BitPath – Label Order Constrained Reachability in Large Graphs

Technical Report

Medha Atre\textsuperscript{1}, Vineet Chaoji\textsuperscript{2}, Mohammed J. Zaki\textsuperscript{1}, and James A. Hendler\textsuperscript{1}
\textsuperscript{1}Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy NY, USA  
\textsuperscript{2}Yahoo! Labs, Bangalore, India

ABSTRACT

With ever expanding size of the web, sizes of graphs with semantic relationships are growing very rapidly. The semantic relationships are typically presented as edge-labeled graphs where nodes are the entities and the edge label signifies the relationships between the two entities. RDF graphs are an example of such graphs. Pure reachability queries on these graphs do not give any information about the semantic association between two remote entities. On the other hand, the typical size of semantic graphs present on the web has made general regular path query processing infeasible.

In this paper we focus on the following graph path query problem – given a source node \( s \), a destination node \( d \) and a sequence of ordered edge labels “label\text{seq}”, does there exist any path between the two nodes which has the edges labels in “label\text{seq}” (in the given order) on the path? We solve this problem by a combination of graph indexing, and a query processing algorithm based on a divide-and-conquer procedure and greedy pruning of the search space. We have evaluated our technique on graphs with more than 22 million edges and 6 million nodes – much larger compared to the datasets used in published work on path queries. We compare our approach with optimized-DFS, optimized-focused-DFS, and bidirectional-BFS methods.

1. INTRODUCTION

More and more data is represented as graphs these days resulting from applications that span social networks (friend-of-a-friend network\textsuperscript{1}), life sciences (e.g. UniProt \textsuperscript{2}) security and intelligence networks, and even Government data (data.gov project\textsuperscript{2}).

For instance, in case of the UniProt protein sequence and annotation data in RDF format, the edge labels may provide important information about the interaction between two proteins or a possible relationship between proteins and genes. Similarly, in social network graphs, the edge labels define a relationship between two persons or entities, e.g., a university and a person. Within security and intelligence networks, these semantically rich graphs are analyzed to identify connections between two seemingly unrelated entities. Also with the advent of Semantic Web, Resource Description Framework (RDF) is becoming a de-facto standard to present semantic relationships between entities. RDF graphs are edge-labeled directed graphs.

Due to the prevalence of graph data, path queries have become an important area of research. The approaches taken to evaluate path queries on XML data (e.g., XPath) cannot be applied to general graphs, since XML is assumed to be tree structured data for all practical purposes. General regular path queries on any graph ask for all pairs of nodes which have at least one path between them that satisfies the given regular path expression. This problem has been proved to be NP-hard \textsuperscript{14}. On the other hand, pure reachability queries on graphs do not give any information about the semantic relationship between two remote nodes. Hence researchers have started to work on a hybrid problem – constrained reachability \textsuperscript{14}. The problem of constrained reachability asks if a destination node \( y \) is reachable from \( x \) through a path which satisfies the constraints in the query. The definition of constraints depends on the targeted problem, e.g., number of hops, type of edge labels, or a regular path expression.

With the growing size of graph datasets, there is also more variety in the semantic relationships between nodes (e.g., an RDF representation of DBLP and IMDB graph contains 145 and 222 distinct edge-labels over 13 million and 5 million total edges, respectively). This poses two main challenges – (1) without knowing the exact structure or schema of the graph, it can be difficult to pose a precise query between two nodes. It can take several iterations of various combination of edge-labels to arrive at the one which fetches the correct answer, and (2) the amount of preprocessing required for answering a path query can limit the scale of the graphs that can be handled by an algorithm. Hence a method which achieves a trade-off between (a) the expressiveness of path query, (b) indexing (preprocessing) time, and (c) query execution time, is more desirable. In this work, we address these issues by proposing a light-weight method to process label order constrained reachability (LOCR) queries.

Given a source node \( x \), a destination node \( y \) and an ordered sequence of edge labels (“\( a \cdot b :: c \cdot d \cdot e \cdot f \cdot g \cdot h \cdot i \cdot j \)\), an LOCR query represented by \( Q(x, y, (a, b, c, d)) \) asks if there exists a path between \( x \) and \( y \), such that edge labels \( a, b, c \) and \( d \) appear somewhere on the path in the given order. The query returns a binary response – Yes or No. The ‘::’ and ‘:*’ in the label order allow any other labels (0 or more) to appear in between on the path. These symbols are used in accordance with the Perl regular expression (regex) syntax. Note that as per Perl regex, ‘:\*’ means 0 or more ‘\a’s which alters the meaning of our query, hence we use ‘\a\*’, which enforces at least one ‘\a’ followed by any other labels allowed by ‘\:*’. If there exists a path between \( x \) and \( y \) with follow-
ing edge labels \("d \ d \ g \ b \ e \ f \ q \ c \ h \ d \ i \ j\)" the answer to the query is YES. On this path given labels \(a, b, c, d\) appear in the given order but also have other labels interleaved among them. But if the only path between \(x\) and \(y\) has edge labels \("d \ a \ e \ f \ b \ b \ c\)\), then answer to the query is NO, because \(a, b, c, d\) labels do not appear in the given order in the query. Such queries can be utilized in social networks or intelligence services graphs, wherein we are not only interested in the presence of a path between the two entities but also the interconnections along the path. For instance, let us assume we are interested in finding the trace to a fraudulent organization \(X Y Z\) through the connections of a political figure \(A B C\). Specifically, we are interested in the query expression \("* friendOf \) \(* hasMembers \) \(* worksFor \) \(* funds \) \(*"\).

In a different setting, consider social network graphs such as Facebook and LinkedIn. LinkedIn contains a variety of relationships among people and between people and organizations/groups, e.g., classmates, colleagues, ex-colleagues, worked-at, working-at, group memberships and so on. With-out complete knowledge of the interconnection between various relationships, we may be interested in finding out if there exists a path between person \(D E F\) and an organization in \(San Francisco\) satisfying the following path \("* classmates \) \(* workingAt \) \(* locatedIn \) \(*"\). We believe that LOCR queries achieve a faster response to flexible path queries over large graphs. The problem of LOCR queries addressed in this paper is different from the problem of LCR queries handled by Jin et al in [14]. The differences are pointed out in Section 2.

Our main contributions in this work are:

1. Introduction of label-order-constrained-reachability problem in the context of edge-labeled graphs.
2. A light-weight graph indexing scheme based on compressed bit-vectors – BitPath – which stores all the successor and predecessor edge sets of nodes in the graph. Successor edges are edges that can be reached from a given node through a depth-first or breadth-first traversal from the given node. Similarly, predecessor edges are edges that appear on any path leading to the given node. The proposed indexing scheme requires just two depth-first passes on the graph.
3. A divide-and-conquer query algorithm using greedy-pruning strategy for efficient pruning of the search space using the aforementioned index. The algorithm recursively divides the given query into sub-queries to arrive at the answer.
4. Experiments on large graphs – more than 22 million edges and 253 distinct edge labels. To the best of our knowledge these graphs are largest amongst the published literature on path queries.

2. RELATED WORK

Early work by Mendelzon and Wood [18] shows that the general regular path query problem on graphs is NP-hard. In general the indexing schemes on graphs for evaluation of path queries can be grouped into the following three categories: (1) P-indexes (path indexes which typically index every unique path), (2) D-indexes [16] (node index – used for determining in constant time ancestor-descendant relationship), (3) T-indexes (for path-pattern such as \(t\) in XML) [23, 5, 6, 17, 9]. D-indexes and T-indexes can only be used in the context of XML graphs as it is non-trivial to decide ancestor-descendant relationship in constant time on a directed graph which does not assume tree structure [21]. Some approaches build P-indexes with equivalence classes of nodes based on the incoming or outgoing paths to and from the nodes [19, 4, 15, 8, 13].

Some other approaches that suggest building P-indexes are Index Fabric [11] and APEX index [10]. Most of the work for building P-indexes has been in the context of XML graphs (except early proposals of P-indexes, e.g., 1-index, 2-index by Milo and Suciu [19]). Among these approaches BmpPath Index (BPI) [12] and BitCube [22] use bit-vector based indexes. The BitPath scheme proposed in this paper also uses compressed bit-vectors for indexes. While BPI and BitCube use the bitmapped indexes to index paths in the XML graphs, BitPath uses compressed bit-vectors to only index the unique edges in the graph. While a vast number of techniques have been proposed for path indexing in XML graphs, those approaches cannot be used for general edge-labeled graphs since XML is widely viewed as tree structured data. On the other hand, number of paths in an RDF graph with 10-15 million unique edges over 5-6 million nodes can be of the order of \(10^{25}\) or more. It is computationally infeasible to index such a large number of paths due to space and runtime constraints.

Recent work by Jin et al. [14], proposes a novel approach for evaluating label constrained reachability (LCR) queries. Given a source node \(x\) and destination node \(y\) and a set of edge labels \(S\), an LCR query checks the existence of at least one path between \(x\) and \(y\) such that each edge label on that path is in \(S\)? No labels other than those in \(S\) can appear on that path. Their solution uses either building complete transitive closure or by utilizing approximate maximal spanning tree on the given graph. Approximate maximal spanning tree is found by recursively constructing different approximate spanning trees. The construction of an approximate maximal spanning tree is to reduce the index space. The LCR query processing algorithm utilizes indexes built on top of the approximate maximal spanning tree, interval labeling, and kd-trees (range search tree). The computational complexity of their index building procedure using approximate maximal spanning tree construction is \(O(n|V||E|((|S|)^2 + n/n_0) + |E| + V|log|V|)|\), and using the generalized transitive closure \(M\) is \(O(|V|^2|2^{|S|}}\), where \(S\) is the set of unique edge labels in the graph, and \(n\) and \(n_0\) are the sample sizes for approximate spanning tree. For example, with 253 labels, as for some of the graphs in our study, their complexity would be \((2^{253})\) or \(2^{253}\), which is prohibitive!

For LCR queries, the set of unique edge labels appearing on the satisfying path is a subset of the labels specified in the query. The satisfying path can have these edge labels in any order. On the other hand, LOCR queries handled by BitPath allow any other edge labels to appear on the path as long as the given order of edge labels in the query is satisfied. Our algorithm tries to keep the index construction light-weight by a) not indexing all paths in the graph, b) utilizes compressed bit-vectors for index representation thereby optimizing the index size, and c) using a divide-and-conquer approach for recursively splitting the query into sub-queries. For splitting the query into two sub-queries it utilizes a greedy pruning heuristic based on the BitPath indexes.

3. BITPATH INDEXES

Let \(G = (V, E, L, f)\) be a directed edge labeled graph, where \(V\) is the set of nodes, \(E\) is the set of edges, \(L\) is the
set of unique edge labels, and \( f \) is a edge-labeling function \( f : E \rightarrow L \), for each edge \((v_i, v_j)\).

Before building the BitPath indexes, we transform the given graph into a directed acyclic graph by collapsing the strongly connected components (SCCs). Since the LOCR queries specify constraints on the edge labels on the path, it is imperative to capture the edges (and their labels) that get collapsed in a SCC. Following steps outline the process:

1. Identify SCCs in the graph using Tarjan’s algorithm\(^3\).

2. Let \( z \) be a new node representing the SCC ‘C’. For each edge \( e \) that gets removed as a result of collapsing the SCC, add a self-edge \((z, z)\) with label \(f(e)\) in the graph.

The purpose of adding self-edges with same edge-labels is to keep track of the edge labels that appear in a given SCC. These edges help in determining paths going through an SCC without having to traverse the entire SCC subgraph at query time.

A label-order-constrained-reachability (LOCR) query requires the knowledge of relative order of edge labels occurring in a given path. Basic methods to answer these queries can run a DFS (depth-first-search) or BFS (breadth-first-search) traversal on the subgraph below the source node while examining each path for label ordering. These approaches are acceptable on small graphs and when the query has a valid answer. But as shown in our evaluation, when these properties are not satisfied, the query performance suffers with such baseline approaches.

Another way to answer these type of queries can be by storing each unique path in the graph separately indexed by its source and destination node. But as pointed out in Section 2, it is computationally infeasible to index paths of the order of \(10^{25}\) or more due to time and space constraints. In view of the above mentioned challenges, we solve the LOCR problem by creating 4 types of indices on the graph, and designing a query answering algorithm, based on a combination of greedy-pruning and a divide-and-conquer heuristic.

The 4 types of indices are as follows:

1. **EID (edge-to-ID):** For each edge \( e \in E \) is mapped to a unique integer ID. For instance, for the graph shown in Figure 1, edge \((\text{the_matrix}, \text{movie})\) with label \(\text{rdf:type}\) is mapped to ID 5.

2. **N-SUCC-E (node’s successor edges):** For each node, we index IDs of all the successor edges, i.e., edges that will get visited if we traverse the subgraph under the given node. The self edges added to the graph as a result of collapsing an SCC can be handled trivially by examining the head and tail of the given edge. Following the example in Figure 1, node \(\text{the_thirteenth_floor}\) will have edge IDs 1, 2, 3, 4, 5, 6 in its successor list.

3. **N-PRED-E (node’s predecessor edges):** Similarly, for each node we index the predecessor edges, i.e., edges that will get visited if we make a backward traversal on the entire subgraph above the given node. In Figure 1, node \(\text{movie}\) will have edge IDs 2, 5, 6 in its predecessor list.

4. **EL-ID (edge label to edge ID):** For each unique edge label \( l \in L \), we index IDs of all the edges in \( E \) which have edge label \( l \). In Figure 1, edge label \(\text{rdf:type}\) will have IDs 5 and 6 in its list.

In practice, we use bit-vectors of length \(|E|\) (total number of edges in the graph), for building N-SUCC-E, N-PRED-E, and EL-ID indexes. Each bit position in the bit-vector corresponds to the unique ID assigned to an edge as per the EID index. For the node \(\text{the_thirteenth_floor}\) in Figure 1, its N-SUCC-E bit-vector index will be “111111”, for the node \(\text{movie}\) its N-PRED-E index will be “010011”, and the EL-ID index of edge label \(\text{rdf:type}\) will be “000011”. We apply run-length-encoding on N-SUCC-E and N-PRED-E indices of each node depending on the compression ratio. Typically, run-length encoding delivers high compression ratio if there are large-gaps in the bitvector. A bit-vector with large gaps is one where a lot of 0s or 1s appear together. For instance, a bit-vector “11111000001111111” has large-gaps as opposed “1100101010101010” which has smaller gaps. The structure of the graph and the ordering of edges impacts the gaps in these indices. Typically the unique edge labels in the graph are much fewer compared to the number of nodes. Hence in an EL-ID index of an edge label, there are many more interleaved 0s and 1s as compared to an N-SUCC-E or N-PRED-E index. Since the EL-ID index typically has many small gaps, we do not apply run-length encoding on the EL-ID index. Note that at the time of querying we do not uncompress any compressed indices. All the algorithms are implemented to perform bitwise operations on both the gap-compressed indices as well as non-compressed indices.

### 3.1 BitPath Index Creation Algorithm

![Figure 1: BitPath indexes](http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm)
We create EID, N-SUCC-E, and EL-ID indexes in one DFS pass over the DAG generated after collapsing the SCCs as outlined in Section 3. N-PRED-E index is created by making one backward DFS pass on the graph and using the EID mapping generated in the first pass.

In the first pass, we find all the nodes with 0 in-degree (the root nodes). Starting with the first root node, we make a DFS pass over the entire subgraph below it, and do the same for other roots. Starting with ID 1, every new edge encountered in the DFS pass is given a new ID sequentially. For instance, in Figure 1, starting at root node :the_thirteenth_floor, we visit edge (:the_thirteenth_floor “1999”) with label :releasedIn first and assign ID “1” to it. Next when we visit edge (:the_thirteenth_floor “the_matrix”) with label :similar_to it is given ID “2”. Since this is a DFS traversal, we continue traversing the subgraph below node :the_matrix, and sequentially assign ID “3” to next edge visited. While assigning IDs to edges, we simultaneously maintain N-SUCC-E list for each node encountered. Every time we visit a new node, it is pushed on a DFS node stack. This stack keeps track of all the nodes that were visited on a path from root node to the current node. While exploring an unvisited node, we add all the outgoing edges of that node in the N-SUCC-E list of each node in the DFS stack. The node is popped out of the stack when it is marked as “visited”, i.e., when the entire subgraph below the node has been traversed. If a “visited” node is encountered through a different path, instead of exploring it again, we simply add all the edge IDs in its N-SUCC-E list to the N-SUCC-E list of all the nodes in the DFS node stack.

Once a node is marked “visited”, we build a bit-vector of N-SUCC-E index. Each bit position marked 1 in this bit-vector corresponds to the EIDs of the node’s successor edges. Since IDs to the edges were assigned as they are visited (while constructing the EID index), this scheme generates N-SUCC-E bit-vectors with large gaps for most nodes. We make use of this fact to apply run-length encoding on N-SUCC-E bitvectors. Note that this was a heuristic observed for many real-life graphs, and it is possible to generate a pathological graph where run-length encoding does not fetch the desired benefit.

Since for a typical RDF or any edge-labeled graph, |L| is much smaller than |V|, EL-ID index is typically smaller than EID and N-SUCC-E indexes (and even N-PRED-E discussed below). At the end of the first DFS pass, we get EID, N-SUCC-E, and EL-ID indexes.

In the second pass, we start from the leaf nodes – nodes with 0 out-degree – and make a backward DFS traversal on the graph. N-PRED-E index for each node is built in the same way as the N-SUCC-E index. The only difference is that in the second pass, we utilize the EID index that was populated in the first pass. Hence every time we encounter an edge, we simply look up its ID in the EID index and use it to construct the N-PRED-E index. When a node is marked “backward-visited” in this pass, we generate its bit-vector N-PRED-E index in the same manner as N-SUCC-E index. While building N-PRED-E indexes, sometimes we encounter bit-vectors with a lot of small gaps. Hence we use following heuristic – if the gap-compressed size of the bit-vector is larger than half of the size of the corresponding uncompressed version of the bit-vector, then store the bit-vector in uncompressed form. At the end of the second pass, all the indexes are written to the disk. In the next section, we describe the LOCR query processing algorithm using these 4 indexes.

4. BITPATH QUERY ALGORITHM

In an LOCR query \((x, y, (a, b, c, ..., l))\) – we want to find if there exists any path between a source \(x\) and destination \(y\), such that labels \((a, b, c, ..., l)\) appear on that path in the given order (other edge labels can appear on this path as well, ref. Section 1). The “...” denotes that there can be any number of labels specified in the order-constraint. The evaluation of an LOCR query can be broken down into the following steps:

1. Is \(y\) is reachable from \(x\)?
2. If the earlier condition is satisfied, we want to find if there exists any path between \(x\) and \(y\), such that all of the labels \(a, b, c, ... l\) appear somewhere on that path in that order.
3. Suppose we know that there exist some path where label \(b\) appears somewhere on the path. We want to find an edge \((m, n)\) with label \(b\) on that path.
4. Next, we want to find if there exists a path between \(x\) and \(m\), such that label \(a\) appears somewhere on it.
5. If we find that such a path exists between \(x\) and \(m\), we want to find if there exists a path between \(n\) and \(y\), such that labels \(c, ... l\) appear somewhere on it in the given order.

As can be seen by the steps outlined above, we recursively divided the original query into smaller sub-queries for evaluation. We make use of the 4 indexes – N-SUCC-E, N-PRED-E, EL-ID, and EID – to evaluate the sub-queries at every step.

The N-SUCC-E index of a node gives us all the edges that can be reached from that node and N-PRED-E index of a node gives us the edges that can eventually lead to that node. So if the intersection of N-SUCC-E index of node \(x\) and N-PRED-E index of node \(y\) is non-empty, i.e., if they have at least one edge common between them, node \(y\) is reachable from \(x\). This answers the first point above. If \([N-SUCC-E(x) \cap N-PRED-E(y) \cap EL-ID(b)] \neq \phi\), i.e., the intersection of successor index of \(x\), predecessor index of \(y\) and EL-ID index of label \(b\) is non-empty, there is at least one path from \(x\) to \(y\), where edge label \(b\) appears somewhere on the path. This solves our second step. N-SUCC-E\((x)\), N-PRED-E\((y)\) and EL-ID\((b)\) are bit-vectors and their intersections requires two bitwise AND operations, but this operation costs \(O(|E|) \approx O(|V|)\) for sparse graphs. Let the result of this bitwise AND operation be INTSECT.

Position of a 1 bit in INTSECT bit-vector gives EID of an edge with label \(b\). A reverse look up in the EID index gives us an edge, say \((m, n)\), with label \(b\). This satisfies the third step above. If \([N-SUCC-E(x) \cap N-PRED-E(m) \cap EL-ID(a)] \neq \phi\), it means that there exists at least one path between \(x\) and \(m\) where edge label \(a\) appears somewhere on the path. This addresses the forth step. If we put the earlier result and this result together, it tells us that there exists at least one path between \(x\) and \(y\), such that edge label \(a\) appears somewhere before label \(b\). Recursively, we solve our fifth step to find if there exists any path between \(n\) and \(y\) such that edge labels \(c, ... l\) appear somewhere on it.

In the example above, we chose to split the edge label order “\(a \ b \ c \ ... \ l\)” on label \(b\) first for the case of understanding. But for further optimization, we can choose the split-point depending on the selectivity of the intersection of N-SUCC-
Algorithm 1 greedy_pruning(x, y, label_seq)

1: min_label = 0
2: min_edges = \infty
3: for each l in label_seq do
4: \text{intsect} = N-SUCC-E(x) \cap \text{N-PRED-E(y)} \cap \text{EL-ID}(l)
5: if \text{intsect} < [min_edges] then
6: min_edges = \text{intsect}
7: min_label = l
8: return pair(min_edges, min_label)

Algorithm 1 outlines the greedy-pruning strategy. label_seq is the order of edge labels in the query. If at any given sub-query node in the divide-and-conquer tree (see Figure 2) there are more than one edge labels in the given label_seq, the greedy-pruning strategy takes intersection of \( N-SUCC-E(x) \cap N-PRED-E(y) \cap EL-ID(l) \) for each label \( l \) (lines 3–7 in Algorithm(1)). The edge label \( \text{min}_{\text{label}} \) which generates the smallest intersection set \( \text{min}_{\text{edges}} \) are returned by the greedy-pruning method (Line 8 in Algorithm(1)) and are subsequently used to partition the initial query into two sub-queries.

For a typical real life graph like RDF, there are few distinct edge labels \( L \) is very small as compared to the total number of edges. For instance, the UniProt RDF graph of 22 million edges has only 91 distinct edge labels. Moreover, the distribution of these edge labels is not uniform, a large number of edges have few distinct edge labels. In the UniProt dataset, the edge label "rdf:type" appears on \( \sim 5 \) million edges, about 10 edge labels appear on 1 million edges each and about 20 edge labels occupy \( \sim 100,000\)–\(200,000 \) edges each. We exploit this skewed label distribution to effectively prune the potentially large search space of edges. Although the skewed edge distribution holds true for most real life graphs, it is possible to synthetically build graphs where there is one root node, one sink node and a set of edge labels that follow uniform distribution. For such a graph, if we are given the root and sink nodes and a list of edge labels in a query, the greedy_selection will not be able to achieve any pruning because every edge will be in the N-SUCC-E index of source node and N-PRED-E index of the sink node. Now, Algorithm 2 describes the divide-and-conquer strategy.

Algorithm 2 divide-and-conquer(x, y, label_seq)

1: res = FAIL
2: if topo_order[y] - topo_order[x] < label_seq.size() then
3: return FAIL
4: res == SUCCESS
5: pair(min_edges, min_label) = greedy_pruning(x, y, label_seq)
6: if min_edges == \emptyset then
7: return FAIL
8: if label_seq.size() <= 1 then
9: return SUCCESS
10: for each eid in min_edges do
11: edge = eid_to_edge(eid) \( (k,l) \) is the tail, and \( k \) is the head of the edge
12: res = divide-and-conquer(x, e.tail, belq1)
13: if res == SUCCESS then
14: res = divide-and-conquer(e.head, y, belq2)
15: if res == SUCCESS then
16: break
17: return res

For the sake of illustration, let us assume that the LOCR query checks if the label order \( (a, b, c, d, e) \) is satisfied between source node \( x \) and destination node \( y \). To evaluate this query, Algorithm(2) is called on \( x \) and \( y \) nodes with label_seq containing all the edge labels \( (a, b, c, d, e) \). Although this example considers a sequence for length five, the algorithm is invariant to the length of the query. Also, an edge label can be repeated any number of times in the sequence, e.g., \( (a, a, b, c, c) \) or \( (a, a, a, b, b) \). The label_seq can as well be empty, in which case it simply translates to a reachability query.

If difference between the topological order of \( x \) and \( y \) is lesser than the length of label_seq, it means that there is no path from \( x \) to \( y \) of length \( |\text{label_seq}| \) or more. Divide-and-conquer uses this simple heuristic to preclude exploring paths shorter than \( |\text{label_seq}| \) (Line 2 in Algorithm(2)). Using greedy_pruning (line 5 in Algorithm(2)) we first get the minimal set of EIDs (\( \text{min}_{\text{edges}} \)) such that they are common between the successor edges of \( x \), predecessor edges of \( y \), and also contain a label \( \text{min}_{\text{label}} \in \text{label_seq} \). If the minimum set of edges for any label \( l \) in label_seq is empty, it clearly implies that nodes \( x \) and \( y \) do not have any path with label \( l \) between them. In this case, we stop exploring the paths further and return (line 8 in Algorithm(2)). Otherwise, we split the label_seq into 2 parts, such that, if the \( \text{min}_{\text{label}} \) is \( c \) and the original label_seq is \( (a, b, c, d, e) \), we split it into \( (a, b) \) and \( (d, e) \) (lines 12, 13 in Algorithm(2)). For each edge \( e \in \text{min}_{\text{edges}}, f(e) = c \). Let \( e \) be an edge over nodes \( (a, k, e, \text{head} = l) \) (line 16). Now the original query is divided into 2 parts – (1) there is any path from node \( x \) to e.tail such that it satisfies a label order \( (a, b, c, d, e) \).
b)?, (2) is there any path from node e.head to y such that it satisfies a label order (d, e)? If the label_seq is split over a or e, one of the sub-queries will have an empty label_seq. A sub-query with empty label_seq is just a reachability query. Such sub-query is skipped as the reachability of the nodes is previously evaluated in the greedy-pruning step.

With respect to the runtime complexity, divide-and-conquer algorithm’s in-memory program stack size is at most the size of original label sequence of the query. If we assume a uniform distribution of edge labels in the graph, in the worst case the divide-and-conquer algorithm can get called as many as $(2 \times |E|/|L|)$ times (Lines 15–21 in Algorithm(2)) for a given call to the divide-and-conquer method. Hence the worst case complexity of the entire runtime of the algorithm is $O((|E|/|L|)(|label_seq|))$. Since the BitPath algorithm will be called at most $|E|/|L|$ times for each label in the sequence. This is true for a query on a graph with one super root and one super sink node, where the root and sink nodes are the target nodes in the query. But as for most real life graphs, the edge labels follow a non-uniform distribution and rarely there is a single root and sink node in the graph. The complexity of greedy-pruning is $O(|label_seq|\times |E|)$. But since we use bit-vectors for storing N-SUCC-E, N-PRED-E, and EL-indexes, for all practical purposes greedy-pruning does not imply traversing the entire graph $|label_seq|$ times.

### 4.1 Handling Nodes and Paths in SCC

Since every node in an SCC is reachable to every other node in the same SCC, it is difficult to decide the start and end of a path going through SCC and the order of the edge-labels on that path – which is required in LOCR query processing. Hence presently we process the paths going through SCCs by simply checking the self-edges introduced while merging the SCCs (ref. Section 3).

Let an LOCR query be $(x, y, (a, b, c, d, e))$, such that $x$ and $y$ are part of the same SCC in the original graph. They are represented by node $z$ in the graph obtained by merging SCCs. For such a query, we simply check if there exist 5 self-edges $(z, z)$ with labels $a, b, c, d,$ and $e$. As another example, a query with label_seq $(a, a, a, b, c)$ can be satisfied by traversing the edge $(z, z)$ labeled ‘a’ thrice and checking other self-edges $(z, z)$ for labels ‘b’ and ‘c’. If either $x$ or $y$ are part of different SCCs, say $t$ and $u$ respectively, we evaluate the original LOCR query as $(t, u, (a, b, c, d, e))$. We use this same technique for BitPath as well as the baseline methods used for performance evaluation, hence for all practical purposes we have sampled queries for experimental evaluation on the directed acyclic graph (with self-edges) obtained after merging the SCCs.

### 5. EVALUATION

BitPath indexing and query processing algorithm is developed in C and compiled using g++ – (v4.4) with -O3 optimization flag. We used an OpenSUSE 11.2 machine with Intel Xenon X5650 2.67GHz processor, 48 GB RAM with 64 bit Linux kernel 2.6.31.5 for our experiments. Although we used a desktop with 48 GB memory, the BitPath index size for the datasets is much smaller than that (refer Section 5.5).

#### 5.1 Competitive Methods

As outlined in Section 2, path indexing approaches suggested in the context of XML/XPath query processing cannot be used to evaluate LOCR queries on general graphs. RDF graphs can be represented in XML format\(^5\), hence we explored the options of evaluating LOCR queries on XML representation of an RDF graph. This requires translating the given LOCR query into equivalent XML path query. A faithful translation of an LOCR query into equivalent query on the XML graph of RDF does not represent a path query. It often has to be processed using iterative join of two or more tree patterns (twig) queries. Native XQuery specifications do not support recursive joins of tree pattern queries, where the number of recursions are not known at the query time. An example of this is given in Appendix A. Also the BitPath method of indexing and processing LOCR queries can be applied to any other edge-labeled directed graph. But any edge-labeled directed graph – which does not satisfy RDF constraints – cannot be represented as an XML graph. Hence for our evaluation we used optimized versions of DFS and BFS as our baseline methods for comparative performance.

1. **Optimized DFS (DFS):** Given an LOCR query between nodes $x$ (source) and $y$ (destination), we check the out-degree of $x$ and in-degree of $y$. If the out-degree of $x$ is lesser than in-degree of $y$, we start the DFS walk from $x$ and continue until $y$ is reached and the given path satisfies LOCR label_seq. If $y$’s in-degree is lesser, we start a reverse-DFS walk from $y$ with reversed order of labels in the query and continue up to $x$. This method is referred to as “DFS” in the rest of the text.

2. **Optimized-Focused DFS (F-DFS):** This method is same as optimized DFS, but additionally at every node we check the reachability of the destination node $y$, (or reachability of node $x$ if we perform reverse DFS) We check reachability by using the intersection of N-SUCC-E and N-PRED-E BitPath indices. If $y$ is not reachable from the given node $n$, $N$-SUCC-E($n$) $\cap$ $N$-PRED-E($y$) = φ. This method is further enhanced as follows: Maintain a reachability array. The very first time node $n$ is explored, update reachability[n] to note if $y$ is reachable from $n$. If node $n$ is visited again through a different path, next time just look up reachability[n] to decide if the paths below $n$ should be explored or discontinued.

3. **Optimized-Bidirectional-BFS (B-BFS):** In bidirectional BFS, we traverse down the subgraph below $x$ and above $y$ one step at a time, maintaining forward and reverse BFS queues. Each node enqueued in the BFS queue has its own label_seq, associated with it, which tells which labels in the original sequence have been seen on a path to node $n$. Similarly we maintain a reverse label_seq, for nodes in the reverse BFS queue. At every iteration we perform an intersection of nodes in the forward and reverse BFS queues. If there is a common node in two BFS queues, we join their label_seq to check the satisfiability of the query.

We further optimize bidirectional-BFS as follows: if a node in the BFS queue is reached through another “better” path\(^6\) before being taken out of the BFS queue, that node’s label_seq is updated with the label_seq seen on the “better” path. In our experience, the optimized bidirectional BFS performed better than the naïve bidirectional BFS by an order of magnitude. For simplicity the optimized-bidirectional-BFS method is referred to as “B-BFS” in the rest of the text.

#### 5.2 Datasets and Queries

We used 2 real RDF datasets: RDFized DBLP dataset by LSDIS lab – SwetoDBLP \(^2\) and a smaller subset of UniProt

\(^5\)http://www.w3.org/TR/rdf-syntax-grammar/

\(^6\)A path is “better” if it has seen more labels in the LOCR label-order.
Table 1: Datasets Characteristics

<table>
<thead>
<tr>
<th></th>
<th># Edges</th>
<th># Nodes</th>
<th># Edge labels</th>
<th>Max-indeg</th>
<th>Max-outdeg</th>
<th>Avg-in/outdeg</th>
<th>Largest depth</th>
<th>SCCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Mat</td>
<td>13,991,226</td>
<td>4,085,180</td>
<td>293</td>
<td>19</td>
<td>13,054</td>
<td>3.65</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>UniProt</td>
<td>22,589,927</td>
<td>6,634,185</td>
<td>91</td>
<td>800,127</td>
<td>1046</td>
<td>3.35</td>
<td>10</td>
<td>424</td>
</tr>
<tr>
<td>SwetoDBLP</td>
<td>13,378,152</td>
<td>5,458,220</td>
<td>145</td>
<td>907,731</td>
<td>9245</td>
<td>2.44</td>
<td>66</td>
<td>146</td>
</tr>
</tbody>
</table>

Figure 3: Mean runtime with standard deviation for varying query size. Row 1: Positive queries for R-Mat, Row 2: Negative queries for R-Mat, Row 3: Positive queries for UniProt, Row 4: Negative queries for UniProt, Row 5: Positive and negative queries for SwetoDBLP (Y-axis scale of each graph is different).

We also used a synthetically generated dataset using the R-Mat algorithm [7, 1]. R-Mat graph is assigned edge labels using Zipf [20] distribution with distribution parameter (s) as 2.95. The statistical characteristics of these datasets are given in Table 1. “Largest depth” is the largest depth of a node in the topological sort. Note that edge and node counts given here are after collapsing the strongly connected components. The original edge and node counts are only slightly higher. These graphs are much larger as compared to the previous published results related to path query processing.

We generated two types of queries – (1) those which have a path satisfying the given LOCR query - we call these positive queries, (2) those which do not have any path satisfying the LOCR query but the destination node is reachable from the source node – negative queries.

In all we generated 50K positive and 50K negative queries.

7The most recent work by Jin et al [14] have used graphs up to 100k nodes.
on the three datasets using following procedure: First, 100K paths are generated through a “backward” traversal over the directed acyclic graph, starting from the leaf nodes. At each node during the backward traversal, a parent node is selected with either 1) uniform probability, or 2) probability proportional to the topological order of the node. The second strategy is incorporated specifically to discover longer paths in the graph. Subsequently, these paths are used to generate both positive and negative queries.

For positive queries, each edge label appearing on a path is removed with uniform probability. To generate negative queries, we introduce an extra edge label which does not appear on the given path and shuffle the edge labels to generate a random order. Note that this method can still result in incorrect negative queries. Without the knowledge of all paths between a pair of nodes, it is possible that a randomly generated negative LOCR query might be satisfied by some other path between the same pair of nodes. In such cases, we simply discard the query.

As outlined previously in Section 4, most real-life graphs follow a non-uniform the edge-label distribution, which benefits in evaluation of LOCR queries using the greedy-pruning strategy. Figure 4 gives the edge-labels’ frequency for each dataset.

![Figure 4: Edge Label Frequencies](image)

### 5.3 Evaluation Metrics

We considered the following metrics for evaluation –

1. Average time to run queries of the same query-length, i.e., we grouped queries of same query-length, and computed average query processing time and standard deviation for those queries. Length of the query is \( |\text{label}_{seq}| \), e.g., an LOCR query \( (x, y, (a, a, b, b, c, d)) \) has query-length 6. Since we chose each label on randomly generated paths with uniform probability, typically an LOCR query of length 6 was generated from a randomly generated path of length 12. Observe that the query length does not indicate the length of the **actual path** in the graph which satisfies that query.

2. BitPath index construction time for each dataset.

3. Cumulative size of BitPath indexes for each dataset.

Since we generated the queries with random walks, for the interested reader we have given the distribution of queries by query length in Appendix B.

### 5.4 Query Performance

We ran the 50K positive and negative queries each on all four methods – BitPath, DFS, F-DFS, and B-BFS. For long running queries, we set a threshold of 15 minutes/query, i.e., if a query ran for more than 15 minutes, it is abandoned. Note the following:

- For SwetoDBLP’s positive as well as negative queries, DFS as well as F-DFS methods took significant time to finish on most queries. Hence we abandoned evaluating the queries using the two DFS methods.
- For SwetoDBLP’s negative queries, B-BFS was taking significantly longer time to finish (more than 24 hours for 15K queries), hence we have shown results on only 15K queries.
- For R-Mat’s positive queries, B-BFS was taking long time to finish, hence after running the process for 24 hours, we abandoned further query processing and have shown results on 43K queries.
- For R-Mat’s negative queries, B-BFS was extremely slow taking more than 15 minutes for most queries, hence we have shown results only on 980 queries.

Figure 3 shows summarized results of evaluation parameters (1) and (2) as outlined in the previous subsection. Each graph shows the average runtime (by the tick on the vertical line) for a group of queries with same query length along with the standard deviation for that specific group. For example, row 1 column 1 shows BitPath’s performance on R-Mat dataset for positive queries. It shows that for a group of queries of length 9, the average query runtime is 0.0003 sec and the standard deviation for this group of queries is 0.0002 sec. Note that the scale of Y-axis on each graph is different, e.g., the Y-axis for graph in row 5 column 1 for BitPath’s performance over SwetoDBLP is from 0–0.1 whereas for graph in row 5 column 2 for B-BFS’ performance on SwetoDBLP is from 0–3.

Further analysis of the results shows that: B-BFS has inferior performance on the R-Mat graphs. This can be attributed to the **flatter** structure with not many long paths of R-Mat graphs (refer to Table 1 which shows that the “largest depth” of a node is 12), whereas SwetoDBLP graph of similar size and number of nodes has **deeper** structure with a lot of **interleaved** paths (“largest depth” of a node in Swe-toDBLP is 66). The flatter structure of R-Mat favors DFS method over B-BFS while on the other hand DFS method is inferior on SwetoDBLP due to its deeper structure. Analysis of UniProt graphs shows that it contains many disconnected subgraphs, as a result of which B-BFS, F-DFS, as well as DFS fair well on this graph.

B-BFS method delivers acceptable performance on Swe-toDBLP graph on positive queries. Note that this was possible due to our **optimized** version of B-BFS (ref. Section 5.1). The naive bidirectional BFS method was not able to deliver same performance. But as it can be noted, the performance of B-BFS method deteriorates as the length of the query increases. For negative queries, B-BFS method suffers on SwetoDBLP graph as it has to explore the entire subgraph between source and destination node. Note that on all 3 datasets with varying characteristics, BitPath delivers uniform performance on positive as well as negative queries. On average BitPath’s performance is 50 to 1000 times better for positive queries, and 1000 to 100k times better for negative queries compared to the baseline methods.

### 5.5 BitPath Index Size and Construction Time

The BitPath index construction time for R-Mat, UniProt, and SwetoDBLP datasets is 933 sec, 292 sec, and 809 sec respectively. Since the procedure of merging strongly connected components (SCCs) is same across all methods, the index construction time here does not include identification and merging of SCCs.

The cumulative on-disk size of N-SUCC-E, N-PRED-E, EL-ID and EID indexes for R-Mat, UniProt, and Sweto-
BLP datasets are 3.7 GB, 5 GB, and 3 GB, whereas the on-disk size of these graphs is 243 MB, 394 MB, and 232 MB respectively.

The uncompressed size of N-SUCC-E and N-PRED-E bit-vector indexes for R-Mat, UniProt, and SweToDBLP graphs would have been 15,269 GB, 37,466 GB and 18,255 GB respectively (since each node has a bit-vector index of successor and predecessor edges, uncompressed size of these indices in bytes would be \#nodes \times \#edges / 8). Note that this size is excluding the size of EL-ID and EID indexes, whereas the cumulative index sizes given above include size of all 4 indexes (N-SUCC-E, N-PRED-E, EL-ID, EID). Hence with our approach of numbering the edges and applying run-length-encoding on the N-SUCC-E and N-PRED-E indexes (ref. Section 3) reduces the on-disk size of indexes by a factor of 5000-6000.

6. CONCLUSION

In this paper we introduce the reachability problem with label-order constraints. This problem is of specific interest for large graphs with diverse relationships (i.e., large number of edge labels in the graph). For instance, graphs representing social networks, intelligence services, government data—where the precise knowledge of paths between various entities (nodes) is not known in advance. More sophisticated methods of graph indexing for the constrained reachability work well on smaller graphs, but they often face scalability issues for large real-life graphs. Similarly, the complexity of indexing all paths prohibits its use in practice for graphs which do not assume tree-structure.

We propose a simple method of building light-weight indexes on graphs using compressed bit-vectors. Our novel divide-and-conquer algorithm along with greedy-pruning strategy delivers more uniform performance across graphs of different structural characteristics. We have evaluated our method over graphs of more than 6 million nodes and 22 million edges—the best of our knowledge, the largest among the published literature in the context of path queries on graphs. In the future, we plan to improve this method to process a wider range of path expressions and also to incorporate ways of enumerating actual path description.

7. REFERENCES


APPENDIX

A. LOCNR QUERY FOR XML REPRESENTATION OF RDF

Consider the following RDF/XML representation of a DBLP dataset. For simplicity we have three sample entries—“Book1”, “Article2”, and “Article1000”, although the dataset can contain several such entries, denoted by “......” in the example. The XML tree representation of this dataset with the three entries is shown in Figure 5. Note that by default RDF to XML conversion tools do not add ID/IDREFs to cross reference same URIs in the XML data. The corresponding RDF graph is shown in Figure 6. The cloud shown in the figure represents many more edges and nodes in between the citations.

Suppose the RDF graph has a transitive path with edges labeled “cites” from element “Book1” to element “Article1000”. We are interested in the following LOCNR query, (Book1, Article1000, (cites)). Looking at the XML representation of the RDF graph in Figure 5, there is no such path between element “Book1” and “Article1000”, although such a path may exist between corresponding nodes in the RDF graph. This example shows that an LOCNR query cannot always be translated into an equivalent path query over XML graph.

<DBLP>
  <Book rdf:about="Book1">
    <chapter>
      <Chapter rdf:about="Introduction">
        <section>
          <Section rdf:about="Section1">
            <cite rdf:resource="Article2"/>
            <figure>Example RDF graph</figure>
          </Section>
        </section>
      </Chapter>
    </chapter>
  </Book>
</DBLP>

http://www.w3.org/TR/rdf-syntax-grammar/
Figure 5: Example XML representation of RDF graph of DBLP data

B. DISTRIBUTION OF QUERIES

Figure 6: Example RDF graph of DBLP data

Figure 7: Distribution of queries as per query-length, Row 1: R-Mat queries, Row 2: UniProt queries, Row 3: SwetoDBLP queries.