs, i.e., each edge has a label that denotes the relationship between the two vertices connected by the edge. However, none of the techniques for estimating shortest paths work very well when we have additional constraints on the labels associated with edges that constitute the path.

In this work, we define the problem of retrieving shortest length path between two given nodes which also satisfies user-provided constraints on the set of edge labels involved in the path. We have developed SkIt index structure, which supports a wide range of label constraints on paths, and returns an accurate estimation of the shortest path that satisfies the constraints. We have conducted experiments over graphs such as social networks, and knowledge graphs that contain millions of nodes/edges, and show that SkIt index is fast, accurate in the estimated distance and has a high recall for paths that satisfy the constraints.

Acknowledgments

First and foremost, I offer my sincerest gratitude to my supervisor, Dr. Srikanta Bedathur, who has supported me throughout my thesis with his patience and knowledge while allowing me the room to work in my own way.

I dedicate this thesis to my family who have always supported me and believed that I could do it.

I would like to thank all my friends for their invaluable help and moral support.

Finally, I would like to thank Max-Planck Partner Group for the assistance and financial support that I received from them.

Contents

List of Figures

List of Tables

Chapter 1

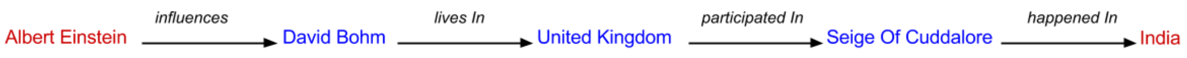
Introduction

1.1 Motivation

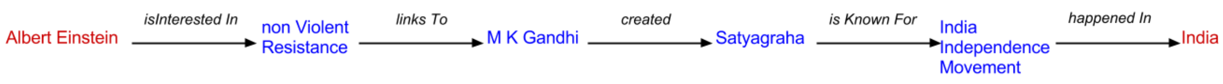
Finding shortest paths between two given nodes in a graph is a problem of fundamental importance in computer science. It has a variety of applications ranging from network-routing [11] to its use as a data mining primitive [7]. Computing shortest paths in an on-line manner for each query pair over massive graphs is common in domains such as road networks, social networks, the World Wide Web, knowledge networks etc., and is computationally challenging. This has given rise to recent research in efficiently estimating and computing the shortest path distances by pre-processing the graph. Some of these methods can be extended to generate corresponding shortest paths themselves efficiently.

As the modeling of networks gets richer, we have graphs that have certain properties associated with nodes and edges in the form of labels. For instance, in a large knowledge graph such as Yago[26], the relationship between two entities has one of many possible canonicalized relationship identifiers, and entity nodes themselves are typed (i.e., an entity "Albert_Einstein" has types "Physicist", "Vegetarian", "German", etc.). Even in social networks, it has become common to have edge-labels such as "Friend", "Colleague", "Family", and so on, to distinguish the nature of relationship between people. Further, if we consider general RDF(Resource Description Framework) databases that form the substrate for Semantic Web efforts, all relationships are *named* – i.e., contain labels.

As a natural consequence, many modern practical uses of shortest path computation demand certain constraints to be placed on the labels of edges/nodes that are involved in the path. For instance, many queries in social networks seek to discover how one person A relates to another person B. Similarly, while reasoning over large knowledge networks, one would be interested in finding connections between two entities. The commonly used shortest-path metric can return seemingly nonsense paths as we lack a way to avoid using certain edges/paths while searching paths. Consider, the shortest path between two entities Albert Einstein and India in Yago database returned by Dijkstra's:



This is an arbitrary path with path length: 4, and it may mislead the reader as Albert Einstein was against violence and war. Now, consider the following path between same entities as above:



Clearly, this path is more informative than the shortest path between Albert Einstein and India. Here, the path length is 5 which is more than the shortest path but certainly it tells us about how MK Gandhi's non-violence movement influenced Albert Einstein.

In a large knowledge database like Yago[1], a practical application would be to search for relationship, common ancestry or acquaintance, between two entities. The relation types usually consist of rather different concepts; meta-data on one hand, statistical facts and family relations on the other hand which most probably are not desired to be intermingled among each other on the shortest path.

In [2] authors have claimed that edge-label setting is useful in biological network such as graph database of KEGG[2] metabolic pathways. They have shown this using the following example:

Example 1. *Consider, a metabolic path between two compounds (C00267, alpha-D-Glucose and C00074, Phosphoenolpyruvate) which is invoked by a chain of reactions in presence of enzymes (edge-labels in a graph database). In [2] Atre et al are interested in finding out (1) if there exists path way between these two compounds for some other animal (2) and check if there exist a path between these two compounds for dog such that enzymes on the returned path way follows the regular expression of enzymes given in the query.*

The above example is a reachability query i.e. they only check for existence of a path, and do not return the path. In this work, we are interested in processing queries to allow or disallow certain edge-labels on the returned path. This will help us in finding informative and insightful paths between two entities. The purpose of this work is to develop an indexing framework and an on-line algorithm, to determine shortest length path between two given nodes which also satisfy the user-given constraints on the edge labels involved in the path.

1.2 Terms and Definitions

We use edge-labeled directed as well as edge-labeled undirected graphs, i.e. a graph G is tuple $G = (V, E, \Sigma)$, where V is a finite set of nodes, E is finite set of edges, and Σ is finite set of edge-labels. An edge is represented as :

$$e \in (V \times \Sigma \times V)$$

Here, the components are from-node, edge-label and to-node respectively. The vertices are called v_i where $i \in 1, ..., |V|$ and the edge-labels are called σ_i where $i \in 1, ..., |\Sigma|$.

A path p from vertex u to v is an sequence of (distinct) vertices, $p = (u, v_1, ..., v_{i-1}, v_i, ..., v)$, and an edge between v_i and v_{i+1} is denoted as $e_{i,i+1}$.

Definition 1. Shortest Path: Given a labeled-graph $G = (V, E, \Sigma)$ directed or undirected, for a source vertex $s \in V$ and a target vertex $t \in V$, the shortest path from s to t is the path

[1]http://www.mpi-inf.mpg.de/yago-naga/yago/
[2]Kyoto Encyclopedia of Genes and Genomes http://www.genome.jp/kegg/

$p = (v_1, ...v_n)$ (where, $v_1 = s$ and $v_n = t$) that over all possible n minimizes the sum $\sum_{i=1}^{n-1} f(e_{i,i+1})$, where $f : E \to \{1\}$ i.e. each edge in the graph has unit weight or, this is equivalent to finding the path with fewest edges.

Definition 2. Label Constrained Reachability(LCR): Given labeled-directed or undirected graph $G = (V, E, \Sigma)$, two vertices u and v and a label set C, where u, $v \in V$ and $C \subseteq \Sigma$, if there exists a path p from vertex u to v whose path labels $L(p)$ is a subset of C, i.e., $L(p) \subseteq C$, then we say u can reach v with label-constraint C, denoted as $u \xrightarrow{C} v$, or simply v is C-reachable from u. We also refer to path p as a C-path from u to v. Given two vertices u and v, and a label set C, the LCR query asks if v is C-reachable from u.

1.3 Problem Statement

In this work, we consider how to answer Label Constrained Shortest Path queries efficiently. In particular, we have two forms of constraints on the edge-labels:

Positive label restrictions: The set of edge-labels on the qualifying paths should be a subset of the specified set of edge-labels.
Negative label restrictions: As opposed to above, user specifies the set of labels that must not appear on the qualifying paths.

Label Constrained Shortest Path(LCSP) problem over G is defined as follows:

Definition 3. Label Constrained Shortest Path(LCSP): Given an edge-labeled directed/undirected graph, $G = (V, E, \Sigma)$, a source vertex s, a target vertex t and an edge-label constraint set $C \subseteq \Sigma$, a Label Constrained Shortest Path, $LCSP(s, t, C)$ is given by the shortest path p between s and t such that path labels $L(p) \subseteq C$ in case of positive label restriction or $L(p) \subseteq \bar{C}$ in case of negative label restriction.

From the definition above, it is clear that the edge-label positive-restriction and negative-restriction can be treated uniformly. It is also quite straightforward to see that, in practice, the value of $|C|$ is smaller in case of positive-restriction than in the edge-label negative-restriction setting.

1.4 Problem Difficulty

It seems quite straightforward to incorporate such label constraints over an on-line shortest path algorithm such as the Dijkstra's algorithm. These algorithms, with some carefully designed optimizations such as A*-search [7, 11], find favor in answering similar queries that are encountered in road networks for route planning. But social networks and knowledge networks that we consider in our work, are quite different from road networks – they lack near-planarity, hierarchical structure, and low-degree nodes that are critical for the efficiency of algorithms over road networks.

A*-search explores the graph by expanding the most promising node according to some specified rules. In social networks there are no hierarchies and edge-labels are also treated uniformly, so no rules are specified. Thus, A*-search will become similar to constrained Dijkstra's algorithm(i.e. only over allowed edges) for these networks. In other words, A*-search has to examine all allowed edges in the worst case. Although, introduction of label-constraints to Dijkstra's algorithm, favours in terms of computational complexity because less number of nodes are required to be processed now, since the edges that do not satisfy the label conditions are simply ignored. Thus, only a sub-graph is processed that contains edges which satisfies the label condition. But considering the size of graph we have considered in our experiments, this will still be large in number. Note that, the overhead of checking label condition can be safely ignored.

1.4.1 Precomputing Label-Constrained Shortest Paths

Let us consider the problem of computing shortest paths for each and every pair of nodes and with all possible label constraint combination, this corresponds to generalized transitive closure. The pre-computation of all-pair shortest paths using Floyd-Warshall algorithm on unlabeled graph can be materialized in $O(|V|^2)$ space. In case of edge-labeled graph for different possible constraint set i.e. for each possible subset of $|\Sigma|$, we require $O(2^{|\Sigma|} * |V|^2)$ space. This would also mean that the complexity of running time would increase by a factor of $2^{|\Sigma|}$ because in worst case for each $C \subseteq \Sigma$ the shortest path between s and t will be different that respects C.

1.4.2 Label Constrained Reachability vs Label Constrained Shortest Path

In this section, we discuss that the pre-processing required to answer LCR efficiently is not sufficient to answer constrained shortest path efficiently. Jin et al in [19] introduced the Label Constrained Reachability (LCR) problem, which states that a node v is reachable from another node u with edge labels on the path, are a subset of user provided constraint set. In [19] authors use generalized transitive closure and approximate maximal spanning tree (DAG) to store the edge label information and at the time of query processing they use interval labeling and kd-trees(range search trees)[9] to answer the query.

Atre et al in [2] have gone a step ahead and have introduced Labeled Order Constrained Reachability problem i.e. they not only consider the containment but also the order in which the edge-labels occur on the path between u and v. In [2] authors first transform the graph into DAG by collapsing the strongly connected components(SCC) and then perform DFS over DAG to create BitPath indexes. The BitPath indexes can also be considered as equivalent to interval labeling containing information of successors, predecessors and edge label information of all the nodes with respect to root node of DAG.

With the above two approaches, we will loose path information either by creating approximate maximal spanning tree as stated in [19] or by collapsing the SCC components into a graph as stated in [2]. In order to answer LCSP queries using reachability approach, one way is to check if a node should be expanded using the index before expanding it in Dijkstra's. This will have worst case complexity of $O(|V| + |E|)$.

1.4.3 Conclusion

Clearly, storing all-pair shortest path for all possible combinations of constraints will be expensive computationally as well as storage-wise and reachability index does not help in answering path queries because these are generally light-weight indices and for answering path queries will require look up in original graph that will have high computational cost as discussed in previous section. To answer path queries efficiently commonly used methods are graph based embedding techniques[10, 23]. The embedding technique used here is selection of few nodes as reference nodes or landmarks out of all the nodes and computing offline the distances from each node in the graph to those landmarks. At run time, when the distance between a pair of nodes is required, it can be estimated quickly by combining the precomputed distances. This basic idea is further improved to also generate some paths corresponding to the estimated shortest distance [17]. Thus, in this work we extend the landmark-based path-sketches with label information to answer LCSP queries efficiently.

1.5 Contributions

In this work, we consider how to answer LCSP queries efficiently by augmenting the landmark-based path-sketches [17].

1. We have proposed and developed SkIt(Sketch augmented with inverted indices), that enables us with an efficient estimation algorithm for edge-label constrained shortest paths between two given nodes of an entirely disk-resident graph. A powerful feature of the proposed solution is its ability to handle *arbitrary label constraints* on the edge-labels. In particular, we show how SkIt can efficiently support the following forms of constraints on the edge-labels:

   Label white-listing: The set of edge-labels on the qualifying paths should be a subset of the specified white-list of edge-labels.
   Label black-listing: As opposed to above, user specifies the set of labels that must not appear on the qualifying paths.

2. In order to empirically establish the efficiency of SkIt, we implement it within the same framework that was used in path-sketches [17] against which we compare the performance of our index. Specifically, we implement label-augmented landmark-based sketches within the RDF3x database[22], and maintain the inverted index that is used to filter out the partial exploration required by the TreeSketch algorithm [17]. We compare the performance of SkIt against standard TreeSketch that approximately enumerates the paths in increasing order of their distance, and also against the path query performance in Neo4J[1] – a high-performance, industry-standard graph data management system. As a baseline, we also implement the standard Dijkstra's algorithm which can trivially support all forms of label constraints as it explores the graph. Our evaluation using multiple large-scale labeled graphs show that the use of SkIt makes complex Label-Constrained Shortest Path discovery highly scalable.

3. As an extension of our work:

   (a) We have defined different classes of queries that SkIt can support and have tested on label counting based constraints i.e. with each edge-label in constraint set C we also mention a count k such that this edge-label should not occur more than k times in the path.

   (b) We have also tested ranking extension of constraint satisfying shortest paths, based on the number of constraints they satisfy. We look at a weight function $W : l \to w$ which maps a label l to a positive real weight w, and rank paths based on the weights induced by this weight function. Currently, we rank all the paths computed by our on-line algorithm.

1.6 Outline

The remainder of the thesis is structured as follows. In Chapter 2, we discuss related work done and some important ideas used in the thesis. In Chapter 3 we explore and propose different indexing framework and analyse their applicability for answering LCSP queries efficiently. We also discuss in detail how path-sketches are augmented with edge-labels to result in SkIt structure. In chapter 4 we first define different classes of queries that SkIt can support, then we describe the constrained tree sketch algorithm(extended from tree-sketch algorithm in [17]) that encompasses the developed SkIt structure to answer LCSP queries. Following this, in Chapter 5 we describe our experimental setup including data sets and queries, and present the results. Finally, we conclude in Chapter 6.

Chapter 2

Related Work

In this thesis, we focus on answering LCSP queries efficiently. There is a significant amount of work done in areas such as shortest path queries, regular path queries and constrained reachability. In this chapter we first introduce some of the well-known shortest path algorithms in section 2.1. We then give an outline of current state-of-the-art algorithms to solve or approximate the point-to-point shortest path queries, constraint reachability queries, constrained shortest path queries and regular path queries in section 2.2 and 2.3.

2.1 Single Source Shortest Path Algorithms

2.1.1 On-line Computation

These are among the most important classical shortest path algorithms. In this section V denotes the number of vertices and E the number of edges in a graph. The vertices are called v_i where $i \in 1, ..., V$.

- **Dijkstra's algorithm** [9] is the classical solution to compute single-source shortest paths on arbitrary graphs (non-negative edge weights provided). It computes a shortest path tree by visiting nodes in ascending order of their distance from the source node, relaxing all of its outgoing edges. Once a node is being visited, its distance from the source is known to be minimal. Thus the algorithm terminates if the target node is being visited. By saving the predecessor of each visited node (and updating it when relaxing an edge) the algorithm outputs not only the distances but also the shortest paths itself. Its worst case running time is $O(E + V logV)$.

- **A^*-search** [18] uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A^* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

- **ALT** are based on A^*-search, landmarks and the triangle inequality. Goldberg et al in [15] proposed this first and have shown that this outperforms previous algorithms, in particular A^*-search with Euclidean bounds, by a wide margin on road networks and on some synthetic problem families.

2.1.2 Precomputed

- **Floyd Warshall algorithm** [9] computes all-pairs shortest paths and stores distances (and optionally also path information) between any two nodes in $O(V^3)$ time and $O(V^2)$ space. It uses dynamic programming to save all-pairs distances considering only paths via vertices $v_1, ..., v_k$ and increasing k step-wise from 1 to V .
- **The Bellman Ford algorithm** [9] The algorithm by Bellman and Ford computes single-source shortest paths by initially setting all distances except for the source nodes to infinity and then repeatedly relaxing all edges. It does this $V-1$ times which guarantees optimality as in the absence of negative cycles a shortest path can visit each node only once. Its running time is $O(VE)$ in the worst case.

Although there exist exact algorithms for computing shortest paths, these algorithms cannot be adopted for real world massive graphs, especially in on-line applications where query response time must be in few milliseconds. This calls for approximation algorithms. In next section we discuss about approximate algorithms used for computing shortest paths in large graphs.

2.2 Single Source Shortest Path Estimation

The problem of answering shortest path distance queries approximately on large undirected graphs has been studied theoretically in [6, 8, 13, 5, 29]. Thorup and Zwick in [29] introduced a distance oracle that answers such a query in approximately $O(k)$ time with a theoretical upper bound on the approximation ratio of 2k-1.

There have been methods[23, 10, 17, 24] for estimating shortest path distances between two vertices using graph based embedding techniques. The embedding technique used here is selection of few nodes as reference nodes out of all the nodes and projecting distances from these nodes to all other nodes.

Potamias et al in [23] has performed empirical studies on the approximation quality of landmark-based shortest path algorithms on several real-world social network databases. They focus on the efficiency of different landmark selection methods.

Das Sarma et al in [10] have shown that using randomly sampled set of landmarks; as such, works very well on complex graphs such as the web graph. In offline computation they sample a set of seed nodes, and store the closest seed from every node along with its distance and have termed it as sketch algorithm, then do simple computation to estimate the distance. They have tested their algorithm on a partial crawl of the web graph, both directed as well as undirected.

Gubichev et al in [17] have extended the sketch algorithm above to return not only an approximate distance but also estimate the paths. Moreover they use the path information in the sketch to improve upon the distances. The extended sketch algorithm to compute path-sketches is described in more detail in Chapter 3.

Recently, [24] Qiao et al have proposed coverage-based landmark selection approach to answer shortest path queries with error bounds. They have used graph partitioning based heuristics to reduce the offline computational complexity and have compared their results with centrality based reference node selection approach proposed in [23] by Potamias et al, but the approximation quality factor of returned paths using centrality based approach is better than partitioning based approach. Although, they have shown that without graph partitioning they achieve better path quality but pre-processing becomes computationally expensive.

2.3 Constrained Reachability and Single Source Shortest Path

2.3.1 Reachability

Jin et al.[19] devised an algorithm that precomputes a spanning tree of the graph and a large index that accounts for all essential non-tree paths to allow fast reachability queries with label constraints on runtime. They have tested their algorithm on some synthetic and real-world social network graphs with maximum of 100,000 nodes and 150,000 edges.

In [30] Xu et al have extended the work by Jin et al in [19] and have theoretically proposed optimization for path-label transitive-closure computation. They have also addressed the scalability issue, but have considered maximum of 200k nodes which is still small in number as compared to what we have considered for our experiments.

Atre et al [2] have proposed a light-weight indexes on graphs using compressed bit-vectors and divide-and-conquer algorithm along with greedy-pruning strategy to answer Label-Constrained ordered queries. They have evaluated their method over a subset of large real data set graphs, which is created by collapsing strongly connected components. They do not consider the whole graph in their work.

2.3.2 Shortest Path

Barrett et al in [3, 4] have considered several more general forms of the constrained shortest path problem, e.g. the language-constrained shortest path problem that accepts a path only if the edge labels along it belong to a regular language. They studied the problem of linear regular expression (LRE) constrained shortest path problem [3] which imposes the order in which labels may appear along a path. The algorithms for the latter two have been tested on several road and railway networks.

Rice et al.[25] extended the contraction hierarchies introduced in [14] to support dynamic label constraints and were able to show an improvement of query speed of 3 orders of magnitude compared to Dijkstras algorithm on the North American road network.

A recent work [20] by Kirchler et al have proposed an algorithm SDALT (State Dependent uniALT) adaptation of the speed-up technique uniALT in order to accelerate generalization of Dijkstra's algorithm to solve regular language constrained shortest path problem. They have shown an improvement of factor 2 to 20 with respect to Dijkstra's algorithm over road and public transportation network of the French region.

2.3.3 Regular Path Acceleration

Fan et al [12] have addressed the problem of adding regular expressions and patterns to the reachability queries. They have evaluated their algorithm on the graphs of up to 1 million nodes and 4 million edges (synthetic).

Gubichev et al [16] have implemented a technique of evaluating path queries over RDF graphs using purely database style indexing and efficient join processing techniques. Although they have shown experiments on very large RDF graphs, their queries use more join-like expressions than rich path-patterns.

M. Zhou et al [31] have proposed a Constrained Acyclic Path(CAP) problem. To express CAP search queries they have proposed cSPARQL an extension of SPARQL. They have basically addressed search queries problems that are used to express complex patterns that satisfy constraints on nodes and edges.

Koschmieder et al in [21] have addressed the problem of answering regular path queries on large graphs. Their main idea is based on the fact: to structure a graph traversal along those labels from a query that are infrequent in the graph. They use these labels in parallel to process the query and then merge the results to obtain the paths. They have shown that this considerably improves scalability with regard to the size of graph. Although their entire optimization depends on this assumption that these rare-labels are guaranteed to occur in matching path.

Conclusion: All the previously done work, to the best of our knowledge do not address the problem of answering constrained shortest path queries on large social and knowledge networks. Barrett et al in [3, 4] and Rice et al in [25] have considered answering constrained shortest path but on road networks which provides them to exploit hierarchical features absent in social and knowledge graphs.

Since, real world graphs are generally edge labeled these days, and shortest path problem is a fundamental problem to be addressed on such graphs. To answer such queries and considering the size of graphs, this calls for algorithms with moderate precomputation cost and index size, yet still fast query time.

Chapter 3

Index Framework

In this chapter we discuss about two indexing framework in section 3.1.1 and 3.2 that were proposed and proved to be in-optimal for large graphs. In between the above mentioned section we give brief introduction about path-sketches and TreeSketch algorithm in section 3.1.2 and 3.1.3, respectively. In section 3.3 we propose and discuss in detail the construction of SkIt which we have used to answer LCSP queries efficiently.

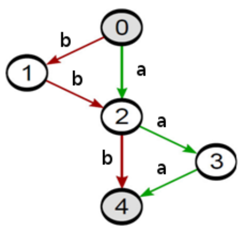
3.1 Background

3.1.1 PCA based Landmarks

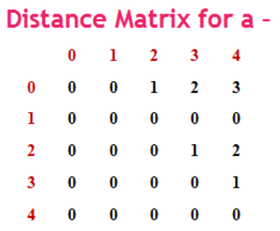
In this section, we propose an approach adapted from [27] and [28]. The Virtual Landmarks method [27] is based on two ideas: First, it uses a Lipschitz embedding [6] of nodes into a high dimensional space. In a Lipschitz embedding, the distances to a set of landmarks are taken as the coordinates of the given node. Second, it uses dimensionality reduction via Principal Component Analysis(PCA) to reduce the higher-dimensional space of Lipshitz embedding to a lower-dimensional space. However, [27] did not address the question of landmark selection explicitly. In [28] authors have explored different landmark selection techniques over geometrical setting.

Here, we have proposed an indexing scheme which utilizes PCA to reduce higher dimensional distance matrix and k-means based landmark selection approach proposed in [28] to select the landmarks.

- First, we construct distance matrices for all combinations of labels(distribution over edges) as shown in figure 3.1.
- We compress each distance matrix using PCA(Prinicipal Component Analysis). By applying PCA to above matrices, we can do dimensionality reduction, while approximately preserving the distances.
- After this we plot, top 2-principal components along X-axis and Y-axis in a geometric plane and perform k-means algorithm to select landmarks.
- Once landmarks are obtained we can build path-sketches(discussed in next section) using them.

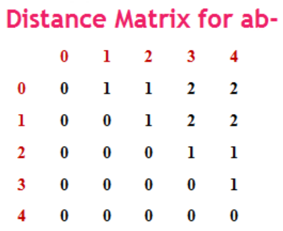


(a) sample graph



Distance Matrix for a -

	0	1	2	3	4
0	0	0	1	2	3
1	0	0	0	0	0
2	0	0	0	1	2
3	0	0	0	0	1
4	0	0	0	0	0



Distance Matrix for ab-

	0	1	2	3	4
0	0	1	1	2	2
1	0	0	1	2	2
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	0

(b)Distance matrices for C = {a} and C = {a,b}

Figure 3.1: Distance matrices for different constraint sets

Complexity of PCA is $O(nD^2)$, where n is the input data and D is the number of dimensions. For a distance matrix since D is same as n, thus the computational complexity becomes $O(n^3)$ in our setting. Hence, this exhaustive approach is not practical for large-graphs with number of nodes and edge labels.

3.1.2 Path-sketches

One of the most popular shortest path estimation techniques over directed, unweighted graphs (such as those we consider here), is the landmark-based shortest distance oracles [29]. Although there are many different variants, the underlying idea is as follows: We first select a set of special nodes, L, which we call *landmarks*, and compute shortest path distances to/from each of the landmarks to all other nodes in the graph. We associate a vector of these distances along with the corresponding landmark with every node, and call this a *shortest-path distance sketch* of the graph. After this pre-computation, we can estimate the shortest path between any two nodes u and v using triangle inequality as follows:

$$\hat{\delta}(u,v) = \min_{l \in L} \delta(u,l) + \delta(l,v),$$

where $\delta(x,y)$ denotes the directed shortest distance between x and y. One variant of this method [10], proposed to use set-based landmark selection, where we sample landmark sets of exponentially increasing sizes $S_1,\ldots,S_r$ with $r = \log(n)$. For each such set, every vertex v maintains two sketch entries: the distance to the *closest* landmark node in the set, and the distance from a landmark node from which v is at shortest distance. Formally, the two entries are

$$\begin{aligned} &(f_i, \delta(v,f_i)) \quad \text{where} \quad f_i = \arg\min_{x \in S_i} \delta(v,x) \\ &(b_i, \delta(b_i,v)) \quad \text{where} \quad b_i = \arg\min_{x \in S_i} \delta(x,v) \end{aligned}$$

The sketch for a vertex v is:

$$\begin{aligned} s_f(v) &= \{(f_i, \delta\,(v,f_i)) \mid i = 1\ldots r\} \\ b_f(v) &= \{(b_i, \delta\,(b_i,v)) \mid i = 1\ldots r\} \end{aligned}$$

as the two sketch entries corresponding to the *forward* path from the node to a landmark set and the *backward* path from a landmark to the node respectively. Denoting the landmark nodes that appear in the forward and backward sketches of a node v as $L_f(v)$ and $L_b(v)$, respectively, we can write the shortest path distance estimator between two nodes u and v as,

$$\hat{\delta}(u,v) = \min_{x \in L_f(u) \cap L_b(w)} \delta(u,x) + \delta(x,v).$$

Note that these distance sketches maintain only the shortest distance between landmark nodes and regular nodes in the graph. Therefore, in order to reconstruct the actual path that corresponds to the estimated distance between the two nodes, additional accesses to the graph are required. This requirement is lifted by *path-sketches* [17] where the sketch also contains the shortest path between the landmark node and a regular node. That is, the path-sketch entries are of the form

$$s_f(v) = \{(f_i, \delta(v, f_i), p(v, f_i)) \mid i = 1 \dots r\}$$
$$b_f(v) = \{(b_i, \delta(b_i, v), p(b_i, v)) \mid i = 1 \dots r\}$$

More significantly, based on this additional information, it is possible to improve the accuracy of the shortest path distance estimation significantly through the use of a series of improvements culminating in a bounded path computation algorithm called *TreeSketch* over the tree resulting from the union of paths stored in the sketches. In next section we briefly describe *TreeSketch* algorithm used for obtaining the actual path.

3.1.3 TreeSketch Algorithm

Forward and backward sketches are loaded from disk to construct trees rooted at source node and target node, respectively. Trees are constructed by doing union of the paths in their sketches. After that, for every node in the tree, *TreeSketch* performs a k-hop BFS (typically $k = 1$) to see if it possible to reach the target with a shorter distance. Although TreeSketch requires additional accesses to the underlying graph, it has been shown to be relatively inexpensive even when the graph is entirely disk-resident [17], and can generate almost accurate distance estimates.

3.2 Adapting Minimal Sufficient Path Label Set

In this framework, we augment path-sketches proposed in [17] with Minimal Sufficient Path Label Set with each path. Lets understand first Minimal Sufficient Path Label Set.

3.2.1 Minimal Sufficient Path Label Set

Definition 4. Sufficient Path-Label Set: Let S be a set of path labels from vertex u to v. Then, we say S is a Sufficient Path Label set if for any label-constraint set C, u $\xrightarrow{C}$ v, the LCR query returns true iff there is a path-label s ∈ S, such that s ⊆ C.

Let $M(u, v)$ denote minimal sufficient path label set of paths from u to v and $M_k(u, v)$ denote the minimal sufficient path-label set of those paths from u to v whose intermediate vertices are in v_1,..., v_k. The minimal sufficient path-label sets of the paths from each vertex u to v with intermediate vertices up to v_{k+1}, i.e., M_{k+1}(u, v) is computed as follows:

$$M_{k+1}(u,v) = Prune(M_k(u,v) \cup (M_k(u,k) \odot M_k(k,v)));$$
$$M_0(u,v) = \begin{cases} \emptyset & \text{if } (u,v) \in E \\ \lambda((u,v)) & if (u,v) \in E \end{cases} \tag{3.1}$$

Here, *Prune* is a function which simply drops all the path labels that are supersets of other path-labels in the input set. The ⊙ operator joins two sets of sets, such as { s_1, s_2 } ⊙ { $s_1^{'}$, $s_2^{'}$, $s_3^{'}$ } = { $s_1 \cup s_1^{'}$, $s_1 \cup s_2^{'}$, ... , $s_2 \cup s_3^{'}$ }, where s_i and $s_j^{'}$ are sets of labels.

To answer LCSP queries we need to maintain label information with path-sketches. Thus we came up with the solution that, ∀ v ∈ V and 2rk landmarks, M(v, l) can be computed using generalized transitive closure, where l ∈ 2rk landmarks. Moreover, as demonstrated in [19], in case of reachability, the number of constraints that need to be saved can be reduced by considering the minimal sufficient path-label set which is based on the following observation.

Theorem 1. *If a target node is reachable under constraint set $C_1 \subset \Sigma$, then it is reachable under each constraint set C_2 such that $C_1 \subset C_2 \subset \Sigma$.*

This helps in saving space but it does not improve the running time as paths using the superset may be discovered before discovering the path using the subset. As an example consider the path $(0 \to 2 \to 4)$ in the unweighted graph in figure 3.1(a), for constraint set : $\{a, b\}$ it is discovered before the path $(0 \to 1 \to 2 \to 4)$ that uses constraint set : $\{b\}$ which is a subset of the former. It turns out that Theorem 1 does not extend to the shortest path problem: Given a path using a constraint set C_1 there may be a shorter path using a constraint set $C_2 \supset C_1$. The same holds for the path $(0 \to 2 \to 3 \to 4)$ that uses constraint set : $\{a\}$. In total we have $2^{|\Sigma|}$-$1 = 3$ paths that we would need to maintain for node 4 if the source is node 0. In particular we observe that for any possible constraint there is a different shortest path from 0 to 4. This implies the following.

Theorem 2. *Given graph $G = (V, E, \Sigma)$ and vertices s and t, in the worst case for each constraint $C \subset \Sigma$ there is a different shortest path from s to t that respects C.*

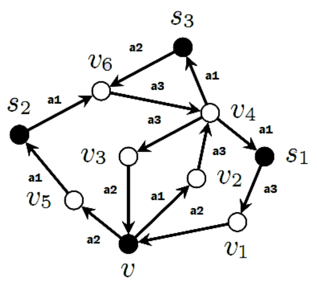
The above result suggests that computing the shortest paths for all constraints basically requires computing shortest paths for each constraint individually. This would mean that the complexity of running time would increase by a factor of $2^{|\Sigma|}$ as compared to the underlying shortest path algorithm. For a single-source, Dijkstra's like in this example would result in a running time of $O((VlogV + E)2^{|\Sigma|})$. Moreover it follows directly from the theorem that the storage space to save shortest paths for all constraints increases by the same factor of $2^{|\Sigma|}$ in the worst case.

3.3 The SkIt Index

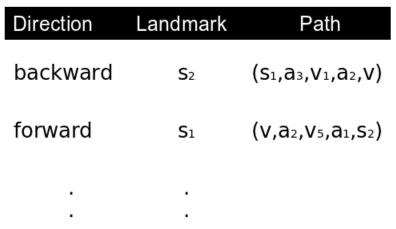
Here, we present a framework that integrates the sketch index and the inverted index into a new index, that we call the **Sk**etch **I**nver**t**ed Index (SkIt). The sketch index here is the path-sketches proposed in [17], that are augmented with the edge label information. Further, in order to construct the actual path during on-line execution we utilize the inverted list structure. We have discussed them as separate parts in following two sections:

3.3.1 Label Path Sketches

As a first step towards extending path-sketches to support the edge-label constraints on the paths, we augment the path information with the edge-label information. We do not incur any extra computation cost to construct labeled-path-sketches. Although, the increase in size of index is by a factor of 1.2 to 2.3 after incorporating the label information with path. This fairly straightforward extension is illustrated using the example graph in Figure 3.2, along with the associated sketch. Consider node v in this figure, it is at shortest distance from node s_1 and s_2 in forward and backward direction respectively. Thus, s_1 and s_2 are landmarks for node v. Note that, in labeled-path-sketches index the labels are stored in sequence, therefore this index scheme can be used for answering edge labeled ordered queries/ regular expression queries as mentioned in chapter 4.1.



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



Direction	Landmark	Path
backward	s_2	(s_1,a_3,v_1,a_2,v)
forward	s_1	(v,a_2,v_5,a_1,s_2)
.	.	
.	.	

(b) Resulting entries in Labeled-Sketch(v)

Figure 3.2: Labeled-path-sketch construction

node	Label with posting	
s_1	$a_3 : v_1$	
s_2	$a_1 : v_6$	
s_3	$a_2 : v_6$	
v_1	$a_2 : v$	
v_2	$a_3 : v_4$	
v_3	$a_2 : v$	
v_4	$a_1 : s_3, s_1$	$a_3 : v_3$
v_5	$a_1 : s_2$	
v_6	$a_3 : v_4$	
v	$a_1 : v_2$	$a_2 : v_5$

(a) Forward Inverted Index

node	Label with posting	
s_1	$a_1 : v_4$	
s_2	$a_1 : v_5$	
s_3	$a_1 : v_4$	
v_1	$a_3 : s_1$	
v_2	$a_1 : v$	
v_3	$a_3 : v_4$	
v_4	$a_3 : v_2, v_6$	
v_5	$a_2 : v$	
v_6	$a_1 : s_2$	$a_2 : s_3$
v	$a_2 : v_1, v_3$	

(b) Backward Inverted Index

Table 3.1: Inverted Indices

3.3.2 Label Inverted Index

During the on-line execution, for every node we perform a k-hop exploration of the graph in order to find a shorter connection between the forward and backward sketch trees (discussed in detail in chapter 4). However, this may lead to many wasted accesses to the graph since they may violate label constraints.

We utilize a compact inverted list structure that, for each node, maintains the list of nodes directly reachable, as well as the list of nodes of that reach the current node, via an edge-label. In other words, each node will have a *forward edge-label* and a *backward edge-label* inverted list. This can be seen as an effective organization of the edge-labeled adjacency list of the graph so as to efficiently decide if a specific node can be expanded or not. We have illustrated the inverted list structure using following example for a sample graph in figure 3.2.

Example 2. *Table 3.1 (a) captures the information of successors for a node and table 3.1 (b) captures the information of predecessors for a node. Consider node v_4 in figure 3.2(a) it has successors s_1, s_3 and v_3 associated with edge label a_1 and a_3, respectively and predecessors v_2, v_6 associated with a_3.*

Chapter 4

Query Processing

In this chapter we first discuss in section 4.1 the classes of queries SkIt can support. Then, we proceed with the discussion of on-line algorithm: Constrained Tree Sketch algorithm in detail in section4.2, which makes use of our proposed index SkIt to answer LCSP queries efficiently. The proposed algorithm is an extension of TreeSketch algorithm proposed in [17].

4.1 Classes of queries

The goal of our work is to extend sketch algorithms that estimate the shortest paths between two nodes with different kinds of label constraints on the retrieved path. In this section we discuss about the usefulness of SkIt by defining different classes of queries that SkIt can support.

The label constraints that we could consider are:

Positive label restrictions Given a set of edge-labels $C \subseteq \Sigma$, the path label set $L(P) \subseteq C$ For example, in a social network, if $C = \{\mathsf{family}, \mathsf{friend}\}$ then we are looking for connections consisting of only family or friend relations i.e. $L(P)$ can be $\{\mathsf{family}\}$ or $\{\mathsf{friend}\}$ or $\{\mathsf{family}, \mathsf{friend}\}$. We do not allow for connections that have either acquaintance or shared interest relationships.

Negative label restrictions Given a set of edge-labels $C \subseteq \Sigma$, the path label set $L(P) \nsubseteq C$

Positive exhaustive label restrictions Given a set of edge-labels $C \subseteq \Sigma$, the path label set $L(P) = C$, i.e *all labels must be covered by the path.*

Soft label restrictions Given a set of edge-labels $C \subseteq \Sigma$, the path label set $L(P)$ should have at most 1 label from C. There is no restriction on labels that are not specified.

Label order constraints Given a sequence of labels, the path should have labels in the same order, but there could be other labels in the sequence. For instance, if the sequence given is $< l_1, l_2, l_3 >$, then a path with label sequence $< l_0, l_1, l_1, l_4, l_2, l_3 >$ is allowed.

Regular path constraints Given a regular expression over labels as described in [2], the path should satisfy the regular expression

Note that, in our experiments we have tested SkIt for Positive label restrictions and Negative label restrictions. But clearly, all these restrictions can be incorporated within SkIt for finding shortest paths, since edge-labels are stored in sequence in labeled-path-sketches. Moreover, while constructing actual paths during on-line computation we have access to the neighbors via edge-labels.

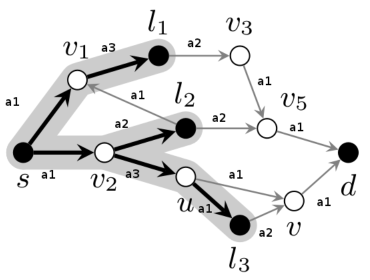
4.2 Constrained Tree-Sketch Algorithm

In original Tree-Sketch algorithm only s-node and t-node are provided as input, here we also provide constraint set C as input with other inputs. In its first step, the algorithm loads all the labeled-path-sketches(disk-resident) for the two given pair of vertices s and t(into main memory). After loading the forward and backward labeled-path sketches, we can proceed to estimate the LCSP by rejecting all sketches that contains paths which does not satisfy the specified edge-label constraints, and similarly rejecting total distance estimations if the corresponding path violates the constraints.

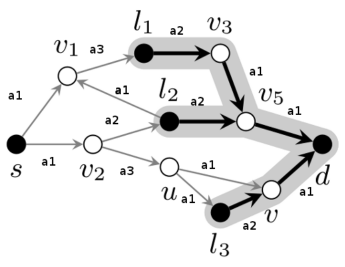
However, this simplistic approach may result in not finding any path between two nodes since we are pruning out candidate paths. For instance, consider the backward path-sketch starting from l_3 to d in figure 4.1, whose edge-label set is $\{a_1, a_2\}$. If the user specified $C = \{a_1, a_3\}$ then rejecting the path-sketch entirely would result in not finding the LCSP between s and d, although TreeSketch allows for it. On the other hand, retaining all the sketches which violate the constraints also is not desirable as it adds needless exploration steps. We trade off these two aspects by truncating the paths from sketches to a prefix (suffix for backward paths) that satisfies the edge-label constraints. For instance, considering the same example, we will truncate the sketch $\langle l_3, a_2, v_6, a_1, t\rangle$ to $\langle v_6, a_1, t\rangle$. Using TreeSketch, we can now find the LCSP between the query vertices.

After obtaining edge-label constrained sketch tree, denoted as CT_s and CT_t, bidirectional BFS from s and t is started. For every visited node the list of its neighbours is loaded from the inverted indices. For every visited pair $(x, y), x \in CT_s, y \in CT_t$ the algorithm thus can check whether x and y are equal. If yes, we construct the path from s to t by concatenating the paths from s to x and from y to t. Note that, neighbours up to 2-hop for visited nodes are loaded, thus we obtain path concatenation up to 3-hops. Whereas, in earlier tree sketch algorithm it was just 1-hop(immediate neighbors) look up. Note that, by further increasing the look ups we will end up processing lot of nodes thus increasing the query processing time. We trade-off between accuracy and query processing time by doing 2-hop look ups. With this, we are able to achieve good accuracy within reasonable processing time as shown in chapter5.

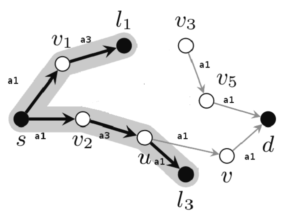
The pseudo code for the Constrained Tree Sketch Algorithm is provided in Algorithm 1.



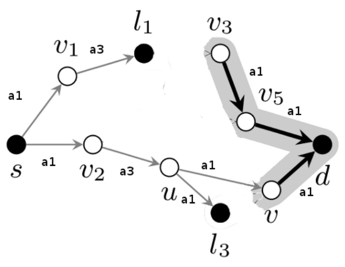
(a) Tree sketch on edges rooted at s



(b) Tree sketch on reversed edges rooted at d



(c) Constrained Tree sketch on edges rooted at s



(d) Constrained Tree sketch on reversed edges rooted at d

Figure 4.1: Path truncation

Algorithm 1: Constrained Tree-Sketch Algorithm

```
   Input: source node $s \in V$ , target node $t \in V$ , label constraint $C \subset \Sigma$
   Result: $Q$, priority queue of paths from $s$ to $t$ respecting $C$
1  $CT_s \leftarrow$ tree of paths from $s$ that respect $C$ (built from sketch);
2  $CT_t \leftarrow$ tree of paths to $t$ that respect $C$ (built from sketch);
3  $Q \leftarrow \emptyset$;
4  $\mu \leftarrow \infty$;
5  $NBFS \leftarrow \emptyset, NRBFS \leftarrow \emptyset$;
6  for all $u \in BFS(CT_s, s)$ and $v \in BFS(CT_t, t)$ do
7      $NBFS \leftarrow NBFS \cup \{u\}$;
8      $p_{v \to t} \leftarrow$ path from $v$ to $t \in CT_t$;
9      for all $x \in NBFS$ do
10         if $v \in upto2HopSuccessors(x, C)$ then
11             $p \leftarrow p_{s \to x} \circ p_{x \to v} \circ p_{v \to t}$;
12             $Q \leftarrow Q \cup \{p\}$;
13             $\mu \leftarrow min\{\mu, \mid p \mid\}$;
14     $NRBFS \leftarrow NRBFS \cup \{v\}$;
15     $p_{s \to u} \leftarrow$ path from $s$ to $u \in CT_s$;
16     for all $x \in NRBFS$ do
17         if $x \in upto2HopSuccessors(u, C)$ then
18             $p \leftarrow p_{s \to u} \circ p_{u \to x} \circ p_{x \to t}$;
19             $Q \leftarrow Q \cup \{p\}$;
20             $\mu \leftarrow min\{\mu, \mid p \mid\}$;
21     if $dist(s, u) + dist(v, d) \geq \mu$ then
22         return ;
```

Chapter 5

Experiments and Results

In this chapter we provide an experimental evaluation of SkIt. In order to empirically establish the efficiency of SkIt, we have implemented it within the same framework that was used in path-sketches [17] against which we compare the performance of our index, discussed in section 5.1. Then, we give an overview of the data sets used in section 5.2. In section 5.3 we describe the generation of test instances used in the subsequent evaluation. Performance metrics for evaluation are described in section 5.4, subsequently followed by results section 5.5. In addition we have tested our framework on label-count-queries in section 5.6. We conclude with section 5.7 where we have done worst case analysis, if paths are ranked.

5.1 Implementation

We have implemented constrained satisfying Dijkstra's algorithm i.e. Dijkstra's over allowed edges, constrained tree sketch algorithm and sketch construction within RDF3X [22]. In our implementation, we store graphs in RDF-3X edgewise with each edge represented as a triple $\langle s, label, t\rangle$. 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.

5.1.1 Sketch Implementation

Implementing Dijkstra's algorithm with in RDF-3X, involves opening a scan over SPO index to determine, for each node visited during the execution of the algorithm, all the successor nodes. For reverse Dijkstras algorithm, needed during computation of sketches, we simply open a scan on the OPS index, and letting the algorithm run. For simplicity, we store the sketches also as RDF triples in a separate database under RDF-3X with following format: $\langle v_i\rangle\langle t\rangle\langle l_{ij} : p_{ij}\rangle$ for forward sketch and $\langle v_i\rangle\langle f\rangle\langle l_{ij} : p_{ij}\rangle$ for backward sketch, where, v_i is the id of the source node, l_{ij} is the landmark for node v_i from the seed-set S_j and p_{ij} refers to the sequence of node-ids and edge label ids alternatively, between the node v_i and the landmark node l_{ij}.

5.1.2 Constrained Dijkstra's Algorithm

For Constrained Dijkstras algorithm over RDF-3X it involves opening a scan over SPO index and then doing a join of all the predicates with constraint set. Alternatively, we can use an aggregated scan i.e. instantiating a scan over SP index for every node and label pair. But, the former is more efficient, hence we use it for obtaining the label satisfying successors of a node. The priority-queue required during Dijkstras algorithm is maintained in memory using an implementation available in GNU-C++ STL.

5.2 DataSet

We have used three data sets for testing the performance of SkIt, two of them(Orkut and SocLive) are social networks and Yago is a knowledge repository. We synthetically labeled the edges on these two social networks with edge-labels following exponential distribution with exponent 0.5. Yago is a large entity-relationship network with every edge labeled. These data sets are described in detail as follows:

Orkut: is a free on-line social network where users form friendship with each other. [1]
SocLive: is a free on-line community which allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong.[2].
Yago: is a huge semantic knowledge base, derived from Wikipedia[3], WordNet[4] and GeoNames[5]. Currently, yago has knowledge of more than 10 million entities and contains more than 120 million facts about these entities.[6] We have used a subset of this data set.

Properties of these data sets are listed in Table 5.1, where $|V|$ is number of vertices, $|E|$ is number of edges, $|\Sigma|$ is number of edge-labels in graph G, $\Omega(G)$ is avg degree of graph G, $\Delta(G_i)$ is maximum in-degree of graph G and $\Delta(G_o)$ is maximum out-degree of graph G.

DataSet	$\|V\|$	$\|E\|$	$\|\Sigma\|$	$\Omega(G)$	$\Delta(G_i)$	$\Delta(G_o)$
orkut directed	3,072,441	117,185,083	324	76.28	3415	33007
socLive directed	4,847,571	68,993,773	315	28.46	13906	20293
orkut undirected	3,072,441	234,370,164	324	152.56	33313	33313
socLive undirected	4,847,571	86,739,238	315	35.79	20335	20335
yago directed	14,395,591	30,717,443	96	4.26	3229	733896

Table 5.1: Dataset Properties

5.3 Query Generation

We have run our experiments for following two sets of queries constructed as follows:

1. *Positive label restrictions* For each data set, queries are generated in the following manner:
    (a) 100 nodes are chosen at random such that their degree is more than the average degree(This will lead to different path length queries).
    (b) A single spanning tree is constructed from these nodes.

[1]http://snap.stanford.edu/data/com-Orkut.html
[2]http://snap.stanford.edu/data/soc-LiveJournal1.html
[3]http://en.wikipedia.org/wiki/MainPage
[4]http://wordnet.princeton.edu/
[5]http://www.geonames.org/
[6]http://www.mpi-inf.mpg.de/yago-naga/yago/

(c) For each path length at-most 50 queries are generated with different set of source and target.

(d) The above process is repeated 3 times leading to generation of 3 spanning trees. We have reported the above details in table 5.2 where $|T|$ is the tree size i.e. number of nodes traversed, T_h is the tree height i.e. maximum path length, C_{max} is the maximum constraint size and construction time is in seconds. Note that, for undirected data sets for each path length at-most 20 queries are generated.

2. *Negative label restrictions*

   (a) Two nodes and constraint set is chosen uniformly at random.

   (b) Size of Constraint set is varied from 1 to 3.

   (c) Each set comprises of 100 queries.

DataSet	$\|T\|$	T_h	C_{max}	time(s)
	3064900	8	8	
	3064250	8	8	
orkut directed	3064829	8	8	864.698
	4400347	12	12	
	4400347	13	13	
socLive directed	4400002	8	8	971.039
	3072441	5	5	
	3072441	6	6	
orkut undirected	3072441	5	5	1148.889
	4843953	10	10	
	4843953	10	10	
socLive undirected	48438953	10	10	958.819
	219480	19	9	
	234402	27	10	
yago directed	221068	19	10	132.105

Table 5.2: Positive label restrictions query set construction statistics

5.4 Evaluation

For evaluation we have defined following three performance metrics:

1. ***Path Approximation Quality***: Let q is an approximation of the shortest path p, we define the approximation error of this path as

$$error(q) := \frac{|q| - |p|}{|p|} = \frac{|q| - dist(u,v)}{dist(u,v)} \in [0, \infty] \tag{5.1}$$

2. ***False negatives***: The fraction of queries which fail to return any path that satisfies the given constraint, although at least one such path exists.

3. ***Avg Running Time***: We compare the performance of our SkIt-based LCSP estimation technique against the performance of traditional Dijkstra's algorithm and standard TreeSketch algorithm. Note that for TreeSketch algorithm, we need to traverse through the heap of paths until a path that satisfies the given constraints is found. Finally, we also compare the performance against performance of constrained shortest path available in Neo4J (`http://www.neo4j.org`) – an open-source, enterprise-grade, high-performance graph management system.

All the experiments are conducted on systems with following configuration,

- System-D: Processor: Intel(R) Core(TM) i3 CPU 550 @ 3.20GHz RAM: 4GB and OS: Linux Mint 12
- System-S: Processor: Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz RAM: 64GB and OS: OpenSUSE.

5.5 Results

In this section firstly we report pre-processing time and disk-space of SkIt. Then, we assess approximation error and false negative ratio of SkIt, subsequently followed by average query running time reported in subsection 5.5.3.

5.5.1 Index Construction Time and Size

The index size reported here (sketch index and inverted index), includes all the 12 indexes automatically created by RDF3x, although we only use SPO index here. Thus, disk consumption can further be reduced. These indices are created over System-S.

It can be observed from table 5.3 that the construction time for path-sketches and labeled-path-sketches are in sync, hence we can compute labeled-path sketches without incurring any extra computation cost. Note that, the time reported here is the total time to construct the indices in forward direction and backward direction, both.

We can also observe that labeled-path-sketch index size for undirected data sets is less than their directed versions. This is because of the fact that path length in case of undirected data sets is smaller than their directed version.

Note that, the inverted index is created in separate round, thus we can further minimize the total pre-processing time of SkIt.

DataSet	path-sketch(s)	labeled-path-sketch(s)	inverted index(s)	neo4j(s)
orkut directed	17323	17944	10706	8765
socLive directed	21647	24015	5553	5236
orkut undirected	30425	31266	15402	20902
socLive undirected	24234	25739	4884	6911
yago directed	11529	11000	12148	104

Table 5.3: Index construction time

DataSet	path-sketch(gb)	labeled-path-sketch(gb)	inverted index(gb)	neo4j(gb)
orkut directed	10.3	20.8	9.4	3.7
socLive directed	21.0	45.6	6.4	2.4
orkut undirected	9.8	16.5	8.1	7.1
socLive undirected	20.9	42.3	4.0	2.7
yago directed	13.4	14.7	3.2	1.6

Table 5.4: Index size

Dataset	query-type	SkIt-1hop		SkIt-2hop		path-sketch	
		μ	σ	μ	σ	μ	σ
orkut-directed	positive	0.0073	0.0503	0.0045	0.0331	0.0112	0.0605
	negative	0.1197	0.2135	0.0437	0.1281	0.1325	0.2219
socLive-directed	positive	0.00	0.00	0.00	0.00	0.0076	0.0619
	negative	0.0390	0.0731	0.0051	0.0295	0.0392	0.0732
orkut-undirected	positive	0.0019	0.0313	0.00	0.00	0.0108	0.0733
	negative	0.0679	0.0225	0.0182	0.0016	0.0679	0.0225
socLive-undirected	positive	0.00	0.00	0.00	0.00	0.0072	0.0678
	negative	0.0142	0.0484	0.0026	0.0203	0.0145	0.0535
yago	positive	0.0108	0.0532	0.0106	0.0529	0.0111	0.0533

Table 5.5: Approximation error measure for positive and negative label restriction query set

Dataset	query-type	SkIt-1hop	SkIt-2hop	path-sketch
orkut-directed	positive	0.163	0.1227	0.088
	negative	0.147	0.0568	0.136
socLive-directed	positive	0.082	0.056	0.024
	negative	0.0084	0	0.0042
orkut-undirected	positive	0.1589	0.0927	0.0728
	negative	0.00	0.00	0.00
socLive-undirected	positive	0.0845	0.0695	0.0169
	negative	0.0066	0.0033	0
yago	positive	0.0759	0.0629	0.0724

Table 5.6: False Negative Ratio for positive and negative label restriction query set

5.5.2 Approximation Quality

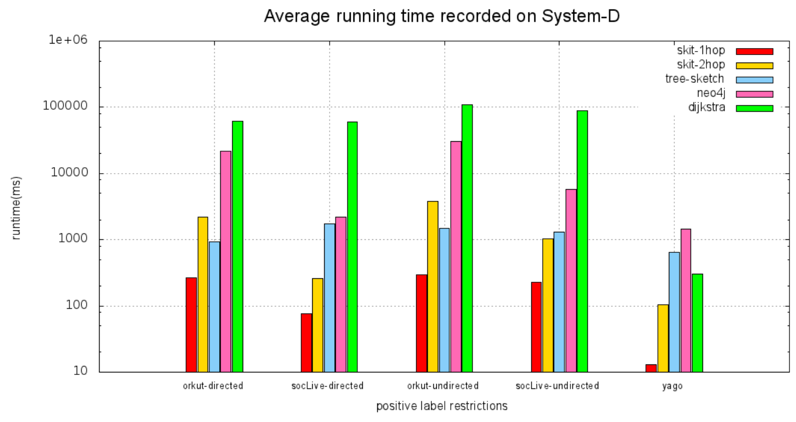
Average approximation error and false negative ratio are reported in table 5.5 and table 5.6, respectively. In table 5.5 μ refers to average approximation error and σ refers to standard deviation.

It can be observed that the average approximation error for SkIt-1hop and SkIt-2hop is always less than that for TreeSketch. This is particularly noticeable in social networks (orkut and socLive) where we see one order of improvement in approximation error of LCSP over TreeSketch.

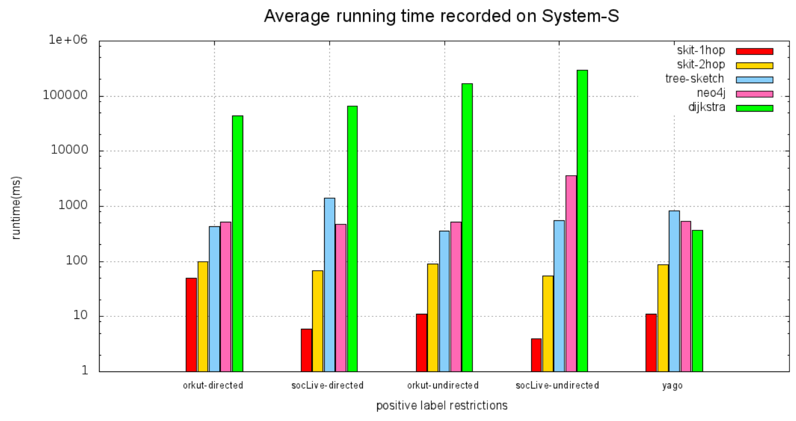
Although, false negative ratio for SkIt is inferior to tree sketch, because in tree-sketch pruning based on allowed/disallowed edge labels is done afterwards once all the paths are computed whereas in SkIt pruning is done at initial stages (during constrained tree sketch construction), which also leads to pruning of potential paths.

For Yago, Dijkstra's failed to return any paths for multiple choices of negative label restrictions queries we tried. This is so, because it is very less probable that there exists a path between two nodes uniformly chosen at random as it has low average degree and is more of a tree like graph, therefore we omit its results.

Thus, we can infer from table 5.5 and table 5.6 SkIt-2hop performs better than SkIt-1hop and Tree-sketch in terms of path quality. False negative ratio of SkIt-2hop is also comparable to Tree Sketch.



(a)



(b)

Figure 5.1: Average running time for positive label restrictions

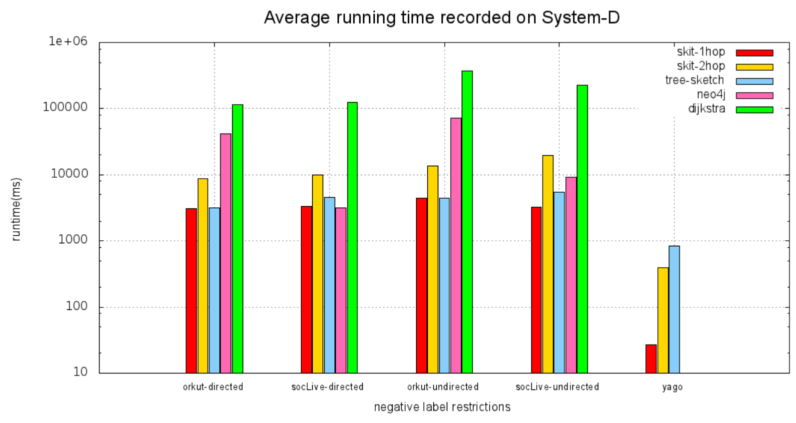
5.5.3 Query Processing Efficiency

Figure 5.1(a) and (b) shows average query execution time over System-D and System-S for positive label restrictions and figure 5.2(a) and (b) shows average query execution time over both the systems for negative label restrictions in logarithmic scale.

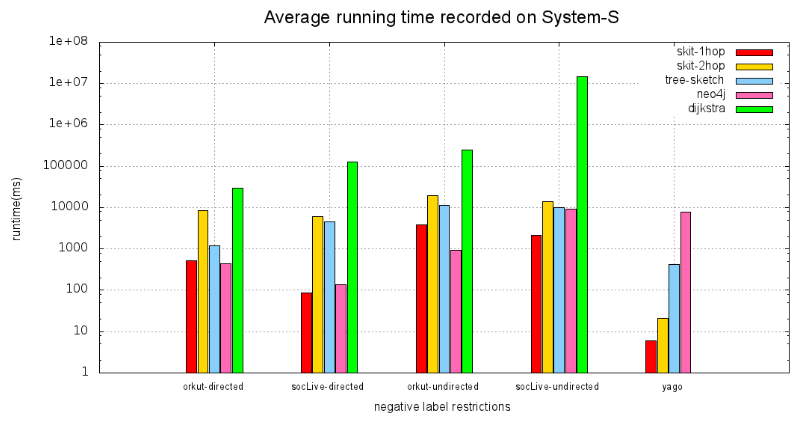
We can observe that, Neo4j being largely a in-memory graph database performs significantly better on System-S as compared to System-D. Moreover, it ran out of memory in case of negative queries for Yago on System-D. Nevertheless, SkIt-1hop outperforms it by almost an order of magnitude over large graphs on both System-S as well as System-D.

The performance of SkIt in case of negative label restrictions is slightly inferior because of the size of dictionary (created by RDF3x). Although, this can further be reduced.

The trade-off between path quality and execution time is also highlighted here i.e. execution time for SkIt-2hop is more than that for SkIt-1hop by one order of magnitude. In other words, the query execution time increases as the path quality improves.



(a)



(b)

Figure 5.2: Average running time for negative label restrictions

5.6 Label-with-Count Queries

In this type of query we also specify a count 'X' with each edge label in the constraint set C. This count implies that the label should not occur more than 'X' times. Clearly, if X is greater than diameter of the graph, then there is no chance of getting a path.

5.6.1 Implementation

For answering label-with-count-queries, slight modification is made in constrained tree sketch algorithm(1). We now maintain two map structures to keep track of edge label information one during constrained tree sketch construction and other one, during looking up of neighbours. These, two map structures are then used for verification of count i.e. a label does not occur more than its count during path construction (line 11 and 18 in Algorithm 1).

5.6.2 Query Generation

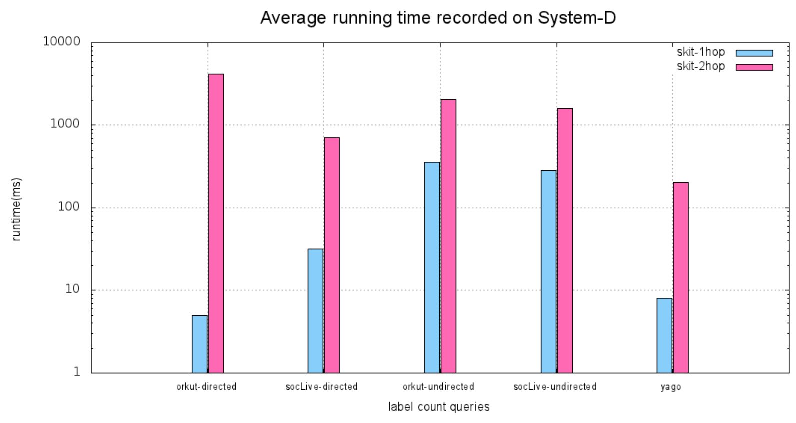
The trees generated in section 5.3 for obtaining the positive label restriction queries, are used here to generate label-with-count queries. Paths with constraint set size less than its path length, is our candidate set. This leads to query generation with edge-label count > 1 for at-least one edge label. From this candidate set we have chosen paths for which all edge labels present in the path has count > 1. Note that, these queries are of different path lengths and constraint set size.

5.6.3 Results

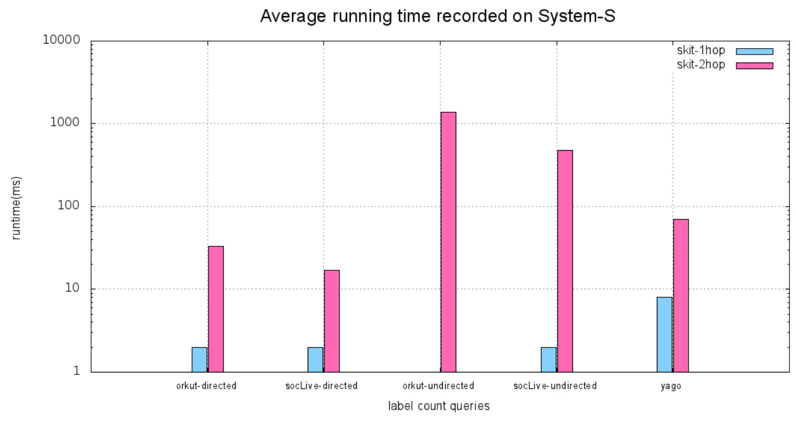
Table 5.7 reports false negative ratio for this query type using SkIt, where #queries are the number of queries executed, $max|\delta|$ is the path length and $max|C|$ is the maximum constraint set size. We can observe that false negative ratio has increased in comparison to positive label restrictions and negative label restrictions over same data sets. Surprisingly, average approximation error for all the data sets is 0 here. Figure 5.3 reports average execution time on logarithmic scale.

Dataset	#queries	$max\|\delta\|$	$max\|C\|$	skit-1hop	skit-2hop
orkut-directed	153	5	2	0.307	0.222
socLive-directed	220	8	4	0.309	0.263
orkut-undirected	150	4	2	0.413	0.373
socLive-undirected	218	7	3	0.353	0.312
yago-directed	496	12	4	0.316	0.262

Table 5.7: False Negative Ratio for label-with-count query set



(a)



(b)

Figure 5.3: Execution time over label count queries on different systems

5.7 Ranking Paths

In this section we have tried out the extension of SkIt and have compared the execution time of ranked paths with above execution time i.e. without ranking.

5.7.1 Implementation

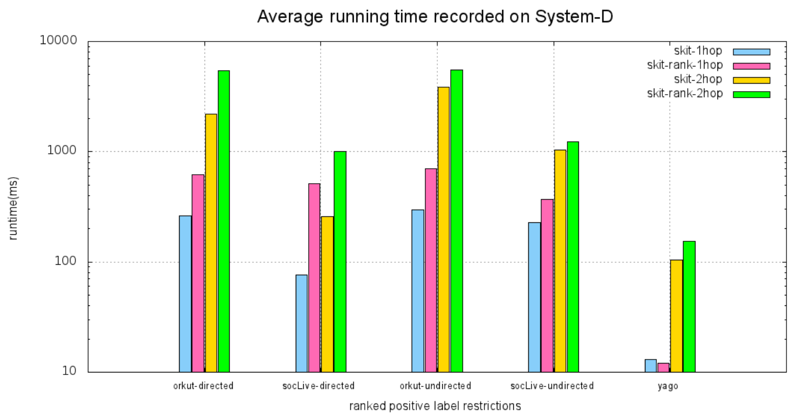
Similar modifications are made in constrained tree sketch algorithm1 as discussed in section 5.6.1 along with look up in weight function $W : l \to w$ which maps a label l to a positive real weight w, for calculating the score of a path($score(p)$) during its construction. Score of a path p is defined as :

$$score(p) = \Sigma w_i$$

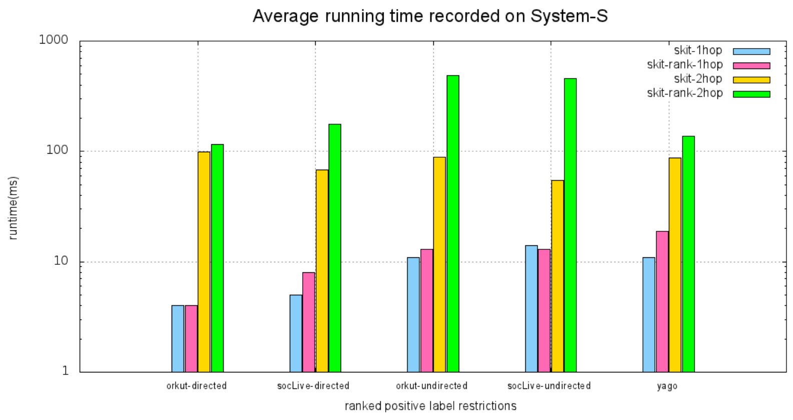
where $w_i = W(l_i)$ and $l_i \in C$. Note that, this is similar to worst case analysis for ranking of paths as score for all paths are computed.

5.7.2 Results

We have tested the ranking modification on positive label restrictions. Since, this modification also depends upon the constraint set size, hence we also need to come with a way so that ranked paths can be answered efficiently for all types of queries. Figure 5.4 shows execution time for our testing data sets with and without ranking using SkIt. It can be seen that with ranking the execution time has increased but in most of the cases it is of same order as without ranking in logarithmic scale.



(a)



(b)

Figure 5.4: Execution time with and without ranking on different systems

Chapter 6

Conclusion and Future Work

We introduced label constrained shortest path(LCSP) problem as an extension of shortest path problem that allows a shortest path query to specify which edge labels are allowed or not allowed. We then showed that we cannot use indexing approach devised for constraint reachability to answer LCSP queries efficiently.
We then discussed about two other indexing strategies proposed for LCSP queries and showed them to be in-optimal for answering LCSP on large graphs. We then came up with different indexing strategy SkIt for effective and efficient estimation of edge-label constrained shortest paths. Through experiments on large-scale graphs we demonstrate that SkIt outperforms its competitors for both positive label restrictions and negative label restrictions.
In further extension of our work we have tested SkIt performance on label-count-queries and ranking extension (i.e. if all computed paths are ranked). In continuation of this work, we plan to support even richer set of label constraints, and also reduce the size of SkIt index.

Bibliography

[1] neo4j open source graph database.

[2] M. Atre, V. Chaoji, and M. J. Zaki. Bitpath – label order constrained reachability queries over large graphs. *CoRR*, abs/1203.2886, 2012.

[3] C. Barrett, K. Bisset, M. Holzer, G. Konjevod, M. Marathe, and D. Wagner. Engineering label-constrained shortest-path algorithms. In *Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, AAIM '08, pages 27–37, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30:200–0, 2000.

[5] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *In Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 161–168, 1998.

[6] J. Bourgain. On lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, 52(1):46–52, Mar. 1985.

[7] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. F. Werneck. Shortest-path feasibility algorithms: An experimental evaluation. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill, 1990.

[10] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 401–410, New York, NY, USA, 2010. ACM.

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

[12] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In S. Abiteboul, K. Böhm, C. Koch, K. L. Tan, S. Abiteboul, K. Böhm, C. Koch, and K. L. Tan, editors, *ICDE*, pages 39–50. IEEE Computer Society, 2011.

[13] J. Feigenbaum, S. Kannan, A. Mcgregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *In ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, 2005.

[14] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[16] A. Gubichev and T. N. 0001. Path query processing on very large rdf graphs. In A. Marian and V. Vassalos, editors, *WebDB*, 2011.

[17] A. Gubichev, S. J. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In J. Huang, N. Koudas, G. J. F. Jones, X. Wu, K. Collins-Thompson, and A. An, editors, *CIKM*, pages 499–508. ACM, 2010.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost pathss. *SIGART Bull.*, (37):28–29, Dec. 1972.

[19] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 123–134. ACM, 2010.

[20] D. Kirchler, L. Liberti, T. Pajor, and R. W. Calvo. UniALT for regular language contrained shortest paths on a multi-modal transportation network. In A. Caprara and S. Kontogiannis, editors, *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–75, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[21] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proceedings of the 24th international conference on Scientific and Statistical Database Management*, SSDBM'12, pages 177–194, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[23] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 867–876, New York, NY, USA, 2009. ACM.

[24] M. Qiao, H. Cheng, and J. X. Yu. Querying shortest path distance with bounded errors in large graphs. In *Proceedings of the 23rd international conference on Scientific and statistical database management*, SSDBM'11, pages 255–273, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] M. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *Proceedings of the VLDB Endowment*, 4(2):69–80, 2010.

[26] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[27] L. Tang and M. Crovella. Virtual landmarks for the internet. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, IMC '03, pages 143–152, New York, NY, USA, 2003. ACM.

[28] L. Tang and M. Crovella. Geometric exploration of the landmark selection problem. In *In PAM*, pages 63–72, 2004.

[29] M. Thorup and U. Zwick. Approximate distance oracles. In *STOC*, pages 183–192, 2001.

[30] K. Xu, L. Zou, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Answering label-constraint reachability in large graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1595–1600. ACM, 2011.

[31] M. Zhou, Y. Pan, and Y. Wu. Efficient association discovery with keyword-based constraints on large graph data. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2441–2444. ACM, 2011.