Optimal Route Queries with Arbitrary Order Constraints

Jing Li, Yin Yang, Nikos Mamoulis

Abstract—Given a set of spatial points DS, each of which is associated with categorical information, e.g., restaurant, pub, etc., the optimal route query finds the shortest path that starts from the query point (e.g., a home or hotel), and covers a user-specified set of categories (e.g., {pub, restaurant, museum}). The user may also specify partial order constraints between different categories, e.g., a restaurant must be visited before a pub. Previous work has focused on a special case where the query contains the total order of all categories to be visited (e.g., museum $\rightarrow$ restaurant $\rightarrow$ pub). For the general scenario without such a total order, the only known solution reduces the problem to multiple, total-order optimal route queries. As we show in this paper, this naïve approach incurs a significant amount of repeated computations, and, thus, is not scalable to large datasets. Motivated by this, we propose novel solutions to the general optimal route query, based on two different methodologies, namely backward search and forward search. In addition, we discuss how the proposed methods can be adapted to answer a variant of the optimal route queries, in which the route only needs to cover a subset of the given categories. Extensive experiments, using both real and synthetic datasets, confirm that the proposed solutions are efficient and practical, and outperform existing methods by large margins.

Index Terms—H.2.4.h Query processing, H.2.4.k Spatial databases

1 Introduction

Consider a tourist who will have a free day to travel around Hong Kong. Without much knowledge about the city, s/he searches online maps to plan for a trip. Usually, s/he has a fixed starting point, e.g., her/his hotel, and certain objectives in mind, such as visiting a museum, dining at a fine restaurant, and enjoying a few drinks at a local pub. Meanwhile, some destinations may need to be visited in a certain order. For instance, the trip should have a pub after a restaurant. The ideal route should cover all the destinations, satisfy all order constraints, and minimize the total travel length. Searching for such a route is captured by the optimal route query [4], [10], [13], which usually has a vast search space, and, consequently, is too tedious to be done manually. Currently, major online map providers have already shown interest in tools that assist such trip planning tasks. For example, Google City Tours (*citytours.googlelabs.com*) provides suggested tours for a given starting address. However, these tours are pre-defined, and cannot be customized according to the user's plans. Yahoo Travel (*travel.yahoo.com*) has a similar service that allows users to search and share trips, which, unfortunately, cannot answer optimal route queries either.

Figure 1 illustrates an example optimal route query on a dataset DS with 6 locations p_1-p_6. Each location is associated with one category C_p, e.g., p_1, p_2 are museums; p_3, p_4 are pubs; and p_5, p_6 are restaurants. (If a location belongs to multiple categories, e.g., a restaurant and pub, we conceptually split it into multiple points with identical coordinates, each associated to a single category.) The query contains two parameters: a starting point q, and a directed acyclic graph G_Q called the *visit order graph*. Each vertex in G_Q corresponds to a category and each edge $\langle C, C' \rangle$ indicates that a point of category C must be visited before another of category C'. In our example, G_Q signifies that a restaurant must be visited before a pub. We follow a common assumption that each category appears at most once in G_Q [4], [10], [13]. In addition, to represent the fact that q must be the first point in the route, we create an artificial category C_q containing a single point q, and add an edge connecting C_q and every other vertex in G_Q without an in-edge. The result of the query is the shortest route that visits all categories in G_Q, while satisfying the visit order constraints. In our example, such a route is $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$. In practice, the user may not have sufficient time to visit all the categories. In this situation, a reasonable compromise is to find a route that covers a subset of l categories from G_Q, where l is a user-specified parameter. We call this variant the *size-l optimal route query*.

A *Greedy* algorithm[13] to answer the optimal route query first finds the nearest neighbor of q that is allowed to be visited right after q according to G_Q. In the running example, Greedy chooses point p_2 (note that p_4 cannot be selected, since G_Q requires that a pub is visited after a restaurant). Then, Greedy adds p_2 to the current route, and continues to compute the nearest allowable point according to G_Q to be added to the route, which is p_5.

- *J. Li is with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: jli@cs.hku.hk*
- *Y. Yang is with Advanced Digital Sciences Center, Singapore. E-mail: yin.yang@adsc.com.sg*
- *N. Mamoulis is with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: nikos@cs.hku.hk*

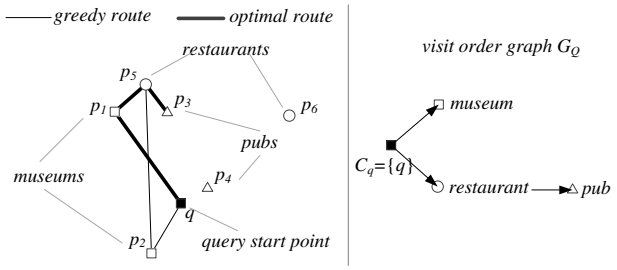


Fig. 1. Example of optimal route query

After that, Greedy finds the nearest allowable point after p_5, i.e., p_3. Since all categories in G_Q are visited, Greedy returns the route $q \rightarrow p_2 \rightarrow p_5 \rightarrow p_3$. Observe that this is longer than the optimal route $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$. The reason is that although p_2 is closer to q than p_1, the latter leads to a shorter sub-route that covers the remaining categories. In fact, the optimal route query is proven to be NP-hard [13], and heuristics-based algorithms such as Greedy cannot guarantee optimality of the result.

Previous work on the optimal route query, e.g., [13], has mainly focused on a special case where G_Q defines a *total order* of categories to be visited. A naïve approach for the general case, where G_Q is a partial order, is to enumerate all total orders in G_Q and process each of them individually. As we explain in Section 2.1, this method is inefficient as it incurs considerable repeated work. Motivated by this, we propose several efficient solutions to the general-case optimal route query. Specifically, we investigate two methodologies: backward search and forward search. The former computes the optimal route from the last point to the first, while the latter follows the first-to-last order of points. Furthermore, all proposed solutions extend naturally to size-l optimal route processing. Extensive experiments, using large-scale real and synthetic datasets, confirm that the proposed methods are efficient and practical.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 and 4 present solutions for the optimal route query, following the backward search and forward search frameworks, respectively. Section 5 extends the proposed solutions to the size-l optimal query. Section 6 contains an extensive set of experiments. Finally, Section 7 concludes the paper.

2 Related Work

Section 2.1 reviews existing solutions to the optimal route query. Section 2.2 surveys other related queries that operate on spatial data with categorical information.

2.1 Optimal Route Query Processing

Early work on optimal route computation focuses on greedy solutions. Chen et al. [4] use the same query definition as this paper, and propose two heuristics. The first, namely NNPSR, resembles the greedy approach described in Section 1; the second retrieves the nearest point of the query start position q in every category, and then connects them to form a route. In addition, [4] also describe a simple combination of NNPSR and R-LORD [13], which answers a special case of the optimal route query with a total order of the categories to be visited. The hybrid solution first runs NNPSR to find a greedy route; then, it extract the category of each point on the greedy route, and runs R-LORD with this category sequence as input. None of the solutions in [13] guarantees the quality of the results; these methods usually return sub-optimal routes according to the experiments in [4]. Li et al. [10] study a variant of the optimal route query that specifies both a start point q_{start} and an end position q_{end}, but no order constraint between the data categories. This is equivalent to a visit order graph G_Q that contains two artificial categories $C_{start} = \{q_{start}\}$ and $C_{end} = \{q_{end}\}$, and two edges $\langle C_{start}, C\rangle$ and $\langle C, C_{end}\rangle$ for each category C in the dataset. The solutions of [10] report approximate query results; on the other hand, this paper focuses on efficient, exact methods for the general optimal route problem.

Sharifzadeh et al. [13] propose R-LORD, the first exact solution for optimal route queries with a total order. In the example of Figure 1, suppose that G_Q specifies total order $q \rightarrow$ museum $\rightarrow$ restaurant $\rightarrow$ pub; then, R-LORD is directly applicable. Specifically, let r^* be the optimal route; an important observation made in [13] is that any suffix r of r^* is also the shortest among all routes that (i) start at the first point of r, and (ii) visit the same categories as r, in the same order. In our example, the best answer to the query is $r^* = q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$. Its length-2 suffix $p_5 \rightarrow p_3$ is the shortest route that starts at p_5 and visits a restaurant followed by a pub. Similarly, its length-3 suffix $p_1 \rightarrow p_5 \rightarrow p_3$ is the shortest path that originates at p_1 and follows the category sequence museum $\rightarrow$ restaurant $\rightarrow$ pub. This fact enables *dynamic programming*, which gradually fills an *optimal suffix table*.

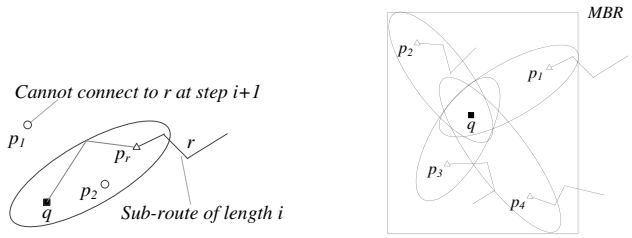
In particular, R-LORD first uses a greedy algorithm to compute a route that satisfies the query, as well as its length θ. Then, the method computes length-1 optimal suffixes, which are points from the last category in the visit order that are within θ-distance to the query start position q. In our example, R-LORD obtains pubs p_3 and p_4, and stores them in the optimal suffix table shown in Table 1. Next, R-LORD retrieves points from the second-to-last category that are no farther than θ from q, i.e., restaurants p_5 and p_6, and prepends them to the optimal length-1 suffixes to form optimal length-2 suffixes $p_5 \rightarrow p_3$ and $p_6 \rightarrow p_4$. Note that $p_5 \rightarrow p_4$ and $p_6 \rightarrow p_3$ are discarded, as they have the same starting points and category sequences as their shorter counterparts $p_5 \rightarrow p_3$ and $p_6 \rightarrow p_4$, respectively. In the third step, R-LORD retrieves museums p_1, p_2, combines them with the optimal length-2 suffixes, and obtains optimal length-3 suffixes $p_1 \rightarrow p_5 \rightarrow p_3$ and $p_2 \rightarrow p_5 \rightarrow p_3$. Finally, R-LORD connects them with q, and selects the shortest one $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$ as the answer to the query.

During the computation of the optimal suffix table, R-LORD uses a pruning technique to eliminate sub-routes

TABLE 1
Optimal suffix table used in R-LORD

Suffix Length	Start Point	Optimal Suffix
1	p_3	p_3
	p_4	p_4
2	p_5	$p_5 \rightarrow p_3$
	p_6	$p_6 \rightarrow p_4$
3	p_1	$p_1 \rightarrow p_5 \rightarrow p_3$
	p_2	$p_2 \rightarrow p_5 \rightarrow p_3$

that cannot participate in the optimal solution. Figure 2 illustrates this technique, which we call *elliptic pruning*. Suppose that at step i, R-LORD has computed an optimal sub-route r of length i. Let p_r be the first point of r, $length(r)$ be the total length of r, and θ be the length of the greedy route. Then, at step $i+1$, R-LORD connects r only to points whose total distance to q and p_r is no larger than $\theta - length(r)$. Thus, the range for points allowed to connect to r is an ellipse with foci q and p_r and major diameter $length(r)$. For example, in Figure 2(a), point p_1 is not connected to sub-route r, as the former falls outside the latter's ellipse. This is true even when the combination of p_1 and r leads to an optimal sub-route of length $i+1$. Thus, elliptic pruning reduces the number of stored optimal sub-routes and, thus, improves both memory consumption and CPU time. Furthermore, to minimize I/O costs, R-LORD computes the minimum bounding rectangle (MBR) of all ellipses generated from length-i optimal sub-routes, as shown in Figure 2(b), and uses this MBR as a range query to retrieve points from the R-tree [9] that indexes the category to be examined during the $(i+1)^{th}$ step.



(a) with one ellipse (b) with multiple ellipses

Fig. 2. Elliptic pruning in R-LORD

PLUB [11] decomposes a general optimal route query to multiple total-order queries and processes them individually, e.g., using R-LORD. For instance, the query in Figure 1 is decomposed into three total-order queries: museum $\rightarrow$ restaurant $\rightarrow$ pub, restaurant $\rightarrow$ museum $\rightarrow$ pub, and restaurant $\rightarrow$ pub $\rightarrow$ museum. This incurs significant amounts of repeated computations for longer sequences. For example, assume that in the query of Figure 1 there is an additional category (e.g., mall) that does not have any order constraints with other categories. The decomposition of this new query involves multiple total orders that share a common suffix, such as mall $\rightarrow$ museum $\rightarrow$ restaurant $\rightarrow$ pub and museum $\rightarrow$ mall $\rightarrow$ restaurant $\rightarrow$ pub. Consequently, the processing of both orders involves the computation of optimal sub-routes that start at a restaurant and are followed by a pub. This problem is amplified, as the number of categories in G_Q increases, since the number of total orders that share a common suffix increases exponentially.

Finally, Chen et al. [6] study the k Best Connected Trajectories (k-BCT) query, which resembles the optimal route query in that a k-BCT query consists of a set of (ordered or un-ordered) spatial locations, and each of it results should cover all locations in the query set. However, unlike the optimal route query which constructs routes on the fly, k-BCT retrieves k existing trajectories from a database with the lowest aggregate distance to the query points. The focus of [6] is clearly different from our work, and its methods do not apply to the optimal route query.

2.2 Spatial Search with Categorical Information

Besides the optimal route query, categorical information has been used to identify locations with good surrounding facilities. Yiu et al. [15] study the spatial preference query, which contains a list of desired categories. Data points are then ranked by their total distances to nearest points of these categories and those with top-k best scores are returned to the user. Martinenghi and Tagliasacchi [12] introduce the proximity rank join operator, which searches for clusters of points that cover all categories specified by the user and are close to a given point and to each other.

Another class of related work concerns spatial keyword search in collections of documents, which are associated to spatial locations (e.g., derived from the content of the document [1]). The query contains both a spatial component (e.g., nearest neighbor search) and a set of keywords. A keyword set is similar to a category in that they are both non-spatial properties that can be used to select a set of points (i.e., document locations). However, the number of different keyword sets is significantly larger than the number of categories and, thus, the former require specialized data structures (e.g., inverted lists) and search techniques (e.g., inverted list intersection) to select relevant points. To accelerate spatial keyword search, a common approach is to combine a spatial index, e.g., R-tree with inverted lists or signature techniques, to form a composite index [8], [16], [7], [5]. The relevance of a document to a query is calculated by combining textual relevance with spatial distance; the top-k objects with the highest overall scores are returned to the user [7]. Besides simple similarity retrieval, the mCK query [17] identifies clusters of points with minimum diameters that match all query keywords. The top-k prestige query [3] retrieves points based on prestige scores, which originate from matching keywords and flows to nearby points. Finally, the continuous top-k spatial keyword query [14] returns a validity region to the user; as long as the query point stays in the validity region, the query results remain the same.

3 BACKWARD SEARCH SOLUTIONS

In this section, we present the first methodology for answering optimal route queries. Similar to R-LORD [13], the backward search methodology computes the optimal routes in reverse order of its points. Before explaining the methods that fit this framework in detail, we first present an important property of the general sub-route query, as follows.

Lemma 1 (Suffix Optimality)**.** *Given a query $\langle q, G_Q\rangle$ and its optimal solution r^*, let $r \subseteq r^*$ be any suffix of r^*, p be the start point of r, and V be the set of categories covered by r. Meanwhile, let $G \subseteq G_Q$ be the sub-graph of G_Q that contains the set of categories V and all edges between these categories in G_Q. Then, r is the optimal solution for query $\langle p, G\rangle$.*

Proof (By contradiction): Suppose that there is a better solution r' than r for the query $\langle p, G\rangle$, i.e., *length*(r')<*length*(r). Since r and r' have the same starting point p, we can replace the suffix r with r' in r^*, and obtain a new route r'^* such that *length*(r'^*)=*length*(r^*)−*length*(r)+*length*(r') < *length*(r^*). Meanwhile, since r' is a valid solution to the query $\langle p, G\rangle$, r' covers the same category set V as r, and satisfies the visit orders in $G \subset G_Q$. Because G contains all visit orders about V, replacing suffix r with r' in r^* does not violate G_Q. Hence, r'^* also satisfies G_Q. This means that r'^* is a better solution to query $\langle q, G_Q\rangle$ than r^*, which contradicts with the optimality of the r^*. □

Consider again the example in Figure 1, where the optimal solution for the query $\langle q, G_Q\rangle$ is $r^* = q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$. The length-2 suffix of the optimal route is $r_2 = p_5 \rightarrow p_3$, which starts at point p_5, and covers two categories $V_2 = \{$restaurant, pub$\}$. Clearly, r is the shortest route that starts at p_5 and covers V_2, since p_3 is the nearest pub with respect to p_5. Likewise, the length-3 suffix $r_3 = p_1 \rightarrow p_5 \rightarrow p_3$ of r^* is the optimal route that (i) starts at p_1, (ii) covers category set $V_3 = \{$museum, restaurant, pub$\}$, and (iii) satisfies the constraint that a restaurant must be visited before a pub. In general, all suffixes of the query result are also optimal routes for their respective starting point and categories visited and the idea of backward search is to enumerate all possible such suffixes. The suffix-optimality result in [13] is a special case of Lemma 1, with the limitation that a total order exists for all categories in G_Q.

Based on Lemma 1, we develop two algorithms SBS and BBS, presented in Sections 3.1 and 3.2 respectively. SBS directly extends R-LORD to the general optimal route problem, while BBS improves the performance of SBS through batch processing. Table 2 summarizes frequently used notations throughout the paper.

TABLE 2
List of common symbols

Symbol	Meaning
DS, N	Dataset and its cardinality
q, G_Q	Query start point and visit order graph
m	Total number of categories in G_Q
$dist(p_1, p_2)$	Euclidean distance between points p_1 and p_2
$mindist(M_1, M_2)$	Minimum distance between MBRs M_1, M_2
$length(r)$	Length of route r
$minlen(R)$	Minimum length among the set R of routes
$p \rightarrow r$ ($r \rightarrow p$)	A route that first visits point p (follows sub-route r) and then follows sub-route r (visits point p)
θ	Length of a known route that satisfies the query
CS	Set of points that may appear in the optimal route
$\Omega_{p,V}$	Shortest route that starts at point p and visits all categories in set V
$\mathbf{\Omega}_{P,V}$	Set of shortest routes that start at a point $p \in P$ and visits all categories in set V
C_p, C_P	Category of a point p and that of a set P of points having the same category, respectively

3.1 Simple Backward Search

Algorithm 1 illustrates the simple backward search (SBS) method. Initially, SBS computes an upper bound θ of the optimal route length, using a greedy algorithm (lines 1-2), e.g., the one described in Section 1. Then, SBS retrieves the set CS of candidate points that may be part of the optimal route (line 3), which are those that (i) belong to any category contained in the visit order graph G_Q, (ii) fall within distance θ to the query start point q. This can be performed efficiently, e.g., by executing a circular range query on each R-tree that indexes a category of points relevant to the query. In the example of Figure 1, SBS obtains all points p_1-p_6. Note that this is different from R-LORD [13], which only loads points belonging to the last category of the total-ordered query in the initial step, e.g., pubs p_3, p_4. In out setting, there is neither a total order or the concept of the last category.

Algorithm 1 Simple backward search algorithm

SBS(q, G_Q) // SBS stands for simple backward search
// Input: q, G_Q: query start point and visit order graph respectively
// Output: the optimal route that satisfies the query
1: Use a greedy algorithm to obtain a route r_g
2: Initialize threshold θ to *length*(r_g)
3: Retrieve the set CS of points within θ distance to q, whose categories appear in G_Q
4: Initialize route set R_1 to empty
5: **for** each point p in CS that can be the last point according to G_Q **do**
6: Add $\langle p\rangle$ to R_1
7: **for** $i = 1$ to $m - 1$ **do** call R_{i+1} =*BSJoin*(q, G_Q, θ, R_i, CS)
8: Select from R_m sub-route r^* that minimizes *length*($q \rightarrow r^*$)
9: Return $q \rightarrow r^*$ as the query result

After loading all candidate points, SBS continues to compute the optimal route sets R_1-R_m (lines 4-7). In particular, route set R_i ($1 \leq i \leq m$) contains all possible length-i suffixes of the query solution. According to Lemma 1, these suffixes must be the optimal routes for their respective start point and the set of categories covered. Table 3 lists all routes contained in R_1-R_m in our running example. Specifically, R_1 consists of 4 single-point routes: museums p_1, p_2, and pubs p_3, p_4. Restaurants are not included in R_1, since they must be visited before a pub and, thus, cannot be valid length-1 suffixes of the query solution. Route sets R_2-R_m are computed through *backward joins*, to be explained soon. Continuing the example, R_2 contains all optimal suffixes that cover two categories. Again, a route covering $\{$museum, restaurant$\}$ cannot be a suffix of the query result, since it would place a pub before a restaurant, violating G_Q.

Similarly, R_3 contains all suffixes containing all three query categories. After obtaining R_m (i.e., R_3 in the example), SBS connects the query point q with the start point of each route in R_m, and selects the route with shortest total length (lines 8-9) as the query answer. Here we use the notation $p \rightarrow r$ to denote a route that starts at p and follows the sub-route r, e.g., when $r = p_2 \rightarrow p_3$, $p_1 \rightarrow r = p_1 \rightarrow p_2 \rightarrow p_3$.

TABLE 3
Optimal suffix table used in SBS

R_i	Start point	Categories Covered	Optimal Suffix
1	p_1	{museum}	p_1
	p_2		p_2
	p_3	{pub}	p_3
	p_4		p_4
2	p_1	{museum, pub}	$p_1 \rightarrow p_3$
	p_2		$p_2 \rightarrow p_4$
	p_3		$p_3 \rightarrow p_1$
	p_4		$p_4 \rightarrow p_2$
	p_5	{restaurant, pub}	$p_5 \rightarrow p_3$
	p_6		$p_6 \rightarrow p_4$
3	p_1	{museum, restaurant, pub}	$p_1 \rightarrow p_5 \rightarrow p_3$
	p_2		$p_1 \rightarrow p_5 \rightarrow p_3$
	p_5		$p_5 \rightarrow p_3 \rightarrow p_1$
	p_6		$p_6 \rightarrow p_3 \rightarrow p_1$

It remains to clarify the backward join module BSJoin, shown in Algorithm 2 (also used in our other methods described later). Besides the query parameters, the main inputs are (i) a set of points P, which is the entire candidate set CS in SBS and (ii) a set of routes R (R_i in SBS), each of which is optimal for the combination of its start point and categories covered. The join results (R_{i+1} in SBS) consist of routes of the form $p \rightarrow r$, i.e., start point p followed by sub-route r, where $p \in P$ and $r \in R$. Note that in SBS, the computation of R_{i+1} ($1 \leq i < m$) only involves R_i and CS, meaning that after obtaining R_{i+1}, R_1-R_i can be safely discarded to conserve memory.

Algorithm 2 Algorithm for backward join

BSJoin(q, G_Q, θ, R, P)
// Input: q, G_Q: query start point and visit order graph respectively
// θ: length of a known route that satisfies the query
// R, P: a set of routes and points respectively
// Output: backward join results of P and R
1: Initialize route set R' to empty
2: Partition R based on the set of categories they cover
3: **for** each point $p \in P$ **do**
4: **for** each partition R_V of R covering the same category set V **do**
5: **if** connecting p with a route in R_V satisfies G_Q **then**
6: Find the route $r \in R_V$ that minimizes the length of $p \rightarrow r$ among all routes in R_V
7: **if** $dist(q,p)$+$length(p \rightarrow r) < \theta$ **then** add $p \rightarrow r$ to R'
8: Return R'

BSJoin selects join results based on three criteria. The first concerns the visit order constraints G_Q (line 5). Specifically, the route $p \rightarrow r$ itself must satisfy G_Q and not contain any duplicate categories. Meanwhile, since $p \rightarrow r$ is expected to be the suffix of a solution to the query, G_Q must allow all categories not covered by $p \rightarrow r$ be visited before $p \rightarrow r$. In the example of Figure 1, BSJoin eliminates all join outputs that either has a pub before a restaurant (which directly violate G_Q), and those that contains a restaurant, but not a pub (which cannot be suffixes of legal routes). Second, according to Lemma 2, $p \rightarrow r$ should be the optimal among all routes that start at p and cover the same categories as $p \rightarrow r$ (line 6). Finally, $p \rightarrow r$ must survive elliptic pruning [13] (line 7), described in Section 2.1. Unlike R-LORD [13] which uses the MBR of the ellipses to prune, BSJoin directly applies elliptic pruning, which is more efficient according to our experiments. The reason is that the MBR usually covers a significantly larger area than the ellipses (e.g., in Figure 2(b)), leading to poor pruning effectiveness; moreover, computing the MBR itself consumes considerable CPU time, sometimes defeating the purpose of pruning. The complexity of SBS is given by the following lemma.

Lemma 2. *The SBS algorithm finds the optimal solution of the query using $O(N \cdot 2^m \cdot m)$ memory, and $O(N^2 \cdot 2^m)$ time.*

Proof: During the i^{th} backward join (line 7 of Algorithm 2), SBS maintains in memory the set R_i of optimal sub-routes of length i. Each sub-route has length i, and there are at most $N \cdot \binom{m}{i}$ routes in R_i, where N is total number of points in the dataset and $\binom{m}{i}$ is the number of category combinations of length i, from a total of m categories. Because SBS also needs to compute R_{i+1}, the total memory consumption at this step is $O(N \cdot \binom{m}{i} \cdot i + N \cdot \binom{m}{i+1} \cdot (i+1))$. At the $(i+1)^{th}$ backward join, SBS releases the memory occupied by R_i, since it no longer affects subsequent optimal sub-route computations. Therefore, the peak memory usage of SBS is $O\left(\max_{i=1}^{m-1}(N \cdot \binom{m}{i} \cdot i + N \cdot \binom{m}{i+1} \cdot (i+1))\right) = O(2^m \cdot N \cdot m)$. Backward joins dominate the runtime cost. In particular, at step i, $1 \leq i \leq m$, SBS joins $O(N \cdot \binom{m}{i})$ sub-routes in R_i with $O(N)$ points in the candidate set CS. The time taken at this step is $O(N^2 \cdot \binom{m}{i})$. Summing up all m steps, we obtain the time complexity of SBS: $O\left(\sum_{i=1}^{m} N^2 \cdot \binom{m}{i}\right) = O(N^2 \cdot 2^m)$ □

SBS is easy to implement and it achieves the same worst-case time complexity as more complex algorithms described later. The main drawback of SBS is that its effectiveness relies heavily on the bound θ provided by the greedy algorithm. When θ is loose (i.e., it is much longer than the optimal length), SBS retrieves a large number of candidate points, and joins them all with the current sub-route set at every step. Moreover, the backward join in SBS is performed in a nested-loop fashion, which applies elliptic pruning on individual results. Consequently, SBS can be rather inefficient for large datasets with a skewed distribution.

3.2 Batch Backward Search

The batch backward search (BBS) method, shown in Algorithm 3, improves SBS by employing batch processing in the backward join operations. Specifically, both the candidate set CS and the route set R_i ($1 \leq i \leq m$) are partitioned into clusters before participating in a backward join (lines 2 and 4). The partitioning of CS first groups points by their category, and then for each group, the points are further partitioned into clusters based on their spatial proximity. The partitioning of route set R_i

follows a similar strategy, by first grouping routes based on the categories they cover, and then clustering each group according to the locations of their start points. The clustering module in BBS must be highly efficient, since it is called during query time. In our implementations we tested two clustering algorithms, both of which employ the Hilbert curve [2]. The first method, which we call *average gap* (*AG*), sorts all points by their Hilbert values; then, for each pair of adjacent points, *AG* computes their gap (i.e., difference) in Hilbert values, and obtains the average value g of such gaps. After that, *AG* scans the sorted points again. When two adjacent points p and p' have a gap smaller than g, *AG* places them in the same cluster; otherwise, *AG* creates a new cluster for p', and continues scanning. The second algorithm, called *maximum area* (*MA*), also sorts points by Hilbert values. Then, starting from the first point, *MA* keeps adding points to the current cluster c_l in increasing order of their Hilbert values, until the *MBR* of c_l has an area exceeding a pre-defined threshold, at which point *MA* starts a new cluster. In our implementation of *MA*, the above threshold is set to a percentage $ma\%$ of the circular area centered at the query start point q, with radius θ, which is the upper length bound of the optimal route.

Algorithm 3 Batch backward search algorithm

BBS(q, G_Q) // BBS stands for batch backward search
//Input and Output: same as algorithm SBS
1: Same as lines 1-6 in algorithm SBS
2: Partition points in CS into clusters, such that points in the same cluster belong to the same category, and are close to each other
3: **for** $i = 1$ to m **do**
4: Partition the sub-routes in R_i into clusters, such that sub-routes in the same cluster cover the same set of categories, and that their start points are close to each other
5: Initialize set R_{i+1} to empty
6: **for** each cluster $R \subset R_i$ **do**
7: **for** each cluster $P \subset CS$ that satisfies G_Q **do**
8: **if** $mindist(q, MBR_P) + mindist(MBR_P, MBR_R) + minlen(R) < \theta$ **then**
9: Temporarily delete all routes $r \in R$ satisfying $mindist(q, MBR_P) + mindist(MBR_P, r) + length(r) \geq \theta$.
10: Temporarily delete all points $p \in P$ satisfying $dist(q, P) + mindist(P, MBR_R) + minlen(R) \geq \theta$
11: Call $R' = BSJoin(q, G_Q, \theta, R, P)$
12: Restore R and P by adding back all deleted elements
13: Merge R' into R_{i+1}
14: Same as lines 8-9 in algorithm SBS

After the partitioning of CS and R_i, BBS proceeds to join them in order to obtain R_{i+1}. This is performed by joining on two levels: the cluster level and the individual point/route level. In the cluster level join, BBS tries to eliminate two participating clusters using a block elliptic pruning test, which is stated in Lemma 3 below.

Lemma 3 (Block Elliptic Pruning)**.** *Given query start point q, a set of points P, and a set R of sub-routes, let MBR_P and MBR_R be the MBR of all points in P and all start points in R, respectively, and θ be a known upper bound for the length of the optimal solution to the query. Let $minlen(R)$ be the length of the shortest route in R, and $mindist()$ be the function that returns the minimum distance between objects and MBRs. If $mindist(q, MBR_P) + mindist(MBR_P, MBR_R) + minlen(R) \geq \theta$, then, connecting any point in P with any route in R cannot possibly lead to a sub-route of the optimal solution to the query.*

Proof: For all $p \in P$ and $r \in R$ with start point p_r, *dist*(q, p)+*dist*(p, p_r)+*length*(r) $\geq$ *mindist*(q, MBR_P) + *mindist*(MBR_P, MBR_R) + *minlen*(R) $\geq \theta$. Following Lemma 2, $\langle p, r \rangle$ cannot be a sub-route of the optimal solution. □

Figure 3 shows an example of block elliptic pruning, which involves a point cluster P and a route cluster R. Note that all points in P belong to the same category (i.e., Category 1), and all routes in R cover the same set of categories (Categories 2-4); meanwhile, the start points of R may belong to different categories. Clearly, the sum of (i) $mindist(q, MBR_P)$, i.e., the minimum distance between q to any point in P, (ii) $mindist(MBR_P, MBR_R)$, which is the minimum distance between any point in P and the start point of any route in R, and (iii) $minlen(R)$, the smallest route length in R, gives a lower bound of any path that starts at q, stops at a point in P, and finally reaches and subsequently follows a route in R. If this sum exceeds the upper bound θ obtained through a greedy algorithm, BBS prunes the combination of P and R, and saves the computations for joining them.

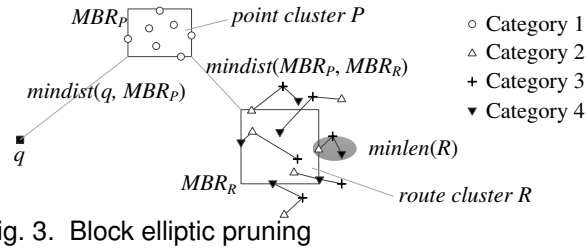


Fig. 3. Block elliptic pruning

After two clusters pass block elliptic pruning, BBS proceeds to join their content on the point/route level. In particular, given a point cluster $P \in CS$ and a route cluster $R \in R_i$, BBS first removes points and routes that cannot form valid join results, based on similar rules as block elliptic pruning (lines 9-10), and then joins the remaining objects using the same BSJoin algorithm as in SBS (line 11). For instance, a point p is removed, if the sum of the minimum route length in R, p's distance to the query point q, and its minimum distance to MBR_R reaches or exceeds the threshold θ. Such element eliminations further reduce the cost of backward joins. After obtaining the join results (let R') of R and P, BBS subsequently merges R' into the result set R_{i+1} as follows. For each route r in R', BBS checks whether there exists a shorter route r' in R_{i+1} that starts at the same point and covers the same categories. If so, r is discarded; otherwise, r is inserted to R_{i+1}, possibly replacing an existing route in the latter with a longer length. Finally, BBS restores clusters R and P, by adding back all deleted points and routes, respectively.

The space and time complexity of BBS is the same as that of SBS; the proof is also similar to that of SBS, and is omitted for brevity. As our experiments demonstrate, BBS significantly improves the efficiency

of SBS. In fact, when the greedy algorithm obtains a close approximation of the optimal route length, BBS can be highly efficient, sometimes even outperforming the more sophisticated techniques explained later. BBS, however, shares the same drawback of SBS: its efficiency depends largely upon the quality of the greedy bound θ. The forward search strategy, described next, removes this limitation, and achieves more stable performance.

4 Forward Search Solutions

The forward search approach traverses the search space in a depth-first manner, and incrementally improves the bound θ for optimal route length. As an additional benefit, forward search methods report results *progressively*, i.e., they first quickly produce one solution to the query, and then incrementally update it, until reaching the optimal one or being terminated by the user. Section 4.1 describes a simple solution SFS based on this idea. Section 4.2 presents an improved BFS algorithm that integrates elements from the backward search methods to achieve more effective pruning.

4.1 Simple Forward Search

Recall from Section 1 that algorithm Greedy computes a route by repeatedly connecting the current location (starting from the query point q) to the nearest point belonging to an *unvisited* category permitted by G_Q. The simple forward search method (SFS) resembles Greedy in that it also extends the current path by adding the nearest point from an unvisited category. A major difference between the two is that SFS *backtracks* after it obtains a complete route. In the running example of Figure 1, after SFS reaches the same route $q \rightarrow p_2 \rightarrow p_5 \rightarrow p_3$ as Greedy, the former backtracks to point p_5, connects it to its next nearest pub p_4, and checks whether the new route $q \rightarrow p_2 \rightarrow p_5 \rightarrow p_4$ is shorter than the current best one $q \rightarrow p_2 \rightarrow p_5 \rightarrow p_3$. After that, SFS backtracks to p_5 again. Since it has tested all pubs, and all other categories are already covered by the current path $q \rightarrow p_2 \rightarrow p_5$, SFS backtracks once more to p_2. It then connects p_2 to its next nearest permissible neighbor p_6 (note that pubs p_3 and p_4 cannot be connected due to order constraints), and continues with the prefix $q \rightarrow p_2 \rightarrow p_6$. The process terminates when all feasible routes are examined, and the shortest route is reported as the query result.

A naïve implementation of forward search clearly takes time exponential to the number of points in the dataset. SFS (shown in Algorithm 4) avoids this by utilizing the suffix optimality property stated in Lemma 1, and incrementally building the optimal suffix table (denoted by Ω), as in the backward search methods. SFS differs from the backward search solutions in that it fills Ω in a different order. Specifically, in SFS all cells of Ω are initialized to a special token *Unknown* (line 2 in function *InitSFS*), which indicates that the corresponding optimal suffix has not been computed yet. Then, whenever SFS backtracks from a point p, the algorithm guarantees that it has obtained the optimal suffix that starts at p, and covers the set of categories V that are not visited by the current path from the query point q to p [1]. Thus, SFS stores this optimal suffix in the corresponding cell in Ω (denoted by $\Omega_{p,V}$), replacing the *Unknown* token (line 9 in function *SFS*). If later SFS needs to compute the same suffix, i.e., when it traverses to p again with the same set of visited categories, it directly appends $\Omega_{p,V}$ to the current path and backtracks, eliminating repeated computations (line 3 in function *SFS*).

Algorithm 4 Simple forward search algorithm

InitSFS(q, G_Q)
// Input and Output: same as algorithm SBS
1: Same as lines 1-4 in algorithm SBS
2: Initialize all elements of Ω to *Unknown*
3: Call SFS(q, G_Q, q, 0, θ), and return its result as the query answer

SFS(q, G, p, l, θ) // SFS stands for simple forward search
// Input: q: query start point
// G: sub-graph of G_Q containing only unvisited categories
// p, l: end point and length of the current route, respectively
// θ: length of a known route that satisfies the uery
// Output: optimal route starting at p, and covering all categories in G
1: Let V be the set of categories in G, category C_p be the category of p, and category set $V' = V \setminus \{C_p\}$
2: Construct new sub-graph G' by removing C_p and all related edges from G
3: **if** $\Omega_{p,V} \neq$ *Unknown* **then** return $\Omega_{p,V}$
4: **if** V contains a single category C_p **then** return single-point route p
5: Compute the distance from p to its nearest neighbor in each category $C \in V$, and set d_{max} to the maximum of these distances
6: **if** *dist*(q, p)+$d_{max} \geq \theta$ **then** set $\Omega_{p,V}$ to *Invalid*, and return *Invalid*
7: **for** each point $p' \in CS$ in increasing order of *dist*(p, p') **do**
8: **if** adding p' after p satisfies G **then**
9: Recursively call $\Omega_{p',V'}$=*SFS*(q, G', p', l+*dist*(p, p'), θ)
10: **if** $length(p \rightarrow \Omega_{p',V'}) < length(\Omega_{p,V})$ **then**
11: Set $\Omega_{p,V}$ to $p \rightarrow \Omega_{p',V'}$
12: **if** *length*($\Omega_{p,V}$)+l<θ **then** update θ and CS

In our running example, after SFS searches all routes with the prefix $q \rightarrow p_2$, it backtracks to q, and connects to its next permissible neighbor p_1. Next, when SFS adds p_5 to the current path, it finds that the optimal suffix starting at p_5 and visiting categories restaurant, pub has already been computed (i.e., $p_5 \rightarrow p_3$), during the previous searches with prefix $q \rightarrow p_2 \rightarrow p_5$. Therefore, SFS directly appends p_3 to p_5, forming a new route $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$, and backtracks to p_1.

Note that SFS iteratively improves the upper bound θ for the length of the optimal query result, whenever it identifies a better solution to the query (line 12 in function *SFS*). Meanwhile, reducing θ also shrinks the candidate set CS, which are the points within θ distance to the query point q. Pruning with θ in SFS, however, is rather tricky. First, the elliptic pruning strategy is no longer as effective as it is in SBS and BBS, since it aims at eliminating the storage and extension of unqualified suffixes. In SFS, although elliptic pruning helps to reduce memory consumption, it has no effect on CPU time, since SFS does not extend suffixes. Second, straightforward pruning using θ and the length of the current prefix may lead to repeated visits to the same prefix paths. We explain this point using the SFS example shown in

1. In our presentation, the set V includes the category C_p of the last point p of the current prefix path.

Figure 4. Assume that the algorithm has reached a point p_1 through a prefix, and needs to visit the remaining categories (i.e., categories 2-4). Note that there is no guarantee on prefix optimality; hence, there can be a shorter path connecting q and p_1 that covers exactly the same categories.

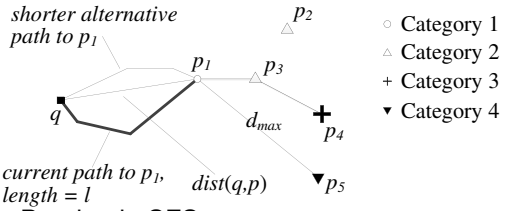


Fig. 4. Pruning in SFS

A simple idea for pruning is to backtrack whenever the length of the current prefix reaches or exceeds the upper bound θ, since subsequent searches based on this prefix cannot possibly lead to the optimal solution to the query. However, with such early backtracking, SFS no longer obtains the optimal suffixes, and, consequently, may perform repeated computations in later steps. In the example of Figure 4, suppose that SFS extends the prefix from p_1 to p_3, and then to p_4, and at this point, the length of the current path exceeds θ. If SFS backtracks, it cannot obtain the optimal suffix starting at p_4, and visiting categories 3-4. Furthermore, when the method later backtracks to p_3, it has not computed the optimal suffix starting at p_3 and covering categories 2-4 either. The problem propagates to p_1 as well, and to all points preceding p_1. Assume that SFS reaches p_1 with a shorter prefix covering the same categories, it cannot reuse the computations performed before (i.e., extending p_1 to p_3 and p_4), and instead must perform these computations from scratch. In general, all pruning strategies based on the prefix length suffer from the same problem. Instead, SFS uses a weaker pruning method, as follows.

Lemma 4 (SFS Pruning)**.** *Given a query $\langle q, G_Q \rangle$, a data point p, and a category $C \in G_Q$ such that $p \notin C$, let p_C the nearest point of p in C, and θ be an upper bound for the length of the optimal solution to the query. If $dist(q,p) + dist(p,p_C) \geq \theta$, then, any route that starts at p and covers C cannot possibly be a sub-route of the optimal solution.*

Proof: Let r be a route that starts at p and covers C. The length of r is at least *dist*(p, p_C), due to triangular inequality and the fact that p_C is the nearest neighbor of p in C. Similarly, the length of the route that connects q to p is at least *dist*(q, p). Therefore, any route that starts at q and has r as a sub-route must have length at least *dist*(q, p)+*dist*(p, p_C)$> \theta$, which exceeds the upper bound θ for the optimal solution. Hence, r cannot be a sub-route the optimal solution. □

In Figure 4, SFS compares θ with the sum of the distance between q and p_1 and that between p_1 and the farthest point p_5 from an unvisited category. If the latter reaches or exeeds θ, SFS safely backtracks, and marks the cell corresponding to the optimal suffix starting at p_1 with a special token *Invalid* (line 6 in function SFS). This pruning technique does not involve prefix length, so it avoids the repeated-work problem described above. SFS has the same worst-case space and time complexity as the backward search methods, as shown below.

Lemma 5. *SFS finds the optimal solution of the query using $O(N \cdot 2^m \cdot m)$ memory, and $O(N^2 \cdot 2^m \cdot m)$ time.*

Proof: In the worst case, SFS stores all possible optimal sub-routes in $\mathbf{\Omega}$. Similar to the proof of Lemma 2, the number of unique optimal length-i sub-routes is $N\binom{m}{i}$. In the worst case, SFS will occupy $O(N \cdot \binom{m}{i} \cdot i)$ memory. Summing up sub-routes of all different lengths, the peak memory usage of SFS is $O\left(\sum_{i=1}^{m} N \cdot \binom{m}{i} \cdot m\right) = O(2^m \cdot N \cdot m)$. The analysis for time complexity is more tricky, as SFS is a recursive procedure. To avoid double counting, for each invocation of function SFS (Algorithm 4), we count only the time for performing simple operations, and exclude the time spent for recursively calling itself. The dominant cost of each call to SFS is the loop that enumerates all possible points to be appended to the current route starting from q, bounded by $O(N)$. Next we derive the number of times such a loop is performed. For each combination of a start point p and category set V, SFS performs the expensive loop only once, because the next call with the same parameters simply returns the route stored in $\mathbf{\Omega}_{p,V}$. Since the number of start points is bounded by $O(N)$ and the number of unique category sets is $O(2^m)$, the total time spent by all invocations of SFS is $O(N \cdot N \cdot 2^m) = O(N^2 \cdot 2^m)$. □

Compared with backward search methods, SFS gradually tightens the upper length bound θ and shrinks the candidate set CS accordingly. Therefore, when the initial value of θ (as well as the initial size of CS) is large, SFS can be significantly more efficient than SBS and BBS, due to the elimination of a large portion of candidate points early. On the other hand, SFS is rather weak in partial route pruning and it has to examine each partial route individually. Batch forward search (BFS), presented next, addresses these issues by integrating elements of backward search.

4.2 Batch Forward Search

BFS follows the same depth-first search paradigm as SFS. However, instead of enumerating individual routes, BFS searches for sequences of clusters, which we call *cluster paths*. Specifically, in a pre-processing step, BFS partitions the candidate set into clusters as in BBS, i.e., the points in each cluster belong to the same category, and are close to each other in space. Figure 5 illustrates BFS on our running example, which involves three clusters P_1-P_3, for museums p_1-p_2, pubs p_3-p_4 and restaurants p_5-p_6 respectively. In general, a category may have multiple clusters. A *cluster path* is a sequence of clusters, e.g., $cp_1 = q \rightarrow P_1 \rightarrow P_3 \rightarrow P_2$. Since each cluster is associated with a single category, we can check whether a cluster path satisfies the visit order graph G_Q similarly to routes.

For instance, $q \rightarrow P_1 \rightarrow P_2 \rightarrow P_3$ violates G_Q, since it contains a pub cluster before a restaurant cluster.

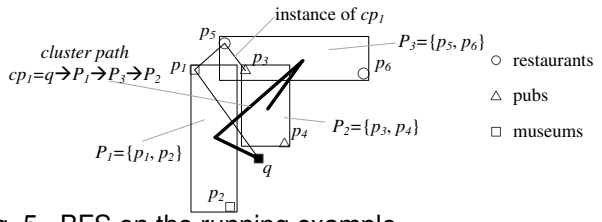


Fig. 5. BFS on the running example

Given a cluster path cp, we can form a route by taking a point from each cluster, and connect them according to the order of their corresponding clusters in cp. We call such a route an instance of cp. In our example, route $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$ is an instance of cluster path cp_1. Clearly, a cluster path satisfies G_Q, if and only if all its instances satisfy G_Q. The main idea of BFS is to enumerate all cluster paths that cover all query categories, while satisfying G_Q; for each such cluster path cp, BFS applies backward search to select its shortest instance, and the best one among all these instances is returned as the query result. Continuing the example, BFS computes the shortest instance of cp_1 by *backward joining* (described in Section 3.1) P_2 with P_3, and then the results with P_1, and finally with $\{q\}$. This resulting instance, i.e., $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_3$, is compared with the best instance of other cluster paths, e.g., $q \rightarrow P_3 \rightarrow P_1 \rightarrow P_2$. Since it is the shortest among all such instances, BFS returns it as the query answer.

Akin to SFS, BFS eliminates repeated computations by materializing optimal suffixes on the cluster path level, into an optimal cluster path suffix table $\mathbf{\Omega}$. In particular, each row in $\mathbf{\Omega}$ corresponds to a cluster P, and every column a category set V. The cell $\mathbf{\Omega}_{P,V}$ contains the set of routes, such that for each point $p \in P$, $\mathbf{\Omega}_{P,V}$ includes the shortest route that starts at p, and covers all categories in V. For instance, in our running example, $\mathbf{\Omega}_{P_3,\{restaurant,pub\}}$ consists of two shortest routes $p_5 \rightarrow p_3$ and $p_6 \rightarrow p_4$ that start at $p_5, p_6 \in P_3$ respectively, and visit a restaurant and a pub. BFS gradually fills $\mathbf{\Omega}$ as it performs the search; when the method needs the same optimal suffix information again later, it directly retrieves it from $\mathbf{\Omega}$. For instance, imagine that there is a fourth category "mall" in our example without any order constraint, and a cluster P_4 of such points. Suppose that BFS has traversed cluster path $q \rightarrow P_4 \rightarrow P_1 \rightarrow P_3 \rightarrow P_2$, and materialized $\mathbf{\Omega}_{P_3,\{restaurant,pub\}}$. Later, when BFS traverses to another prefix cluster path $q \rightarrow P_1 \rightarrow P_4 \rightarrow P_3$, instead of searching for a pub cluster again, BFS directly backward-joins $\mathbf{\Omega}_{P_3,\{restaurant,pub\}}$ with P_4, then the results with P_1, and finally with q, obtaining the shortest instance of any cluster path with prefix $q \rightarrow P_1 \rightarrow P_4 \rightarrow P_3$.

Algorithm 5 shows the pseudo-code of BFS, which closely resembles that of SFS (Algorithm 4), with cluster paths replacing concrete routes. Specifically, the recursive function BFS takes as input a current prefix cluster path with end cluster P, and a subgraph $G \rightarrow G_Q$ that involves a category set V. The function computes and materializes $\mathbf{\Omega}_{P,V}$. To do that, each invocation of BFS appends a cluster to the current prefix cluster path, and recursively calls itself with the longer prefix and one less category to visit (lines 7-14). The order of new clusters to be added to the current prefix is based on their MBRs' minimum distances to the MBR of the current last cluster P (line 7). After searching with such a new cluster P', BFS backtracks, and continues to test other new clusters. The optimal suffixes starting from a point in P' are combined with P through a backward join (line 11), whose results are used to update the optimal suffix set $\mathbf{\Omega}_{P,V}$. In our running example, after testing cluster path $q \rightarrow P_1 \rightarrow P_3 \rightarrow P_2$, BFS backtracks to P_3, backward joins P_2 with P_3, and uses the result to update $\mathbf{\Omega}_{P_3,\{restaurant,pub\}}$, i.e., the set of optimal suffixes starting at a point in P_3, and covering a restaurant and a pub. At this point, since P_2 is the only permissible cluster to be added after P_3, BFS finalizes $\mathbf{\Omega}_{P_3,\{restaurant,pub\}}$, backtracks again to P_1, and backward joins $\mathbf{\Omega}_{P_3,\{restaurant,pub\}}$ with P_1. The results contribute to $\mathbf{\Omega}_{P_1,\{museum,restaurant,pub\}}$. After that, as P_3 is also the only permissible cluster to append to P_1, BFS backtracks for a third time to q, and continues to test the next cluster (i.e., P_3) after q, starting a new branch for searching that eventually leads to $\mathbf{\Omega}_{P_3,\{museum,restaurant,pub\}}$. Finally, the query result is computed based on $\mathbf{\Omega}_{P_1,\{museum,restaurant,pub\}}$ and $\mathbf{\Omega}_{P_3,\{museum,restaurant,pub\}}$.

BFS integrates pruning strategies from both backward and forward search methods; the backward join steps apply elliptic pruning and the extension of the prefix cluster paths uses a variant of SFS pruning, based on the following lemma.

Lemma 6 (BFS Pruning)**.** *Given a query $\langle q, G_Q \rangle$, a point set P, and a category $C \in G_Q$ such that $P \not\subset C$, let P_C the nearest point cluster of P in C, and θ be a known upper bound for the length of the optimal solution to the query. If mindist(q, P)+mindist(P, P_C)$\geq \theta$, then, any route that starts at a point $p \in P$ and covers C cannot possibly be a sub-route of the optimal solution to the query.*

Proof: Let p_C be any point in P_C. Then, *dist*(q, p)+ *dist*(p, p_C)$\geq$*mindist*(q,P)+*mindist* (P, P_C)$\geq$ θ. Following Lemma 4, any route that starts at p and covers C cannot be a sub-route of the optimal solution. □

BFS pruning differs from SFS pruning in that the former applies to the cluster level, rather than individual point/route level. The usage of BFS pruning is also similar to that of SFS pruning; i.e., BFS finds the nearest cluster C of the farthest uncovered category, and tests whether the sum of distances from the current cluster P to the query point q and C (lines 5-6).

Finally, we clarify how BFS updates the upper bound θ. Recall that SFS adds the length l of the current prefix and the length of the optimal suffix, and compares the result with the current θ. In BFS, the current prefix is a

Algorithm 5 Batch forward search algorithm

InitBFS(q, G_Q)
// Input and Output: same as algorithm SBS
1: Same as lines 1-4 in algorithm SBS
2: Initialize all elements of $\mathbf{\Omega}$ to *Unknown*
3: Partition points in CS into clusters, such that points in the same cluster belong to the same category, and are close to each other
4: Call *BFS*(q,G_Q,{q},{q},θ), and return the only route in its result set

BFS(q,G,P,R,θ) // BFS stands for Batch forward search
// Input: q, G, θ: same as in algorithm SFS
// P: tail cluster of the current cluster-route
// R: set of shortest routes from q to each point in P, only includes routes with length shorter than θ
// Output: the set of all optimal routes that starts at a point in P, and covers all categories in G
1: Let V be the set of categories in G, category C_P be the category of all points in P, and set $V' = V \backslash \{C_P\}$
2: Construct new sub-graph G' by removing C_p and all related edges from G
3: **if** $\mathbf{\Omega}_{P,V} \neq$ *Unknown* **then** return $\mathbf{\Omega}_{P,V}$
4: **if** V contains a single category C_P **then** return a set of routes, each containing a single point in P
5: Compute the minimum distance from P to its nearest cluster of each category $C \in V$, and set d_{max} to the maximum value of these distances
6: **if** *mindist*(q, MBR_P)+$d_{max} \geq \theta$ **then** set $\mathbf{\Omega}_{P,V}$ to *Invalid*, and return *Invalid*
7: **for** each cluster $P' \in CS$ in increasing order of *mindist*(MBR_P, $MBR_{P'}$) **do**
8: **if** adding P' to the current cluster route violates G **then** Continue
9: Call R'=*FSJoin*(q, G, R, P', θ)
10: Recursively call $\mathbf{\Omega}_{P',V'} = BFS(q, G', P', R', \theta)$
11: Call *BSJoin*(q,G,P',V',P) and merge results to $\mathbf{\Omega}_{P,V}$
12: **if** $R \neq \emptyset$ **then**
13: Compute routes $r_1 \in R$, $r_2 \in \mathbf{\Omega}_{P,V}$ such that the end point of r_1 is the start point of r_2, and *length*($r_1 \in R$) +*length*(r_2) is minimized
14: **if** *length*(r_1)+*length*(r_2) $< \theta$ **then** Update θ and CS

cluster path and the algorithm computes the length of its shortest instance through *forward joins* (line 9), and uses it in place of l (lines 13-14). Algorithm 6 illustrates the forward join algorithm (*FSJoin*), which resembles backward join (Algorithm 2), with two important differences. First, FSJoin always adds a point to the end of a route, rather than the beginning as in BSJoin. Second and more importantly, the elliptic pruning used in BSJoin does not apply to FSJoin. Instead, the only pruning condition is that the resulting routes must have length shorter than the current threshold θ. This test is sufficient given the fact that the sole purpose of forward join is to improve the bound θ, and a prefix with length reaching or exceeding θ cannot possibly lead to a route with length shorter than θ.

Algorithm 6 Algorithm for forward search join

FSJoin(q, G, R, P, θ)
//Input: q, G: same as in algorithm *BFS*
// R, P: a set of routes and points respectively
// θ: length of a known route the satisfies the query
// Output: forward-join results of R and P
1: Initialize route set R' to empty
2: Partition R based on the set of categories they cover
3: **for** each point $p \in P$ **do**
4: **for** each partition R_V of R' covering the same category set V **do**
5: Find the route $r_V \in R_V$ that minimizes the length of $r_V \rightarrow p$ among all routes in R_V
6: **if** *length*($r_V \rightarrow p$) $< \theta$ **then** add $r_V \rightarrow p$ to R'

The space and time complexity of BFS is the same as the other proposed methods, since the former basically integrates the general forward search framework with the backward join module. The proof follows directly from the proofs of SBS and SFS, and is omitted for brevity. BFS combines the effective pruning of backward search and incremental bound tightening of forward methods, and, thus, enjoys the merits of both frameworks, while avoiding their drawbacks.

5 Size-l Optimal Route Processing

This section studies the *size-l* optimal route query. The objective is to retrieve shortest routes that cover an arbitrary subset of l categories from the given category set, where $l \leq m$ is a query parameter. All proposed methods SBS, BBS, SFS and BFS can be easily adapted to answer size-l optimal route queries. In particular, SBS and BBS are modified in two aspects. First, since the optimal route now has a length of l, the optimal suffix table now contains suffixes of length up to l, rather than m as in the original algorithms. Second, pruning of suffixes using partial order constraints becomes more complex, as not all categories are covered by the resulting route.

Table 4 shows the optimal suffix table for a size-2 optimal route query using the dataset and query graph of Figure 1. Comparing Table 4 with Table 3, the former includes only suffixes of lengths 1 and 2. Meanwhile, Table 4 contains suffixes covering category sets {restaurant} and {museum, restaurant} respectively. Such sub-routes are pruned in SBS and BBS for the original optimal route query, since they can only lead to complete routes (i.e., those covering all 3 categories) that visit a pub before a restaurant, violating G_Q. On the other hand, they are valid suffixes for size-2 routes that do not visit a pub at all. In general, pruning of suffixes based on G_Q must consider that the resulting route may omit an arbitrary combination of $m - l$ categories. Enumerating all such combinations may consume excessive CPU time, and, consequently, defeat the purpose of pruning. Hence, our implementation of SBS and BBS only eliminate suffixes that *directly* violate G_Q, e.g., those that contain a pub before a restaurant in our running example.

TABLE 4
Optimal suffix table for size-2 optimal route query

R_i	Start point	Categories Covered	Optimal Suffix
1	p_1	{museum}	p_1
	p_2		p_2
	p_3	{pub}	p_3
	p_4		p_4
	p_5	{restaurant}	p_5
	p_6		p_6
2	p_1	{museum, pub}	$p_1 \rightarrow p_3$
	p_2		$p_2 \rightarrow p_4$
	p_3		$p_3 \rightarrow p_1$
	p_4		$p_4 \rightarrow p_2$
	p_5	{restaurant, pub}	$p_5 \rightarrow p_3$
	p_6		$p_6 \rightarrow p_4$
	p_1	{museum, restaurant}	$p_1 \rightarrow p_5$
	p_2		$p_2 \rightarrow p_5$
	p_5		$p_5 \rightarrow p_1$
	p_6		$p_6 \rightarrow p_1$

The forward search methods are modified in a similar way to answer size-l optimal route queries. Specifically, both SFS and BFS now backtrack when the current path

(cluster path in BFS) reaches length l; additionally, both methods only detect direct violations of partial order constraints and backtrack accordingly. Finally, the worst-case complexity analyses of all methods are almost the same as for the original query, except that the length of an optimal suffix is now bounded by l rather than m. Thus, the space complexity of all 4 methods decreases to $O(N \cdot 2^m \cdot l)$, whereas their time complexity remains the same. In practice, however, a smaller value of l usually reduces computational costs considerably, as we demonstrate experimentally.

6 Experimental Evaluation

We implemented all 4 algorithms SBS, BBS, SFS, and BFS in C++ and ran our experiments on an Intel Xeon X5400 3.16GHz CPU with 32GBytes of RAM. In each experiment, we issue 1000 queries, and report their average response time. We do not report I/O cost separately, because for all methods in all settings, the I/O time is no more than a few milliseconds; i.e., at least an order of magnitude lower than the total response time. All queries are generated randomly, with their start points uniformly distributed in the entire space.

Table 5 summarizes the parameters under investigation, with their default values in bold. Specifically, m is the total number of categories to be visited. The visit order graph G_Q is another parameter. Since the properties of G_Q are rather difficult to quantify, we consider three specific types of visit order graphs: one without any order constraints between data categories, one with a complete order of all categories, and a bipartite graph that requires half (i.e., $m/2$) of the categories be visited before the remaining ones. Zero-order and total-order graphs are common definitions used in previous studies on the optimal route query, e.g., [11], [13], and the bipartite graph has properties between the two extremes. In addition, we classify queries based on the effectiveness of the greedy algorithm. In particular, let $a\%$ be the ratio between the length of the optimal route and that obtained by the greedy algorithm. We partition the queries into 4 groups based on their values of $a\%$, and report the average response time for each group of queries. Finally, for the synthetic dataset, there are two additional parameters: the total number N of points in all categories and the ratio $c\%$ between the number of points in a larger category to that of a smaller category (see Section 6.1 for details). For each set of experiments, we vary the value of one parameter, and fix other parameters to their default values.

TABLE 5
Parameter values in the experiments

Parameter	Range & Default
m	4, **6**, 8, 10
Type of G_Q	**No order**, bipartite graph, total order
$a\%$	0%-25%, 25%-50%, 50%-75%, 75%-100%
N (synthetic data)	**500k**, 1000k, 1500k, 2000k
$c\%$ (synthetic data)	1%, 10%, 100%

6.1 Experimental Data

Two real datasets were retrieved from their websites in January 2011. The complete *Pocket* (from *www.pocketgpsworld.com*) and *GeoName* (from *www.geonames.org*) datasets contain 168,197 points in 39 categories, and 2,060,001 points in 222 categories, respectively. We randomly selected 10 larger categories from each dataset. Table 6 summarizes the categories in *Pocket* and *GeoName* used in our experiments.

TABLE 6
Categories used in real datasets

Pocket **categories**	**# points**	*GeoName* **categories**	**# points**
Association	3,539	Hospital	13,286
Commercial Site	4,807	Mall	15,789
Hospital	5,691	Campus	17,015
Bank	7,116	Airport	19,543
Supermarket	8,109	Post Office	29,423
Wi-Fi Hotspot	9,543	Park	66,002
Car Services	10,077	Building	74,037
Fuel	10,689	Cemetery	129,595
ATM	12,573	Popular Place	181,535
Food &Drink	21,287	School	197,991
Total	93,431	**Total**	744,216

The synthetic dataset uses real locations from the California Road dataset (available at *www.rtreeportal.org*) and categorical information generated as follows. First, we randomly selected N points from the California Road dataset. Then, we decide the number of points in each cluster. Specifically, half (i.e., $m/2$) of the categories are assigned a larger cardinality n_l, while the remaining ones have a smaller cardinality n_s. The ratio $c\% = n_s/n_l$ is a parameter under investigation. Finally, for each category C, we select a random set of points among those that have not been assigned to any categories before. This process ensures that all categories in *Synthetic* follow the same distribution as the underlying point set.

Table 7 shows statistics on the greedy bound quality in the three datasets. For instance, in the Pocket dataset, 203 out of 1000 queries have $a\%$ value below 25%; i.e., the greedy route is at least 4 times as long as the optimal route. Observe that queries in *Pocket* tend to yield good greedy bounds; those in *GeoName* are more likely to have the poor bounds; and the quality of greedy bounds in the synthetic data lies between the other two datasets. This is reflected in the experimental results.

TABLE 7
Quality of the greedy bound

	Pocket	*GeoName*	*Synthetic*
$0 \leq a\% < 25\%$	203 queries	231 queries	133 queries
$25\% \leq a\% < 50\%$	247 queries	312 queries	297 queries
$50\% \leq a\% < 75\%$	201 queries	205 queries	285 queries
$75\% \leq a\% < 100\%$	349 queries	252 queries	285 queries

6.2 Experiments Using Real Datasets

We first compare SBS with PLUB [11], which decomposes an optimal route query into multiple total orders (see Section 2.1). Table 8 shows the speedup of SBS compared to PLUB with varying values of m, e.g., the value 5.0

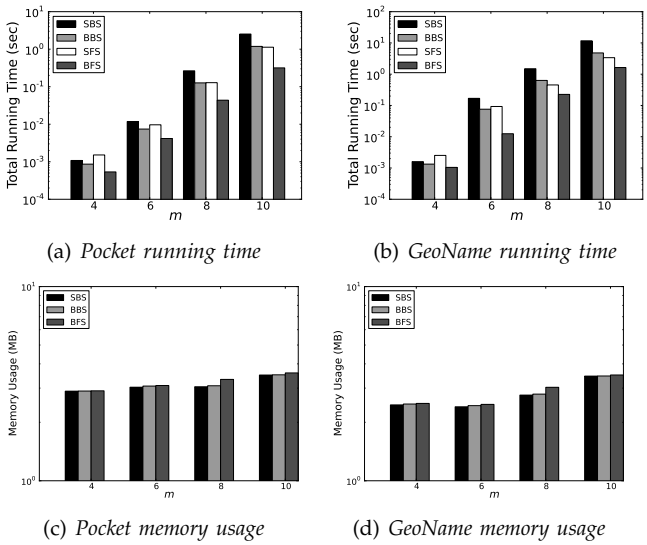
indicates that SBS is 5 times faster than PLUB. SBS is always faster than PLUB, with at least 30% speedup. The advantage of SBS is more pronounced when m is smaller. The main reason is that SBS prunes with only the bound θ obtained by the greedy algorithm, whereas PLUB tightens the the bound whenever a total-ordered subquery returns a better result. As m grows, the accuracy of the greedy method worsens, and, consequently, the performance gap between SBS and PLUB gradually closes. Nevertheless, in all settings of our experiments, SBS always outperforms PLUB; additionally, SBS is usually the least efficient among the proposed methods. Hence, in the next experiments, we exclude the results for PLUB.

TABLE 8
Speedup of SBS with respect to *PLUB*

Dataset	m=4	m=6	m=8	m=10
Pocket	5.0	4.6	2.3	1.3
GeoName	6.4	1.6	2.0	1.6

Figure 6 illustrates the effects of varying number of categories m in the query. Figure 6(a) and Figure 6(b) plot the total response time as a function of m for the *Pocket* and *GeoName* datasets, respectively. The response time of all methods increases exponentially with m, reflecting their exponential time complexity with respect to m. Nevertheless, in practice, queries that cover a large number of categories are expected to be rare. Among all methods, BFS is consistently the most efficient one. Meanwhile, both batch methods consistently beat their simple counterparts in all settings. Comparing BBS and SFS, the former is faster for low values of m. As m grows, however, the performance gap between the two diminishes, and SFS starts to outperform BBS when $m \geq 8$. The main reason is that with increasing m, the optimal route becomes longer and more complex; consequently, the difference in length between the optimal route and the greedy one becomes higher. Since backward search methods prune based solely on the greedy bound, they tend to maintain and process a large number of useless sub-routes. On the other hand, the forward search strategy incrementally improves the bound θ during search, which helps prune more of the search space. The high overhead of SFS for low values of m is due to its weaker pruning condition, which is remedied in BFS, as the latter integrates backward search components.

Regarding main memory consumption, we found that SFS consumes a very large amount of RAM; the remaining three methods incur similar RAM usage, as shown in Figures 6(c) and 6(d). The main reason is that SFS has poor pruning power, requiring the storage of a large number of intermediate sub-routes. The memory consumption of all methods grows with m, since a query with more categories to visit involves a larger number of materialized sub-routes. Comparing SBS, BBS and BFS, SBS consumes slightly less memory than BBS and BFS, since the latter two use more sophisticated pruning, which requires more memory for bookkeeping. Nevertheless, this increased memory usage is more than



(a) *Pocket running time* (b) *GeoName running time*

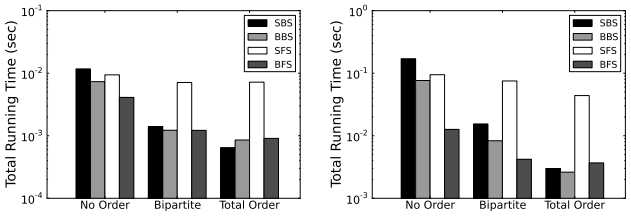
(c) *Pocket memory usage* (d) *GeoName memory usage*

Fig. 6. Effect of the number of query categories

compensated by the good runtime performance of BBS and BFS as described above. Note that memory usage increases at a significantly slower pace than running time; meanwhile, with the single exception of SFS, all methods use less than 1 MBytes of RAM, even for relatively large values of m, e.g., 10. These facts suggest that running time is the bottleneck of the query. Thus, we only compare the runtimes of the 4 methods, next.

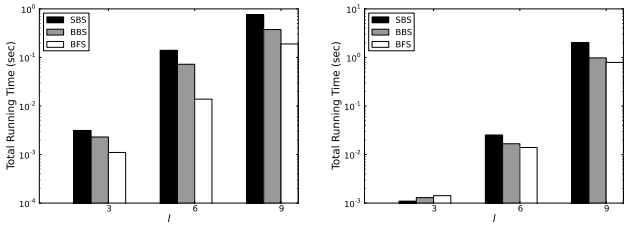
Figure 7 investigates the impact of the visit order graph G_Q, after setting $m = 6$. As expected, G_Q with no order constraint leads to the highest computational costs, since there are an exponential number of category permutations, enlarging the search space for possible routes. As the amount of order constraints increases, the response times of all methods drop. BFS is the winner for all settings without a total order. In the case of totally-ordered queries, the response times of SBS, SFS and BFS range in milliseconds, and are very close to each other. Consequently, their relative performance is affected more by random fluctuations. Batch processing, again, has obvious benefits, except that SBS is slightly faster than BBS on the *Pocket* dataset with a total order. The cost of SFS drops less rapidly than the other 3 methods, due to the overhead of the numerous recursive calls, and generally ineffective pruning. These problems, however, do not occur to BFS, as the latter uses backward search components. SBS is very competitive when there is a total order; however, its performance becomes relatively poor, when there are less strict order constraints. The reason is that batch processing and forward search focus on *pruning* sub-routes. When there is a total order, the search space is relatively small, and the benefits of pruning are offset by the overhead of doing so. With more relaxed order constraints, the search space becomes larger, and the effect of pruning becomes more pronounced; thus, BBS and BFS outperform SBS.

Figure 8 studies the impact of the query start point q.



(a) *Pocket* (b) *GeoName*

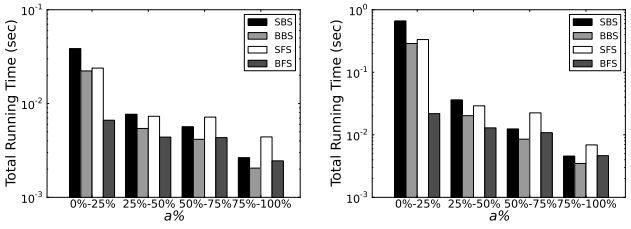
Fig. 7. Effect of the type of G_Q



(a) *Pocket* (b) *GeoName*

Fig. 9. Effect of l for size-l optimal route queries

The influence of q is measured by the ratio $a\%$ between the length of the optimal route and that of the greedy route. The number of queries with value $a\%$ falling in ranges 0-25%, 25%-50%, 50%-75%, 75%-100% is listed in Table 7. For queries where $a\%$ is low, the benefit of BFS is more pronounced, as it gradually shrinks the set of candidate points, and reduces the number of unnecessary sub-route computations. Notably, when $a\% < 25\%$, BFS is over an order of magnitude faster than the remaining methods. SFS has a similar advantage; however, unlike BFS, SFS is heavily burdened by ineffective pruning, leading to uncompetitive response times. The performance of SBS and BBS, on the other hand, is significantly affected by $a\%$, since their pruning effectiveness depends upon the quality of the greedy bound. The use of batch processing alleviates this problem, though the efficiency of BBS still lags behind that of BFS.

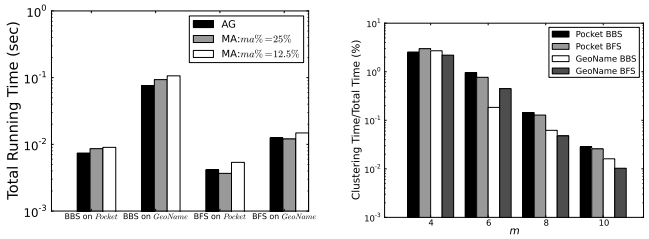


(a) *Pocket* (b) *GeoName*

Fig. 8. Effect of query position

Figure 9 evaluates the proposed methods for size-l optimal route queries. We omit the results for SFS since it is subsumed by BFS. Clearly, a smaller value of l reduces the computational costs dramatically for all methods, since backward search terminates after l rather than m joins, and forward search also backtracks after fewer steps. The relative performance of different methods remains the same as in the case for the original optimal route queries.

Finally, Figure 10 evaluates the clustering modules in algorithms BBS and BFS. Figure 10(a) illustrates the response times of BBS and BFS, using three different clustering methods: *AG*, *MA* with $ma\% = 25\%$, and *MA* with $ma\% = 12.5\%$ (see Section 3.2). All other parameters are set to their default values. The results indicate that *AG* is the best choice among the three for BBS, whereas *MA* with $ma\% = 25\%$ is most suitable for BFS. Meanwhile, the different clustering modules may lead to considerable differences in performance, especially on the *GeoName* dataset. Further tests confirm that these observations hold for other settings as well. In particular, the best value for the parameter $ma\%$ in *MA* is around 22%-28%. Figure 10(b) shows the ratio between the time used by clustering in algorithms BBS and BFS and their respective total running time, varying m. In these experiments, MA is used for clustering with $ma\% = 25\%$, and all other parameters are fixed to their defaults. The results show that the overhead for clustering in both BBS and BFS is only a fraction (less than 5%) of their total computational costs. This is much smaller than the performance advantage of BBS (resp. BFS) over SBS (resp. SFS), therefore clustering is generally worth paying for.



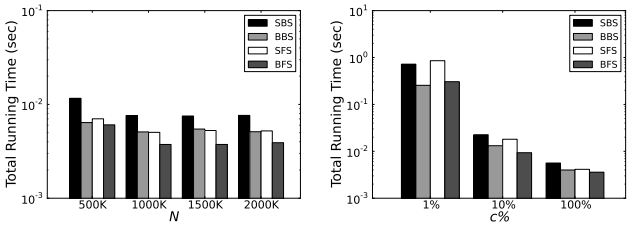
(a) *Different clustering techniques* (b) *Overhead of clustering*

Fig. 10. Evaluation of the clustering module

6.3 Experiments Using Synthetic Datasets

We repeat all experiments on the synthetic dataset. Due to space constraints, we do not show the impact of parameters m, G_Q and $a\%$ on the performance of the algorithms; the results are very similar as the case of real data. Figure 11(a) shows the effect of data cardinality N after setting all other parameters to their default values. Surprisingly, the response times of all algorithms decrease with growing N; because categories are randomly assigned to points in *Synthetic*, when more points are included in the dataset, every category becomes denser. Consequently, the greedy algorithm is more likely to identify a good route whose length is close to the optimal one, meaning that the candidate set CS becomes smaller, leading to decreased join costs. Figure 11(b) shows the impact of different relative category sizes. In this experiment, N is fixed, and we vary the ratio $c\%$

of the cardinality of larger categories to that of smaller categories. When categories become highly imbalanced, all methods become considerably slower, because the optimal route becomes longer in order to reach points of rare categories; this reduces the effectiveness of pruning. The results suggest that the scalability issue of the optimal route query is rather complicated, and it is an interesting topic for future studies.



(a) Effect of N (b) Effect of $c\%$

Fig. 11. Experiments on synthetic data

Summing up, BFS is the most efficient and robust solution overall, with practically low response times (10^{-2}-10^{-1} sec.). If there are strict order constraints or the greedy algorithm returns a relatively accurate bound, backward search solutions also achieve competitive performance, with BBS typically outperforming SBS. In these situations, it is preferable to apply BBS due to its simpler implementation. For total-order queries, SBS (and, thus, R-LORD [13] which SBS reduces to) is still highly efficient. Finally, although the performance of all methods deteriorates with increasingly long routes, they are robust against large datasets. This indicates that the optimal route service can effectively control its workload by adjusting the maximum permissible route length.

7 Conclusion

This paper investigates the problem of optimal route query processing. Existing solutions are either limited to a specific setting of the problem, or incur expensive, redundant computations. Hence, we propose novel and efficient solutions, based on two methodologies: backward and forward search. The solution BFS that combines merits from both backward and forward search achieves the best performance. In the future, we plan to study alternative definitions of the optimal route query, that have temporal constraints (e.g., have lunch at a specified period) or maximize the number of categories to be visited given a total travel length budget.

References

[1] S. Asadi, X. Zhou, and G. Yang. Using local popularity of web resources for geo-ranking of search engine results. *Springer World Wide Web*, 12(2):149–170, 2009.
[2] A. R. Butz. Alternative algorithm for hilbert's space-filling curve. *IEEE Trans. Comput.*, 20:424–442, 1971.
[3] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.
[4] H. Chen, W. S. Ku, M. T. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *GIS*, 2008.
[5] Y. Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.
[6] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, 2010.
[7] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
[8] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.
[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[10] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. H. Teng. On trip planning queries in spatial databases. In *SSTD*, 2005.
[11] X. Ma, S. Shekhar, H. Xiong, and P. Zhang. Exploiting a page-level upper bound for multi-type nearest neighbor queries. In *GIS*, 2006.
[12] D. Martinenghi and M. Tagliasacchi. Proximity rank join. *PVLDB*, 3(1):352–363, 2010.
[13] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.
[14] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*, 2011.
[15] M. Y. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k spatial preference queries. In *ICDE*, 2007.
[16] D. Zhang, Y. M. Chee, A. M., A. Tung, and M. Kitsuregawa. Keyword search in spatial databases: towards searching by document. In *ICDE*, 2009.
[17] D. Zhang, B. C. Ooi, and A. Tung. Locating mapped resources in web 2.0. In *ICDE*, 2010.

Jing LI received the BSc degree in computer science and engineering from Nanjing University. He is currently working toward the PhD degree at the Department of Computer Science, University of Hong Kong. His research interests include data privacy, query processing in spatio-temporal databases and graph databases, data mining in textual data streams.

Yin David Yang is a Research Scientist at the Advanced Digital Sciences Center (ADSC), Singapore, and a Principle Research Affiliate at the University of Illinois at Urbana Champaign, Illinois, USA. He obtained his PhD in Computer Science from the Hong Kong University of Science and Technology (HKUST) in 2009. Before joining ADSC, David worked as an instructor for undergraduate courses at HKUST, and later as a post-doc at the University of Hong Kong. His research interests include database security and privacy, spatial and temporal query processing, and distributed systems.

Nikos Mamoulis received a diploma in Computer Engineering and Informatics in 1995 from the University of Patras, Greece, and a PhD in Computer Science in 2000 from the Hong Kong University of Science and Technology. He is currently a professor at the Department of Computer Science, University of Hong Kong, which he joined in 2001. His research focuses on management and mining of complex data types, privacy and security in databases, and uncertain data management. He served as PC member in more than 80 international conferences on data management and mining. He is an associate editor for IEEE TKDE and the VLDB Journal.