Evaluating Path Queries over Route Collections

Panagiotis Bouros [1]
supervised by Yannis Vassiliou [2]

School Of Electrical and Computer Engineering,
National Technical University of Athens, Greece
[1]pbour@dblab.ece.ntua.gr
[2]yv@dblab.ece.ntua.gr

***Abstract*— Nowadays, vast amount of routing data, like sequences of points of interests, landmarks, etc., are available due to the proliferation of geodata services. We refer to these sequences as *routes* and the involved points simply as *nodes*. In this thesis, we consider the problem of evaluating *path queries* on frequently updated route collections. We present our current work for two path queries: (i) identifying a path between two nodes of the collection, and (ii) identifying a constrained shortest path. Finally, some interesting open problems are described and our future work directions are clearly stated.**

I. Introduction

The recent advances in geodata services have resulted in the abundance of user or machine generated routing data, i.e., sequences of points of interests (POI), landmarks etc. We refer to these sequences as *routes* and the involved points as *nodes*. For example, consider people visiting Athens that have GPS-enabled devices to track their sightseeing and to create routes through interesting places. Web sites such as ShareMyRoutes.com maintain a huge collection of such routes, with POIs from all over the world. Note that in applications like the above, the routes are not necessarily formed over an existing graph. For instance people may create walking routes inside a city park. In contrast, there exist applications where the routes are constructed based on an existing graph, e.g., a road network. For instance, a supplier/delivery service for picking-up and delivering parcels creates and maintains every day a vehicle route collection to satisfy customer requests.

Given the availability of large route collections, the interesting problem of evaluating various types of graph-based queries directly on the collections arises. We present our current work for two of them. In both queries, we are looking for *paths* that connect nodes in route collections. A *path* may contain nodes from different routes of the collection, since reaching a target node may require changing routes using *links*, i.e., nodes shared among routes. The first query, denoted by $\texttt{PATH}(n_s, n_t)$, identifies a path in the route collection from node n_s to n_t. For the second query, called Fewer-Links Shortest Path (FLSP), we consider that each node in a route of the collection is assigned a time interval. In addition, we consider the cost c for changing between two routes, and the cost m for moving among nodes within the routes. In this setting, $\texttt{FLSP}(n_s, I_s, n_t, I_t)$ query identifies the path from n_s to n_t under the temporal constraints introduced by the time intervals that minimizes the cost c of changing routes and the moving cost m within the routes. Note that if two paths have the same changing cost c, we prefer the one with the lowest moving cost m.

The author is supported by the Greek State Scholarships Foundation (IKY).

In the example of touristic routes, given two nodes representing POIs, n_s and n_t, $\texttt{PATH}(n_s, n_t)$ query identifies a sequence of interesting places that connects n_s with n_t. In the context of delivery services, FLSP query is related to the dynamic pickup and delivery problem with time windows [1]. A vehicle v that follows a route r passes through nodes of the collection, i.e., landmarks in a city, to pick up or deliver parcels within a specific time interval. In contrast with most of the existing work, we assume that a parcel may be picked up and delivered by different vehicles. Specifically, two vehicles v_1 and v_2 may exchange a parcel if their routes r_1 and r_2 respectively have a shared node n, i.e., a link, and the time intervals for which vehicles v_1 and v_2 are in n, match. In this context, given a new customer request, $\texttt{FLSP}(n_s, I_s, n_t, I_t)$ query identifies a way to pick up the new parcel from node n_s and deliver it to n_t under the temporal constraints introduced by the time intervals, that minimizes the operational cost, i.e., the changing c and moving cost m, without adding a new route, but exploiting the existing ones.

In this thesis, we study the problem of evaluating path queries on route collections that do not fit in main memory and are frequently updated adding new routes. For instance, a collection of touristic routes is updated as users continuously share new interesting routes. The vehicle route collection of a supplier/delivery service is frequently updated as some of the new customer requests may not be served using current routes and therefore, new routes are included.

A route collection can be transformed to a graph and thus, PATH and FLSP queries can be evaluated using graph-based techniques following one of the two paradigms: (i) direct evaluation, e.g., graph traversal methods, and (ii) preprocessing the graph to compute and store reachability information. The latter techniques are the fastest, but they are mostly suitable for graphs that are not frequently updated, or when the updates are localized. In contrast, the former techniques are easily maintainable, but are slow as they visit a large part of the graph. In Section II, we discuss the existing work that follow the two paradigms above.

Based on these observations, Section III presents our methods from [16] and [17] for answering PATH queries directly on route collections and our work for FLSP. Our framework

TABLE I

SUMMARY OF RELATED WORK ON EVALUATING PATH AND FLSP QUERIES ON GRAPHS.

category	method	graph type	PATH	FLSP	maintenance
Direct evaluation	depth/breadth-first search	all types	yes	no	update adjacency lists
	uniform-cost search/A* [2]	all types	yes	yes	update adjacency lists
	Dijkstra	all types	yes	yes	update adjacency lists
Preprocessing the graph	2-hop [3]	all types,	yes	yes	not discussed
	HOPI [4], [5]	all types	no	no	method in [4]
	Geometry-based [6] and graph partitioning based 2-hop [7]	DAG	no	no	not discussed
	updatable 2-hop [8]	DAG	no	no	node-separation property
	3-hop [9]	DAG	no	no	not discussed
	Interval labelling [10]	DAG	computing ancestors	no	gaps in post order numbers
	Dual Labeling [11]	DAG	no	no	not discussed
	GRIPP [12]	all types	computing descendants	not discussed	not discussed
	Path-cover [13]	DAG	no	no	not discussed
	Graph embedding [14], [15]	all types	yes, using A*	yes, using A*	not considered, structure remains fixed

combines the positive aspects of the two aforementionted paradigms.

Finally, we conclude our discussion and propose future research directions in Section IV.

II. EXISTING WORK

A route collection $\mathbf{P}$ can be mapped to a directed graph. In case of PATH queries, we construct $G_{\texttt{PATH}}(N, E)$, such that N contains each node in $\mathbf{P}$, and E has an edge $n_1 \rightarrow n_2$ for every pair of consecutive nodes n_1, n_2 in a route r. For FLSP queries, we construct $G_{\texttt{FLSP}}(N, E)$, where N has a node of the form (n, r) for each route r containing node n of the collection, and E contains edges with cost given by pair (c, m), of the following two types: (i) an edge $(n_1, r) \rightarrow (n_2, r)$ for every pair of consecutive nodes n_1, n_2 in a route r with $c = 0$ and m equal to the moving cost from n_1 to n_2 in r, and (ii) an edge $(n, r_1) \rightarrow (n, r_2)$ for each link n between routes r_1 and r_2 with c equal to the changing cost from r_1 to r_2 and $m = 0$.

Then, it is straightforward to see that PATH and FLSP queries on $\mathbf{P}$ can be answered on $G_{\texttt{PATH}}$ and $G_{\texttt{FLSP}}$ respectively, following one of the two paradigms: (i) direct evaluation, and (ii) preprocessing the graph. Note that PATH query is closely related to REACH query, studied in the literature. However, $\texttt{REACH}(n_s, n_t)$ only determines if a path from n_s to n_t exists. Thus, an answer to $\texttt{PATH}(n_s, n_t)$ provides an answer also to $\texttt{REACH}(n_s, n_t)$, while the converse does not hold. In addition, a method for FLSP queries provides an answer also to PATH queries, but not in an efficient way. Table I summarizes the existing work in terms of the graph type supported, and the ability to answer PATH and FLSP queries and to handle updates.

Direct evaluation. The simplest way to evaluate PATH and FLSP queries is to traverse the graph at query time exploiting a search algorithm, e.g., depth-first or breadth-first search, and uniform-cost or A* search [2] respectively. Furthermore, especially for FLSP queries, we can exploit the Dijkstra algorithm. This approach has minimum space requirements, since it only stores the adjacency lists of the graph. In addition, the adjacency lists can be easily updated. On the other hand, it may need to visit a large part of the graph to answer a query.

Preprocessing the graph. As a different approach, we preprocess $G_{\texttt{PATH}}$ or $G_{\texttt{FLSP}}$ to compute and store reachability information for efficiently evaluating PATH or FLSP queries respectively. In case of PATH queries, we exploit the transitive closure (TC) of $G_{\texttt{PATH}}$, i.e., the graph $G^*_{\texttt{PATH}}$ that has an edge $n_1 \rightarrow n_2$ if a path from n_1 to n_2 exists in $G_{\texttt{PATH}}$. Using the TC a PATH query can be answered in constant time. However, although efficient algorithms for computing the TC have been proposed, the computation and storage cost are prohibitive for large disk-resident graphs. Thus, various methods that compress the reachability information have been proposed.

In [3], 2-hop identifies a set of nodes, called centers, that best capture the reachability information of a graph as intermediates. Then, each node n is assigned a list $L_{in}[n]$ with the centers that can reach n, and another $L_{out}[n]$ with the centers reachable from n. In addition, the first-edge information is included, i.e., the first node in the path from n to a center in $L_{in}[n]$ and from a center in $L_{out}[n]$ to n. 2-hop can also be exploited to identify the shortest path between two nodes, and thus, for answering FLSP queries. However, computing the optimal 2-hop scheme is NP-hard, and while an approximation algorithm is given in [3], it still requires the computation of the TC. Thus, this approach cannot be applied to large graphs. In addition, the work does not handle frequent updates.

HOPI [4], [5] exploits graph partitioning to reduce the building cost of 2-hop. To deal with updates, it exploits the method in [4]. However, HOPI cannot be used for PATH or FLSP queries, as it can only identify elements that match XPath expressions of the form: `//book//author` (where "`//`" is the ancestor-descendant operator) in an XML document collection, but not to detect explicitly the actual path with all nodes included.

Other efforts, e.g., [6], [7], [8], [9], [10], [11], [13], entirely focus on REACH queries. They first transform the input graph into a DAG by replacing each strongly connected component with a "super" node. For example, [10] proposes an interval labelling scheme based on the postorder traversing of DAG's spanning tree. Updates are handled by leaving gaps in postorder numbers. Although not discussed, PATH queries can be answered on DAG by computing the ancestors of a node. However, all the above works cannot be adopted for PATH queries on $G_{\texttt{PATH}}$ constructed from a route collection since, due to "super" nodes, it is not possible to construct full paths.

Unlike the previous works, the GRIPP scheme [12] assigns an interval label to each node of a graph (not necessarily a

DAG). Although, not discussed, PATH queries can answered by computing the descendants of a node. On the other hand, [12] does not deal with frequent updates.

Finally, [14], [15] use the graph embedding technique to derive lower and/or upper bounds of the distance between two nodes. These bounds are then exploited as heuristics by an A* search for answering shortest path queries. Thus, this approach can be used for FLSP queries on $G_{\texttt{FLSP}}$ graph. However, a basic assumption of these methods is that the graph structure remains fixed, whereas in our setting route collections and $G_{\texttt{FLSP}}$ graphs constructed from them are frequently updated.

III. Our Framework

In this section, we present our methods for evaluating path queries on route collections. First, we introduce our index schemes on route collections, termed: $\mathcal{P}$-$Index$, $\mathcal{H}$-$Index$ and $\mathcal{L}$-$Index$. Then, we present algorithms that exploit our indices for efficiently evaluating PATH and FLSP queries. Finally, we discuss maintenance issues of the indices under frequent updates of the route collections.

A. Indexing Route Collections

The $\mathcal{P}$-$Index$ [16] of a route collection $\mathbf{P}$, denoted by $\mathcal{P}$-$Index(\mathbf{P})$, is an inverted index on $\mathbf{P}$ that associates each node with the routes that contain it. Specifically, for each node n in $\mathbf{P}$, $\mathcal{P}$-$Index(\mathbf{P})$ retains list $routes[n]$ of $\langle r : o\rangle$ entries for all routes r that include n, where o indicates the position of node n in r. Note that the $routes[n]$ list is sorted by the route identifier r.

The $\mathcal{H}$-$Index$ [16] of a route collection $\mathbf{P}$, denoted by $\mathcal{H}$-$Index(\mathbf{P})$, captures all possible transitions among the routes in $\mathbf{P}$ via their links (shared nodes). In particular, for each route r in $\mathbf{P}$, $\mathcal{H}$-$Index(\mathbf{P})$ retains list $edges[r]$ of $\langle r_1, n_l : o : o_1\rangle$ entries for every route r_1 that has a shared node, link n_l, with r. Element o (resp. o_1) denotes the position of link n_l in route r (resp. r_1). Note that the $edges[r]$ list is sorted primarily by route r_1 identifier, and secondarily by o.

Storing all possible transitions among the routes of a collection in $\mathcal{H}$-$Index$ requires a lot of space and has a large maintenance cost. To deal with this problem, we introduce the $\mathcal{L}$-$Index$ [17] of a route collection $\mathbf{P}$, denoted by $\mathcal{L}$-$Index(\mathbf{P})$, taking into account only the links contained in each route. Specifically, for each route r in $\mathbf{P}$, $\mathcal{L}$-$Index(\mathbf{P})$ retains list $links[r]$ of $\langle n_l : o\rangle$ entries for every link n_l contained in route r at position o. Note that the $links[r]$ list is sorted by the link node identifier.

Figure 1 illustrates a route collection and its indices.

B. Evaluating PATH Queries

In [16], we introduce the path-first search (pfs) paradigm. In particular, pfs traverses the nodes in a collection similar to depth-first search. For each node n, it visits part of every route r that contains it at once, i.e., pfs visits all nodes that follow n in r. Then, building upon the pfs paradigm, we propose algorithms pfsP, pfsH and pfsL that exploit the reachability information within the routes to efficiently evaluate PATH queries. The algorithms mainly differ in the indexing scheme from Section III-A they exploit to terminate the search.

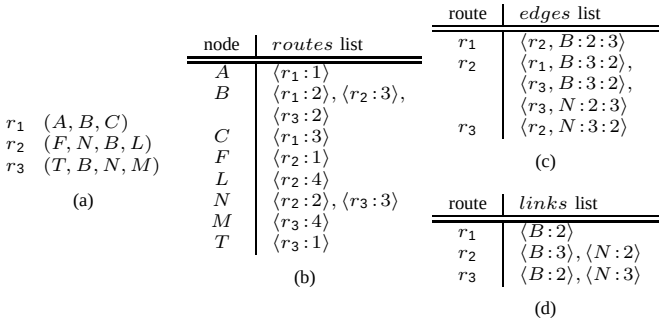

Fig. 1. (a) An example of a route collection $\mathbf{P}$, (b) $\mathcal{P}$-$Index(\mathbf{P})$, (c) $\mathcal{H}$-$Index(\mathbf{P})$, and (d) $\mathcal{L}$-$Index(\mathbf{P})$

The pfsP algorithm exploits the $\mathcal{P}$-$Index$ in two ways. First, it accesses the routes that contain a node n to visit the nodes after it, by performing a linear scan on $routes[n]$. Second, it terminates the search when a route that contains the target n_t of PATH(n_s, n_t) query after a node n is found. Given node n, to perform this check, we join $routes[n]$ and $routes[n_t]$ lists of $\mathcal{P}$-$Index$. The procedure is similar to a merge-join, as both $routes$ are sorted by the route identifier, that finishes as soon as a common route r_c with n before target n_t is found, or one of the lists is traversed to the end. The answer to PATH(n_s, n_t) query is given by the path from source n_s to node n and the part of route r_c from n to target n_t.

The pfsH algorithm exploits the $\mathcal{H}$-$Index$ of a route collection as follows. Intuitively, an entry $\langle r_1, n_l : o : o_1\rangle$ in $edges[r]$ list of $\mathcal{H}$-$Index$ denotes that all nodes in r before link n_l can reach the nodes after n_l in r_1. Based on this, the key idea of the pfsH termination condition is to check for every route r that contains a node n, whether there is a route r_1 in the collection such that: (i) r and r_1 have a common link n_c, (ii) r contains n before link n_c, and (iii) r_1 contains target n_t after link n_c. Specifically, given a node n and a route r that contains it, we join list $edges[r]$ of $\mathcal{H}$-$Index$ with $routes[n_t]$ of $\mathcal{P}$-$Index$ for target n_t. The procedure is similar to a merge-join as both lists are sorted by the route identifier, that finishes as soon as a common route r_c is found and conditions (i), (ii) and (iii) are satisfied, or one of the lists is traversed to the end. The answer to PATH(n_s, n_t) query is given by the path from source n_s to node n, the part of route r from n to link n_c, and the part of r_c from n_c to target n_t.

The basic idea of the pfsL algorithm is to traverse the collection considering only the links of each route while ignoring all other nodes. Given a node n and a route r that contains it, we access the links after n in r by sorting $links[r]$ according to the position of the links in r. Furthermore, pfsL terminates the search after visiting a link that lies before target n_t in a route of the collection. To this end, we first construct a list $\mathcal{T}$ that contains every link n_l before target n_t in a route r of the collection, and the part of route r from n_l to n_t. Note that $\mathcal{T}$ list is sorted by the link node identifier. Then, given a node n and a route r that contains it, we join $links[r]$ list with $\mathcal{T}$ looking for a common link n_c such that n_c is before n in route r. The procedure is similar to a merge-join as both lists

are sorted by the link identifier, that finishes as soon as such a common link n_c is found, or one of the lists is traversed to the end. The answer to PATH(n_s, n_t) query is given by the path from source n_s to node n, the part of route r from n to link n_c, and the path of n_c to target n_t, stored in $\mathcal{T}$.

C. Evaluating FLSP Queries

We adopt the uniform-cost search [2] to give a first-cut solution for FLSP queries. Similarly to the pfs paradigm, we traverse the route collection visiting for each node n all nodes that follow n in every route r of the collection. Building upon this setting, we devise two methods, termed spP and spL, to reduce the iterations performed by the search algorithm.

The key point of both methods is to compute a lower bound of the answer to FLSP query, called *candidate answer*, exploiting the indices of a collection. The candidate answer is continuously being improved until it becomes the correct answer. The role of a candidate answer is twofold: triggering an early termination condition and pruning the search space. Intuitively, we exclude a node n from the search when we determine that expanding n cannot result in an answer p better than current candidate answer. To perform this check, we calculate lower bounds for changing c and moving m cost of p and we compare them against the costs of current candidate answer. We also use this check to early terminate the search.

The spP and spL methods mainly differ in the index they exploit to compute a candidate answer. spP uses $\mathcal{P}$-$Index$ whereas spL exploits $\mathcal{L}$-$Index$. The procedure for computing a candidate answer is similar to the merge-join discussed for pfsP and pfsL with the exception that we identify all common entries (routes for spP or links for spL). In addition, similarly to pfsL, spL considers only the links of each route.

D. Updating Route Collections

As discussed in Section I, we consider route collections that do not fit in main memory and thus, all indices presented in this thesis are stored as inverted files on secondary storage and maintained by batch updates. Inverted files are more efficient when their lists are stored in a contiguous way. Therefore, dealing with each new route separately is not an efficient method for updating the collection. A common solution is to build inverted indices in memory considering all the new routes and to exploit them for evaluating the queries in parallel with the disk-based indices. Each time a set of new routes arrives, only the memory-based indices are updated with minimum cost. Then, to reflect the changes in the disk-based indices, there are three possible strategies: (a) rebuilding them from scratch using both the old and the new routes, (b) merging them with the memory resident ones and (c) lazily updating index lists when they are retrieved from disk during query evaluation. In our work, we adopt the second strategy.

IV. Future Work Directions

We plan to extend our work in three directions: (i) address other kinds of updates on route collections, (ii) evaluate other types of queries mostly considering constraints, and (iii) combine query evaluation with keyword search.

First, we plan to study the maintenance issues in cases apart from adding new routes. For example, in the dynamic pickup and delivery problem, new customer requests can be served by inserting nodes in existing vehicle routes. This update method will also change the time intervals of the nodes in the routes.

Second, we plan to evaluate queries similar to trip planning [18] and optimal sequenced route [19] queries. Specifically, consider a set of classes C such that each node n in a route collection is an instance of a class in C, e.g., the nodes in a touristic route are instances of classes $C = \{Museum, Stadium, Restaurant\}$. An interesting query is to find a path from a node n_s to n_t that passes first through a $Museum$, then a $Stadium$ and finally a $Restaurant$.

Finally, we could also combine query evaluation with keyword search. For example, instead of specific nodes, 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 want the path to pass through a $Restaurant$ with a description relevant to "sea food, lobster" keywords.

References

[1] G. Berbeglia, J.-F. Cordeau, and G. Laporte, "Dynamic pickup and delivery problems," *European Journal of Operational Research*, vol. 202, no. 1, 2010.

[2] S. J. Russell and P. Norvig, *Artificial Interlligence: A Modern Approach, Second Edition*. Prentice Hall, 2003.

[3] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *SODA*, 2002.

[4] R. Schenkel, A. Theobald, and G. Weikum, "Hopi: An efficient connection index for complex xml document collections," in *EDBT*, 2004.

[5] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the hopi index for complex xml document collections," in *ICDE*, 2005.

[6] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computation of reachability labeling for large graphs," in *EDBT*, 2006.

[7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *EDBT*, 2008.

[8] R. Bramandia, B. Choi, and W. K. Ng, "On incremental maintenance of 2-hop labeling of graphs," in *WWW*, 2008.

[9] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *SIGMOD Conference*, 2009.

[10] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD Conference*, 1989.

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

[12] S. Trißl and U. Leser, "Fast and practical indexing and querying of very large graphs," in *SIGMOD Conference*, 2007.

[13] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *SIGMOD Conference*, 2008.

[14] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for a: Efficient point-to-point shortest path algorithms," in *Proc. of the 8th WS on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia*, 2006.

[15] M. Schubert, M. Renz, and H.-P. Kriegel, "Route skyline queries: A multi-preference path planning approach," in *ICDE*, 2010.

[16] P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. K. Sellis, "Evaluating reachability queries over path collections," in *SSDBM*, 2009.

[17] P. Bouros, T. Dalamagas, S. Skiadopoulos, D. Sacharidis, and T. K. Sellis, "Evaluating path queries over frequently updated route collections," KDBS Lab, NTUA, Tech. Rep., 2009.

[18] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *SSTD*, 2005.

[19] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi, "The optimal sequenced route query," *VLDB J.*, vol. 17, no. 4, 2008.