National Technical University of Athens

School of Electrical and Computer Engineering

Division of Computer Science

Evaluating Queries Over Route Collections

PhD Thesis

of

Panagiotis Bouros

Diploma in Electrical and Computer Engineering, N.T.U.A. (2003)

Athens, July 2011

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Evaluating Queries over Route Collections

PhD Thesis

of

Panagiotis Bouros

Diploma in Electrical and Computer Engineering, N.T.U.A. (2003)

Supervising Committee: Y. Vassiliou
T. Sellis
P. Tsanakas

Approved by the Examination Committee 15th July 2011.

...	...	...
Y. Vassiliou	T. Sellis	P. Tsanakas
Prof. NTUA	Prof. NTUA	Prof. NTUA
...	...	...
K. Kontogiannis	F. Afrati	S. Skiadopoulos
Associate Prof. NTUA	Prof. NTUA	Assistant Prof. UoP
	...	
	T. Dalamagas	
	Researcher B	
	IMIS/RC Athena	

Athens, July 2011

...

Panagiotis Bouros
Electrical and Computer Engineer, PhD, N.T.U.A.

PREFACE

This thesis is submitted in fulfilment of the requirements for the degree of Doctor of Philosophy, in the School of Electrical and Computer Engineering, National Technical University of Athens (NTUA), Greece. The presented work describes a framework to evaluate path queries on route collections and has been carried out the last five years in the Knowledge and Database Systems Laboratory (KDBSL) of NTUA.

There are several people who helped and supported me during the work on this thesis. First, I am grateful to Prof. Timos Sellis for his guidance and valuable advice, who took a keen interest not only in this thesis but also in all my work during the last six years of being involved in the activities of the KDBS Lab, and Prof. Yannis Vasileiou for his support and fruitful comments.

In addition, I need to thank Dr. Dimitris Sacharidis for the very fruitful collaboration we had, which helped me to significantly improve the quality of this work. I also want to thank Dr. Theodore Dalamagas for his patience and support throughout all the stressful moments during this work, for his valuable comments and advice on various issues.

Moreover, I would like to thank Prof. Panayiotis Tsanakas, member of my supervising committe, as well as Prof. Foto Afrati (National Technical University of Athens), Assist. Prof. Kostas Kontogiannis (National Technical University of Athens), Assist. Prof. Spiros Skiadopoulos (University of Peloponnese) and Researcher Theodore Dalamagas (Institute for the Management of Information Systems, "Athena" RC) that were pleased to become members of the examination committe.

I also acknowledge the support of the KDBS Lab's staff for their help and advice.

Last but not least, my warmest thanks to all my colleges in the KDBS Lab for their help and cooperation.

Panagiotis Bouros
Athens, July 2011

ABSTRACT

The recent advances in the infrastructure of Geographic Information Systems (GIS), and the proliferation of the GPS technology, have resulted in the abundance of geodata in the form of sequences of spatial locations representing points of interest (POIs), landmarks, waypoints etc. We refer to a set of such sequences as *route collection*. In many applications, the route collections are frequently updated as new routes are continuously created and included, or existing ones are extended or even deleted. This thesis studies three problems where given a frequently updated route collection the goal is to find a *path*, i.e., a sequence of spatial locations, that satisfies a number of constraints. According to the first problem a large collection of touristic routes is available and the goal is find a path that connects two landmarks through locations contained in the routes. Second, we focus on the pickup and delivery problems that appear in various logistics and transportation scenarios. A company that offers pickup and delivery services has already scheduled its fleet of vehicles to follow a collection of routes for servicing a number of customer requests. However during the day, new ad-hoc requests arrive at arbitrary times, and the objective is to find sequences of locations from the vehicle routes, i.e., paths, for picking up and delivering the new objects, and minimizing at the same time the company's expenses. Finally, we consider the problem of providing driving directions from one location of a city to another. In this context a collection of vehicle routes is constructed using everyday driving data on the road network of the city. This collection provides a trusted and "familiar" way of driving through the city, in other words it defines a "known" part of the city's network. The drivers consult the collection whenever they want to travel from one location to another, seeking for a path such that they will drive as less time as possible outside the known part of the road network without significantly increasing, at the same time, the total duration of their journey compared to the fastest way, i.e., the shortest path.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, 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 the proliferation of the GPS technology, have resulted in the abundance of geodata in the form of sequences of spatial locations representing points of interest (POIs), waypoints etc. We refer to a set of such sequences as *route collection*. The characteristics of a route collection vary with respect to the application requirements and its context. In some cases, routes are the only type of data available while in other cases, a graph, e.g., a road network, is primarily available and the routes are created by means of traversing this graph. Further, in some scenarios the routes are directly created and provided by the users whereas in other cases, they are generated after preprocessing the available data. Finally, in many applications, the route collections are frequently updated as new routes are continuously created and included, or existing ones are extended or even deleted. In the following we discuss three examples of route collections.

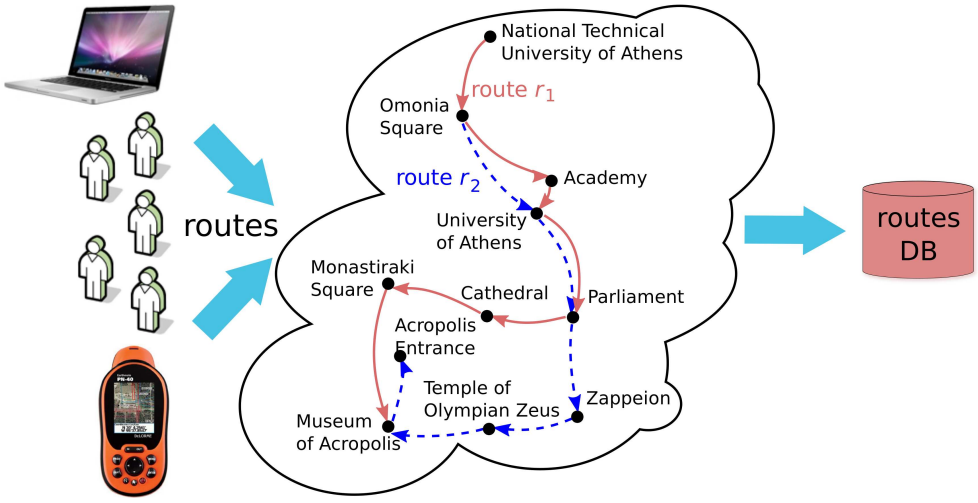


Figure 1.1: *Two touristic routes in the city of Athens.*

As a first example, consider people who visit Athens and use GPS-enabled devices to track their sightseeing. At the end of each day or after they return back home, they create routes through interesting places they visited, either manually, or employing works like [74]. Figure 1.1 shows two touristic routes in Athens. The

first, r_1, starts from the `National Technical University of Athens` and ends at the `Museum of Acropolis`. The second, r_2, starts from the `Omonia Square` and ends at the `Acropolis Entrance`. Web sites such as www.ShareMyRoutes.com and www.TravelByGPS.com maintain a huge collection of routes, like the above, with POIs from all over the world. These collections are frequently updated as users continuously share new interesting routes.

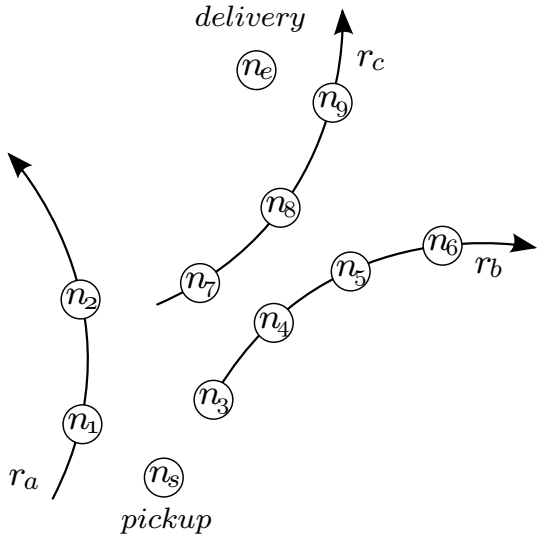


Figure 1.2: *A pickup and delivery scenario.*

Another example of route collections appear in logistics and transportation scenarios, and particularly, in the context of pickup and delivery services offered by a courier company. A transportation request is defined as picking up an object (e.g., package, person, etc.) from one location and delivering it to another; hence the name. Given a set of customer requests, the company constructs a collection of routes that will be followed by its vehicle fleet to pickup and deliver the objects. However, during the day new ad-hoc requests arrive at arbitrary times and thus, the company needs to update the existing vehicle routes in order to satisfy them. Figure 1.2 illustrates such a pickup and delivery scenario. The vehicles routes $r_a(n_1, n_2)$, $r_b(n_3, n_4, n_5, n_6)$ and $r_3(n_7, n_8, n_9)$ are already defined by the company when a new customer request for picking up an object from location n_s and delivering at n_e, arrives.

As a final example of route collections, consider a group of people that track their every day movement with their cars. Given the road network of a city, the movement of a vehicle is captured by a sequence of road intersections, in other words by a route. Then, these people share their driving data and thus, a collection of vehicle routes is defined. This route collection can be viewed as a trusted way of driving through the city, which is frequently updated as people make their way to previously unknown parts of the city or identify better ways for reaching already known locations. The drivers consult this shared route collection whenever they want to travel from one location of the city to another.

1.1 Motivation

Given the availability of route collections such as the aforementioned, new challenges in the field of answering path queries arise. First, it is interesting to examine whether path queries traditionally related to graphs can be posed on route collections, and even more importantly, if the evaluation of these queries can be enhanced by the special characteristics of the routes. For instance, we could think a route as a set of precomputed answers. Consider the collection of Figure 1.1. A fundamental graph query we could pose on the route collection is the reachability query: "Is there a sequence of interesting locations connecting `Academy` to `Zappeion`?". The answer to the reachability query is of course simply "yes" or "no" but this not the case with the path query "Find a sequence of interesting locations from `Academy` to `Zappeion`". An answer to this query could be the sequence of locations or simply the path: `Academy`, `University of Athens` (changing from r_1 to r_2), `Parliament`, `Zappeion`. Note that a path query such as the latter is more general than the reachability query since its answer for two locations n_s and n_t also determines that n_t is reachable from n_s, while the converse does not hold.

A route collection can be trivially transformed to a graph; hence, path and reachability 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.

Another arising challenge is to formulate existing non-graph problems that involve a route collection as path queries on the routes. To this end, we focus on *pickup and delivery problem* and especially on its dynamic version. Consider the pickup and delivery scenario in Figure 1.2. The problem of picking up a new object at location n_s and delivering it at n_e can be formulated as a query seeking for a sequence of locations from the vehicle routes, i.e., a path, that minimizes the company's expenses. To identify this path, we may have to transfer the object of the request among the company's vehicles and in many cases extend the existing routes to pass through the pickup and the delivery locations. For instance, as a possible solution to the problem in Figure 1.2, the vehicle following r_a makes a detour to pickup the object at location n_s. Then, the two vehicles following routes r_a and r_c make a detour at locations n_2 and n_7, respectively, to reach a rendezvous location where the object is transfered to the second vehicle. Finally, the vehicle on r_c delivers the object to its destination, n_e, performing another detour at location n_9. For most of the works that target dynamic pickup and delivery problems, like the above, the rule of thumb is to apply a two-phase local search algorithm to heuristically determine a good request-to-vehicles assignment.

As a final challenge, even new graph queries inspired by the existence of a route collection can be introduced. For example, consider the collection of vehicle routes constructed using every day driving data in the road network of a city. The problem of providing driving directions from one location of the city to another is a well-

studied problem and usually it is solved as a shortest path problem. However, given a share collection of vehicle routes and therefore, a trusted way of driving through the city, we introduce a new path query that captures the actual way people drive through a city. Particularly, people tend to follow roads they use in their every day life or roads that have followed in the past. In addition, even when they want to drive to a location for the first time, they usually ask their friends to recommend a "good" and safe way. In other words, very often, a driver prefers a trusted and familiar way for moving through a city over the fastest way. The answer to a query, like the above, is a sequence of locations such that a driver will drive as mush time as possible on road segments contained in the routes of the collection without significantly increasing, at the same time, the total duration of his journey compared to the fastest way, i.e., the shortest path.

This dissertation presents a framework for the evaluation of path queries over route collections that are frequently updated. The framework involves a set of algorithms for query evaluation and a set of indexing schemes on the routes. In addition, appropriate updating procedures for these schemes are introduced.

1.2 Contributions

Based on the above observations, the contributions of this thesis include the following:

1. We target path query evaluation on large disk resident route collections like the ones containing touristic routes, that are frequently updated. The updates involve additions and deletions of routes. Given two locations n_s and n_t the path query, denoted by `PATH`, returns a sequence of locations contained solely on the existing routes of the collection. We introduce two *evaluation paradigms* that enjoy the benefits of search algorithms (i.e., fast index maintenance) while utilizing transitivity information to terminate the search sooner. In addition, efficient *indexing schemes* and appropriate *updating procedures* are also introduced. The proposed framework, i.e., the indices and the traversal policies, constitutes the basis for applying our work to other types of queries under various constraints. An extensive experimental evaluation verifies the advantages of our methods compared to conventional graph-based search. The methodology discussed and the results obtained appear in [8, 12, 13, 14].

2. We formulate dynamic Pickup and Delivery with Transfers (`dPDPT`) as a path problem. For this purpose, we introduce a conceptual graph, called *dynamic plan graph* that captures all possible actions for picking an object and delivering it to the destination with respect to the existing vehicle routes. Then, we define two cost metrics, termed *operational* and *customer cost*, that capture both the company's and the customer's viewpoints of the problem, respectively, and propose a methodology that identifies the solution computing the shortest path on the dynamic plan graph with respect to these costs. An extensive experimental analysis demonstrates that our method is significantly faster than a two-phase local search method inspired by the related work, while the quality of the solution is only marginally lower. The methodology discussed and the results obtained appear in [9, 11].

3. We introduce the *Most Trusted Near Shortest Path* (`MTNSP`) as a preferable way of driving through a city when a collection of trusted vehicle routes is available. For this purpose, we define the notion of the *known graph* as a subgraph of the road network that is constructed merging the available trusted routes. We also define two costs for a path between two locations on the road network, measuring the total *traveling time* needed and the total *time spent outside* the known graph. Finally, we propose a methodology for identifying the path that has the lowest total time outside the known graph among the paths with length, at most α times larger than the length of the shortest path, as the answer to a `MTNSP` query. An extensive experimental analysis shows the advantage of our methodology compared to a label-setting algorithm that exploits the euclidean distance of the network intersections to prune its search space. The methodology discussed and the results obtained appear in [10].

1.3 Outline

The remainder of this thesis is structured as follows.

Chapter 2 presents our framework for evaluating path queries on large disk resident route collections that are frequently updated, and an extensive experimental evaluation that compares our methods against conventional graph-based search.

Chapter 3 presents our methodology for solving the dynamic Pickup and Delivery with Transfers as a shortest path problem, and an extensive experimental analysis that compares our method against a two-phase local search method inspired by the related work.

Chapter 4 presents our methodology for identifying the Most Trusted Near Shortest Path, and an extensive experimental analysis that compares our methodology against a label-setting algorithm which exploits the euclidean distance of the network intersections to prune its search space.

Finally, Chapter 5 concludes the discussion of this thesis summarizing its contributions, and presents possible extensions and ideas for future work.

Chapter 2

Evaluating Path Queries

Given the availability of large route collections, the problem of identifying *paths* over a route collection arises frequently. Assuming a route collection and two 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. The path query is closely related to the *reachability query*, which is widely studied in the literature. However, a path query identifies a path from n_s to n_t, while a reachability query answers only if such a path exists. Thus, an answer to a path query for nodes n_s and n_t provides an answer also to their reachability query, while the converse does not hold.

This chapter 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 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, the contributions of our work in this chapter can be summarized as follows:

(1) 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, the *route traversal search*, expands the search considering all successor nodes in a route, while the *link traversal search* considers only the next link, i.e., the next node shared with another route. Both paradigms terminate when they reach a route that leads to the target, and are faster than conventional search.

(2) We introduce two basic indexing schemes on a route collection. $\mathcal{R}$-$Index$ associates every node with the routes that contain it, while $\mathcal{T}$-$Index$ captures all possible transitions among routes.

(3) We present two algorithms that follow the route traversal paradigm. RTS employs the $\mathcal{R}$-$Index$ of the route collection while RTST uses $\mathcal{T}$-$Index$ to

achieve earlier termination of the search.

(4) We present three algorithms that follow the link traversal 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.

(5) We discuss efficient maintenance techniques as routes are added and deleted from the collection.

(6) We carry out a thorough experimental study demonstrating that the link traversal search methods always outperform the route traversal ones and a conventional graph traversal algorithm. Finally, among the link traversal search methods, LTS-k is shown to offer the best trade-off between efficiency and maintenance cost.

The remainder of this chapter is structured as follows. Section 2.1 reviews relevant bibliography in detail. Section 2.2 formally defines the problem of evaluating path queries over route collections. Section 2.3 discusses route traversal search, and Section 2.4 introduces the link traversal search paradigm. Then, Section 2.5 discusses maintenance of the index structures under frequent updates of the route collection. Finally, Section 2.6 demonstrates our experimental results and Section 2.7 concludes the chapter.

2.1 Related Work

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 reachability 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 reachability query, (3) support for path query, and (4) capacity to handle updates.

Searching. The simplest way to evaluate path queries is to traverse the graph at query time exploiting a search algorithm, e.g., depth-first or breadth-first search [27]. 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 $G(N, E)$ is the graph $G^*(N, E^*)$, where an edge (n_i, n_j) is in E^* if a path from n_i to n_j exists in G. Using TC a reachability query can be answered in constant time. However, even though efficient algorithms for computing the TC have been proposed, e.g., [2, 50, 39], the computation and storage cost are prohibitive for large disk-resident graphs. Therefore, various methods compress the TC.

category	method	graph type	reachability query	path query	maintenance
searching	depth/breadth-first search [27]	all types	yes	yes	update adjacency lists
TC encoding	2-hop [22, 23]	all types, but small	yes	by including first-edge	not discussed
	HOPI [64, 65]	all types	yes	partially	based on method of [64]
	geometric [20] and graph partitioning 2-hop [21]	DAG	yes	not discussed	not discussed
	updatable 2-hop [15]	DAG	yes	not discussed	based on node-separation
	3-hop [40]	DAG	yes	not discussed	not discussed
	interval labelling [1]	DAG	yes	by computing ancestors	gaps in postorder numbers
	dual labeling [72]	DAG	yes	not discussed	not discussed
	GRIPP [71]	all types	yes	by computing descendants	not discussed
	path-cover [41]	DAG	yes	not discussed	not discussed

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

2-hop [22, 23] 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 [22] and [23] presents an approximation algorithm based on set covering [43] 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 [64, 65] reduces the construction time of 2-hop by exploiting graph partitioning. This approach 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 [64]. 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, [20] proposes a geometric-based method and [21] another one based on graph partitioning for the efficient construction of 2-hop. [40] 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 [41] 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. [72] proposes dual-labeling for sparse graphs. [15] 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 [71] 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, [71] does not handle frequent updates.

2.2 Problem Definition

This section formally defines the path query and introduces the basic notation that will be used in the rest of the chapter.

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

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

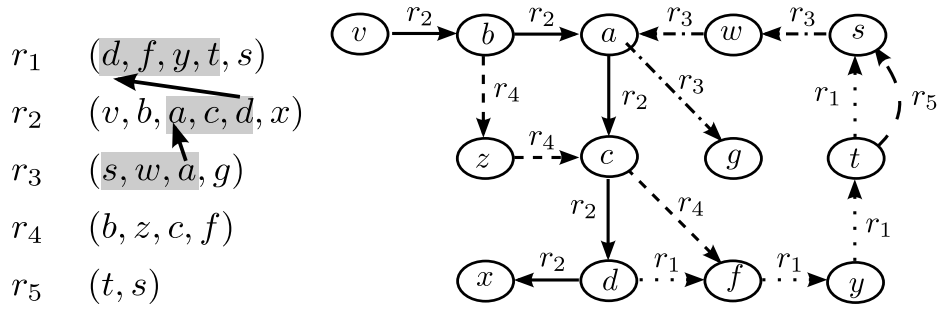
We denote the set of nodes in a route r as $nodes(r)$, and its length as $L_r = |nodes(r)|$.

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

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

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

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



(a) Route collection R (b) Routes graph G_R

Figure 2.1: *A route collection R, an answer to* PATH(s, t)*, and routes graph* G_R*.*

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

Note that a path may involve parts of routes from R.

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

Example 2.2. *Consider the route collection in Figure 2.1(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, and visit f, y and the target t.*

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

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

Example 2.3. *The collection R in Figure 2.1(a) is mapped to routes graph G_R in Figure 2.1(b). The different line styles of the edges denote the five routes in R.*

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

2.3 Route Traversal Search

Section 2.3.1 presents the $\mathcal{R}$-*Index* on route collections and details the RTS algorithm. Section 2.3.2 outlines the RTST algorithm that additionally exploits information about the transitions among routes stored in the $\mathcal{T}$-*Index* structure. Section 2.3.3 presents a detailed complexity analysis.

2.3.1 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 2.7 ($\mathcal{R}$-*Index*)**.** *Given a route collection R and a node $n_i \in nodes(R)$, 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 $n_i \in nodes(R)$.*

Example 2.4. *Table 2.2 illustrates the $\mathcal{R}$-Index for the routes shown in Figure 2.1(a).*

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

Algorithm RTS
Input: nodes n_s and n_t of a route collection 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 to $\mathcal{Q}$;
2. **insert** n_s in $\mathcal{H}$
3. **while** $\mathcal{Q}$ is not empty **do**
4. **pop** n_q from $\mathcal{Q}$; // pop current search node n_q
5. **if** there is a route $r_c \in R$ 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** // access each node n_r after n_q
   // in r until the first n_r node
   // contained in $\mathcal{H}$
9. **push** n_r to $\mathcal{Q}$;
10. **insert** n_r in $\mathcal{H}$;
11. **insert** $\langle n_r^-, n_r\rangle$ in $\mathcal{A}$; // where n_r^- is the direct
    // ancestor of n_r in r_i
12. **let** n_r be the next node in r_i;
13. **end while**
14. **end for**
15. **end while**
16. **return null**;

Figure 2.2: *The* RTS *algorithm.*

Figure 2.2 illustrates the pseudocode of the RTS algorithm. The algorithm takes as inputs: a route collection 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, or 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 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 $\mathcal{A}$.

RTS proceeds similarly to depth-first search. Initially, the stack $\mathcal{Q}$ and $\mathcal{H}$ contain source node n_s, while $\mathcal{A}$ is empty (Lines 1–2). Then, the algorithm proceeds examining the contents of the stack (Lines 3–15). At each iteration, RTS pops a single node n_q from $\mathcal{Q}$ on Line 4, and then, it checks the termination condition on Line 5. 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. 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 $\mathcal{H}$. 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 $\mathcal{A}$. The fact that only new nodes are inserted in $\mathcal{Q}$, ensures that RTS avoids cycles.

Example 2.5. *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 2.2. 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*

$$\mathcal{Q} = \{w, c, d\},$$
$$\mathcal{H} = \{s, w, a, g, c, d, x\} \text{ and}$$
$$\mathcal{A} = \{\langle s, w \rangle, \langle w, a \rangle, \langle a, g \rangle, \langle a, c \rangle, \langle c, d \rangle, \langle d, x \rangle\}.$$

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*).*

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 2.3: *$\mathcal{T}$-Index for the transition graph of Figure 2.3.*

2.3.2 The RTST algorithm

RTST expands the search similar to 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 2.8 (Transition graph)**.** *The* transition graph *of a route collection R is a labeled undirected graph $G_T(R, E_T)$, where its vertices are the routes in R, and a labeled edge (r_i, r_j, n_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 2.6. *Figure 2.3 illustrates the transition graph for the route collection of Figure 2.1(a).*

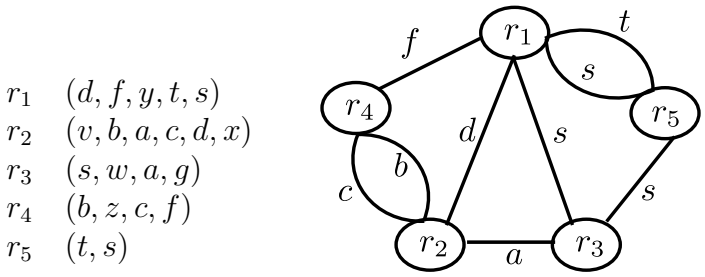


Figure 2.3: *Transition graph for the route collection R.*

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

Definition 2.9 (*$\mathcal{T}$-Index*)**.** *Given a route collection R, for each route $r_i \in R$, $trans[r_i]$ 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 $r_i \in R$.*

Example 2.7. *Table 2.3 illustrates the $\mathcal{T}$-Index for the G_T graph presented in Figure 2.3.*

Algorithm RTST
Input: nodes n_s and n_t of a route collection R, $\mathcal{R}$-*Index*, $\mathcal{T}$-*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 to $\mathcal{Q}$;
2. **insert** n_s in $\mathcal{H}$
3. **insert** $\langle n_s, \varnothing \rangle$ in $\mathcal{A}$;
4. **while** $\mathcal{Q}$ is not empty **do**
5. **pop** n_q from $\mathcal{Q}$;
6. **for** each entry $\langle r_i : o_{qi} \rangle$ in $routes[n_q]$ **do**
7. **if** there is an edge (r_i, r_j, n_ℓ) in G_T such that r_j contains n_t before link n_ℓ and r_i contains n_ℓ after n_q **then**
return ConstructPath$(n_s, n_q, n_t, \mathcal{A}, r_i : o_{\ell i}, r_j : o_{\ell j})$;
8. **let** n_r be the node after n_q in r_i;
9. **while** $n_r \notin \mathcal{H}$ **do** // access each node n_r after n_q
// in r until the first n_r node
// contained in $\mathcal{H}$
10. **push** n_r to $\mathcal{Q}$;
11. **insert** n_r in $\mathcal{H}$;
12. **insert** $\langle n_r^-, n_r \rangle$ in $\mathcal{A}$; // where n_r^- is the direct
// ancestor of n_r in r_i
13. **let** n_r be the next node in r_i;
14. **end while**
15. **end for**
16. **end while**
17. **return null**;

Figure 2.4: *The* RTST *algorithm.*

Figure 2.4 illustrates the pseudocode of the RTST algorithm. RTST proceeds similar to RTS, but involves a different termination check on Line 7. 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.

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 using ConstructPath. 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., $o_{\ell j} < o_{tj}$

Finally, note that compared to RTS, ConstructPath for RTST also 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.

Example 2.8. *Consider query* PATH(s, t) *on the route collection in Figure 2.1(a) indexed by the* $\mathcal{R}$*-Index of Table 2.2 and* $\mathcal{T}$*-Index of Table 2.3. The first two iterations of the* RTST *algorithm are identical to those of* RTS *in Example 2.5. 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)*.*

2.3.3 Complexity analysis

Given a route collection R, let $|R|$ denote the number of routes, $|N| = nodes(R)$ 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 $|R| \cdot L_r$ entries. Therefore, it occupies $O\left(\frac{|R| \cdot L_r}{B_{\mathcal{R}}}\right)$ pages. For the construction of $\mathcal{R}$-$Index$, the entire collection must be accessed at a cost of $O(\frac{|R| \cdot L_r}{B_{\mathcal{N}}})$ I/Os, while the index must be stored at a cost of $O\left(\frac{|R| \cdot L_r}{B_{\mathcal{R}}}\right)$ 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., $O\left(\frac{|R| \cdot L_r}{|N|}\right)$. In the worst case, a node can be contained in all routes, i.e., the $routes[]$ list has $O(|R|)$ entries. In the sequel, we assume the average case holds.

$\mathcal{T}$-*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 $O\left(\frac{|R|^2 \cdot L_r^2}{|N|}\right)$, while each list contains $O\left(\frac{|R| \cdot L_r^2}{|N|}\right)$ entries on average. The $\mathcal{T}$-$Index$ occupies $O\left(\frac{|R|^2 \cdot L_r^2}{|N| \cdot B_{\mathcal{T}}}\right)$ pages. Its construction requires accessing the entire $\mathcal{R}$-$Index$ for a cost of $O\left(\frac{|R| \cdot L_r}{B_{\mathcal{R}}}\right)$ I/Os, and writing the index on disk for $O\left(\frac{|R|^2 \cdot L_r^2}{|N| \cdot B_{\mathcal{T}}}\right)$ 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 $O\left(\frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}}\right)$ I/Os, and then retrieving all routes contained in $routes[n_q]$ for a cost of $O\left(\frac{|R| \cdot L_r}{|N|} \cdot \frac{L_r}{B_{\mathcal{N}}}\right)$ I/Os.

The second task is the termination condition. RTS and RTST need to retrieve the $routes[n_t]$ list of the target incurring $O\left(\frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}}\right)$ 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 $O\left(\frac{|R| \cdot L_r}{|N|} \cdot \frac{|R| \cdot L_r^2}{|N| \cdot B_{\mathcal{T}}}\right)$ I/Os.

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

2.4 Link Traversal Search

Section 2.4.1 discusses the shortcomings of the algorithms in Section 2.3, and proposes the *link traversal search* paradigm that overcomes them. Then, Sections 2.4.2, 2.4.3, 2.4.4 present three novel methods based on this paradigm. Section 2.4.5 discusses their complexity.

2.4.1 The link traversal search paradigm

Although the algorithms of Section 2.3 perform fewer iterations than conventional depth-first search on the route collection graph G_R, they share three shortcomings. First, they perform redundant iterations by visiting non-links. To understand this, consider that the current search node n_q is not a link and belongs to a single route r_i. Further, assume that the algorithm has visited n_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* $G_R^-(N^-, E^-)$ of the route collection, where $N^- \subseteq nodes(R)$ contains all links, and E^- contains all labeled directed edges (n_i, n_j, r_k) such that there exists a route $r_k \in R$ in which n_j is the link immediately following n_i. Note that G_R^- is *not explicitly materialized*, and is introduced to better illustrate the link traversal search paradigm. For example, Figure 2.5 shows the reduced routes graph G_R^- for the collection R of Figure 2.1(a). Observe the differences between G_R in Figure 2.1(b) and the reduced routes graph G_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 PATH(n_s, n_t) query on R is equivalent to finding a path from n_s to n_t in G_R^-, and replacing each (n_i, n_j, r_k) edge in the answer, with the subsequence from n_i to n_j of route r_k.

To tackle the second shortcoming, the algorithms reduce the cost of the termination check by precomputing a *target list* of routes, and checking if the current search

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

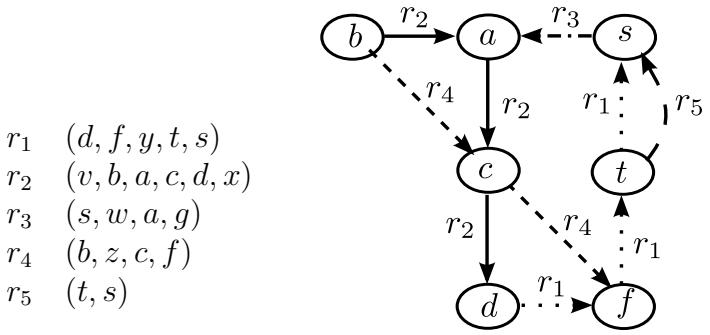


Figure 2.5: *Reduced routes graph for the route collection R.*

node belongs in one of them. This eliminates the recurring I/Os for the check, at the expense of a pre-processing cost for assembling the target list. Regarding the third shortcoming, the traversal policy of the new paradigm dictates that at each iteration only the links immediately after the current search node are inserted in the stack, exactly like a depth-first search on G_R^-.

2.4.2 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.

Definition 2.10 ($\mathcal{R}$-$Index^+$)**.** *Given a route collection R and a node $n_i \in nodes(R)$, $routes^+[n_i]$ 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 $n_i \in nodes(R)$.*

Example 2.9. *Table 2.4 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 2.4.1.

Figure 2.6 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 8–15).

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

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	$\langle r_4{:}2, c\rangle$

Table 2.4: *The $\mathcal{R}$-Index$^+$ for the route collection R.*

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 to $\mathcal{Q}$;
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. **pop** n_q from $\mathcal{Q}$;
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^+ to $\mathcal{Q}$;
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 2.6: *The* LTS *algorithm.*

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 $\langle n_q : r_i : o_{qi}, n_q^+\rangle$ 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 2.4.1.

Example 2.10. *We illustrate* LTS *for* PATH(s, t) *using the $\mathcal{R}$-Index$^+$ of Table 2.4.*

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:*

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

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.*

2.4.3 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 2.3.2.

Figure 2.7 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

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 to $\mathcal{Q}$;
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. **pop** n_q from $\mathcal{Q}$;
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^+ to $\mathcal{Q}$;
15. **insert** n_q^+ in $\mathcal{H}$;
16. **insert** $\langle n_q : r_i : o_{qi}, n_q^+ \rangle$ in $\mathcal{A}$;
17. **end if**
18. **end for**
19. **end while**
20. **return null**;

Figure 2.7: *The* LTST *algorithm.*

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.

Example 2.11. *Consider* PATH(s, t) *and* $\mathcal{R}$-$Index^+$ *and* $\mathcal{T}$-$Index$*, shown in Tables 2.4 and 2.3, 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 2.10. 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.*

2.4.4 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 2.8, 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_R^-. Since G_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 $o_{\ell j}$.

Example 2.12. *We illustrate the* LTS-1 *algorithm ($k=1$) for the* PATH(s,t) *query on the collection of Figure 2.1(a) using the $\mathcal{R}$-$Index^+$ presented in Table 2.4.*

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.*

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 to $\mathcal{Q}$;
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**
18. **while** $\mathcal{Q}$ is not empty **do**
19. **pop** n_q from $\mathcal{Q}$;
20. **for** each entry $\langle r_i : o_{qi}, n_q^+ \rangle$ in $routes^+[n_q]$ **do**
21. **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$(n_s, n_q, n_t, \mathcal{A}, r_i : o_{\ell i}, r_j : o_{\ell j})$;
22. **if** $n_q^+ \notin \mathcal{H}$ **then**
23. **push** n_q^+ to $\mathcal{Q}$;
24. **insert** n_q^+ in $\mathcal{H}$;
25. **insert** $\langle n_q : r_i : o_{qi}, n_q^+ \rangle$ in $\mathcal{A}$;
26. **end if**
27. **end for**
28. **end while**
29. **return null**;

Figure 2.8: *The* LTS-k *algorithm.*

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* $\mathcal{T}$*, 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 2.11, and* LTS-2 *would proceed identically to* LTST.

2.4.5 Complexity analysis

We use the notation introduced in Section 2.3.3. 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 $O\left(\frac{|R|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}\right)$ 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 $O\left(\frac{|R|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}\right)$ I/Os. In addition, LTST retrieves from $\mathcal{T}$-$Index$ the $trans[]$ lists for each route in $routes^+[n_t]$ at a cost of $O\left(\frac{|R|^2\cdot L_r^3}{|N|^2\cdot B_{\mathcal{T}}}\right)$ I/Os. On the other hand, LTS-k retrieves each route referenced in $routes^+[n_t]$ with $O\left(\frac{|R|\cdot L_r}{|N|}\cdot\frac{L_r}{B_{\mathcal{N}}}\right)$ I/Os, and then for each route it reads the $routes^+[]$ list of the k nodes before the target with $O\left(k\cdot\frac{|R|\cdot L_r}{|N|}\cdot\frac{|R|\cdot L_r}{|N|\cdot B^+_{\mathcal{R}}}\right)$ I/Os.

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

2.5 Updating Route Collections

Section 2.5.1 discusses the case when new routes are added in the collection, while Section 2.5.2 addresses deletions. Section 2.5.3 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 maintaining additional information in main memory. Then, at some time, a batch update process reflects all changes to the disk resident indices. Insertions are handled by *merging* memory-resident information with disk-based indices [75], while deletions require *rebuilding* of the affected lists.

2.5.1 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 2.4.4. 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 ± 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 $trans[r_i]$.

2.5.2 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 $trans[r_i]$.

2.5.3 Complexity analysis

We use the notation introduced in Section 2.3.3 and 2.4.5.

Insertions. We assume that $|R_{\mathcal{I}}|$ new routes are inserted in the existing route collection 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 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 $O\left(|N_{\mathcal{I}}|\right)$ I/Os. Then, we append the entries regarding the new routes writing $O\left(\frac{|R_{\mathcal{I}}| \cdot L_r}{B_{\mathcal{R}}}\right)$ 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 $\pm$ prefixed entries. When flushing $\mathcal{R}$-*Index*$^+$, for the $|N_{\mathcal{I}}^+|$ nodes, we read $|N_{\mathcal{I}}^+|$ and write $O\left(|N_{\mathcal{I}}^+| \cdot \frac{|R_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|}\right)$ 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 $O\left(\frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+} \cdot |N_{\mathcal{I}}^{+,\pm}|\right)$ I/Os and write $O\left(\left(\frac{|R| \cdot L_r}{|N|} + \frac{|R_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|}\right) \cdot \frac{|N_{\mathcal{I}}^{+,\pm}|}{B_{\mathcal{R}}^+}\right)$ pages. Thus, flushing $\mathcal{R}$-*Index*$^+$ requires reading $O\left(|N_{\mathcal{I}}^+| + \frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+} \cdot |N_{\mathcal{I}}^{+,\pm}|\right)$ and writing $O\left(\frac{|N_{\mathcal{I}}^+| \cdot |R_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|} + \left(\frac{|R| \cdot L_r}{|N|} + \frac{|R_{\mathcal{I}}| \cdot L_r}{|N_{\mathcal{I}}|}\right) \cdot \frac{|N_{\mathcal{I}}^{+,\pm}|}{B_{\mathcal{R}}^+}\right)$ pages.

Regarding $\mathcal{T}$-*Index*, buffering requires $O\left(\frac{|R| \cdot L_r^2}{|N| \cdot B_{\mathcal{R}}}\right)$ 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 $|R_{\text{aff}}|$ of the old routes, connections with the new routes are identified. Therefore, for these $|R_{\text{aff}}|$ routes, flushing requires reading $|R_{\text{aff}}|$ pages

and writing $O\left(|R_{\text{aff}}| \cdot \frac{|R_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}| \cdot B_{\mathcal{T}}}\right)$. On the other hand for the new routes, we need to write $O\left(\frac{|R_{\mathcal{I}}|}{B_{\mathcal{T}}} \cdot \left(\frac{|R_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}|} + \frac{|R| \cdot L_r^2}{|N|}\right)\right)$ pages. To sum up, flushing $\mathcal{T}$-$Index$ requires reading $|R_{\text{aff}}|$ and writing $O\left(|R_{\text{aff}}| \cdot \frac{|R_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}| \cdot B_{\mathcal{T}}} + \frac{|R_{\mathcal{I}}|}{B_{\mathcal{T}}} \cdot \left(\frac{|R_{\mathcal{I}}| \cdot L_r^2}{|N_{\mathcal{I}}|} + \frac{|R| \cdot L_r^2}{|N|}\right)\right)$ 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 $O\left(\frac{L_r}{B_{\mathcal{N}}} + L_r \cdot \frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}}\right)$ (resp. $O\left(\frac{L_r}{B_{\mathcal{N}}} + L_r \cdot \frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+}\right)$) 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 $O\left(L_r \cdot \frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}}\right)$ (resp. $O\left(L_r \cdot \frac{|R| \cdot L_r}{|N| \cdot B_{\mathcal{R}}^+}\right)$) pages for the updated $routes[]$ (resp. $routes^+[]$) lists. Finally, flushing $\mathcal{T}$-$Index$ requires reading and writing $O\left(L_r \cdot \frac{|R| \cdot L_r}{|N|} \cdot \frac{|R| \cdot L_r^2}{|N| \cdot B_{\mathcal{T}}}\right)$ pages for each deleted route.

2.6 Experimental Analysis

Section 2.6.1 details the setting, while Sections 2.6.2, 2.6.3 and 2.6.4 evaluate index construction, querying and index maintenance, respectively, of all methods.

2.6.1 Setup

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

We generate synthetic route collections varying the following parameters (Table 2.5): (1) the number of routes in the collection, $|R|$, (2) the route length, L_r, (3) the number of distinct nodes in the routes, $|N|$, and (4) the links/nodes ratio α. In each experiment, we vary one of the parameters while we keep the others to their default values.

parameter	values	default value
$\|R\|$	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 2.5: *Experimental parameters*

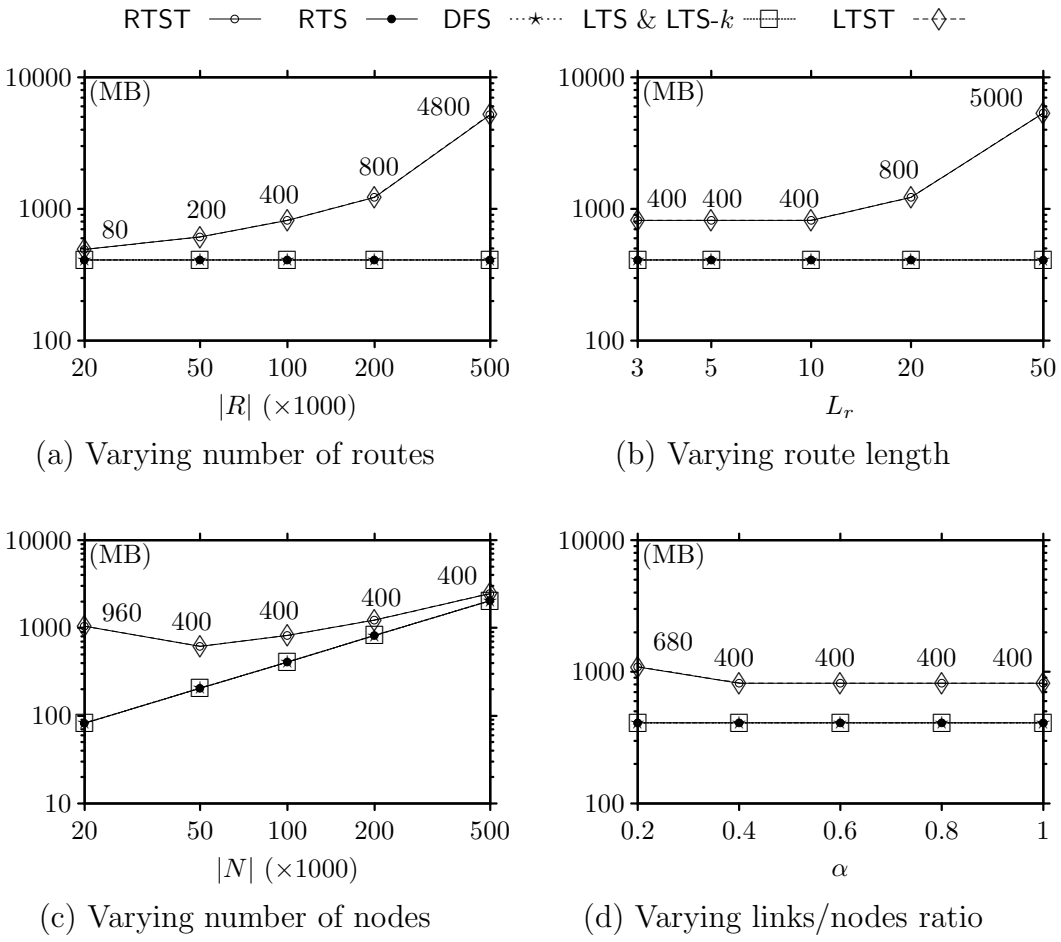
2.6.2 Index size and construction cost

For each method, we measure the time spent to construct the necessary indices and their storage requirement. Table 2.6 shows the indices employed by each method.

Varying the number of routes $|R|$. The disk space requirement of the $\mathcal{R}$-$Index$/

input	method name	index
reduced routes graph G_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 2.6: *Methods for evaluating* `PATH` *queries*



(a) Varying number of routes

(b) Varying route length

(c) Varying number of nodes

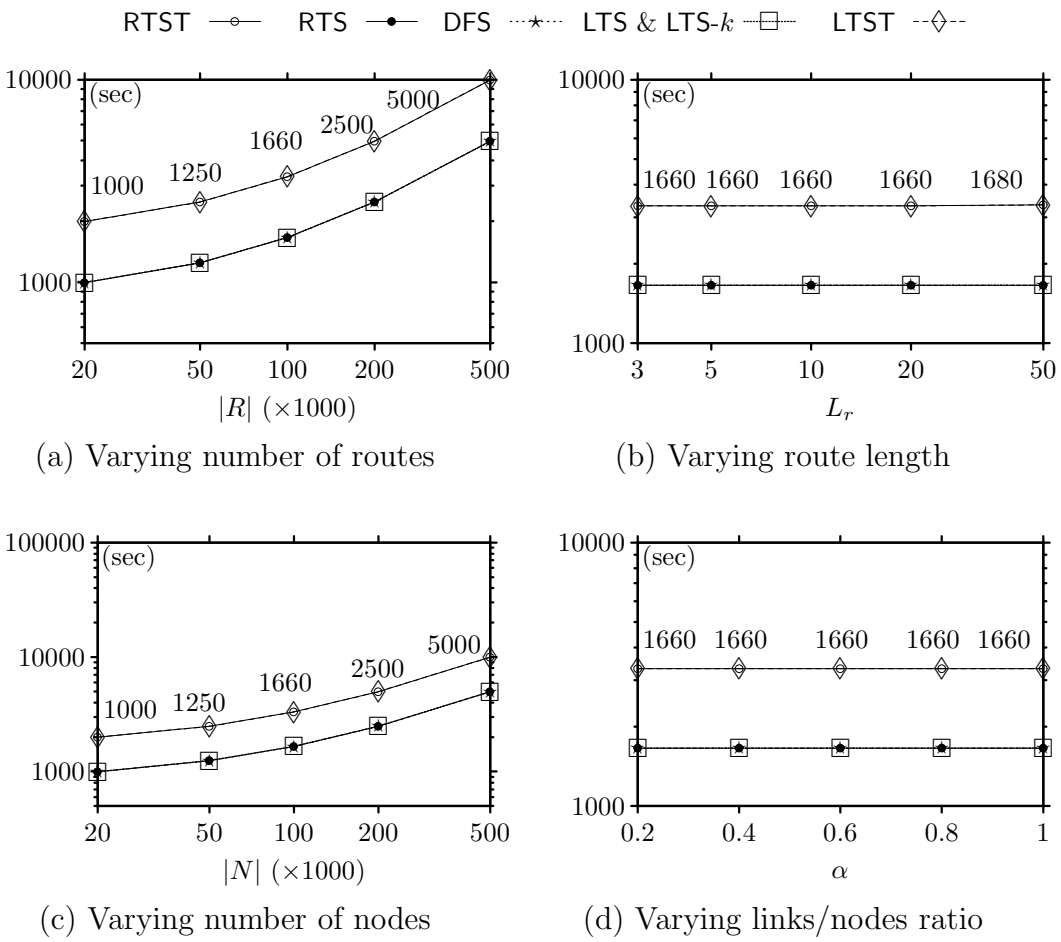
(d) Varying links/nodes ratio

Figure 2.9: *Indices space consumption.*

$\mathcal{R}$-$Index^+$, employed by RTS, LTS, LTS-k, and DFS, [2] depends primarily on the number of nodes $|N|$ (which determines the number of lists $routes[]/routes^+[]$) and not on $|R|$ (which affects the length of the lists). Hence, Figure 2.9(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 $|R|$ increases, the number of edges in the transition graph,

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



(a) Varying number of routes

(b) Varying route length

(c) Varying number of nodes

(d) Varying links/nodes ratio

Figure 2.10: *Indices construction time.*

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 2.10(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 $|R| = 500$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. [3]

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 2.9(b). This is because, as explained in the context of Figure 2.9(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 2.10(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,

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

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 2.9(c). Increasing the number of nodes, while $|R|$ and L_r remain fixed, causes an increase in the number of links (in absolute values). This makes each link appear fewer times in the routes and thus the number of edges in G_T decreases. The space requirement of $\mathcal{T}$-$Index$ decrease from 960MB to its minimum 400MB (1 page for each of the 100K routes). Figure 2.10(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 2.9(c). As the number of links increases ($|R|$ and L_r remain fixed), the transition graph becomes sparser, as explained in the context of Figure 2.9(c); this accounts for the decrease in the space $\mathcal{T}$-$Index$ occupies from 700MB to 400MB. Figure 2.10(d) shows that the construction times for all indices are unaffected by α.

2.6.3 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 2.6.3.1 and 2.6.3.2 all considered queries have an answer, i.e., a path exists; the case of queries with no answer is investigated in Section 2.6.3.3.

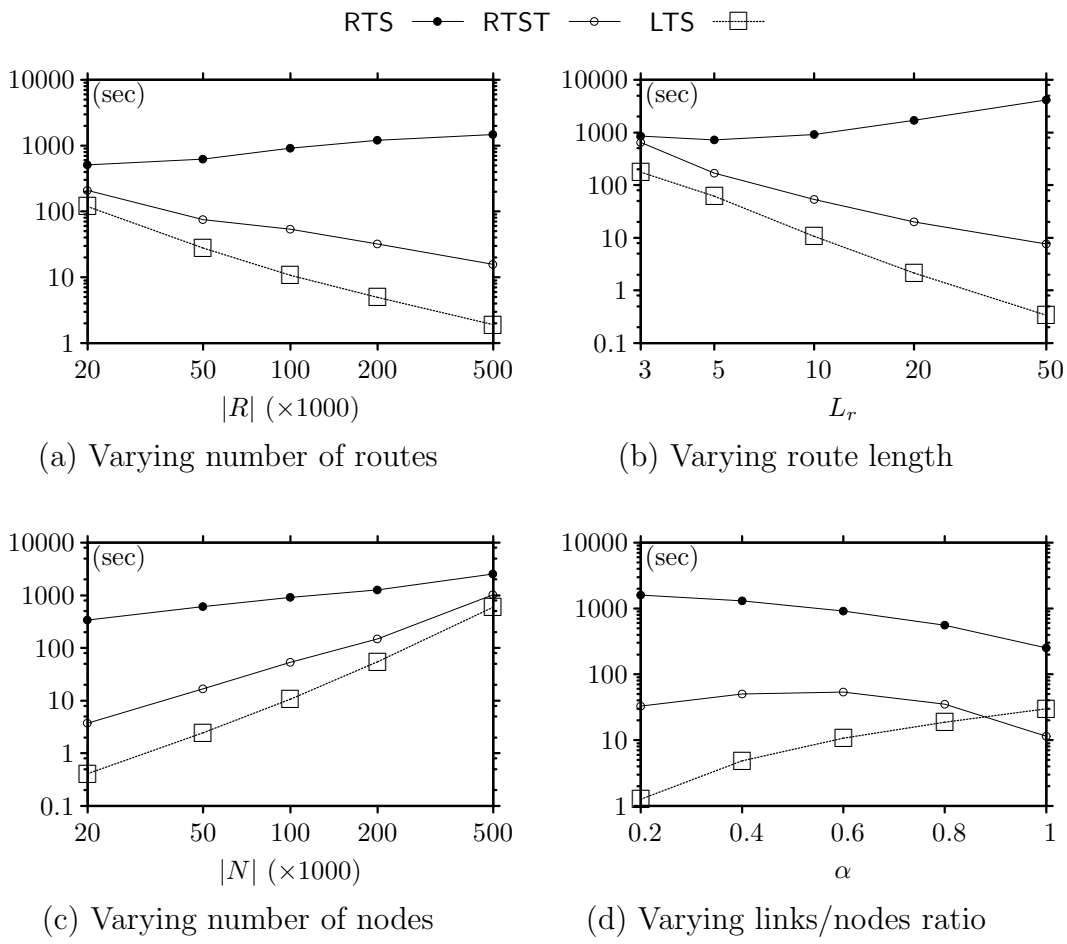
2.6.3.1 Route vs link traversal search

We compare the route traversal search methods RTS and RTST against the basic link traversal search algorithm LTS in terms of the execution time, while varying $|R|$, L_r, $|N|$ and α in Figures 2.11(a), (b), (c) and (d), respectively.

Varying the number of routes $|R|$. As $|R|$ increases, finding a path between two nodes becomes easier. This is exhibited by RTST and LTS in Figure 2.11(a). In contrast, the execution time of RTS increases with $|R|$ as it performs more iterations compared to RTST, which has a stronger termination condition, and to LTS, which only visits links.

Varying the route length L_r. The same observations hold when the route length increases in Figure 2.11(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 2.11(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



(a) Varying number of routes

(b) Varying route length

(c) Varying number of nodes

(d) Varying links/nodes ratio

Figure 2.11: *Link vs route traversal search: execution time.*

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 $\alpha = 0.2$ the link frequency, i.e., the (average) number of routes in which it appears, is 46, and becomes 10 when $\alpha = 1$. In general, as the ratio increases it becomes harder to find a path. Hence, as 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.

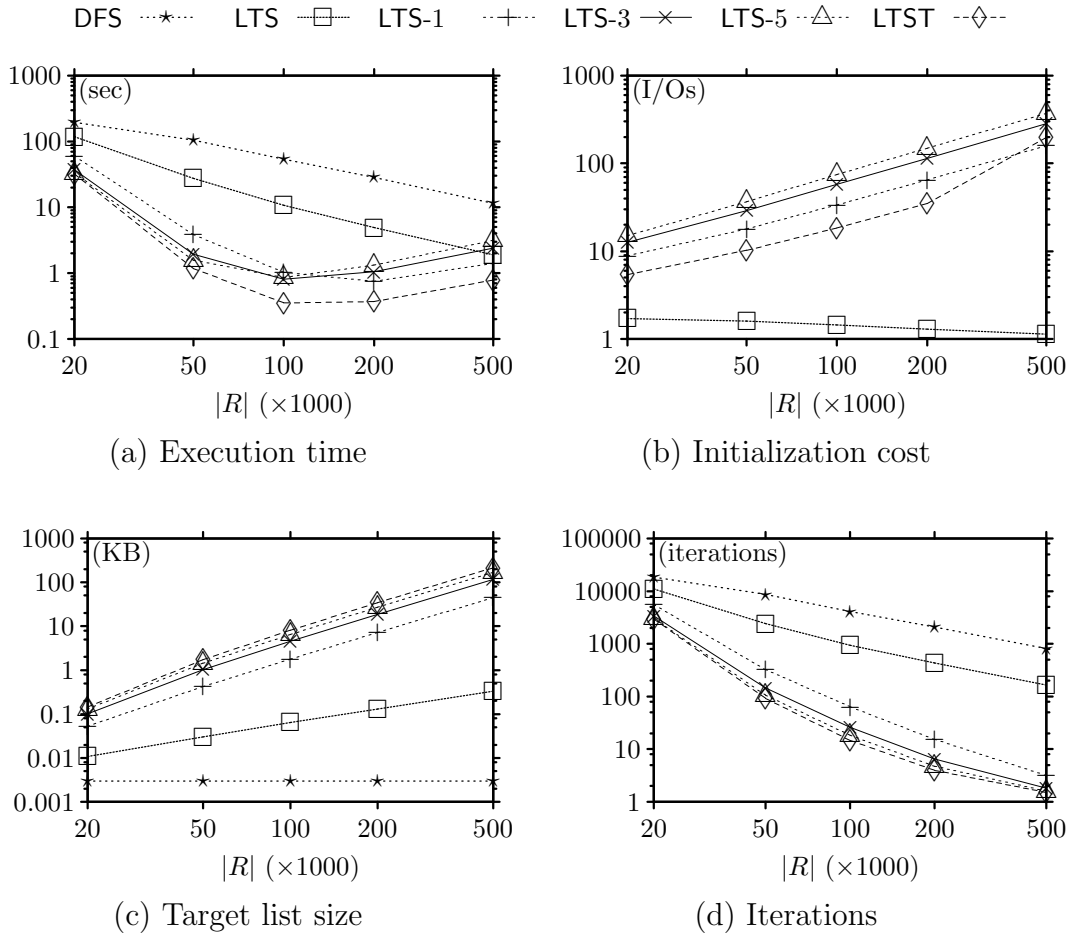
When $\alpha = 1$, all nodes are links (the reduced routes graph G_R^- reduces to the G_R graph) and thus LTS visits exactly the same nodes with RTS. Still LTS is around 8.6 times faster, as it performs fewer accesses per iteration. As before, the argument applies to LTST (not shown in the figure), which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

2.6.3.2 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 2.4.1. 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.



(a) Execution time (b) Initialization cost

(c) Target list size (d) Iterations

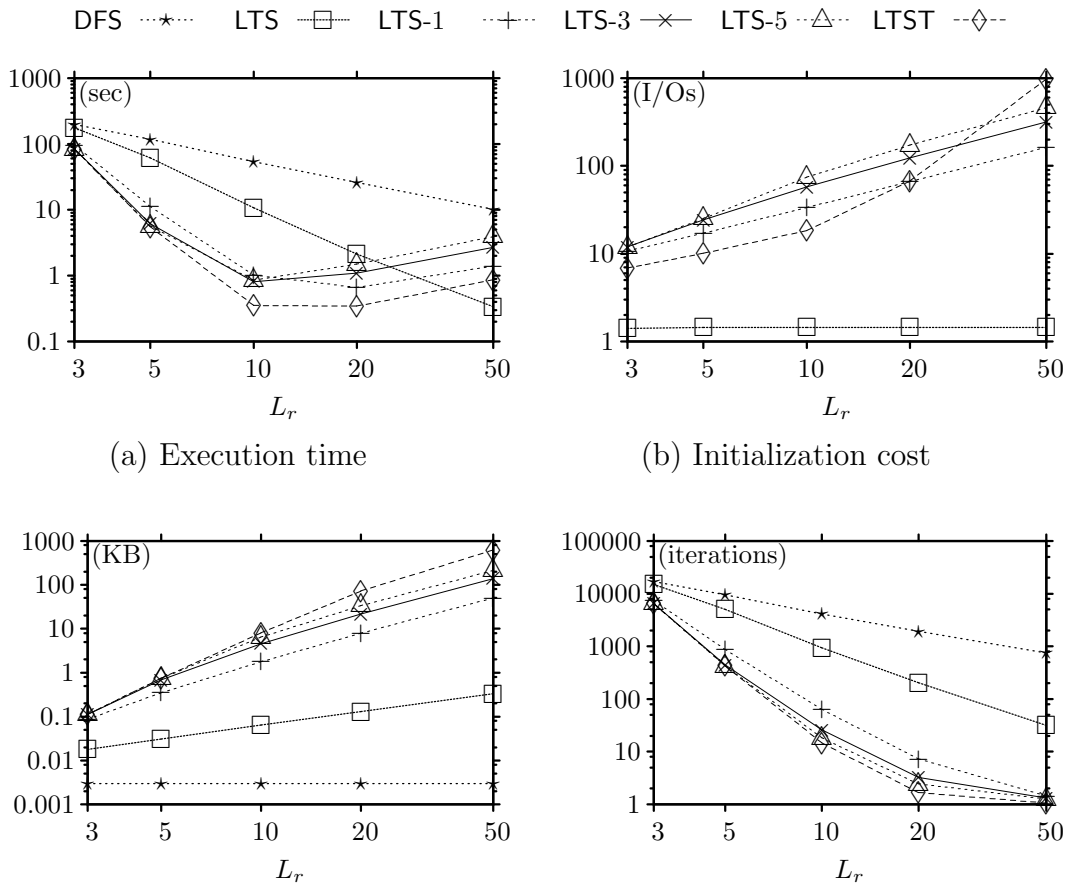
Figure 2.12: *Link traversal search vs* DFS*: varying number of routes.*

Varying the number of routes $|R|$. As $|R|$ increases, every link is contained in more routes and the number of iterations decreases, as shown in Figure 2.12(d). In the initialization phase (after retrieving the links following the source and preceding the target, if necessary), LTS retrieves only $routes^+[n_t]$ and assembles the target list; this has a constant cost as shown in Figure 2.12(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 $|R|$, the initialization cost also increases.

Similar observations apply for the size of the target list $\mathcal{T}$, which increases for all methods as the G_R^- graph becomes denser; see Figure 2.12(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 2.12(c) and 2.12(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 $|R|$, with LTS becoming up to one order of magnitude faster, as shown in Figure 2.12(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.

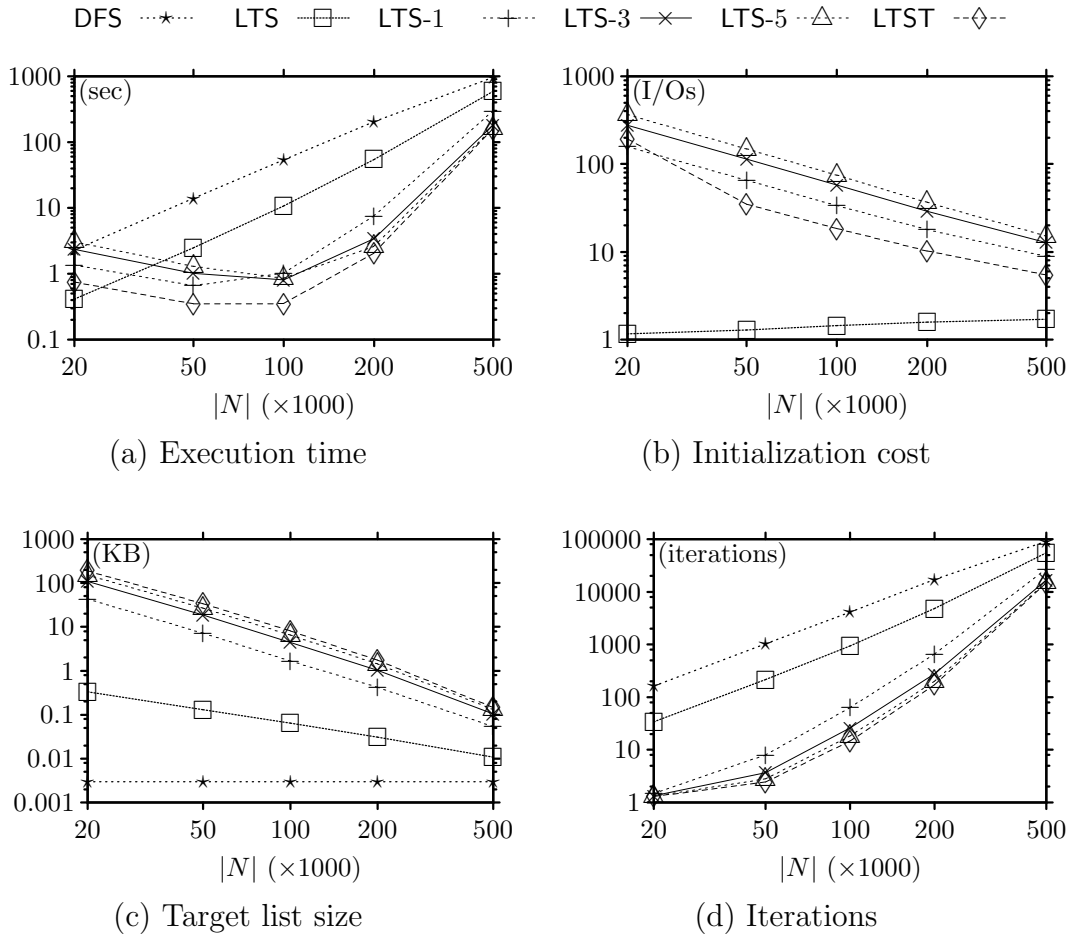


(a) Execution time (b) Initialization cost

(c) Target list size (d) Iterations

Figure 2.13: *Link traversal search vs* DFS*: varying route length.*

Varying the route length L_r. Figure 2.13 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_R^- graph becomes more dense. Therefore, all algorithms exhibit the same trends as in Figure 2.12. Note that LTS becomes the fastest method for $L_r = 50$ outperforming DFS by almost two orders of magnitude.



(a) Execution time (b) Initialization cost

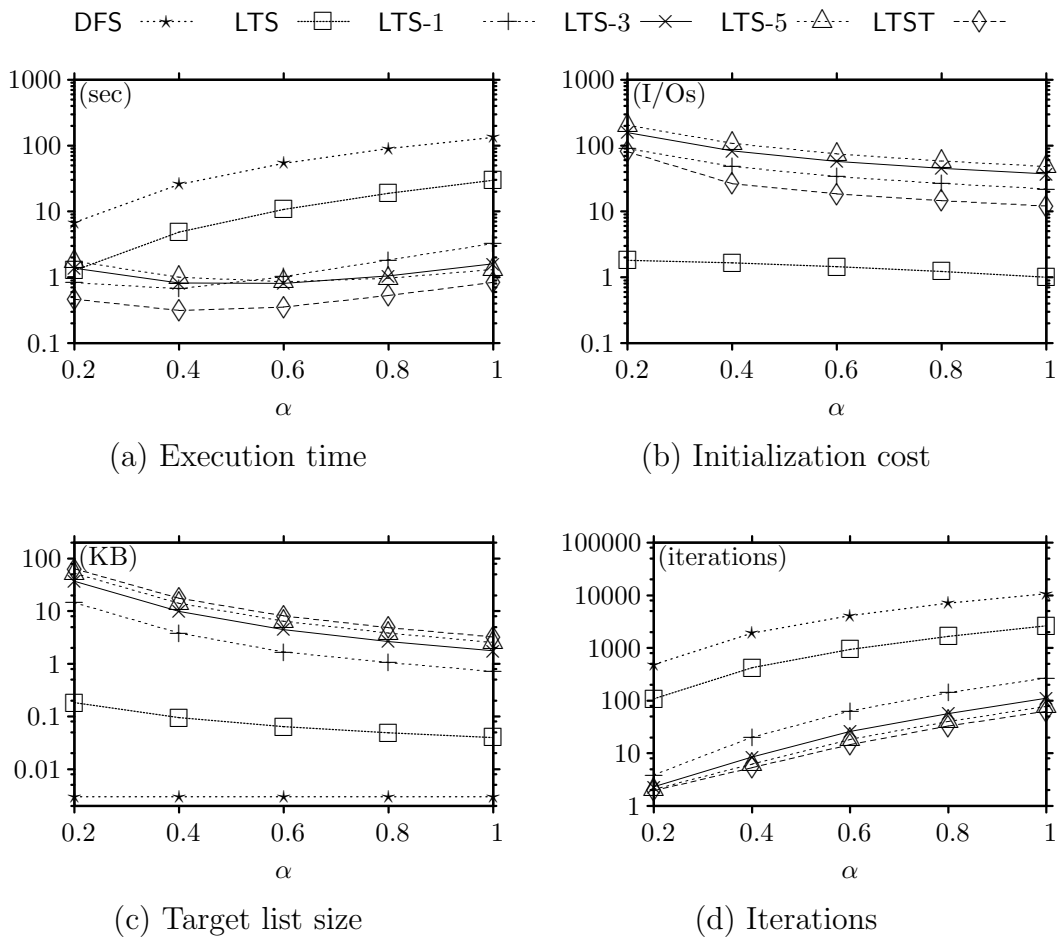
(c) Target list size (d) Iterations

Figure 2.14: *Link traversal search vs* DFS*: varying number of nodes.*

Varying the number of nodes $|N|$. Figure 2.14 studies the effect of increasing $|N|$, 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_R^- becomes sparser, which means that finding a path becomes harder. This is verified in Figure 2.14(c), which depicts that the target list decreases with $|N|$, and in Figure 2.14(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 2.12, 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 perform more iterations. This also holds for LTST, LTS-k for collections of more than 100K nodes. For fewer nodes, the initialization cost of LTST, LTS-k (see Figure 2.14(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.



(a) Execution time (b) Initialization cost

(c) Target list size (d) Iterations

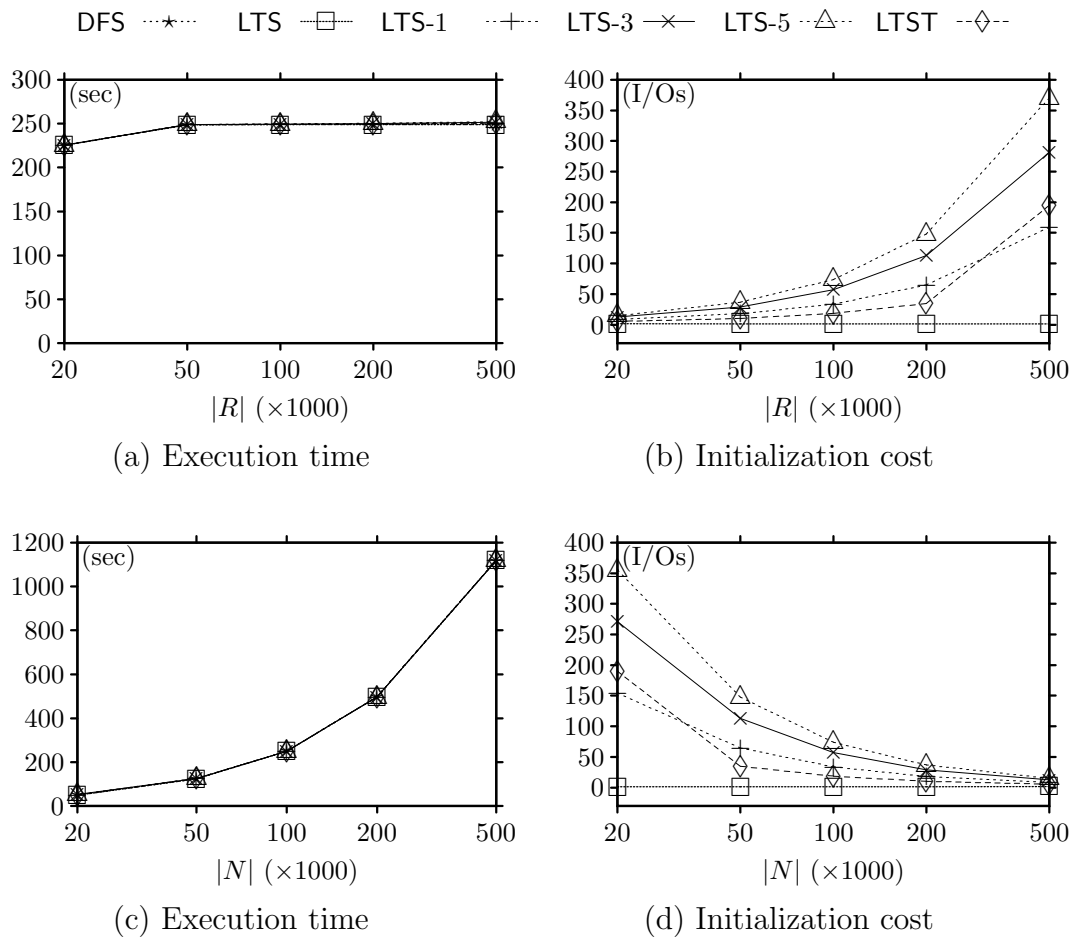
Figure 2.15: *Link traversal search vs* DFS*: varying links/nodes ratio.*

Varying the links/nodes ratio α. As α increases the link frequency decreases and finding a path becomes more difficult, as shown in Figure 2.15(d). For the reasons discussed in the case of varying $|N|$, the target list of all methods decreases with $|N|$ (see Figure 2.15(c)), and the initialization cost of LTST and LTS-k decreases (see Figure 2.15(b)). Correspondingly, the total execution time increases for DFS and LTS, while it first decreases and ultimately increases for LTST and LTS-k.

2.6.3.3 `PATH` queries with no answer

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

Figures 2.16(a) and 2.16(b) display the total execution time and the cost of the initialization phase while the number of routes varies. In accordance to Figure 2.12(b), the initialization cost of LTST and LTS-k increases with $|R|$, while that of LTS remains fixed. In the core phase, all methods perform the same iterations, traversing all links in the component of the source. The number of links in the component do not change with $|R|$. Furthermore, since the execution time is dominated by the traversal cost, all methods require around 250 seconds to determine that a



(a) Execution time (b) Initialization cost

(c) Execution time (d) Initialization cost

Figure 2.16: `PATH` *queries with no answer: execution time and initialization cost.*

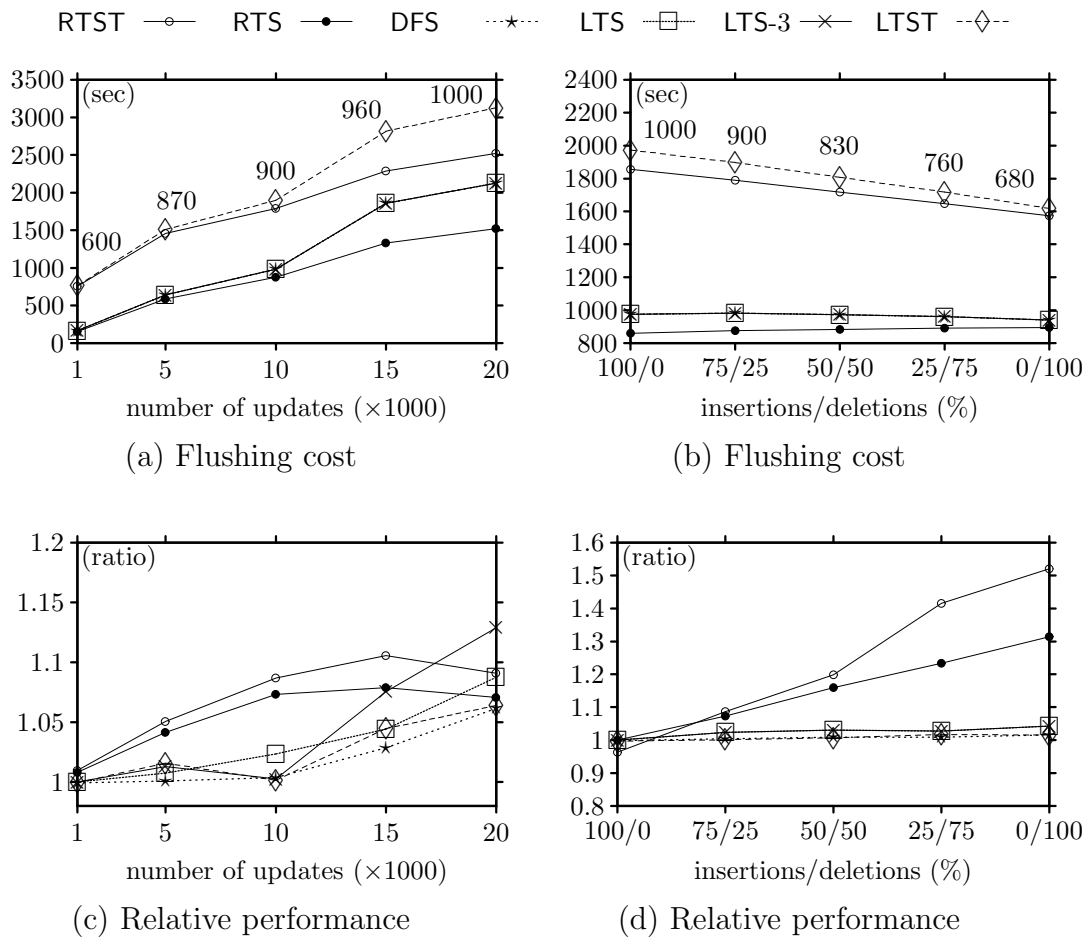
path does not exist, as shown in Figure 2.16(b). The total execution time of LTST and LTS-k, slightly increases with $|R|$ due to the higher initialization cost. In the worst case, the overhead is less than 1.3%.

Figures 2.16(c) and 2.16(d) repeat the measurements while the number of nodes varies. As before, the execution time is dominated by the cost of the core phase, which is identical for all methods. However, since the number of nodes in the source component increases, the performance of all methods degrades with N. In the worst case, when $N = 20$K, LTS-5 is about 6% slower than DFS.

2.6.4 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 50K routes of length $L_r = 10$ and 100K 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;



(a) Flushing cost (b) Flushing cost

(c) Relative performance (d) Relative performance

Figure 2.17: *Updating route collections.*

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

Figure 2.17(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 $|R|$), the flushing cost for the $\mathcal{T}$-$Index$ based methods is lower than the cost of rebuilding indices.

Figure 2.17(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 2.17(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 2.17(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 2.17(d) at the 100%/0% insertions/deletions ratio.

2.7 Conclusions

In this chapter, we considered 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.

Chapter 3

Dynamic Pickup and Delivery with Transfers

The family of pickup and delivery problems covers a broad range of optimization problems that appear in various logistics and transportation scenarios. Broadly speaking, these problems look for an assignment of a set of transportation requests to a fleet of vehicles in a way that satisfies a number of constraints and at the same time minimizes a specific cost function. In this context, a transportation request is defined as *picking up* an object (e.g., package, person, etc.) from one location and *delivering* it to another; hence the name.

In its simplest form, the *Pickup and Delivery Problem* (PDP) only imposes two constraints. The first, termed *precedence*, naturally states that pickup should occur before delivery for each transportation request. The second, termed *pairing*, states that both the pickup and the delivery of each transportation request should be performed by the same vehicle. The *Pickup and Delivery Problem with Transfers* (PDPT) [28, 53] is a PDP variant that eliminates the pairing constraint. In PDPT, objects can be transferred between vehicles. *Transfers* can occur in predetermined locations, e.g., depots, or in arbitrary locations as long as the involved vehicles are in close proximity to each other at some time. We refer to the latter case as *transfer with detours*, since the vehicles may have to deviate from their routes.

Almost every pickup and delivery problem comes in two flavors. In *static*, all requests are known in advance and the goal is to come up with the best vehicle routes from scratch. On the other hand, in *dynamic*, a set of vehicle routes, termed the *static plan*, has already been established. Then, additional requests arrive ad hoc, i.e., at arbitrary times, and the plan must be modified to satisfy them. While algorithms for static problems can also solve the dynamic counterpart, they are rarely used as they take a lot of time to execute. Instead, common practice is to apply two-phase local search algorithms. In the first phase, a quick solution is obtained by assigning each standing request to the vehicle that results in the smallest cost increase. In the second phase, the obtained solution is improved by reassigning requests.

This chapter proposes a methodology for the *dynamic Pickup and Delivery Problem with Transfers* (`dPDPT`). Although works for the dynamic PDP can be extended to consider transfers between vehicles, to the best of our knowledge, this is the first work targeting `dPDPT`. Our solution processes requests independently, and does not follow the two-phase paradigm. Satisfying a request is treated as a shortest path

problem in a conceptual graph. Intuitively, the object must travel from the pickup to the delivery location following the vehicles' routes and schedules.

Based on these observations, the contributions of our work in this chapter can be summarized as follows:

(1) We formulate dPDPT as a graph problem. For this purpose, we introduce a conceptual graph, called *dynamic plan graph* that captures all possible *actions* for picking up a package and delivering it to the destination.

(2) To satisfy a dPDPT request, we compute the shortest path p on the dynamic plan graph considering the operational and the customer cost of p. The *operational cost* O_p measures the additional time (total delay), with respect to the static plan, incurred by the vehicles in order to accommodate the solution p. The *customer cost* C_p is the delivery time of the object. These costs are in general conflicting, as they represent two distinct views. For example, the path with the earliest delivery time may require significant changes in the schedule of the vehicles and cause large delays on the static plan. In contrast, the path with the smallest operational cost could result in late delivery. In this work, we consider the operational cost as more important; the customer cost is used to solve ties.

(3) We show that computing the shortest path, according to the operational and the customer cost, on the dynamic plan graph is not straightforward. The reason is that the weights of the edges depend on both the operational and customer cost of the path that led to this edge.

(4) We show, contrary to other time-dependent networks, that the dynamic plan graph does not exhibit the *principle of optimality*, which is necessary to apply efficient Bellman-Ford or Dijkstra-like algorithms. Hence, we have to enumerate all possible paths, and for this purpose, we introduce the SP algorithm.

(5) We come up with a breakdown of the operational cost in two new cost metrics, and introduce the *modified dynamic plan graph*. The *current operational cost* O_p^* is the delay incurred by the vehicle currently carrying the package. The residual operational cost O_p^R is the delay incurred by all other vehicles. Then, we show that the modified dynamic plan graph exhibits the principle of optimality, which allows us to eliminate paths and dealing with the shortcomings of SP. To this end, we present the SPM algorithm.

(6) We perform an extensive experimental analysis verifying the advantage of SPM over SP and demonstrating that SPM is significantly faster than a two-phase local search method adapted for dPDPT, while the quality of the solution is only marginally lower.

The remainder of this chapter is organized as follows. Section 3.1 reviews related work in detail. Section 3.2 formally defines dynamic Pickup and Delivery with Transfers. Section 3.3 introduces the dynamic plan graph and Section 3.4 proposes the SP algorithm. Then, Section 3.5 introduces the modified dynamic plan graph and Section 3.6 proposes the SPM algorithm that efficiently solves dPDPT. Finally, Section 3.7 presents an extensive experimental evaluation and Section 3.8 concludes this work.

3.1 Related Work

Our work in this chapter is related to pickup and delivery, and to shortest path problems.

Pickup and Delivery Problem. In the *Pickup and Delivery Problem* (PDP) objects must be transported by a fleet of vehicles from a pickup to a delivery location with the minimum cost, under two constrains: (1) pickup occurs before delivery (*precedence*), and (2) pickup and delivery of an object is performed by the same vehicle (pairing). PDP is NP-hard since it generalizes the well-known Traveling Salesman Problem (`TSP`). Exact solutions employ column generation [32, 68, 73], branch-and-cut [26, 62] and branch-and-cut-and-price [61] methodologies. On the other hand, the heuristics for the approximation methods take advantage of local search [4, 49, 53, 63].

Other PDP variations introduce additional constraints. For instance, in the *Pickup and Delivery Problem with Time Windows* (PDPTW), pickups and deliveries are accompanied with a time window that mandates when the action can take place. In the *Capacitated Pickup and Delivery Problem* (CPDP), the amount of objects a vehicle is permitted to carry at any time is bounded by a given capacity. In the *Pickup and Delivery Problem with Transfers* (DPDT), studied in this paper, the pairing constraint is lifted. [28] proposes a branch-and-cut strategy for DPDT. [53] introduces the Pickup and Delivery Problem with Time Windows and Transfers and employs a local search optimization approach. In all the above problems, the transportation requests are known in advance, hence they are characterized as *static*. A formal definition of static PDP and its variants can be found in [6, 25, 56].

Almost all PDP variants also have a *dynamic* counterpart. In this case, a set of vehicle routes, termed the *static plan*, has already been established, and additional requests arrive ad hoc, i.e., at arbitrary times. Thus, the plan must be modified to satisfy them. A survey on dynamic PDP can be found in [5]. Typically, two-phase local search methods are applied for the dynamic problems. The first phase applies an insertion heuristic [58], whereas the second employs tabu search [33, 51, 52]. To the best of our knowledge our work is the first to address the *dynamic Pickup and Delivery Problem with Transfers* (`dPDPT`).

Shortest Path Problem and its variants. Bellman-Ford and Dijkstra are the most well-known algorithms for finding the *shortest path* between two nodes in a graph. The ALT algorithms [34, 35, 57] perform a bidirectional A* search and exploit a lower bound of the distance between two nodes to direct the search. To compute this bound they construct an embedding on the graph. There exist a number of materialization techniques [3, 42, 44] or encoding/labeling schemes [19, 22] that can be used to efficiently compute the shortest path. Both the ALT algorithms and the materialization and encoding methods are mostly suitable for graphs that are not frequently updated, since they require expensive precomputation.

In *multi-criteria shortest path problems* the quality of a path is measured by multiple metrics, and the goal is to find all paths for which no better exists. Algorithms are categorized into three classes. The methods of the first class (e.g., [18]) apply a user preference function to reduce the original multi-criteria problem to a conventional shortest path problem. The second class contains the interactive methods (e.g., [36]) that interact with a decision maker to come up with the answer

path. Finally, the third class includes label-setting and label-correcting methods (e.g., [37, 47, 70]). These methods construct a label for every path followed to reach a graph node. Then, at each iteration, they select the path with the minimum cost, defined as the combination of the given criteria, and expand the search extending this path.

In *time-dependent shortest path problems* the cost of traveling from node n_i to n_j in a graph (e.g., the road network of a city) depends on the departure time t from n_i. [24] is the first attempt to solve this problem using a Bellman-Ford based solution. However, as discussed in [31], Dijkstra can also be applied for this problem, as long as the earliest possible arrival time at a node is considered. In the context of transportation systems, the *FIFO* (a.k.a. *non-overtaking*) property of a road network is considered as a necessity in order to achieve an acceptable level of complexity. According to this property delaying the departure from a graph node n_i to reach n_j cannot result in arriving earlier at n_j. However, even when the FIFO property does not hold it is possible to provide an efficient solution [30, 55] by properly adjusting the weights in graph edges [30].

3.2 Problem Definition

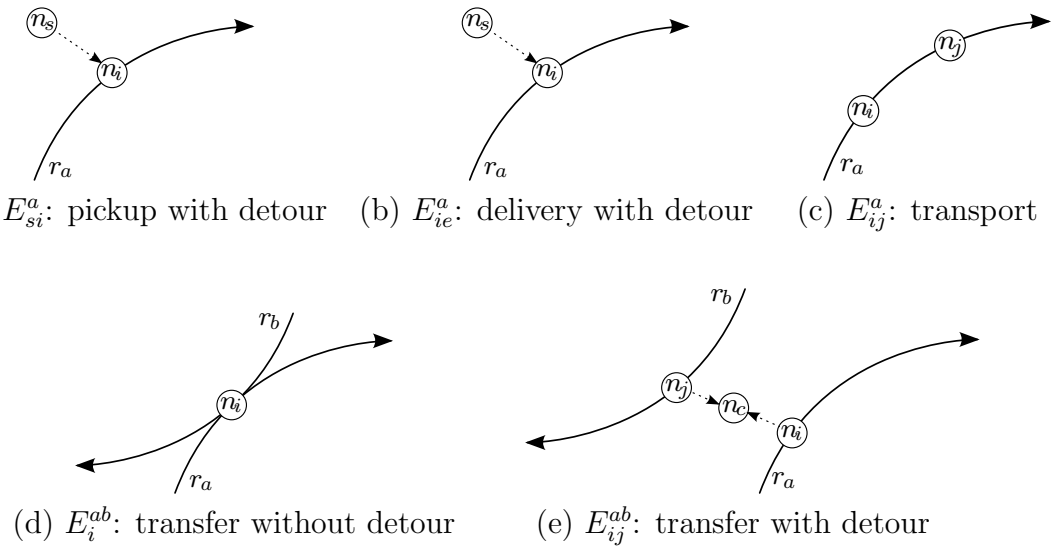
Section 3.2.1 provides basic definitions and introduces the dynamic Pickup and Delivery Problem with Transfers. Section 3.2.2 details the actions allowed for satisfying a request and their costs.

3.2.1 Definitions

Assume that a company has already scheduled its fleet of vehicles to service a number of requests. We refer to this schedule as the *static plan*, since we assume that it is given as input. The static plan consists of a set of vehicles following some routes; we overload notation r_a to refer to both a vehicle and its route. The *route* of a vehicle r_a is a sequence of distinct spatial locations, where each location n_i is associated with an *arrival time* A_i^a and a *departure time* D_i^a. Note that the requirement for distinct locations within a route is introduced to simplify notation and avoid ambiguity when referring to a particular location. Besides, if a vehicle visits a location multiple times, its route can always be represented as a set of distinct-locations routes. The difference $D_i^a - A_i^a$ is a non-negative number; it may be zero indicating that vehicle r_a just passes by n_i, or a positive number corresponding to some service at n_i, e.g., pickup, delivery, mandatory stop, etc. For two consecutive locations n_i and n_j on r_a, the difference $A_j^a - D_i^a \geq 0$ corresponds to the travel time from n_i to n_j.

An ad-hoc `dPDPT` *request* is a pair of locations n_s and n_e, signifying that a package must be picked up at n_s and be delivered at n_e. In order to complete a request, it is necessary to perform a series of modifications to the static plan. There are five types of modifications allowed, termed *actions*: pickup, delivery, transport, transfer without detours, and transfer with detours. Each action, described in detail later, results in the package changing location and/or vehicle. A sequence of actions is called a *path*. If the initial and final location of the package in a path are n_s and n_e, respectively, the path is called a *solution* to the request.

There are two costs associated with an action. The *operational cost* measures the time spent by vehicles in order to perform the action, i.e., the delay with respect to



(a) E^a_{si}: pickup with detour (b) E^a_{ie}: delivery with detour (c) E^a_{ij}: transport

(d) E^{ab}_i: transfer without detour (e) E^{ab}_{ij}: transfer with detour

Figure 3.1: *Actions allowed for satisfying a* `dPDPT` *request.*

the static plan. The *customer cost* represents the time when the action is completed. Furthermore, the operational cost O_p of a path p is defined as the sum of operational costs for each action in the path, and the customer cost C_p is equal to the customer cost of the final action in p. Therefore, for a solution p, O_p signifies the company's cost in accommodating the request, while C_p determines the delivery time of the package according to p.

Any monotonic combination (e.g., weighted sum, product, min, max, average etc.) of the two costs could be a meaningful measure for assessing the quality of a solution. In the remainder of this chapter, we assume that the operational cost is more important, and that the customer cost is of secondary importance. Therefore, a path p is preferred over q, if $O_p < O_q$, or $O_p = O_q \wedge C_p < C_q$. Equivalently, we may define the *combined cost* of a path p as:

$$cost(p) = M \cdot O_p + C_p \ , \tag{3.1}$$

where M is a sufficiently large number (greater than the largest possible customer cost divided by the smaller possible operational cost) whose sole purpose is to assign greater importance to the operational cost. Based on this definition, the *optimal solution* is the one that has the lowest combined cost, i.e., the minimum customer cost among those that have the least operational cost. The *dynamic Pickup and Delivery with Transfers* (`dPDPT`) problem is to find the optimal solution path.

3.2.2 Actions

It is important to note that, throughout this chapter, we follow the convention that an action is completed by vehicle r_a at a location n_i just before r_a departs from n_i. Since r_a can have multiple tasks to perform at n_i according to the static plan, this convention intuitively means that we make no assumptions about the order in which a vehicle performs its tasks. In any case, the action will have concluded by the time r_a is ready to depart from n_i.

Consider a path p with operational and customer costs O_p and C_p, respectively. Further, assume that the last action in p results in the package being onboard vehicle r_a at location n_i. Let p' denote the path resulting upon performing an additional action E on p. In the following, we detail each possible action E, and the costs of the resulting path p', denoted as $O_{p'}$, $C_{p'}$, which may depend on the current path costs O_p, C_p.

Pickup

The pickup action involves a single vehicle, r_a, and appears once as the first action in a solution path. Hence, initially the package is at the pickup location n_s of the request, p is empty, and $O_p = C_p = 0$.

We distinguish two cases for this action. First, assume that n_s is included in the vehicle's route, and let A_s^a, D_s^a denote the arrival and departure times of r_a at n_s according to the static plan. In this case, the pickup action is denoted as E_s^a. No modification in r_a's route is necessary, and thus there is zero additional operational cost for executing E_s^a. The customer cost for the resulting path p' becomes equal to the scheduled (according to the static plan) departure time D_s^a from n_s; without loss of generality, we make the assumption that the request arrives at time 0. Therefore,

$$\left.\begin{array}{l} O_{p'} = 0 \\ C_{p'} = D_s^a \end{array}\right\} \text{ for } p' = E_s^a. \tag{3.2a}$$

In the second case, the pickup location n_s is not in the r_a route. Let n_i be a location in the r_a route that is *sufficiently close* to n_s; then, r_a must take a detour from n_i to n_s. A location is sufficiently close to n_s if the detour is short, i.e., its duration, denoted as T_{si}^a, is below some threshold (a system parameter). Hence, it is possible that a sufficiently close location does not exists for route r_a; clearly, if no such location exists for any route, then the request is unsatisfiable. When such a n_i exists, the pickup action is denoted as E_{si}^a. Figure 3.1(a) shows a pickup action with detour. The solid line in the figure denotes the vehicle route r_a and the dashed line denotes the detour performed by r_a from n_i to n_s to pickup the package. The operational cost of a pickup action with detour is equal to the delay T_{si}^a due to the detour. The customer cost of p' is the scheduled departure time from n_i incremented by the delay. Therefore,

$$\left.\begin{array}{l} O_{p'} = T_{si}^a \\ C_{p'} = D_i^a + T_{si}^a \end{array}\right\} \text{ for } p' = E_{si}^a. \tag{3.2b}$$

Delivery

The delivery action involves a single vehicle, r_a, and appears once as the last action in a solution path. Similar to pickup, two cases exist for this action. In the first case, n_e appears in the route r_a, and delivery is denoted as E_e^a. The costs for path p' are shown in Equation 3.3a. In the second case, a detour of length T_{ie}^a at location n_i is required, and delivery is denoted as E_{ie}^a. Figure 3.1(b) presents an E_{ie}^a delivery action with detour. The costs for p' are shown in Equation 3.3b, where we make the assumption that it takes $T_{ie}^a/2$ time to travel from n_i to n_e.

$$\left.\begin{array}{l} O_{p'} = O_p \\ C_{p'} = C_p \end{array}\right\} \text{ for } p' = pE_e^a. \tag{3.3a}$$

$$\left.\begin{array}{l} O_{p'} = O_p + T^a_{ie} \\ C_{p'} = C_p + T^a_{ie}/2 \end{array}\right\} \text{ for } p' = pE^a_{ie}. \tag{3.3b}$$

Transport

The transport action involves a single vehicle, r_a, and corresponds to the carrying of the package by a vehicle between two successive locations on its route. Figure 3.1(c) illustrates such a transportation action from location n_i to n_j onboard vehicle r_a. As assumed, path p results in the package being onboard r_a at location n_i. The transport action, denoted as E^a_{ij}, has zero operational cost, as the vehicle is scheduled to move from n_i to n_j anyway. The customer cost is incremented by the time required by vehicle r_a to travel from n_i to n_j and finish its tasks at n_j. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p \\ C_{p'} = C_p + D^a_j - D^a_i \end{array}\right\} \text{ for } p' = pE^a_{ij}. \tag{3.4}$$

Transfer without detours

The transfer without detours action, denoted as E^{ab}_i, involves two vehicles, r_a and r_b, and corresponds to the transfer of the package from r_a to r_b at a common location n_i, e.g., a depot, drop-off/pickup point, etc. For example, Figure 3.1(d) shows such a transfer action via the common location n_i. Let A^b_i, D^b_i be the arrival and departure times of vehicle r_b at location n_i. We distinguish three cases.

In the first, the last action in path p concludes after vehicle r_b arrives and before it departs from n_i, i.e., $A^b_i \leq C_p \leq D^b_i$. Since there is no delay in the schedule of vehicles, the action's operational cost is zero, while the customer cost of the resulting path p' becomes equal to the scheduled departure time of r_b from n_i. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p \\ C_{p'} = D^b_i \end{array}\right\} \text{ for } p' = pE^{ab}_i, \text{ if } A^b_i \leq C_p \leq D^b_i. \tag{3.5a}$$

In the second case, the last action in path p concludes before vehicle r_b arrives at n_i, i.e., $C_p < A^b_i$. For the transfer to proceed, vehicle r_a must wait at n_i until r_b arrives. The operational cost is incremented by the delay, which is equal to $A^b_i - C_p$. On the other hand, the customer cost becomes equal to the scheduled departure time of r_b from n_i. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p + A^b_i - C_p \\ C_{p'} = D^b_i \end{array}\right\} \text{ for } p' = pE^{ab}_i, \text{ if } C_p < A^b_i. \tag{3.5b}$$

In the third case, the last action in p concludes after vehicle r_b is scheduled to depart from n_i, i.e., $C_p > D^b_i$. This implies that r_b must wait at n_i until the package is ready for transfer. The delay is equal to $C_p - D^b_i$, which contributes to the operational cost. The customer cost becomes equal to the delayed departure of r_b from n_i, which coincides with C_p. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p + C_p - D^b_i \\ C_{p'} = C_p \end{array}\right\} \text{ for } p' = pE^{ab}_i, \text{ if } C_p > D^b_i. \tag{3.5c}$$

Transfer with detours

Consider distinct locations n_i on r_a and n_j on r_b. Assume that short detours from n_i and n_j are possible, i.e., the detour durations are below some threshold, and that

they have a common rendezvous point. The transfer with detours action, denoted as E_{ij}^{ab}, involves the two vehicles, r_a and r_b, and corresponds to the transportation of the package on vehicle r_a via the n_i detour to the rendezvous location, its transfer to vehicle r_b, which has taken the n_j detour, and finally its transportation to n_j. Figure 3.1(e) illustrates a transfer action between vehicles r_a and r_b via a detour to their common rendezvous point n_c. Notice the difference with Figure 3.1(d) where the transfer action occurs without a detour. To keep the notation simple, we make the following assumptions: (1) the n_i detour travel time of r_a is equal to that of the n_j detour of r_b, denoted as T_{ij}^{ab}; and (2) it takes $T_{ij}^{ab}/2$ time for both r_a and r_b to reach the rendezvous location.

Similar to transferring without detours, we distinguish three cases. In the first, the package is available for transfer at the rendezvous location, at time $C_p + T_{ij}^{ab}/2$, after the earliest possible and before the latest possible arrival of r_b, i.e., $A_j^b + T_{ij}^{ab}/2 \leq C_p + T_{ij}^{ab}/2 \leq D_j^b + T_{ij}^{ab}/2$. Both vehicles incur a delay in their schedule by T_{ij}^{ab}. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p + 2 \cdot T_{ij}^{ab} \\ C_{p'} = D_j^b + T_{ij}^{ab} \end{array}\right\} \text{ for } p' = pE_{ij}^{ab}, \text{ if } A_j^b \leq C_p \leq D_j^b. \tag{3.6a}$$

In the second case, the package is available for transfer before the earliest possible arrival of r_b at the rendezvous location, i.e., $C_p + T_{ij}^{ab}/2 < A_j^b + T_{ij}^{ab}/2$. Vehicle r_a must wait for $A_j^b - C_p$ time. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p + A_j^b - C_p + 2 \cdot T_{ij}^{ab} \\ C_{p'} = D_j^b + T_{ij}^{ab} \end{array}\right\} \text{ for } p' = pE_{ij}^{ab}, \text{ if } C_p < A_j^b. \tag{3.6b}$$

Finally, in the third case, the package is available for transfer after the latest possible arrival of r_b at the rendezvous location, i.e., $C_p + T_{ij}^{ab}/2 > D_j^b + T_{ij}^{ab}/2$. Vehicle r_b must wait for $C_p - D_j^b$ time. Therefore,

$$\left.\begin{array}{l} O_{p'} = O_p + C_p - D_j^b + 2 \cdot T_{ij}^{ab} \\ C_{p'} = C_p + T_{ij}^{ab} \end{array}\right\} \text{ for } p' = pE_{ij}^{ab}, \text{ if } C_p > D_j^b. \tag{3.6c}$$

3.3 The Dynamic Plan Graph

We construct a weighted directed graph, termed *dynamic plan graph* and denoted by G_R, in a way that a sequence of actions corresponds to a simple path on this graph. A vertex of the graph corresponds to a spatial location. In particular, a vertex V_i^a represents the spatial location n_i of route r_a. Additionally, there exist two special vertices, V_s and V_e, which represent the request's pickup and delivery, respectively, locations. Therefore, five types of edges exist:

(1) A *pickup edge* E_{si}^a connects V_s to V_i^a, and represents a pickup action by vehicle r_a with a detour at n_i. Edge E_{ss}^a from V_s to V_s^a (two distinct vertices that correspond to the same spatial location n_s) represents the case of pickup with no detour.

(2) A *delivery edge* E_{ie}^a connects V_i^a to V_e, and represents a delivery action by vehicle r_a with a detour at n_i. Edge E_{ee}^a from V_e to V_e^a represents the case of pickup with no detour.

Pickup	Delivery	Transport
$w(E^a_{si}) = \langle T^a_{si}, D^a_i + T^a_{si} \rangle$ (3.7)	$w(E^a_{ie}) = \langle T^a_{ie}, T^a_{ie}/2 \rangle$ (3.8)	$w(E^a_{ij}) = \langle 0, D^a_j - D^a_i \rangle$ (3.9)

Transfer
$w(E^{ab}_{ij}) = \begin{cases} \langle 2 \cdot T^{ab}_{ij}, D^b_j - C_p + T^{ab}_{ij} \rangle, & \text{if } A^b_j \leq C_p \leq D^b_j \\ \langle A^b_j - C_p + 2 \cdot T^{ab}_{ij}, D^b_j - C_p + T^{ab}_{ij} \rangle, & \text{if } C_p < A^b_j \\ \langle C_p - D^b_j + 2 \cdot T^{ab}_{ij}, T^{ab}_{ij} \rangle, & \text{if } C_p > D^b_j. \end{cases}$ (3.10)

Table 3.1: *Edge weights of the dynamic plan graph.*

(3) A *transport* edge E^a_{ij} connects V^a_i to V^a_j, and represents a transport action by r_a from n_i to its following location n_j on the route.

(4) A *transfer without detours* edge E^{ab}_i connects V^a_i to V^b_i, and represents a transfer from r_a to r_b at common location n_i.

(5) A *transfer with detours* edge E^{ab}_{ij} connects V^a_i to V^b_j, and represents a transfer from r_a to r_b at a rendezvous location via detours at n_i and n_j.

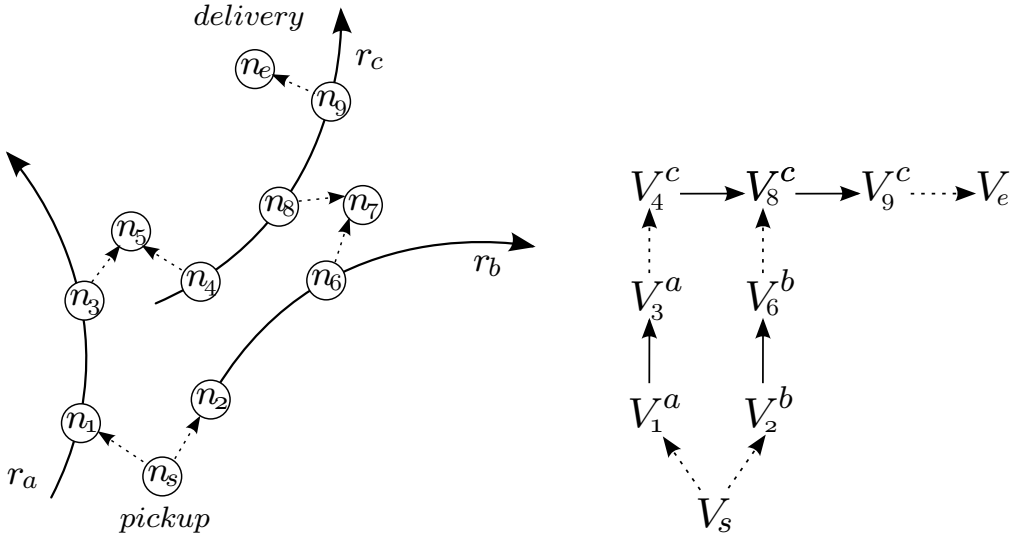
Based on the above definitions, a simple path on the graph is a sequence of distinct vertices that translates into a sequence of actions. Further, a solution for the request is a path that starts from V_s and ends in V_e.

The final issue that remains is to define the weights $\mathcal{W}$ of the edges. We assign edge E a pair of weights $w(E) = \langle w_O(E), w_C(E) \rangle$, so that $w_O(E)$ (resp. $w_C(E)$) corresponds to the operational (resp. customer) cost of performing the action associated with the edge E. Recall from Section 3.2.2 that the costs of the last action in a sequence of actions depends on the total costs incurred by all previous actions. Consequently, the weights of an edge E from V to V' are *dynamic*, since they depend on the costs of the path p that lead to V. Assuming O_p and C_p are the costs of p, and $O_{p'}$ and $C_{p'}$ those of path $p' = pE$ upon executing E, we have that $w(E) = \langle O_{p'} - O_p, C_{p'} - C_p \rangle$. Table 3.1 summarizes the formulas for the weights of all edge types; note that the weights for actions with no detours are obtained by setting the corresponding T value to zero. In the formulas, A^b_j, D^a_i, D^a_j and D^b_j have fixed values determined by the static plan. On the other hand, C_p depends on the path p that leads to V^a_i.

Clearly, a path from V_s to V_e that has the lowest combined cost according to Equation 3.1 is an optimal solution.

Proposition 3.1. *Let R be a collection of vehicles routes and (n_s, n_e) be a* `dPDPT` *request over R. The solution to the request is the shortest path from vertex V_s to V_e on the dynamic plan graph G_R with respect to cost() of Equation 3.1.*

Example 3.1. *Figure 3.2(a) pictures a collection of vehicle routes $R = \{r_a(n_1, n_3), r_b(n_2, n_6), r_c(n_4, n_8, n_9)\}$, and the pickup n_s and the delivery location n_e of a* `dPDPT`



(a) Vehicle routes R

(b) Dynamic plan graph G_R

Figure 3.2: *A collection of vehicles routes R, a* `dPDPT` *request (n_s, n_e) over R, and the dynamic plan graph G_R. The solid lines denote the vehicle routes/transport edges while the dashed lines denote the pickup, delivery and transfer with detour actions/edges.*

request. Locations n_1 on route r_a and n_2 on r_b are sufficiently close to location n_s and thus, pickup actions E^a_{s1} and E^b_{s2} are possible. Similar, the E^c_{9e} delivery action is possible at location n_9 on route r_c. Finally, we also identify two transfer actions, E^{ac}_{34} and E^{bc}_{68}, as locations n_3, n_4 and n_6, n_8 have common rendezvous points n_5 and n_7, respectively.

To satisfy the `dPDPT` *request (n_s, n_e) we define the dynamic plan graph G_R in Figure 3.2(b) containing vertices $V_s, V^a_1, V^b_2, \ldots, V_e$. Notice that the graph does not include any vertices for the rendezvous points n_5 and n_7. Dynamic plan graph G_R contains two paths from V_s to V_e which means that there two different ways to satisfy the* `dPDPT` *request: $p_1(V_s, V^a_1, V^a_3, V^c_4, V^c_8, V^c_9, V_e)$ and $p_2(V_s, V^b_2, V^b_6, V^c_8, V^c_9, V_e)$. Next, assume, for simplicity, that the detour cost is equal to T for all possible actions. Further, consider paths $p'_1(V_s, V^a_1, V^a_3)$ and $p'_2(V_s, V^b_2, V^b_6)$, i.e., just before the transfer of the package takes place, and assume that $A^c_4 < C_{p'_1} < D^c_4$ and $C_{p'_2} > D^c_8$ hold. Note, that the operational cost of the two paths is exactly the same, i.e., $O_{p'_1} = O_{p'_2} = T$, coming from the pickup of the package at n_s. Now, according to Equation 3.10 and after the transfers E^{ac}_{34} and E^{bc}_{68} take place, we get path $p''_1 = p'_1 E^{ac}_{34}$ and $p''_2 = p'_2 E^{bc}_{68}$ with $O_{p''_1} = 3 \cdot T < O_{p''_2} = 3 \cdot T + C_{p'_2} - D^c_8$, and therefore, $cost(p''_1) < cost(p''_2)$. Finally, since no other transfer incurs in order to delivery the package, this holds also for paths p_1 and p_2, i.e., $cost(p_1) < cost(p_2)$, and thus, the solution to the* `dPDPT` *request (n_s, n_e) is path $p_1(V_s, V^a_1, V^a_3, V^c_4, V^c_8, V^c_9, V_e)$ with $O_{p_1} = 4 \cdot T$ and $C_{p_1} = D^c_9 + \frac{3 \cdot T}{2}$.*

3.4 The SP Algorithm

According to Proposition 3.1, the next step is to devise an algorithm that computes the two-criterion shortest path w.r.t. *cost*() on the dynamic plan graph. Unfortunately, the dynamic weights of the edges in the graph violate the *subpath optimality*; that is, the lowest combined cost path from V_s to V_e that passes through some vertex V may not contain the lowest combined cost path from V_s to V. The following theorem supports this claim.

Theorem 3.1. *The dynamic plan graph does not have subpath optimality for any monotonic combination of the operational and customer costs.*

Proof. Let p, q be two paths from V_s to V_i^a, with costs O_p, C_p and O_q, C_q, respectively, such that $O_p < O_q$ and $C_p < C_q$, which implies that for any monotonic combination of the operational and customer costs, p has lower combined cost than q. Let p' and q' be the paths resulting after traversing a transfer without detours edge E_i^{ab}.

Assume that $C_p < C_q < A_i^b$, so that the second case of Equation 3.10 applies for the weight $w(E_i^{ab})$ (setting $T_{ij}^{ab} = 0$). Then, $O_{p'} = O_p + A_i^b - C_p$, $O_{q'} = O_q + A_i^b - C_q$, and $C_{p'} = C_{q'} = D_i^b$. Assuming that $O_p - C_p > O_q - C_q$, we obtain that $O_{q'} < O_{p'}$. Therefore, for any monotonic combination that considers both costs, q''s combined cost is lower that that of p''s. □

As a result, efficient algorithms based on the subpath optimality, e.g., Dijkstra and Bellman-Ford cannot be employed to compute the shortest path on the dynamic plan graph. In contrast, an exhaustive enumeration of all paths from V_s to V_e is necessary, and for this purpose, we introduce a label-setting algorithm called SP. Note that, in the sequel, we only discuss the case when actions occur with detours as it is more general.

The SP algorithm has the following key features. First, similar to all algorithms for multi-criteria shortest path, it may visit a vertex V_i^a more than once following multiple paths from the initial vertex V_s. For each of these paths p, the algorithm defines a *label* in the form of $\langle V_i^a, p, O_p, C_p \rangle$, where O_p is the operational and C_p the customer cost of path p as introduced in Section 3.3. Second, at each iteration, SP selects the label $\langle V_i^a, p, O_p, C_p \rangle$ with the currently selected path p having the lowest combined cost $cost(p)$, and expands the search considering the outgoing edges from V_i^a on the dynamic plan graph G_R. If vertex V_i^a has an outgoing delivery edge E_{ie}^a, SP identifies a path from initial vertex V_s to final V_e called *candidate* solution. The candidate solution is an *upper bound* to the final solution and it is progressively improved until it becomes equal to the final. The role of a candidate solution is twofold; it triggers the termination condition and prunes the search space. Finally, the algorithm terminates the search when $cost(p)$ of the path p with the lowest combined cost, is equal to or higher than $cost(p_{cand})$ of the current candidate solution p_{cand} which means that neither p or any other path at future iterations can be better than current p_{cand}.

Figure 3.3 illustrates the pseudocode of the SP algorithm. SP takes as inputs: a dPDPT request (n_s, n_e) and the dynamic plan graph G_R of a collection of vehicle routes R. It returns the shortest path from V_s to V_e on G_R with respect to *cost*(). The algorithm uses the following data structures: (1) a priority queue $\mathcal{Q}$, (2) a path p_{cand}, and (3) a list $\mathcal{T}$. The priority queue $\mathcal{Q}$ is used to perform the search by storing

Algorithm SP
Input: dPDPT request (n_s, n_e), dynamic plan graph G_R
Output: shortest path from V_s to V_e w.r.t. $cost()$
Parameters:
priority queue $\mathcal{Q}$: the search queue sorted by $cost()$ in ascending order
path p_{cand}: the candidate solution to the dPDPT request
list $\mathcal{T}$: the target list
Method:

1. **construct** pickup edges E^a_{si};
2. **construct** delivery edges E^a_{ie};
3. **for** each pickup edge $E^a_{si}(V_s, V^a_i)$ in G_R **do**
   push label $\langle V^a_i, E^a_{si}, T^a_{si}, D^a_i + T^a_{si}\rangle$ to $\mathcal{Q}$;
4. **for** each delivery edge $E^a_{ie}(V^a_i, V_e)$ in G_R **do**
   insert $\langle V^a_i, T^a_{ie}, T^a_{ie}/2\rangle$ in $\mathcal{T}$;
5. **while** $\mathcal{Q}$ is not empty **do**
6. **pop** label $\langle V^a_i, p, O_p, C_p\rangle$ from $\mathcal{Q}$;
7. **if** $cost(p) \geq cost(p_{cand})$ **then return** p_{cand};
8. ImproveCandidateSolution$(p_{cand}, \mathcal{T}, \langle V^a_i, p, O_p, C_p\rangle)$;
9. **for** each outgoing transport E^a_{ij} **or** transfer edge E^{ab}_{ij} in G_R **do**
10. **extend** path p and **create** p';
11. **compute** $O_{p'}$ and $C_{p'}$;
12. **if** $cost(p') < cost(p_{cand})$ **then**
    ignore path p';
13. **else**
14. **push** label $\langle V', p', O_{p'}, C_{p'}\rangle$ to $\mathcal{Q}$ **where** V' is the last vertex in p';
15. **end if**
16. **end for**
17. **end while**
18. **return** p_{cand} **if exists**, **otherwise null**;

Figure 3.3: *The* SP *algorithm.*

every label $\langle V^a_i, p, O_p, C_p\rangle$ to be checked, sorted by $cost(p)$ in ascending order. The target list $\mathcal{T}$ contains entries of the form $\langle V^a_i, T^a_{ie}, T^a_{ie}/2\rangle$, where V^a_i is a vertex of the dynamic plan graph involved in a delivery edge E^a_{ie}, List $\mathcal{T}$ is used to construct or improve the candidate solution p_{cand}.

The execution of the SP algorithm involves two phases: the *initialization* and the *core* phase. In the initialization phase (Lines 1–4), SP first creates the pickup E^a_{si} and delivery edges E^a_{ie} on the dynamic plan graph G_R. For this purpose it identifies each location n_i on every vehicle route r_a that is sufficiently close to pickup location n_s (resp. delivery n_e), i.e., the duration T^a_{si} (resp. T^a_{ie}) of the detour from n_s to n_i (resp. n_i to n_e) is below some threshold (a system parameter). Then, the algorithm initializes the priority queue $\mathcal{Q}$ adding every vertex V^a_i involved in a pickup edge E^a_{si} on G_R and constructs the target list $\mathcal{T}$. In the core phase (Lines 5–17), the algorithm performs the search. It proceeds iteratively popping, first, the label $\langle V^a_i, p, O_p, C_p\rangle$ from $\mathcal{Q}$ on Line 6. Path p has the lowest $cost(p)$ value compared to all others paths in $\mathcal{Q}$. Next, SP checks the termination condition (Line 7). If the check succeeds, i.e., $cost(p) \geq cost(p_{cand})$, then current candidate p_{cand} is returned as the final solution.

If the termination condition fails, the algorithm first tries to improve candidate solution p_{cand} calling the ImproveCandidateSolution$(p_{cand}, \mathcal{T}, \langle V^a_i, p, O_p, C_p\rangle)$ function on Line 8. The function checks if the target list $\mathcal{T}$ contains an entry $\langle V^a_i, T^a_{ie}, T^a_{ie}/2\rangle$ for the vertex V^a_i of the current label and constructs the path pE^a_{ie} from V_s to V_e. If $cost(pE^a_{ie}) < cost(p_{cand})$ then a new improved candidate solution is identified and thus, $p_{cand} = pE^a_{ie}$. Finally, SP expands the search considering all outgoing transport and transfer edges from V^a_i on G_R (Lines 9–16). Specifically, the path p of the current

label is extended to $p' = pE^a_{ij}$ (transport edge) or to $p' = pE^{ab}_{ij}$ (transfer edge), and the operational $O_{p'}$ and the customer cost $C_{p'}$ of the new path p' are computed according to Equations 3.9 and 3.10. Then, on Line 12, the algorithm determines whether p' is a "promising" path and thus, it must be extended at a future iteration, or it must be discarded. The algorithm discards path p' if $cost(p') \geq cost(p_{cand})$ which means that p' cannot produce a better solution than current p_{cand}. Otherwise, p' is a "promising" path, and SP inserts label $\langle V', p', O_{p'}, C_{p'}\rangle$ in $\mathcal{Q}$ where V' is the last vertex in path p'.

Example 3.2. *We illustrate the* SP *algorithm using Figure 3.2. To carry out the search we make the following assumptions, similar to Example 3.1. The detour cost is equal to T for all edges. For the paths $p'_1(V_s, V^a_1, V^a_3)$ and $p'_2(V_s, V^b_2, V^b_6)$, i.e., just before the transfer of the package takes place, $A^c_4 < C_{p'_1} < D^c_4$ and $C_{p'_2} > D^c_8$ hold. Finally, we also assume that $D^a_1 < D^a_3 < D^b_2 < D^b_6$.*

First, SP *initializes the priority queue $\mathcal{Q} = \{\langle V^a_1,\ (V_s, V^a_1), T, D^a_1 + T\rangle, \langle V^b_2,\ (V_s, V^b_2), T, D^b_2 + T\rangle\}$ and constructs the target list $\mathcal{T} = \{\langle V^c_9, T, T/2\rangle\}$. Note that the leftmost label in $\mathcal{Q}$ always contains the path with the lowest cost() value. At the first iteration, the algorithm pops label $\langle V^a_1,\ (V_s, V^a_1), T, D^a_1 + T\rangle$, considers transport edge E^a_{13}, and pushes $\langle V^a_3, p'_1(V_s, V^a_1, V^a_3), T, D^a_3 + T\rangle$ to $\mathcal{Q}$. Next, at the second iteration,* SP *pops label $\langle V^a_3, p'_1(V_s, V^a_1, V^a_3), T, D^a_3 + T\rangle$ from $\mathcal{Q}$, considers the transfer edge E^{ac}_{34}, and pushes $\langle V^c_4, p''_1(V_s, V^a_1, V^a_3, V^c_4), 3 \cdot T, D^c_4 + T\rangle$ to $\mathcal{Q}$ (remember $A^c_4 < C_{p'_1} < D^c_4$). The next two iterations are similar, and thus, after the fourth iteration we have:*

$$\mathcal{Q} = \{\langle V^c_4, p''_1(V_s, V^a_1, V^a_3, V^c_4), 3 \cdot T, D^c_4 + T\rangle, \quad \langle V^c_8, p''_2(V_s, V^b_2, V^b_6, V^c_8), 3 \cdot T + C_{p'_2} - D^c_8, C_{p'_2} + T\rangle\} \qquad \text{and } p_{cand} = \textbf{\textit{null}}$$

Now, at the next two iterations, SP *expands path p''_1 considering transport edges E^c_{48} and E^c_{89} as $O_{p''_1} < O_{p''_2}$. Therefore, at the seventh iteration, the algorithm pops label $\langle V^c_9, (V_s, V^a_1, V^a_3, V^c_4, V^c_8, V^c_9), 3 \cdot T, D^c_9 + T\rangle$ from $\mathcal{Q}$. Since the target list $\mathcal{T}$ contains an entry for vertex V^c_9,* SP *identifies candidate solution $p_{cand} = p_1(V_s, V^a_1, V^a_3, V^c_4, V^c_8, V^c_9, V_e)$ with $O_{p_1} = 4 \cdot T$ and $C_{p_1} = D^c_9 + \frac{3 \cdot T}{2}$. Finally, assuming without loss of generality that $D^b_6 > D^c_8$ also holds and therefore, $C_{p'_2} - D^c_8 = D^b_6 + T - D^c_8 > T$, at the eighth iteration, the algorithm pops $\langle V^c_8, p''_2(V_s, V^b_2, V^b_6, V^c_8), 3 \cdot T + C_{p'_2} - D^c_8, C_{p'_2} + T\rangle$ and terminates the search because $O_{p''_2} = 3 \cdot T + C_{p'_2} - D^c_8 > 4 \cdot T = O_{p_1}$ and thus, $cost(p''_2) > cost(p_1)$. The solution to the* dPDPT *request (n_s, n_e) is $p_1(V_s, V^a_1, V^a_3, V^c_4, V^c_8, V^c_9, V_e)$.*

3.5 The Modified Dynamic Plan Graph

According to Theorem 3.1, it is possible that a path q with high costs leads to a better solution than a path p with lower costs; hence q cannot be eliminated. However, there exist cases when q can be safely pruned as it definitely cannot contribute a better solution than p. A simple example of such a case is when, for each vehicle, q incurs a delay greater than what p incurs. Clearly, operational and customer costs are higher for q. Additionally, any path that extends q in some way is worse than the path obtained by extending p in that same way.

Motivated by this observation, we seek a breakdown of the operational cost in a manner that allows us to eliminate paths, circumventing thus Theorem 3.1. Consider a path p that ends in vertex V^a_i. We define two new cost metrics. The *current*

Pickup	Delivery	Transport
$w(E^a_{si}) = \langle T^a_{si}, 0\rangle$ (3.11)	$w(E^a_{ie}) = \langle T^a_{ie}/2, T^a_{ie}/2\rangle$ (3.12)	$w(E^a_{ij}) = \langle 0, 0\rangle$ (3.13)

Transfer
$w(E^{ab}_{ij}) = \begin{cases} \langle T^{ab}_{ij} - O^*_p, T^{ab}_{ij} + O^*_p\rangle, & \text{if } A^b_j \le D^a_i + O^*_p \le D^b_j \\ \langle T^{ab}_{ij} - O^*_p, T^{ab}_{ij} + A^b_j - D^a_i\rangle, & \text{if } D^a_i + O^*_p < A^b_j \\ \langle T^{ab}_{ij} + D^a_i - D^b_j, T^{ab}_{ij} + O^*_p\rangle, & \text{if } D^a_i + O^*_p > D^b_j \end{cases}$ (3.14)

Table 3.2: *Edge weights of the modified dynamic plan graph.*

operational cost O^*_p is the delay incurred by the vehicle, r_a in this example, currently carrying the package. The *residual operational cost* O^R_p is the delay incurred by all other vehicles.

Obviously, for any path, the operational cost O_p is the sum of these costs, i.e., $O_p = O^R_p + O^*_p$. Furthermore, observe that, according to Equations 3.2a – 3.6c, the customer cost C_p is determined only by the delay in the schedule of the current vehicle, i.e., $C_p = D^a_i + O^*_p$. Therefore, any combined cost, including the one used throughout this work (Equation 3.1), can be rewritten using the current and residual operational costs.

Consider the *modified dynamic plan graph* denoted by G^*_R that has the same vertices and edges, but modified weight, compared to the dynamic plan graph. The modified weight of an edge E, is defined as the pair $w'(E) = \langle w_*(E), w_R(E)\rangle$, so that $w_*(E)$ (resp. $w_R(E)$) is the current (resp. residual) operational cost of action E. The formulas for the modified weights for all types of edges are summarized in Table 3.2; note that the weights for actions with no detours are obtained by setting the corresponding T value to zero.

According to the next theorem, the graph with these modified weights exhibits subpath optimality. Therefore, the `dPDPT` problem can be solved using efficient algorithms, which we discuss in the following section.

Theorem 3.2. *The modified dynamic plan graph has subpath optimality for any monotonic combination of the operational and customer costs.*

Proof. First, note that any monotonic combination of the operational and customer costs can be written as a monotonic combination of the current and residual operational costs. Let p, q be two paths from V_s to V^a_i, with current and residual operational costs O^*_p, O^R_p and O^*_q, O^R_q, respectively, such that $O^*_p < O^*_q$ and $O^R_p < O^R_q$. This implies that for any monotonic combination of these two costs, p has lower combined cost than q.

Assume p' and q' are the paths after traversing a transfer without detours edge E^{ab}_i; the transfer with detours edge proof is similar, while the proof for all other edges is trivial. We must prove that, in all possible cases, denoted as Cases I–III, for p, q, the statement $O^*_{p'} < O^*_{q'}$ and $O^R_{p'} < O^R_{q'}$ is true; note that to obtain the

equations for the transfer without detours edge E_i^{ab} from Equation 3.14, set $T_{ij}^{ab} = 0$ and $j = i$.

Assume that Case I applies for p; thus, the current and residual operational costs of p' become $O_{p'}^* = 0$, $O_{p'}^R = O_p^R + O_p^*$. If Case I applies for q, q''s costs become $O_{q'}^* = 0$, $O_{q'}^R = O_q^R + O_q^*$; clearly the statement is true. Case II cannot apply for q, since $D_i^a + O_q^r > D_i^a + O_p^r \geq A_i^b$. If Case III applies for q, q''s costs become $O_{q'}^* = O_q^* + D_i^a - D_i^b$, $O_{q'}^R = O_q^R + O_q^*$. The statement is true, because $O_q^r + D_i^a - D_i^b > 0$ in Case III.

Now, assume that Case II applies for p; thus, the costs of p' become $O_{p'}^* = 0$, $O_{p'}^R = O_p^R + A_i^b - D_i^a$. If Case I applies for q, q''s costs become $O_{q'}^* = 0$, $O_{q'}^R = O_q^R + O_q^*$. The statement is true, because $A_i^b \leq D_i^a + O_q^r$ in Case I. If Case II applies for q, q''s costs become $O_{q'}^* = 0$, $O_{q'}^R = O_q^R + A_i^b - D_i^a$, and the statement is clearly true. If Case III applies for q, q''s costs become $O_{q'}^* = O_q^* + D_i^a - D_i^b$, $O_{q'}^R = O_q^R + O_q^*$. Regarding the current operational cost, observe that $D_i^a + O_q^* > D_i^b$ in Case III, and thus $O_q^* > D_i^b - D_i^a \geq A_i^b - D_i^a$. Regarding the residual operational cost, observe that $D_i^a + O_q^* > D_i^b$ in Case III. Therefore, the statement is true.

Finally, assume that Case III applies for p; thus, the costs of p' become $O_{p'}^* = O_p^* + D_i^a - D_i^b$, $O_{p'}^R = O_p^R + O_p^*$. Only Case III can apply for q and so q''s costs become $O_{q'}^* = O_q^* + D_i^a - D_i^b$, $O_{q'}^R = O_q^R + O_q^*$; clearly, the statement is true. □

3.6 The SPM Algorithm

Following Theorem 3.2, we introduce the SPM algorithm that efficiently solves the dPDPT problem. Similar to SP, SPM is a label-setting algorithm that computes the solution to a dPDPT request identifying the shortest path from vertex V_s to V_e on the *modified* dynamic plan graph G_R^*, with respect to $cost()$ as defined in Equation 3.1.

The SPM algorithm has similar key features to SP but it considers some additional optimizations. Thus, for a path p from the initial vertex V_s to a vertex V_i^a found, it defines a *label* in the form of $\langle V_i^a, p, O_p^*, O_p^R \rangle$, where O_p^* is the current operational cost and O_p^R is the residual operational cost of path p as introduced in Section 3.5. Note that the operational O_p and the customer cost C_p are not included in the label since they can be computed at any time using O_p^* and O_p^R. However, in order to speedup the search, SPM does not consider the labels of all the paths to a vertex V_i^a, like SP does. It only stores the labels of the most "promising" paths. A path $p(V_s, \ldots, V_i^a)$ is less "promising" than another $q(V_s, \ldots, V_i^a)$ if extending p can never provide a better solution than extending q. Further, similar to SP, at each iteration, SPM selects the label $\langle V_i^a, p, O_p^*, O_p^R \rangle$ with the lowest combined cost $cost(p)$, identifies a candidate solution path p_{cand} if V_i^a has an outgoing delivery edge E_{ie}^a, and then, expands the search considering the outgoing edges from V_i^a on the modified dynamic plan graph G_R^*. During this expansion phase, for every newly created path $(V_s, \ldots, V')$, SPM determines whether p' is a "promising" path considering not only current candidate answer p_{cand} (like SP does), but also the most "promising" paths from V_s to V' constructed at previous iterations. This enables SPM to further prune the search space compared to SP. Finally, SPM terminates the search, similar to SP, when $cost(p)$ of the currently selected path p with the lowest combined cost, is equal to or higher than $cost(p_{cand})$ of the current candidate solution p_{cand} which means that neither p or any other path at future iterations can

be better than current p_{cand}.

Algorithm SPM
Input: dPDPT request (n_s, n_e), modified dynamic plan graph G_R^*
Output: shortest path from V_s to V_e w.r.t. $cost()$
Parameters:

priority queue $\mathcal{Q}$:	the search queue sorted by $cost()$ in ascending order
path p_{cand}:	the candidate solution to the dPDPT request
list $\mathcal{T}$:	the target list
set $\mathcal{P}[V_i^a]$:	the most "promising" paths to each vertex V_i^a contained in labels in $\mathcal{Q}$

Method:

1. **construct** pickup edges E_{si}^a;
2. **construct** delivery edges E_{ie}^a;
3. **for** each pickup edge $E_{si}^a(V_s, V_i^a)$ in G_R^* **do**
4. **push** label $\langle V_i^a, E_{si}^a, T_{si}^a, 0\rangle$ to $\mathcal{Q}$;
5. **insert** entry $\langle T_{si}^a, 0\rangle$ in $\mathcal{P}[V_i^a]$;
6. **end for**
7. **for** each delivery edge $E_{ie}^a(V_i^a, V_e)$ in G_R^* **do**
   insert $\langle V_i^a, T_{ie}^a/2, T_{ie}^a/2\rangle$ in $\mathcal{T}$;
8. **while** $\mathcal{Q}$ is not empty **do**
9. **pop** label $\langle V_i^a, p, O_p^*, O_p^R\rangle$ from $\mathcal{Q}$;
10. **if** $cost(p) \geq cost(p_{cand})$ **then return** p_{cand};
11. ImproveCandidateSolution$(p_{cand}, \mathcal{T}, \langle V_i^a, p, O_p^*, O_p^R\rangle)$;
12. **for** each outgoing transport E_{ij}^a **or** transfer edge E_{ij}^{ab} in G_R^* **do**
13. **extend** path p and **create** p';
14. **compute** $O_{p'}^*$ and $O_{p'}^R$;
15. **let** V' be the last vertex in p';
16. **if** exists entry $\langle O_q^*, O_q^R\rangle$ in $\mathcal{P}[V']$ **with** $O_{p'}^* \geq Q_q^*$ **and** $O_{p'}^R \geq Q_q^R$ **then**
    ignore path p';
17. **else if** $cost(p') < cost(p_{cand})$ **then**
    ignore path p';
18. **else**
19. **push** label $\langle V', p', O_{p'}*, O_{p'}^R\rangle$ to $\mathcal{Q}$;
20. **insert** entry $\langle O_{p'}^*, O_{p'}^R\rangle$ in $\mathcal{P}[V']$;
21. **delete** each entry $\langle O_q^*, O_q^R\rangle$ from $\mathcal{P}[V']$ **where** $O_{p'}^* \leq Q_q^*$ **and** $O_{p'}^R \leq Q_q^R$ and its corresponding labels from $\mathcal{Q}$;
22. **end if**
23. **end for**
24. **end while**
25. **return null**;

Figure 3.4: *The* SPM *algorithm.*

Figure 3.4 illustrates the pseudocode of the SPM algorithm. SPM takes as inputs: a dPDPT request (n_s, n_e) and the modified dynamic plan graph G_R^* of a collection of vehicle routes R. It returns the shortest path from V_s to V_e on G_R^* with respect to $cost()$. Similar to SP, the algorithm uses (1) a priority queue $\mathcal{Q}$, (2) a path p_{cand}, and (3) a target list $\mathcal{T}$, but also (4) a set $\mathcal{P}[V_i^a]$ for each vertex V_i^a contained in the labels inserted in $\mathcal{Q}$, The set $\mathcal{P}[V_i^a]$ contains entries in the form of $\langle O_p^*, O_p^R\rangle$ for each "promising" path p from V_s to V_i^a found so far. The entries in $\mathcal{P}[V_i^a]$ are used to prune the non "promising" paths to vertex V_i^a, constructed during the search.

The execution of the SPM algorithm is similar to SP involving an *initialization* (Lines 1–7) and a *core* phase (Lines 8–24). Since, SPM only differs from SP in the expansion of the search during the core phase (Lines 12–23), we detail this process. Consider a path $p'(V_s, \ldots, V_i^a, V')$ created after extending path p of the current label $\langle V_i^a, p, O_p^*, O_p^R\rangle$. SPM discards path p' if either of the following holds:

(1) $\mathcal{P}[V']$ contains an entry $\langle O_q^*, O_q^R\rangle$ such that $O_{p'}^* \geq Q_q^*$ and $O_{p'}^R \geq Q_q^R$ (Line 16)

(2) $cost(p') \geq cost(p_{cand})$ (Line 17)

Following Case (1) p' cannot contribute a better solution than q, whereas with Case (2) p' cannot produce a better solution than current p_{cand}. Note that the SP algorithm considers only Case (2). If neither Case (1) or (2) holds, p' is a "promising" path, and SPM inserts label $\langle V', p', O_{p'}^*, O_{p'}^R\rangle$ in $\mathcal{Q}$ and entry $\langle O_{p'}^*, O_{p'}^R\rangle$ in $\mathcal{P}[V']$. In addition, on Line 21, the algorithm updates $\mathcal{P}[V']$ removing every entry $\langle O_q^*, O_q^R\rangle$ with $O_{p'}^* \leq Q_q^*$ and $O_{p'}^R \leq Q_q^R$, and the corresponding labels in $\mathcal{Q}$, since such path q will never contribute a better solution than p'.

Example 3.3. *We illustrate the* SPM *algorithm and discuss its differences with* SP *using Example 3.2. Recall that we make the following assumptions: (1) the detour cost is equal to T for all edges, (2) for paths* $p'_1(V_s, V_1^a, V_3^a)$ *and* $p'_2(V_s, V_2^b, V_6^b)$*, i.e., just before the transfer of the package takes place,* $A_4^c < C_{p'_1} < D_4^c$ *and* $C_{p'_2} > D_8^c$ *hold, and (3)* $D_1^a < D_3^a < D_2^b < D_6^b$*. Finally, also remember that the leftmost label in the priority queue* $\mathcal{Q}$ *always contains the path with the lowest cost() value.*

First, SPM *initializes the priority queue* $\mathcal{Q} = \{\langle V_1^a, (V_s, V_1^a), T, 0\rangle, \langle V_2^b, (V_s, V_2^b), T, 0\rangle\}$ *and constructs the target list* $\mathcal{T} = \{\langle V_9^c, T/2, T/2\rangle\}$*. In addition, we have* $\mathcal{P}[V_1^a] = \{\langle T, 0\rangle\}$ *and* $\mathcal{P}[V_2^b] = \{\langle T, 0\rangle\}$*. Note that path* (V_s, V_1^a) *has lowest cost() compared to* (V_s, V_2^b) *because* $D_1^a < D_2^b$*. At the first four iterations,* SPM *proceeds similar to* SP *and thus, after the fourth iteration and considering edges* E_{13}^a*,* E_{34}^{ac}*,* E_{26}^b *and* E_{68}^{bc} *we have:*

$$\mathcal{Q} = \{\langle V_4^c, p''_1(V_s, V_1^a, V_3^a, V_4^c), T, 2\cdot T\rangle, \langle V_8^c, p''_2(V_s, V_2^b, V_6^b, V_8^c), T + C_{p'_2} - D_8^c, 2\cdot T\rangle\}$$
$$\mathcal{P}[V_1^a] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_2^b] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_3^a] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_4^c] = \{\langle T, 2\cdot T\rangle\}$$
$$\mathcal{P}[V_6^b] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_8^c] = \{\langle T + C_{p'_2} - D_8^c, 2\cdot T\rangle\}$$
$$p_{cand} = \textbf{\textit{null}}$$

Note that from $O_{p''_1}^*$*,* $O_{p''_1}^R$*,* $O_{p''_2}^*$ *and* $O_{p''_2}^R$*, we get* $O_{p''_1} = 3\cdot T$*,* $C_{p''_1} = D_4^c + T$*,* $O_{p''_2} = 3\cdot T + C_{p'_2} - D_8^c$ *and* $C_{p''_2} = C_{p'_2} + T$*, i.e., the costs computed for path* p''_1 *and* p''_2 *also by* SP *in Example 3.2.*

Next, at the fifth iteration, the algorithm pops label $\langle V_4^c, p''_1(V_s, V_1^a, V_3^a, V_4^c), T, 2\cdot T\rangle$*, considers transport edge* E_{48}^c *and creates path* $p'''_1(V_s, V_1^a, V_3^a, V_4^c, V_8^c)$ *with* $O_{p'''_1}^* = T$ *and* $O_{p'''_1}^R = 2\cdot T$*. Since* $\mathcal{P}[V_8^c]$ *is not empty,* $\mathcal{P}[V_8^c] = \{T + C_{p'_2} - D_8^c, 2\cdot T\rangle\}$*,* p'''_1 *is not the only path leading to* V_8^c*.* SPM *compares* $p'''_1(V_s, V_1^a, V_3^a, V_4^c, V_8^c)$ *with* $p''_2(V_s, V_2^b, V_6^b, V_8^c)$ *and determines that* p'''_1 *is more "promising" than* p''_2 *as* $O_{p''_2}^* > O_{p'''_1}^*$ *and* $O_{p''_2}^R = O_{p'''_1}^R$ *(recall that* $C_{p'_2} > D_8^c$*). Thus, the algorithm inserts the entries in* $\mathcal{Q}$ *and* $\mathcal{P}[V_8^c]$ *for path* p'''_1 *and, removes the entries for* p''_2*. So after the fifth iteration, we have:*

$$\mathcal{Q} = \{\langle V_8^c, p'''_1(V_s, V_1^a, V_3^a, V_4^c, V_8^c), T, 2\cdot T\rangle\}$$
$$\mathcal{P}[V_1^a] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_2^b] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_3^a] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_4^c] = \{\langle T, 2\cdot T\rangle\}$$
$$\mathcal{P}[V_6^b] = \{\langle T, 0\rangle\},\ \mathcal{P}[V_8^c] = \{\langle T, 2\cdot T\rangle\}$$
$$p_{cand} = \textbf{\textit{null}}$$

Note that at the same iteration, $\mathcal{Q}$ *for* SP *also contained the label* $\langle V_8^c, p''_2(V_s, V_2^b, V_6^b, V_8^c), 3\cdot T + C_{p'_2} - D_8^c, C_{p'_2} + T\rangle$ *for path* p''_2*.*

Finally, at the seventh iteration SPM *depletes* $\mathcal{Q}$ *popping label* $\langle V_9^c, (V_s, V_1^a, V_3^a, V_4^c, V_8^c, V_9^c), T, 2 \cdot T \rangle$ *from* $\mathcal{Q}$ *and since the target list* $\mathcal{T}$ *contains an entry for vertex* V_9^c*, it identifies final solution* $p_{cand} = p_1(V_s, V_1^a, V_3^a, V_4^c, V_8^c, V_9^c, V_e)$ *with* $O_{p_1} = 4 \cdot T$ *and* $C_{p_1} = D_9^c + \frac{3 \cdot T}{2}$*. Note that* SP *did not terminate at the seventh iteration as its queue was not empty, i.e.,* $\mathcal{Q} = \{\langle V_8^c, p''_2(V_s, V_2^b, V_6^b, V_8^c), 3 \cdot T + C_{p'_2} - D_8^c, C_{p'_2} + T \rangle\}$.

3.7 Experimental Evaluation

In this section, we present an experimental study of our methodology for solving dynamic Pickup and Delivery Problem with Transfers. We compare our methods against HTT, a rival method inspired by [53]. All methods are written in C++ and compiled with gcc. The evaluation is carried out on a 3Ghz Core 2 Duo CPU with 4GB RAM running Debian Linux.

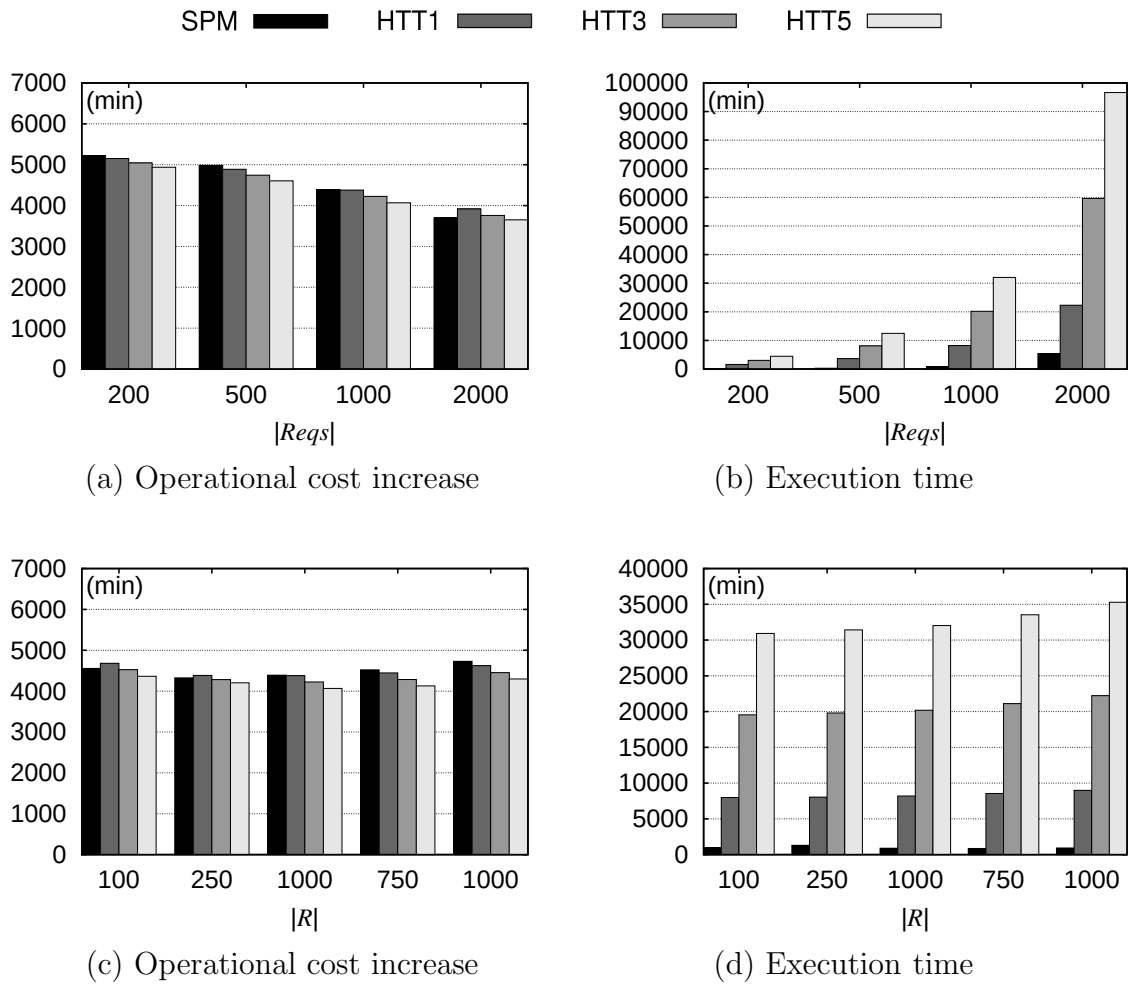
3.7.1 The HTT method

Satisfying dPDPT requests with HTT involves two phases. In the first phase, for every new dPDPT request, the method includes the pickup n_s and the delivery location n_e in two vehicle routes r_s and r_e, respectively. If n_s and n_e are included in the same route then a solution without a transfer of the package is identified otherwise the package will be transferred from vehicle r_s to r_e. To determine the vehicle routes r_s and r_e, HTT employs the cheapest insertion heuristic. Particularly, first, it examines every vehicle route r_s (resp. r_e) and for each pair of consecutive locations n_i and n_{i+1} in r_s (resp. r_e), that form an insertion "slot", it computes the detour cost $DS = dist(n_i, n_s) + dist(n_s, n_{i+1}) - dist(n_i, n_{i+1})$ for inserting pickup n_s (resp. delivery n_e) in between n_i and n_{i+1}. The detour cost DS signifies the extra time vehicle r_s (resp. r_e) must spend and therefore, it increases the total operational cost. Then, HTT considers every pair of routes r_s and r_e and determines the minimum detour cost for transferring the package from r_s to r_e; if $r_s = r_e$ the detour cost is of course zero. The idea is similar to the transfer with detour action introduced in Section 3.3. Finally, HTT selects the best overall combination of actions, i.e., including pickup n_s in vehicle route r_s, delivery n_e in r_e and transferring the package from r_s to r_e, that minimizes the increase of the total operational cost.

The second phase of HTT takes place periodically after k requests are satisfied during the first phase. It involves a tabu search improvement procedure that reduces the total operating cost. At each iteration, the tabu search considers every satisfied request and calculates what would be the change (increase or decrease) in the total operational cost removing the request from its current vehicle route (resp. pair of routes in case a transfer is considered) and inserting it to another route or pair of routes. Then, the tabu search selects the request with the best combination of removal and insertion, and performs these actions. Finally, the selected combination is characterized as tabu and cannot be executed for a number of future iterations.

3.7.2 Setup

To conduct our experiments, we consider the road networks of two cities; Oldernburg (OL) with 6,105 spatial locations and Athens (ATH) with 22,601 locations. First, we generate random pickup and delivery requests at each network and exploit the



(a) Operational cost increase

(b) Execution time

(c) Operational cost increase

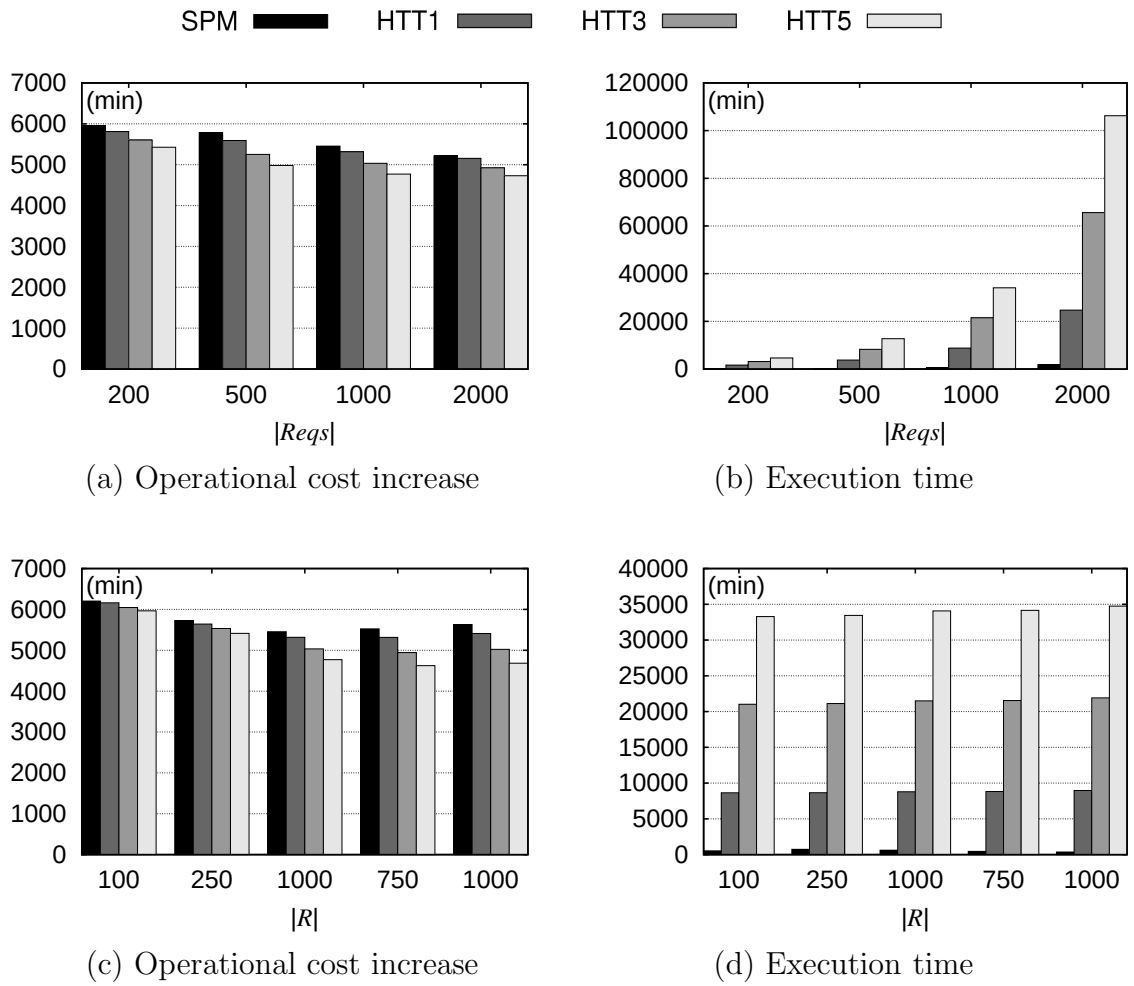
(d) Execution time

Figure 3.5: SPM *vs* HTT *at the OL road network.*

HTT method to construct collections of vehicle routes varying either the number of routes $|R|$, from 100 to 1000, or the number of requests $|Reqs|$ involved, from 200 to 2000. Then, for each of these route collections, we generate 500 random dPDPT requests and employ our best method SPM, and the HTT rival method to satisfy them. For HTT, we introduce three variations HTT1, HTT3 and HTT5 such that the tabu search is invoked once (after 500 requests are satisfied), three times (after 170) and five times (after 100), respectively. In addition, each time the tabu search is invoked, it performs 10 iterations. For each method, we measure (1) the increase in the total operational cost of the vehicles after all 500 requests are satisfied and (2) the total time needed to identify the solution to all the requests. Finally, note that we store both the road network and the vehicle routes on disk.

3.7.3 Experiments

Examining Figures 3.5 and 3.6 we make the following observations. The SPM method requires significantly less time to satisfy the 500 ad-hoc dPDPT requests, for all the values of the $|Reqs|$ and $|R|$ parameters, and for both the underlying road networks. In fact, when varying $|R|$, SPM is always one order of magnitude faster than all three HTT variants. In contrast, SPM results in slightly increased total



(a) Operational cost increase

(b) Execution time

(c) Operational cost increase

(d) Execution time

Figure 3.6: SPM *vs* HTT *at the ATH road network.*

operational cost compared to HTT, in most of the cases, and especially for large road networks as ATH. However, this advantage of HTT comes with a unavoidable trade-off between the increase of the total operational cost and the time needed to satisfy the ad-hoc dPDPT requests. The more often HTT employs the tabu search, the lower the increase of the total operational cost of the vehicles is. But, on the other hand, since each iteration of the tabu search needs to examine every route and identify the best reassignment for all the existing requests, the total time of HTT5 is higher than the time of HTT3 and HTT1.

Finally, we notice that as the number of pickup and delivery requests $|Reqs|$ involved in the initial static plan increases, satisfying the 500 ad-hoc dPDPT requests, either with HTT or SPM, results in a lower increase of the total operational cost but the total time needed to satisfy these requests increases. Notice that this is true regardless of the size of the underlying road network. As $|Reqs|$ increases and while $|R|$ remains fixed, the vehicle routes contain more spatial locations. This provides more insertion "slots" and enables both HTT and SPM to include the pickup and the delivery location of a dPDPT request with a lower cost. On the other hand, HTT slows down since it has to examine the reassignment of more requests during the tabu search, and SPM needs more time because the modified dynamic plan graph

is larger. Similar observations can be made in case of varying the number of routes $|R|$ and the HTT method.

3.8 Conclusions

In this chapter, we studied the dynamic Pickup and Delivery Problem with Transfers (dPDPT). This is the first work addressing the dynamic flavor of the problem. We propose a methodology that formulates dPDPT as a graph problem and identifies the solution to a request as the shortest path from a node representing the pickup location to that of the delivery location. Our experimental analysis shows that our method is able to find dPDPT solutions significantly faster than a two-phase local search algorithm that allows at most one transfer, while the quality of the solution is only marginally lower.

Chapter 4

Most Trusted Near Shortest Path

Identifying the *shortest path* (SP) on a graph is a fundamental and well-studied problem in the literature. In fact, there exist numerous other problems and applications that employ the computation of SP in the core of their methodology. Providing driving directions is an example of such applications. Particularly, many car navigation and route planning systems recommend the shortest path as the preferable way of driving from one location of a city to another. However, the shortest path is not always the most preferable way of driving around a city. To this end, in the context of transportation systems, works like [30, 55] consider a dynamic counterpart of SP, called *time-dependent shortest path* (a.k.a. time-dependent fastest path). In this case, the cost of traveling from a location n_i to another n_j on the road network depends on the departure time from n_i. For instance, driving through large roads like a freeway that connects the suburbs to the city center, takes more time in the morning hours than using the same road at night. Thus, the recommended path changes with respect to the time of the day a driver asks for directions.

Although computing the time-dependent SP may provide better driving directions than SP, they both fail to capture the actual way people drive around a city. Specifically, people tend to follow roads they already know and use in their every day driving e.g., to their work place, to their children's school etc., or roads that have followed in the past. In addition, even when they want to drive to a location for the first time, they usually ask their friends to recommend a "good" and safe way. In both cases, the goal is to avoid specific roads which, for example, are either included in high crime areas, or more likely to become dangerous in case of bad weather conditions, or have increased traffic. Thus, very often, a driver would rather follow a trusted and familiar way to move around the city over the fastest way.

Based on the above observations, the contributions of our work in this chapter can be summarized as follows:

(1) We introduce the *Most Trusted Near Shortest Path* (`MTNSP`) as a preferable way of driving around a city when a collection of trusted routes is available. For this purpose, we first define the notion of the *known graph* as a subgraph of the road network that is constructed merging the available trusted routes. Then, we seek for a path on the road network such that a driver will drive as less time as possible outside the known graph without significantly increasing, at the same time, the total duration of his journey compared to the fastest way, i.e., the shortest path.

(2) We define two costs for a path p between any pair of nodes n_s and n_t on the road network. The *length* L_p of p measures the total traveling time from n_s to n_t. The *unknown time* U_p of path p measures the total time spend outside the known graph. The `MTNSP` between two nodes n_s to n_t is the path p that has the lowest unknown time U_p among the paths with length L_p, at most α times larger than the length of the shortest path from n_s to n_t.

(3) We propose a methodology for identifying `MTNSP` that involves an *offline* and an *online* processing phase. The offline processing phase constructs an embedding $\mathcal{E}$ on the road network that efficiently indexes the distances of the network intersections and the time spend outside the known graph, i.e., the unknown time. Then, during the online phase a label-setting algorithm, called TRUSTME, employs the embedding in order to prune the search space and thus, speedup the evaluation of the `MTNSP` queries.

(4) We perform an extensive experimental analysis demonstrating the advantage of our methodology compared to a label-setting algorithm that exploits the euclidean distance of the network intersections to prune its search space.

The remainder of this chapter is organized as follows. Section 4.1 reviews the related work, and then, Section 4.2 provides an overview of the space embedding techniques. Section 4.3 formally defines the problem of the Most Trusted Near Shortest Path, and Section 4.4 presents our methodology for solving it. Finally, Section 4.5 presents an extensive experimental evaluation and Section 4.6 concludes this work.

4.1 Related Work

Our work in this chapter is related to graph embedding techniques and to shortest path problems.

Embedding techniques. In domains with a computationally expensive distance function, significant speed-ups can be obtained by embedding objects into another space and using a more efficient distance function, such as one of the Minkowski metrics. For instance, several methods have been proposed to embed a space into the Euclidean space [7, 38].

For graph problems, the Lipschitz embedding with singleton reference sets called landmarks or reference nodes is a widely adopted embedding technique. The basic idea is to precompute the distances of each graph node to every of the k landmarks selected and then, represent a node by a k-dimensional vector. This technique has been employed in the field of computer networks in many applications such as round-trip propagation and transmission delay [29, 54, 60, 69], and furthermore, in the field of spatial networks, for speeding up the evaluation of proximity [45, 46] and kNN [66] queries, and the computation of shortest path [34, 35] and of multi-criteria shortest path [47]. However, there exist very few works, e.g., [59], that deal with the important issue of how to select the landmarks for the embedding.

Shortest path problem and its variants. Bellman-Ford and Dijkstra are the most well-known algorithms for finding the *shortest path* between two nodes in a graph. The ALT algorithms [34, 35, 57] perform a bidirectional A* search and exploit

a lower bound of the distance between two nodes to direct the search. There exist a number of materialization techniques [3, 42, 44] or encoding/labeling schemes [19, 22] that can be used to efficiently compute the shortest path. Both the ALT algorithms and the materialization and encoding methods are mostly suitable for graphs that are not frequently updated, since they require expensive precomputation.

In *multi-criteria shortest path problems* the quality of a path is measured by multiple metrics, and the goal is to find all paths for which no better exists. Algorithms are categorized into three classes. The methods of the first class (e.g., [18]) apply a user preference function to reduce the original multi-criteria problem to a conventional shortest path problem. The second class contains the interactive methods (e.g., [36]) that interact with a decision maker to come up with the answer path. Finally, the third class includes label-setting and label-correcting methods (e.g., [37, 47, 70]). These methods construct a label for every path followed to reach a graph node. Then, at each iteration, they select the path with the minimum cost, defined as the combination of the given criteria, and expand the search extending this path.

Finally, in *near-shortest path problem* the goal is to identify the paths whose length is within a factor of $1 + \epsilon$ of the shortest-path length for some user-specified $\epsilon \geq 0$. [16] is among the first works that focus on the near-shortest path adopting ideas from dynamic programming to come up with a solution. Later, [17] extend this solution to speed up the computation of the near-shortest paths.

4.2 Background on Space Embedding Techniques

Let (S, d_S) be a finite metric space where S is a finite set of objects and $d_S : S \times S \rightarrow \mathbb{R}^+$ is a distance metric over S. The *embedding*, or *transformation*, of the finite metric space (S, d_S) into a vector space $(\mathbb{R}^k, d^r)$ is a mapping $\mathcal{E} : S \rightarrow \mathbb{R}^k$ where k is the dimensionality of the vector space and d^r is one of the Minkowski metrics in $\mathbb{R}^k$. Formally:

$$d^r(x, y) = \left[\sum_{i=1}^{k} |x_i - y_i|^r \right]^{1/r}$$

where x_i and y_i are the i-th coordinates of points x and y in space, respectively, and r is the order of the Minkowski metric. For instance, when $r = 1$ the metric is known as the Manhattan distance and when $r = 2$ is known as the Euclidean distance.

The objective of an embedding is to provide a fast and computationally simple d^r function such that $d_S(x, y) \cong d^r(\mathcal{E}(x), \mathcal{E}(y))$. In other words, the distance between two objects x and y in the original metric space should be close enough to the distance between their corresponding embedded points $\mathcal{E}(x)$ and $\mathcal{E}(y)$ in the embedding space. The quality of an embedding $\mathcal{E}$ is measured by means of the *distortion* and the *stress*. The distortion specifies the maximum difference between distance functions d_S and d^r, and it is equal to $c_1 \cdot c_2$ with $c_1, c_2 > 1$, when it is guaranteed that for the embedding $\mathcal{E}$:

$$\frac{d_S(x, y)}{c_1} \leq d^r(\mathcal{E}(x), \mathcal{E}(y)) \leq c_2 \cdot d_S(x, y)$$

for all pairs of objects $x, y \in S$. Stress represents the overall deviation in the distance and is defined as:

$$stress = \frac{\sum_{x,y \in S} \left(d^r(\mathcal{E}(x), \mathcal{E}(y)) - d_S(x,y)\right)^2}{\sum_{x,y \in S} \left(d_S(x,y)\right)^2}$$

The optimum d^r function generates zero stress, equivalent to no distortion (i.e., $c_1 = c_2 = 1$).

In Section 4.4.1, we discuss how the Lipschitz embedding technique can be employed for computing the Most Trusted Near Shortest Path.

4.3 Problem Definition

This section formally defines the problem of computing the *Most Trusted Near Shortest Path* and introduces the basic notation that will be used in the rest of the chapter.

First, we define the notion of a *network graph* that represents the road network of a city.

Definition 4.1 (Network graph)**.** *The* network graph *is an undirected weighted graph $G_N(N, E, W)$ where N is a set of nodes that represent the intersections of a road network, $E \subset N \times N$ is a set of edges that represent the road segments connecting the network's intersections, and function $W : E \to \mathbb{R}^+$ associates each edge (n_i, n_j) to a positive real number denoting the traveling time between the intersections represented by nodes n_i and n_j.*

Given two nodes n_i and n_j in a network graph G_N their *network distance* $d_N(n_i, n_j)$ is defined as the total traveling time of the shortest path from n_i to n_j or vice versa on G_N.

Consider a group of people that track their every day movement on the road and then, share their driving data. Given the network graph G_N of a city, the movement of a vehicle is captured by a sequence of graph nodes, termed *route*, and the shared collection of such routing data defines a subgraph of the network graph, called *known graph*.

Definition 4.2 (Known graph)**.** *Let $G_N(N, E, W)$ be a network graph and R be a collection of available routes. The* known graph *$G_K(N_U, E_U)$ is a subgraph of G_N where the set of nodes $N_K \subset N$ contains a node n if n is included in a route of R, and the set of edges $E_K \subset E$ contains an edge (n_i, n_j) if a route of R has a pair of consecutive nodes (n_i, n_j) or (n_j, n_i).*

The known subgraph G_K of a network graph G_N provides a *trusted* and *familiar* way of driving around the city. Therefore, the drivers consult G_K whenever they want to travel from one location of the city to another. Normally, e.g., if the group of people does not contain any taxi drivers, the known graph is smaller then the network graph G_N. The rest of the network graph is consider as *unknown* and driving through its nodes and its edges is avoided as much as possible.

Definition 4.3 (Unknown graph)**.** *Let $G_N(N, E, W)$ be a network graph and $G_K(N_K, E_K)$ be a known subgraph of G_N. The* unknown graph *$G_U(N_U, E_U)$ w.r.t. to G_K is a subgraph of G_N where $N_U \subset N$ is the set of nodes and the set of edges $E_U = E \setminus E_K$ contains every edge (n_i, n_j) not contained in G_K.*

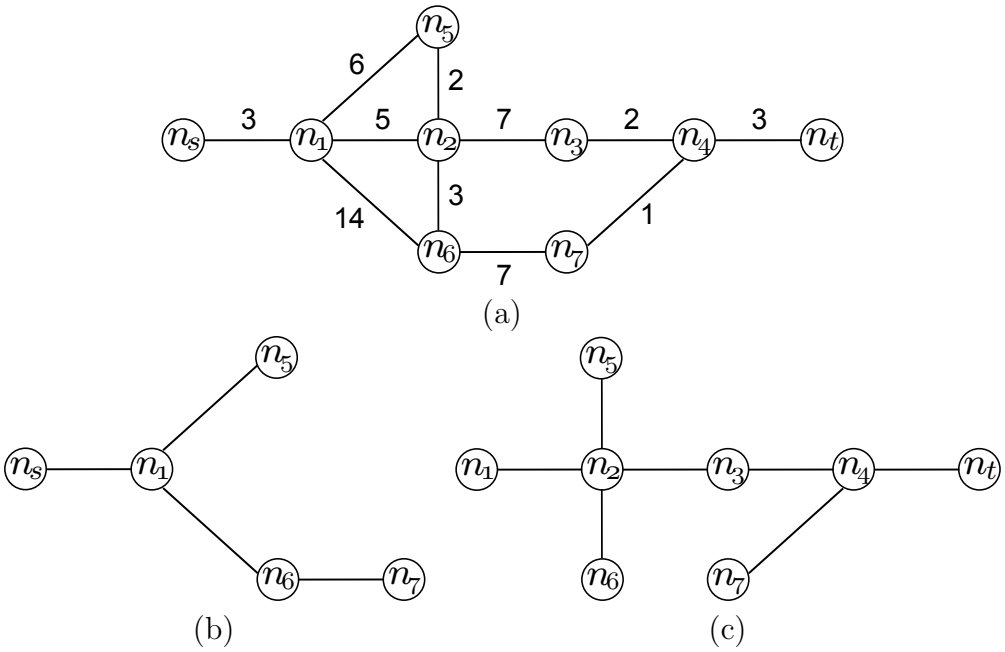


Figure 4.1: *An example of a network graph G_N, a known subgraph G_K of G_N, and the unknown subgraph G_U of G_N w.r.t. G_K*

Note that the unknown graph is not necessarily connected; it may be composed by a number of disconnected small graphs. This is because people tend to drive through the segments of large roads like freeways while they avoid the small local roads connected to them.

Example 4.1. *Consider the network graph G_N in Figure 4.1(a). Based on the routes available, the known subgraph G_K of G_N in Figure 4.1(b) is defined. Consequently, Figure 4.1(c) shows the unknown subgraph G_U of G_N w.r.t. G_K. Notice that nodes n_1, n_5, n_6 and n_7 are contained in both G_K and G_U, whereas every edge of the network graph G_N is contained only in one of the subgraphs.*

Given two nodes n_s and n_t on G_N, a *path* is a sequence of nodes $(n_s, \ldots, n_t)$ on G_N that represents a way of driving from n_s to n_t. There are two costs associated with a path $p(n_s, \ldots, n_t)$. The *length* L_p of p measures the total traveling time from n_s to n_t and it is defined as the sum of the edge weights $w(n_i, n_{i+1})$ for every consecutive pair of nodes n_i and n_{i+1} in p. The *unknown time* U_p of path p measures the total time spend for moving on the unknown graph G_U, and it is defined as the sum of the weights $w(n_i, n_{i+1})$ for every edge (n_i, n_{i+1}) of G_U included in p.

Assume that a driver wants to go from node n_s to n_t of a network graph G_N and that a known subgraph G_K of G_N is available. Based on the previous definitions, the driver seeks for a path $p(n_s, \ldots, n_t)$ on G_N such that he will drive as less time as possible on the unknown graph G_U but, in addition, this will not significantly increase the total duration of his journey compared to the fastest way, i.e., the shortest path from n_s to n_t. This is the intuition behind the *Most Trusted Near Shortest Path* query.

Definition 4.4 (`MTNSP` query)**.** *Let G_N be a network graph, G_K be a known subgraph of G_N and G_U be the unknown subgraph of G_N w.r.t. G_K. The* Most Trusted Near Shortest Path *query denoted by* `MTNSP`(n_s, n_t, α) *returns the path $p(n_s, \ldots, n_t)$*

from n_s to n_t that has the lowest unknown time U_p among the paths of length, $L_p \leq \alpha \cdot d_N(n_s, n_t)$.

Example 4.2. *Consider the network graph G_N in Figure 4.1(a) and the known G_K and the unknown graph G_U in Figure 4.1(b) and Figure 4.1(c) respectively. Assume that a driver poses query* `MTNSP`$(n_s, n_t, 1.1)$*, i.e., he wants to go from node n_s to n_t on G_N traveling as less time as possible on the unknown graph G_U, and he wants his journey to be at most 10% longer than the fastest way.*

The shortest (fastest) path from n_s to n_t on G_N is $sp(n_s, n_1, n_2, n_3, n_4, n_t)$ with $L_{sp} = 20 = d_N(n_s, n_t)$ and $U_{sp} = 17$, since the driver has to follow edges (n_1, n_2), (n_2, n_3), (n_3, n_4) and (n_4, n_t) on G_U. This means that a path p on G_N is acceptable by the driver only if $L_p \leq 22$. Thus, the answer to the query is $p_1(n_s, n_1, n_2, n_6, n_7, n_4, n_t)$ with $L_{p_1} = 22$ and $U_{p_1} = 12$. Note that p_1 does not have the lowest unknown time compared to all paths from n_s to n_t; path $p_2(n_s, n_1, n_6, n_7, n_4, n_t)$ has $U_{p_2} = 4$, but p_2 is not acceptable as $L_{p_2} = 28 > 22$.

4.4 Evaluating `MTNSP` Queries

Our methodology for answering `MTNSP` queries involves an *offline* and an *online processing* phase.

4.4.1 Offline processing

Consider a network graph G_N, a known subgraph G_K of G_N and the unknown subgraph G_U of G_N w.r.t. G_K. The key idea of the offline processing phase is to efficiently index the network distances on G_N and the time spend on G_U in order to speedup the evaluation of `MTNSP` queries during the online phase. The proposed indexing scheme should meet the following two objectives:

(1) Provide a lower and an upper bound of the network distance $d_N(n_i, n_j)$, for every pair of nodes n_i and n_j in G_N.

(2) Provide a lower and an upper bound of the unknown time for every path $p(n_i, \ldots, n_j)$ between two nodes n_i and n_j in G_N.

We adopt an approach similar to the Lipschitz embedding technique with singleton reference sets, called *landmarks* (a.k.a. *reference nodes*) which in our case are the nodes contained in the known graph $G_K(N_K, E_K)$. Particularly, for each node n_i in the network graph, we precompute all the shortest paths from n_i to every landmark n_{ℓ_j}, and store (1) the network distance $d_N(n_i, n_{\ell_j})$ and (2) the lowest unknown time among these shortest paths. Thus, every node n_i in G_N is represented by the embedding $\mathcal{E}$ on G_N as a $(2 \times k)$ vector:

$$\mathcal{E}(n_i) = \begin{pmatrix} d_N(n_i, n_{\ell_1}) & U_{i,\ell_1} \\ \vdots & \vdots \\ d_N(n_i, n_{\ell_k}) & U_{i,\ell_k} \end{pmatrix}$$

where $k = |N_K|$ and U_{i,ℓ_j} is the lowest unknown time of the shortest paths from n_i to landmark n_{ℓ_j}.

N \ N_K	n_s	n_1	n_5	n_6	n_7
n_s	0	3	9	11	18
n_1	3	0	6	8	15
n_2	8	5	2	3	10
n_3	15	12	9	10	3
n_4	17	14	11	8	1
n_5	9	6	0	5	12
n_6	11	8	5	0	7
n_7	18	15	12	7	0
n_t	20	17	14	11	4

Table 4.1: *Distances to every landmark.*

N \ N_K	n_s	n_1	n_5	n_6	n_7
n_s	0	0	0	8	8
n_1	0	0	0	8	8
n_2	5	5	2	3	3
n_3	12	12	9	3	3
n_4	14	14	11	1	1
n_5	0	0	0	5	5
n_6	8	8	5	0	0
n_7	8	8	5	0	0
n_t	17	17	14	4	4

Table 4.2: *Lowest unknown time of the shortest paths to every landmark.*

Example 4.3. *Consider the network graph G_N, the known subgraph G_K and the unknown subgraph G_U in Figure 4.1. We construct the embedding $\mathcal{E}$ on G_N selecting nodes n_s, n_1, n_5, n_6 and n_7 as landmarks, i.e., the nodes in G_K. Table 4.1 and Table 4.2 show the distances of every node n_i in G_N to each landmark n_{ℓ_j} and the lowest unknown time among the shortest paths $(n_i, \ldots, n_{\ell_j})$, respectively. For instance, node n_1 is represented as the following vector:*

$$\mathcal{E}(n_1) = \begin{pmatrix} 3 & 0 \\ 0 & 0 \\ 6 & 0 \\ 8 & 8 \\ 15 & 8 \end{pmatrix}$$

Note that in the case of landmark n_7 there exist two shortest paths from n_1 to n_7, $p_1(n_1, n_2, n_6, n_7)$ with $L_{p_1} = 15$ and $U_{p_1} = 8$, and $p_2(n_1, n_2, n_3, n_4, n_7)$ with $L_{p_2} = 15$ and $U_{p_2} = 15$, but we only consider p_1 as it has the lowest unknown time.

Next, we discuss how the lower and the upper bound of the network distance between two nodes can be determined using the embedding $\mathcal{E}$ on G_N. The idea is to exploit the triangle inequality for any three nodes in G_N.

Proposition 4.1 (Lower and upper bound of network distance)**.** *Let $G_N(N, E, W)$ be a network graph, $G_K(N_K, E_K)$ be a known subgraph of G_N, and $\mathcal{E}$ be the embedding on G_N using the nodes in N_K as landmarks. For every pair of nodes n_s and n_t in G_N, the following hold:*

$$d_N(n_s, n_t) \geq \max_{i=1}^{k} |d_N(n_s, n_{\ell_i}) - d_N(n_t, n_{\ell_i})|$$

and

$$d_N(n_s, n_t) \leq \min_{j=1}^{k} \{d_N(n_s, n_{\ell_j}) + d_N(n_t, n_{\ell_j})\}$$

where nodes n_{ℓ_i} and n_{ℓ_j} are two landmarks, and $k = |N_K|$.

Proof. The bounds can be directly deduced from the fact that the network distance on G_N is a metric, and therefore, it satisfies the triangle inequality for any three nodes n_s, n_i and n_t in G_N:

$$d_N(n_s, n_t) \geq |d_N(n_s, n_i) - d_N(n_t, n_i)|$$

and

$$d_N(n_s, n_t) \leq d_N(n_s, n_i) + d_N(n_t, n_i)$$

□

We denote the lower and the upper bound of the network distance $d_N(n_s, n_t)$ as $\underline{d_N}(n_s, n_t)$ and $\overline{d_N}(n_s, n_t)$, respectively.

Finally, we present how the lower $\underline{U_p}$ and the upper bound $\overline{U_p}$ of the unknown time for a path $p(n_s, \ldots, n_t)$ on network graph G_N is determined. Observe that the unknown time U_p of this path cannot be greater than the network distance $d_N(n_s, n_t)$. In fact, the worst case $U_p = d_N(n_s, n_t)$ occurs only when p contains only edges from the unknown subgraph G_U and therefore, we obtain $\overline{U_p} = \overline{d_N}(n_s, n_t)$ as the upper bound for the unknown time U_p. Yet, we can have a better bound for U_p considering the path p' of length equal to the upper bound of the network distance $d_N(n_s, n_t)$, i.e., $L_{p'} = \overline{d_N}(n_s, n_t)$. This path passes through a landmark n_{ℓ_j} such that $\overline{d_N}(n_s, n_t) = d_N(n_s, n_{\ell_j}) + d_N(n_t, n_{\ell_j})$ and thus, the upper bound of the unknown time of path p is $\overline{U_p} = U_{s,\ell_j} + U_{t,\ell_j}$, where U_{s,ℓ_j} (U_{t,ℓ_j}) is the lowest unknown time of the shortest paths from n_s (n_t) to landmark n_{ℓ_j}.

On the other hand, it is not straightforward how to find a lower bound for U_p. However this is possible when n_s is contained in the known subgraph G_K and not in the unknown G_U, and n_t is in G_U but not in G_K. In this case, there must exist a path from n_s to n_t via some landmark n_{ℓ_j} that lies on the border of the known and unknown subgraphs. This landmark must be the closest to n_s landmark, and thus, the network distance $d_N(n_s, n_{\ell_j})$ is a lower bound to U_p . Since the network has no direction, the reverse, i.e., when n_s and n_t are interchanged, also holds.

Proposition 4.2 (Lower bound of unknown time)**.** *Let $G_N(N, E, W)$ be a network graph, $G_K(N_K, E_K)$ be a known subgraph of G_N, $G_U(N_U, E_U)$ be the known subgraph of G_N w.r.t. G_K, and $\mathcal{E}$ be the embedding on G_N using the nodes in G_K as landmarks. Given two nodes n_s and n_t in G_N, the following holds for every path $p(n_s, \ldots, n_t)$ on G_N:*

$$U_p \geq \begin{cases} \min_{j=1}^{k} d_N(n_s, n_{\ell_j}), & \text{if } n_s \in G_U \text{ and } n_s \notin G_K, \text{ and } n_t \in G_K \text{ and } n_t \notin G_U \\ \min_{j=1}^{k} d_N(n_t, n_{\ell_j}), & \text{if } n_s \in G_K \text{ and } n_s \notin G_U, \text{ and } n_t \in G_U \text{ and } n_t \notin G_K \\ 0, & \text{otherwise} \end{cases}$$

where n_{ℓ_j} is a landmark and $k = |N_K|$.

Proof. For simplicity, we prove only the case when $n_s \in G_U$ and $n_t \in G_K$. Assume that there exists a landmark $n' \neq n_{\ell_j}$ with $d_N(n_s, n') > d_N(n_s, n_{\ell_j})$ and for the unknown time of path $p'(n_s, \ldots, n')$, $U_{p'} < U_{s,\ell_j}$ holds. Since, n_{ℓ_j} is the nearest landmark to n_s w.r.t. the network distance $\big(d_N(n_s, n_{\ell_j}) = \min_{i=1}^{k} d_N(n_s, n_{\ell_i})\big)$, the shortest path from n_s to n_{ℓ_j} does not contain any other landmark, and therefore, $U_{s,\ell_j} = d_N(n_s, n_{\ell_j})$. Similarly, since n' is the nearest landmark to n_s w.r.t. the unknown time, also path p' does not contain any other landmark, and $U_{p'} = d_N(n_s, n')$. Thus, from $U_{p'} < U_{s,\ell_j}$ we have $d_N(n_s, n') < d_N(n_s, n_{\ell_j})$ which cannot be true. □

4.4.2 Online processing

During the online processing phase, we employ a label setting algorithm, termed TRUSTME, to answer a $\texttt{MTNSP}(n_s, n_t, \alpha)$ query. The algorithm has the following key features. First, it may visit a node n in a network graph G_N more than once following multiple paths from source n_s. For each of these paths $p(n_s, \ldots, n)$, a label $\langle n, p, L_p, U_p \rangle$ is defined, where L_p is the length of the path and U_p is its unknown time. This is similar to the algorithms for multi-criteria shortest path as there exist two costs associated with a path on G_N. However, in order to speedup the search, TRUSTME retains, for every node n, the labels of the most "promising" paths from n_s to n. Second, the algorithm computes an upper bound of the unknown time for the answer path when either the target n_t is reached or Proposition 4.3, described below, can be applied. This estimation is progressively improved during the search until it becomes equal to the unknown time of the answer path. Finally, TRUSTME traverses the network graph w.r.t. the L_p cost. Thus, at each iteration it selects the label $\langle n_q, p_q, L_{p_q}, U_{p_q} \rangle$ with the lowest length cost and expands the path p_q considering every (n_q, n) edge in G_N. During this expansion phase, TRUSTME exploits the upper and the lower bound of both the network distance and the unknown time, provided by the embedding on the network graph, in order to discard the non "promising" paths and prune the search space.

Proposition 4.3 (Upper bound of MTNSP)**.** *Let p_{SOL} be the answer path to query $\texttt{MTNSP}(n_s, n_t, \alpha)$ and $p(n_s, \ldots, n_i)$ be a path from n_s to node n_i. The following holds for path p^{SOL}:*

$$U_{p^{SOL}} \leq \begin{cases} U_p + \overline{U_{p'}}, & \text{if } L_p \leq d_N(n_s, n_t) \text{ and } L_p + \overline{d_N}(n_i, n_t) \leq \alpha \cdot L_p \text{ or} \\ & \quad L_p > d_N(n_s, n_t) \text{ and } L_p + \overline{d_N}(n_i, n_t) \leq \alpha \cdot d_N(n_s, n_t) \\ \infty, & \text{otherwise} \end{cases}$$

where $\overline{U_{p'}}$ is the upper bound of the unknown time for a path $p'(n_i, \ldots, n_t)$.

Proof. In order for $U_p + \overline{U_{p'}}$ to be a upper bound for the unknown time of the answer path p^{SOL}, it must also hold that its length, upper bounded by $L_p + \overline{d_N}(n_i, n_t)$, is lower than $\alpha \cdot d_N(n_s, n_t)$. This is straightforward in the second case, i.e., when $L_p \geq d_N(n_s, n_t)$. In the first case, i.e., when $L_p < d_N(n_s, n_t)$ and $L_p + \overline{d_N}(n_i, n_t) \leq \alpha \cdot L_p$, we need to prove that $L_p + \overline{d_N}(n_i, n_t) < \alpha \cdot d_N(n_s, n_t)$ also holds. This however is true because from $L_p < d_N(n_s, n_t)$ and $\alpha > 1$ we get $\alpha \cdot L_p < \alpha \cdot d_N(n_s, n_t)$. □

Algorithm TRUSTME
Input: $\texttt{MTNSP}(n_s, n_t, \alpha)$, network graph G_N, embedding $\mathcal{E}$ of G_N
Output: path $p(n_s, \ldots, n_t)$ on G_N with lowest U_p and $L_p \leq \alpha \cdot d_N(n_s, n_t)$
Parameters:

priority queue $\mathcal{Q}$:	the search queue sorted by L_p in ascending order
set $\mathcal{P}[n]$:	the most "promising" paths to each node n contained in a label of $\mathcal{Q}$
path p_{cand}:	a candidate answer path to the MTNSP query
cost $\overline{L}$:	an upper bound for $d_N(n_s, n_t)$
cost $\overline{U}$:	an upper bound for the unknown time of the answer path

Method:

1: $\overline{L} = \overline{d_N}(n_s, n_t)$;
2: $\overline{U} = \infty$;
3: **push** label $\langle n_s, (n_s), 0, 0\rangle$ to $\mathcal{Q}$;
4: **insert** entry $\langle 0, 0\rangle$ in set $\mathcal{P}[n_s]$
5: **while** $\mathcal{Q}$ is not empty **do**
6: **pop** label $\langle n_q, p_q, L_{p_q}, U_{p_q}\rangle$ from $\mathcal{Q}$;
7: UpdateCandidateAnswerAndBounds$(\langle n_q, p_q, L_{p_q}, U_{p_q}\rangle, p_{cand}, \overline{L}, \overline{U})$;
8: **for** each edge (n_q, n) in G_N **do**
9: **create** path $p = p_q \cup (n_q, n) = (n_s, \ldots, n_q, n)$;
10: **let** $\underline{U_n}$ be the lower bound of the unknown time for a path $(n, \ldots, n_t)$;
11: **if** $L_p + \underline{d_N}(n, n_t) > \alpha \cdot \overline{L}$ **then**
ignore path p;
12: **else if** $\overline{L} < L_p + \underline{d_N}(n, n_t) \leq \alpha \cdot \overline{L}$ **and** $U_p + \underline{U_n} > \overline{U}$ **then**
ignore path p;
13: **else if** exists entry $\langle L_{p'}, U_{p'}\rangle$ in $\mathcal{P}[n]$ **with** $L_{p'} \leq \alpha \cdot \overline{L}$ **and** $L_p \geq L_{p'}$ **and** $U_p \geq U_{p'}$ **then**
ignore path p;
14: **else**
15: **push** label $\langle n, p, L_p, U_p\rangle$ to $\mathcal{Q}$;
16: **insert** entry $\langle L_p, U_p\rangle$ in $\mathcal{P}[n]$;
17: **delete** every entry $\langle L_{p'}, U_{p'}\rangle$ in $\mathcal{P}[n]$ **where** $L_{p'} \geq L_p$ **and** $U_{p'} \geq U_p$ and the corresponding labels
18: in $\mathcal{Q}$;
19: **end if**
20: **end for**
21: **end while**
22: **return** p_{cand};

Figure 4.2: *The* TRUSTME *algorithm.*

Figure 4.2 illustrates the pseudocode of the TRUSTME algorithm. TRUSTME takes as inputs: a $\texttt{MTNSP}(n_s, n_t, \alpha)$ query, a network graph G_N and the embedding $\mathcal{E}$ on G_N, and returns the answer path to the query. The algorithm uses the following data structures: (1) a priority queue $\mathcal{Q}$, (2) a set $\mathcal{P}[n]$ for each node n contained in the labels inserted in $\mathcal{Q}$, (3) a path p_{cand}, and (4) costs $\overline{L}$ and $\overline{U}$. The priority queue $\mathcal{Q}$ is used to perform the search storing every label $\langle n, p, L_p, U_p\rangle$ to be checked, sorted by L_p in ascending order. The set $\mathcal{P}[n]$ contains entries in the form of $\langle L_p, U_p\rangle$ for each "promising" path p from n_s to n found. The entries in $\mathcal{P}[n]$ are used to prune the non "promising" paths to node n constructed during the search. Cost $\overline{L}$ is an upper bound of the network distance $d_N(n_s, n_t)$. Initially, $\overline{L}$ is equal to $\overline{d_N}(n_s, n_t)$, given by the embedding $\mathcal{E}$, and after computing the shortest path from n_s to n_t, it becomes equal to $d_N(n_s, n_t)$. Finally, cost $\overline{U}$ is an upper bound of the unknown time for the answer path. Initially, $\overline{U}$ is set to ∞ and during the search it is progressively improved. Both $\overline{L}$ and $\overline{U}$ are employed to prune the non "promising" paths constructed during the search.

The TRUSTME algorithm proceeds as follows. On Lines 1–4, it initializes the costs $\overline{L}$ and $\overline{U}$, the priority queue $\mathcal{Q}$ with label $\langle n_s, (n_s), 0, 0\rangle$ for source node n_s and set $\mathcal{P}[n_s]$. Then, it iteratively examines the contents of $\mathcal{Q}$ (Lines 5–21) until the queue is depleted. Each iteration involves the following three steps. First,

the label $\langle n_q, p_q, L_{p_q}, U_{p_q}\rangle$ with the lowest L_{p_q} cost is popped from $\mathcal{Q}$ on Line 6. Next, TRUSTME invokes the UpdateCandidateAnswerAndBounds procedure (Line 6) to update the candidate answer path p_{cand} and costs $\overline{L}$ and $\overline{U}$. UpdateCandidateAnswerAndBounds considers two cases.

First, if node n_q of current label $\langle n_q, p_q, L_{p_q}, U_{p_q}\rangle$ is the target n_t then a path $p(n_s, \ldots, n_t)$ is found. The first path ever identified by UpdateCandidateAnswerAndBounds is always the shortest path from source n_s to target n_t since TRUSTME traverses the graph w.r.t. the L_p cost in ascending order. Thus, in this case, bound $\overline{L}$ is set to $d_N(n_s, n_t)$. For every path $p_q(n_s, \ldots, n_t)$ found (including the shortest path), U_{p_q} is compared against current upper bound $\overline{U}$. If $U_{p_q} \leq \overline{U}$ then a new candidate answer path is identified, $p_{cand} = p_q$ and therefore, current upper bound $\overline{U}$ is improved, $\overline{U} = U_{p_q}$. On other hand, if node n_q of current label is not target n_t, UpdateCandidateAnswerAndBounds employs Proposition 4.3 to compute an upper bound of the unknown time for an answer path through node n_q, and if it is necessary the current upper bound $\overline{U}$ is properly updated.

Finally, on Lines 8–20, TRUSTME expands the search considering every (n_q, n) edge on the network graph G_N. Specifically, first, path $p_q(n_s, \ldots, n_q)$ is extended to create $p(n_s, \ldots, n_q, n)$. Then, on Lines 10–19, the algorithm determines whether p is a "promising" path and thus, it must be extended at a future iteration, or it must be discarded. Path p is discarded in three cases:

(1) if $L_p + \underline{d_N}(n, n_t) > \alpha \cdot \overline{L}$, i.e., the extension of path p towards target n_t would have length at least equal to $L_p + \underline{d_N}(n, n_t)$ which exceeds the constraint imposed by the `MTNSP` query (Line 11),

(2) if $\overline{L} < L_p + \underline{d_N}(n, n_t) \leq \alpha \cdot \overline{L}$ cand $U_p + \underline{U_n} > \overline{U}$, where $\underline{U_n}$ is the lower bound of the unknown time for a path $(n, \ldots, n_t)$ computed using Proposition 4.2. In this case, path p is acceptable w.r.t. to its length but even the lower bound of the unknown time $U_p + \underline{U_n}$ needed to reach the target is higher than current upper bound $\overline{U}$ (Line 12), and

(3) if set $\mathcal{P}[n]$ contains an entry $\langle L_{p'}, U_{p'}\rangle$ such that $L_p \geq L_{p'}$ and $U_p \geq U_{p'}$, i.e., extending path p will not ever contribute a better solution than extending p' (Line 13).

Otherwise, p is a "promising" path, and TRUSTME inserts label $\langle n, p, L_p, U_p\rangle$ in $\mathcal{Q}$ and entry $\langle L_p, U_p\rangle$ in $\mathcal{P}[n]$. In addition, on Line 16, the algorithm updates set $\mathcal{P}[n]$ removing every entry $\langle L_{p'}, U_{p'}\rangle$ with $L_{p'} \geq L_p$ and $U_{p'} \geq U_p$, and the corresponding labels in $\mathcal{Q}$, since such path p' will never contribute a better solution than p.

Example 4.4. *We illustrate* TRUSTME *for query* `MTNSP`$(n_s, n_t, 1.3)$ *using the graphs and the embedding of Example 4.3 with nodes* n_s, n_1, n_5, n_6 *and* n_7 *as landmarks. In addition, recall from Example 4.2 that* $d_N(n_s, n_t) = 20$.

Initially, we have priority $\mathcal{Q} = \langle n_s, (n_s), 0, 0\rangle$, *set* $\mathcal{P}[n_s] = \{\langle 0, 0\rangle\}$, *and costs* $\overline{L} = \overline{d_N}(n_s, n_t) = 20$ *and* $\overline{U} = \infty$. *At the first iteration, the algorithm pops label* $\langle n_s, (n_s), 0, 0\rangle$. *After considering edge* (n_s, n_1) *in* G_N, *it inserts label* $\langle n_1, (n_s, n_1), 3, 0\rangle$ *in* $\mathcal{Q}$ *and entry* $\langle 3, 0\rangle$ *in set* $\mathcal{P}[n_1]$. *Similar, at the second iteration, label* $\langle n_1, (n_s, n_1), 3, 0\rangle$ *is popped from* $\mathcal{Q}$ *and* TRUSTME *considers edges* (n_1, n_s), (n_1, n_2), (n_1, n_5) *and* (n_1, n_6). *Particularly:*

- *For edge* (n_1, n_s)*, the created path* (n_s, n_1, n_s) *of length* 6 *and unknown time* 0 *is discarded due to the entry* $\langle 0, 0\rangle$ *in* $\mathcal{P}[n_s]$*.*

- *For edge* (n_1, n_2)*, the label* $\langle n_2, (n_s, n_1, n_2), 8, 5\rangle$ *is pushed to* $\mathcal{Q}$ *and the entry* $\langle 8, 5\rangle$ *is inserted in* $\mathcal{P}[n_2]$*.*

- *For edge* (n_1, n_5)*, the label* $\langle n_5, (n_s, n_1, n_5), 9, 0\rangle$ *is pushed to* $\mathcal{Q}$ *and the entry* $\langle 9, 0\rangle$ *is inserted in* $\mathcal{P}[n_5]$*.*

- *For edge* (n_1, n_6)*, the created path* (n_s, n_1, n_6) *of length* 17 *is discarded since* $17 + \underline{d_N}(n_6, n_t) = 17 + 11 = 28 > 1.3 \cdot \overline{L} = 26$*.*

Thus, after the second iteration we have:

$$\begin{aligned}
&\mathcal{Q} = \{\langle n_2, (n_s, n_1, n_2), 8, 5\rangle, \langle n_5, (n_s, n_1, n_5), 9, 0\rangle\}\\
&\mathcal{P}[n_s] = \{\langle 0, 0\rangle\},\ \mathcal{P}[n_1] = \{\langle 3, 0\rangle\},\ \mathcal{P}[n_2] = \{\langle 8, 5\rangle\},\ \mathcal{P}[n_5] = \{\langle 9, 0\rangle\}\\
&\overline{L} = 20\\
&\overline{U} = \infty
\end{aligned}$$

Note that the leftmost label in $\mathcal{Q}$ *always contains the path with the lowest* L_p *cost.*

TRUSTME *proceeds similarly and at the tenth iteration label* $\langle 7, p(n_s, n_1, n_2, n_6, n_7), 18, 8\rangle$ *is popped from* $\mathcal{Q}$*. For landmark node* n_7 *we have* $L_p = 18 < d_N(n_s, n_t)$ *and* $L_p + d_N(n_7, n_t) = 22 < 1.3 \cdot L_p = 23.4$*. Following Proposition 4.3 an upper bound of the unknown time for the answer path is found,* $\overline{U} = U_p + U_{n_7} = 12$*. Thus, after the tenth iteration we have:*

$$\begin{aligned}
&\mathcal{Q} = \{\langle n_4, (n_s, n_1, n_2, n_3, n_4), 17, 14\rangle, \langle n_7, (n_s, n_1, n_2, n_6, n_7), 18, 8\rangle,\\
&\qquad \langle n_3, (n_s, n_1, n_5, n_2, n_3), 18, 9\rangle, \langle n_7, (n_s, n_1, n_5, n_2, n_6, n_7), 21, 5\rangle\}\\
&\mathcal{P}[n_s] = \{\langle 0, 0\rangle\},\ \mathcal{P}[n_1] = \{\langle 3, 0\rangle\},\ \mathcal{P}[n_2] = \{\langle 8, 5\rangle, \langle 11, 2\rangle\},\\
&\mathcal{P}[n_3] = \{\langle 15, 12\rangle, \langle 18, 9\rangle\},\ \mathcal{P}[n_4] = \{\langle 17, 14\rangle, \langle 19, 9\rangle\},\ \mathcal{P}[n_5] = \{\langle 9, 0\rangle\},\\
&\mathcal{P}[n_6] = \{\langle 11, 8\rangle, \langle 14, 5\rangle\},\ \mathcal{P}[n_7] = \{\langle 18, 8\rangle, \langle 21, 5\rangle\},\ \mathcal{P}[n_t] = \{\langle 20, 17\rangle\}\\
&\overline{L} = 20\\
&\overline{U} = 12
\end{aligned}$$

TRUSTME *proceeds similarly and at the thirteenth iteration label* $\langle n_t, (n_s, n_1, n_2, n_3, n_4, n_t), 20, 17\rangle$ *is popped from* $\mathcal{Q}$*. This means that the shortest path from source* n_s *to* n_t *is identified, and therefore,* $\overline{L} = d_N(n_s, n_t) = 20$*. On the other hand, the upper bound* $\overline{U}$ *is not improved since current* $\overline{U} = 12 < 17$*.*

Finally, at the seventeenth iteration, the algorithm empties $\mathcal{Q}$ *popping label* $\langle n_t, (n_s, n_1, n_5, n_2, n_6, n_7, n_4, n_t), 25, 9\rangle$*, and identifies answer path* $(n_s, n_1, n_5, n_2, n_6, n_7, n_4, n_t)$ *of length* 25 *and unknown time* 9*.*

4.5 Experimental Analysis

Section 4.5.1 details the setting of our analysis, while Section 4.5.2 compares our method for evaluating MTNSP queries against a rival method, called SP−EUCLIDEAN.

4.5.1 Setup

To conduct our experiments, we consider the road network of the city of San Joaquin County (TG) which contains 18,263 intersections and 23,874 road segments connecting them. Then, we generate a number of known graphs for the network graph of TG. The idea is the following.

Normally, i.e., if we exclude professionals like taxi drivers, people drive around specific locations on the road network of a city. For instance, they move around the location of their house, their work place, their children's school etc. In other words, a driver is familiar with the road segments on specific parts of the city, or simply with specific neighborhoods. To capture this behavior, we identify 50 neighborhoods on the TG road network employing a conventional clustering method that considers the euclidean distance of the road intersections. Next, we construct a set of known graphs varying the number of familiar neighborhoods $|H|$ from 3 to 30. For each value of $|H|$ we select the nodes of the familiar graph and generate random routes, i.e., sequence of nodes, to define the edges of the known graph. Finally, to enrich the diversity of the known graphs, we adopt three strategies for selecting their nodes:

(S1) All the nodes in the $|H|$ neighborhoods are contained in the known graph (Figure 4.3). In this case, the $|H|$ neighborhoods are selected such that they are situated at the same part of the city, e.g., at the south suburbs.

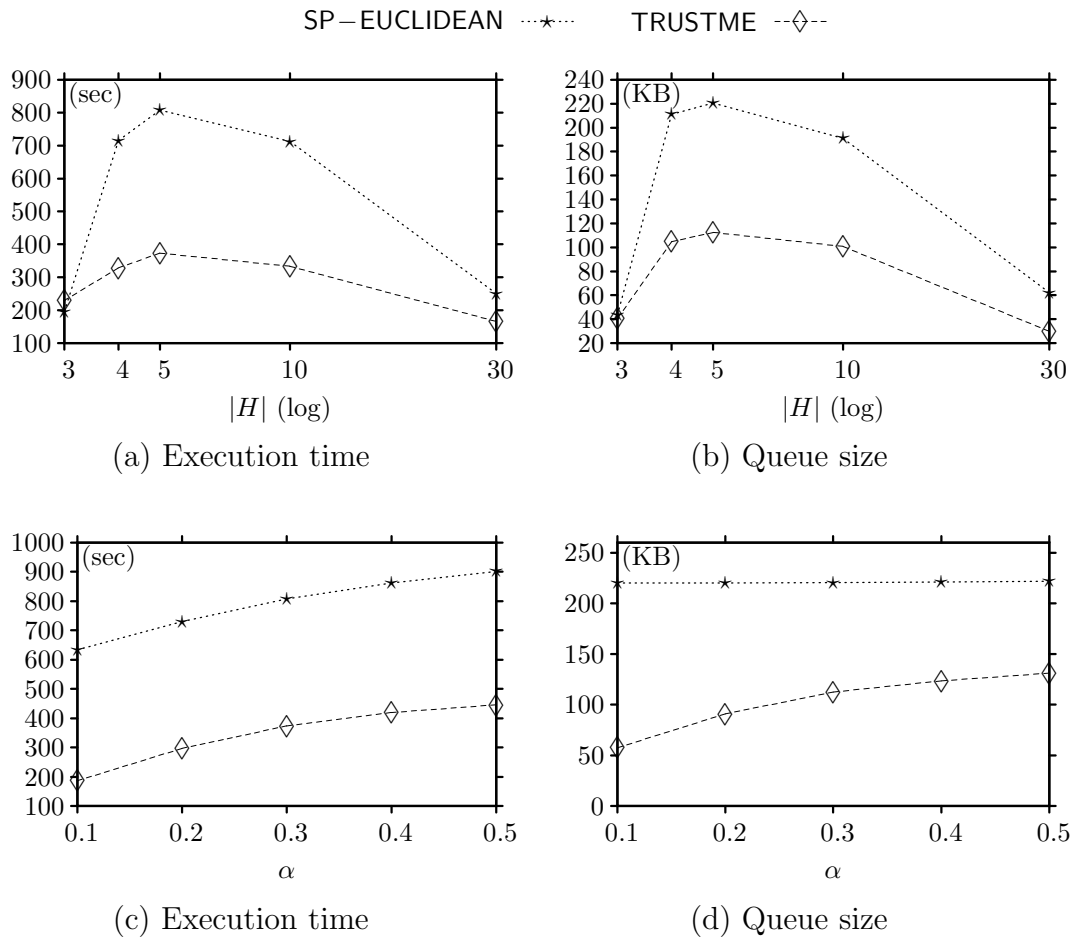
(S2) All the nodes included in the shortest paths between the centers of the $|H|$ neighborhoods are contained in the known graph (Figure 4.4). The center of a neighborhood is the closest intersection to the cluster centroid, w.r.t. the euclidean distance.

(S3) All the nodes in the $|H|$ neighborhoods and the nodes included in the shortest paths between the centers are contained in the known graph (Figure 4.5).

4.5.2 Experiments

To the best of our knowledge, this is the first work that deals with `MTNSP` queries. Therefore, there is no ready-to-use method coming from the literature.

As a simple and straightforward solution we could follow a brute force approach; implement a label-setting algorithm that constructs almost every path between the source n_s and the target node n_t, and finally, return the answer. Of course, after finding the shortest path, we would have both the network distance $d_N(n_s, n_t)$ and an upper bound for the unknown time of the answer path, and thus, we could discard non "promising" paths. However, without any estimation at least for the minimum time needed to reach the target, i.e., a lower bound of the L_p cost, this solution cannot be used in practice for real-world road networks like TG. To this end, we introduce two enhancements that significantly reduce the execution time, and devise the SP−EUCLIDEAN method. First, we consider the euclidean distance of any two nodes n_i and n_j in the network graph which provides a lower bound of their network distance $d_N(n_i, n_j)$. Yet, this lower bound can be used only after the shortest path and the network distance between the source and the target node of the query is computed. So, the idea for the second enhancement is to employ Dijkstra algorithm at the beginning of the SP−EUCLIDEAN method to compute $d_N(n_s, n_t)$. Although Dijkstra introduces an additional computational cost, we notice that the benefit



(a) Execution time (b) Queue size

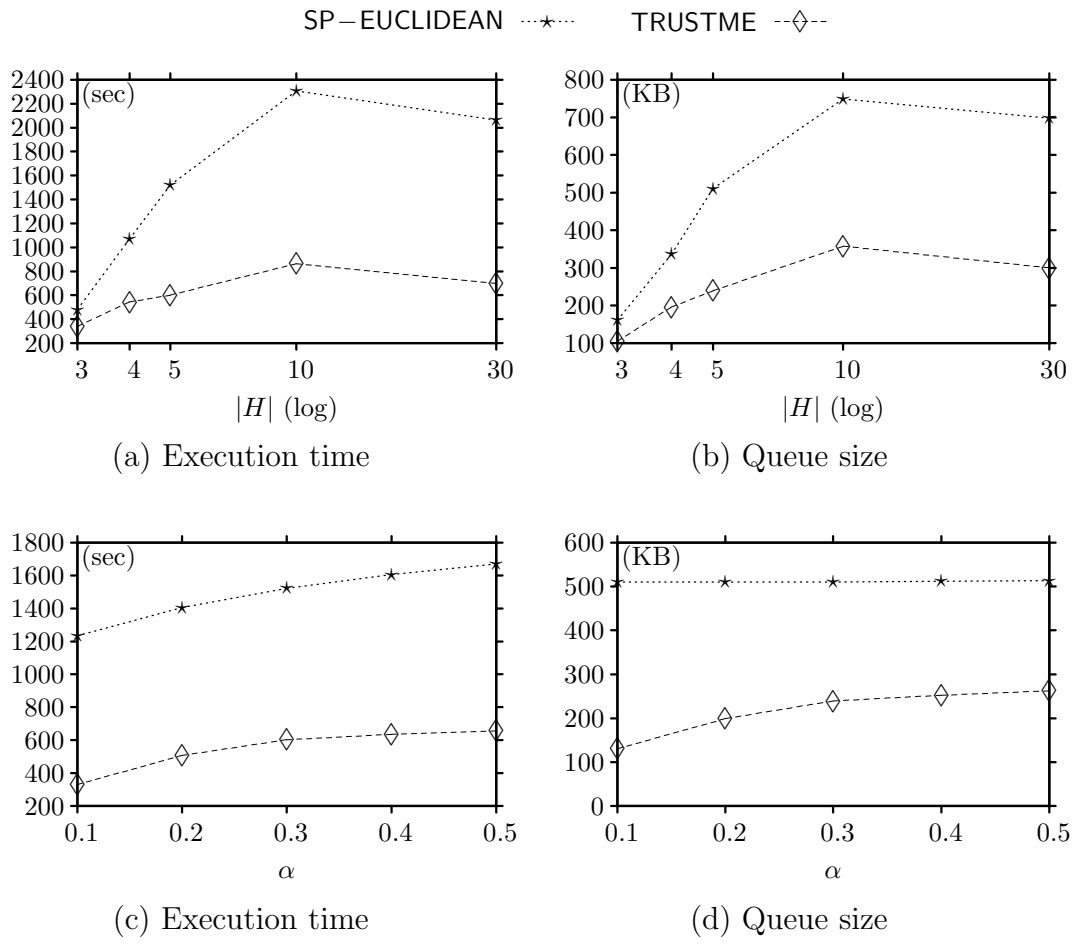
(c) Execution time (d) Queue size

Figure 4.3: *S1 strategy: all the nodes in the neighborhoods are contained in the known graph.*

from being able to prune non "promising" paths during the expensive search for the answer to the `MTNSP` query, is large enough to speed up the entire process.

We implement both TRUSTME and SP−EUCLIDEAN methods in C++ and compile them with gcc. The evaluation is performed on a 3 Ghz Intel Core 2 Duo CPU with 4GB RAM running Debian Linux. We generate $1,000$ random $\texttt{MTNSP}(n_s, n_t, \alpha)$ queries and employ TRUSTME and SP−EUCLIDEAN to compute the answer paths, varying at the same time the α factor from 1.1 to 1.5. For each method, we measure (1) the total time needed to answer a query (sub-figures (a) and (c)), and (2) the maximum size of the priority queue in KBs (sub-figures (b) and (d)). Note that when varying the number of familiar neighborhoods $|H|$ we retain factor α fixed at 1.3, and when varying α, $|H| = 5$. Finally, also note that we store both the network graph and its embeddings on disk.

Examining Figures 4.3, 4.4 and 4.5, we make the following observations. Method TRUSTME outperforms SP−EUCLIDEAN. In fact, on some datasets, the TRUSTME method is over two times faster than SP−EUCLIDEAN. Similarly, TRUSTME always needs less space in main memory to store the priority queue of the search, than SP−EUCLIDEAN. The advantage of TRUSTME over SP−EUCLIDEAN was expected due to the following two reasons. The first and most important reason is related to the way each method prunes the search space. SP−EUCLIDEAN discards a path



(a) Execution time (b) Queue size

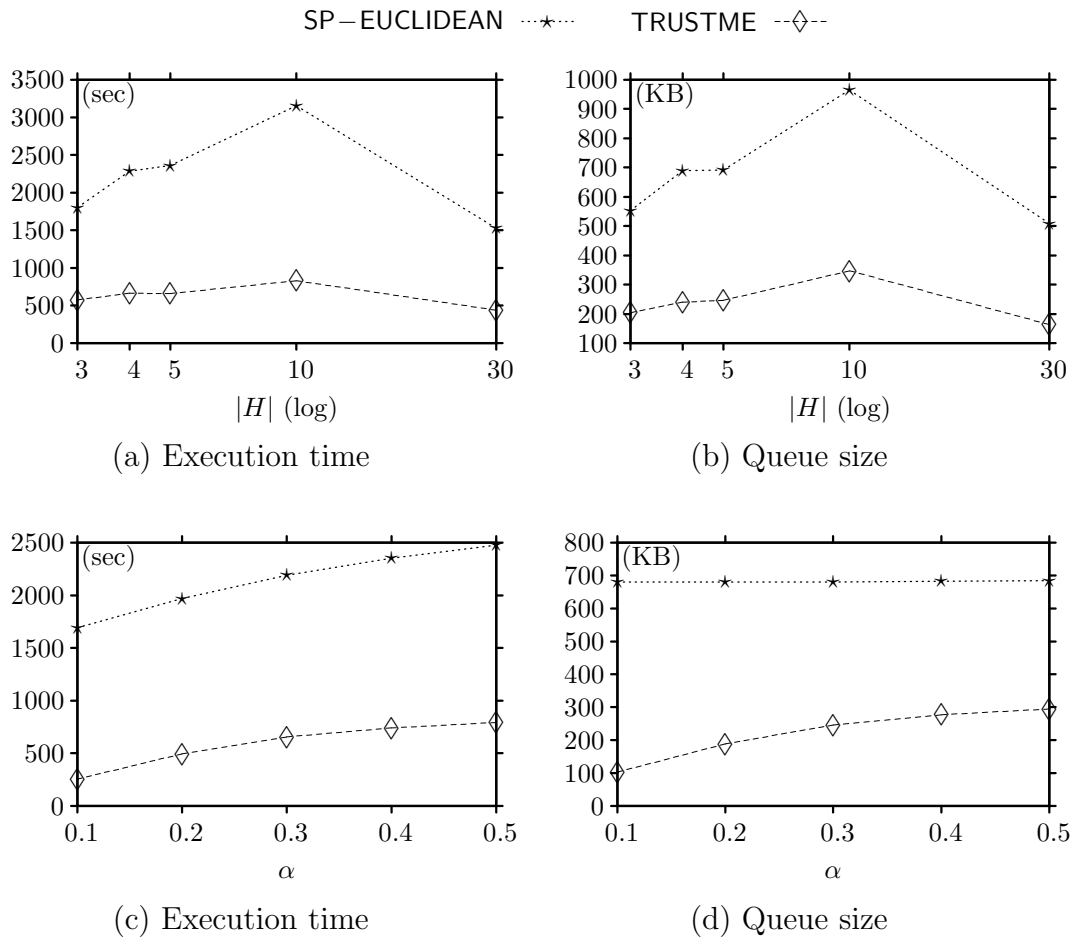
(c) Execution time (d) Queue size

Figure 4.4: *S2 strategy: all the nodes included in the shortest paths between the centers of the neighborhoods are contained in the known graph.*

only w.r.t. its length, i.e., using the lower bound of the time needed to reach the target n_t and the network distance $d_N(n_s, n_t)$. On the other hand, TRUSTME also employs the lower and the upper bound of the unknown time to reach n_t. Second, the embedding on a network graph gives a better estimation of the distance between two nodes compared to the euclidean distance of the nodes, especially as the number of familiar neighborhoods $|H|$ increases, and thus, the number of landmarks also increases.

Finally, we discuss in detail the behaviour of TRUSTME and SP−EUCLIDEAN methods when varying parameters $|H|$ and α.

Varying the number of familiar neighborhoods $|H|$. As a general observation for both the methods, we notice that as the number of familiar neighborhoods $|H|$ increases, initially, the execution time per query increases too. But, when becoming larger than a specific value of $|H|$, usually 5 or 10, the execution time goes down, regardless the strategy we have employed to construct the known graphs. In fact this is the case also for the priority queue size when varying $|H|$. The reason for this behaviour is the following. For small values of $|H|$ where the known subgraph is much smaller than the network graph, the search traverses mainly the unknown subgraph resembling to a shortest path search w.r.t. to the unknown time. In other words, given a node n most of the paths p reaching n from the source node n_s have



(a) Execution time (b) Queue size

(c) Execution time (d) Queue size

Figure 4.5: *S3 strategy: all the nodes in the neighborhoods and the nodes included in the shortest paths between the centers are contained in the known graph.*

$U_p \approx L_p$, and therefore, they can be easily discarded exploiting the lower bound estimation of L_p and the network distance $d_N(n_s, n_t)$ (resp. SP−EUCLIDEAN) or the upper bound $\overline{d_N}(n_s, n_t)$ (resp. TRUSTME). As $|H|$ increases, more paths pass through the known subgraph. Thus, there exist a lot of "promising" paths that must be extended forcing both methods to perform more iterations. On the other hand, for large values of $|H|$, i.e., usually larger than 10, the methods are forced to traverse mainly the known subgraph, which covers a large part of the network graph. In this case the search resembles to a shortest path w.r.t. to the time spend on the known subgraph.

Finally, we also notice that when employing the S1 (Figure 4.3) or the S2 strategy (Figure 4.4) for selecting the nodes in the known graphs, and with $|H| = 3$, SP−EUCLIDEAN is nearly as efficient as TRUSTME for answering a `MTNSP` query. In both these cases, the known graph contains very few nodes which means the embedding of the network graph cannot provide good estimations (lower and upper bounds) of the network distance and the unknown time.

Varying the factor α. As factor α increases we notice that both TRUSTME and SP−EUCLIDEAN require more time to answer a `MTNSP` query. This is expected since the constraint regarding the length of the answer path compared to the length of the shortest path becomes less strict and therefore, there exist more "promising" paths

to be extended during the search. Finally, TRUSTME requires more space in main memory for its search queue as α increases, while the size of SP−EUCLIDEAN's queue remains the same. This is because for SP−EUCLIDEAN the largest queue size is witnessed during the execution of the Dijkstra algorithm and not during the search of the answer path to the MTNSP query. Remember that the computational cost of Dijkstra only depends only on the size of the network graph and not on α

4.6 Conclusions

In this chapter, we introduced the Most Trusted Near Shortest Path (MTNSP) as a preferable way of driving through a city when a collection of trusted routes is available. For this purpose, we first defined the notion of the known graph as a subgraph of the road network that is constructed merging the available trusted routes. Then, we defined MTNSP as the problem of identifying the path p on the road network with the lowest time spend outside the known graph among the paths with length, at most α times larger than the length of the shortest path. Finally, we proposed a methodology for identifying MTNSP that involves the offline construction of an embedding on the road network, and a label-setting algorithm, called TRUSTME, that employs the embedding in order to speedup the evaluation of the MTNSP queries. Our experimental analysis shows that our method is able to find MTNSP faster than a label-setting algorithm that exploits the euclidean distance of the network intersections to prune its search space.

Chapter 5

Conclusions and Future Work

This thesis presents a framework for the evaluation of path queries over route collections that are frequently updated. The framework involves a set of algorithms for query evaluation and a set of indexing schemes on the routes. In addition, appropriate updating procedures for these schemes are introduced.

5.1 Summary

Initially, we targeted path query evaluation on large disk resident route collections like the ones containing touristic routes. Such collections are frequently updated since new routes are added or existing ones are deleted. Given two locations n_s and n_t the path query, denoted by `PATH`, returns a sequence of locations contained solely on the existing routes of the collection. We introduced the route and the link traversal evaluation paradigms that enjoy the benefits of search algorithms (i.e., fast index maintenance) while utilizing transitivity information to terminate the search sooner. In addition, we introduced the $\mathcal{R}$-$Index$ and the $\mathcal{T}$-$Index$ of a route collection and presented appropriate updating procedures. The proposed framework, i.e., the indices and the traversal policies, constitutes the basis for applying our work to other types of queries under various constraints. An extensive experimental evaluation verified the advantages of our methods compared to conventional graph-based search.

Next, we studied the problem of dynamic Pickup and Delivery with Transfers (`dPDPT`). To the best of our knowledge this is the first work that targets the dynamic version of this problem and in addition, the first time such a problem is formulated as a path query. For this purpose, we introduced the dynamic plan graph that captures all possible actions for picking up an object and delivering it to the destination with respect to the existing vehicle routes. Then, we defined the operational and the customer cost of a path, that capture both the company's and the customer's viewpoints of the problem, respectively, and proposed algorithms SP and SPM that identify the solution computing the shortest path on the dynamic plan graph with respect to these costs. An extensive experimental analysis demonstrated that our method is significantly faster than a two-phase local search method inspired by the related work, while the quality of the solution is only marginally lower.

Finally, we introduced the Most Trusted Near Shortest Path (`MTNSP`) as a preferable way of driving through a city when a collection of trusted vehicle routes is available. For this purpose, we defined the known subgraph of the road network that

represents the familiar and trusted part of the network. Then, for a path between two network locations, we defined its length and its unknown time that measure the total traveling time needed and the total time spent outside the known graph, respectively. We proposed a methodology that identifies the path having the lowest unknown time among the paths with length, at most α times larger than the length of the shortest path, as the answer to a `MTNSP` query. An extensive experimental analysis showed the advantage of our methodology compared to a label-setting algorithm that exploits the euclidean distance of the network intersections to prune its search space.

5.2 Future Work

During the course of this dissertation, we have identified the following interesting aspects that we propose as future work:

- In the context of touristic route collections, we plan to extend our work to evaluate queries similar to trip planning [48] and optimal sequenced route [67] queries. Specifically, consider a set of classes C such that each location in a route is an instance of a class in C. For example a location is an instance of classes $C = \{Museum, Stadium, Restaurant\}$. An interesting query is to find a path between two given locations that passes first through a $Museum$, then a $Stadium$ and finally a $Restaurant$.

- Another challenge in the context of touristic routes is to combine query evaluation with keyword search. For example, instead of specific locations, the source and the target of a query could be given as a set of keywords, or in the query discussed in the previous paragraph, we could seek for the path that passes through a Restaurant with a description relevant to "sea food, lobster" keywords.

- In the context of the dynamic Pickup and Delivery problem with Transfers, an interesting challenge is to adopt ideas from the evaluation framework presented in Section 2 for `PATH` queries. Recall that the SP and the SPM algorithms identify a candidate solution only when they reach a vertex of graph that has a delivery edge. However, exploiting an $\mathcal{R}$-$Index$ or a $\mathcal{T}$-$Index$ on the vehicle routes of the collection, we could identify a candidate solution earlier in the search, and thus, boost the satisfaction of a `dPDPT` request.

- Another direction for `dPDPT` is to consider additional constraints like the vehicle capacity or the existence of predefined time windows. As an example for the latter case, a customer requests that the object pickup and delivery takes place within specific time periods of the day.

- For the problem of Most Trusted Near Shortest Path, we plan to extend the scheme computed during the offline processing phase such that we can efficiently handle updates like additions or deletions of routes. The intuition is the following. Adding or removing routes affects only the known graph while the network graph remains unchanged. In practice, the changes in the known graph involve adding or deleting edges. Thus, for every pair of a graph node

n_i and a landmark n_{ℓ_j}, we have to update the unknown time U_p for all shortest paths $p(n_i, \ldots, n_{\ell_j})$ precomputed for the embedding scheme, and determine, if needed, which is the shortest path $(n_i, \ldots, n_{\ell_j})$ with the lowest unknown time. For this purpose, we could employ an inverted index on the precomputed shortest paths such that when an edge is added to or deleted from the known graph, we can easily identify the shortest paths that involve this edge and update their unknown time.

- Finally, for both the `dPDPT` and the `MTNSP` problems we plan to conduct a detailed complexity analysis similar to `PATH`.

Bibliography

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish, *Efficient management of transitive relationships in large data and knowledge bases*, Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1989, pp. 253–262.

[2] Rakesh Agrawal and H. V. Jagadish, *Direct algorithms for computing the transitive closure of database relations*, Proceedings of the International Conference on Very Large Data Bases (VLDB), 1987, pp. 255–266.

[3] ———, *Materialization and incremental update of path information*, Proceedings of the IEEE International Conference on Data Engineering (ICDE), 1989, pp. 374–383.

[4] Russell Bent and Pascal Van Hentenryck, *A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows*, Computers and Operations Research **33** (2006), no. 4, 875–893.

[5] Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte, *Dynamic pickup and delivery problems*, European Journal of Operational Research **202** (2010), no. 1, 8–15.

[6] Gerardo Berbeglia, Jean-Francois Cordeau, Irina Gribkovskaia, and Gilbert Laporte, *Static pickup and delivery problems: a classification scheme and survey*, TOP **15** (2007), 1–31.

[7] J. Bourgain, *On lipschitz embedding of finite metric spaces in hilbert space*, Israel Journal of Mathematics **52** (1985), no. 1, 46–52.

[8] Panagiotis Bouros, Theodore Dalamagas, Spiros Skiadopoulos, and Timos K. Sellis, *Evaluating "find a path" reachability queries*, Proceedings of the European Conference on Artificial Intelligence (ECAI) Workshop on Spatial and Temporal Reasoning (Patras, Greece), July 22 2008.

[9] Panagiotis Bouros and et al., *Efficient dynamic pickup and delivery with transfer*, Technical report, KDBS Lab, NTU Athens, 2011.

[10] ———, *Most trusted near shortest path*, Technical report, KDBS Lab, NTU Athens, 2011.

[11] Panagiotis Bouros, Dimitris Sacharidis, Theodore Dalamagas, and Timos K. Sellis, *Dynamic pickup and delivery with transfer*, Proceedings of the Symposium on Spatial and Temporal Databases (SSTD), 2011.

[12] Panagiotis Bouros, Dimitris Sacharidis, Theodore Dalamagas, Spiros Skiadopoulos, and Timos K. Sellis, *Evaluating path queries over frequently updated route collections*, IEEE Transactions on Knowledge and Data Engineering (TKDE) **99** (2011), no. PrePrints.

[13] Panagiotis Bouros, Spiros Skiadopoulos, Theodore Dalamagas, Dimitris Sacharidis, and Timos K. Sellis, *Evaluating reachability queries over path collections*, Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), 2009, pp. 398–416.

[14] Panagiotis Bouros and Yannis Vassiliou, *Evaluating path queries over route collections*, Proceedings of the IEEE International Conference on Data Engineering (ICDE) PhD Workshop, 2010, pp. 325–328.

[15] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng, *On incremental maintenance of 2-hop labeling of graphs*, Proceedings of the International World Wide Web Conferences (WWW), 2008, pp. 845–854.

[16] Thomas H. Byers and Michael S. Waterman, *Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming*, Operations Research **32** (1984), 1381–1384.

[17] W. Matthew Carlyle and R. Kevin Wood, *Near-shortest and k-shortest simple paths*, Networks **46** (2005), no. 2, 98–109.

[18] Robert L. Carraway, Thomas L. Morin, and Herbert Moskowitz, *Generalized dynamic programming for multicriteria optimization*, European Journal of Operational Research **44** (1990), no. 1, 95–104.

[19] Jiefeng Cheng and Jeffrey Xu Yu, *On-line exact shortest distance query processing*, Proceedings of the International Conference on Extending Database Technology (EDBT), 2009, pp. 481–492.

[20] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu, *Fast computation of reachability labeling for large graphs*, Proceedings of the International Conference on Extending Database Technology (EDBT), 2006, pp. 961–979.

[21] ———, *Fast computing reachability labelings for large graphs with high compression rate*, Proceedings of the International Conference on Extending Database Technology (EDBT), 2008, pp. 193–204.

[22] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick, *Reachability and distance queries via 2-hop labels*, Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 937–946.

[23] ———, *Reachability and distance queries via 2-hop labels*, SIAM Journal on Computing **32** (2003), no. 5, 1338–1355.

[24] Kenneth L. Cooke and Eric Halsey, *The shortest route through a network with time-dependent internodal transit times*, Journal of Mathematical Analysis and Applications **14** (1966), no. 3, 493–498.

[25] Jean-François Cordeau, Gilbert Laporte, and Stefan Ropke, *Vehicle routing: Latest advances and challenges*, ch. Recent Models and Algorithms for One-to-One Pickup and Delivery Problems, pp. 327–357, Kluwer, 2008.

[26] Jean-Francois Cordeau, *A branch-and-cut algorithm for the dial-a-ride problem*, Operations Research **54** (2006), no. 3, 573–586.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, second edition*, The MIT Press and McGraw-Hill Book Company, 2001.

[28] Cristián E. Cortés, Martín Matamala, and Claudio Contardo, *The pickup and delivery problem with transfers: Formulation and a branch-and-cut solution method*, European Journal of Operational Research **200** (2010), no. 3, 711–724.

[29] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris, *Vivaldi: a decentralized network coordinate system*, Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer communications (SIGCOMM), 2004, pp. 15–26.

[30] Bolin Ding, Jeffrey Xu Yu, and Lu Qin, *Finding time-dependent shortest paths over large graphs*, Proceedings of the International Conference on Extending Database Technology (EDBT), 2008, pp. 205–216.

[31] Stuart E. Dreyfus, *An appraisal of some shortest-path algorithms*, Operations Research **17** (1969), no. 3.

[32] Yvan Dumas, Jacques Desrosiers, and Francois Soumis, *The pickup and delivery problem with time windows*, European Journal of Operational Research **54** (1991), no. 1, 7–22.

[33] Michel Gendreau, Francois Guertin, Jean-Yves Potvin, and René Séguin, *Neighbourhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries*, Transportation Research Part C: Emerging Technologies **14** (2006), no. 3, 157 – 174.

[34] Andrew V. Goldberg and Chris Harrelson, *Computing the shortest path: A search meets graph theory*, Proceedings of the 16th annual ACM-SIAM symposium on Discrete algorithms (SODA), 2005, pp. 156–165.

[35] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck, *Reach for A*: Efficient point-to-point shortest path algorithms*, Proceedings of the 8th WS on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia, 2006, pp. 129–143.

[36] Janusz Granat and Francesca Guerriero, *The interactive analysis of the multicriteria shortest path problem by the reference point method*, European Journal of Operational Research **151** (2003), no. 1, 103–118.

[37] F. Guerriero and R. Musmanno, *Label correcting methods to solve multicriteria shortest path problems*, Journal of Optimization Theory and Applications **111** (2001), no. 3, 589–613.

[38] Gísli R. Hjaltason and Hanan Samet, *Properties of embedding methods for similarity searching in metric spaces*, IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI) **25** (2003), no. 5, 530–549.

[39] Yannis E. Ioannidis and Raghu Ramakrishnan, *Efficient transitive closure algorithms*, Proceedings of the International Conference on Very Large Data Bases (VLDB), 1988, pp. 382–394.

[40] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry, *3-hop: a high-compression indexing scheme for reachability query*, Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2009, pp. 813–826.

[41] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang, *Efficiently answering reachability queries on very large directed graphs*, Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2008, pp. 595–608.

[42] Ning Jing, Yun-Wu Huang, and Elke A. Rundensteiner, *Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation*, IEEE Transactions on Knowledge and Data Engineering (TKDE) **10** (1998), no. 3, 409–432.

[43] David S. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences **9** (1974), no. 3, 256–278.

[44] Sungwon Jung and Sakti Pramanik, *An efficient path computation model for hierarchically structured topographical road maps*, IEEE Transactions on Knowledge and Data Engineering (TKDE) **14** (2002), no. 5, 1029–1046.

[45] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Matthias Renz, and Tim Schmidt, *Proximity queries in large traffic networks*, Proceedings of the ACM International Symposium on Geographic Information Systems (GIS), 2007, p. 21.

[46] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Tim Schmidt, *Hierarchical graph embedding for efficient query processing in very large traffic networks*, Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), 2008, pp. 150–167.

[47] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert, *Route skyline queries: A multi-preference path planning approach*, Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2010, pp. 261–272.

[48] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng, *On trip planning queries in spatial databases*, Proceedings of the Symposium on Spatial and Temporal Databases (SSTD), 2005.

[49] Haibing Li and Andrew Lim, *A metaheuristic for the pickup and delivery problem with time windows*, Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2001, pp. 333–340.

[50] Hongjun Lu, *New strategies for computing the transitive closure of a database relation*, Proceedings of the International Conference on Very Large Data Bases (VLDB), 1987, pp. 267–274.

[51] Snezana Mitrović-Minić, R. Krishnamurti, and Gilbert Laporte, *Double-horizon based heuristics for the dynamic pickup and delivery problem*, Transportation Research Part B: Methodological **38** (2004), no. 7, 669–685.

[52] Snezana Mitrović-Minić and Gilbert Laporte, *Waiting strategies for the dynamic pickup and delivery problem with time windows*, Transportation Research Part B: Methodological **38** (2004), no. 7, 635–655.

[53] ———, *The pickup and delivery problem with time windows and transshipment*, INFOR **44** (2006), no. 3, 217–228.

[54] T. S. Eugene Ng and Hui Zhang, *Predicting internet network distance with coordinates-based approaches*, Proceedings of the IEEE International Conference on Computer Communications (INFOCOM), 2002.

[55] Ariel Orda and Raphael Rom, *Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length*, Journal of the ACM **37** (1990), no. 3, 607–625.

[56] Sophie Parragh, Karl Doerner, and Richard Hartl, *A survey on pickup and delivery problems, Part II: Transportation between pickup and delivery locations*, Journal für Betriebswirtschaft **58** (2008), 81–117.

[57] I Pohl, *Bi-directional search*, Machine Intelligence **6** (1971), 127–140.

[58] Douglas A. Popken, *Controlling order circuity in pickup and delivery problems*, Transportation Research Part E: Logistics and Transportation Review **42** (2006), no. 5, 431–443.

[59] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis, *Fast shortest path distance estimation in large networks*, Proceedings of the International Conference on Information and Knowledge Management (CIKM), 2009, pp. 867–876.

[60] Matthew J. Rattigan, Marc Maier, and David Jensen, *Using structure indices for efficient approximation of network properties*, Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining, Washington (SIGKDD), 2006, pp. 357–366.

[61] Stefan Ropke and Jean-François Cordeau, *Branch and cut and price for the pickup and delivery problem with time windows*, Transportation Science **43** (2009), no. 3, 267–286.

[62] Stefan Ropke, Jean-François Cordeau, and Gilbert Laporte, *Models and branch-and-cut algorithms for pickup and delivery problems with time windows*, Networks **49** (2007), no. 4, 258–272.

[63] Stefan Ropke and David Pisinger, *An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows*, Transportation Science **40** (2006), no. 4, 455–472.

[64] Ralf Schenkel, Anja Theobald, and Gerhard Weikum, *Hopi: An efficient connection index for complex xml document collections*, Proceedings of the International Conference on Extending Database Technology (EDBT), 2004, pp. 237–255.

[65] ———, *Efficient creation and incremental maintenance of the hopi index for complex xml document collections*, Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2005, pp. 360–371.

[66] Cyrus Shahabi, Mohammad R. Kolahdouzan, and Mehdi Sharifzadeh, *A road network embedding technique for k-nearest neighbor search in moving object databases*, GeoInformatica **7** (2003), no. 3, 255–273.

[67] Mehdi Sharifzadeh, Mohammad R. Kolahdouzan, and Cyrus Shahabi, *The optimal sequenced route query*, VLDB Journal **17** (2008), no. 4.

[68] M. Sigurd, D. Pisinger, and M. Sig, *Scheduling transportation of live animals to avoid spread of diseases*, INFORMS transportation science **38** (2004), 197–209.

[69] Liying Tang and Mark Crovella, *Virtual landmarks for the internet*, Proceedings of the Internet Measurement Conference, 2003, pp. 143–152.

[70] Yuan Tian, Ken C. K. Lee, and Wang-Chien Lee, *Finding skyline paths in road networks*, Proceedings of the ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL GIS), 2009, pp. 444–447.

[71] Silke Trißl and Ulf Leser, *Fast and practical indexing and querying of very large graphs*, Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2007, pp. 845–856.

[72] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu, *Dual labeling: Answering graph reachability queries in constant time*, Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2006, p. 75.

[73] Hang Xu, Zhi-Long Chen, Srinivas Rajagopal, and Sundar Arunapuram, *Solving a practical pickup and delivery problem*, Transportation Science **37** (2003), no. 3, 347–364.

[74] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma, *Mining interesting locations and travel sequences from gps trajectories*, Proceedings of the International World Wide Web Conferences (WWW), 2009, pp. 791–800.

[75] Justin Zobel and Alistair Moffat, *Inverted files for text search engines*, ACM Computing Surveys (CSUR) **38** (2006), no. 2.

Chapter 6

Curriculum Vitae

Contact Information

Knowledge and Database Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
9 Iroon Polytechniou, Polytechnioupoli Zographou
157 80 Athens, Greece

Telephone: (+30) 210 772 1602
Fax: (+30) 210 772 1442
E-mail: pbour@dblab.ece.ntua.gr
Homepage: http://www.dblab.ece.ntua.gr/~pbour

Education

- **National Technical University of Athens**, Greece (2005–2011)
  PhD, School of Electrical and Computer Engineering
  Title: Evaluating Queries over Route Collections
  Supervisor: Prof. Yannis Vassiliou

- **National Technical University of Athens**, Greece (1998–2003)
  Diploma, School of Electrical and Computer Engineering
  Grade: 7.60/10
  Diploma Thesis: Query Language for Hierarchical Structures
  Supervisor: Prof. Timos Sellis

Research Interests

- Query evaluation on spatio-temporal databases
- Personalization of topic directories
- Indexing set valued attributes and containment queries

Distinctions

- **Greek State Scholarships Foundation (IKY)** (2006-2010)
  Field: Artificial Intelligence and Applications

- **Institute for Language and Speech Processing (ILSP) Scholarship** (2001-2005)
  Department of Electronic Lexicography

Academic Experience

- **National Technical University of Athens**, Greece (2005-2009)
  Teaching Assistant

  - Database Systems (Fall 2009)
  - Database Systems (Fall 2008)
  - Database Systems (Fall 2007)
  - Database Systems (Fall 2006)
  - Introduction to Programming - PASCAL laboratories (Fall 2005)

  Diploma Thesis co-Supervisor

  - *D. Kontogianni*, **GRAPHIT-DB: Graph Data Management System II** (2009)
  - *V. Giannopoulos*, **P-Miner+: Portal Catalogs Administration Supporting Usage Data Mining Processes** (2009)
  - *I. Liagouris & T. Farmakakis*, **, DataBase supported Reasoning System (DBRS)** (2008)
  - *L. Boula*, **GRAPHIT-DB: Graph Data Management System** (2007)
  - *T. Galanis*, **P-Miner: Portal Catalogs Administration Supporting Usage Data Mining Processes** (2006)

- **National Kapodistrian University of Athens**, Greece (2005)
  Teaching Assistant

  - Software Engineering (Fall 2005)
  - System Analysis (Spring 2005)

Academic Projects

- **National Technical University of Athens**, Greece

  - **Advance Semantic Web Technologies to Support Ontology-based Enterprise Content Management** (2006)
    Greece – China, Joint Research and Technology Program, EPAN, GSRT

- **National Kapodistrian University of Athens**, Greece

– **SODIUM** (2005)
Studying Semantic Web Services initiatives: OWL-S, WSMO, METEOR-S
Web Services development considering the OWL-S Framework
Development in Java of an OWL-S handler and QoS handler based on the WS-QoS framework

– **INTEROP NoE** (2005)
Creation of partners knowledge map using Protégé

External Referee

- SIGMOD, CIKM, SSTD, TPDL (2011)
- ICDE (2010)
- VLDB, EDBT, CIKM, BPM (2009)
- JIR (2008)
- PCI, PersDL (2007)
- PSI, DaWaK, HDMS (2006)
- DEXA, EC-Web, I3E, ICWS (2005)

Programme Committe

- CIKM Posters (2011)

Publications

1. Panagiotis Bouros, Dimitris Sacharidis, Theodore Dalamagas, Spiros Skiadopoulos and Timos Sellis, **Evaluating Path Queries over Frequently Updated Route Collections**, IEEE Transactions on Knowledge and Data Engineering (TKDE). (to appear)

2. Panagiotis Bouros, Dimitris Sacharidis, Theodore Dalamagas and Timos Sellis, **Dynamic Pickup and Delivery with Transfers**, in Proceedings of the 12th International Symposium on Spatial and Temporal Databases (SSTD'11), Minneapolis, MN, USA, August 24-26, 2011.

3. Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis, Timos Sellis and Nikos Mamoulis, **Efficient Answering of Set Containment Queries for Skewed Item Distributions**, in Proceedings of the 14th International Conference on Extending Database Technology (EDBT'11), Uppsala, Sweden, March 21-25, 2011.

4. Panagiotis Bouros and Yannis Vassiliou, **Evaluating Path Queries over Route Collections**, in Proceedings of the PhD Workshop in conjunction with the 26th IEEE International Conference on Data Engineering (ICDE'10), Long Beach, California, USA, March 5, 2010.

5. Panagiotis Bouros, Spiros Skiadopoulos, Theodore Dalamagas, Dimitris Sacharidis and Timos Sellis, **Evaluating reachability queries over path collections**, in Proceedings of the 21st Proceedings International Conference on Scientific and Statistical Database Management (SSDBM'09), New Orleans, Louisiana USA, June 2-9, 2009.

6. Panagiotis Bouros, Theodore Dalamagas, Spiros Skiadopoulos and Timos Sellis, **Evaluating "Find a Path" Reachability Queries**, in Proceedings of the Workshop on Spatial and Temporal Reasoning in conjunction with the 18th European Conference on Artificial Intelligence (ECAI'08), Patras, Greece, July 22, 2008.

7. Dimitris Sacharidis, Panagiotis Bouros, and Timos Sellis, **Caching Dynamic Skyline Queries**, in Proceedings of the 20th Proceedings International Conference on Scientific and Statistical Database Management (SSDBM'08), Hong Kong, China, July 9-11, 2008.

8. Theodore Dalamagas, Panagiotis Bouros, Theodore Galanis, Magdalini Eirinaki and Timos Sellis, **Mining User Navigation Patterns for Personalizing Topic Directories**, in Proceedings of the 9th ACM International Workshop on Web Information and Data Management (WIDM'07) in conjunction with the 16th ACM International Conference on Information and Knowledge Management (CIKM'07), Lisbon, Portugal, November 9, 2007.

9. Panagiotis Bouros, Aggeliki Fotopoulou and Nicholas Glaros, **An interactive environment for creating and validating syntactic rules**, in Proceedings of the 5th International Conference on Recent Advances in Natural Language Processing (RANLP'05), Borovets, Bulgaria, September 21-23, 2005.

10. Aggeliki Koukoutsaki, Theodore Dalamagas, Timos Sellis and Panagiotis Bouros, **PatMan: A Visual Database System to Manipulate Path Patterns and Data in Hierarhical Catalogs**, in Proceedings of the International Workshop of the EU Network of Excellence DELOS on Audio-Visual Content and Information Visualization in Digital Libraries (AVIVDiLib'05), Cortona, Italy, May 4-6, 2005.

11. Panagiotis Bouros, Theodore Dalamagas, Timos Sellis and Manolis Terrovitis, **PatManQL: A language to manipulate patterns and data in hierarchical catalogs**, in Proceedings of the 1st International Workshop on Pattern Representation and Management (PaRMa'04) in conjunction with the 9th International Conference on Extending Database Technology (EDBT'04), Heraklion, Crete, Greece, March 18, 2004.

Presentations

1. Panagiotis Bouros, Dimitris Sacharidis, Theodore Dalamagas and Timos Sellis, **Dynamic Pickup and Delivery with Transfers**, presented in the 10th Hellenic Data Management Symposium, (HDMS'11), Athens, Greece, June 17-18, 2011.

2. Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis, Timos Sellis and Nikos Mamoulis, **Efficient Answering of Set Containment Queries for Skewed Item Distributions**, presented in the 10th Hellenic Data Management Symposium, (HDMS'11), Athens, Greece, June 17-18, 2011.

3. Panagiotis Bouros, Spiros Skiadopoulos, Theodore Dalamagas, Dimitris Sacharidis and Timos Sellis, **Evaluating reachability queries over path collections**, in 8th Hellenic Data Management Symposium (HDMS'09), Athens, Greece, August, 2009.

4. Dimitris Sacharidis, Panagiotis Bouros and Timos Sellis, **Caching Dynamic Skyline Queries**, presented in the 7th Hellenic Data Management Symposium (HDMS'08), Heraklion, Greece, July 2008.

Professional Experience

- **Biovista Knowledge Management for Life Sciences**, Greece (2006)
  Software engineer - Database optimization
  Performance enhancement of BEA search engine core, MySQL server tuning,database and query analysis and optimization
  Implementation of an Inverted File based on Berkeley DB
  (MySQL, Berkeley DB, C/C++)

- **Institute for Language and Speech Processing (ILSP)**, Greece (2001-2005)
  Software engineer - Department of Electronic Lexicography
  Symfonia, advanced spelling checker
  Rules for syntactic analysis of human text
  Environment for creating and validating syntactic rules of Symfonia
  (C++, Visual Basic, SQL, Programming on Word, DLL creation, DAO, ODBC, ACCESS, PrimeBase)

- **iLang Language Innovation**, Greece (2004-2005)
  Software engineer
  Development of an electronic edition for a thesaurus
  (Visual Basic, XML, XSLT, MS Access)

Technical Skills

- **Programming:**: C/C++, Java

- **Operating Systems**: Linux, Unix, MacOS, iOS, Windows

- **Database Systems**: MySQL, PostgreSQL, Berkeley DB, MS SQL Server, Oracle, PrimeBase

Foreign Languages

- English (Cambridge Proficiency)

- French (Delf)

Hobbies

Traveling, cinema, music, guitar, basketball