Evaluating Path Queries over Frequently Updated Route Collections

Panagiotis Bouros, Member, IEEE, Dimitris Sacharidis, Member, IEEE, Theodore Dalamagas, Member, IEEE, Spiros Skiadopoulos, and Timos Sellis, Fellow, IEEE.

Abstract—The recent advances in the infrastructure of Geographic Information Systems (GIS), and the proliferation of GPS technology, have resulted in the abundance of geodata in the form of sequences of points of interest (POIs), waypoints etc. We refer to sets of such sequences as route collections. In this work, we consider path queries on frequently updated route collections: given a route collection and two points \( n_s \) and \( n_t \), a path query returns a path, i.e., a sequence of points, that connects \( n_s \) to \( n_t \). We introduce two path query evaluation paradigms that enjoy the benefits of search algorithms (i.e., fast index maintenance) while utilizing transitivity information to terminate the search sooner. Efficient indexing schemes and appropriate updating procedures are introduced. An extensive experimental evaluation verifies the advantages of our methods compared to conventional graph-based search.

Index Terms—Route collections, path queries, frequent updates.

1 INTRODUCTION

Several applications involve storing and querying large volumes of data sequences. For instance, the recent advances in the infrastructure of Geographic Information Systems (GIS) and geodata services, and the proliferation of GPS technology, have resulted in the abundance of user or machine generated geodata in the form of point of interest (POI) sequences. We refer to sets of such sequences as route collections.

As an example, consider people visiting Athens, having GPS-enabled devices to track their sightseeing and create routes through interesting places. Figure 1 shows two routes in Athens. The first, \( r_1 \), starts from the National Technical University of Athens and ends at the new Museum of Acropolis. The second, \( r_2 \), starts from the Omonia Square and ends at the Acropolis Entrance. Web sites such as ShareMyRoutes.com and TravelByGPS.com maintain a huge collection of routes, like the above, with POIs from all over the world, that are frequently updated as users continuously share new interesting routes.

Given the availability of large route collections, the problem of identifying paths on the route collection arises frequently. Given a route collection and two POIs, hereafter called nodes, \( n_s \) and \( n_t \), the path query returns a path, i.e., a sequence of nodes, that connects \( n_s \) to \( n_t \). As an example, consider the route collection of Figure 1 and a path query: “Find a path from Academy to Zappeion following existing routes”.

This work targets path query evaluation on large disk-resident route collections that are frequently updated. Updates involve additions and deletions of routes. A route collection can be trivially transformed to a graph; hence, path queries can be evaluated using standard graph search techniques. Such methods follow one of two paradigms. The first employs graph traversal methods, such as depth-first search. The second compresses the graph’s transitive closure, which contains reachability information, i.e., whether a path exists between any pair of nodes. Both paradigms share their strengths and weaknesses. While the latter techniques are the fastest, they are mostly suitable for datasets that are not frequently updated, or when the updates are localized, since they require expensive precomputation. On the other hand, the former are easily maintainable, but are slow as they may visit a large part of the graph.

Based on these observations, we propose two generic search-based paradigms that exploit transitivity information within the routes, and differ in their expansion.
phase. For each route that contains the current search node, route traversal search, introduced in [1], expands the search considering all successor nodes in the route, while link traversal search considers only the next link. Both paradigms terminate when they reach a route that leads to the target, and are faster than conventional search.

We present two algorithms, discussed in [1], based on the first paradigm. RTS employs an inverted file, R-Index, on the nodes of the route collection. RTST uses the T-Index (in addition to R-Index) that captures transitions among routes allowing earlier termination.

We then introduce three novel methods based on the second paradigm. LTS employs an augmented variant of R-Index and features a similar termination condition to RTS. Similarly, LTST has a stronger condition based on the T-Index. The LTS-k algorithm forgoes the high storage and maintenance cost of T-Index and features a tunable termination condition, which is at least as strong as that of LTS and can become as strong as that of LTST.

Furthermore, we discuss efficient maintenance techniques as routes are added and deleted from the collection. A thorough experimental study demonstrates that the link traversal search methods always outperform a conventional graph traversal algorithm. Among them, LTS-k is shown to offer the best trade-off between efficiency and maintenance cost.

The remainder of this paper is structured as follows. Section 2 establishes the necessary background and discusses the related work in detail. Section 3 discusses route traversal search, and Section 4 introduces the link traversal search paradigm. Then, Section 5 discusses maintenance of the index structures under frequent updates of the route collection. Section 6 demonstrates our experimental results and Section 7 concludes the paper.

2 PRELIMINARIES AND RELATED WORK

Section 2.1 establishes the necessary background, while Section 2.2 reviews relevant literature.

2.1 Preliminaries

Let $N$ denote a set of nodes, e.g., POIs, waypoints, etc.

Definition 2.1 (Route) A route $r(n_1, \ldots, n_k)$ over $N$ is a sequence of distinct nodes $(n_1, \ldots, n_k) \in N$.

We denote the set of nodes in a route $r$ as $nodes(r)$, and its length as $|r| = |nodes(r)|$.

Definition 2.2 (Route collection) A route collection $R$ over $N$ is a set of routes $\{r_1, \ldots, r_m\}$ over $N$.

We denote all nodes in a route collection as $nodes(R)$.

Definition 2.3 (Link) A node in $nodes(R)$ is called link if it is contained in at least two routes in $R$.

Example 2.1 Figure 2(a) illustrates a route collection $R = \{r_1, r_2, r_3, r_4, r_5\}$. Nodes $a, b, c, d, f, s, t$ are links.

Definition 2.4 (Path) A path on a route collection $R$ is a sequence of distinct nodes $(n_1, \ldots, n_k) \in nodes(R)$ such that for every pair of consecutive nodes $(n_i, n_{i+1})$, $n_{i+1}$ is the immediate successor of $n_i$ in some route of $R$.

In Figure 2(a), note that a path may involve parts of routes from $R$. Fig. 2. A route collection $R$, an answer to PATH($s, t$), and routes graph $G_R$.}

Definition 2.5 (PATH queries) Let $R$ be a route collection, and $n_s$ and $n_t$ be two nodes in $nodes(R)$. The path query $PATH(n_s, n_t)$ returns a path from $n_s$ to $n_t$ on $R$.

Example 2.2 Consider the route collection in Figure 2(a). Path $(s, w, a, c, d, f, y, t)$ is an answer to query $PATH(s, t)$, constructed by (1) visiting the nodes $w$ and $a$ after $s$ in $r_3$, then, (2) using link $a$ to change from route $r_3$ to $r_2$ and visit $c$ and $d$, and finally, (3) using link $d$ to change from route $r_2$ to $r_1$, visit $f$, $y$ and the target $t$.

A route collection $R$ can be easily mapped to a graph by merging all routes in $R$.

Definition 2.6 (Routes graph) The routes graph of a route collection $R$ is a labeled directed graph $G_R(N, E)$, where $N = nodes(R)$, and an edge $(n_i, n_j, r_k) \in E$ if there exists a route $r_k \in R$ with $n_j$ immediately after $n_i$.

Example 2.3 The collection $R$ in Figure 2(a) is mapped to routes graph $G_R$ in Figure 2(b). The different line styles of the edges denote the five routes in $R$.

Storing the route identifiers as labels is necessary to handle deletions. Therefore, multiple edges between two nodes may exist, e.g., $(t, s, r_1)$ and $(t, s, r_5)$ in Example 2.3. Note that connectivity from $t$ to $s$ is only lost when both routes are removed from the collection.

2.2 Related Work

A path query on $R$ can be answered directly on routes graph $G_R$ using standard techniques, which we review next. Note that we distinguish between two distinct but closely related queries, studied in the literature: path and reachability queries. A path query $PATH(n_s, n_t)$ identifies a path from $n_s$ to $n_t$, while a reachability query $REACH(n_s, n_t)$ answers if such a path exists. Thus, an answer to $PATH(n_s, n_t)$ provides an answer to $REACH(n_s, n_t)$, while the converse does not hold.

Techniques for evaluating path/reachability queries follow two paradigms: (1) searching, and (2) encoding the graph’s transitive closure (TC). Searching methods deal with PATH queries, while TC methods primarily target REACH queries. As we discuss next, some of the TC techniques can be extended to evaluate PATH queries. Table 1 summarizes the related work in terms of the: (1) graph type supported, (2) support for REACH, (3) support for PATH, and (4) capacity to handle updates.

Searching. The simplest way to evaluate PATH queries is to traverse the graph at query time exploiting a search algorithm, e.g., depth-first or breadth-first search [2]. This approach has minimum space requirements, since it only needs to store the adjacency lists of the graph. In
addition, the adjacency lists can be easily updated. On the other hand, in the worst case, it may need to visit all nodes of the graph to answer a query.

Encoding the TC. The transitive closure (TC) of a graph $G_{R}(N, E)$ is the graph $G_{R}^{*}(N, E^*)$, where an edge $(n_i, n_j)$ is in $E^*$ if a path from $n_i$ to $n_j$ exists in $G_{R}$. Using TC a REACH query can be answered in constant time. However, even though efficient algorithms for computing the TC have been proposed, e.g., [14], [15], [16], the computation and storage cost are prohibitive for large disk-resident graphs. Therefore, various methods compress the TC.

2-hop [3], [17] identifies a set of nodes, called centers, that best capture the reachability information of a graph as intermediates. Thus, for each node $n_i$ the method constructs a list $L_{in}[n]$ with the centers that can reach $n$ and another $L_{out}[n]$ with those reachable from $n$. To determine the existence of a path from $n_s$ to $n_t$, it checks if $L_{out}[n_s]$ and $L_{in}[n_t]$ have a common center. To identify the path, along with the center $n_c$, the first node in the path from $n$ to $n_t$ (resp. $n_s$) must also be stored.

Computing the optimal 2-hop scheme is NP-hard. The work in [3] and [17] presents an approximation algorithm based on set covering [18] that constructs a 2-hop scheme larger by a logarithmic factor than the optimal one, but it still requires the computation of the TC. Therefore, this approach cannot be applied to large graphs. In addition, the work does not handle frequent updates. Compared to 2-hop our methodology is less efficient in evaluating path queries, but is significantly cheaper to construct and maintain.

HOPI [4], [5] reduces the construction time of 2-hop by exploiting graph partitioning, which works well for forests with few connections between the different subgraphs, e.g., collections of XML documents. Updates are handled by applying the construction method of [4]. HOPI is able to find elements, e.g., book, citation, author, in an XML document that match XPath expressions, e.g., /book//citation//author (where "//" is the ancestor-descendant operator). However, the focus of the work is to identify these elements and not detect the full path on the XML documents that contains them.

There is a number of works that transform the input graph to a DAG by replacing each strongly connected component with a super node. For example, [6] proposes a geometric-based method and [7] another one based on graph partitioning for the efficient construction of 2-hop. [9] proposes the 3-hop indexing scheme. The basic idea is to use a chain of nodes, instead of a single node, to encode the reachability information. [10] proposes a labeling scheme that assigns to each node a sequence of intervals, based on the postorder traversing of graph’s spanning tree. Updates are handled by leaving gaps in postorder numbers. Although not discussed, PATH queries can be answered on the DAG by computing the ancestors of the target node. The idea in [13] is to partition the graph into a set of paths and then use the path-tree cover, instead of assigning the intervals based on the graph’s spanning tree. [11] proposes dual-labeling for sparse graphs. [8] introduces the updatable 2-hop based on the node-separation property.

All the above techniques cannot evaluate PATH queries as the initial graph is collapsed. On the other hand, the GRIPP scheme [12] for graphs (not only DAGs) assigns to each node an interval label. Although not discussed, PATH queries can be answered by finding the descendants of the source node of the query. However, [12] does not handle frequent updates.

3 Route Traversal Search

This section revisits our work from [1] for evaluating PATH queries over route collections. Section 3.1 presents the $R$-Index on route collections and details the RTS algorithm. Section 3.2 outlines the RTST algorithm that additionally exploits information about the transitions among routes stored in the $T$-Index structure. Section 3.3 presents a detailed complexity analysis.

3.1 The RTS Algorithm

The Route Traversal Search (RTS) algorithm has the following key features. First, it traverses nodes in a manner similar to depth-first search. However, when expanding the current search node $n_q$, RTS considers all successor nodes for each route that includes $n_q$. Second, it employs a termination check, based on the reachability information within the routes, to considerably shorten the search. Both principles depend on the inverted file $R$-Index on the route collection which associates a node with the routes that contain it.

**Definition 3.1 (R-Index)** Given a route collection $R$ and a node $n_i \in nodes(R)$, $routes[n_i]$ is the ordered list of $(r_j,a_{ij})$ entries for all routes $r_j$ that include $n_i$ at their $a_{ij}$-th position, sorted on the route identifier $r_j$. $R$-Index contains the lists $routes[n_i]$ for all $n_i \in nodes(R)$.

**Example 3.1** Table 2 illustrates the $R$-Index for the routes shown in Figure 2(a).
The RTS algorithm takes as inputs: a route collection \( R \), the \( R \)-Index, the source \( n_s \) and target node \( n_t \) and returns a path from \( n_s \) to \( n_t \) if one exists, or null otherwise. The algorithm uses the following data structures: (1) a search stack \( Q \), (2) a history set \( H \), which contains all nodes that have been pushed in \( Q \), and (3) an ancestor set \( A \), which stores the direct ancestor of each node in \( Q \). \( H \) is used to avoid cycles during the traversal, and \( A \) to extract answer paths. Note that RTS visits each node once and, thus, there is a single entry per node in \( A \).

RTS proceeds similarly to depth-first search. Initially, the stack \( Q \) and \( H \) contain source node \( n_s \), while \( A \) is empty. RTS proceeds iteratively popping a single node \( n_q \) from \( Q \) at each iteration. The algorithm terminates when there exists a route \( r_e \) that contains both \( n_q \) and target \( n_t \), such that \( n_t \) comes after \( n_q \). Specifically, to check if the above condition holds, RTS looks for entries \( \langle r_e ; o_{eq} \rangle \) and \( \langle r_e ; o_{te} \rangle \) in lists \( \text{routes}[n_q] \) and \( \text{routes}[n_t] \) of \( R \)-Index respectively, such that \( o_{eq} < o_{te} \). The procedure is similar to a merge-join, as both \( \text{routes}[n_q] \) and \( \text{routes}[n_t] \) lists are sorted by the route identifier, that finishes when a common route \( r_e \) is found.

If such a route \( r_e \) is identified, the search terminates and the answer path is extracted. Specifically, starting from \( n_q \), RTS uses the information in \( A \) to backtrack to source \( n_s \) constructing the \( (n_q, \ldots, n_t) \) path. Then, it concatenates this path with the part of route \( r_e \) from \( n_q \) up to \( n_t \). During concatenation, the procedure ensures that each node is contained only once in the answer path.

If a common route \( r_e \) is not found, RTS expands the search retrieving \( \text{routes}[n_q] \) and considering all routes that contain \( n_q \). For each such route \( r_i \) and for each node \( n_r \) after \( n_q \) in \( r_i \), that is not in \( H \) (i.e., it has never been pushed in \( Q \)) the algorithm performs the following tasks. Node \( n_r \) is pushed in \( Q \) and inserted in \( H \). In addition, the entry \( \langle n_r ; n_q \rangle \), where \( n_r \) is the direct ancestor of \( n_r \) in route \( r_i \), is inserted in \( A \). The fact that only new nodes are inserted in \( Q \), ensures that RTS avoids cycles.

Example 3.2 We illustrate the RTS algorithm for the \( \text{PATH}(s, t) \) query on the route collection of Figure 2(a) using the \( R \)-Index presented in Table 2. Initially, we have: \( Q = \{ s \} \), \( \mathcal{H} = \emptyset \) and \( A = \emptyset \). At the first iteration, RTS pops \( s \) from \( Q \) and checks for termination joining lists \( \text{routes}[s] = \{ (r_1 : 5) \} \cdot (r_5 : 2) \} \) of current search node \( s \) with \( \text{routes}[t] = \{(r_1 : 4), (r_5 : 1)\} \) of target \( t \). The join identifies common route entries, i.e., \( r_1 \) and \( r_5 \), but in both cases \( s \) is after \( t \) and therefore, RTS needs to further search the collection.

Node \( s \) is contained in routes \( r_1, r_3 \) and \( r_5 \). When processing \( r_1(d, f, y, t, s) \) and \( r_5(t, s) \), the algorithm does not add anything to \( Q \), \( \mathcal{H} \) and \( A \) since there are no nodes after \( s \). When processing \( r_3(s, w, a, g) \), the algorithm adds to \( Q \) and \( \mathcal{H} \), nodes \( w, a \) and \( g \), and to \( A \), pairs \( (s, w), (w, a) \) and \( (a, g) \). After the fourth iteration, we have \( Q = \{ w, c, d, \mathcal{H} = \{ s, w, a, g, c, d, x \} \), and \( A = \{ (s, w), (w, a), (a, g), (c, d), (d, x) \} \).

At the fifth iteration, \( d \) is popped. Then, RTS joins lists \( \text{routes}[d] = \{ (r_1 : 1), (r_5 : 3) \} \) with \( \text{routes}[t] = \{(r_1 : 4), (r_5 : 1)\} \) and identifies entries \( (r_1 : 1), (r_5 : 4) \) in the common route \( r_1 \). Since in \( r_1 \), \( d \) is before \( t \), the search terminates successfully. The answer path \( (s, w, a, c, d, f, y, t) \) is the concatenation of \( (s, w, a, c, d) \) (the path from \( s \) to current search node \( d \), constructed using \( A \)) and \( (d, f, y, t) \) (the part of \( r_1 \) that connects \( d \) to target \( t \)).

#### 3.2 The RTST Algorithm

This section presents the Route Traversal Search with Transitions (RTST) algorithm for \( \text{PATH} \) queries. RTST expands the search as RTS, but employs a stronger termination check based on the transitions between routes. This additional reachability information is modeled by the transition graph \( G_T \), and is explicitly materialized in the \( T \)-Index structure.

Definition 3.2 (Transition graph) The transition graph of a route collection \( R \) is a labeled undirected graph \( G_T(R, E_T) \), where its vertices are the routes in \( R \), and a labeled edge \( (r_i, r_j, n_k) \) exists in \( E_T \) if \( r_i \) and \( r_j \) share a link node \( n_k \).

Intuitively, an edge \( (r_i, r_j, n_k) \) in the \( G_T \) denotes that all nodes in \( r_i \) before link \( n_k \) can reach those after \( n_k \) in \( r_j \), and vice versa, i.e., all nodes in \( r_j \) before \( n_k \) can reach those after \( n_k \) in \( r_i \).

Example 3.3 Figure 3 illustrates the transition graph for the route collection of Figure 2(a).

![Fig. 3. Transition graph for the route collection R.](image-url)

The transition graph is stored in a modified adjacency list representation denoted as \( T \)-Index.

Definition 3.3 \( (T \)-Index) Given a route collection \( R \), for each route \( r_i \in R \), \( \text{trans}[r_i] \) is the ordered list of \( (r_j, n_k; o_{ti}; o_{kj}) \) entries for all \( (r_i, r_j, n_k) \) edges of \( G_T \), where \( o_{ti} \) and \( o_{kj} \) denote the position of the link \( n_k \) in routes \( r_i \) and \( r_j \), respectively. The entries are sorted on the route identifier \( r_j \) solving ties with \( o_{ti} \). \( T \)-Index contains the lists \( \text{trans}[r_i] \) for all routes \( r_i \in R \).

Example 3.4 Table 3 illustrates the \( T \)-Index for the \( G_T \) graph presented in Figure 3.

RTST proceeds similar to RTS, but involves a different termination check. For each route \( r_i \) that contains the current search node \( n_q \), it checks if there exists an edge.
TABLE 3

<table>
<thead>
<tr>
<th>route</th>
<th>trans</th>
<th>list of route in $G_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>$(r_2, d: 1: 3)$, $(r_3, s: 5: 3)$, $(r_4, f: 2: 4)$, $(r_5, t: 4: 1)$, $(r_6, s: 5: 2)$</td>
<td></td>
</tr>
<tr>
<td>$r_2$</td>
<td>$(r_1, d: 5: 1)$, $(r_3, a: 3: 3)$, $(r_4, b: 2: 1)$, $(r_5, c: 4: 3)$</td>
<td></td>
</tr>
<tr>
<td>$r_3$</td>
<td>$(r_1, s: 1: 5)$, $(r_2, a: 3: 3)$, $(r_3, s: 1: 2)$</td>
<td></td>
</tr>
<tr>
<td>$r_4$</td>
<td>$(r_1, f: 4: 2)$, $(r_2, b: 1: 2)$, $(r_3, c: 3: 4)$</td>
<td></td>
</tr>
<tr>
<td>$r_5$</td>
<td>$(r_1, t: 1: 3)$, $(r_2, s: 2: 5)$, $(r_3, s: 2: 1)$</td>
<td></td>
</tr>
</tbody>
</table>

$(r_i, r_j, n_k)$ in $G_T$ such that $r_j$ contains target $n_t$, link $n_t$ is after $n_q$ in $r_i$ and $n_k$ is before $r_j$ in $r_i$.

If the above hold, then a path from $n_q$ to target $n_t$ via link $n_k$ exists, and thus a path from source $n_s$ to $n_t$ can be found. To perform this check RTST scans lists $\text{trans}[r_i]$ and $\text{routes}[n_i]$ from $T$-Index and $R$-Index, respectively, similar to a merge-join as both lists are sorted on the route identifier. The scan terminates when entries $(r_j, n_k; o_{j_1}; o_{j_2})$ of $\text{trans}[r_i]$ and $(r_j; o_{j_2})$ of $\text{routes}[n_i]$ match, i.e., when the following conditions are satisfied:
1. the entries correspond to the same route $r_j$,
2. $r_j$ contains link $n_k$ after $n_q$, i.e., $o_{j_1} > o_{q_j}$, and
3. $r_j$ contains $n_k$ before $n_t$, i.e., $o_{j_2} < o_{t_j}$

Example 3.5 Consider query PATH($s, t$) on the route collection in Figure 2(a) indexed by the $R$-Index of Table 2 and $T$-Index of Table 3. The first two iterations of the RTST algorithm are identical to those of RTS in Example 3.2. Then, the third iteration processes $a$. According to $R$-Index, $a$ is contained in routes $r_3(v, b, a, c, d, x)$ and $r_3(s, w, a, g)$. To check the termination condition for $r_3$, RTST joins list $\text{trans}[r_2]$ of $T$-Index with routes[] of $R$-Index. This results in the common route $r_1$ (condition (1)) with link $d$ of $(r_3, r_1, d)$ edge contained after $a$ in $r_2$ (condition 2) and before target $t$ in $r_1$ (condition 3). Thus, the answer path is $(s, w, a, c, d, f, y, t)$.

3.3 Complexity Analysis

Given a route collection $R$, let $|R|$ denote the number of routes, $|N|$ the number of distinct nodes, and $L_r$ the length of a route, assuming all routes have equal length. In the following, we assume that a disk page can store $B_N$ nodes, $B_R$ routes[], entries, and $B_T$ trans[] entries.

$R$-Index. The $R$-Index structure contains an entry for each node in every route, for a total of $|R| \cdot L_r$ entries. Therefore, it occupies $O\left(\frac{|R| \cdot L_r}{B_R}\right)$ pages. For the construction of $R$-Index, the entire collection must be accessed at a cost of $O\left(\frac{|R| \cdot L_r}{B_R}\right)$ I/Os, while the index must be stored at a cost of $O\left(\frac{|R| \cdot L_r}{B_N}\right)$ I/Os. An important factor in the performance of the algorithms is the size, in terms of entries, $|\text{routes}[]|$ of a $R$-Index list. In the average case, each list has the same number of entries, i.e., $O\left(\frac{|R| \cdot L_r}{|N|}\right)$. In the worst case, a node can be contained in all routes, i.e., the routes[] list has $O(|R|)$ entries. In the sequel, we assume the average case holds.

$T$-Index. Consider the $\text{routes}[n_i]$ list that contains an entry for each route $n_i$ belongs to. This node contributes $O(|\text{routes}[n_i]|^2)$ pairs of intersecting routes, and thus as many $T$-Index entries. Consequently, the total number of entries in $T$-Index is $O\left(\frac{|R|^2 \cdot L_r^2}{|N|^2}\right)$, while each list contains $O\left(\frac{|R|^2 \cdot L_r^2}{|N|^2}\right)$ entries on average. The $T$-Index occupies $O\left(\frac{|R|^2 \cdot L_r^2}{|N|^2} + \frac{|R|^2 \cdot L_r}{|N|^3}\right)$ pages. Its construction requires accessing the entire $R$-Index for a cost of $O\left(\frac{|R| \cdot L_r}{|N|^2 \cdot B_R}\right)$ I/Os, and writing the index on disk for $O\left(\frac{|R|^2 \cdot L_r^2}{|N|^2} + \frac{|R|^2}{|N|^3} + \frac{|R|^2}{|N|^4}\right)$ I/Os.

RTS and RTST. At each iteration, after node $n_q$ is popped, RTS and RTST perform two tasks. The first is to insert new nodes in the search stack and is common in both methods. This task requires retrieving the entire list $\text{routes}[n_q]$ at a cost of $O\left(\frac{|R| \cdot L_r}{|N|^2 \cdot B_R}\right)$ I/Os, and then retrieving all routes contained in $\text{routes}[n_q]$ at a cost of $O\left(\frac{|R| \cdot L_r + |R|^2}{|N|^2 \cdot B_R}\right)$ I/Os.

The second task is the termination condition. RTS and RTST need to retrieve the $\text{routes}[n_i]$ list of the target incurring $O\left(\frac{|R| \cdot L_r}{|N|^2 \cdot B_R}\right)$ I/Os. RTST additionally retrieves the $\text{trans}[r_i]$ list of $T$-Index for each route $r_i$ contained in $\text{routes}[n_i]$, at a cost of $O\left(\frac{|R| \cdot L_r}{|N|^2} + \frac{|R|^2 \cdot L_r}{|N|^3}\right)$ I/Os. Aggregating for $|N|$ nodes in the worst case scenario, we obtain the following complexities. RTS requires $O\left(\frac{|R| \cdot L_r}{|N|^2} + \frac{|R|^2 \cdot L_r}{|N|^3}\right)$ I/Os. RTST requires $O\left(\frac{|R| \cdot L_r}{|N|^2} + \frac{|R|^2 \cdot L_r}{|N|^3} + \frac{|R|^2 \cdot L_r}{|N|^4}\right)$ I/Os.

4 LINK TRAVERSAL SEARCH

Section 4.1 discusses the shortcomings of the algorithms in Section 3, and proposes the link traversal search paradigm that overcomes them. Then, Sections 4.2, 4.3, 4.4 present three novel methods based on this paradigm. Section 4.5 discusses their complexity.

4.1 The Link Traversal Search Paradigm

Although the algorithms of Section 3 perform fewer iterations than conventional depth-first search on the route collection graph $G_R$, they share three shortcomings. First, they perform redundant iterations by visiting non-links. To understand this, consider that the current search node $n_q$ is not a link and belongs to a single route $r_i$. Further, assume that the algorithm has visited $n_t$, which is the link immediately before $n_q$. Observe that if the termination condition does not hold at $n_t$, then it neither holds at $n_q$. To make matters worse, retrieving $\text{routes}[n_q]$ is pointless as it contains a single route $r_i$ in which all nodes after $n_q$ are already in the stack.

The second shortcoming is that the termination check is expensive. For current search node $n_q$, recall that both RTS and RTST retrieve lists $\text{routes}[n_q]$ and $\text{routes}[n_i]$ from $R$-Index, while RTST additionally retrieves all lists $\text{trans}[r_i]$ from $T$-Index for each $r_i$ included in $\text{routes}[n_q]$. This cost is amplified by the number of iterations, as the algorithms perform the check for every node popped.

The final shortcoming is due to the traversal policy. For each route that the current search node belongs to, the algorithms insert into the stack route subsequences
that contain a very large number of nodes. This increases
the space requirements of \( Q \) (and consequently of sets
\( H, A \)). More importantly, however, some of these nodes
may never be visited, which results to redundant I/Os incurred to retrieve them.

The next subsections introduce three methods, LTS,
LTST and LTS-k, that follow the link traversal search
paradigm. To deal with the first shortcoming, all algo-
rithms avoid visiting non-link nodes and conceptually
traverse the reduced routes graph \( G_R^-(N^-, E^-) \) of the route
collection, where \( N^- \subseteq \text{nodes}(R) \) contains all links, and \( E^- \)
contains all labeled directed edges \((n_i, n_j, r_k)\) such
that there exists a route \( r_k \in R \) in which \( n_j \) is the link
immediately following \( n_i \). Note that \( G_R^- \) is not explicitly
materialized, and is introduced to better illustrate the
link traversal search paradigm. For example, Figure 4
shows the reduced routes graph for the collection \( R \)
of Figure 2(a). Observe the differences between \( G_R \) in
Figure 2(b) and the reduced routes graph \( G_R^- \).

![Fig. 4. Reduced routes graph for the route collection \( R \).](image)

In the sequel, we assume that the source and target
nodes are always links. Otherwise, we set as source
(resp. target) the link immediately following (resp. pre-
ceding) it; if no such link exists, then no path can be
found. ¹ Under this assumption, a \( \text{PATH}(n_s, n_t) \) query on
\( R \) is equivalent to finding a path from \( n_s \) to \( n_t \) in \( G_R^- \),
and replacing each \((n_i, n_j, r_k)\) edge in the answer, with
the subsequence from \( n_i \) to \( n_j \) of route \( r_k \).

To tackle the second shortcoming, the algorithms reduce
the cost of the termination check by precomputing
a \( \text{target list} \) of routes, and checking if the current
search node belongs in one of them. This eliminates the
recurring I/Os for the check, at the expense of a pre-
processing cost for assembling the target list. Regarding
the third shortcoming, the traversal policy of the new
paradigm dictates that at each iteration only the links
immediately after the current search node are inserted
in the stack, exactly like a depth-first search on \( G_R^- \).

### 4.2 The LTS Algorithm

The Link Traversal Search (LTS) algorithm features a ter-
mination condition equivalent to that of RTS: the search
stops as soon as LTS visits a node (link) that lies on
the same route with the target. To traverse the routes
and check for termination, the algorithm employs an
augmented inverted file on the route collections, termed
\( \mathcal{R}\text{-Index}^+ \), which associates a node with the routes that
contain it and the immediately following link.

¹ A special case arises when both \( n_s \) and \( n_t \) are in the same route
and no link between them exists.

---

### TABLE 4

<table>
<thead>
<tr>
<th>node</th>
<th>routes(^+) list</th>
<th>node</th>
<th>routes(^+) list</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>{( r_2 : 3, c ), ( r_3 : 3, \varnothing )}</td>
<td>( t )</td>
<td>{( r_1 : 4, s ), ( r_5 : 1, s )}</td>
</tr>
<tr>
<td>( b )</td>
<td>{( r_2 : 2, a ), ( r_4 : 1, c )}</td>
<td>( v )</td>
<td>{( r_2 : 1, b )}</td>
</tr>
<tr>
<td>( c )</td>
<td>{( r_2 : 4, d ), ( r_3 : 3, f )}</td>
<td>( w )</td>
<td>{( r_3 : 2, a )}</td>
</tr>
<tr>
<td>( d )</td>
<td>{( r_1 : 1, f ), ( r_5 : 5, \varnothing )}</td>
<td>( x )</td>
<td>{( r_2 : 6, \varnothing )}</td>
</tr>
<tr>
<td>( f )</td>
<td>{( r_1 : 2, t ), ( r_4 : 4, \varnothing )}</td>
<td>( y )</td>
<td>{( r_1 : 3, t )}</td>
</tr>
<tr>
<td>( g )</td>
<td>{( r_3 : 4, \varnothing )}</td>
<td>( z )</td>
<td>{( r_4 : 2, c )}</td>
</tr>
</tbody>
</table>

**Definition 4.1 (\( \mathcal{R}\text{-Index}^+ \))** Given a route collection \( R \) and a node \( n_i \in \text{nodes}(R) \), \( \text{routes}^+[n_i] \) is the ordered list of \( \{r_j : o_{ij}, n_j^+\} \) entries for all routes \( r_j \) that include \( n_i \) at their \( o_{ij\text{-th}} \) position, where \( n_j^+ \) is the link immediately following \( n_i \) if one exists, or \( \varnothing \) otherwise. The entries are sorted on the route identifier \( r_j \), \( \mathcal{R}\text{-Index}^+ \) contains the lists \( \text{routes}^+[n_i] \) for all \( n_i \in \text{nodes}(R) \).

**Example 4.1** Table 4 illustrates the \( \mathcal{R}\text{-Index}^+ \) for the routes shown in Figure 2(a).

Note that the \( \mathcal{R}\text{-Index}^+ \) contains lists for non-link nodes as well, so that a link immediately following a non-link source or immediately preceding a non-link target can be identified, as discussed in Section 4.1.

Figure 5 presents the pseudocode of the LTS algorithm for evaluating a \( \text{PATH}(n_s, n_t) \) query. Similar to RTS, it uses stack \( Q \), and sets \( H \) and \( A \). Initially, \( Q \) contains the source \( n_s \) (Line 1). LTS constructs a target list \( T \) that contains entries \( \{r_i : o_{qi}\} \) for all routes that contain the target \( n_t \) (Lines 3–5). Then, LTS proceeds iteratively (Lines 6–16) until \( Q \) is depleted. At each iteration, assuming the current search link popped from the stack is \( n_q \) (Line 7), LTS examines each entry of \( \text{routes}^+[n_q] \) (Lines 8–14).

The algorithm terminates if there exists an entry in \( T \) indicating that \( n_q \) lies before \( n_t \) on a common route (Line 9), a condition which is identical to that of RTS. In that case, \( \text{ConstructPath} \) composes an answer path using the information in \( A \). Otherwise, if the next link node \( n_q^+ \) has not been previously visited, it is pushed in the stack and in \( H \). Further, the entry \( \{n_q^+, n_q : r_q : o_{qj}\} \) is inserted in \( A \) indicating that \( n_q^+ \) is reached from \( n_q \) following route \( r_q \). The position \( o_{qj} \) is used by \( \text{ConstructPath} \) to quickly identify the subroute of \( r_i \) between links \( n_q \) and \( n_q^+ \) if required, as explained in Section 4.1.

**Example 4.2** We illustrate LTS for \( \text{PATH}(s,t) \) using the \( \mathcal{R}\text{-Index}^+ \) of Table 4. Initially, LTS accesses \( \text{routes}^+[t] \) and constructs the target list \( T = \{r_1 : 4, r_5 : 1\} \).

At the first iteration, LTS pops \( s \) from \( Q \) and retrieves \( \text{routes}^+[s] \) that contains three entries. The termination
check (Line 9) for entries \( \{r_1 : 5, \varnothing\} \) and \( \{r_5 : 2, \varnothing\} \) fails, since the entries about \( r_1 \) and \( r_5 \) in \( T \) does not match \( s \) (not before \( t \)). \( Q \), \( H \) and \( A \) do not change as there is no link after \( s \) in \( r_1 \) and \( r_5 \). The check for \( \{r_3 : 1, a\} \) also fails as there is no entry for \( r_3 \) in \( T \). LTS inserts the link \( a \) into \( Q \) and \( H \), and the pair \( \{s : r_3 : 1, a\} \) into \( A \), and thus: \( Q = \{a\}, H = \{s, a\} \), and \( A = \{s : r_3 : 1, a\} \).

LTS proceeds expanding \( a \) and then \( c \). After the third
iteration we have \( Q = \{d, f\}, H = \{s, a, c, d, f\} \), and
\( A = \{s : r_3 : 1, a\}, \{a : r_2 : 3, c\}, \{c : r_4 : 2, d\}, \{c : r_4 : 3, f\} \).
Algorithm LTS
Input: n0, nℓ, R-Index+
Output: a path from n0 to nℓ if it exists, null otherwise
Parameters:
- stack Q: the search stack
- set H: contains all nodes pushed in Q
- set A: contains the direct ancestor link of each node in H
- list T: stores all routes that contain target nℓ

Method:
1. push n0 in Q;
2. insert n0 in H;
3. for each entry (rj: oj1, oj2) in routes+[n0] do
   4. insert (rj: oj1) in T;
5. end for
6. while Q is not empty do
   7. let nq = pop(Q);
   8. for each entry (rj: oq1, oq2) in routes+[nq] do
      9. if there is an entry (rj: oj1) in T such that oq1 < oj1 then
         return ConstructPath(nq, nq, nℓ, A: rj: oj1);
10. if nq ̸∈ H then
   11. push nq in H;
   12. insert nq in H;
   13. insert (nq: rj: oq1, oq2) in A;
14. end if
15. end for
16. end while
17. return null;

Fig. 5. The LTS algorithm.

At the next iteration, f is popped and LTS retrieves list routes+[f]. The entry \( \langle r_1 : 2, t \rangle \) matches the corresponding \( (r_1 : 4) \) in \( T \), since route \( r_1 \) contains the current search node \( f \) before target \( t \). Thus, LTS terminates the search and uses \( A \) to identify a sequence of links \( (s: r_3 : 1, (a: r_2 : 2), (c: r_4 : 3), (f: r_1 : 2), (t: r_2 : 4)) \) that leads to the target. \( (a: r_2 : 3) \) denotes that \( a \) is at position 3 in \( r_2 \); the symbol \( * \) implies that the position can be inferred. After retrieving the required parts of these routes, the path \( (s, u, a, c, f, y, t) \) is constructed.

4.3 The LTST Algorithm

The Link Traversal Search with Transitions (LTST) algorithm enforces a stronger termination check than LTS using the transition graph of the route collection. In particular, the LTST algorithm, similar to RTST, finishes when it reaches a node that is closer than two routes away from the target. To achieve this, LTST uses information from the \( T \)-Index, discussed in Section 3.2.

Figure 6 presents the pseudocode of the LTST algorithm, which is similar to that of LTS. The basic difference is in the contents of the target list \( T \) (Lines 3-8), which allow LTST to terminate sooner. The algorithm retrieves list routes+[n] from R-Index+ and accesses \( T \)-Index to retrieve lists \( \text{trans}[r_i] \) for all routes \( r_i \) in \( \text{routes}^{+}[n] \). Just like LTS, it inserts into \( T \) all routes that contain the target (Line 4). Moreover, LTST includes all routes \( r_j \) that can lead to \( n_\ell \) via some link \( n_k \) that resides on the same route \( r_j \) with \( n_\ell \) (Line 6). Intuitively, this implies that \( T \) contains routes \( (r_j)'s \) that directly lead to the target, and, in addition, routes \( (r_j)'s \) that intersect with them (provided that the intersection occurs before \( n_\ell \)). Therefore, \( T \) includes all routes that are less than two routes away from the target.

An entry of \( T \) has the form \( \langle r_j : o_\ell j, r_i : o_i \rangle \), which means that route \( r_j \) leads to the target \( n_\ell \) in route \( r_i \) via some link \( n_k \) that lies at the \( o_\ell j \)-th position in \( r_j \) and at the \( o_i \)-th position in \( r_i \) before \( n_\ell \). An entry \( \langle r_i : o_i, \emptyset \rangle \) implies that route \( r_i \) contains the target \( n_\ell \) (see Line 4). Note that the first item in the pair is used in the termination check, while the second in the construction of the answer path.

LTST terminates if current search node \( n_q \) lies on a route \( r_j \) that leads, via some link \( n_\ell \), to a route \( r_j \) that contains the target. Specifically, the algorithm checks if there exist entries \( \langle r_i : o_i1, r_j : o_j \rangle \) in \( T \) and \( \langle r_i : o_i, n_q \rangle \) in \( \text{routes}^{+}[n_q] \) such that \( o_i < o_i1 \) (Line 2). Note that based on its construction, the target list may contain multiple entries for the same route \( r_i \) corresponding to different via-links \( n_\ell \). However, as the termination check suggests, it suffices to retain only the entry with the latest via-link, i.e., that with the highest \( o_i1 \). A final subtle difference with LTS is that ConstructPath requires routes \( r_i, r_j \) and the positions of the via-link \( o_i1, o_j \) in them, so as to compose the answer path.

Algorithm LTST
Input: n0, nℓ, R-Index+, T-Index
Output: a path from n0 to nℓ if it exists, null otherwise
Parameters:
- stack Q: the search stack
- set H: contains all nodes pushed in Q
- set A: contains the direct ancestor link of each node in H
- list T: stores all routes that contain n0 and their intersecting routes

Method:
1. push n0 in Q;
2. insert n0 in H;
3. for each entry \( \langle r_i : o_i1, r_j : o_j \rangle \) in \( \text{routes}^{+}[n_0] \) do
   4. insert \( (r_i : o_i1) \) in T;
5. end for
6. while Q is not empty do
   7. let \( n_q = \text{pop}(Q) \);
   8. for each entry \( \langle r_i : o_i1, r_j : o_j \rangle \) in \( \text{routes}^{+}[n_q] \) do
      9. if there is an entry \( (r_i : o_i1) \) in T such that \( o_i < o_i1 \) then
         return ConstructPath(nq, nq, nℓ, A: rj: oj);
10. end if
11. end for
12. end while
13. return null;

Fig. 6. The LTST algorithm.

Example 4.3 Consider \( \text{PATH}(s, t) \) and \( R-Index^+ \) and \( T-Index \), shown in Tables 4 and 3, respectively. First, LTST retrieves \( \text{routes}^{+}[t] \) from \( R-Index^+ \), which contains two routes \( r_1 \) and \( r_5 \). Then, it retrieves the corresponding lists \( \text{trans}[r_1] \) and \( \text{trans}[r_5] \) from \( T-Index \), and constructs the target list \( T = \{ \langle r_1 : 4, \emptyset \rangle, \langle r_2 : 5, r_1 : 1 \rangle, \langle r_3 : 1, r_5 : 2 \rangle, \langle r_3 : 4, r_1 : 2 \rangle, \langle r_5 : 1, \emptyset \rangle \} \).

The first iteration of LTST is identical to that in Example 4.2. At the second iteration, LTST pops link a and retrieves list \( \text{routes}^{+}[a] \) from \( R-Index^+ \). The first entry \( \langle r_2 : 3, c \rangle \) in this list matches the entry \( (r_2 : 5, r_1 : 1) \) in the target list, since in \( r_2 \) node a lies at position 3 before some node \( n_k \) at position 5 that leads to the target via route \( r_1 \) (at this point LTST does not know that \( n_k \) corresponds to
Subsequently, in the second phase (Lines 13–17), the algorithm scans list \( \mathcal{L} \) and uses the \( \mathcal{R}-\text{Index}^+ \) to insert into \( \sigma \) all routes that contain a link of \( \mathcal{L} \). Similar to LTST, LTST-k retains a single entry \( (r_j; o_{ij}, r_i; o_{ik}) \) per route, that of the highest via position \( o_{ij} \).

**Algorithm LTST-k**

**Input:** \( n_s, n_t, k, \mathcal{R}-\text{Index}^+ \)

**Output:** a path from \( n_s \) to \( n_t \) if it exists, null otherwise

**Parameters:**
- stack \( \mathcal{Q} \): the search stack
- set \( \mathcal{H} \): contains all nodes pushed in \( \mathcal{Q} \)
- set \( \mathcal{A} \): contains the direct ancestor link of each node in \( \mathcal{H} \)
- list \( \mathcal{L} \): stores all links that are within \( k \) links from \( n_s \) in some route
- list \( \mathcal{T} \): stores all routes that contain a node in \( \mathcal{L} \)

**Method:**
1. push \( n_s \) in \( \mathcal{Q} \);
2. insert \( n_s \) in \( \mathcal{H} \);
3. for each entry \( (r_i; n_{ol}, n_{ot}) \) in \( \mathcal{R}-\text{Index}^+[n_i] \) do
   4. insert \( (n_o, r_i, o_{ik}) \) in \( \mathcal{L} \);
   5. retrieve route \( r_i \) and let \( o_{ik} = o_{ik} \);
   6. for \( m = 1 \) up to \( k \) do
      7. repeat // make \( o_{ik} \) point at the previous link in \( r_i \)
      8. let \( o_{ik} = o_{ik} - 1 \);
      9. until \( o_{ik} \) is a link;
   10. insert \( (n_t, r_i, o_{ik}) \) in \( \mathcal{L} \);
   11. end for
  12. end for
13. for each entry \( (n_t, r_i, o_{ik}) \) in \( \mathcal{L} \) do
  14. for each entry \( (r_j; o_{ij}, n_j) \) in \( \mathcal{R}-\text{Index}^+[n_t] \) do
  15. insert \( (r_j; o_{ij}, r_i; o_{ik}) \) in \( \mathcal{T} \);
  16. end for
  17. end for


Fig. 7. The LTST-k algorithm.

**Example 4.4** We illustrate the LTST-1 algorithm \((k = 1)\) for the \( \text{PATH}(s,t) \) query on the collection of Figure 2(a) using the \( \mathcal{R}-\text{Index}^+ \) presented in Table 4.

Initially, the algorithm constructs the list of links \( \mathcal{L} \) that are within one link from \( t \). It accesses \( \mathcal{R}-\text{Index}^+[t] = \{ (r_1; 4, s), (r_5; 1, s) \} \) and retrieves routes \( r_1 \) and \( r_5 \) at positions 4 and 1, respectively. Moving backwards in \( r_1, \) LTST-1 identifies \( f \) as the one link before \( t \); route \( r_5 \) contains no nodes before \( t \). Therefore, \( \mathcal{L} = \{ (t, r_1: 4), (f, r_1: 2) \} \) contains an entry for the target \( t \) and \( f \).

Subsequently, LTST-1 accesses the lists \( \mathcal{R}-\text{Index}^+[t] = \{ (r_1; 4, s), (r_5; 1, s) \} \) and \( \mathcal{R}-\text{Index}^+[f] = \{ (r_1; 2, t), (r_4; 4, \emptyset) \} \) for the two links in \( \mathcal{L} \). Then, it creates the target list that contains an entry for each route \( r_1, r_4 \) and \( r_5 \): \( \mathcal{T} = \{ (r_1; 4, \emptyset), (r_4; 4, r_1; 2), (r_5; 1, \emptyset) \} \).

The first two iterations of LTST-1 are the same as those of LTST. At the third iteration, LTST-1 pops link \( c \) and accesses list \( \mathcal{R}-\text{Index}^+[c] = \{ (r_2; 4, d), (r_4; 3, f) \} \) from \( \mathcal{R}-\text{Index}^+ \). The second entry matches the entry \( \{ r_4; 4, r_1; 2 \} \) in \( \mathcal{L} \), since \( c \) is on \( r_4 \) before position 4. Therefore, the algorithm identifies a sequence of links \((s; r_3; 1), (a; r_3; \emptyset), (a; r_2; 3), (c; r_2; \emptyset), (c; r_4; 3), (s; r_4; 4), (s; r_1; 2), (t; r_1; \emptyset) \) that leads to the target. After retrieving the required parts of routes \( r_1, r_2, r_3 \) and \( r_4 \), the answer path \((s, w, a, c, f, y, t) \) is constructed.

Note that for \( k = 2 \), the list \( \mathcal{L} \) would also contain link \( d \). In that case, the target list of LTST-2 would be exactly the same as that in Example 4.3, and LTST-2 would proceed identically to LTST.
4.5 Complexity Analysis

We use the notation introduced in Section 3.3. In addition, we assume that a disk page contains $|B_R|$ routes+ entries.

$\mathcal{R}$-Index+. The analysis is similar to $\mathcal{R}$-Index, substituting $B_R$ with $|B_R|$. LTS, LTST and LTS-k. Evaluating a PATH query according to the link traversal search paradigm consists of two phases. In the first, the target list $T$ is constructed, while in the second the collection is traversed.

The second phase is identical for all algorithms. At each iteration, after node $n_i$ is popped, they access the routes+[ni] at a cost of $O \left( \frac{|R|L^3}{|N|B_R} \right)$ I/Os. Note that the termination condition of the link traversal search algorithms incurs no I/O cost.

In the first phase, all algorithms retrieve list routes+[ni] at a cost of $O \left( \frac{|R|L}{|N|B_R} \right)$ I/Os. In addition, LTST retrieves from $\mathcal{T}$-Index the trans lists for each route in routes+[ni] at a cost of $O \left( \frac{|R|L^3}{|N|B_R} \right)$ I/Os. On the other hand, LTS-k retrieves each route referenced in routes+[ni] with $O \left( \frac{|R|L}{|N|B_R} \cdot \frac{|L|}{|B_N|} \right)$ I/Os, and then for each route it reads the routes+[ni] list of the $k$ nodes before the target with $O \left( k \cdot \frac{|R|L}{|N|B_R} \cdot \frac{|L|}{|B_N|} \right)$ I/Os.

Aggregating for $|N|$ nodes in the worst case traversal scenario, we obtain the following complexities. LTS requires $O \left( \frac{|R|L^2}{|N|B_R} + \frac{|R|L}{|N|B_R} \right)$ I/Os. LTST requires $O \left( \frac{|R|L^2}{|N|B_R} + \frac{|R|L^3}{|N|B_R} \right)$ I/Os. LTS-k requires $O \left( \frac{|R|L^2}{|N|B_R} + \frac{|R|L^2}{|N|B_R} + k \cdot \frac{|R|L}{|N|B_R} \right)$ I/Os.

5 Updating Route Collections

Section 5.1 discusses the case when new routes are added in the collection, while Section 5.2 addresses deletions. Note that all index structures are stored as inverted files on secondary storage. To handle frequent updates, we perform lazy updates, deferring propagation of changes to the disk by maintaining additional information in main memory. Then, at some time, a batch update process reflects all changes to the disk resident indices. Insertions are handled by merging memory-resident information with disk-based indices [19], while deletions require rebuilding of the affected lists.

5.1 Insertions

To support lazy updating for an insertion, we maintain a main memory list for each disk resident list affected. The main memory lists contain two types of entries. An entry prefixed with the + symbol is new and must be added to the disk-based list. An entry prefixed with the ± symbol exists on disk but must be updated.

Indices are updated in two phases. Buffering updates the memory resident lists and occurs online every time a new route is inserted in the collection. Flushing propagates all changes to the disk-based indices and is thus executed periodically offline. Between two flushing phases, the algorithms must also take into account the main memory lists. When retrieving a disk-based list: (1) all main memory (+ and ±) entries are also considered, and (2) all disk-based entries that have a corresponding ± main memory entry are ignored. In the following, we detail the two phases for each of the three indices used. $\mathcal{R}$-Index. Assume that a new route $r_i$ arrives. Then, for each node $n_j$ at position $o_{ji}$ in $r_i$, insert the entry $(r_i; o_{ji})$ at the end of the main memory list routes+$[nj]$ (the list may need to be constructed if it does not exist). Note that buffering requires no disk access.

In the flushing phase, each main memory list routes+$[nj]$ is merged with the corresponding disk-based list routes+$[nj]$. Recall that entries in routes+$[nj]$ are sorted ascending on the route identifier. Therefore, since all entries in routes+$[nj]$ are about new routes, the merging operation simply requires appending routes+$[nj]$ at the end of routes+$[nj]$.

$\mathcal{R}$-Index+. Assume that $r_i$ is added to the collection. For each node $n_j$ in $r_i$, main memory lists routes+$M$ are created similar to the buffering phase of $\mathcal{R}$-Index. However, an additional step is required, as the next link information in some entries may change. This is necessary when a node $n_j$ in the newly added route $r_i$ becomes a link. Let $r_k$ be the only route that $n_j$ belonged to before the update, and let $n_j^+$ (resp. $n_j^-$) denote the link immediately before (resp. after) $n_j$ in $r_k$. Then, when $r_i$ is added, all nodes in $r_k$ after $n_j^-$ and before $n_j$ should have node $n_j$ as their next link, instead of $n_j^+$.

After inserting a new route $r_i$, a node $n_j$ of $r_i$ becomes a link, if $n_j$ already appears in the collection but is not a link. Thus, to detect this case, we use the main memory data structure for distinguishing links from non-link nodes discussed in Section 4.4. In case $n_j$ becomes a link, we retrieve the route $r_k$ and identify all nodes after $n_j^-$ and before $n_j$, where $n_j^+$ is the first link before $n_j$. For each such node $n_m$, we insert into the main memory list routes+$M[nm]$ the entry $\pm \langle r_k: o_{mk}, n_j \rangle$.

In the flushing phase, if an $\mathcal{R}$-Index+$^+$ list contains only entries with the plus sign, it is simply appended at the end of the corresponding disk resident list, similar to the case of $\mathcal{R}$-Index. On the other hand, a routes+$M[nm]$ list that contains entries prefixed with ± must update those in routes+$[nm]$; this operation is similar to a merge-join of the two lists, as both are sorted on the route identifier.

$\mathcal{T}$-Index. As before, assume that a new route $r_i$ arrives. For each link $n_j$ at position $o_{ji}$ in $r_i$ retrieve the $\mathcal{R}$-Index (or $\mathcal{R}$-Index+$^+$) lists from disk and main memory. Then, for each entry $\langle r_k: o_{jk} \rangle$ ($r_i \neq r_k$) in these lists: (1) insert the entry $\langle r_i, n_j, o_{ji}, o_{jk} \rangle$ at the end of the main memory list trans+$M[r_i]$, and (2) insert the entry $\langle r_i, n_j, o_{jk}, o_{kj} \rangle$ at the end of the main memory list trans+$M[r_k]$. Note that the buffering phase for $\mathcal{T}$-Index requires retrieving from disk routes+$[nj]$ for each link in $r_i$ that is not new.

The flushing phase merges each memory resident list with the corresponding on the disk. Similar to the case of
R-Index, as the list \(\text{trans}^M[r_j]\) of an old route \(r_j\) contains only entries about new routes, \(\text{trans}^M[r_j]\) is appended at the end of \(\text{trans}[r_j]\). Finally, the list \(\text{trans}^M[r_i]\) of a newly added route \(r_i\) has no counterpart on the disk, and thus it becomes the disk-based list \(\text{trans}[r_i]\).

5.2 Deletions

Deletions need different treatment compared to insertions, as many entries across multiple lists are affected. Identifying them would require a large number of disk accesses. Therefore, a buffering phase does not occur when a route deletion arrives. Rather, a list is maintained that contains all routes deleted since the last flushing. Then, during the execution of a search, retrieved entries that contain a deleted route are simply discarded. This design choice may influence the performance of the link traversal search algorithms, mainly because the demotion of a link to a non-link node is not captured and thus non-link nodes may be visited. However, the deletion of a node is captured and hence the correctness of all algorithms is not affected. Next, we detail the flushing phase, which rebuilds the affected lists, for each index. R-Index and R-Index\(^+\). For each deleted route \(r_i\), retrieve the corresponding route from disk. For each node \(n_j\) of \(r_i\), retrieve the list \(\text{routes}[n_j]\) (or \(\text{routes}^+[n_j]\)) and delete the entry corresponding to \(n_j\). During this process, the main memory data structure for distinguishing links from non-link nodes is updated.

T-Index. Flushing of T-Index occurs in parallel to flushing R-Index/R-Index\(^+\). Assume that the routes\([n_j]\) (or routes\(^+[n_j]\)) list for the node \(n_j\) of the deleted route \(r_i\) is considered. Then, for each non-deleted entry \(\langle r_k: o_{jk}, n_j^+\rangle\) in the list, retrieve from T-Index the list \(\text{trans}[r_k]\) and remove from it the entry concerning \(r_i\). Additionally, delete the entire list \(\text{trans}[r_i]\).

6 EXPERIMENTAL EVALUATION

This section presents a detailed study of all algorithms introduced. Section 6.1 details the setting, while Sections 6.2, 6.3 and 6.4 evaluate index construction, querying and index maintenance, respectively, of all methods.

6.1 Experimental Setup

We study the route traversal methods, RTS and RTST, and the link traversal algorithms, LTS, LTST and LTS-k. To gauge performance we compare against conventional depth-first search (DFS) on the reduced routes graph \(G_{R}^r\). All algorithms are written in C++ and compiled with gcc. The evaluation is performed on a 3 Ghz Intel Core 2 Duo CPU with 4GB RAM running Debian Linux.

We generate synthetic route collections varying the following parameters (Table 5): (1) the number of routes in the collection, \(|R|\), (2) the route length, \(L_r\), (3) the number of distinct nodes in the routes, \(|N|\), and (4) the links/nodes ratio \(\alpha\). In each experiment, we vary one of the parameters while we keep the others to their default values.

### Table 5

<table>
<thead>
<tr>
<th>parameter</th>
<th>values</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>R</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>N</td>
<td>)</td>
</tr>
<tr>
<td>(\alpha)</td>
<td>0.2, 0.4, 0.6, 0.8, 1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

6.2 Index Size and Construction Cost

For each method, we measure the time spent to construct the necessary indices and their storage requirement. Table 6 shows the indices employed by each method.

### Table 6

Methods for evaluating PATII queries

<table>
<thead>
<tr>
<th>input</th>
<th>method name</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduced routes graph (G_{R}^r)</td>
<td>DFS</td>
<td>adjacency lists</td>
</tr>
<tr>
<td>route collection</td>
<td>LTS</td>
<td>R-Index &amp; T-Index</td>
</tr>
<tr>
<td></td>
<td>LTST</td>
<td>R-Index</td>
</tr>
<tr>
<td></td>
<td>LTS-(k)</td>
<td>R-Index(^+)</td>
</tr>
</tbody>
</table>

Varying the number of routes \(|R|\). The disk space requirement of the R-Index/R-Index\(^+\), employed by RTS, LTS, LTS-\(k\), and DFS, \(^2\) depends primarily on the number of nodes \(|N|\) (which determines the number of lists \(\text{routes}[]/\text{routes}^+[]\)) and not on \(|R|\) (which affects the length of the lists). Hence, Figure 8(a), shows that the space for these methods remains constant. Note that RTS and LTS/LTS-\(k\) exhibit the same space consumption, in terms of disk pages, although an R-Index\(^+\) compared to an R-Index entry contains additional information.

On the other hand, as \(|R|\) increases, the number of edges in the transition graph, and thus the size of the T-Index employed by RTST and LTST, quickly increases. The values above the RTST line quantify the difference between RTST and RTS (and between LTST and LTS/LTS-\(k\)), which corresponds to the T-Index size. The construction time for all indices, shown in Figure 9(a), increases as the collection becomes larger. The values above the RTST line measure the time required for building T-Index only. Note that for \(|R| = 500K\) the construction time of T-Index increases modestly compared to the six-fold increase of the index size. This occurs because the majority of the T-Index pages are written sequentially on the disk. \(^3\)

Varying the route length \(L_r\). The space consumption of the R-Index/R-Index\(^+\) does not change with \(L_r\), as shown in Figure 8(b). This is because, as explained in the context of Figure 8(a), the number of routes\([/\text{routes}^+[]\) lists remains fixed and while the lists become longer they still fit within the same number of pages. In contrast, the space for RTST and LTST increases rapidly with \(L_r\), since \(G_{T}\), encoded by T-Index, becomes denser.

The construction times for RTST and LTST in Figure 9(b) increase modestly with \(L_r\), although T-Index

\(^2\) In fact, DFS operates on the adjacency lists of the reduced routes graph, where a list contains for each adjacent link the routes it belongs to and its position in them; thus, an adjacency list contains equivalent information to the corresponding R-Index\(^+\) list.

\(^3\) On our system, a sequential access is around 350 times faster than a random one.
becomes much larger. This occurs because the number of random accesses (that depends on the number of \textit{trans} lists) remains constant, as the increase in \textit{T-Index}'s size is due to its lists occupying more pages. Recall, that the contents of a list are written sequentially on disk.

\textbf{Varying the number of nodes } $|N|$. The number of routes lists depends on $|N|$ and thus the total size of the $R$-Index scales linearly as shown in Figure 8(c). Increasing the number of nodes, while $|R|$ and $L_r$ remain fixed, causes an increase in the number of links (in absolute values). This makes each link appear fewer times in the routes and thus the number of edges in $G_T$ decreases. The space requirement of \textit{T-Index} decrease from 960MB to its minimum 400MB (1 page for each of the 100K routes). Figure 9(c) shows that the construction time for all methods decreases with $|N|$ due to the increase of the $R$-Index size.

\textbf{Varying the links/nodes ratio } $\alpha$. The space required for $R$-Index scales depends on the number of nodes $|N|$ and not on the links/nodes ratio, hence the constant lines in Figure 8(d). As the number of links increases ($|R|$ and $L_r$ remain fixed), the transition graph becomes sparser, as explained in the context of Figure 8(c); this accounts for the decrease in the space \textit{T-Index} occupies from 700MB to 400MB. Figure 9(d) shows that the construction times for all indices are unaffected by $\alpha$.

\section{6.3 Evaluating PATH Queries}

We study the efficiency of the proposed methods for processing PATH queries. All reported values are the averages taken by posing 5,000 distinct queries. Note that in Sections 6.3.1 and 6.3.2 all considered queries have an answer, i.e., a path exists; the case of queries with no answer is investigated in Section 6.3.3.

\subsection{6.3.1 Route vs link traversal search}

We compare the route traversal search methods RTS and RTST against the basic link traversal search algorithm LTS in terms of the execution time, while varying $|R|$, $L_r$, $|N|$ and $\alpha$ in Figures 10(a), (b), (c) and (d), respectively.

\textbf{Varying the number of routes } $|R|$. As $|R|$ increases, finding a path between two nodes becomes easier. This is exhibited by RTST and LTS in Figure 10(a). In contrast, the execution time of RTS increases with $|R|$ as it performs more iterations compared to RTST, which has a stronger termination condition, and to LTS, which only visits links.

\textbf{Varying the route length } $L_r$. The same observations hold when the route length increases in Figure 10(b). The performance of RTS deteriorates faster, since, in addition to requiring more iterations, each iteration costs more, as RTS inserts in the stack longer subsequences of routes.

\textbf{Varying the number of nodes } $|N|$. When $|N|$ increases, finding a path becomes harder, as shown in Figure 10(c). The advantage of RTST over RTS decreases with $|N|$, because the benefit of a stronger termination condition diminishes as the total execution time is dominated by the number of iterations required. The advantage of LTS over RTS decreases because the benefit of traversing the links diminishes as each link is contained in fewer routes. Note that even for large $|N|$, not examined in this experiments set, RTS can never outperform LTS as they employ the same termination condition and RTS will always need more iterations than LTS. The same argument carries to RTST compared to LTST.

\textbf{Varying the links/nodes ratio } $\alpha$. When the links/nodes ratio increases, there are more links in the collection, but each of them appears in fewer routes. In particular, when $\alpha = 0.2$ the link frequency, i.e., the (average) number

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig8}
\caption{Indices space consumption.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig9}
\caption{Indices construction time.}
\end{figure}
of routes in which it appears, is 46, and becomes 10 when $\alpha=1$. In general, as the ratio increases it becomes harder to find a path. Hence, as $\alpha$ increases LTS needs more iterations to reach the target, and its execution time increases. It is important to notice the behavior of RT5 and RTST. At each iteration and after a link is popped, these algorithms need to retrieve as many routes as the link frequency. This implies that the cost per iteration decreases with $\alpha$. On the other hand, since the number of iterations increases, the total execution time increases slightly for small $\alpha$ values, but ultimately decreases.

When $\alpha = 1$, all nodes are links (the reduced routes graph $G_R$ reduces to the $G_R$ graph) and thus LTS visits exactly the same nodes with RT5. Still LTS is around 8.6 times faster, as it performs fewer accesses per iteration. As before, the argument applies to LTST (not shown in the figure), which outperforms RTST.

### 6.3.2 Link traversal search vs DFS

In the following sets of experiments, we investigate the performance of LTS, LTST and LTS-$k$ ($k$ is set to 1, 3 and 5), using depth-first search DFS as the baseline.

To better understand the algorithms’ behavior, we distinguish two phases when processing a $\text{PATH}(n_s, n_t)$ query: initialization and core. In the first phase, all methods need to retrieve the first link $n_i^+$ before $n_i$ (resp. $n_i^+$ after $n_i$) when the target (resp. source) is not a link, as discussed in Section 4.1. In addition, the link traversal search methods assemble the target list $T$ by accessing the index structures. The core phase involves traversing the nodes and checking for termination.

In each setting, we measure the average value of: (1) the total execution time, (2) the cost of the initialization phase in terms of I/O operations, (3) the size of the target list in KBs, and (4) the number of iterations, i.e., nodes visited, during the core phase.

#### Varying the number of routes $|\mathcal{R}|$

As $|\mathcal{R}|$ increases, every link is contained in more routes and the number of iterations decreases, as shown in Figure 11(d). In the initialization phase (after retrieving the links following the source and preceding the target, if necessary), LTS retrieves only $\text{routes}^+[n_i]$ and assembles the target list; this has a constant cost as shown in Figure 11(b). On the other hand, LTST-$k$ and LTST retrieve multiple $\text{routes}^+[]$ and $\text{trans}[]$ lists, respectively, depending on the number of routes a link appears in. Since this factor increases with $|\mathcal{R}|$, the initialization cost also increases.

Similar observations apply for the size of the target list $T$, which increases for all methods as the $G_R$ graph becomes denser; see Figure 11(c). LTST has the largest while LTS has the smallest $T$. In comparison, the target list size for the LTS-$k$ methods increases with $k$ and ranges between that of LTS and LTST (recall that LTS-0 corresponds to LT5). However, all LTS-$k$ methods have higher initialization costs compared to LTST, because the latter has access to the $T$-Index. Note that the size of the target list $T$ portrays the strength of the termination condition; compare the trends in Figures 11(c) and 11(d). Among the link traversal search methods, LTS has the weaker and LTST the stronger termination condition.

Putting the cost of the two phases together, we reach the following conclusions. The total execution time of DFS and LTS decreases with $|\mathcal{R}|$, with LTS becoming up to one order of magnitude faster, as shown in Figure 11(a). Similarly, the processing time of LTST and LTS-$k$ decreases rapidly up to 100K routes, and LTST becomes more than two orders of magnitude faster than DFS. On the other hand, when the collection contains more than 100K routes (while the number of nodes and route length remain fixed) the initialization cost of the LTST, LTS-$k$ methods dominates the total time, as less than 10 iterations are required to find a path. Hence the execution time slightly increases for these methods.

#### Varying the route length $L_r$

Figure 12 illustrates the impact of varying $L_r$ on evaluating $\text{PATH}$ queries. As $L_r$ increases, every link is contained in more routes and the reduced $G_R$ graph becomes more dense. Therefore, all algorithms exhibit the same trends as in Figure 11(d). Note that LTS becomes the fastest method for $L_r = 50$ outperforming DFS by almost two orders of magnitude.

#### Varying the number of nodes $|\mathcal{N}|$

Figure 13 studies the effect of increasing $|\mathcal{N}|$, while the number of routes in the collection and route length remain fixed. As $|\mathcal{N}|$ increases, even though the number of links increases, each of them is contained in fewer routes. Therefore, the reduced routes graph $G_R$ becomes sparser, which means that finding a path becomes harder. This is verified in Figure 13(c), which depicts that the target list decreases with $|\mathcal{N}|$, and in Figure 13(d), which shows that more nodes are visited as $|\mathcal{N}|$ increases.

Subsequently, the initialization cost of LTST and LTS-$k$ decreases with $|\mathcal{N}|$. As explained in the context of Figure 11, since LTS accesses a single $\text{routes}^+[]$ list, its
initialization cost is independent of the number of nodes.

The total execution time of DFS and LTS increases with $|N|$, as they perform more iterations. This also holds for LTST, LTS-$k$ for collections of more than 100,000 nodes. For fewer nodes, the initialization cost of LTST, LTS-$k$ (see Figure 13(b)) dominates the total execution time, which decreases. In the worst case, LTS and LTST are 1.6 and 16 times, respectively, faster than DFS.

**Varying the links/nodes ratio $\alpha$.** As $\alpha$ increases the link frequency decreases and finding a path becomes more difficult, as shown in Figure 14(d). For the reasons discussed in the case of varying $|N|$, the target list of all methods decreases with $|N|$ (see Figure 14(c)), and the initialization cost of LTST and LTS-$k$ decreases (see Figure 14(b)). Correspondingly, the total execution time increases for DFS and LTS, while it first decreases and ultimately increases for LTST and LTS-$k$.

6.3.3 PATH queries with no answer

We study the performance of LTS, LTST and LTS-$k$ compared to DFS for queries that return no answer, i.e., no path exists between the source and the target. The collections in this section, induce a reduced routes graph with two components that share no edge. We perform 5,000 PATH queries selecting the source from one
component and the target from the other so that a path never exists.

Figures 15(a) and 15(b) display the total execution time and the cost of the initialization phase while the number of routes varies. In accordance to Figure 11(b), the initialization cost of LTST and LTS-k increases with |R|, while that of LTS remains fixed. In the core phase, all methods perform the same iterations, traversing all links in the component of the source. The number of links in the component do not change with |R|. Furthermore, since the execution time is dominated by the traversal cost, all methods require around 250 seconds to determine that a path does not exist, as shown in Figure 15(b). The total execution time of LTST and LTS-k, slightly increases with |R| due to the higher initialization cost. In the worst case, the overhead is less than 1.3%.

Figures 15(c) and 15(d) repeat the measurements while the number of nodes varies. As before, the execution time is dominated by the cost of the core phase, which is identical for all methods. However, since the number of nodes in the source component increases, the performance of all methods degrades with N. In the worst case, when N = 20K, LTS-5 is about 6% slower than DFS.

6.4 Index Maintenance

In this section, we evaluate the performance of all methods in terms of (1) the cost of the buffering phase, (2) the cost of the flushing phase, and (3) the performance hit introduced by not immediately updating the indices, while routes are added and deleted from a collection initially containing 50,000 routes of length L_r = 10 and 100,000 nodes with the fraction of α = 0.6 being links.

At the buffering phase, each insertion and deletion is treated independently. Thus, we only discuss the case of a single update. All methods require no disk access for a deletion. RTS performs no I/O for an insertion. DFS, LT5, and LTS-k must retrieve for each node in the new route that becomes a link, its other route; with L_r = 10 and α = 0.6, this costs 4 random I/Os (and maybe a few sequential I/Os). RTST and LTST (in addition to the operations needed by LT5) must retrieve from disk the routes[]/routes+[] list for each link in the route; with L_r = 10 and all nodes becoming links this costs 10 random I/Os (and maybe a few sequential I/Os).

Figure 16(a) shows the cost of the flushing phase as we vary the number of updates from 1,000 to 20,000; the ratio of insertions to deletions is fixed to 75%/25%.

The values above the RTST (resp. LTST) line measure the time required for updating T-Index only. The cost of all methods increases sublinearly justifying our lazy updates strategy. Note that the flushing cost of DFS, LTS, LTS-k is higher than the cost of RTST due to the additional pages retrieved when nodes become links. The same observations hold for LTST and RTST. An important observation is that for more than 15,000 updates, the flushing cost for the R-Index/R-Index+ based methods is higher than building the indices from scratch. On the other hand, even when 20,000 updates occur (40% of |R|), the flushing cost for the T-Index based methods is lower than the cost of rebuilding indices.

Figure 16(b) investigates the cost for 10,000 updates as we vary the insertions/deletions ratio. When the number of deletions increases, the G_T contains fewer edges and the cost to update T-Index becomes smaller. As before, the values above the RTST (resp. LTST) line measure the time required for updating T-Index only.

Finally, we study how our lazy updating strategy affects the performance of all methods. We investigate two scenarios: the first assumes that the flushing phase has been performed, and the second assumes the opposite. Intuitively, the former simulates the ideal scenario where all updates are immediately reflected on the disk resident indices. We perform 5,000 PATH queries and measure the relative performance of each method, i.e., its execution time in the second scenario divided by that of the first.

Figure 16(c) shows the performance hit as we vary the number of updates while the insertions/deletions ratio is fixed to 75%/25%. The execution time of all methods increases with the number of updates but stays within 13% of the execution time in the ideal scenario. Figure 16(d) keeps the number of updates fixed to 10,000 and varies the insertions/deletions ratio. In this setting, the performance hit of the RTS and RTST becomes more pronounced as the number of deletions increases. Note that it is possible for a method to execute faster when the flushing has not been performed, as parts of the disk-based indices are kept in memory. This appears in Figure 16(d) at the 100%/0% insertions/deletions ratio.

7 CONCLUSIONS

We consider the problem of evaluating path queries on large disk-resident routes collection that are frequently updated. We introduced two generic search-
Based paradigms, route traversal search and link traversal search, that exploit local transitivity information to expedite path query evaluation. The involved index structures and their maintenance strategies are designed to cope with frequent updates.

REFERENCES