# Multi-Approximate-Keyword Routing in GIS Data

Bin Yao[1] Mingwang Tang[2] Feifei Li[2]
yaobin@cs.sjtu.edu.cn[1], {tang,lifeifei}@cs.utah.edu[2]
Department of Computer Science and Engineering, Shanghai JiaoTong University[1]
School of Computing, University of Utah[2]

## ABSTRACT

For GIS data situated on a road network, shortest path search is a basic operation. In practice, however, users are often interested at routing when certain constraints on the textual information have been also incorporated. This work complements the standard shortest path search with multiple keywords and an approximate string similarity function, where the goal is to find the shortest path that passes through at least one matching object per keyword; we dub this problem the multi-approximate-keyword routing (MAKR) query. We present both exact and approximate solutions. When the number $\kappa$ of query keywords is small (e.g., $\kappa \leq 6$), the exact solution works efficiently. However, when $\kappa$ increases, it becomes increasingly expensive (especially on large GIS data). In this case, our approximate methods achieve superb query efficiency, excellent scalability, and high approximation quality, as indicated in our extensive experiments on large, real datasets (up to 2 million points on road networks with hundreds of thousands of nodes and edges). We also prove that one approximate method has a $\kappa$-approximation in the worst case.

## Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems. Subject: Query processing*

## General Terms

Algorithms, Performance

## 1. INTRODUCTION

GIS data refer to the full spectrum of digital geographic data. Often time, objects in GIS data consist of both spatial and textual information, where the spatial information represents the location of the object and the textual information contains a set of keywords (in string format) describing the object at a particular location. The most useful GIS data usually situate on a road network, e.g., the OpenStreetMap project, where a basic operation is to find the shortest path for any given source and destination pair $(s, e)$. In practice, however, users are often interested at routing between $(s, e)$ when certain constraints on the textual information have been incorporated as well. This work complements the standard shortest path search with multiple keywords and an approximate string similarity function, where the goal is to find the shortest path that passes through at least one matching object per keyword; we dub this problem the multi-approximate-keyword routing (MAKR) query.

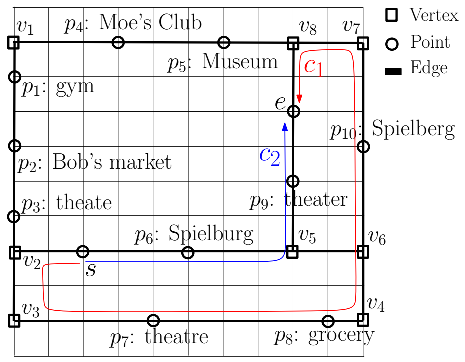


**Figure 1: Example: $\psi = \{(\textbf{theater}, 2), (\textbf{Spielberg}, 1)\}$.**

There are a lot of possible metrics for matching keywords, and keyword matching based on approximate string similarity using the *string edit distance* is a popular choice [9]. Specifically, given two strings $\sigma_1$ and $\sigma_2$, the edit distance between $\sigma_1$ and $\sigma_2$, denoted as $\varrho(\sigma_1, \sigma_2)$, is defined as the minimum number of *edit operations* required to transform one string into the other. The *edit operations* refer to an insertion, deletion, or substitution of a single character. For example, let $\sigma_1$ ='Spielburg' and $\sigma_2$ ='Spielberg', then $\varrho(\sigma_1, \sigma_2) = 1$, by substituting the first 'u' with 'e' in $\sigma_1$. The standard method for computing $\varrho(\sigma_1, \sigma_2)$ is a dynamic programming formulation. For two strings with lengths $|\sigma_1|$ and $|\sigma_2|$ respectively, it has a complexity of $O(|\sigma_1||\sigma_2|)$.

That said, an example of an MAKR query is given in Figure 1, where a road network consisting of vertices (squares), edges (bold line segments) and a set of points of interest (circles) residing on the edges is shown. This road network is aligned in a grid with unit length to ease the illustration. Each point of interest is associated with a string to describe its information (e.g., name, service). Suppose that a family wants to visit a "theatre" and a bar like "Spielberg" from $s$ to $e$ with the shortest distance. Clearly, due to errors/uncertainty in the database and/or the query keywords, a path that involves the locations named as "theater" and "Spielburg" should also be a candidate answer. Furthermore, in many cases the order in which they visit these desired locations does not matter. This is precisely the multi-approximate-keyword routing (MAKR) query. In this example, $c_1 = \{s, p_7, p_{10}, e\}$ and $c_2 = \{s, p_6, p_9, e\}$ can be candidate paths (there are more). Note that a segment in a path is defined by the shortest path connecting its two end-points. For example, the segment $\{p_7, p_{10}\}$ in $c_1$ is given by $p_7 \to v_4 \to v_6 \to p_{10}$ in Figure 1. The challenge in an MAKR query is to find the shortest path among all candidate paths efficiently. In this case, $c_2$ is the answer.

Suppose there are $\kappa$ query keywords and they are specified (together with the corresponding edit distance thresholds) in a set $\psi$. Answering an MAKR query exactly is a very challenging problem. In fact, we can show that it is NP-hard w.r.t $\kappa$. Nevertheless, when the number of query keywords is small, we show that it is possible to design efficient algorithms to answer MAKR queries exactly. When $\kappa$ and/or the dataset are large, we resort to efficient and effective approximate solutions. More precisely, our main contributions in this work are summarized as follows:

- We formalize the notion of MAKR queries in Section 2.
- We present exact solutions for MAKR queries in Section 4, which are also extended to top-$k$ MAKR queries.
- We show that MAKR is NP-hard. Thus, we also design three approximation algorithms in Section 5. They are efficient and easy to implement. We also prove that one is a $\kappa$-approximation in the worst case compared to the exact solution (in addition, the approximations in practice are usually much better).
- We show empirically the efficiency and effectiveness of our methods in Section 7. Our study uses large real road networks (up to 2 million points on road networks with hundreds of thousands of nodes and edges). Our approximate solutions are particularly appealing, where they can answer an MAKR query up to 10 query keywords on large datasets in less than a second to a few seconds, with high quality approximations.

We introduce relevant background (of some basic techniques) for our algorithms in Section 3, and survey the related work in Section 8. The paper concludes in Section 9.

## 2. PROBLEM FORMULATION

Formally, a GIS database contains a set $P$ of points on a road network $G = (V, E)$. Without loss of generality, we assume that each point is associated with only one keyword. That said, a point from $P$ is defined by $(p_i, w_i)$ where $p_i$ represents its location and $w_i$ is the associated keyword. Generalizing our problem to handle multiple keywords per point is straightforward and discussed in Section 6.

An MAKR query is specified by two points $s$ and $e$, as well as a set $\psi$ of $\kappa$ query keyword and threshold value pairs $\{(\sigma_1, \tau_1), \ldots, (\sigma_\kappa, \tau_\kappa)\}$, where $\sigma_i$ is the $i$th query keyword and $\tau_i$ is the desired edit distance for deciding whether a keyword from an object matches with $\sigma_i$ under $\varrho$.

The road network is an undirected connected graph $G = (V, E)$, where $V$ ($E$) denotes the set of vertices (edges) in $G$. A point $p \in P$ locates on an edge $e \in E$. The distance between any two points (or vertices), denoted as $d(p_i, p_j)$, is the length of the shortest path connecting them. We use $w(p)$ to denote the string associated with $p$, e.g., $w(p_i) = w_i$.

**Definition 1** *Given an MAKR query $q = (s, e, \psi)$, a path $c = \{s, p_{x_1}, p_{x_2}, \cdots, p_{x_\ell}, e\}$, where $p_{x_i} \in P$ for $i = 1, \ldots, \ell$ and $\ell \leq \kappa$, is a complete candidate path for $q$ if and only if for any $(\sigma_i, \tau_i) \in \psi$, there exists $j \in [1, \ell]$, such that $\varrho(w_{x_j}, \sigma_i) \leq \tau_i$; and for any $j \in [1, \ell]$, $\exists i \in [1, \kappa]$ such that $\varrho(w_{x_j}, \sigma_i) \leq \tau_i$ (we say $p_{x_j}$ covers $(\sigma_i, \tau_i)$). We denote the set of all complete candidate paths as $\mathcal{A}_c$.*

| Symbol | Description |
|---|---|
| $G = (V, E)$ | road network with vertex (edge) set $V$ ($E$) |
| $P$ | the set of points of interest on $G$ |
| $d(p_1, p_2)$ | network distance between $p_1$ and $p_2$ |
| $d^-(p_1, p_2)$ | a lower bound of $d(p_1, p_2)$ |
| $d(c)$ | the length of the path $c$ |
| $d^-(c)$ | the lower bound of $d(c)$ |
| $P(c), W(c)$ | set of points, their strings on a path $c$ |
| $\|c\|$ | number of points on the path $c$, $\|c\| = \|P(c)\|$ |
| $\psi$ | the set of query (keyword, threshold) pairs |
| $\psi(c)$ | set of query keywords covered by a path $c$ |
| $\kappa$ | number of pairs in $\psi$ |
| $(\sigma_i, \tau_i)$ | $i$th query string and associated threshold |
| $s, e$ | the starting and ending points for a path |
| $w(p)$ | the string associated with a point $p$ |
| $\varrho(w_1, w_2)$ | the edit distance between strings $w_1$ and $w_2$ |
| $P(\sigma, \tau)$ | subset of points from a point set $P$ with their strings satisfying $\varrho(w, \sigma) \leq \tau$ |
| $P(\psi)$ | $\{P(\sigma_1, \tau_1), \ldots, P(\sigma_\kappa, \tau_\kappa)\}$ |
| $\mathcal{A}_c$ | set of complete candidate paths |

**Table 1: Frequently used notations**

We omit any road network vertices and query-irrelevant points from a path in Definition 1. A path $c$ is still well-defined as any two consecutive points in $c$ are connected by their shortest path. We denote the path between the $i$th point and $(i+1)$th point in a path $c$ *the ith segment* of $c$ ($s$ is the 0th point and $e$ is the $(\ell+1)$th point). The length of a path $c = \{s, p_{x_1}, p_{x_2}, \cdots, p_{x_\ell}, e\}$, $d(c)$, is defined as follows:

$$d(c) = d(s, p_{x_1}) + d(p_{x_1}, p_{x_2}), \ldots, +d(p_{x_\ell}, e).$$

For any path $c$, we use $P(c)$ to denote the set of points on $c$, and $W(c)$ to denote the set of strings from $P(c)$. A query keyword $(\sigma_i, \tau_i)$ is said to be *covered by* $c$ if $\exists w \in W(c)$ such that $\varrho(w, \sigma_i) \leq \tau_i$. The set of query keywords from $\psi$ that has been covered by $c$ is denoted as $\psi(c)$. Clearly, $\psi(c) \subset \psi$ for a *partial candidate path* (e.g., $\{s, p_3\}$ in Figure 1) and $\psi(c) = \psi$ for a *complete candidate path* (e.g., $c_1$ and $c_2$ in Figure 1). Note that for a complete candidate path $c$, $\psi = \psi(c) \subseteq W(c)$ and $|P(c)| \geq \kappa + 2$, as $c$ may pass through irrelevant points in the shortest path in any of its segment.

**Definition 2** *A multi-approximate-keyword routing (MAKR) query $q = (s, e, \psi)$ asks for a complete candidate path $c^* \in \mathcal{A}_c$ with the smallest length, i.e., $c^* = \operatorname{argmin}_{c \in \mathcal{A}_c} d(c)$.*

We can easily generalize an MAKR query to a top-$k$ MAKR query, where the answer consists of the top-$k$ shortest candidate paths from $\mathcal{A}_c$, instead of only the shortest one.

Lastly, suppose we have an algorithm $A$ that can retrieve a complete candidate path $c$ for any MAKR query $q$ where the optimal path is $c^*$. If $d(c^*) \leq d(c) \leq \alpha d(c^*)$, then $A$ is called an $\alpha$-approximate algorithm, and $c$ is an $\alpha$-approximate path. For a summary of the notations frequently used in the paper, please refer to Table 1.

## 3. PRELIMINARIES

Our goal is to handle large-scale GIS data on road networks. Thus, we need to design and leverage on disk-based data structures to facilitate our algorithms. We discuss these structures in this section.

We adopt a disk-based storage model for the representation of a road network, which groups network nodes based on their connectivity and distance, as proposed in [14]. We introduced some necessary adjustment to this model to make it work for our algorithms. Figure 2 illustrates an instance of our model for the road network shown in Figure 1. In our model, the adjacency list of the road network nodes and the points of interest on the network edges, along with their associated strings, are stored in two separate files, the adjacency list file and the points file respectively. Each file is then indexed by a separate B+ tree.

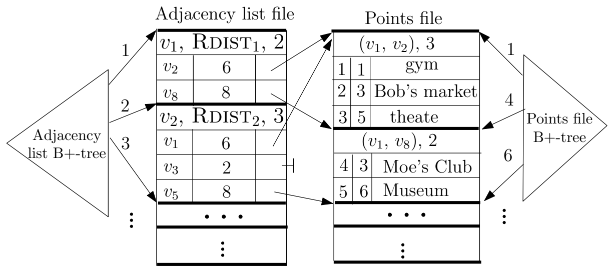


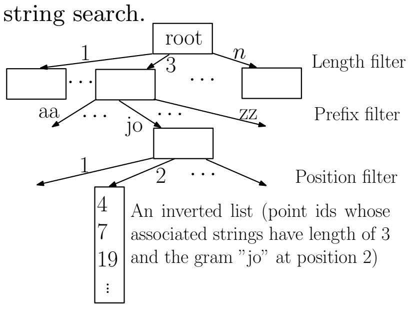
**Figure 2: Disk-based storage of the road network.**

To facilitate our query algorithms, a small set $V_R$ of nodes from $V$ is selected as the *landmarks*. The distance between two nodes, two points, or a node and a point is the length of the shortest (network) path connecting two objects of concern. For each node $v_i \in V$, at the beginning of its adjacency list, we store its distance to each landmark in $V_R$ (the set of all such distances is collectively denoted by RDIST$_i$ in Figure 2) and the number of adjacent nodes of $v_i$ (e.g., 2 for $v_1$ in Figure 2). How to select $V_R$ and compute RDIST$_i$ efficiently for each node $v_i$ are discussed in Section 4, when RDIST$_i$ is used in our query algorithms. Next in the adjacency list of $v_i$, for each adjacent node $v_j$ of $v_i$, we store the adjacent node id, the length of the edge $(v_i, v_j)$ and a pointer to the points group in the points file for points of interest on the edge $(v_i, v_j)$. If an edge does not contain any point, a null pointer is stored. The adjacency lists of all nodes are stored in the adjacency list file and they are ordered in the ascending order of the node id. A B+ tree is built on the adjacency list file. The key of this tree is the node id and the value is a pointer to its adjacency list. For example, in Figure 2, given the node id $v_1$, we can find its adjacency list from the B+ tree which contains $v_1$'s RDIST$_i$, number of adjacent nodes, and each of its adjacent nodes ($v_2$ and $v_8$): their distances to $v_1$ and the pointers to the points group on each corresponding edge.

The points file collects and stores the *points group* (points on the same edge) on all edges of the road network. In each points group, we first store the edge containing this points group and the number of points in it. Subsequently, for each point, we store the point id, the distance of this point to *the node with smaller id* on this edge, and its associated string. The ids of points are assigned in such a way that for points on the same edge ($v_i$, $v_j$), points are stored by their offset distances *to the node with smaller id* in ascending order, and their ids are then sequentially assigned (crossing over different edges as well). Note that these points groups are stored in the *points file* in the ascending order of the (smaller) node id defining an edge.

For example, the points file in Figure 2 partially reflects the example in Figure 1 and it indicates that the edge $(v_1, v_2)$ contains three points $\{p_1, p_2, p_3\}$; $d(p_1, v_1) = 1$ and $w_1$ is "gym", $d(p_2, v_1) = 3$ and $w_2$ is "Bob's market", and $d(p_3, v_1) = 5$ and $w_3$ is "theate". For any edge $\{v_i, v_j\}$, we represent it by placing the node with the smaller id first. If $v_i < v_j$, then in the adjacency list of $v_j$, the entry for $v_i$ will have its pointer of the points group pointing to the points group of $(v_i, v_j)$, i.e., no duplication of points group will be stored. For example, the pointer of the points group for $v_1$ in the adjacency list of $v_2$ points to the points group of $(v_1, v_2)$ as shown in Figure 2. A B+ tree is also built on the points file, where each data entry corresponds to a points group with keys being the first point id of each points group and values being the pointer to the corresponding points group. Clearly, this allows fast retrieval of a point $p$ and the points group containing it from the points file, using $p$'s id.

Our storage model is simple and effective. It supports our query algorithms seamlessly and efficiently. Our design is partly inspired by the adjacency list module in [12, 14]. Furthermore, we also build auxiliary string indices to index the collections of strings in a road network, for efficient approximate string search.



root
1
3
n
Length filter
aa
jo
zz
Prefix filter
1
2
Position filter
4
7
19
An inverted list (point ids whose associated strings have length of 3 and the gram "jo" at position 2)

**Figure 3: An example of a FilterTree from [10].**

Given a collection of strings, we use the FilterTree [10] to index them. The FilterTree combines several string filters (e.g., length filter, position filter, count filter) with the $q$-gram inverted lists for a collection of strings and organizes these filters in a tree structure, in order to support efficient approximate string search. An example of a FilterTree is shown in Figure 3. The nodes in the first level of the tree are grouped by the string length. The second level nodes are grouped by different grams (in this example, 2-grams). Then we identify the children of each gram by the position of that gram in the corresponding strings. Each such child node maintains an inverted list of point ids (whose associated strings contain this gram at a particular position of a given length). Using our structures, these point ids can be used to search the B+ tree of the points file to retrieve the corresponding strings and other relevant information.

In our algorithms, we either build one FilterTree for the entire collection of strings on a road network, or multiple FilterTrees for a set of disjoint partitions of the road network (one for each collection of strings from a partition). The details will become clear when we introduce the respective algorithms. Given a collection of points $P$ with strings, a query string $\sigma$ and a threshold $\tau$, the FilterTree on $P$ can efficiently find all points $P^+(\sigma, \tau)$ in $P$ that are most likely having their strings' edit distances to $\sigma$ less than or equal to $\tau$, possibly with false positives (but there are no false negatives). The exact set of points, denoted by $P(\sigma, \tau)$, that are similar to $\sigma$ with edit distances less than or equal to $\tau$, can then be identified by calculating the exact edit distances between every candidate point in $P^+(\sigma, \tau)$ and $\sigma$.

Lastly, Dijkstra's algorithm [5] is the classical method for finding the shortest path between two points on a graph. ALT algorithm [8] has introduced significant improvement by reducing the expansion area of Dijkstra's algorithm. The basic idea in ALT algorithm is to use a small set of reference nodes from the graph, then to compute and store the network distances between each node and each reference node. Using simple triangle inequality, lower bound of the distance between two points can be easily obtained using these reference distances; during the expansion from the source to the destination, a node that offers the smallest possible distance from the source to the destination (through it) is selected as the next node in the expansion (which is achieved by leveraging the lower bound for the distance between a node and the destination). The landmarks in our data structure serve as the reference nodes, and support the calculation of the lower bound distance of any two nodes (or two points, or a point and a node) efficiently.

Consider deriving $d^-(p_i, p_j) \leq d(p_i, p_j)$, without loss of generality, suppose $p_i \in (v_1, v_2)$ and $p_j \in (v_3, v_4)$. From $\textsc{Rdist}_i$ (see Figure 2), we can find the distance between a node $v_i$ and any landmark (reference node) in $V_R$. For any reference node $v_r$, clearly, $d(p_i, p_j) \geq |d(p_i, v_r) - d(p_j, v_r)| = d^-(p_i, p_j)$. Note that $d(p_i, v_r) = \min(d(p_i, v_1) + d(v_1, v_r), d(p_i, v_2) + d(v_2, v_r))$, where $d(v_1, v_r)$, $d(v_2, v_r)$ are available from $\textsc{Rdist}_1$ and $\textsc{Rdist}_2$ respectively, and $d(p_i, v_1)$, $d(p_i, v_2)$ are available from the points group on the edge $(v_1, v_2)$. Similarly, $d(p_j, v_r)$ can be easily computed. This process can be easily generalized to find $d^-(p_i, v_j)$ or $d^-(v_i, v_j)$.

Now, given any path $c = \{s, p_{x_1}, \dots, p_{x_\ell}, e\}$, a lower bound distance $d^-(c)$ for $d(c)$ can be easily computed as the summation of the lower bound distance of each segment (which is the distance between two points), i.e.,

$$d^-(c) = d^-(s, p_{x_1}) + d^-(p_{x_1}, p_{x_2}) + \cdots + d^-(p_{x_\ell}, e). \quad (1)$$

## 4. EXACT METHODS

We propose two progressive *p*ath *e*xpansion and *r*efinement (PER) algorithms which build up partial candidate paths progressively and refine them until the complete, exact shortest path is guaranteed. Both algorithms follow similar intuitions: we start with the shortest path between $s$ and $e$ and progressively add point $p$ (one at a time) to an existing (partial candidate) path $c$ whose string $w(p)$ is similar to a query keyword $(\sigma_i, \tau_i)$ from $\psi$ (that has not been covered in $c$), while keeping in mind to minimize the impact to $d(c)$ by including $p$ (to decide which point to include). Note that a partial candidate path $c$ is any path that connects $s$ and $e$ and has passed through some points with strings similar to a subset of query keywords from $\psi$. Clearly, in this process a lot of partial candidate paths will be generated and the challenge is to decide what partial candidate paths we generate and the order in which we generate and expand them.

To reduce the cost of shortest path distance calculation for any path, we try to cut down the number of times of exact shortest path distance calculation during the query processing. Hence, both algorithms use the lower bound distance $d^-(c)$ to estimate $d(c)$ for any path $c$ and only attempt to find $d(c)$ exactly after $c$ becomes a complete candidate path.

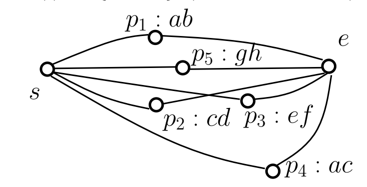
**Pre-processing.** We rely on the landmarks $V_R$ to estimate the lower bound distance of any two objects (and a path). It also helps us in computing the shortest path between two points. How $V_R$ is selected greatly affects its effectiveness, thus the efficiency of our solutions. We adopt the best selection strategy proposed in [8]. Essentially, landmarks should be picked up on the boundary of the road network and as far away from each other as possible. Once we have the landmarks, we compute the distance between all network nodes and all landmarks (the $\textsc{Rdist}_i$'s) by using the Dijstra's algorithm $|V_R|$ times. Each time, we start from one landmark and expand until we meet all the network vertices. It is also shown in [8] that a small constant number of landmarks (e.g., 16) will be enough to achieve excellent performance even for large road networks. This only introduces a small space overhead, which is acceptable in most applications.

For exact methods, we build one FilterTree $T$ for all points of interest $P$ on the road network.

### 4.1 The PER-full algorithm

For a query keyword set $\psi$, we use $P(\psi)$ to denote the set $\{P(\sigma_1, \tau_1), \dots, P(\sigma_\kappa, \tau_\kappa)\}$.

We first present some utility methods that help design the PER-full algorithm. The first method is *NextMinLen*, which takes as input the path $c = \{s, \dots, p_{x_j}, e\}$ and $P(\sigma_i, \tau_i)$ (assume $p_{x_j}$ covers the query keyword $(\sigma_i, \tau_i) \in \psi$). Clearly, $p_{x_j} \in P(\sigma_i, \tau_i)$. Suppose we sort any point $p \in P(\sigma_i, \tau_i)$ in the ascending order of $d(p_{x_{j-1}}, p) + d(p, e)$ and obtain a list $L(P(\sigma_i, \tau_i))$. NextMinLen simply returns the successor of $p_{x_j}$ from this list. For example, given $c = \{s, p_1, e\}$ and $\psi = \{(ab, 1), (cd, 1), (ef, 1)\}$ in Figure 4, clearly $p_1$ covers $(ab, 1)$. NextMinSum will find $p_4$ as the answer in this case, since $L(P(ab, 1)) = \{p_1, p_4\}$ (an ordered set).



$q : s, e, \psi = \{(ab, 1), (cd, 1), (ef, 1)\}$

**Figure 4: Example for MinLen and NextMinLen.**

The next method takes as input a path $c$ and $P(\psi - \psi(c))$, and returns one point $p$ for every keyword $(\sigma_i, \tau_i) \in \psi - \psi(c)$ (i.e., keywords that are not yet covered by $c$), subject to the following constraint:

$$p = \underset{p \in P(\sigma_i, \tau_i)}{\operatorname{argmin}} \; d(p_{x_j}, p) + d(p, e), \quad (2)$$

where $p_{x_j}$ is the last point in $c$ before $e$. Intuitively, given $c = \{s, \dots, p_{x_j}, e\}$, $p$ minimizes the distance for the path $c' = \{s, \dots, p_{x_j}, p, e\}$, among all points whose strings are similar to an uncovered keyword $\sigma_i$. We will find $|\psi - \psi(c)|$ number of such points (one per uncovered keyword by $c$) and we denote these points as a set $\overline{P}(c)$. Note that $|\overline{P}(c)|$ could be smaller than $|\psi - \psi(c)|$, as one point could be the answer for (2) for more than one uncovered keywords. Nevertheless, $\overline{P}(c) \subseteq P - P(c)$. We denote this method as *MinLen*.

For example, suppose $\psi = \{(ab, 1), (cd, 1), (ef, 1)\}$ and $c = \{s, e\}$ in Figure 4. Then, $\psi(c) = \emptyset$, $P(c) = \{s, e, p_5\}$. *MinLen* will return $\overline{P}(c) = \{p_1, p_2, p_3\}$, where $p_1$ matches $(ab, 1)$, $p_2$ matches $(cd, 1)$, and $p_3$ matches $(ef, 1)$. Consider the uncovered keyword $(ab, 1)$ , $P(ab, 1) = \{p_1, p_4\}$, but among the two paths $c_1 = \{s, p_1, e\}$ and $c_2 = \{s, p_4, e\}$ $c_1$ is shorter than $c_2$, hence $p_1$ is favored over $p_4$.

In practice, realizing both *NextMinLen* and *MinLen* exactly is expensive, which involves exact shortest path com-

```
Algorithm PER-full $(s, e, \psi)$
1.  MinHeap $H = \emptyset$, entries in $H$ are (path, score) pairs and
    sorted by the ascending order of their scores;
2.  $c = \{s, e\}$; add the entry $(c, d^-(c))$ into $H$;
3.  set $d^+ = +\infty$;
4.  while (true)
5.      remove the top entry $t = (c, t(c))$ ($t(c)$ is $c$'s score)
        from $H$; suppose $c = \{s, p_{x_1}, \cdots, p_{x_j}, e\}$;
6.      if ($t$'s complete flag is set)
7.          if ($t$'s exact flag is set) return $c$;
8.          else
9.              RefinePath($c$);
10.             if ($d(c) < d^+$)
11.                 $d^+ = d(c)$; add $(c, d(c))$ into H and set
                    its exact flag;
12.     else
13.         $\overline{p} = MinLen(c, P(\psi - \psi(c))$;
14.         for each $p \in \overline{p}$
15.             let $c' = \{s, p_{x_1}, \cdots, p_{x_j}, p, e\}$;
16.             if ($d^-(c') < d^+$)
17.                 add $(c', d^-(c'))$ into $H$; set its complete
                    flag if $\psi(c') = \psi$;
18.     suppose $p_{x_j}$ covers $(\sigma_i, \tau_i)$ from $\psi$, let
        $p =$ NextMinLen($c$, $P(\sigma_i, \tau_i)$);
19.     let $c' = \{s, p_{x_1}, \cdots, p_{x_{j-1}}, p, e\}$;
20.     if ($d^-(c') < d^+$)
21.         add $(c', d^-(c'))$ to $H$, it inherits $t$'s complete flag;
```

**Figure 5: The PER-full Algorithm**

putation, especially when these methods are executed frequently. To improve efficiency, we use the lower bound distance between two points to approximate $d(p_{x_{j-1}}, p)+d(p, e)$ in *NextMinLen* and (2) in *MinLen*, respectively. Furthermore, to reduce space consumption, we do not materialize the entire list of $L(P(\sigma_i, \tau_i))$ at once in *NextMinLen*; rather, we incrementally materialize $k$ elements of this list each time when needed, for some small constant $k$.

The last utility method is *RefinePath*, which takes $c$ as input and computes its exact distance $d(c)$. Basically, it uses the ALT algorithm to compute the shortest path and the corresponding distance for each segment on $c$. The output is simply the summation of these distances.

That said, the PER-full algorithm is presented in Figure 5. The basic intuition is to progressively build partial candidate path and each time a partial candidate path $c$ is expanded to include one more uncovered query keyword from $\psi - \psi(c)$. This process stops when a complete candidate path is identified and we are sure that it has the shortest length among all possible partial and complete candidate paths. The challenge lies in how to design an efficient and effective terminating condition.

To do so, we maintain a min-heap $H$ to store the partial and complete candidate paths, and their scores, found so far. The score of any path $c$, $t(c)$, is defined as $d^-(c)$ when $d(c)$ is not available for $c$; otherwise $d(c)$ is the score. Entries in $H$ are sorted in the ascending order by their scores. We initialize $H$ with $(c = \{s, e\}, d^-(s, e))$. The PER-full algorithm keeps popping out the top entry from $H$, which is a path $c$ with the minimum score. If $c$ is a complete candidate path in $\mathcal{A}_c$ and its score is its exact distance $d(c)$, the algorithm terminates and outputs $c$ as the final answer (line 7). Otherwise, if $c \in \mathcal{A}_c$ but its score is not $d(c)$, we use *RefinePath* to get its exact distance $d(c)$, and insert $(c, d(c))$ back to $H$. We set the *exact bit-flag* of this entry, so that checking if the score of an entry is a path's exact distance in line 7 is easy.

We also maintain a global variable $d^+$ to track the minimum exact distance among complete paths with exact distances identified so far in $H$ (lines 9-11). This helps prune partial and complete paths whose scores (either $d^-(c)$ or $d(c)$) are larger than $d^+$ (lines 10, 16, 20), and it is not necessary to expand or examine them any more, which reduces the number of entries to insert into $H$.

If a partial path $c = \{s, p_{x_1}, \cdots, p_{x_j}, e\}$ is popped out from $H$, we first consider the possibility of expanding $c$ to get a new path that covers one more query keyword that is not yet covered by $c$ (lines 13-17). Recall that the set of query keywords already covered by $c$ is denoted by $\psi(c)$. So the set of uncovered keywords by $c$ is $\psi - \psi(c)$. For each uncovered query keyword $(\sigma_i, \tau_i) \in \psi - \psi(c)$, there are multiple points $P(\sigma_i, \tau_i)$ matching $\sigma_i$. We can choose any one point from $P(\sigma_i, \tau_i)$ and append it to $p_{x_j}$ in $c$ to get a new path $c'$ which now covers $\psi(c') = \{\psi(c), (\sigma_i, \tau_i)\}$. To make a good guess, we should first consider the point $p \in P(\sigma_i, \tau_i)$ that potentially leads to a new path $c'$ with the shortest distance among all possibilities, which is precisely the constraint in (2) in formulating the *MinLen* method. Hence, *MinLen* precisely returns these points $\overline{p}$ (line 13), one for each uncovered query keyword from $\psi - \psi(c)$ (possibly with duplicate points, but only unique points are included in $\overline{p}$); PER-full just expands $c$ with each point in $\overline{p}$ by adding it in between the last point $p_{x_j}$ and $e$ in $c$ (lines 14-15). It also sets its *complete* bit-flag when the new path to insert into $H$ now covers all query keywords (line 17).

But this is not enough. Given either a partial path or a complete path without its exact flag set, $c = \{s, p_{x_1}, \cdots, p_{x_j}, e\}$, suppose that its last point $p_{x_j}$ covers $(\sigma_i, \tau_i) \in \psi$. With only the above expansions, when expanding the path $c' = \{s, p_{x_1}, \cdots, p_{x_{j-1}}, e\}$ into $c$ to cover $(\sigma_i, \tau_i)$, all points in $P(\sigma_i, \tau_i)$ are possible candidates, but only $p_{x_j}$ is used to expand $c'$. Note that $p_{x_j}$ is the top/first point from the list $L(P(\sigma_i, \tau_i)$ (which is discussed and defined in the *NextMinLen* method at the beginning of this section). Clearly, to ensure correctness, we must explore the rest of possibilities as well. The PER-full algorithm does this progressively. After expanding $c'$ with $p_{x_j}$ to get $c$ and insert $c$ into $H$, when $c$ is eventually popped out, not only we expand $c$ just as what we did for $c'$, but also we consider expanding $c'$ with the next point from the list $L(P(\sigma_i, \tau_i))$. This means that we need to replace the last point $p_{x_j}$ in $c$ with the next point from $L(P(\sigma_i, \tau_i))$ (line 19), which is precisely the point returned by *NextMinLen* in line 18.

**Theorem 1** *The PER-full algorithm finds the exact answer for any MAKR query.*

PROOF. Our main task is to ensure that all possible partial paths, whose distances are smaller than the first complete path with an exact distance that shows up at the top of $H$, should have been explored by PER-full (either they are pruned away or inserted into $H$). Clearly, when this holds, the terminating condition in line 7 correctly finds the exact answer for an MAKR query and guarantees the correctness of the PER-full algorithm.

This is exactly what is achieved by the way PER-full algorithm expands or modifies a path when it is popped out from $H$ (lines 13-21). There are only two possibilities: 1) expand a partial path to cover a new query keyword and do this for all uncovered query keywords (lines 13-17); 2) for the query keyword covered by the last expanded point in a path, consider another matching point to cover the same query keyword and ensure all such matching points have a chance to be considered in some order (lines 18-21).

Finally, as we discussed, for efficiency purpose, in practice we use the lower bound distances in realizing *MinLen* and *NextMinLen* methods. But clearly, this does not affect the correctness of the PER-full algorithm. Since we used lower bound distances, this still ensures that when a complete candidate path with exact distance is popped out from $H$, it will be the shortest path from $\mathcal{A}_c$. □

Lastly, $d^-(c')$ for a new path $c'$ obtained from a path $c$ in lines 16, 20 can be easily computed incrementally based on $d^-(c)$ and lower bound distances of affected segments.

## 4.2 The PER-partial algorithm

In the PER-full algorithm, when a complete candidate path $c$ is popped out from $H$ without exact distance, we find $d(c)$ and insert it back to $H$. Finding $d(c)$ could be very expensive and this effort will be a waste if $c$ turns out not being the final exact answer. An alternative approach is to *refine* $d^-(c)$ by finding the exact distance of one segment of $c$ each time when $c$ is popped out, and insert $c$ and refined, tighter $d^-(c)$ back to $H$. The rest of the algorithm is the same, with minor changes as when we set the *exact* bit-flag and keeping additional bit-flags with each complete path $c$ to indicate which segment $c$ has its exact distance calculated already. That said, we can modify the *RefinePath* method to refine only one segment and return the summation of exact and lower bound distances of various segments. We denote this approach as the PER-partial algorithm.

## 5. APPROXIMATE METHODS

We first present a negative result.

**Theorem 2** *The MAKR problem is NP-hard.*

PROOF. The classical traveling salesman problem (TSP) can be reduced to the MAKR problem. Given a TSP instance, it requires starting at a point $o$, traveling through a set of points $\{p_1, \dots, p_\kappa\}$ on a connected graph $G$, and returning to $o$ in the end. Finding the shortest trip that satisfies this constraint is NP-hard [5]. We can construct an MAKR instance that the above TSP instance can reduce into. Let $s = e = o$ and set $\psi = \{(\sigma_1, 0), \dots, (\sigma_\kappa, 0)\}$ for any $\kappa$ string constants. In the graph $G$, we associate $\sigma_i$ with $p_i$ for $i \in [1, \kappa]$ and make sure any other points in $G$ having strings different from any $\sigma_i$. Clearly, this MAKR query solves the above TSP instance. Thus, the MAKR problem is NP-hard. □

That said, our exact methods are still efficient heuristics that work well when $\kappa$ and/or $P(\psi)$ is not too large (as seen in our experiment). However, when they keep increasing, we need to design approximate methods that have good approximation quality and scalability in practice.

## 5.1 The local minimum path algorithms

The first class of approximate methods is greedy algorithms that greedily attempt to add a point $p$, to a partial candidate path $c$, that is similar to one of the uncovered query keywords and minimizes the distance (after being updated) of the segment affected when adding $p$ to it. We denote this class of algorithms as the *local minimum path* algorithm, or $A_{\text{LMP}}$, since they target at minimizing the local change to one segment when deciding which qualified point to include into a partial candidate path.

The first algorithm ($A_{\text{LMP1}}$) works as follows. It starts with the path $\{s, e\}$. In a given step, assume we have a partial candidate path $c = \{s, p_{x_1}, \dots, p_{x_j}, e\}$ with $(j+1)$ segments. $A_{\text{LMP1}}$ iterates through the segments in round robin fashion. For the $i$th segment ($i \in [0, j]$), $\{p_{x_i}, p_{x_i+1}\}$ (let $p_{x_0} = s$ and $p_{x_{j+1}} = e$), we find a point $p$ subject to:

$$p = \operatorname*{argmin}_{p \in P(\psi - \psi(c))} d(p_{x_i}, p) + d(p, p_{x_{i+1}}), \tag{3}$$

and we update $c$ by adding $p$ to the $i$th segment, i.e., $c$ becomes $c = \{s, p_{x_1}, \dots, p_{x_i}, p, p_{x_{i+1}}, \dots, p_{x_j}, e\}$. $A_{\text{LMP1}}$ terminates whenever the current path $c$ turns into a complete candidate path and outputs $c$ as the answer.

Realizing (3) is easy. We maintain a priority-queue sorted in the ascending order of an entry's score where each entry is a point $p \in P(\psi - \psi(c))$; let $c_l = \{p_{x_i}, p, p_{x_{i+1}}\}$, an entry's score is initialized as $d^-(c_l)$. Whenever an entry is popped out from the queue, we update its score to $d(c_l)$ and insert it back to the queue. The answer to (3) is identified as soon as an entry with its score being $d(c_l)$ is popped.

The second variant ($A_{\text{LMP2}}$) processes the query keywords in $\psi$ one by one. It also starts with the path $\{s, e\}$. The $y$th step processes $(\sigma_y, \tau_y)$. Suppose the current partial candidate path is $c = \{s, p_{x_1}, \dots, p_{x_j}, e\}$, $A_{\text{LMP2}}$ finds a point $p$ to include into the $i$th segment of $c$, subject to:

$$(p, i) = \operatorname*{argmin}_{p \in P(\sigma_y, \tau_y), \text{ and } i \in [0,j]} d(p_{x_i}, p) + d(p, p_{x_{i+1}}), \tag{4}$$

i.e., it finds a point $p$ from $P(\sigma_y, \tau_y)$ to cover $\sigma_y$ that minimizes the change in distance to the current path $c$ after including $p$ into one of the segments on $c$ (by considering all segments for possible inclusion of $p$). Then, it updates $c$ to $c = \{s, p_{x_1}, \dots, p_{x_i}, p, p_{x_{i+1}}, \dots, p_{x_j}, e\}$. $A_{\text{LMP2}}$ terminates when $c$ becomes a complete candidate path and regards $c$ as the final answer. Note that (4) can be implemented in a similar way as we did for (3).

## 5.2 The global minimum path algorithm

The next approximate algorithm generalizes the intuitions behind the greedy algorithms from the last section using a more global view. Doing so allows us to prove its approximation bound in the worst case. The global minimum path algorithm, or $A_{\text{GMP}}$, works as follows. For each $(\sigma_i, \tau_i) \in \psi$ for $i \in [1, \kappa]$, it finds a point $p$ subject to:

$$p = \operatorname*{argmin}_{p \in P(\sigma_i, \tau_i)} d(s, p) + d(p, e). \tag{5}$$

Note that (5) again can be easily realized in a similar way as (3) and (4). We denote these points as $P'$ (after removing duplicates). Note that $|P'|$ may be less than $\kappa$ as the same point might be returned by (5) for different query keywords. Suppose $\text{nn}(q, P)$ finds the nearest neighbor of $q$ from a set of points $P$ (based on the network distance). $A_{\text{GMP}}$ first finds $p = \text{nn}(s, P')$, constructs a path $\{s, p, e\}$ and removes $p$ from $P'$. Next, in each step, given the current partial candidate path $c = \{s, p_{x_1}, \dots, p_{x_j}, e\}$, $A_{\text{GMP}}$ finds $p = \text{nn}(p_{x_j}, P')$, updates $c$ to $c = \{s, p_{x_1}, \dots, p_{x_j}, p, e\}$, and removes $p$ from $P'$. It terminates when $P'$ is empty.

**Theorem 3** *The $A_{\text{GMP}}$ algorithm gives a $\kappa$-approximate path. This bound is tight.*

PROOF. In fact, we can show that once $P'$ is identified as in $A_{\text{GMP}}$, any order of visiting them starting from $s$ and ending at $e$ gives a $\kappa$-approximate path (the nearest neighbor strategy adopted in $A_{\text{GMP}}$ is just a simple heuristic). Given a query $\psi = \{(\sigma_1, \tau_1), \ldots, (\sigma_\kappa, \tau_\kappa)\}$ on $G$ and $P$, suppose the optimal path for this MAKR query is $c^*$.

Without loss of generality, we also assume that in our proof, in any path each point covers only one query keyword. Note that this does not affect the correctness of our following analysis, as we can easily generalize a path to satisfy this condition as follows. If a point $p$ in a path covers $b$ number of query keywords, we can consider there are $b$ distinct points at $p$'s location, one for each query keyword that $p$ is responsible for.

That said, we can assume that $|P'| = \kappa$ and let $c = \{s, p_{x_1}, \cdots, p_{x_\kappa}, e\}$ denote a path formed by a random permutation of the points in $P'$. Clearly, $d(c^*) \leq d(c)$, and:

$$\begin{aligned} d(c) &= d(s,p_{x_1}) + d(p_{x_1},p_{x_2}) + \cdots + d(p_{x_{\kappa-1}},p_{x_\kappa}) + d(p_{x_\kappa},e) \\ &\leq d(s,p_{x_1}) + [d(p_{x_1},e) + d(e,p_{x_2})] \\ &\qquad + [d(p_{x_2},s) + d(s,p_{x_3})] + \cdots \\ &\qquad + [d(p_{x_{\kappa-2}},e) + d(e,p_{x_{\kappa-1}})] \\ &\qquad + [d(p_{x_{\kappa-1}},s) + d(s,p_{x_\kappa})] + d(p_{x_\kappa},e) \\ &= [d(s,p_{x_1}) + d(p_{x_1},e)] + [d(e,p_{x_2}) + d(p_{x_2},s)] + \cdots \\ &\quad + [d(e,p_{x_{\kappa-1}}) + d(p_{x_{\kappa-1}},s)] + [d(s,p_{x_\kappa}) + d(p_{x_\kappa},e)] \\ &= \sum_{i=1}^{\kappa} [d(s,p_{x_i}) + d(p_{x_i},e)]. \end{aligned} \tag{6}$$

Suppose the optimal path is $c^* = \{s, p^*_{x_1}, \cdots, p^*_{x_\kappa}, e\}$. Since any $p_{x_i}$ from $c$ is from $P'$, and by the construction in (5), we have:

$$d(s,p_{x_i}) + d(p_{x_i},e) \leq d(s,p^*_{x_i}) + d(p^*_{x_i},e). \tag{7}$$

Also, it is trivial to see that:

$$d(s,p^*_{x_i}) + d(p^*_{x_i},e) \leq d(c^*) \text{ for any } i \in [1,\kappa]. \tag{8}$$

Now, we can easily show that:

$$\begin{aligned} d(c) &\leq \sum_{i=1}^{\kappa} [d(s,p_{x_i}) + d(p_{x_i},e)] && \text{by (6)} \\ &\leq \sum_{i=1}^{\kappa} [d(s,p^*_{x_i}) + d(p^*_{x_i},e)] && \text{by (7)} \\ &\leq \kappa d(c^*) && \text{by (8)} \end{aligned} \tag{9}$$

We provide an instance to show that this bound is also tight. We construct a road network $G$ as in Figure 6, where the solid lines are the only edges in $G$. We place $\kappa$ points $\{p_1, \ldots, p_\kappa\}$, randomly on the edge of a circle centered at a point $s$ with a fixed radius $\beta$. We then place another $\kappa$ points $\{p^*_1, \ldots, p^*_\kappa\}$ at one location that is $(\beta + \varepsilon)$ away from $s$ for some small value $\varepsilon$. Now we find $\kappa$ distinct strings $\{w_1, \ldots, w_\kappa\}$ and assign $w_i$ to both $p_i$ and $p^*_i$. Let $e = s$, and for the query $\psi = \{(w_1, 0), \ldots, (w_\kappa, 0)\}$, clearly, the optimal path $c^* = \{s, p^*_1, \ldots, p^*_\kappa, e\}$ and $d(c^*) = 2(\beta+\varepsilon)$. However, using (5) $P' = \{p_1, \ldots, p_\kappa\}$. In this case, the nearest neighbor of $s$ can be any point in $P'$ and we simply take it as $p_1$. Clearly, the $A_{\text{GMP}}$ algorithm will form and return a path $c = \{s, p_1, p_2, \ldots, p_\kappa, e\}$. Note that for any segment $\{p_i, p_{i+1}\}$ in $c$, the shortest path for it in $G$ is $\{p_i, s, p_{i+1}\}$. Hence, $d(c) = 2\kappa\beta$. Observing that $\varepsilon$ can be arbitrarily small, this completes the proof. □

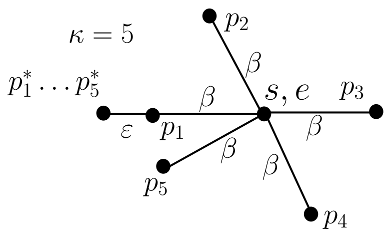


**Figure 6: Tightness of $A_{\text{GMP}}$.**

Theorem 3 gives the tight worst-case bound for $A_{\text{GMP}}$, but the worst case scenario rarely happens in practice, where $A_{\text{GMP}}$ achieves much better approximation quality than what the worse-case bound has suggested.

## 5.3 Improvement by network partitioning

To further improve the performance of our approximate methods, we partition the road network into disjoint partitions and utilize the lower bound distances from object(s) of interest to partitions to prune. In a nutshell, in the preprocessing step, we randomly select $m$ points $P_s$ from $P$ as *seeds* and construct the voronoi-diagram-like partition of the network using these seeds. We denote this method as the *RP* algorithm. Without loss of generality, assume $P_s = \{p_1, \ldots, p_m\}$. *RP* first initializes $m$ empty subgraphs $\{G_1, \ldots, G_m\}$, and assigns $p_i$ as the "center" of $G_i$. Next, for each node $v \in V$, *RP* finds $p = \text{nn}(v, P_s)$, and computes $d(v, p)$. This can be done efficiently using Erwig and Hagen's algorithm [6], with $G$ and $P_s$ as the input. Next, for each edge $e \in E$ with $e = (v_i, v_j)$, *RP* inserts $e$ into $G_i$ whose center $p_i$ minimizes $\min\{d(p, v_i), d(p, v_j)\}$ among all subgraphs. When all edges are processed, *RP* returns the $m$ edge-disjoint subgraphs constructed. For each subgraph $G_i$, let the points on $G_i$ be $P_i$; we build a FilterTree $T_i$ for $P_i$.

We use $A_{\text{GMP}}$ to illustrate how these partitions help improve its efficiency. First, we define the lower bound distance between a point $p$ and a subgraph $G_i$, denoted as $d^-(p, G_i)$, as the minimum lower bound distance from $s$ to any boundary nodes of $G_i$. Clearly, $d^-(p, G_i)$ also provides a lower bound distance for distance between $p$ and any point in $P_i$ (points in $G_i$). Note that the most expensive part of $A_{\text{GMP}}$ is to compute (5) for each query keyword. We next show how to do this for query keyword $(\sigma_i, \tau_i)$.

Using the partitions, we can compute $d^-(s, G_i) + d^-(G_i, e)$ for all subgraphs $G_i$'s. We can easily show that:

$$d^-(s, G_i) + d^-(G_i, e) \leq d^-(s, p) + d^-(p, e) \text{ for any } p \in P_i.$$

Let $d^-(s, G_i) + d^-(G_i, e)$ be the score of $G_i$, we initialize a priority queue with entries of (subgraph, score) pairs for each subgraph. Whenever a subgraph $G_i$ is popped out from the queue, we retrieve $P_i(\sigma_i, \tau_i)$ using the FilterTree $T_i$. For each $p \in P_i(\sigma_i, \tau_i)$, we insert an entry $(p, d^-(s, p) + d^-(p, e))$ into the queue. Whenever a point $p$ with the lower bound distance $d^-(s, p) + d^-(p, e)$ is popped out, we find $d(s, p) + d(p, e)$ and insert $(p, d(s, p) + d(p, e))$ into the queue. The algorithm terminates when a point with exact distance is popped out. The previous approach initializes the queue with $P(\sigma_i, \tau_i)$, resulting a much larger queue. Using the subgraphs, we may prune some subgraphs completely without the need to explore $P_i$.

Similar methods can be developed using the subgraphs for improving the efficiency of $A_{\text{LMP1}}$ and $A_{\text{LMP2}}$ algorithms. We omit the details. This idea can also help improve the efficiency of *MinLen* and *NextMinLen* methods in the exact algorithms, but the bottleneck there is the number of partial complete paths examined. So the improvement in efficiency for the exact methods is not significant.

## 6. EXTENSIONS

**Top-$k$ MAKR query.** The exact methods can easily answer the top-$k$ MAKR queries after making two changes. First, the algorithms terminate only when $k$ complete candidate paths with exact distances have been popped out from the heap. Second, $d^+$ in Figure 5 should store the $k$th minimum exact distance found so far. The rest of the algorithms remains the same. All approximate methods can be modified by taking the top-$k$ points satisfying the constraint in (3), (4), and (5) respectively, and constructing $k$ candidate paths accordingly. Using a similar analysis, we can show that the same approximation bound in Theorem 3 still holds for the modified $A_{\text{GMP}}$ algorithm w.r.t. top-$k$ MAKR queries.

**Multiple strings.** In the general case, each point in $P$ may contain multiple keywords. This is easy to handle for all our algorithms. Suppose a point $p \in P$ contains $b$ keywords, we simply consider $p$ as $b$ distinct points, each sharing the same location as $p$ but having one distinct keyword from the $b$ keywords that $p$ originally has.

**Updates.** Our data structures support efficient updates, by following the update algorithms for B+ trees and FilterTrees respectively, with no changes to query algorithms.

## 7. EXPERIMENTAL EVALUATION

We implemented all methods in C++. For the FilterTree, we use the FLAMINGO library [1]. All experiments were performed on a Linux machine with an Intel Xeon 2.13GHz CPU and 6GB memory.

**Datasets.** We used three real road networks, City of Oldenburg (OL), California (CA), and North America (NA), from the Digital Chart of the World Server. OL contains 6105 vertices and 7029 edges, CA contains $21,048$ vertices and $21,693$ edges, and NA contains $175,813$ vertices and $179,179$ edges. For CA and NA networks, we obtained the points of interest associated with strings, from the OpenStreetMap project from California and North America respectively. In each network, we then randomly select 2 million points to create the largest dataset $P_{\max}$ and form smaller datasets $P$ based on $P_{\max}$. For each $P$, we assign points into the road network w.r.t. their normalized coordinates and edges' normalized coordinates. For the OL network, we simply reuse the dataset $P$ from either CA or NA and assign them into OL using the normalized coordinates. For all datasets the average length of the strings is approximately 14.

**Setup.** When we vary $P$ on a network $G$ to test different algorithms, we make sure smaller datasets are always subsets of larger datasets, in order to isolate the impact of $|P|$. To make sure that $\mathcal{A}_c \neq \emptyset$ for any query, we randomly select $\psi$ from strings in the dataset $P$ to be used for this query. For simplicity, we use the same edit distance threshold for all keywords in any query, and denote it as $\tau$. The default values of some parameters are as follows: $\kappa = 6$, $\tau = 2$, $|V_R| = 8$ and $m = 10$ (recall $m$ is the number of subgraphs, as discussed in Section 5.3, in the approximation methods). The default dataset is CA. Suppose an approximate method

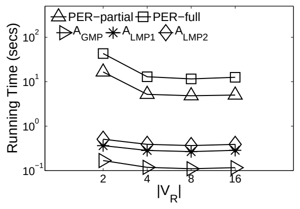


Figure 7: Effect of $|V_R|$

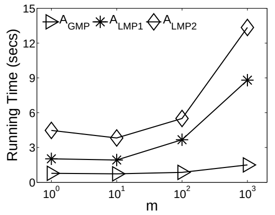


Figure 8: Effect of $m$

$A$ returns a path $c$ for an MAKR query instance with the optimal path $c^*$, $A$'s approximation ratio in this instance is $r = d(c)/d(c^*)$. In all experiments we report the averages over 50 randomly generated queries.

### 7.1 Impacts of Landmarks and Partitions

The impact of the landmarks comes in two folds: the selection strategy and the number of landmarks. Our experiments and observations confirmed the conclusion and results in [8] that the *optimized planar* algorithm in [8] is always the best strategy (essentially, select nodes on the boundary of the road network and as far away from each other as possible), given the same $|V_R|$. Clearly, with more landmarks, the lower bounds tend to be tighter which leads to better pruning. However, using more landmarks also leads to higher computation costs in calculating the lower bounds. Therefore, we expect to see a sweet point of the overall running time w.r.t. different $|V_R|$ values. Figure 7 reflects exactly this trend, where $|P| = 10,000$ points were used in the CA network. The same trend holds for all algorithms on different datasets and on OL and NA as well. They consistently show that $|V_R| = 8$ tends to provide the best running time in all cases. Thus, we use $|V_R| = 8$ across the board.

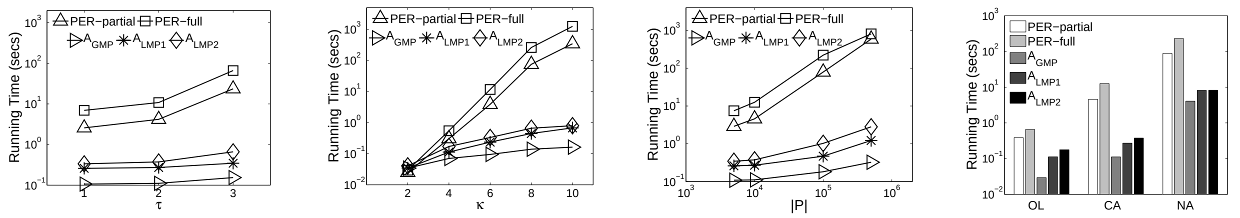
Next, we study the effect of $m$, number of subgraphs, on approximation algorithms (as explained in Section 5.3, partitioning has no significant effect on the overall running time of exact methods). Using 1 million points on CA, Figure 8 shows that when $m \leq 10$, the running time reduces with more subgraphs, as more subgraphs lead to better pruning (which avoid searching FilterTrees). However, having more and more subgraphs also means more access to smaller FilterTrees, which introduces query overhead compared to searching over fewer, larger FilterTrees. Eventually, such overheads dominate over the benefit of the pruning by more and smaller subgraphs. Hence, the query costs start to increase after $m > 10$ in Figure 8. Other datasets and other networks show similar results and $m = 10$ tends to achieve a good balance, which will be used by default.

Lastly, selecting $V_R$, computing RDIST$_i$'s, building FilterTrees, and running *RP* (the partitioning algorithm) are all very efficient (in terms of both time and space). We omit these results due to the space constraint.

### 7.2 Query Performance

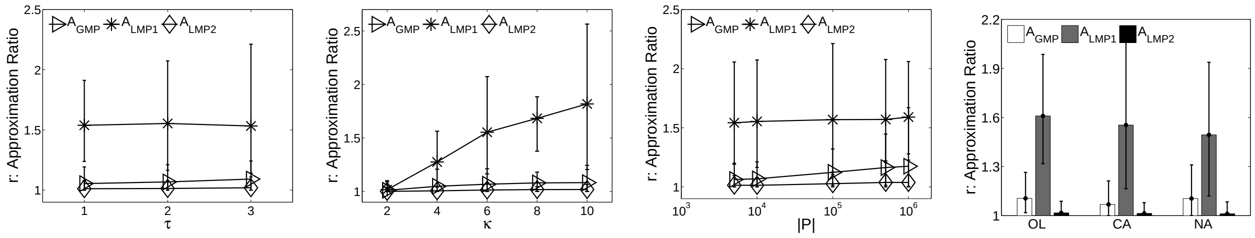
We first investigate the performance of all methods using relatively smaller datasets, to ensure that exact methods can complete in reasonable amount of time. In the default case, $|P| = 10,000$. Figure 9 shows the running time in *log scale*. Clearly, in all cases, PER-partial improves the performance of PER-full, and all approximate methods outperform the exact methods by orders of magnitude.

Figure 9(a) shows the effect of different $\tau$ values. It indicates the running time for all methods increases with larger $\tau$ values, simply due to more points becoming candidates



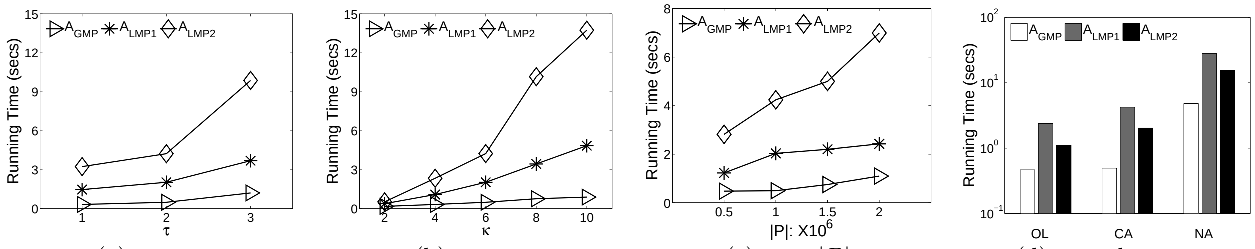
(a) vary $\tau$ (b) vary $\kappa$ (c) vary $|P|$ (d) vary datasets

**Figure 9: Query time**



(a) vary $\tau$ (b) vary $\kappa$ (c) vary $|P|$ (d) vary datasets

**Figure 10: Approximation quality**



(a) vary $\tau$ (b) vary $\kappa$ (c) vary $|P|$ (d) vary datasets

**Figure 11: Scalability of approximate methods**

to consider. It also reveals that the exact methods are reasonably efficient in this case; when $\tau = 2$, both methods complete in less than 10 seconds. The fastest approximate method $A_{\text{GMP}}$ completes in around 0.1 second in all cases.

Figure 9(b) shows the query time when we vary $\kappa$, the number of the query keywords. Clearly, the query time of exact solutions increases very quickly w.r.t. $\kappa$ as expected (since more combinations are possible to form candidate paths). The PER-partial algorithm does improve the running time compared to the PER-full algorithm in all cases. Nevertheless, when $\kappa$ is not too large, say $\kappa \leq 6$, the running time of our exact methods are reasonably efficient. For example, when $\kappa = 4$, they take less than a second to complete. However, for larger $\kappa$, approximate methods are clear winners. Not only they outperform exact methods by as far as 4 orders of magnitude in $\kappa = 10$, but also they achieve very small running time overall. When $\kappa \leq 4$, all of them take less than 0.1 second. Even when $\kappa = 10$, all approximate methods take less than a second to complete; and the fastest one $A_{\text{GMP}}$ still just takes around 0.1 second!

The next experiment in Figure 9(c) investigates the scalability of all methods w.r.t. $|P|$, when it changes from 5000 to $500,000$. Not surprisingly, approximate methods have achieved much better scalability than exact methods. They outperform exact methods by 3-4 orders of magnitude when $|P|$ is half million, and only take about or less than a second. The fastest method $A_{\text{GMP}}$ just needs about 0.3 second in this case. Nevertheless, by carefully introducing various pruning and heuristics, our exact methods achieve acceptable performance (given that the problem is NP-hard). For example, with $10,000$ points, PER-partial only needs a few seconds; with $100,000$ points, it just uses about 80 seconds; with $500,000$ points, it takes about 900 seconds.

Figure 9(d) shows the running time using different road networks. The trends are clearly consistent across different networks: 1) larger networks are costly; 2) approximate methods are significantly faster; 3) PER-partial ($A_{\text{GMP}}$) is the fastest exact (approximate) method respectively. We would like to point out that even on NA (with hundreds of thousands of nodes and edges), $A_{\text{GMP}}$ only takes a few seconds and PER-partial uses less than 100 seconds.

In terms of approximation quality, Figure 10 shows the approximation ratios of all approximate methods for the same set of experiments as in Figure 9. In this case, we plotted both the average and the $5-95\%$ interval (to illustrate the variance of the approximation quality). In all cases, clearly, $A_{\text{LMP1}}$ has the worst ratio, averaging around 1.6 and exceeding 2 in the worst case. $A_{\text{LMP2}}$ and $A_{\text{GMP}}$ have achieved similar ratios, while $A_{\text{LMP2}}$'s ratio being consistently, slightly better. In fact, both of them achieve average ratios that are close to 1 in all cases! They also enjoy very small variance, with the worst ratio close to 1.3 in all cases. Clearly, $A_{\text{GMP}}$'s approximation quality in practice is significantly better than what its worst case theoretical bound suggests (which is $\kappa$).

We further investigate the scalability of our approximate methods on very large datasets. Specifically, in the following experiments, the default $|P|$ is 1 million points. We omit the exact methods from the results in Figure 11, since they take more than 2000 seconds to complete in this case. These results clearly show that $A_{\text{GMP}}$ is extremely efficient and scalable w.r.t. all factors. For example, Figure 11(b) indicates that $A_{\text{GMP}}$ takes only 1 second for 1 million points with 10 query keywords; Figure 11(c) shows that $A_{\text{GMP}}$ completes in 1.5 seconds for 2 million points with 6 query keywords; and Figure 11(d) reveals that $A_{\text{GMP}}$ still just needs a few seconds for 1 million points, 6 query keywords on NA network which has hundreds of thousands of nodes and edges.

The approximation ratios of all approximate methods in this study are almost identical to what reported in previous set of experiments (as shown in Figure 10), indicating that larger datasets do not degrade their approximation qualities. For brevity we omit these figures.

**Summary.** These results show that our exact method PER-partial is efficient when the number of keywords and the dataset size are small to moderate. To deal with more keywords and very large datasets, approximate methods should be employed, and $A_{\text{GMP}}$ has the fastest query time and almost as good approximation quality as the best approximation achieved (by $A_{\text{LMP2}}$). In particular, $A_{\text{GMP}}$ completes a query, with up to 10 keywords, in less than a second to only a few seconds even with millions of points on a network with hundreds of thousands of nodes and edges; and almost always finds a path that is nearly as good as the optimal path.

## 8. RELATED WORK

Spatial keyword queries in GIS data have been gaining attentions lately. The most relevant work to our study appears in [3], where Cao et al. studied the spatial group keyword query in Euclidean space. Given a query point $q$ and a group of keywords, the goal is to find a group of points from the database that 1) covers all query keywords; 2) minimizes the sum of their distances to $q$. There is also a variant on the second condition to minimize the sum of the maximum distance to $q$ and the maximum inter-distance for the group of points returned. The $m$CK query in Euclidean space was studied in [16, 17], which takes a set of $m$ keywords as input and returns $m$ objects from the database so that they cover these keywords and minimize the maximum distance between any two returned objects. The IR$^2$-tree [7] was proposed to answer range and nearest neighbor queries combined with keyword matching. Then, the IR-tree [4] was designed to find top-$k$ most-relevant (weighted-sum of spatial and keyword distances) spatial objects. All these studies integrated spatial indexes in the Euclidean space (mainly the R-tree) with keyword-processing structures (e.g., inverted lists, bitmaps). They also focused on exact keyword matching.

The optimal sequenced route (OSR) query [13] is also relevant where each point in $P$ is assigned a fixed category. Then, given $s$ and $e$, and an *ordered set* $D$ of categories, an OSR query asks for the shortest path $c$ between $s$ and $e$ that visits at least one point from each category. An additional constraint is that $c$ must visit categories in the *same sequence* as they appear in the query set $D$. One may adapt the solution to OSR queries to our setting: each query keyword in an MAKR query defines a category of points from $P$ (those that are similar to it). Then, we execute $\kappa!$ number of OSR queries, one for each possible permutation of $\psi$, to find the shortest path. This is clearly prohibitive. Similarly, in trip planning queries categories were predefined [11]. It reveals that in the MAKR problem: 1) there are no pre-defined "categories" associated with points in $P$; rather, their "categories" change dynamically w.r.t. different queries; 2) there is no user-defined fixed order/sequence to visit keywords.

Approximate string search has been extensively studied in the literature (please refer to an excellent tutorial [9] and references therein). To the best of our knowledge, integrating approximate string search with spatial queries has only been examined recently in [2, 15]. However, both work focused on integrating approximate string search with range and nearest neighbor queries in the Euclidean space.

## 9. CONCLUSION

This paper studies the Multi-Approximate-Keyword Routing (MAKR) problem that clearly has a lot of applications in practice. We proposed both exact and approximate methods. Our approximate methods are especially appealing. Not only one ($A_{\text{GMP}}$) comes with guaranteed approximation bound (for the worst case), but also all approximate methods demonstrate excellent query performance, scalability, and approximation quality on real large datasets. Experiments also reveal that $A_{\text{GMP}}$ achieves much better approximations in practice than what suggests by its theoretical worse-case bound. Future work include the study of MAKR queries in the Euclidean space, and the continuous monitoring version of the MAKR queries when $s$ and/or $e$ move on $G$.

## 10. ACKNOWLEDGMENT

Bin Yao, Mingwang Tang, and Feifei Li were supported in part by NSF Grant IIS-0916488. Feifei Li was also supported in part by the HP IRP award for the Year 2011 round.

## 11. REFERENCES

[1] Flamingo web site. `http://flamingo.ics.uci.edu/`.
[2] S. Alsubaiee, A. Behm, and C. Li. Supporting location-based approximate-keyword queries. In *GIS*, 2010.
[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, 2011.
[4] 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.
[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 1997.
[6] M. Erwig and F. Hagen. The graph voronoi diagram with applications. *Networks*, 36:156–163, 2000.
[7] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.
[8] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, 2005.
[9] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2):1660–1661, 2009.
[10] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.
[11] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, 2005.
[12] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, 2003.
[13] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDBJ*, 17(4):765–787, 2008.
[14] S. Shekhar and D. ren Liu. Ccam: A connectivity-clustered access method for networks and network computations. *IEEE TKDE*, 9:410–419, 1997.
[15] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, 2010.
[16] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, 2009.
[17] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, 2010.