Evaluating Path Queries over Frequently Updated Route Collections

TR-2010-6

Panagiotis Bouros, Dimitris Sacharidis, Theodore Dalamagas,
Spiros Skiadopoulos, Timos Sellis

Knowledge and Database Systems Laboratory,
National Technical University of Athens, Greece

October, 2010

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.

Chapter 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 1.1 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.

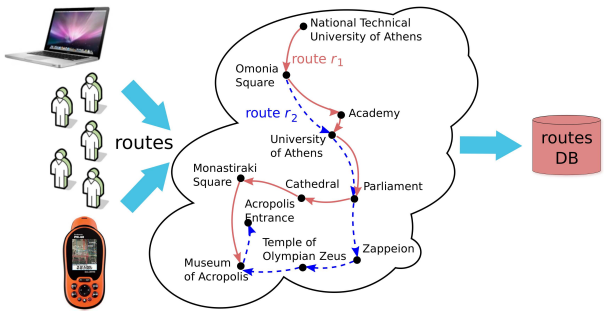


Figure 1.1: Examples of routes in city of Athens.

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 1.1 and a path query: "Find a path from Academy to Zappeion following existing routes". Note that a path may contain nodes from different routes, since reaching n_t from n_s may require changing routes using *links*, i.e., nodes shared among routes. For instance, an answer to the previous query is the path: Academy, University of Athens (changing from r_1 to r_2), Parliament, Zappeion.

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 [3], 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 [3], based on the first paradigm. RTS employs an inverted file, $\mathcal{R}$-$Index$, on the nodes of the route collection. RTST uses the $\mathcal{T}$-$Index$ (in addition to $\mathcal{R}$-$Index$) 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 $\mathcal{R}$-$Index$ and features a similar termination condition to RTS. Similarly, LTST has a stronger condition based on the $\mathcal{T}$-$Index$. The LTS-k algorithm forgoes the high storage and maintenance cost of $\mathcal{T}$-$Index$ 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 report is structured as follows. Chapter 2 estab-

lishes the necessary background and discusses the related work in detail. Chapter 3 discusses route traversal search, and Chapter 4 introduces the link traversal search paradigm. Then, Chapter 5 discusses maintenance of the index structures under frequent updates of the route collection. Chapter 6 demonstrates our experimental results and Chapter 7 concludes the report.

Chapter 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 2.1.1 (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 $L_r = |nodes(r)|$.

Definition 2.1.2 (Route collection) A *route collection* $\mathbf{R}$ over N is a set of routes $\{r_1, \ldots, r_m\}$ over N. □

We denote all nodes in a route collection as $nodes(\mathbf{R})$.

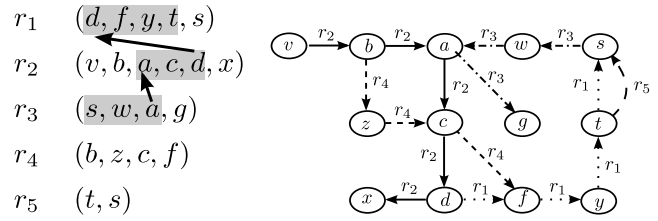
Definition 2.1.3 (Link) A node in $nodes(\mathbf{R})$ is called *link* if it is contained in at least two routes in $\mathbf{R}$. □

Example 2.1.1 Figure 2.1(a) illustrates a route collection $\mathbf{R} = \{r_1, r_2, r_3, r_4, r_5\}$. Nodes a, b, c, d, f, s, t are links. □

Definition 2.1.4 (Path) A *path* on a route collection $\mathbf{R}$ is a sequence of distinct nodes $(n_1, \ldots, n_k) \in nodes(\mathbf{R})$, 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 $\mathbf{R}$.

Note that a path may involve parts of routes from $\mathbf{R}$.

Definition 2.1.5 (`PATH` queries) Let $\mathbf{R}$ be a route collection, and n_s and n_t be two nodes in $nodes(\mathbf{R})$. The *path query* $\texttt{PATH}(n_s, n_t)$ returns a path



(a) Route collection $\mathbf{R}$ (b) Routes graph $G_{\mathbf{R}}$

Figure 2.1: A route collection $\mathbf{R}$, an answer to $\texttt{PATH}(s, t)$, and routes graph $G_{\mathbf{R}}$.

from n_s to n_t on $\mathbf{R}$. □

Example 2.1.2 Consider the route collection in Figure 2.1(a). Path (s, w, a, c, d, f, y, t) is an answer to query $\texttt{PATH}(s, t)$, 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 $\mathbf{R}$ can be easily mapped to a graph by merging all routes in $\mathbf{R}$.

Definition 2.1.6 (Routes graph) The *routes graph* of a route collection $\mathbf{R}$ is a labeled directed graph $G_{\mathbf{R}}(N, E)$, where $N = \mathit{nodes}(\mathbf{R})$, and an edge $(n_i, n_j, r_k) \in E$ if there exists a route $r_k \in \mathbf{R}$ with n_j immediately after n_i. □

Example 2.1.3 The collection $\mathbf{R}$ in Figure 2.1(a) is mapped to routes graph $G_{\mathbf{R}}$ in Figure 2.1(b). The different line styles of the edges denote the five routes in $\mathbf{R}$. □

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 2.1.3. 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 $\mathbf{R}$ can be answered directly on routes graph $G_{\mathbf{R}}$ 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* $\texttt{PATH}(n_s, n_t)$ identifies a path from n_s to n_t, while a *reachability query* $\texttt{REACH}(n_s, n_t)$ answers if such a path exists. Thus, an answer to $\texttt{PATH}(n_s, n_t)$ provides an answer to $\texttt{REACH}(n_s, n_t)$, 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 2.1 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.

category	**method**	**graph type**	`REACH`	`PATH`	**maintenance**
searching	depth/breadth-first search [9]	all types	yes	yes	update adjacency lists
TC encoding	2-hop [7]	all types, but small	yes	by including first-edge	not discussed
	HOPI [15, 16]	all types	yes	partially	based on method of [15]
	geometric [5] and graph partitioning 2-hop [6]	DAG	yes	not discussed	not discussed
	updatable 2-hop [4]	DAG	yes	not discussed	based on node-separation
	3-hop [11]	DAG	yes	not discussed	not discussed
	interval labelling [1]	DAG	yes	by computing ancestors	gaps in postorder numbers
	dual labeling [18]	DAG	yes	not discussed	not discussed
	GRIPP [17]	all types	yes	by computing descendants	not discussed
	path-cover [12]	DAG	yes	not discussed	not discussed

Table 2.1: Summary of related work on handling reachability and path queries on graphs.

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 [9]. 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 $G_{\mathbf{R}}(N, E)$ is the graph $G^*_{\mathbf{R}}(N, E^*)$, where an edge (n_i, n_j) is in E^* if a path from n_i to n_j exists in $G_{\mathbf{R}}$. 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., [2, 14, 10], the computation and storage cost are prohibitive for large disk-resident graphs. Therefore, various methods compress the TC.

2-hop [7, 8] identifies a set of nodes, called centers, that best capture the reachability information of a graph as intermediates. Thus, for each node n, 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 (resp. n_c) to n_c (resp. n) must also be stored.

Computing the optimal 2-hop scheme is NP-hard. The work in [7] and [8] presents an approximation algorithm based on set covering [13] that constructs a 2-hop scheme larger by a logarithmic $O(log|N|)$ 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 [15, 16] reduces the construction time of 2-hop by exploiting graph partitioning, which works well for forests with few connections between the different sub-graphs, e.g., collections of XML documents. Updates are handled by applying the construction method of [15]. 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, [5] proposes a geometric-based method and [6] another one based on graph partitioning for the efficient construction of 2-hop. [11] 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. [1] 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 [12] 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. [18] proposes dual-labeling for sparse graphs. [4] 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 [17] 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, [17] does not handle frequent updates.

Chapter 3

Route Traversal Search

This chapter revisits our work from [3] for evaluating `PATH` queries over route collections. Section 3.1 presents the $\mathcal{R}$-*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 $\mathcal{T}$-*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 $\mathcal{R}$-*Index* on the route collection which associates a node with the routes that contain it.

Definition 3.1.1 ($\mathcal{R}$-*Index*) Given a route collection $\mathbf{R}$ and a node $n_i \in nodes(\mathbf{R})$, $routes[n_i]$ is the ordered list of $\langle r_j : o_{ij} \rangle$ entries for all routes r_j that include n_i at their o_{ij}-th position, sorted on the route identifier r_j. $\mathcal{R}$-*Index* contains the lists $routes[n_i]$ for all $n_i \in nodes(\mathbf{R})$. □

Example 3.1.1 Table 3.1 illustrates the $\mathcal{R}$-*Index* for the routes shown in Figure 2.1(a). □

Figure 3.1 illustrates the pseudocode of the RTS algorithm. The algorithm takes as inputs: a route collection $\mathbf{R}$, the $\mathcal{R}$-*Index*, the source n_s and target node n_t and returns a path from n_s to n_t, if one exists, and null otherwise. The algorithm uses the following data structures: (1) a search stack $\mathcal{Q}$, (2) a history set $\mathcal{H}$, which contains all nodes that have been pushed in $\mathcal{Q}$, and (3) an ancestor set $\mathcal{A}$, which stores the direct ancestor of each

node	*routes*[] **list**
a	$\langle r_2 : 3 \rangle, \langle r_3 : 3 \rangle$
b	$\langle r_2 : 2 \rangle, \langle r_4 : 1 \rangle$
c	$\langle r_2 : 4 \rangle, \langle r_4 : 3 \rangle$
d	$\langle r_1 : 1 \rangle, \langle r_2 : 5 \rangle$
f	$\langle r_1 : 2 \rangle, \langle r_4 : 4 \rangle$
g	$\langle r_3 : 4 \rangle$
s	$\langle r_1 : 5 \rangle, \langle r_3 : 1 \rangle, \langle r_5 : 2 \rangle$
t	$\langle r_1 : 4 \rangle, \langle r_5 : 1 \rangle$
v	$\langle r_2 : 1 \rangle$
w	$\langle r_3 : 2 \rangle$
x	$\langle r_2 : 6 \rangle$
y	$\langle r_1 : 3 \rangle$
z	$\langle r_4 : 2 \rangle$

Table 3.1: The $\mathcal{R}$-*Index* for the route collection $\mathbf{R}$.

Algorithm RTS
Input: nodes n_s and n_t of a route collection $\mathbf{R}$, $\mathcal{R}$-*Index*
Output: a path from n_s to n_t
Parameters:
stack $\mathcal{Q}$: // the search stack
set $\mathcal{H}$: // contains all nodes pushed in $\mathcal{Q}$
set $\mathcal{A}$: // contains the direct ancestor of each node in $\mathcal{H}$

Method:

1. push$(n_s, \mathcal{Q})$;
2. **insert** n_s in $\mathcal{H}$
3. **while** $\mathcal{Q}$ is not empty **do**
4. **let** n_q = pop$(\mathcal{Q})$;
5. **if** there is a route $r_c \in \mathbf{R}$ containing n_q before n_t **then**
   return ConstructPath$(n_s, n_q, n_t, \mathcal{A}, r_c)$;
6. **for** each entry $\langle r_i : o_{qi} \rangle$ in $routes[n_q]$ **do**
7. **let** n_r be the node after n_q in r_i;
8. **while** $n_r \notin \mathcal{H}$ **do**
9. push$(n_r, \mathcal{Q})$;
10. **insert** n_r in $\mathcal{H}$;
11. **insert** $\langle n_r, n_r^- \rangle$ in $\mathcal{A}$;
12. **let** n_r be the next node in r_i;
13. **end while**
14. **end for**
15. **end while**
16. **return null**;

Figure 3.1: The RTS algorithm.

node in $\mathcal{Q}$. $\mathcal{H}$ is used to avoid cycles during the traversal, and $\mathcal{A}$ to extract answer paths. Note that RTS visits each node once and, thus, there is a single entry per node in $\mathcal{A}$.

The RTS algorithm proceeds similarly to depth-first search as follows.

Initially, the stack $\mathcal{Q}$ and $\mathcal{H}$ contain source node n_s (Lines 1–2). Then, RTS proceeds iteratively (Lines 3–15) popping a single node n_q from $\mathcal{Q}$ at each iteration (Line 4). The algorithm terminates when there exists a route r_c that contains both n_q and target n_t, such that n_t comes after n_q (Line 5). Specifically, to check if the above condition holds, RTS looks for entries $\langle r_c : o_{qc} \rangle$ and $\langle r_c : o_{tc} \rangle$ in lists $routes[n_q]$ and $routes[n_t]$ of $\mathcal{R}$-$Index$ respectively, such that $o_{qc} < o_{tc}$. The procedure is similar to a merge-join, as both $routes[n_q]$ and $routes[n_t]$ lists are sorted by the route identifier, that finishes when a common route r_c is found.

If such a common route r_c is identified, the search terminates and the answer path is extracted by the ConstructPath procedure. Specifically, starting from n_q, ConstructPath uses the ancestor information of $\mathcal{A}$ to backtrack to source n_s constructing $(n_s, \ldots, n_q)$ path. Then, it concatenates path $(n_s, \ldots, n_q)$ with the part of route r_c 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_c is not found, RTS expands the search retrieving $routes[n_q]$ and considering all routes that contain n_q (Lines 6–14). For each such route r_i and for each node n_r after n_q in r_i that is not in $\mathcal{H}$ (i.e., it has never been pushed in $\mathcal{Q}$) the algorithm performs the following tasks. Node n_r is pushed in $\mathcal{Q}$ and inserted in $\mathcal{H}$ (Lines 9–10). In addition, the entry $\langle n_r^-, n_r \rangle$, where n_r^- is the direct ancestor of n_r in route r_i, is inserted in $\mathcal{A}$ (Line 11). The fact that only new nodes are inserted in $\mathcal{Q}$, ensures that RTS avoids cycles.

Example 3.1.2 We illustrate the RTS algorithm for the $\texttt{PATH}(s, t)$ query on the route collection of Figure 2.1(a) using the $\mathcal{R}$-$Index$ presented in Table 3.1. Initially, we have: $\mathcal{Q} = \{s\}$, $\mathcal{H} = \{s\}$ and $\mathcal{A} = \{\}$. At the first iteration, RTS pops s from $\mathcal{Q}$ and checks for termination joining lists $routes[s] = \{\langle r_1 : 5 \rangle, \langle r_3 : 1 \rangle, \langle r_5 : 2 \rangle\}$ of current search node s with $routes[t] = \{\langle r_1 : 4 \rangle, \langle r_5 : 1 \rangle\}$ 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 $\mathcal{Q}$, $\mathcal{H}$ and $\mathcal{A}$ since there are no nodes after s. When processing $r_3(s, w, a, g)$, the algorithm adds to $\mathcal{Q}$ and $\mathcal{H}$, nodes w, a and g, and to $\mathcal{A}$, pairs $\langle s, w \rangle$, $\langle w, a \rangle$ and $\langle a, g \rangle$. After the fourth iteration, we have $\mathcal{Q} = \{w, c, d\}$, $\mathcal{H} = \{s, w, a, g, c, d, x\}$, and $\mathcal{A} = \{\langle s, w \rangle, \langle w, a \rangle, \langle a, g \rangle, \langle a, c \rangle, \langle c, d \rangle, \langle d, x \rangle\}$.

At the fifth iteration, d is popped. Then, RTS joins lists $routes[d] = \{\langle r_1 : 1 \rangle, \langle r_2 : 5 \rangle\}$ with $routes[t] = \{\langle r_1 : 4 \rangle, \langle r_5 : 1 \rangle\}$ and identifies entries $\langle r_1 : 1 \rangle$, $\langle r_1 : 4 \rangle$ 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 $\mathcal{A}$) 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 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 $\mathcal{T}$-*Index* structure.

Definition 3.2.1 (Transition graph) The *transition graph* of a route collection $\mathbf{R}$ is a labeled undirected graph $G_T(\mathbf{R}, E_T)$, where its vertices are the routes in $\mathbf{R}$, and a labeled edge (r_i, r_j, n_ℓ) exists in E_T if r_i and r_j share a link node n_ℓ. □

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

Example 3.2.1 Figure 3.2 illustrates the transition graph for the route collection of Figure 2.1(a). □

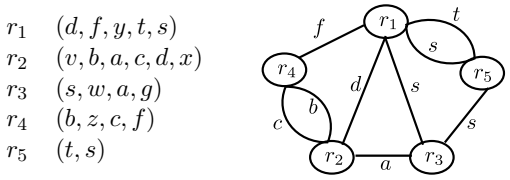


Figure 3.2: Transition graph for the route collection $\mathbf{R}$.

The transition graph is stored in a modified adjacency list representation denoted as $\mathcal{T}$-*Index*.

Definition 3.2.2 ($\mathcal{T}$-*Index*) Given a route collection $\mathbf{R}$, for each route $r_i \in \mathbf{R}$, $trans[r_i]$ is the ordered list of $\langle r_j, n_\ell : o_{\ell i} : o_{\ell j} \rangle$ entries for all (r_i, r_j, n_ℓ) edges of G_T, where $o_{\ell i}$ and $o_{\ell j}$ denote the position of the link n_ℓ in routes r_i and r_j, respectively. The entries are sorted on the route identifier r_j solving ties with $o_{\ell i}$. $\mathcal{T}$-*Index* contains the lists $trans[r_i]$ for all routes $r_i \in \mathbf{R}$. □

Example 3.2.2 Table 3.2 illustrates the $\mathcal{T}$-*Index* for the G_T graph presented in Figure 3.2. □

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

route	$trans[]$ **list of route in** G_T
r_1	$\langle r_2, d{:}1{:}5\rangle, \langle r_3, s{:}5{:}1\rangle, \langle r_4, f{:}2{:}4\rangle, \langle r_5, t{:}4{:}1\rangle, \langle r_5, s{:}5{:}2\rangle$
r_2	$\langle r_1, d{:}5{:}1\rangle, \langle r_3, a{:}3{:}3\rangle, \langle r_4, b{:}2{:}1\rangle, \langle r_4, c{:}4{:}3\rangle$
r_3	$\langle r_1, s{:}1{:}5\rangle, \langle r_2, a{:}3{:}3\rangle, \langle r_5, s{:}1{:}2\rangle$
r_4	$\langle r_1, f{:}4{:}2\rangle, \langle r_2, b{:}1{:}2\rangle, \langle r_2, c{:}3{:}4\rangle$
r_5	$\langle r_1, t{:}1{:}4\rangle, \langle r_1, s{:}2{:}5\rangle, \langle r_3, s{:}2{:}1\rangle$

Table 3.2: $\mathcal{T}$-$Index$ for the transition graph of Figure 3.2.

If the above hold, then a path from n_q to target n_t *via* link n_ℓ exists, and thus a path from source n_s to n_t can be found. To perform this check RTST scans lists $trans[r_i]$ and $routes[n_t]$ from $\mathcal{T}$-$Index$ and $\mathcal{R}$-$Index$, respectively, similar to a merge-join as both lists are sorted on the route identifier. The scan terminates when entries $\langle r_j, n_\ell : o_{\ell i} : o_{\ell j}\rangle$ of $trans[r_i]$ and $\langle r_j : o_{tj}\rangle$ of $routes[n_t]$ match, i.e., when the following conditions are satisfied:

(1) the entries correspond to the same route r_j,

(2) r_i contains link n_ℓ after n_q, i.e., $o_{\ell i} > o_{qi}$, and

(3) r_j contains n_ℓ before n_t, i.e., $o_{\ell j} < o_{tj}$

Example 3.2.3 Consider query PATH(s, t) on the route collection in Figure 2.1(a) indexed by the $\mathcal{R}$-$Index$ of Table 3.1 and $\mathcal{T}$-$Index$ of Table 3.2. The first two iterations of the RTST algorithm are identical to those of RTS in Example 3.1.2. Then, the third iteration processes a. According to $\mathcal{R}$-$Index$, a is contained in routes $r_2(v, b, a, c, d, x)$ and $r_3(s, w, a, g)$. To check the termination condition for r_2, RTST joins list $trans[r_2]$ of $\mathcal{T}$-$Index$ with $routes[t]$ of $\mathcal{R}$-$Index$. This results in the common route r_1 (condition (1)) with link d of (r_2, 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 $\mathbf{R}$, let $|\mathbf{R}|$ 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 $B_\mathcal{N}$ nodes, $B_\mathcal{R}$ $routes[]$ entries, and $B_\mathcal{T}$ $trans[]$ entries.

$\mathcal{R}$-$Index$. The $\mathcal{R}$-$Index$ structure contains an entry for each node in every route, for a total of $|\mathbf{R}| \cdot L_r$ entries. Therefore, it occupies $O\left(\frac{|\mathbf{R}| \cdot L_r}{B_\mathcal{R}}\right)$ pages. For the construction of $\mathcal{R}$-$Index$, the entire collection must be accessed at a cost of $O(\frac{|\mathbf{R}| \cdot L_r}{B_\mathcal{N}})$ I/Os, while the index must be stored at a cost of $O\left(\frac{|\mathbf{R}| \cdot L_r}{B_\mathcal{R}}\right)$

I/Os. An important factor in the performance of the algorithms is the size, in terms of entries, $|routes[]|$ of a $\mathcal{R}$-$Index$ list. In the average case, each list has the same number of entries, i.e., $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|}\right)$. In the worst case, a node can be contained in all routes, i.e., the $routes[]$ list has $O\left(|\mathbf{R}|\right)$ entries. In the sequel, we assume the average case holds.

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

The second task is the termination condition. RTS and RTST need to retrieve the $routes[n_t]$ list of the target incurring $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B_{\mathcal{R}}}\right)$ I/Os. RTST additionally retrieves the $trans[r_i]$ list of $\mathcal{T}$-$Index$ for each route r_i contained in $routes[n_q]$, at a cost of $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|}\cdot\frac{|\mathbf{R}|\cdot L_r^2}{|N|\cdot B_{\mathcal{T}}}\right)$ I/Os.

Aggregating for $|N|$ nodes in the worst case scenario, we obtain the following complexities. RTS requires $O\left(\frac{|\mathbf{R}|\cdot L_r}{B_{\mathcal{R}}}+\frac{|\mathbf{R}|\cdot L_r^2}{B_{\mathcal{N}}}\right)$ I/Os. RTST requires $O\left(\frac{|\mathbf{R}|\cdot L_r}{B_{\mathcal{R}}}+\frac{|\mathbf{R}|\cdot L_r^2}{B_{\mathcal{N}}}+\frac{|\mathbf{R}|^2\cdot L_r^3}{|N|\cdot B_{\mathcal{T}}}\right)$ I/Os.

Chapter 4

Link Traversal Search

Section 4.1 discusses the shortcomings of the algorithms in Chapter 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 Chapter 3 perform fewer iterations than conventional depth-first search on the route collection graph $G_{\mathbf{R}}$, 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_ℓ, which is the link immediately before n_q. Observe that if the termination condition does not hold at n_ℓ, then it neither holds at n_q. To make matters worse, retrieving $routes[n_q]$ 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 $routes[n_q]$ and $routes[n_t]$ from $\mathcal{R}$-$Index$, while RTST additionally retrieves all lists $trans[r_i]$ from $\mathcal{T}$-$Index$ for each r_i included in $routes[n_q]$. 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 $\mathcal{Q}$ (and consequently of sets $\mathcal{H}$, $\mathcal{A}$). 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 algorithms avoid visiting non-link nodes and *conceptually* traverse the *reduced routes graph* $G^-_{\mathbf{R}}(N^-, E^-)$ of the route collection, where $N^- \subseteq nodes(\mathbf{R})$ contains all links, and E^- contains all labeled directed edges (n_i, n_j, r_k) such that there exists a route $r_k \in \mathbf{R}$ in which n_j is the link immediately following n_i. Note that $G^-_{\mathbf{R}}$ is *not explicitly materialized*, and is introduced to better illustrate the link traversal search paradigm. For example, Figure 4.1 shows the reduced routes graph for the collection $\mathbf{R}$ of Figure 2.1(a). Observe the differences between $G_{\mathbf{R}}$ in Figure 2.1(b) and the reduced routes graph $G^-_{\mathbf{R}}$.

r_1 (d, f, y, t, s)
r_2 (v, b, a, c, d, x)
r_3 (s, w, a, g)
r_4 (b, z, c, f)
r_5 (t, s)

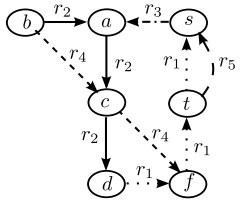


Figure 4.1: Reduced routes graph for the route collection $\mathbf{R}$.

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. preceding) it; if no such link exists, then no path can be found. [1] Under this assumption, a $\texttt{PATH}(n_s, n_t)$ query on $\mathbf{R}$ is equivalent to finding a path from n_s to n_t in $G^-_{\mathbf{R}}$, 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 *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 $G^-_{\mathbf{R}}$.

4.2 The LTS Algorithm

The *Link Traversal Search* (LTS) algorithm features a termination 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 $\mathcal{R}$-$Index^+$, 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 4.1: The $\mathcal{R}$-$Index^+$ for the route collection $\mathbf{R}$.

node	$routes^+[]$ **list**
a	$\langle r_2{:}3, c\rangle, \langle r_3{:}3, \varnothing\rangle$
b	$\langle r_2{:}2, a\rangle, \langle r_4{:}1, c\rangle$
c	$\langle r_2{:}4, d\rangle, \langle r_4{:}3, f\rangle$
d	$\langle r_1{:}1, f\rangle, \langle r_2{:}5, \varnothing\rangle$
f	$\langle r_1{:}2, t\rangle, \langle r_4{:}4, \varnothing\rangle$
g	$\langle r_3{:}4, \varnothing\rangle$
s	$\langle r_1{:}5, \varnothing\rangle, \langle r_3{:}1, a\rangle, \langle r_5{:}2, \varnothing\rangle$
t	$\langle r_1{:}4, s\rangle, \langle r_5{:}1, s\rangle$
v	$\langle r_2{:}1, b\rangle$
w	$\langle r_3{:}2, a\rangle$
x	$\langle r_2{:}6, \varnothing\rangle$
y	$\langle r_1{:}3, t\rangle$
z	$\langle r_4{:}2, c\rangle$

Definition 4.2.1 ($\mathcal{R}$-$Index^+$) Given a route collection $\mathbf{R}$ and a node $n_i \in nodes(\mathbf{R})$, $routes^+[n_i]$ is the ordered list of $\langle r_j{:}o_{ij}, n_i^+\rangle$ entries for all routes r_j that include n_i at their o_{ij}-th position, where n_i^+ is the link immediately following n_i in r_j if one exists, or $\varnothing$ otherwise. The entries are sorted on the route identifier r_j. $\mathcal{R}$-$Index^+$ contains the lists $routes^+[n_i]$ for all $n_i \in nodes(\mathbf{R})$. □

Example 4.2.1 Table 4.1 illustrates the $\mathcal{R}$-$Index^+$ for the routes shown in Figure 2.1(a). □

Note that the $\mathcal{R}$-$Index^+$ 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 4.2 presents the pseudocode of the LTS algorithm for evaluating a PATH(n_s, n_t) query. Similar to RTS, it uses stack $\mathcal{Q}$, and sets $\mathcal{H}$ and $\mathcal{A}$. Initially, $\mathcal{Q}$ contains the source n_s (Line 1). LTS constructs a target list $\mathcal{T}$ that contains entries $\langle r_i{:}o_{ti}\rangle$ for all routes that contain the target n_t (Lines 3–5). Then, LTS proceeds iteratively (Lines 6–16) until $\mathcal{Q}$ is depleted. At each iteration, assuming the current search link popped from the stack is n_q (Line 7), LTS examines each entry of $routes^+[n_q]$ (Lines 8–14).

The algorithm terminates if there exists an entry in $\mathcal{T}$ 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, ConstructPath composes an answer path using the information in $\mathcal{A}$. Otherwise, if the next link node n_q^+ has not been previously visited, it is pushed in the stack and in $\mathcal{H}$. Further, the entry $\langle n_q^+, n_q : r_i : o_{qi}\rangle$ is inserted in $\mathcal{A}$ indicating that n_q^+ is reached from n_q

following route r_i. The position o_{qi} is used by ConstructPath to quickly identify the subroute of r_i between links n_q and n_q^+ if required, as explained in Section 4.1.

Algorithm LTS
Input: n_s, n_t, $\mathcal{R}$-$Index^+$
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 $\mathcal{T}$:	stores all routes that contain target n_t

Method:

1. **push** n_s in $\mathcal{Q}$;
2. **insert** n_s in $\mathcal{H}$;
3. **for** each entry $\langle r_i : o_{ti}, n_t^+ \rangle$ in $routes^+[n_t]$ **do**
4. **insert** $\langle r_i : o_{ti} \rangle$ in $\mathcal{T}$;
5. **end for**
6. **while** $\mathcal{Q}$ is not empty **do**
7. **let** $n_q = \mathsf{pop}(\mathcal{Q})$;
8. **for** each entry $\langle r_i : o_{qi}, n_q^+ \rangle$ in $routes^+[n_q]$ **do**
9. **if** there is an entry $\langle r_i : o_{ti} \rangle$ in $\mathcal{T}$ such that $o_{qi} < o_{ti}$ **then**
return ConstructPath$(n_s, n_q, n_t, \mathcal{A}, r_i : o_{ti})$;
10. **if** $n_q^+ \notin \mathcal{H}$ **then**
11. **push** n_q^+ in $\mathcal{Q}$;
12. **insert** n_q^+ in $\mathcal{H}$;
13. **insert** $\langle n_q : r_i : o_{qi}, n_q^+ \rangle$ in $\mathcal{A}$;
14. **end if**
15. **end for**
16. **end while**
17. **return null**;

Figure 4.2: The LTS algorithm.

Example 4.2.2 We illustrate LTS for PATH(s, t) using the $\mathcal{R}$-$Index^+$ of Table 4.1. Initially, LTS accesses $routes^+[t]$ and constructs the target list $\mathcal{T} = \{\langle r_1 : 4 \rangle, \langle r_5 : 1 \rangle\}$.

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

LTS proceeds expanding a and then c. After the third iteration we have:

$\mathcal{Q} = \{d, f\}$,
$\mathcal{H} = \{s, a, c, d, f\}$, and
$\mathcal{A} = \{\langle s : r_3 : 1, a\rangle, \langle a : r_2 : 3, c\rangle, \langle c : r_2 : 4, d\rangle, \langle c : r_4 : 3, f\rangle\}$.

At the next iteration, f is popped and LTS retrieves list $routes^+[f]$. The entry $\langle r_1 : 2, t\rangle$ matches the corresponding $\langle r_1 : 4\rangle$ in $\mathcal{T}$, since route r_1 contains the current search node f before target t. Thus, LTS terminates the search and uses $\mathcal{A}$ to identify a sequence of links ($\langle s{:}r_3{:}1\rangle, \langle a{:}r_3{:}*\rangle, \langle a{:}r_2{:}3\rangle, \langle c{:}r_2{:}*\rangle, \langle c{:}r_4{:}3\rangle, \langle f{:}r_4{:}*\rangle, \langle f{:}r_1{:}2\rangle, \langle t{:}r_1{:}4\rangle$) that leads to the target. $\langle a{:}r_2{:}3\rangle$ 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, w, 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 $\mathcal{T}$-$Index$, discussed in Section 3.2.

Figure 4.3 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 $\mathcal{T}$ (Lines 3–8), which allow LTST to terminate sooner. The algorithm retrieves list $routes^+[n_t]$ from $\mathcal{R}$-$Index^+$ and accesses $\mathcal{T}$-$Index$ to retrieve lists $trans[r_i]$ for all routes r_i in $routes^+[n_t]$. Just like LTS, it inserts into $\mathcal{T}$ all routes that contain the target (Line 4). Moreover, LTST includes all routes r_j that can lead to n_t via some link n_ℓ that resides on the same route r_i with n_t (Line 6). Intuitively, this implies that $\mathcal{T}$ contains routes (r_i'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_t). Therefore, $\mathcal{T}$ includes all routes that are less than two routes away from the target.

An entry of $\mathcal{T}$ has the form $\langle r_j : o_{\ell j}, r_i : o_{\ell i}\rangle$, which means that route r_j leads to the target n_t in route r_i via some link n_ℓ that lies at the $o_{\ell j}$-th position in r_j and at the $o_{\ell i}$-th position in r_i before n_t. An entry $\langle r_i : o_{ti}, \varnothing\rangle$ implies that route r_i contains the target n_t (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_i that leads, via some link n_ℓ, to a route r_j that contains the target. Specifically, the algorithm checks if there exist entries $\langle r_i : o_{\ell i}, r_j : o_{\ell j}\rangle$ in $\mathcal{T}$ and $\langle r_i : o_{qi}, n_q^+\rangle$

in $routes^+[n_q]$ such that $o_{qi} < o_{\ell i}$ (Line 12). 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_ℓ. However, as the termination check suggests, it suffices to retain only the entry with the latest via-link, i.e., that with the highest $o_{\ell i}$. A final subtle difference with LTS is that ConstructPath requires routes r_i, r_j and the positions of the via-link $o_{\ell i}$, $o_{\ell j}$ in them, so as to compose the answer path.

Algorithm LTST
Input: n_s, n_t, $\mathcal{R}$-$Index^+$, $\mathcal{T}$-$Index$
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 $\mathcal{T}$:	stores routes that contain n_t and their intersecting routes

Method:

1. **push** n_s in $\mathcal{Q}$;
2. **insert** n_s in $\mathcal{H}$;
3. **for** each entry $\langle r_i : o_{ti}, n_t^+ \rangle$ in $routes^+[n_t]$ **do**
4. **insert** $\langle r_i : o_{ti}, \varnothing \rangle$ in $\mathcal{T}$;
5. **for** each entry $\langle r_j, n_\ell : o_{\ell i} : o_{\ell j} \rangle$ in $trans[r_i]$ **do**
6. **if** $o_{\ell i} < o_{ti}$ **then**
insert $\langle r_j : o_{\ell j}, r_i : o_{\ell i} \rangle$ in $\mathcal{T}$;
7. **end for**
8. **end for**
9. **while** $\mathcal{Q}$ is not empty **do**
10. **let** $n_q = \mathsf{pop}(\mathcal{Q})$;
11. **for** each entry $\langle r_i : o_{qi}, n_q^+ \rangle$ in $routes^+[n_q]$ **do**
12. **if** there is an entry $\langle r_i : o_{\ell i}, r_j : o_{\ell j} \rangle$ in $\mathcal{T}$ such that $o_{qi} < o_{\ell i}$ **then**
return ConstructPath$(n_s, n_q, n_t, \mathcal{A}, r_i : o_{\ell i}, r_j : o_{\ell j})$;
13. **if** $n_q^+ \notin \mathcal{H}$ **then**
14. **push** n_q^+ in $\mathcal{Q}$;
15. **insert** n_q^+ in $\mathcal{H}$;
16. **insert** $\langle n_q^+, n_q : r_i : o_{qi} \rangle$ in $\mathcal{A}$;
17. **end if**
18. **end for**
19. **end while**
20. **return null**;

Figure 4.3: The LTST algorithm.

Example 4.3.1 Consider PATH(s, t) and $\mathcal{R}$-$Index^+$ and $\mathcal{T}$-$Index$, shown in Tables 4.1 and 3.2, respectively. First, LTST retrieves $routes^+[t]$ from $\mathcal{R}$-$Index^+$, which contains two routes r_1 and r_5. Then, it retrieves the corresponding lists $trans[r_1]$ and $trans[r_5]$ from $\mathcal{T}$-$Index$, and constructs the target list $\mathcal{T} = \{\langle r_1 : 4, \varnothing \rangle, \langle r_2 : 5, r_1 : 1 \rangle, \langle r_3 : 1, r_5 : 2 \rangle, \langle r_4 : 4, r_1 : 2 \rangle, \langle r_5 : 1, \varnothing \rangle\}$.

The first iteration of LTST is identical to that in Example 4.2.2. At the second iteration, LTST pops link a and retrieves list $routes^+[a]$ from $\mathcal{R}$-$Index^+$. The first entry $\langle r_2 : 3, c \rangle$ in this list matches the entry $\langle r_2 : 5, r_1 : 1 \rangle$

in the target list, since in r_2 node a lies at position 3 before some node n_ℓ at position 5 that leads to the target via route r_1 (at this point LTST does not know that n_ℓ corresponds to d). Therefore, the search terminates and LTST identifies a sequence of links $(\langle s : r_3 : 1\rangle, \langle a : r_3 : *\rangle, \langle a : r_2 : 3\rangle, \langle * : r_2 : 5\rangle, \langle * : r_1 : 1\rangle, \langle t : r_1 : *\rangle)$ that leads to the target; the symbol * indicates that the node or its position can be inferred. After retrieving the required parts of routes r_1, r_2 and r_3, the answer path (s, w, a, c, d, f, y, t) is constructed. □

4.4 The LTS-k Algorithm

The LTST algorithm terminates as soon as the current search node is within two routes from the target. This strong termination condition is possible due to the information stored in $\mathcal{T}$-$Index$. However, the size of $\mathcal{T}$-$Index$ is quadratic with respect to the number of routes, which makes it impractical for large collections.

This section presents the LTS-k algorithm that operates without the $\mathcal{T}$-$Index$, and features a tunable termination condition based on parameter k. Particularly, LTS-k stops when it reaches a route r_i that leads via link n_ℓ to a route r_j containing the target n_t, with the requirement that n_ℓ is at most k links before n_t in r_j. Note that when k is set to 0, the algorithm reduces to LTS. On the other hand, for a sufficiently high k value (larger than the maximum number of links in any route), LTS-k terminates when it visits a node that is less than two routes from n_t, exactly like LTST. In this case, however, LTS-k spends more time compiling the target list compared to LTST, since the latter has access to $\mathcal{T}$-$Index$ that materializes the transition information between any two routes.

LTS-k, shown in Figure 4.4, requires $\mathcal{R}$-$Index^+$ (but not $\mathcal{T}$-$Index$), and the parameter k. Since LTS-k only differs from LTST in the target list construction, we only detail this process that involves two phases (Lines 3–17).

In the first phase (Lines 3–12), LTS-k constructs a list $\mathcal{L}$ with all links that are within k links from the target in some route, including n_t itself (Line 4). To find these nodes, the algorithm retrieves all routes that contain n_t (Line 5) and inserts into $\mathcal{L}$ the k links before n_t (if they exist) in each route (Lines 6–11). An entry of $\mathcal{L}$ has the form $\langle n_\ell, r_i : o_{\ell i}\rangle$, which means that link n_ℓ lies in the same route r_i with target n_t and is within k links away from it. Note that although a link in $\mathcal{L}$ may appear in multiple routes, LTS-k only keeps a single entry per link.

At this point we make two important notes. First, LTS-k must distinguish between links and non-link nodes, when retrieving a route. Therefore, the algorithm needs to keep in main memory either a compressed bitmap of length equal to the number of nodes, or a hash index storing only the links, in the case when the collection has much fewer links than nodes.

Second, $\mathcal{L}$ is *not* the set of all links that are within k links from the target. Rather, $\mathcal{L}$ contains a subset of only those links that are in the *same route* with n_t, primarily for efficiency reasons. In order to reach all links within k links from n_t, the algorithm would need to perform a breadth-first search starting from n_t following the reverse edges of the conceptual reduced routes graph $G^-_{\mathbf{R}}$. Since $G^-_{\mathbf{R}}$ is not materialized, this operation would have to retrieve a much larger set of routes.

Subsequently, in the second phase (Lines 13–17), the algorithm scans list $\mathcal{L}$ and uses the $\mathcal{R}$-$Index^+$ to insert into $\mathcal{T}$ all routes that contain a link of $\mathcal{L}$. Similar to LTST, LTS-k retains a single entry $\langle r_j : o_{\ell j}, r_i : o_{\ell i} \rangle$ per route, that of the highest via position $o_{\ell j}$.

Algorithm LTS-k
Input: n_s, n_t, k, $\mathcal{R}$-$Index^+$
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_t 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 $\langle r_i : o_{ti}, n_t^+ \rangle$ in $routes^+[n_t]$ **do**
4. **insert** $\langle n_t, r_i : o_{ti} \rangle$ in $\mathcal{L}$;
5. **retrieve** route r_i and **let** $o_{\ell i} = o_{ti}$;
6. **for** $m = 1$ up to k **do**
7. **repeat** // make $o_{\ell i}$ point at the previous link in r_i
8. **let** $o_{\ell i} = o_{\ell i} - 1$;
9. **until** n_ℓ is a link;
10. **insert** $\langle n_\ell, r_i : o_{\ell i} \rangle$ in $\mathcal{L}$;
11. **end for**
12. **end for**
13. **for** each entry $\langle n_\ell, r_i : o_{\ell i} \rangle$ in $\mathcal{L}$ **do**
14. **for** each entry $\langle r_j : o_{\ell j}, n_\ell^+ \rangle$ in $routes^+[n_\ell]$ **do**
15. **insert** $\langle r_j : o_{\ell j}, r_i : o_{\ell i} \rangle$ in $\mathcal{T}$;
16. **end for**
17. **end for**

[**proceeds as in Lines 9–20 of the** LTST **algorithm**]

Figure 4.4: The LTS-k algorithm.

Example 4.4.1 We illustrate the LTS-1 algorithm ($k=1$) for the $\texttt{PATH}(s,t)$ query on the collection of Figure 2.1(a) using the $\mathcal{R}$-$Index^+$ presented in Table 4.1.

Initially, the algorithm constructs the list of links $\mathcal{L}$ that are within one link from t. It accesses $routes^+[t] = \{\langle r_1 : 4, s \rangle, \langle r_5 : 1, s \rangle\}$ and retrieves routes r_1 and r_5 at positions 4 and 1, respectively. Moving backwards in r_1, LTS-1 identifies f as the one link before t; route r_5 contains no nodes before

t. Therefore, $\mathcal{L} = \{\langle t, r_1 : 4\rangle, \langle f, r_1 : 2\rangle\}$ contains an entry for the target t and f.

Subsequently, LTS-1 accesses the lists $routes^+[t] = \{\langle r_1 : 4, s\rangle, \langle r_5 : 1, s\rangle\}$ and $routes^+[f] = \{\langle r_1 : 2, t\rangle, \langle r_4 : 4, \varnothing\rangle\}$ 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: $\mathcal{T} = \{\langle r_1 : 4, \varnothing\rangle, \langle r_4 : 4, r_1 : 2\rangle, \langle r_5 : 1, \varnothing\rangle\}$.

The first two iterations of LTS-1 are the same as those of LTS. At the third iteration, LTS-1 pops link c and accesses list $routes^+[c] = \{\langle r_2 : 4, d\rangle, \langle r_4 : 3, f\rangle\}$ from $\mathcal{R}$-$Index^+$. The second entry matches the entry $\langle r_4 : 4, r_1 : 2\rangle$ in $\mathcal{L}$, since c is on r_4 before position 4. Therefore, the algorithm identifies a sequence of links $(\langle s : r_3 : 1\rangle, \langle a : r_3 : *\rangle, \langle a : r_2 : 3\rangle, \langle c : r_2 : *\rangle, \langle c : r_4 : 3\rangle, \langle * : r_4 : 4\rangle, \langle * : r_1 : 2\rangle, \langle t : r_1 : *\rangle)$ 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 LTS-2 would be exactly the same as that in Example 4.3.1, and LTS-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 $B^+_{\mathcal{R}}$ $routes^+[]$ entries.

$\mathcal{R}$-$Index^+$. The analysis is similar to $\mathcal{R}$-$Index$, substituting $B_{\mathcal{R}}$ with $B^+_{\mathcal{R}}$.

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 $\mathcal{T}$ is constructed, while in the second the collection is traversed.

The second phase is identical for all algorithms. At each iteration, after node n_q is popped, they access the $routes^+[n_q]$ at a cost of $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}\right)$ 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 $routes^+[n_t]$ at a cost of $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}\right)$ I/Os. In addition, LTST retrieves from $\mathcal{T}$-$Index$ the $trans[]$ lists for each route in $routes^+[n_t]$ at a cost of $O\left(\frac{|\mathbf{R}|^2\cdot L_r^3}{|N|^2\cdot B_{\mathcal{T}}}\right)$ I/Os. On the other hand, LTS-k retrieves each route referenced in $routes^+[n_t]$ with $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|}\cdot\frac{L_r}{B_{\mathcal{N}}}\right)$ I/Os, and then for each route it reads the $routes^+[]$ list of the k nodes before the target with $O\left(k\cdot\frac{|\mathbf{R}|\cdot L_r}{|N|}\cdot\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}\right)$ I/Os.

Aggregating for $|N|$ nodes in the worst case traversal scenario, we obtain the following complexities. LTS requires $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}+\frac{|\mathbf{R}|\cdot L_r}{B^+_{\mathcal{R}}}\right)$ I/Os, LTST requires $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}+\frac{|\mathbf{R}|^2\cdot L_r^3}{|N|^2\cdot B_{\mathcal{T}}}+\frac{|\mathbf{R}|\cdot L_r}{B^+_{\mathcal{R}}}\right)$ I/Os, and finally, LTS-k re-

quires $O\left(\frac{|\mathbf{R}|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}} + \frac{|\mathbf{R}|\cdot L_r^2}{|N|\cdot B_{\mathcal{N}}} + k\cdot\frac{|\mathbf{R}|^2\cdot L_r^2}{|N|^2\cdot B^+_{\mathcal{R}}} + \frac{|\mathbf{R}|\cdot L_r}{B^+_{\mathcal{R}}}\right)$ I/Os.

Chapter 5

Updating Route Collections

Section 5.1 discusses the case when new routes are added in the collection, while Section 5.2 addresses deletions. Section 5.3 analyzes the complexity of our update mechanism.

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 maintain 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 contains 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 $+\langle r_i{:}o_{ji}\rangle$ at the end of the main memory list $routes^M[n_j]$ (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 $routes^M[n_j]$ is merged with the corresponding disk-based list $routes[n_j]$. Recall that entries in $routes[n_j]$ are sorted ascending on the route identifier. Therefore, since all entries in $routes^M[n_j]$ are about new routes, the merging operation simply requires appending $routes^M[n_j]$ at the end of $routes[n_j]$.

$\mathcal{R}$-$Index^+$. 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 $routes^{+M}[n_m]$ 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 $routes^{+M}[n_m]$ list that contains entries prefixed with $\pm$ must update those in $routes[n_m]$; 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 $+\langle r_k, n_j{:}o_{ji}{:}o_{jk}\rangle$ at the end of the main memory list $trans^M[r_i]$, and (2) insert the entry $+\langle r_i, n_j{:}o_{jk}{:}o_{ji}\rangle$ 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 $routes[n_j]$ 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 $\mathcal{R}$-$Index$, as the list $trans^M[r_j]$ of an old route r_j contains only entries about new routes, $trans^M[r_j]$ is appended at the end of $trans[r_j]$. Finally, the list $trans^M[r_i]$ of a newly added route r_i has no counterpart on the disk, and thus it *becomes* the disk-based list $trans[r_i]$.

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.

$\mathcal{R}$-*Index* and $\mathcal{R}$-$Index^+$. For each deleted route r_i, retrieve the corresponding route from disk. For each node n_j of r_i, retrieve the list $routes[n_j]$ (or $routes^+[n_j]$) and delete the entry corresponding to r_i. During this process, the main memory data structure for distinguishing links from non-link nodes is updated.

$\mathcal{T}$-*Index* . Flushing of $\mathcal{T}$-$Index$ occurs in parallel to flushing $\mathcal{R}$-$Index$/$\mathcal{R}$-$Index^+$. 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 $\mathcal{T}$-$Index$ the list $trans[r_k]$ and remove from it the entry concerning r_i. Additionally, delete the entire list $trans[r_i]$.

5.3 Complexity Analysis

We use the notation introduced in Section 3.3 and 4.5.

Insertions. We assume that $|\mathbf{R}_\mathcal{I}|$ new routes are inserted in the existing route collection $\mathbf{R}$ containing $|N_\mathcal{I}|$ nodes. The worst case scenario is when $|N_\mathcal{I}|$ is a subset of the nodes $|N|$ contained in the existing collection $\mathbf{R}$.

For $\mathcal{R}$-$Index$, buffering incurs no cost. In flushing, we read the last of the pages containing $routes[n_i]$, for each node n_i in $|N_\mathcal{I}|$, at a total cost of $O(|N_\mathcal{I}|)$ I/Os. Then, we append the entries regarding the new routes writing $O\left(\frac{|\mathbf{R}_\mathcal{I}| \cdot L_r}{B_\mathcal{R}}\right)$ pages on disk.

For $\mathcal{R}$-$Index^+$, buffering incurs $O\left(\frac{L_r^2}{B_\mathcal{N}}\right)$ I/Os for each new route. Specifically, we assume that, in the worst case, every node n_j in a new route r_i becomes a link, and thus, we retrieve the $routes^+[n_j]$ list containing only one route, r_k, at a cost of one I/O, and then, retrieve r_k at a cost of $O\left(\frac{L_r}{B_\mathcal{N}}\right)$ I/Os. We distinguish between the $|N_\mathcal{I}^+|$ nodes whose $routes^{+M}[]$ list contain only entries prefixed with $+$, and the $|N_\mathcal{I}^{+,\pm}|$ nodes with both $+$ and $\pm$ pre-

fixed entries. When flushing $\mathcal{R}$-$Index^+$, for the $|N_{\mathcal{I}}^+|$ nodes, we read $|N_{\mathcal{I}}^+|$ and write $O\left(|N_{\mathcal{I}}^+| \cdot \frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|}\right)$ pages similar to flushing $\mathcal{R}$-$Index$, while for the $|N_{\mathcal{I}}^{+,\pm}|$ nodes, we retrieve their entire $routes^+[]$ list from disk at a total cost of $O\left(\frac{|\mathbf{R}| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+} \cdot |N_{\mathcal{I}}^{+,\pm}|\right)$ I/Os and write $O\left(\left(\frac{|\mathbf{R}| \cdot L_r}{|N|} + \frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|}\right) \cdot \frac{|N_{\mathcal{I}}^{+,\pm}|}{B_{\mathcal{R}}^+}\right)$ pages. Thus, flushing $\mathcal{R}$-$Index^+$ requires reading $O\left(|N_{\mathcal{I}}^+| + \frac{|\mathbf{R}| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+} \cdot |N_{\mathcal{I}}^{+,\pm}|\right)$ and writing $O\left(\frac{|N_{\mathcal{I}}^+| \cdot |\mathbf{R}_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|} + \left(\frac{|\mathbf{R}| \cdot L_r}{|N|} + \frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|}\right) \cdot \frac{|N_{\mathcal{I}}^{+,\pm}|}{B_{\mathcal{R}}^+}\right)$ pages.

Regarding $\mathcal{T}$-$Index$, buffering requires $O\left(\frac{|\mathbf{R}| \cdot L_r^2}{|N| \cdot B_{\mathcal{R}}}\right)$ I/Os for each new route since we retrieve the $routes[]$ for each of the L_r nodes in the route. After buffering occurs, assume that for $|\mathbf{R}_{\text{aff}}|$ of the old routes, connections with the new routes are identified. Therefore, for these $|\mathbf{R}_{\text{aff}}|$ routes, flushing requires reading $|\mathbf{R}_{\text{aff}}|$ pages and writing $O\left(|\mathbf{R}_{\text{aff}}| \cdot \frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}| \cdot B_{\mathcal{T}}}\right)$. On the other hand for the new routes, we need to write $O\left(\frac{|\mathbf{R}_{\mathcal{I}}|}{B_{\mathcal{T}}} \cdot \left(\frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}|} + \frac{|\mathbf{R}| \cdot L_r^2}{|N|}\right)\right)$ pages. To sum up, flushing $\mathcal{T}$-$Index$ requires reading $|\mathbf{R}_{\text{aff}}|$ and writing $O\left(|\mathbf{R}_{\text{aff}}| \cdot \frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}| \cdot B_{\mathcal{T}}} + \frac{|\mathbf{R}_{\mathcal{I}}|}{B_{\mathcal{T}}} \cdot \left(\frac{|\mathbf{R}_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}|} + \frac{|\mathbf{R}| \cdot L_r^2}{|N|}\right)\right)$ pages.

Deletions. In the presence of deletions, buffering incurs no I/O cost for any index. On the other hand, flushing for $\mathcal{R}$-$Index$ (resp. $\mathcal{R}$-$Index^+$) requires reading $O\left(\frac{L_r}{B_{\mathcal{N}}} + L_r \cdot \frac{|\mathbf{R}| \cdot L_r}{|N| \cdot B_{\mathcal{R}}}\right)$ (resp. $O\left(\frac{L_r}{B_{\mathcal{N}}} + L_r \cdot \frac{|\mathbf{R}| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+}\right)$) pages for each deleted route r_i, to retrieve r_i and the $routes[]$ (resp. $routes^+[]$) list for each of the L_r contained nodes. Then, it requires writing $O\left(L_r \cdot \frac{|\mathbf{R}| \cdot L_r}{|N| \cdot B_{\mathcal{R}}}\right)$ (resp. $O\left(L_r \cdot \frac{|\mathbf{R}| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+}\right)$) pages for the updated $routes[]$ (resp. $routes^+[]$) lists. Finally, flushing $\mathcal{T}$-$Index$ requires reading and writing $O\left(L_r \cdot \frac{|\mathbf{R}| \cdot L_r}{|N|} \cdot \frac{|\mathbf{R}| \cdot L_r^2}{|N| \cdot B_{\mathcal{T}}}\right)$ pages for each deleted route.

Chapter 6

Experimental Evaluation

This chapter 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 $G^{-}_{\mathbf{R}}$. 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.

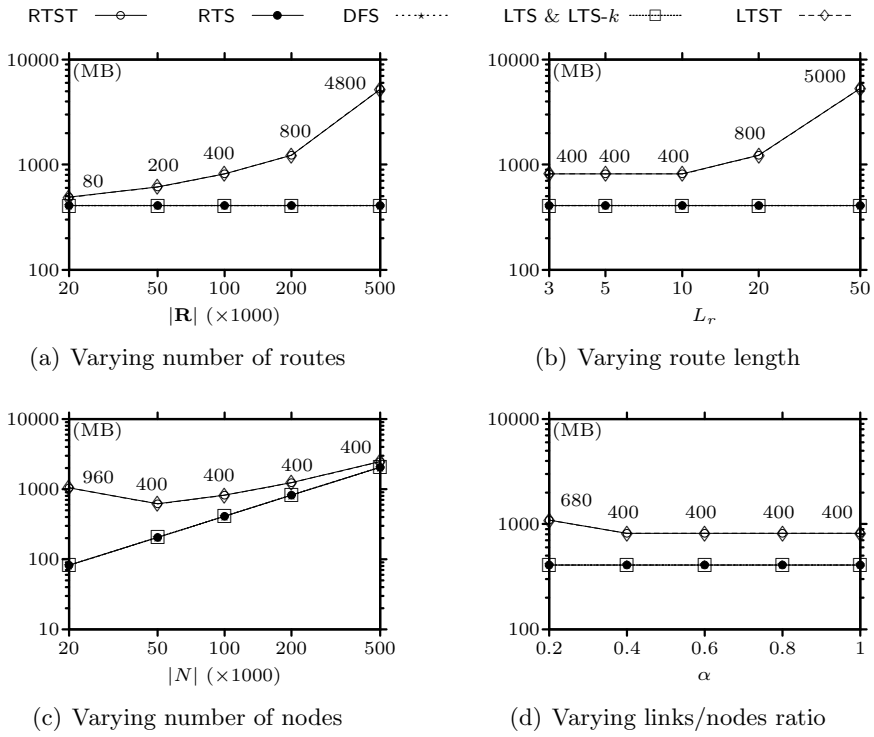
We generate synthetic route collections to test the methods, varying the following parameters (Table 6.1): (1) the number of routes in the collection, $|\mathbf{R}|$, (2) the route length, L_r, (3) the number of distinct nodes in the routes, $|N|$, and (4) the links/nodes ratio α. In each experiment, we vary one of the parameters while we keep the others to their default values.

parameter	**values**	**default value**
$\|\mathbf{R}\|$	20K, 50K, 100K, 200K, 500K	100K
L_r	3, 5, 10, 20, 50	10
$\|N\|$	20K, 50K, 100K, 200K, 500K	100K
α	0.2, 0.4, 0.6, 0.8, 1	0.6

Table 6.1: Experimental parameters

input	method name	index
reduced routes graph $G_{\mathbf{R}}^{-}$	DFS	adjacency lists
route collection	RTS	$\mathcal{R}$-$Index$
	RTST	$\mathcal{R}$-$Index$ & $\mathcal{T}$-$Index$
	LTS	$\mathcal{R}$-$Index^{+}$
	LTST	$\mathcal{R}$-$Index^{+}$ & $\mathcal{T}$-$Index$
	LTS-k	$\mathcal{R}$-$Index^{+}$

Table 6.2: Methods for evaluating `PATH` queries



(a) Varying number of routes

(b) Varying route length

(c) Varying number of nodes

(d) Varying links/nodes ratio

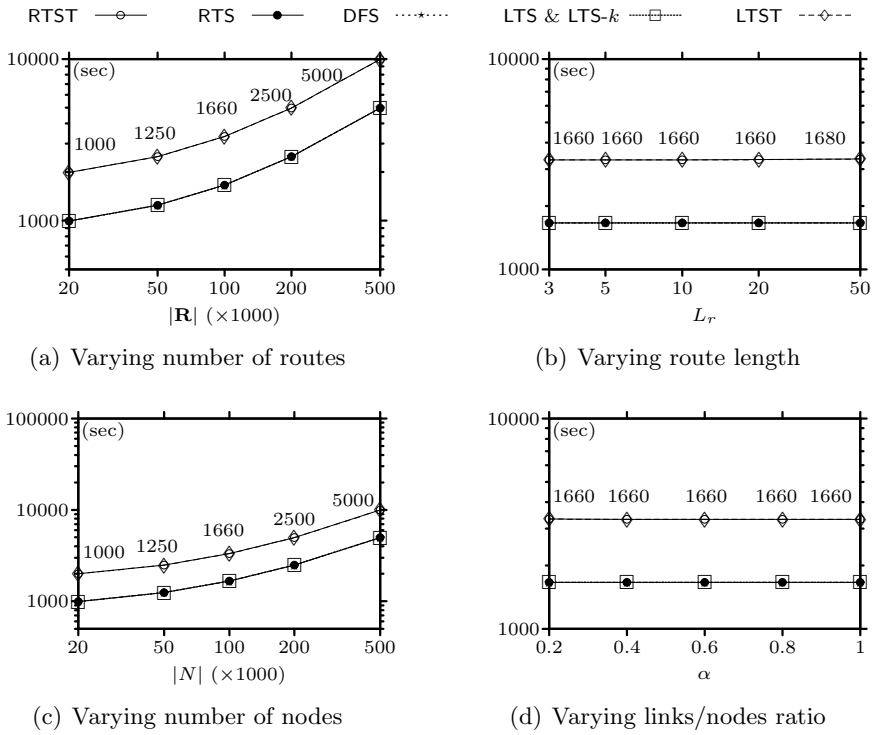
Figure 6.1: Indices space consumption.

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 6.2 shows the indices employed by each method.

Varying the number of routes $|\mathbf{R}|$. The disk space requirement of the $\mathcal{R}$-$Index$/$\mathcal{R}$-$Index^{+}$, employed by RTS, LTS, LTS-k, and DFS, [1] depends

[1] In fact, DFS operates on the adjacency lists of the reduced routes graph, where a list



(a) Varying number of routes

(b) Varying route length

(c) Varying number of nodes

(d) Varying links/nodes ratio

Figure 6.2: Indices construction time.

primarily on the number of nodes $|N|$ (which determines the number of lists $routes[]/routes^+[]$) and not on $|\mathbf{R}|$ (which affects the length of the lists). Hence, Figure 6.1(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 $\mathcal{R}$-$Index^+$ compared to an $\mathcal{R}$-$Index$ entry contains additional information.

On the other hand, as $|\mathbf{R}|$ increases, the number of edges in the transition graph, and thus the size of the $\mathcal{T}$-$Index$ 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 $\mathcal{T}$-$Index$ size. The construction time for all indices, shown in Figure 6.2(a), increases as the collection becomes larger. The values above the RTST line measure the time required for building $\mathcal{T}$-$Index$ only. Note that for $|\mathbf{R}| = 500$K the construction time of $\mathcal{T}$-$Index$ increases modestly compared to the six-fold increase of the index size. This occurs because the majority of the $\mathcal{T}$-$Index$ pages are written sequentially on the disk. [2]

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 $\mathcal{R}$-$Index^+$ list.

[2]On our system, a sequential access is around 350 times faster that a random one.

Varying the route length L_r. The space consumption of the $\mathcal{R}$-$Index$/$\mathcal{R}$-$Index^+$ does not change with L_r, as shown in Figure 6.1(b). This is because, as explained in the context of Figure 6.1(a), the number of $routes[]/routes^+[]$ 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 $\mathcal{T}$-$Index$, becomes denser.

The construction times for RTST and LTST in Figure 6.2(b) increase modestly with L_r, although $\mathcal{T}$-$Index$ becomes much larger. This occurs because the number of random accesses (that depends on the number of $trans[]$ lists) remains constant, as the increase in $\mathcal{T}$-$Index$'s size is due to its lists occupying more pages. Recall, that the contents of a list are written sequentially on disk.

Varying the number of nodes $|N|$. The number of $routes[]/routes^+[]$ lists depends on $|N|$ and thus the total size of the $\mathcal{R}$-$Index$/$\mathcal{R}$-$Index^+$ scales linearly as shown in Figure 6.1(c). Increasing the number of nodes, while $|\mathbf{R}|$ 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 $\mathcal{T}$-$Index$ decrease from 960MB to its minimum 400MB (1 page for each of the 100K routes). Figure 6.2(c) shows that the construction time for all methods increases with $|N|$ due to the increase of the $\mathcal{R}$-$Index$/$\mathcal{R}$-$Index^+$ size.

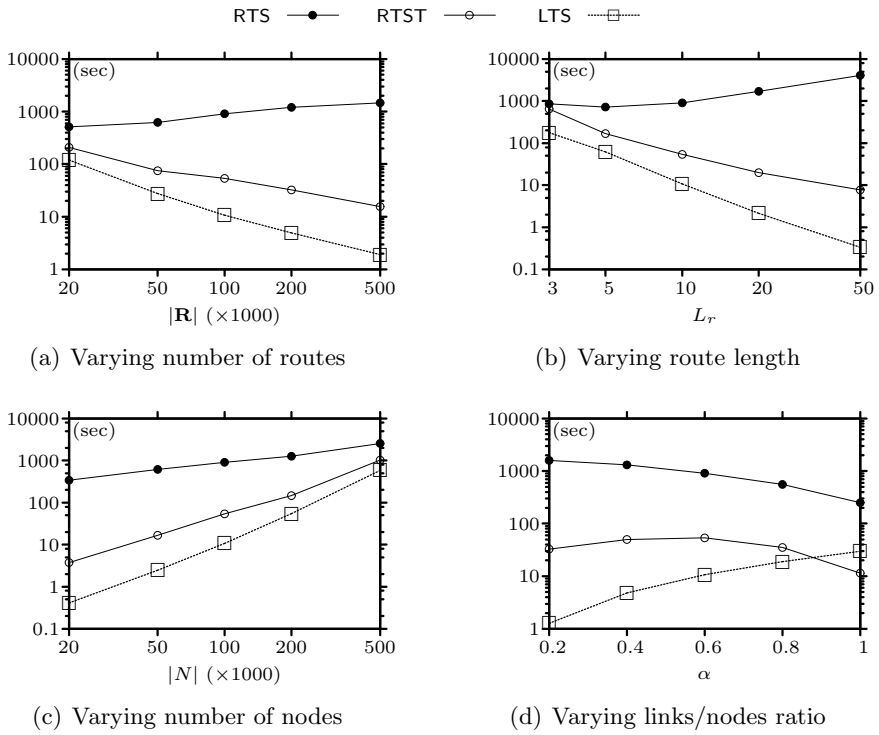
Varying the links/nodes ratio α. The space required for $\mathcal{R}$-$Index$/$\mathcal{R}$-$Index^+$ depends on the number of nodes $|N|$ and not on the links/nodes ratio, hence the constant lines in Figure 6.1(d). As the number of links increases ($|\mathbf{R}|$ and L_r remain fixed), the transition graph becomes sparser, as explained in the context of Figure 6.1(c); this accounts for the decrease in the space $\mathcal{T}$-$Index$ occupies from 700MB to 400MB. Figure 6.2(d) shows that the construction times for all indices are unaffected by α.

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.

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 $|\mathbf{R}|$, L_r, $|N|$ and α in Figures 6.3(a), (b), (c) and (d), respectively.



(a) Varying number of routes

(b) Varying route length

(c) Varying number of nodes

(d) Varying links/nodes ratio

Figure 6.3: Link vs route traversal search: execution time.

Varying the number of routes $|\mathbf{R}|$. As the number of routes increases, finding a path between two nodes becomes easier. This is exhibited by RTST and LTS in Figure 6.3(a). In contrast, the execution time of RTS increases with $|\mathbf{R}|$ as it performs more iterations compared to RTST, which has a stronger termination condition, and to LTS, which only visits links.

Varying the route length L_r. The same observations hold when the route length increases in Figure 6.3(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.

Varying the number of nodes $|N|$. When $|N|$ increases, finding a path becomes harder, as shown in Figure 6.3(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.

Varying the links/nodes ratio α. When the links/nodes ratio increases, there are more links in the collection, but each of them appears in fewer routes. In particular, when α=0.2 the link frequency, i.e., the (average) number of routes in which it appears, is 46, and becomes 10 when α=1. In general, as the ratio increases it becomes harder to find a path. Hence, as α increases LTS needs more iterations to reach the target, and its execution time increases. It is important to notice the behavior of RTS 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 α. On the other hand, since the number of iterations increases, the total execution time increases slightly for small α values, but ultimately decreases.

When $\alpha = 1$, all nodes are links (the reduced routes graph $G^{-}_{\mathbf{R}}$ reduces to the $G_{\mathbf{R}}$ graph) and thus LTS visits exactly the same nodes with RTS. 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 even in this extreme setting.

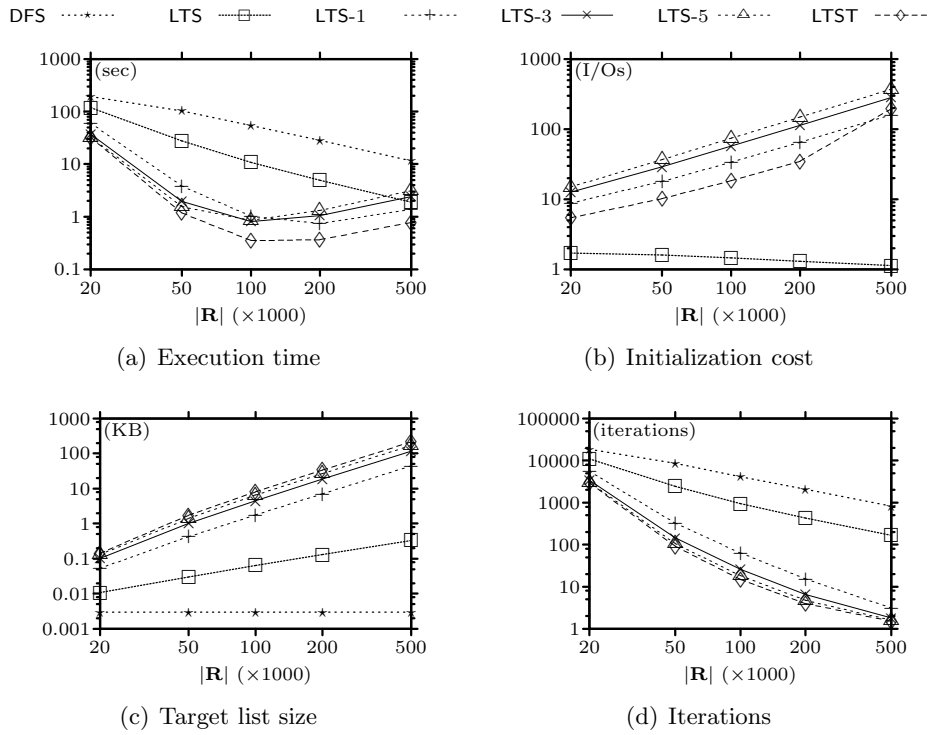
6.3.2 Link traversal search vs DFS

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 $\texttt{PATH}(n_s, n_t)$ query: *initialization* and *core*. In the first phase, all methods need to retrieve the first link n_t^- before n_t (resp. n_s^+ after n_s) 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 $\mathcal{T}$ 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 $|\mathbf{R}|$. As $|\mathbf{R}|$ increases, every link is contained in more routes and the number of iterations decreases, as shown in Figure 6.4(d). In the initialization phase (after retrieving the links following the source and preceding the target, if necessary), LTS retrieves only single $routes^+[]$ list to determine the routes that n_t belongs to and assembles the target list; this has a constant cost as shown in Figure 6.4(b). On the other hand, LTS-k and LTST retrieve multiple $routes^+[]$ and $trans[]$ lists, respectively, depending on the number of routes a link appears in. Since this factor increases with $|\mathbf{R}|$, the initialization cost also increases.



(a) Execution time (b) Initialization cost

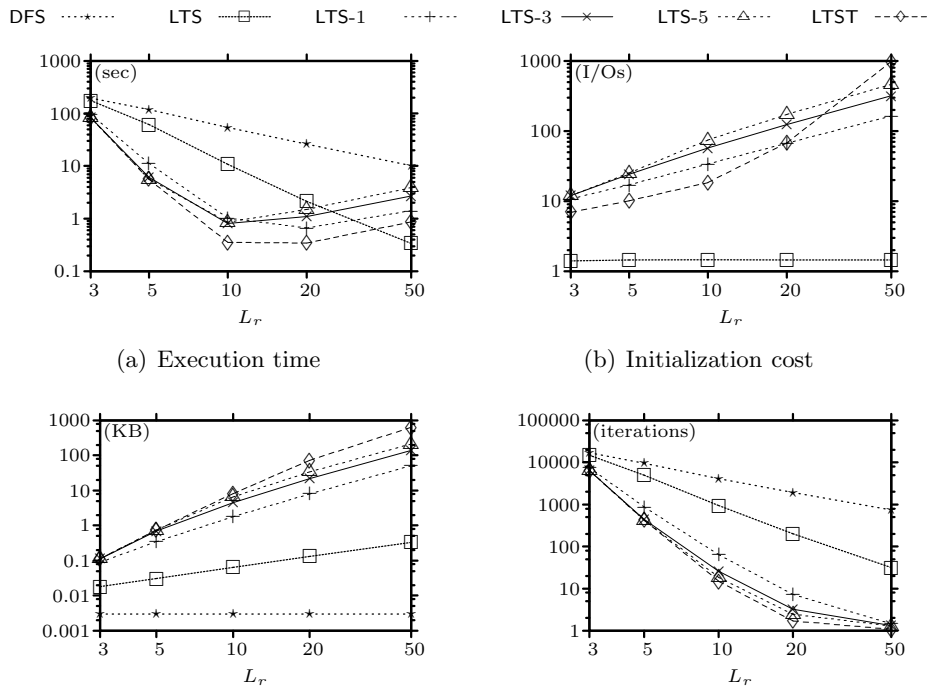
(c) Target list size (d) Iterations

Figure 6.4: Link traversal search vs DFS: varying number of routes.

Similar observations apply for the size of the target list $\mathcal{T}$, which increases for all methods as the $G^{-}_{\mathbf{R}}$ graph becomes denser; see Figure 6.4(c). LTST has the largest while LTS has the smallest $\mathcal{T}$. 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 LTS). However, all LTS-k methods have higher initialization costs compared to LTST, because the latter has access to the $\mathcal{T}$-$Index$. Note that the size of the target list $\mathcal{T}$ portrays the strength of the termination condition; compare the trends in Figures 6.4(c) and 6.4(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 $|\mathbf{R}|$, with LTS becoming up to one order of magnitude faster, as shown in Figure 6.4(a). Similarly, the processing time of LTST and LTS-k decreases rapidly up to 100K routes, and LTST becomes more that two orders of magnitude faster than DFS. On the other hand, when the collection contains more that 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.



(a) Execution time

(b) Initialization cost

(c) Target list size

(d) Iterations

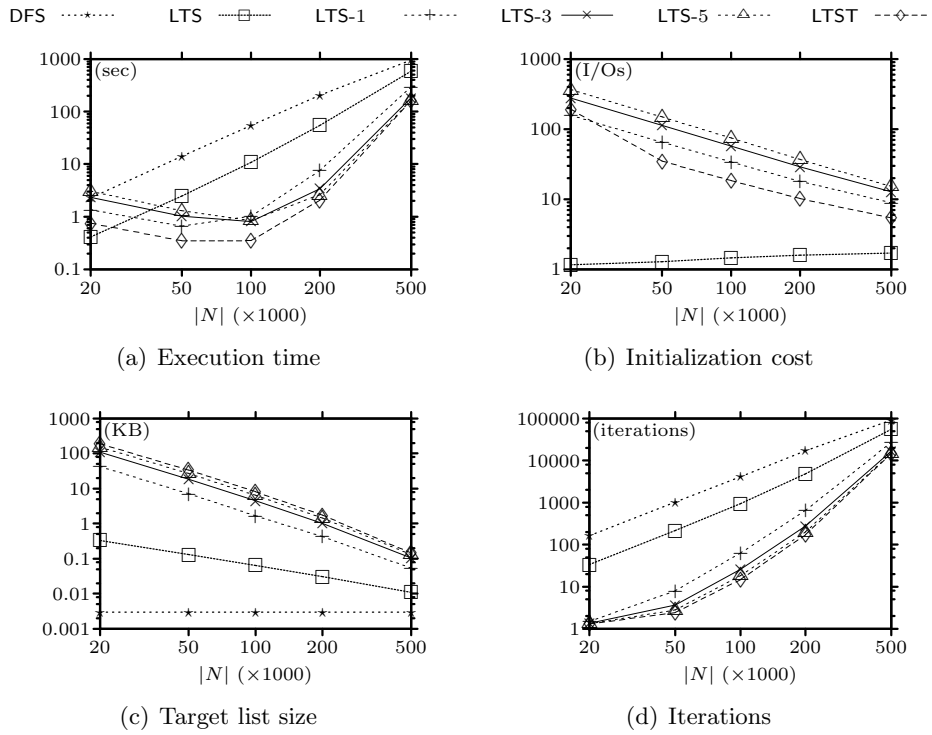
Figure 6.5: Link traversal search vs DFS: varying route length.

Varying the route length L_r. Figure 6.5 illustrates the impact of varying L_r on evaluating PATH queries. As L_r increases, every link is contained in more routes and the reduced $G^-_{\mathbf{R}}$ graph becomes more dense. Therefore, all algorithms exhibit the same trends as in Figure 6.4(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 $|N|$. Figure 6.6 studies the effect of increasing the number of nodes in the collection, while the number of routes in the collection and route length remain fixed. As $|N|$ increases, even though the number of links increases, each of them is contained in fewer routes. Therefore, the reduced routes graph $G^-_{\mathbf{R}}$ becomes sparser, which means that finding a path becomes harder. This is verified in Figure 6.6(c), which depicts that the target list decreases with $|N|$, and in Figure 6.6(d), which shows that more nodes are visited as $|N|$ increases.

Subsequently, the initialization cost of LTST and LTS-k decreases with $|N|$. As explained in the context of Figure 6.4, since LTS accesses a single $routes^+[]$ list, its initialization cost is independent of the number of nodes.

The total execution time of DFS and LTS increases with $|N|$, as they per-



(a) Execution time

(b) Initialization cost

(c) Target list size

(d) Iterations

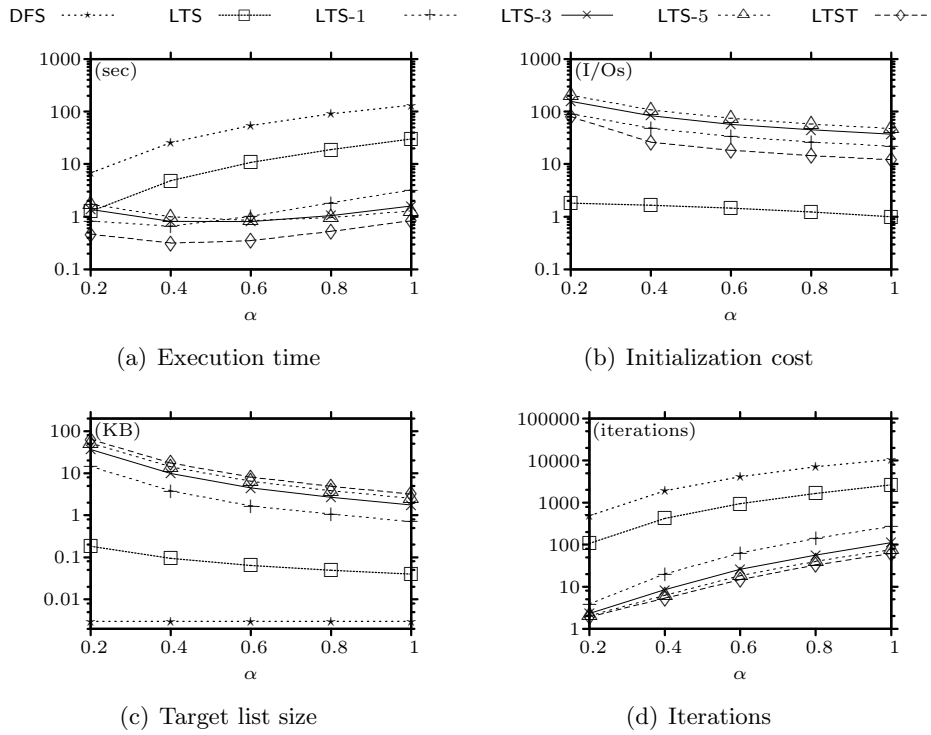
Figure 6.6: Link traversal search vs DFS: varying number of nodes.

form 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 6.6(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 α. As α increases the link frequency decreases and finding a path becomes more difficult, as shown in Figure 6.7(d). For the reasons discussed in the case of varying $|N|$, the target list of all methods decreases with $|N|$ (see Figure 6.7(c)), and the initialization cost of LTST and LTS-k decreases (see Figure 6.7(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 reduced routes graphs corresponding to the route collections of Sections 6.3.1 and 6.3.2 are strongly connected, and thus a path always



(a) Execution time

(b) Initialization cost

(c) Target list size

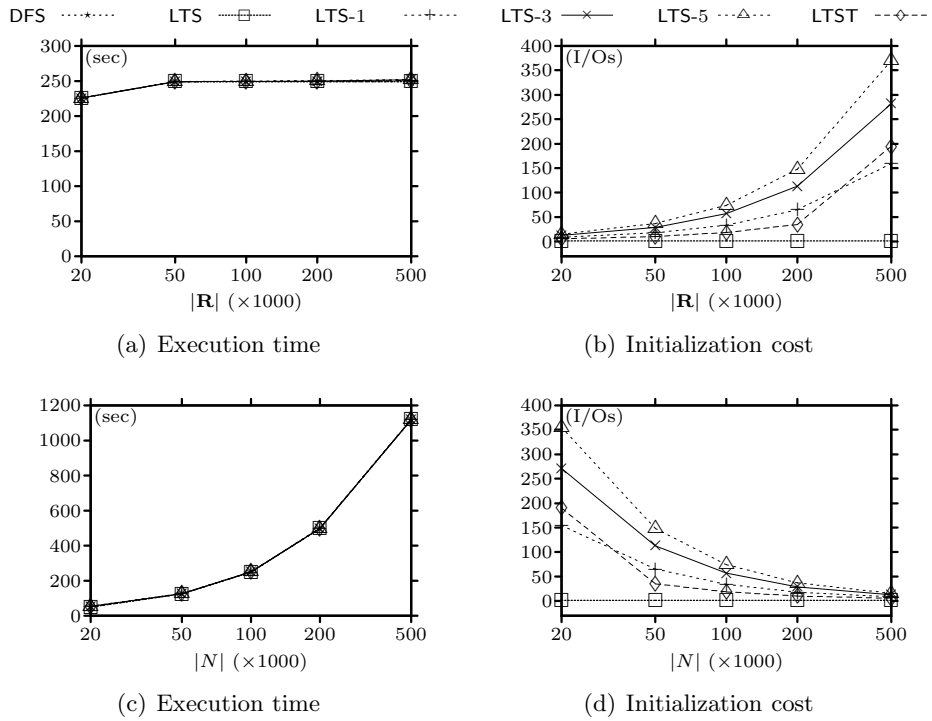
(d) Iterations

Figure 6.7: Link traversal search vs DFS: varying links/nodes ratio.

exists. 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 6.8(a) and 6.8(b) display the total execution time and the cost of the initialization phase while the number of routes varies. In accordance to Figure 6.4(b), the initialization cost of LTST and LTS-k increases with $|\mathbf{R}|$, 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 $|\mathbf{R}|$. 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 6.8(b). The total execution time of LTST and LTS-k, slightly increases with $|\mathbf{R}|$ due to the higher initialization cost. In the worst case, the overhead is less than 1.3%.

Figures 6.8(c) and 6.8(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



(a) Execution time (b) Initialization cost

(c) Execution time (d) Initialization cost

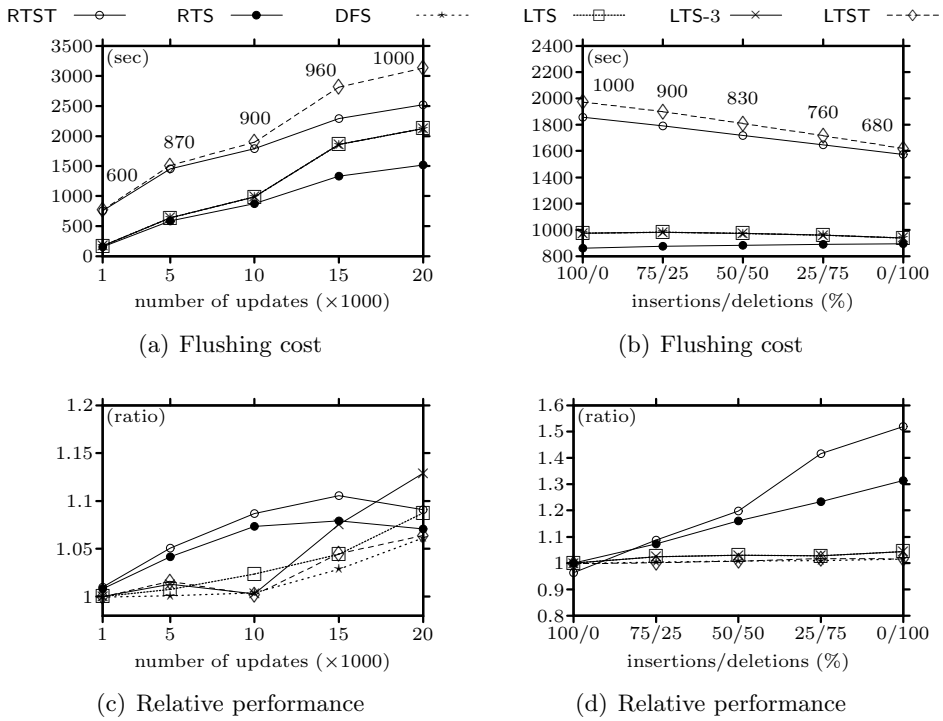
Figure 6.8: PATH queries with no answer: execution time and initialization cost.

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 $\alpha = 0.6$ 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, LTS, and LTS-k must retrieve for each node in the new route that becomes a link, its other route; with $L_r = 10$ and $\alpha = 0.6$, this costs 4 random I/Os (and maybe a few sequential I/Os). RTST and LTST (in addition to the operations needed by LTS) must retrieve from disk the $routes[]/routes^+[]$ list for each link in the route; with $L_r = 10$ and all nodes becoming links



(a) Flushing cost

(b) Flushing cost

(c) Relative performance

(d) Relative performance

Figure 6.9: Updating route collections.

this costs 10 random I/Os (and maybe a few sequential I/Os).

Figure 6.9(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 $\mathcal{T}$-*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 RTS 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 $\mathcal{R}$-*Index*/$\mathcal{R}$-*Index*$^+$ based methods is higher than building the indices from scratch. On the other hand, even when 20,000 updates occur (40% of $|\mathbf{R}|$), the flushing cost for the $\mathcal{T}$-*Index* based methods is lower than the cost of rebuilding indices.

Figure 6.9(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 $\mathcal{T}$-*Index* becomes smaller. As before, the values above the RTST (resp. LTST) line measure the time required for updating $\mathcal{T}$-*Index* only.

Finally, we study how our lazy updating strategy affects the performance

of all methods. To quantify this, 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 6.9(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 6.9(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 6.9(d) at the 100%/0% insertions/deletions ratio.

Chapter 7

Conclusions

We consider the problem of evaluating path queries on large disk-resident routes collections 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.

Bibliography

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD Conference*, pages 253–262, 1989.

[2] R. Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *VLDB*, pages 255–266, 1987.

[3] P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. K. Sellis. Evaluating reachability queries over path collections. In *SSDBM*, pages 398–416, 2009.

[4] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, pages 845–854, 2008.

[5] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[6] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[10] Y. E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *VLDB*, pages 382–394, 1988.

[11] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[12] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*, pages 595–608, 2008.

[13] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[14] H. Lu. New strategies for computing the transitive closure of a database relation. In *VLDB*, pages 267–274, 1987.

[15] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.

[16] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[17] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, pages 845–856, 2007.

[18] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[19] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.