Geoinformatica (2015) 19:29–60
DOI 10.1007/s10707-014-0204-8

Efficient continuous top-*k* spatial keyword queries on road networks

Long Guo · Jie Shao · Htoo Htet Aung · Kian-Lee Tan

Received: 25 July 2013 / Revised: 15 October 2013
Accepted: 14 January 2014 / Published online: 11 February 2014
© Springer Science+Business Media New York 2014

Abstract With the development of GPS-enabled mobile devices, more and more pieces of information on the web are geotagged. Spatial keyword queries, which consider both spatial locations and textual descriptions to find objects of interest, adapt well to this trend. Therefore, a considerable number of studies have focused on the interesting problem of efficiently processing spatial keyword queries. However, most of them assume Euclidean space or examine a single snapshot query only. This paper investigates a novel problem, namely, continuous top-*k* spatial keyword queries on road networks, for the first time. We propose two methods that can monitor such moving queries in an incremental manner and reduce repetitive traversing of network edges for better performance. Experimental evaluation using large real datasets demonstrates that the proposed methods both outperform baseline methods significantly. Discussion about the parameters affecting the efficiency of the two methods is also presented to reveal their relative advantages.

Keywords Top-*k* spatial keyword queries · Continuous queries · Road networks

1 Introduction

With the rapid development of GPS-enabled mobile devices, there is a prevalent trend that web content is increasingly being geotagged, such as messages on Facebook and Twitter.

L. Guo (✉) · J. Shao · H. H. Aung · K.-L. Tan
School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417, Singapore
e-mail: guolong@comp.nus.edu.sg

J. Shao
e-mail: shaojie@comp.nus.edu.sg

H. H. Aung
e-mail: h2aung@comp.nus.edu.sg

K.-L. Tan
e-mail: tankl@comp.nus.edu.sg

Spatial keyword queries, which find best objects of interest in terms of both spatial proximity to query location and textual relevance to query keywords, adapt well to this trend, and thus, become a new interest of the research community. Most of the existing studies on this topic [4, 7, 8, 17, 20, 21, 23–26] are restricted to Euclidean space only. In many applications, position and accessibility of spatial-textual objects are constrained by network connectivity, and spatial proximity should be determined by shortest path distance (rather than Euclidean distance, which may be inappropriate especially in urban areas). Recently, Rocha-Junior and Nørvåg [18] studied static top-k spatial keyword queries on road networks and proposed algorithms conducted in the spatial and textual domains alternately to constrain the search region. However, there is still no research effort on continuous monitoring of spatial keyword queries on road networks. In location-based services, besides snapshot queries that are evaluated only once, continuous queries are also very appealing, since queries can be evaluated on dynamic settings more realistically. Consider a scenario that Alice and Bob are visiting a foreign city. When Bob intends to find a buffet seafood restaurant for lunch, a spatial keyword query can be submitted to obtain information about some buffet seafood restaurants opening for lunch nearest to them. However, Alice is not satisfied with these results. With continuous spatial keyword queries, they can just keep traveling and up-to-date results will be reported to let them choose from, until an attractive one appears.

In this paper, we study how to efficiently process continuous top-k spatial keyword queries when query and objects of interest are on a road network. Although continuous queries with network distance can provide up-to-date and accurate results as query point moves, it is often costly to monitor such moving queries. A straightforward solution is to traverse the road network to find the top-k results every time the query is evaluated. However, such a scheme will lead to lots of unnecessary repetitive traversing of network edges. Many proposals for moving queries in Euclidean space such as [3, 10, 14, 21] unitize the concept of safe region. In response to a query, its result can be computed together with a safe region within which the result remains valid. Only when the query exits its safe region will a new query (additional processing) be invoked, which repeats the above process. However, query processing on road networks is fundamentally different. Compared with previously proposed continuous *spatial* queries on road networks such as [5, 6, 13], how to exploit the opportunities of simultaneously using spatial proximity and textual relevancy for joint pruning brings new challenges to query optimization of our problem.

We propose two methods that traverse the network in an incremental manner, namely, *query-centric algorithm* (QCA) and *object-centric algorithm* (OCA). In principle, both methods benefit from the reduction of the problem from finding the objects of interest for moving queries to examining static network nodes. QCA starts traversing the network from an end node of the edge on which the query location lies, until it finds the top-k results. Meanwhile, it maintains an expansion tree to avoid unnecessary traversing of some network edges for subsequent processing. OCA takes a different approach and starts the traversing from objects which are relevant to the query keywords. In this way, after the initial processing, an order-k shortest path tree is constructed and subsequent processing can use this tree to significantly reduce the number of edges traversed.

Our contributions are summarized as follows.

– We formalize the problem of continuous top-k spatial keyword queries in the domain of road networks. To the best of our knowledge, this is the first attempt on this important problem with real-world applications.

– We propose two algorithms that incrementally expand the network from the query location and objects relevant to the query keywords respectively. Both of them can reduce lots of unnecessary repetitive traversing of network edges for continuous monitoring.
– We report an extensive set of experiments conducted with real road network datasets to compare our algorithm performance with three baseline methods. In some settings, the cost saving can be as much as one order of magnitude. Results reveal that each of the proposed methods may perform best under different parameter settings.

The rest of this paper is organized as follows. First, we present formal definition of the problem in Section 2. Then, we describe the query-centric algorithm in Section 3, followed by the object-centric algorithm in Section 4. We report the experimental evaluation in Section 5. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Problem statement

Table 1 summarizes the notations frequently used in the paper.

Road network A road network is generally represented by a connected and undirected planar graph $G(N, E)$, where N is the set of nodes and E is the set of edges. In this work, we treat any road segment with multiple edges whose (i) two endpoints are intersection nodes (with degree above 2) or terminal nodes (with degree 1) and (ii) intermediate nodes all have degree 2 as a single edge. For convenience, in the following context when we use *edge*, it indicates that this edge has two endpoints, which are either intersections or terminal nodes.

Object set Let O represent a set of spatial-textual objects on the edges E of G, where each object $o \in O$ has a spatial location $o.l$ and a textual description (or called *document*) $o.d$. Denote $|o, n|$ and $|o, n'|$ as the distances between an object o and two end nodes of the edge (n, n') on which it lies. The shortest path distance between two objects o and o' on G is defined as $d(o, o')$.

Top-k spatial keyword query on a road network Define $q =< q.l, q.d, q.k, q.r >$ to be a top-k spatial keyword query on a road network G, where $q.l$ is the query location, $q.d$ is the set of query keywords, $q.k$ is the number of requested results and $q.r$ is the moving direction of q.[1] Given a set O of spatial-textual objects on G, a top-k spatial keyword query q returns $q.k$ objects from O in ascending order of score τ, which is defined as

$$\tau(q, o) = \alpha \cdot \delta(q.l, o.l) + (1 - \alpha) \cdot [1 - \theta((q.d,\ o.d)], \tag{1}$$

where $\delta(q.l, o.l)$ reflects spatial proximity of $o.l$ from the query location $q.l$, and $\theta(q.d, o.d)$ reflects textual relevance of $o.d$ with respect to the query keywords $q.d$. Like other typical work of spatial keyword queries such as [7], a preference parameter $\alpha \in (0, 1)$ is used to define relative importance of one measure over the other. For example, $\alpha < 0.5$ increases the weight of textual relevance over spatial proximity.

[1] In this paper, we use an undirected graph $G(N, E)$ and transmit the direction of q explicitly. We cannot model the road network as a directed graph and get the direction of q from the directed graph. The reason is that if one road is bidirectional, the direction cannot be obtained.

Table 1 Frequently used notations

Symbol	Description
$G(N, E)$	the graph model of road network
edge(n_i,n_j)	the edge that connects n_i to n_j
$\mathcal{T}$	the expansion tree
O	the set of spatial-textual objects on G
O_r	the set of objects relevant with the query keywords
O_{re}	the set of relevant objects in the expansion tree
$O_{(n_i,n_j)}$	the set of objects lying on the edge(n_i,n_j)
o_i	a spatial-textual object
q	a top-k spatial-keyword query
τ	the score which is used to rank the objects
δ	spatial proximity
θ	textual relevance
R_q	the top-k results of query q
R_{n_i}	the top-k results of node n_i
O_{d1}, O_{d2}	the set of objects in R_q that can be replaced
O_{r1}, O_{r2}	the set of objects that can replace the objects in R_q
$d(p_1, p_2)$	the shortest path distance between two points p_1 and p_2
$\|p_1, p_2\|$	the length of the segment between p_1 and p_2

Spatial proximity (δ) Spatial proximity measure can be defined as

$$\delta(q.l, o.l) = \frac{d(q.l, o.l)}{d_{max}}, \tag{2}$$

where d_{max} is the largest network distance between any object and any location in G. d_{max} can be obtained by traversing the network from each object until the entire network is expanded and keeping the maximum distance. δ is in the range of [0,1].

Textual relevance (θ) Textual relevance measure can be captured by any information retrieval model. In this work, cosine similarity [18, 27] is used to evaluate the similarity between $q.d$ and $o.d$, which is defined as

$$\theta(q.d, o.d) = \frac{\sum_{t \in q.d} \omega_{t,q.d} \cdot \omega_{t,o.d}}{\sqrt{\sum_{t \in q.d} (\omega_{t,q.d})^2 \cdot \sum_{t \in o.d} (\omega_{t,o.d})^2}}, \tag{3}$$

the weight $\omega_{t,q.d} = \ln\left(1 + \frac{|O|}{df_t}\right)$, where $|O|$ is the number of objects in O and df_t is the number of objects with t in their descriptions (document frequency); and the weight $\omega_{t,o.d} = 1 + \ln(f_{t,o.d})$, where $f_{t,o.d}$ is the number of occurrences (frequency) of term t in $o.d$. θ is in the range of [0,1] (property of cosine).

For simplicity of computation, Eq. 3 can be rewritten in the *impact* form as

$$\theta(q.d, o.d) = \sum_{t \in q.d} \lambda_{t,q.d} \cdot \lambda_{t,o.d}, \tag{4}$$

where the impact $\lambda_{t,d} = \frac{\omega_{t,d}}{\sqrt{\sum_{t \in d} (\omega_{t,d})^2}}$ is normalized weight of the term in the document, by taking document length into account [1, 19].

A lower score τ means the object is better. In this paper, we study the efficient processing of *continuous top-k spatial keyword* (CkSK) queries on road networks which is defined as follows.

Definition 1 (**CkSK queries on road networks**) Given a set O of spatial-textual objects and a moving query q on a road network G, *CkSK* queries continuously return k ranked objects in ascending order of score τ.

3 Query-centric algorithm

In this section, we present our first method named *query-centric algorithm* (QCA). We start with the basic idea of our query processing in Section 3.1, followed by the snapshot query algorithm in Section 3.2. We present the usage of expansion tree in Section 3.3 and show how the top-k results can be derived and safe segment can be computed in Section 3.4. Finally, we give complete QCA monitoring algorithm in Section 3.5.

3.1 Basic idea

To solve the CkSK queries, we examine result updates for intersections along the trajectory which the query point moves. Specifically, our method is based on the following lemma:

Lemma 1 *The top-k results R_q of any spatial keyword query q whose query location $q.l$ lies on an edge(n_i, n_j) are in the union of (i) the set of relevant objects $O_{(n_i,n_j)}$ on the edge, and (ii) the top-k results R_{n_i} and R_{n_j} of the end nodes of the edge, which can be formulated as:*

$$R_q \subseteq (O_{(n_i,n_j)} \cup R_{n_i} \cup R_{n_j}),$$

where relevant object refers to object that matches at least one of the query keywords in its description.

Proof We prove this lemma by contradiction. First assume that there exists an object o satisfying $o \in R_q$ and $o \notin (O_{(n_i,n_j)} \cup R_{n_i} \cup R_{n_j})$. Since $o \notin O_{(n_i,n_j)}$, o is not on the edge. The shortest path from the query location $q.l$ to o then must go through either n_i or n_j. Without loss of generality, assume o is closer to n_i. Let o' be any one of the objects in R_{n_i}. Since $o \notin R_{n_i}$, we know that $\tau(n_i, o) > \tau(n_i, o')$. Because $|q.l, n_i|$ can be added to the spatial proximity part of both sides of the above inequality and the textual relevance part remains constant, we have $\tau(q, o) > \tau(q, o')$. This means that all of the top-k results in R_{n_i} have a lower score than o with respect to $q.l$. Thus, o should not be in the top-k results R_q. This is contradictory to the initial assumption that $o \in R_q$. □

We adopt a client-server architecture along with safe segment. When a top-k spatial keyword query q is submitted by a client, the edge on which it lies can be located and the result sets of the two end nodes of this edge are computed first. Then, the top-k results of q can be derived from the result sets of the end nodes, which are sent back to the client. Only when the client exits the safe segment will it send a location update to the server, which repeats the above process.

3.2 Snapshot query algorithm

Our work focuses on continuous monitoring, and in QCA, we employ a method similar to the algorithm named *enhanced* presented in [18] as our underlying snapshot query algorithm for a single spatial keyword query on a road network. It incrementally expands the network from a query node which is similar in spirit to Dijkstra's algorithm, but it further uses a pair of upper and lower bounds to exploit both spatial and textual domains for joint pruning. Besides the basic steps used in [18], our QCA maintains an expansion tree in $\mathcal{T}$ while traversing the network edges (these steps are underlined in Algorithm 1). The expansion tree is a critical component that is necessary to facilitate efficient processing of moving queries. We defer the discussion of the expansion tree and the related algorithms to the next subsection. First, we briefly introduce the indexing structure the snapshot algorithm used (refer to [18] for more details).

- **Spatial component.** This component is used to locate the edge on which the query lies.
- **Adjacency component.** This component points to adjacent nodes of a given node permitting traversing the network from node to node. We use a B-tree to point to the block in the adjacency file where the adjacent nodes of a given node are. The adjacency file stores for each node: (i) the id of each edge, and (ii) the length of the edge.
- **Mapping component.** We also use a B-tree that maps a key composed of the pair of *edge id* and *term id* to the inverted list that contains the objects lying on the edge with the term in their descriptions. This component contains also the *maximum impact* of a given term t among the descriptions of the objects lying on a given edge. The maximum impact is an upper-bound impact for any object on the edge that contains t. Therefore, the inverted list of a term t on an edge is accessed only if the lower-bound score derived by minimum distance and maximum impact may turn an object, present on the edge, inside the top-k objects found so far.
- **Inverted file component.** This component contains inverted lists and a vocabulary. Each inverted list stores the objects lying on an edge with a term in their descriptions. For each object, the inverted list stores: (i) the distance between the object and the reference node of the edge, and (ii) the impact of the term in the description of the object. The vocabulary file stores the document frequency df_t of each term. This information is used to compute textual relevance of the object for a given query.

Algorithm 1 provides detailed steps of how snapshot query results can be obtained. The algorithm receives a query node s whose result set is to be computed and the edge e on which s lies as input, and returns a result set R_q of k ranked objects in ascending order of score τ. A priority queue (implemented as min-heap) $\mathcal{N}$, which is initially empty, is used to organize the encountered nodes in increasing order of distance from s. First, the algorithm updates ϵ, which is the score of the current k^{th} object in R_q (line 1). Then, it locates the other end node n' of e and sets s as the root of $\mathcal{T}$ with child n' (lines 2–3). Next, s and n' are inserted into $\mathcal{N}$ and marked as visited (line 4). After that, it uses a *FindCandidates* procedure to retrieve candidate objects C lying on the edge(s, n') with score lower than ϵ, and updates R_q and ϵ using the objects in C (lines 5–6). Subsequently, the algorithm dequeues the nearest node n from s in $\mathcal{N}$ (line 7), and processes non-visited adjacent nodes of n (lines 9–13). The algorithm terminates when the entire network is expanded, or the minimum network distance to any remaining object produces a *lower-bound score* higher or equal to the score of the k^{th} object already found (line 8). The lower-bound score of a node n is obtained through the network distance between s and n and the maximum textual relevance ($\theta = 1$). Therefore, if the lower-bound score of a node is higher than or equal

to ϵ, it means that even if there is a non-visited object o matching all the query keywords with maximum textual relevance, its ranking cannot be better than the k^{th} object in R_q, as $\delta(s, o.l) \geq \delta(s, n)$. This is guaranteed by the fact that the algorithm *strictly* expands the node with minimum distance from s.

Algorithm 1: SnapshotQueryResult(Node s, Edge e)

```
   environment: Objects O, Query q, MinHeap N, Tree T
   input      : Node s, Edge e
   output     : Results R_q
 1 update ε using R_q; //k^th score in R_q; while |R_q| < q.k, ε ← 1
 2 (s, n') ← e;
 3 set n' as child of s in T;
 4 insert s and n' into N, mark s, n' as visited;
 5 C ← FindCandidates(e.ID, s, ε);
 6 update R_q and ε with o ∈ C;
 7 n ← N.pop(); //node n in N with minimum d(s, n)
 8 while n ≠ φ and α · δ(s, n) < ε do
 9     foreach non-visited adjacent node n' of n do
10         set n' as the child of n in T;
11         insert n' into N, mark n' as visited;
12         C ← FindCandidates((n, n').ID, n, ε);
13         update R_q and ε with o ∈ C;
14     n ← N.pop();
15 return R_q;
```

Algorithm 2: FindCandidates(EdgeID *eid*, NodeID *nid*, Threshold ϵ)

```
   input: EdgeID eid, NodeID nid, Threshold ε
   output: Candidates C
 1 compute θ_max and τ_min;
 2 if θ_max > 0 then
 3     O_(n,n') ← objects on the edge;
 4     foreach o ∈ O_(n,n') do
 5         o.parent = nid;
 6         insert o into O_re; //preserve relevant objects in expansion tree
 7     if τ_min < ε then
 8         foreach o ∈ O_(n,n') do
 9             compute o.score;
10             if o.score < ε then
11                 insert o into C;
12 return C;
```

Algorithm 2 provides detailed steps of the *FindCandidates* procedure. It first computes a maximum textual relevance θ_{max} using the maximum impact λ_{max} of each term $t \in q.d$ stored in the mapping component, and then a lower-bound score τ_{min} using the minimum network distance between the edge and the query node s and the maximum textual relevance θ_{max} (line 1). Only if the lower-bound score τ_{min} is smaller than ϵ will it compute exact scores of the objects on the edge and returns a candidate set C of objects with scores lower than ϵ (lines 7–12).

3.3 Using expansion tree

When we need to obtain the top-k results of the end node of an edge, we can use Algorithm 1 described above. However, after getting the result set of one end node n, if we retraverse the network from the other end point n' for its result set, potentially there are many redundant operations here. This is the reason why we maintain the expansion tree. The sub-tree of n' of the expansion tree rooted at n is actually still valid when computing the result set of n' and thus, we can reuse this information to avoid repetitive traversing of some network edges.

Figure 1 shows an example of top-2 spatial keyword query on a road network at $q.l$ with $q.d = \{\text{c}\}$. For ease of description, we only mark the objects that contain the term 'c', and assume that the textual relevance θ is the number of occurrences of the query keywords in the description of an object $o.d$ divided by the number of keywords in the document. For example, textual relevance θ of o_1 whose $o_1.d = \{a, c\}$ is 0.5. In addition, we assume the maximum distance used to normalize spatial proximity δ is 30 units and the preference parameter α is 0.5.

First, we can get the result set of n_1 by using Algorithm 1, which is $\{o_9, o_1\}$. The score of o_9 is $0.5 \times \frac{17}{30} + 0.5 \times (1 - 1) = 0.28$ and the score of o_1 is $0.5 \times \frac{3}{30} + 0.5 \times \left(1 - \frac{1}{2}\right) = 0.3$. The expansion tree of n_1 is shown in Fig. 2, where the valid sub-tree of n_2 is shown in the ellipse. The part which is not included in the ellipse is invalid because there may be an optimal path from n_2 to the nodes in the invalid part. For example, path $\{n_2 \rightarrow n_4 \rightarrow n_6\}$ is shorter than path $\{n_2 \rightarrow n_1 \rightarrow n_6\}$. We use a tree structure $\mathcal{T}$ to maintain the expansion tree. For each node, if it is a non-leaf node, we preserve a child list for it (line 3 & 10).

In order to get the result set of n_2, we use the following steps.

– First, we update the expansion tree $\mathcal{T}$ by removing the invalid part. This can be achieved by removing the nodes which are not descendants of n_2. We first build a new tree root using n_2. Then we remove the ancestors of n_2 and their children.

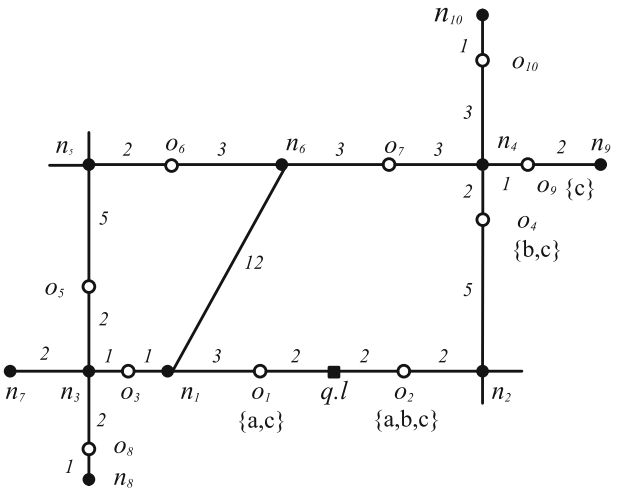


Fig. 1 Graph representing a road network and objects

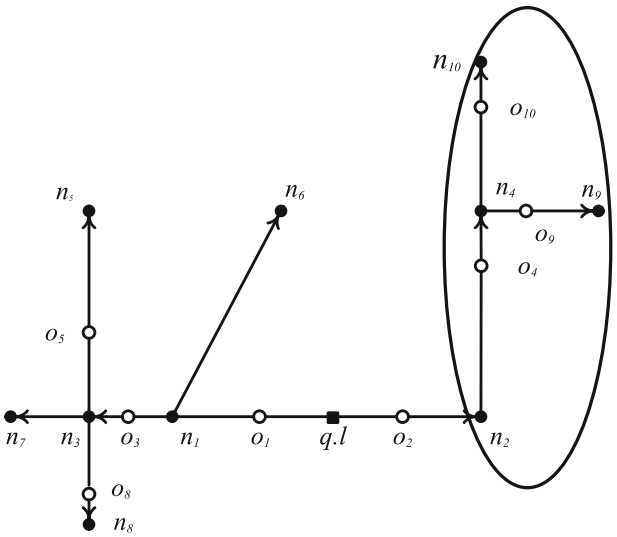


Fig. 2 Valid part when computing the result set of n_2 (the sub-tree of n_2) of the expansion tree rooted at n_1

– Second, we insert the objects lying on the valid sub-tree whose textual relevance θ is larger than zero into the current result set R_{n_2} of n_2 (with updated scores). A key observation here is that we need to consider all the relevant objects in O_{re} belonging to the valid sub-tree rather than only considering the results of n_1 that fall in the valid sub-tree. The reason is that the distances from the query node (which is n_2 now) to the objects in O_{re} become smaller, which makes it possible for the objects to replace the top-k results of n_1 which belong to the invalid part. For example, we need to insert o_4 and o_9 into R_{n_2} rather than just o_9. We need to consider o_4 because it can replace o_1. The top-2 results of n_2 are $\{o_9, o_4\}$ with scores of 0.13 $\left(0.5 \times \frac{8}{30} + 0.5 \times 0\right)$ and 0.33 $\left(0.5 \times \frac{5}{30} + 0.5 \times \frac{1}{2}\right)$ respectively. As can be seen, o_4 has a score of 0.33 lower than the score 0.35 $\left(0.5 \times \frac{6}{30} + 0.5 \times \frac{1}{2}\right)$ of o_1 and replaces o_1, which demonstrates the observation. Although o_4 is not a result of n_1, it has a probability to be a result of n_2. Therefore, we need to preserve the qualifying objects ($\theta > 0$) in $O_{(n,n')}$ when the *FindCandidates* procedure is called, which can be found in Algorithm 2 (lines 4–6). $o.parent$ is used to tell whether object o belongs to the valid sub-tree.
– Finally, we compute the remaining top-k results of n_2 with Algorithm 1 by initializing $\mathcal{N}$ to contain the leaves of the valid sub-tree and n_2 (with updated distances from s). Let us consider the leaves of the valid sub-tree. They consist of two node types: (i) the node that does not have adjacent nodes or whose adjacent nodes have all been visited, and (ii) the node n with $\alpha \cdot \delta(n, s) \geq \epsilon$ that stops Algorithm 1. After adding these two types of nodes to $\mathcal{N}$, the algorithm can start from the leaves to further traverse the network, in order to get remaining results.

3.4 Deriving top-k results and safe segment

We have shown that the top-k results of any query location are contained in the union of the relevant objects on the edge on which the query location lies and the result sets of two end nodes of this edge (Lemma 1). In this subsection, we present how to analytically derive the top-k results of any location on an edge from this union.

Top-k results We use Fig. 3 to illustrate how the top-k results can be derived from the the union based on Lemma 1. Let d denote the distance from n_1 to $q.l$ along the edge (n_1, n_2). In the example, the query point is on edge (n_1, n_2) with d of 5 units, and $length = |n_1, n_2|$ of 9 units. We have known from the above subsection that the top-2 results of n_1 and n_2 are $\{o_9, o_1\}$ and $\{o_9, o_4\}$ respectively. For each object o from the results of both nodes that does not lie on the edge on which the query location lies, we set $d(q.l, o.l)$ to the minimum distance of $(d(n_1, o.l)+d)$ and $(d(n_2, o.l)+length-d)$ and update $o.score$ using $\delta(q.l, o.l)$. For the objects lying on the edge, we compute their scores directly. Then, we select the top-k objects with the lowest scores. Applying this principle to o_1, o_2, o_4, o_9, we can see that:

- $\delta(q.l, o_1.l) = \frac{2}{30}$; $\tau(q, o_1) = 0.5 \times \frac{2}{30} + 0.5 \times \left(1 - \frac{1}{2}\right) = 0.28$
- $\delta(q.l, o_2.l) = \frac{2}{30}$; $\tau(q, o_2) = 0.5 \times \frac{2}{30} + 0.5 \times \left(1 - \frac{1}{3}\right) = 0.37$
- $\delta(q.l, o_4.l) = \frac{min\{14+5, 5+9-5\}}{30}$; $\tau(q, o_4) = 0.5 \times \frac{9}{30} + 0.5 \times \left(1 - \frac{1}{2}\right) = 0.4$
- $\delta(q.l, o_9.l) = \frac{min\{17+5, 8+9-5\}}{30}$; $\tau(q, o_9) = 0.5 \times \frac{12}{30} + 0.5 \times (1 - 1) = 0.2$

Therefore, the top-2 results of q should be $\{o_9, o_1\}$.

Safe segment A straightforward solution to the continuous monitoring of spatial keyword queries on road networks is to periodically invoke the snapshot query algorithm (we use this method as a comparative method in the experiments). However, this method can yield excessive costs. In this work, we adopt a standard server-client architecture with *safe segment* [9, 22], which is defined as follows.

Definition 2 (**safe segment**) A safe segment is a portion of an edge which can guarantee that as long as the client stays in it, its top-k results remain valid.

In the following, we introduce how a safe segment can be computed. Assume we know the moving direction of the client, as depicted in Fig. 4. There are two key observations that support for the computation of the safe segment, which are introduced as follows.

Observation 1 (**the 'replaced' rule**) *When the client moves towards the direction within the edge(n_1,n_2), only two parts of the top-k results R_q can be replaced, which are (i) the objects lying on the left of $q.l$, and (ii) the objects lying between $q.l$ and n_2. We use O_{d1} to denote the first part of objects and O_{d2} to denote the second part of objects.*

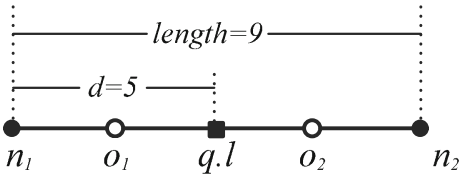


Fig. 3 Derivation of top-k results

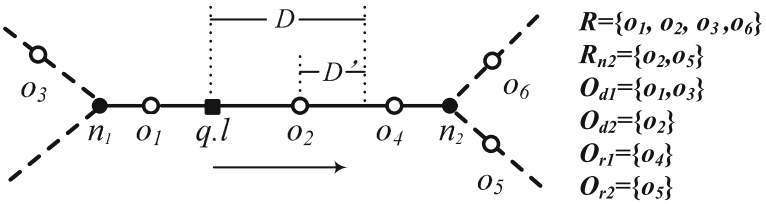


Fig. 4 Derivation of safe segment

Proof An example of O_{d1} and O_{d2} is shown in Fig. 4. When the client moves from $q.l$ toward n_2, some objects in R_q may be replaced by other objects. The object o in R_q may be replaced only when the distance from the new query location to o becomes larger. In this case, the score $\tau(q, o)$ becomes larger. Thus, o may be replaced by other objects whose score τ becomes smaller. The part of R_q that satisfy the larger distance condition are O_{d1} and O_{d2}. For other objects in R_q lying on the right of n_2, such as o_6, the distance becomes smaller, resulting in a smaller score τ. Thus, we only need to consider O_{d1} and O_{d2}, which may be replaced. □

Observation 2 (**the 'replace' rule**) *When the client moves towards the direction within the edge(n_1,n_2), only two parts of objects can replace some objects in the top-k results R_q, which are (i) the objects lying on the right of n_2 which belong to the top-k results of n_2, and (ii) the objects lying between $q.l$ and n_2. We use O_{r1} to denote the first part of objects and O_{r2} to denote the second part of objects.*

Proof An example of O_{r1} and O_{r2} is shown in Fig. 4. When the client moves from $q.l$ toward n_2, some objects may replace the objects in R_q. The object o may replace some object in R_q only when the distance from the new query location to o becomes smaller. In this case, the score $\tau(q, o)$ becomes smaller. Thus, o may replace some object in R_q whose score τ becomes larger. The objects that satisfy the smaller distance condition are O_{r1} and O_{r2}. According to Lemma 1, we do not need to consider the objects lying on the right of n_2 which do not belong to the top-k results of n_2. □

According to the two observations, we compute the safe segment in the following two steps.

First, we consider the objects in O_{d1} and the objects in O_{r1} and O_{r2}. For every object o_{d1} belonging to O_{d1} and object o_r belonging to O_{r1} or O_{r2}, the distance from the new query location to o_{d1} becomes larger and the distance from the new query location to o_r becomes smaller. Thus, we can compute a candidate safe segment D using Eq. 5. Note that D should be normalized when added to $\delta(q.l, o.l)$.

$$\begin{aligned}&\alpha \cdot \left[\delta(q.l, o_{d1}.l) + \frac{D}{d_{max}}\right] + (1-\alpha)\cdot[1-\theta(q.d, o_{d1}.d)]\\ &\quad = \alpha \cdot \left[\delta(q.l, o_r.l) - \frac{D}{d_{max}}\right] + (1-\alpha)\cdot[1-\theta(q.d, o_r.d)]\end{aligned} \tag{5}$$

Second, we consider the objects in O_{d2} and the objects in O_{r1} and O_{r2}. We need to be more careful here. For every object o_{d2} belonging to O_{d2} and object o_r belonging to O_{r1} or O_{r2}, only when the client moves to the right of o_{d2} will it possible for o_{d2} to be replaced, since the distance from o_{d2} to the new query location will become larger from then on.

Therefore, we first compute a offset D' that indicates how far the client moves to the right of o_{d2} when o_{d2} is replaced using Eq. 6. An example is shown in Fig. 4. We then get a candidate safe segment D using Eq. 7 which simply adds the offset D' to the distance of q and o_{d2}.

$$\begin{aligned} &\alpha \cdot D' + (1-\alpha) \cdot [1 - \theta(q.d, o_{d2}.d)] \\ &\quad = \alpha \cdot [\delta(q.l, o_r.l) - \delta(q.l, o_{d2}.l) - D'] + (1-\alpha) \cdot [1 - \theta(q.d, o_r.d)] \qquad (6) \\ &D \;=\; [\delta(q.l, o_{d2}.l) + D'] \cdot d_{max} \qquad (7) \end{aligned}$$

If o_r belongs to O_{r2}, only the positive D value smaller than $|q.l, o_r.l|$ is valid. This is because after the client moves to the right of o_r, the distance between the new query location and o_r will become larger. Thus, o_r loses the ability to replace some object in R_q. If o_r belongs to O_{r1}, only the positive D value smaller than $|q.l, n_2|$ is valid. After the above two steps, we calculate the minimum distance d_{min} from the valid candidate safe segments and d_{min} is the safe segment obtained. What can happen is that there is no valid safe segment at last, which indicates that before the client reaches n_2, the top-k results remain valid. In this case, the length of safe segment is $|q.l, n_2|$. Note that as we have reduced the problem to examining static network nodes (Lemma 1), the safe segment does not contribute much to computation cost reduction. However, it can greatly reduce the communication cost between client and the server.

3.5 Complete QCA monitoring

In this subsection, we describe the complete QCA monitoring. A standard server-client architecture is adopted to monitor the moving queries. There are four types of messages: (i) M_1 that a client first submits a query, (ii) M_2 that the client exits a safe segment, (iii) M_3 that the client exits an edge and (iv) M_4 that the client changes direction. In the following, we introduce how the server responses to each message type.

Algorithm 3 provides the steps of how the server responses to different messages sent from the client. There are four types of messages: (i) M_1 that a client first submits a query (line 2), (ii) M_2 that the client exits a safe segment (line 15), (iii) M_3 that the client exits an edge (line 17) and (iv) M_4 that the client changes direction (line 30). In the following, we introduce how the server responses to each message type.

- For M_1, the server takes some initialization steps. First, the spatial component is used to find the edge on which $q.l$ lies (line 3). Then, the moving direction $q.r$ of the client is used to locate the next encountered node (lines 4–5). Finally, polyline of the edge is used to compute network distances between $q.l$ and the end nodes of the edge (line 6). After the initialization steps, the server computes the result sets of the two end nodes and then gets safe segment as described in the last subsection. The server sends the top-k results and safe segment to the client (line 14).
- It is easy for the server to process M_2. As the client does not leave the edge, the server just recomputes the top-k results and safe segment (line 16).
- When the client moves out of the edge, there are two cases. If the new edge is adjacent with the old edge (line 22), the result set of their intersecting node does not need to be recomputed. The server can just reuse the result set it computed last time. In order to guarantee the reuse, when processing M_1 and M_3, the server locates the intersecting node using the moving direction of the client (lines 4–5 and lines 19–20). Then the server reuses the result set of the intersecting node and computes the result set of the

other node (lines 23–24). There is also possibility that the client moves to a new edge which is not adjacent with the old edge due to some unpredictable factor, such as the fast speed of the client or the communication problem between the client and the sever. In this case, the server recomputes the result set of the two end nodes using the expansion tree $\mathcal{T}$ (lines 26–29).

Algorithm 3: QCA Monitoring

environment: Graph G, Objects O

```
1  foreach moving client c do
2      if receive M_1 with query q =< q.l, q.d, q.k, q.r > then
3          e ← network edges on which q.l lies;
4          n_1 ← the next encountered node using q.r;
5          n_2 ← the other node of e;
6          compute |q.l, n_1| and |q.l, n_2|;
7          R_{n_2} ← SnapshotQueryResult(n_2, (n_1, n_2));
8          delete the invalid part of T;
9          insert the objects in O_re falling in T to R_{n_1};
10         insert the leaves of T to N;
11         R_{n_1} ← SnapshotQueryResult(n_1, (n_1, n_2));
12         R_q ← get results using R_{n_1}, R_{n_2} and |q.l, n_1|;
13         S ← safe segment;
14         send R_q and S to the client;
15     if receive M_2 then
16         lines 12-14;
17     if receive M_3 then
18         e' ← network edges on which q.l_new lies;
19         n_1 ← the next encountered node using q.r;
20         n_2 ← the other node of e;
21         compute |q.l_new, n_1| and |q.l_new, n_2|;
22         if e' is adjacent with e then
23             R_{n_2} ← R_{n_1};
24             lines 8-14;
25         else
26             delete the invalid part of T;
27             insert the objects in O_re falling in T to R_{n_2};
28             insert the leaves of T to N;
29             lines 7-14;
30     if receive M_4 then
31         n_1 ← the next encountered node using q.r_new;
32         n_2 ← the other node of e;
33         lines 12-14;
```

– The algorithm can also process the situation when the client changes moving direction. The server first relocates the next encountered node (lines 31–32), and then recomputes the top-*k* results and safe segment (line 33) due to the fact that the client still moves on the same edge.

4 Object-centric algorithm

In this section, we present our second method named *object-centric algorithm* (OCA). We start with the basic idea of our query processing in Section 4.1, and then in Section 4.2 we

introduce the network additively weighted Voronoi diagram which can be used for monitoring continuous spatial keyword queries. To support obtaining top-k results instead of a single result, we present the order-k shortest path tree (kSPT) used in OCA in Section 4.3 and show how a kSPT can be incrementally constructed in Section 4.4. Finally, we give complete OCA monitoring algorithm in Section 4.5.

4.1 Basic idea

A shortcoming of using QCA for the CkSK queries is that it has to reevaluate the query results of each unvisited node encountered by the moving query point, since a new node may make some shortest paths invalid. Although it can avoid repetitive traversing of some network edges using expansion tree, QCA still has to traverse many edges that belong to the invalid part of the expansion tree. In this section, we introduce a different approach that applies a special property of the studied continuous monitoring problem: the textual relevance θ is independent of the query point movement. This motivates us to utilize the concept of *network additively weighted Voronoi diagram* (NAWVD) for monitoring spatial keyword queries.

At the beginning, OCA loads relevant objects that match at least one of the query keywords in their descriptions. Instead of traversing from the end nodes of the edge on which the query location lies, OCA starts traversing the network from a relevant object and constructs a shortest path tree. We get the results of the two end nodes of the edge on which the query location lies using the shortest path tree. Moreover, the incremental shortest path tree construction is characterized by allowing the construction process to halt when the top-k results of the requested node are found and to resume when more results are required. OCA does not suffer from repetitive result evaluation, since query results of each node are obtained via object-centric expansion, where the shortest paths remain valid. Moreover, when constructing the shortest path tree, OCA also obtains results or partial results of surrounding nodes, which makes it suitable for the CkSK queries.

4.2 Network AW-Voronoi diagram

Ordinary Voronoi diagram over a set of n points (or called *generators*) is a partition of the space into n disjoint *Voronoi cells*, where the nearest neighbor of any point inside a Voronoi cell is the generator of that Voronoi cell. Network Voronoi diagram [11, 15] can be analogously constructed to partition the space into network Voronoi cells (or *Voronoi edge sets*), by restricting objects on edges that connect nodes and considering network distance.

The weighted Voronoi diagram family (including multiplicatively, additively, compoundly, etc.) differs from the ordinary Voronoi diagram in that the generators do not have the same weight, reflecting their variable properties [15]. In our problem, in order to find the result for each individual query q, we consider not only the spatial proximity measure $\delta(q.l, o.l)$ but also the textual relevance measure $\theta(q.d, o.d)$, and the latter can be regarded as a *weight* to be considered in addition to the former. Thus, we can utilize *additively weighted Voronoi diagram* (or in short, AW-Voronoi diagram) for processing the CkSK queries. By regarding the set of objects $O_r = \{o_1, \ldots, o_n\}$ where $o_i \neq o_j$ for $i \neq j$, i, j

$\in I_n = \{1, \ldots, n\}$ as weighted generators,[2] *additively weighted distance* between a point p and one of the generators o_i can be written as

$$\begin{aligned} D_{AW}(p, o_i) &= D(p.l, o_i.l) + \omega_i \\ &= \delta(p.l, o_i.l) + \frac{1-\alpha}{\alpha}[1 - \theta(q.d, o_i.d)] \end{aligned} \tag{8}$$

where ω_i is the designated weight of the generator o_i, corresponding to $(1-\alpha)/\alpha[1 - \theta(q.d, o_i.d)]$ (which is a non-spatial attribute associated with o_i, and independent of the query point movement).

Given a road network $G(N, E)$, a set of edges $E = \{e_1, \ldots, e_m\}$ connect the nodes N in G. For $j \in I_n - \{i\}$, the dominance region of o_i over o_j on edges specifies all locations on the edges in E that are closer to o_i or of equal distance to o_j with additively weighted network distance, which is

$$Dom(o_i, o_j) = \left\{ p | p \in \bigcup_{l=1}^{m} e_l, D_{AW}(p, o_i) \leq D_{AW}(p, o_j) \right\}. \tag{9}$$

The network Voronoi cell generated by o_i is the closures of these regions and can be defined as

$$V_{edge}(o_i) = \bigcap_{j \in I_n \setminus \{i\}} Dom(o_i, o_j). \tag{10}$$

The network AW-Voronoi diagram over a set of generators O partitions E, and each network Voronoi cell $V_{edge}(o_i)$ includes edges or portions of edges. A network AW-Voronoi cell $V_{edge}(o_i)$ specifies all the locations on the network where o_i should be the top ranked object of the spatial keyword query. This is because the generator o_i of the network AW-Voronoi cell that contains the query location $q.l$ has the smallest value of $D_{AW}(q, o)$ among all generators, while $D_{AW}(q, o) = \tau(q, o)/\alpha$ according to Eqs. 1 and 8.

Next, we briefly describe how a network AW-Voronoi diagram can be constructed based on a technique using the extended shortest path trees [16]. The shortest path tree (SPT), defined as the set of edges connecting all nodes such that the sum of the edge lengths from a given root to each node is minimum. A simple example is shown in Fig. 5a, where a network, and the SPT rooted at T, *SPT(T)*, are signified by the line segments and thick line segments, respectively. The edges signified by the dotted line segments which are referred to as *uncovered edges* are not included in the SPT. To cover the whole network, the shortest path tree is extended as shown in Fig. 5b in the following manner. On each uncovered edge, we first find a *break point*, b_i, such that the shortest path distance from the point b_i to the root node through one end node of the uncovered edge is equal to the shortest path distance to the root node through the other end node, as signified by the small black dots in the figure. We then cut the network at these break points and add nodes on both cut ends. For this modified network, we again construct the ordinary SPT. The result is called the *extended shortest path tree* (ESPT). It gives the information on the shortest path from any arbitrary point on the network to a given root node. Given a road network $G = (N, E)$, the network AW-Voronoi diagram can be obtained using the ESPT. First, we assume a dummy node T as the root node and join T to each generator o_i with the length $\frac{(1-\alpha)}{\alpha}[1 - \theta(p.d, o_i.d)]$ as the weight. Then, we construct the extended shortest path tree of T.

[2]Note that for the CkSK queries, the generators are the relevant objects in O_r that match at least one of the query keywords in their descriptions.

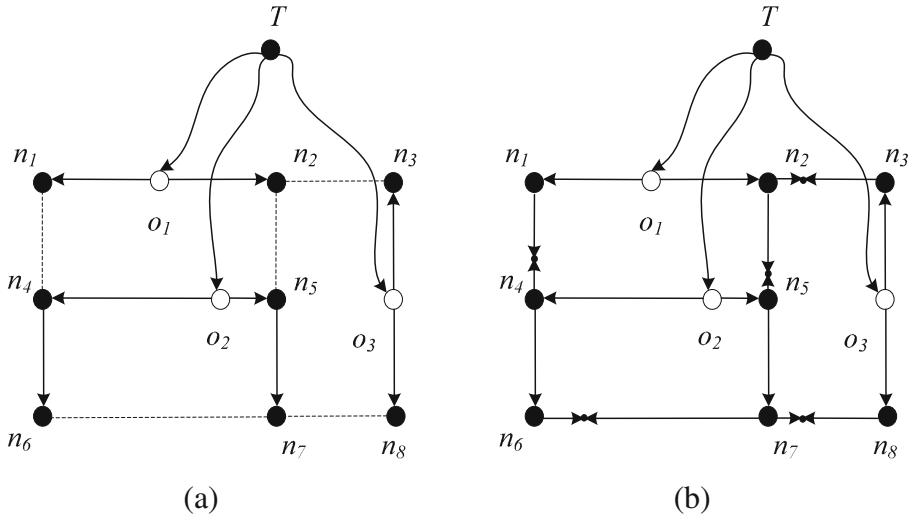


Fig. 5 Network AW-Voronoi diagram construction using the extended shortest path tree technique

There are two major limitations of using the above NAWVD construction technique for the CkSK queries. First, it can only handle a *single* object (top ranked) but cannot support top-k results. Second, it requires to construct the *whole* network AW-Voronoi diagram. In the following, we generalize the SPT technique to the order-k SPT (kSPT), and show how kSPT can be constructed in an incremental manner.

4.3 Order-k shortest path tree

In order to support obtaining top-k results, we can extend SPT by introducing overlaps between branches. The kSPT branches are overlapped in such a way that each node appears in the tree exactly k times, in k different branches. We first introduce the data structure used by our algorithm of constructing kSPT.

Indexing structure OCA uses the spatial component and the adjacency component (introduced in Section 3.2) to locate the edges on which the objects lie and to get the adjacent nodes of a given node, respectively. Besides, our algorithm uses an inverted index to load some objects, which match at least one of the query keywords in their descriptions, at the beginning of the algorithm. We use a B-tree that maps the *term id* to the inverted list that contains the objects with term t in their documents.

Node structure The structure of a node n_i contains the following attributes:

- *ID*: the node identification;
- *LabelList*: a list of (at most) k labels. For each label (o, d) in the label list of n_i, o represents an object, and labeling distance d represents $\frac{\tau(q,o)}{\alpha}$;
- *Type*: a node type is 'Labelable' by default and becomes 'Permanent' upon completion of k labels.

Figure 6 shows the order-2 shortest path tree for the network shown in Fig. 1, where the X coordinate represents the labeling distance d. T is the dummy root node as illustrated in Fig. 5. We construct the order-k shortest path tree similar with SPT, but we stop the construction until each node in the network appears in the tree exactly twice, in two different branches. Each generator (i.e., o_9, o_4 and o_1) has a branch. The first label indicates the labeling distance between the corresponding node and the generator whose branch the node locates on is the shortest, while the second label indicates a second shortest labeling distance. For example, n_1 first appears in the branch of o_9 and then in the branch of o_1. Thus, the top-2 results of n_1 should be o_9 and o_1.

We describe kSPT construction steps in Algorithm 4. The algorithm receives a query q and a set O_r of relevant objects that match at least one of the query keywords in their descriptions as input. The output kSPT is provided as $G(N, E)$ with top-k result information embedded. Specifically, we obtain k labels for each n_i in N. The initialization (lines 1–10) includes the following steps.

- First, a priority queue PQ is initialized. An entry of PQ is a tuple (n, o, d), where n is the node to which the entry corresponds, and the other two elements o and d form a labeling candidate for an entry in n.LabelList. Entries in PQ are ranked according to the labeling distance d.
- Second, for each object $o \in O_r$, we create a node entry n_o and insert it into $G(N, E)$ where affected edges in E are accordingly modified (lines 3–4). We create an entry of PQ for n_o with the associated object o and the labeling distance d of $\frac{1-\alpha}{\alpha}[1 - \theta(q.d, o.d)]$ (lines 5–7).
- Third, for each network node, we create an empty label list and set the node type to 'Labelable' (lines 8–10).

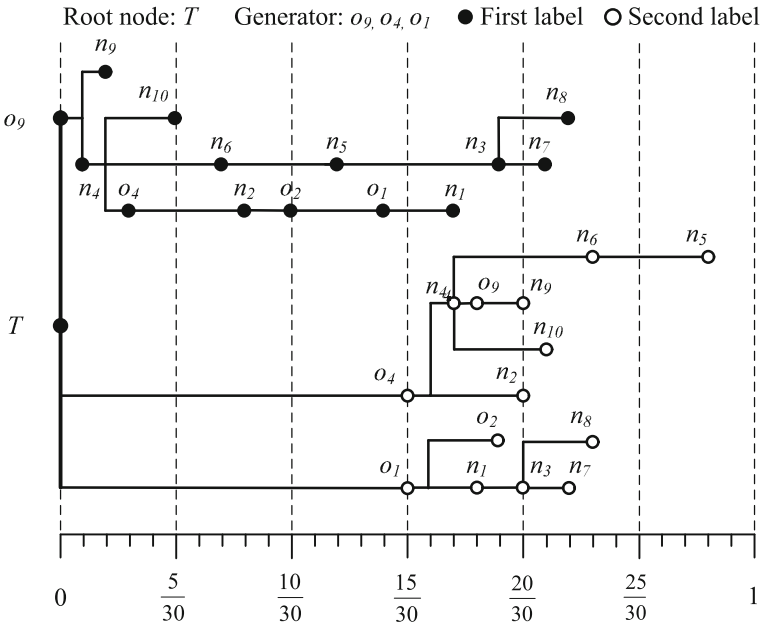


Fig. 6 Order-2 shortest path tree for the network in Fig. 1

Algorithm 4: Construct-kSPT

```
   environment: Graph G(N, E), MinHeap PQ
   input      : Query q, Objects O_r
   output     : Labeled G
 1 Initialize PQ;
 2 foreach o ∈ O_r do
 3     Node n_o ← create a network node from o;
 4     G(N, E).Insert(n_o);
 5     Distance d ← ((1−α)/α)[1 − θ(q.d, o.d)];
 6     PQEntry e ← tuple (n_o, o, d);
 7     insert e into PQ;
 8 foreach n ∈ N do
 9     n.LabelList ← create an empty list of labels;
10     n.Type ← Labelable;
11 while PQ is not empty do
12     PQEntry (n, o, d) ← PQ.pop();
13     if n.Type is Labelable and no existing label with o then
14         n.LabelList.Add((o, d));
15         if n.LabelList.Length = k then
16             n.Type← Permanent;
17         foreach adjacent node n_a of n and n_a is Labelable do
18             Distance ω ← e(n_a,n).Weight / d_max;
19             PQEntry e_a ← tuple (n_a, o, d + ω);
20             PQ.Insert(e_a);
```

Best-first search is handled by the **`while`** loop (lines 11–20). The first step of each iteration is to dequeue the head entry (n, o, d) from PQ (line 12). Since we are interested in only the first k labels of each node, the entry is ignored if the label list of the node already contains k labels, i.e., the node is 'Permanent'. In addition, to ensure that each node is associated with k unique results, the entry is also ignored if there exists an entry with object o as the associated object in the label list. Otherwise, a label (o, d) is added to the label list of the node (line 14). The node type becomes 'Permanent' if this label is the k^{th} entry in the label list (lines 15–16). In lines 18–20, for each node n_a adjacent to n, we create a PQ entry e_a, $(n_a, o, d + \omega)$, where

- n_a is the node to which this entry corresponds;
- o is the associated object;
- $(d + \omega)$ is the labeling distance calculated by adding the current labeling distance d to the normalized weight of edge (i.e., divided by d_{max}) between n and n_a.

The entry e_a is then inserted into PQ. The **`while`** loop continues until PQ is exhausted, i.e., every node is labeled k times.

Let us consider the first few steps of the algorithm, using the network in Fig. 1. Likewise, we also assume the maximum distance d_{max} used to normalize δ is 30 units and α is 0.5. After the initialization steps, the priority queue PQ has the following initial entries:

$$\left[(o_9, o_9, 0), \left(o_4, o_4, \frac{1}{2}\right), \left(o_1, o_1, \frac{1}{2}\right), \left(o_2, o_2, \frac{2}{3}\right)\right].$$

Then the first entry $(o_9, o_9, 0)$ is dequeued from PQ. As a result, node o_9 is labeled with o_9 itself as the first object of interest and the labeling distance of 0, which is obtained using

$\frac{1-\alpha}{\alpha}[1-\theta(q.d, o_9.d)]$. Next, two entries $\left(n_4, o_9, \frac{1}{30}\right)$ and $\left(n_9, o_9, \frac{2}{30}\right)$ are created using the two nodes adjacent to o_9, n_4 and n_9, respectively. These entries are then inserted into PQ, resulting in the following entries in PQ:

$$\left[\left(n_4, o_9, \frac{1}{30}\right), \left(n_9, o_9, \frac{2}{30}\right), \left(o_4, o_4, \frac{1}{2}\right), \left(o_1, o_1, \frac{1}{2}\right), \left(o_2, o_2, \frac{2}{3}\right)\right].$$

The entry that is dequeued next is $\left(n_4, o_9, \frac{1}{30}\right)$, so we apply the label $\left(o_9, \frac{1}{30}\right)$ to n_4. The same process continues until PQ is exhausted.

A drawback of Algorithm 4 is that it requires global access to all nodes, which can be disadvantageous especially in a large network. Next, we show how this drawback can be alleviated.

4.4 Incremental kSPT construction

We now present our OCA approach which incrementally retrieves objects and computes node labels as the monitoring process progresses. The cost of order-k shortest path tree construction can be greatly reduced by exploiting the fact that the offset assigned to each object o corresponds to the textual relevance $\frac{1-\alpha}{\alpha}[1-\theta(q.d, o.d)]$. Thus, objects that are not very relevant to $q.d$ are likely to be involved in the computation later than objects relevant to $q.d$. Based on this property, we devise a mechanism which is incremental (it halts when a desired label list is obtained and resumes when more label lists are required). Incremental kSPT construction usually requires only local information. As a result, we can eliminate the global access requirement of the network nodes and limit the search region to a much smaller size.

As the labeling process progresses, objects are incrementally retrieved according to their textual relevance to $q.d$ (the most relevant object is first retrieved). The scope of this object retrieval is denoted as a search radius r, where r is set to $\frac{1-\alpha}{\alpha}[1-\theta(q.d, o.d)]$. Whenever a new object o is retrieved, r is updated. The value of r indicates whether a node is safe to label or more objects are needed.

Algorithm 5 provides detailed steps of how the label list of a node is obtained. The first step is to check whether the k labels of the node already exist, in which case the labels are returned straightaway without further traversal of the network (lines 1–2). If the requested label list is otherwise incomplete, we proceed to the main **`while`** loop (lines 3–25). The **`while`** loop in Algorithm 5 is similar to that in Algorithm 4. The following modifications are applied to make Algorithm 5 incremental.

- The first modification is the search radius check (lines 5–11), which ensures that the value of r is not smaller than the labeling distance d. Specifically, until r is larger than or equal to d, the following steps are repeated:
  - retrieving the next relevant object with respect to $q.d$ (line 6),
  - performing graph modification and priority queue insertion (lines 7–10) similar to Algorithm 4,
  - setting the search radius r to $\frac{1-\alpha}{\alpha}[1-\theta(q.d, o.d)]$ of object o (line 11);
- The second modification is deferral of node initialization (lines 14–16);
- The third modification is a halt to the node labeling process after the requested node has k labels (lines 24–25).

Algorithm 5: GetLabelList(Node s)

environment: Graph $G(N, E)$, Objects O_r, Query q, MinHeap PQ
input : Node s
output : LabelList $\langle l_1, \ldots, l_k \rangle$

```
1  if s has been initialized and s.Type is Permanent then
2      return s.LabelList;
3  while PQ is not empty do
4      PQEntry (n, o, d) ← PQ.head();
5      while r < d do
6          Object o ← O_r.pop() ;
7          Node n_o ← create a network node from o;
8          G(N, E).Insert(n_o);
9          PQEntry e ← tuple (n_o, o, (1−α)/α [1 − θ(q.d, o.d)]);
10         insert e into PQ;
11         r = (1−α)/α [1 − θ(q.d, o.d)];
12     PQEntry (n, o, d) ← PQ.pop();
13     if n.Type is Labelable and no existing label with o then
14         if n is not initialized then
15             n.LabelList ← create an empty list of labels;
16             n.Type ← Labelable;
17         n.LabelList.Add((o, d));
18         foreach adjacent node n_a of n and n_a is Labelable do
19             Distance ω ← e(n_a,n).Weight / d_max;
20             PQEntry e_a ← tuple (n_a, o, d + ω);
21             insert e_a into PQ;
22         if n.LabelList.Length = k then
23             n.Type← Permanent;
24             if n.ID=s.ID then
25                 return n.⟨L_1, ..., L_k⟩;
```

Figure 7 presents a stepped explanation of how the k labels of n_1 can be computed though incremental object retrieval and incremental node labeling.

- The first retrieved object (the object most relevant with the query keywords) is o_9. The search radius r is set to $\frac{1-\alpha}{\alpha}[1 - \theta(q.d, o_9.d)] = 0$. The only node that can be labeled with this r value is o_9 itself. After the labeling, we proceed to consider the next object.
- The next retrieved object is o_4. The search radius r is updated to $\frac{1}{2}$, which allows the nodes belonging to $r = \frac{1}{2}$ and o_4 itself to be labeled. After that, the object o_1 is retrieved. As the search radius r remains to be $\frac{1}{2}$, only o_1 is labeled.
- Then o_2 is retrieved. The search radius r is updated to $\frac{2}{3}$, which allows the nodes belonging to $r = \frac{2}{3}$ to be labeled. The labeling process halts upon completion of the second label of n_1. From the figure, we can see that n_1 appears first in the branch of o_9 and again in the branch of o_1. We can therefore infer that the top-2 result set of n_1 is $\{o_9, o_1\}$.

Compared with Fig. 6 where the whole order-2 shortest path tree is built, Fig. 7 only need to build a part of the order-2 shortest path tree, and thus can reduce the edge traversing cost especially in a large network.

Figure 8 shows how a subsequent top-2 result set of the other end node n_2 of the edge on which the query location lies can be obtained. As can be seen, the second label of n_2 can be obtained by resuming the labeling process halted after the completion of the second label

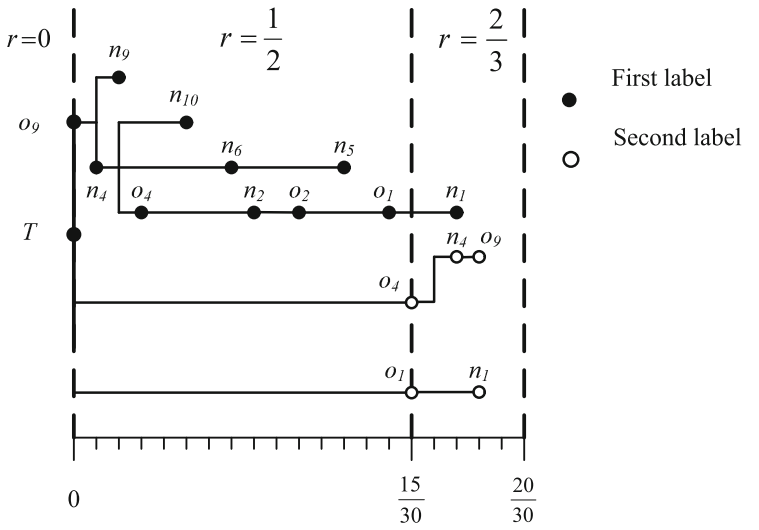


Fig. 7 kSPT constructed when getting the top-2 results of n_1

of n_1. The figure also shows that we only need to label o_2, n_3 and n_2 before obtaining the second label of n_2. Likewise, we know the top-2 result set of n_2 is $\{o_9, o_4\}$.

4.5 Complete OCA monitoring

In this subsection, we describe the complete OCA monitoring. The OCA algorithm has the following parameters as input: a network graph G and a set of objects O.

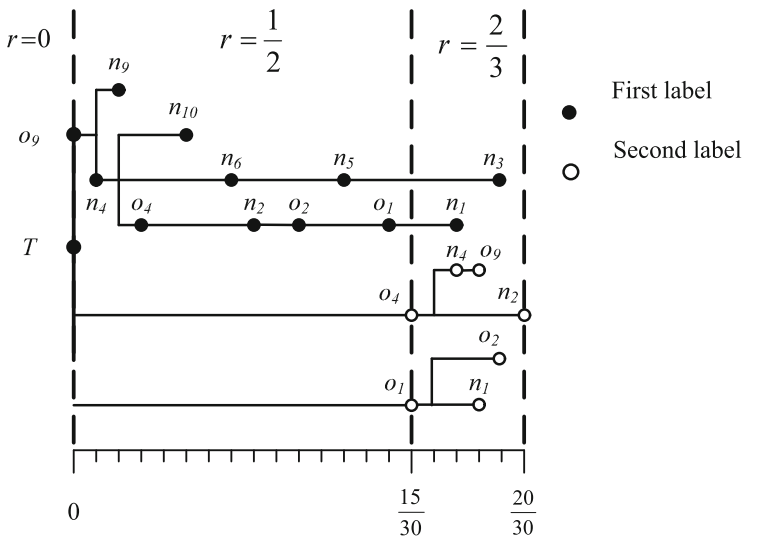


Fig. 8 kSPT constructed when getting the top-2 results of n_2

The initialization steps (lines 2–8) include:

- loading the relevant objects into a priority queue O_r, which is accomplished by first computing the textual relevance θ for the relevant objects using the inverted index and then inserting them into O_r;
- retrieving the most relevant object to $q.d$ as the initial object o which provides a bound of search space, and inserting it into $G(N, E)$ and PQ;
- setting the search radius r to $\frac{1-\alpha}{\alpha}[1 - \theta(q.d, o.d)]$.

Algorithm 6: OCA Monitoring

environment: Graph G, Objects O

```
1  foreach moving client c do
2      O_r ← load relevant objects;
3      o ← O_r.pop() ;
4      r = (1−α)/α [1 − θ(q.d, o.d)];
5      Node n_o ← create a network node from o;
6      G(N, E).Insert(n_o);
7      PQEntry e ← tuple (n_o, o, (1−α)/α [1 − θ(q.d, o.d)]);
8      insert e into PQ;
9      while receive message of first query or exiting the safe segment do
10         (n_i, n_j) ← network edge on which q.l lies;
11         LabelList L_i ← GetLabelList(n_i);
12         LabelList L_j ← GetLabelList(n_j);
13         R_q ← get results using LabelList L_i and LabelList L_j;
14         S ← safe segment;
15         send R_q and S to the client
```

After the initialization steps, the OCA algorithm enters the monitoring stage (lines 9–15). Once it receives the message from the client when the client first submits a query or exits the safe segment, it locates the edge on which $q.l$ lies and obtains the top-k result sets for the two end nodes using Algorithm 5 (lines 11–12). Since OCA uses shortest path tree instead of expansion tree as used in QCA, the server does not need the direction information of the client. Note that only if the client enters a new edge, do the top-k result sets for the two end nodes need to be recomputed. Moreover, with the help of shortest path tree, only a few more edges need to be traversed. Similar to QCA, OCA also makes use of Lemma 1 to examine network nodes. From the labels of two end nodes of the edge on which the query location lies, we can derive query results easily using the method presented in Section 3.4 (line 13). OCA computes the safe segment using the same method as used in QCA (line 14). To provide a more comprehensive cost comparison of the above mentioned methods, experimental results are reported in the next section.

5 Experimental evaluation

In this section, we evaluate the efficiency of QCA and OCA. To test the advantage of using expansion tree in QCA, we implement a baseline method called *incremental network expansion* (INE), which is essentially QCA without using expansion tree. To test the advantage of incrementally constructing the shortest path tree used in OCA, we implement a baseline

method called *order-k shortest path tree* (kSPT), which corresponds to the non-incremental Algorithm 4 in Section 4.3. We also implement a straightforward method (STM) for CkSK queries on road networks that computes query results from scratch at every timestamp using the snapshot query algorithm similar to Algorithm 1 (the snapshot query algorithm does not maintain the expansion tree, and can start a query from any location on the network). We can understand the effect of using Lemma 1 and safe segment by comparing STM and INE. These five methods return the same result set for a query and are all implemented in Java.

5.1 Experimental setup

We use three real datasets, Singapore, London and Australia, for evaluation. Singapore is obtained from the Land Transport Authority, Singapore. The original road network has 57,138 edges. We reformatted the network to make the endpoints of its edges have degree equal to 1 or larger than 2, resulting in a network with 15,076 edges. London and Australia datasets are obtained from [18], and the formats already satisfy our requirement. Table 2 presents some characteristics of each dataset.

We use Brinkhoff's generator [2] to generate the trajectories of moving queries. The input of the generator is the road network of the dataset used. The output is a set of objects (e.g., cars or pedestrians) moving on the network, where each object is represented by its location at consecutive timestamps. An object appears on a network node, completes the shortest path to a random destination, and then disappears. We generate each trajectory with 100 points, where each location is generated per timestamp (second), i.e., we monitor each trajectory for 100 seconds. The keyword set of each query is generated based on the word distribution of the vocabulary dataset used. Each experiment has 100 moving queries with such trajectories and the average result is reported.

Table 3 shows the main parameters and values used throughout the experiments (default values are in bold). In the experiments, we measure the following metrics:

(i) *execution time*, which is the amount of time an algorithm runs to process a query trajectory;
(ii) *edges expanded*, which is the number of edges expanded before an algorithm finishes processing;
(iii) *memory cost*, which is the memory space consumed to process a query trajectory;

Table 2 Characteristics of the datasets

Attribute	Singapore	London	Australia
Total size	5 MB	51 MB	560 MB
Total no. of nodes	13,023	203,383	1,181,142
Total no. of edges	15,076	274,947	1,631,421
Avg. no. of lines per edge	3.79	5.79	13.65
Avg. edge length (m)	175.00	105.12	740.47
Total no. of objects	5,387	34,162	69,884
Avg. no. of objects per edge	0.35	0.12	0.04
Total no. of words	17,049	121,049	225,865
Total no. of distinct words	1,007	12,551	18,875
Avg. no. of distinct words per object	0.20	3.35	3.04

Table 3 Parameters evaluated in the experiments

Parameter	Values
Monitoring length l	0, 20, **40**, 60, 80
Number of keywords n	1, 2, **3**, 4, 5
Number of results k	**5**, 15, 25, 35, 45
Preference parameter α	0.1, 0.3, 0.5, **0.7**, 0.9
Dataset	Singapore, **London**, Australia

(iv) *communication cost*, which is the total number of objects transferred by the server to the client;

(v) *communication frequency*, which is the probability of sending a request to the server, and can be computed by the ratio of the number of messages sent to the server by the client to the number of timestamps of the query trajectory.

As a remark, the main memory cost for QCA is the maintaining of the expansion tree, and the main memory cost for kSPT and OCA is the maintaining of the relevant objects and the order-k shortest path tree. For other algorithms, the memory cost is low. Thus, we only test the memory cost for QCA, kSPT and OCA. In addition, since all testing algorithms except STM use safe segments and have the same communication cost and communication frequency, we only test one of these algorithms and compare the results with those of STM. Note that for STM, the communication frequency is always 1, since the client will send a location update message to the server at each timestamp. Our experiments are conducted on a 3.20 GHz Intel Core i5 machine with 4 GB of RAM.

5.2 Experimental results

In this subsection, we report the experiment results of STM, INE, kSPT, QCA and OCA methods.

Effect of l In this experiment, we evaluate the effect of the monitoring length l, as shown in Fig. 9. The value of l is ranged from 0 to 80 points. The value of 0 corresponds to the situation when the query is used as a snapshot query. As can be seen in Fig. 9a and b, when used as snapshot query, STM, INE and QCA behave better than kSPT and OCA. However, as l increases, execution time and edges expanded of these three methods consistently increase. INE has a better performance than STM, because it does not have to reevaluate the query and retraverse the network at each timestamp by using safe segment and Lemma 1, respectively. QCA behaves better than INE as QCA can avoid some repetitive traversing using expansion tree, which can be seen in Fig. 9b. In spite of this, it is still outperformed by OCA. This is because QCA has to reevaluate the query results of each unvisited node encountered by the query point. Although it can avoid repetitive traversing of some network edges using expansion tree, QCA still has to traverse the edges which belong to the invalid part of the tree. For kSPT, both execution time and edges expanded remain unchanged, because kSPT computes the k results for all nodes on the network. For OCA, changes in l do not produce a noticeable effect on both cost measures, because when computing the top-k results of the initial query node, OCA also obtains results or partial results of its surrounding nodes. Thus, the incremental cost of computing subsequent nodes can be negligible. OCA behaves much better than kSPT, because OCA adopts an incremental version of kSPT and halts when top-k

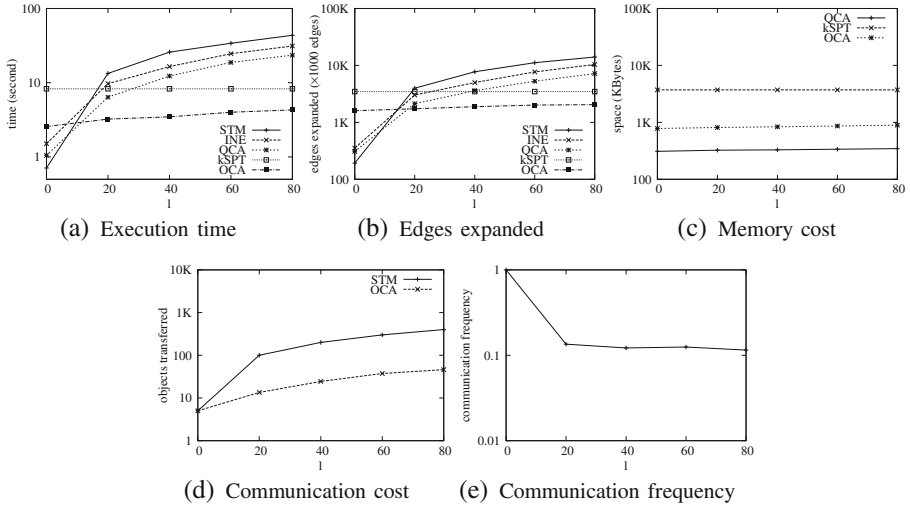


Fig. 9 Effect of the monitoring length (l)

results of the requested nodes are found. The experimental results show that both QCA and OCA perform much better than STM, while OCA is the best method when used for longer query trajectories. For the memory cost, Fig. 9c shows that OCA consumes more space than QCA. This is because QCA only needs to preserve the node information for the expansion tree, while OCA has to preserve not only the node information but also the label information for each node. In addition, OCA also has to preserve the relevant objects. Note that compared with the order-k shortest path tree preserved by OCA, the relevant objects consume rather low memory. On the other hand, OCA consumes much less space than kSPT, since OCA adopts an incremental version of kSPT. Actually, the memory consumed by kSPT is

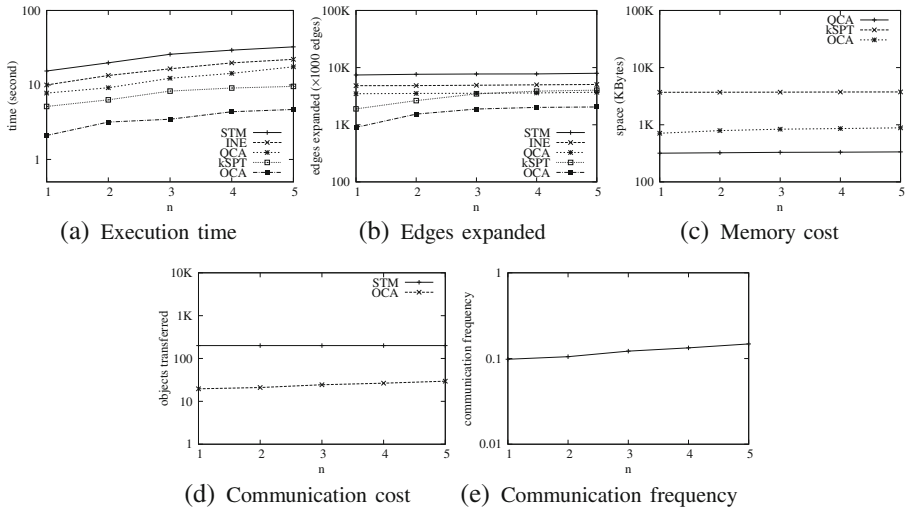


Fig. 10 Effect of the number of keywords (n)

an upper bound for OCA. However, as can be seen, the incremental method is very effective and OCA consumes only a small part of the memory. For the communication cost and communication frequency, we can see that with the help of safe segment, all four methods have a very low cost compared with STM. The communication frequency is insensitive to l.

Effect of n In this experiment, we evaluate the effect of n, the number of query keywords, as shown in Fig. 10. As can be seen in Fig. 10a and b, there is a slight increase of execution time and edges expanded when n increases. This is because the larger the number of keywords in the query, the larger the number of objects that may be relevant for the query. For STM, INE and QCA, more objects need to be examined during the expansion phase of the snapshot query algorithm. While for kSPT and OCA, more relevant objects have to be taken into account and loaded into the object heap at the beginning of the algorithm and more edges have to be expanded before terminating the algorithm. QCA and OCA still perform much better than STM, while OCA remains to be the best method. Fig. 10c shows that the memory cost increases slightly with the number of query keywords. Fig. 10d and e show that the communication cost and communication frequency increases slightly with the number of query keywords. This is because larger query keyword sets increase the possibility of top-k objects to be replaced by other candidate objects, which yields shorter safe segments.

Effect of k In this experiment, we evaluate the effect of k, the number of requested results. Fig. 11a and b show that k has no noticeable effect on the execution time or edges expanded of STM, INE and QCA. This is because when running the snapshot query algorithm, the algorithms also considers some relevant objects that are not in the final results. However, they may fall in the results when k increases. Therefore, although k increases, these methods consider similar number of relevant objects and do not induce much additional processing cost. On the other hand, kSPT and OCA are more sensitive to the parameter k. This is because k determines the number of labels for each node. For kSPT, more edges need to be expanded to complete k labels of each node. For OCA, the larger the number of results,

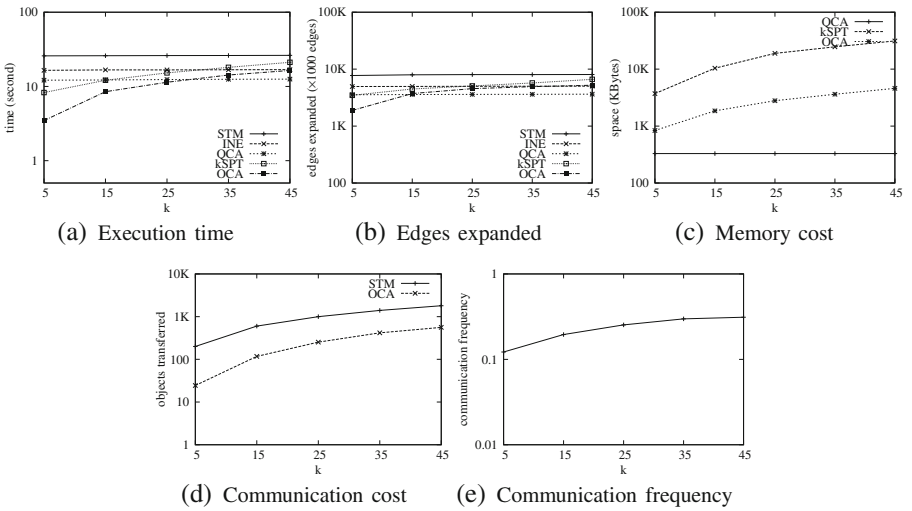
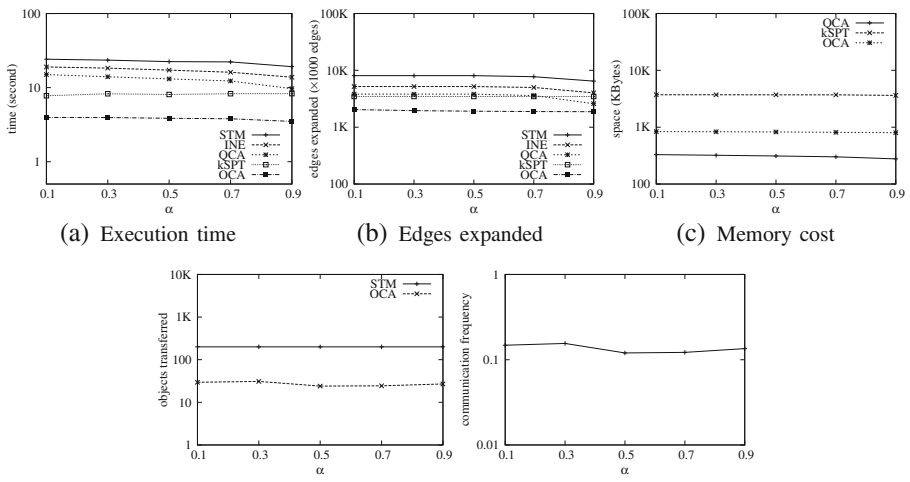


Fig. 11 Effect of the number of results (k)

the later OCA terminates when requested nodes are complete, which means more edges need to be expanded and longer execution time. These can also explain the results shown in Fig. 11c. As k increases, the memory cost for kSPT and OCA increase quickly, while the memory cost for QCA has no noticeable increase. In this case, QCA scales better than OCA when k increases. Fig. 11d and e show that the communication cost and communication frequency increase when k increases. This is because more objects need to be considered due to a larger k, which yields shorter safe segments. In addition, for the communication cost, a larger k means the server needs to send more objects to the client.

Effect of α In this experiment, we evaluate the effect of the query preference parameter α, as shown in Fig. 12. A small value of α gives more preference to the textual description of the objects, while a large value of α gives more preference to the network proximity. As can be seen, kSPT and OCA are not sensitive to α. This is because α has little influence on the node labeling process of the requested nodes. All the other three methods perform slightly better for larger values of α. This is reasonable as the objects near the query location have a lower score when α increases, and thus, STM, INE and QCA can process fewer edges or terminates the algorithm earlier. Fig. 12c shows that the memory cost behaves similarly as the execution time. As can be seen in Fig. 12d and e, the communication cost and communication frequency are insensitive to α.

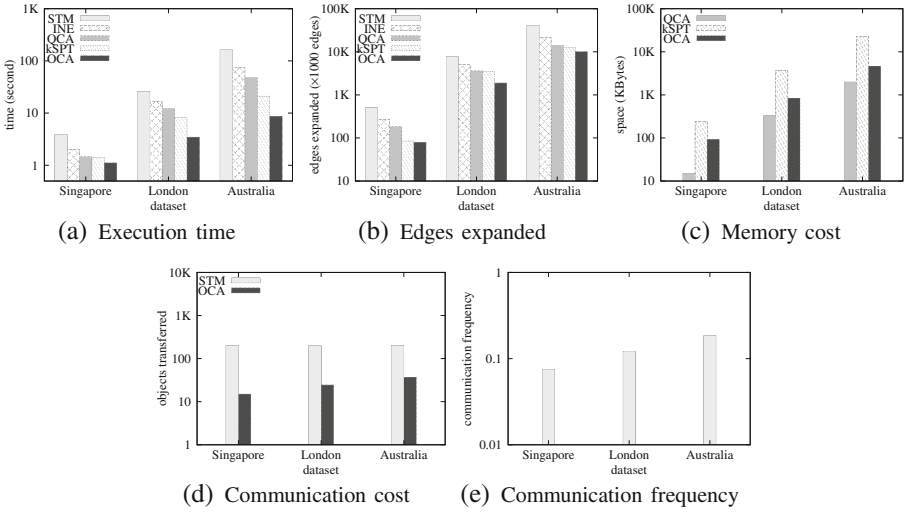
Effect of different datasets In this experiment, we study the total running time and the number of edges expanded for three real road network datasets of different sizes, as shown in Fig. 13a and b. As can be seen, OCA scales well when the dataset size increases. The OCA method is more than one order of magnitude better than baseline methods STM and INE in terms of execution time for the Australia dataset. Fig. 13c shows that QCA consumes the least space among the three methods, while OCA consumes much less space than kSPT. We also test the communication cost and communication frequency for these three datasets, as shown in Fig. 13d and e. All four methods have a low communication cost and communication frequency compared with STM due to the use of safe segments.



(a) Execution time (b) Edges expanded (c) Memory cost

(d) Communication cost (e) Communication frequency

Fig. 12 Effect of the preference parameter (α)



(a) Execution time (b) Edges expanded (c) Memory cost

(d) Communication cost (e) Communication frequency

Fig. 13 Performance on different datasets

5.3 Discussion

We have tested the effect of different parameters about CkSK queries. QCA behaves better than STM and INE because of using Lemma 1 with safe segment and expansion tree, while OCA behaves better than kSPT because of incremental order-k shortest path tree construction. OCA scales better than QCA as the monitoring length l increases, while QCA scales better than OCA as the number of results k increases. Thus, this provides an insight on which method should be used in the realistic monitoring applications. If the number of results k required by the client is large, it is better to use QCA to answer the query. Otherwise, it is better to use OCA. Overall, OCA behaves better than QCA for CkSK queries with a realistic parameter setting. One disadvantage of OCA is that it consumes more memory than QCA, which makes it be able to support less clients simultaneously than QCA. However, this disadvantage can be alleviated by the fact that the order-k shortest path tree is valid for all the queries with the same query keywords, since OCA starts traversing the network from the relevant objects rather than the query location. Thus, for the queries with the same keywords which are submitted to the server during the same period, the server only needs to build one order-k shortest path tree. By this way, both the execution time and the memory cost can be reduced dramatically, and the server can support more clients simultaneously. In addition, the server can cache the order-k shortest path tree built for the frequent keywords, which can be used for the subsequent queries with these frequent keywords. QCA also has a potential advantage that other optimized snapshot query algorithms for top-k spatial keyword queries on road networks (such as the overlay method in [18]) could be employed to further improve the performance, which is orthogonal to our work. Moreover, with the help of safe segment, QCA and OCA can have a rather low communication frequency, which is very important in the realistic monitoring applications.

6 Related work

In this section, we discuss some related studies of spatial keyword query, and continuous monitoring of moving queries.

6.1 Spatial keyword queries

Spatial keyword queries have drawn lots of attention in recent years. Zhou et al. [26] evaluate three different hybrid indexing structures of integrating inverted files and R^*-trees. Their experiment shows that building an inverted index on top of R*-tree is the best scheme. The IR-tree [7] also incorporates the inverted files and R*-trees to answer a top-k spatial keyword query. Unlike the method in [26] which first uses one index to filter web documents and then uses the other index to process the query, the IR-tree combines these two indexes to jointly prune the search space. In [20], the IR-tree is further extended with multiple query optimization for a group of top-k spatial keyword queries to simultaneously prune the search space. Felipe et al. [8] propose an index called IR^2-tree which integrates an R-tree and signature files to answer a top-k spatial keyword query. Rocha-Junior et al. [17] propose the S2I index which uses textual-first partition and splits the database into inverted lists. If a keyword is frequent, an aggregated R-tree is built. Otherwise, the infrequent keywords are stored in a flat file. Zhang et al. [25] propose the I^3 index which also uses textual-first partition. Unlike the S2I in [17], I^3 maps a keyword to a list of keyword cells which are generated using Quadtree. Zhang et al. [24] propose an m-closest keywords (mCK) query that retrieves the spatially closest objects which match m user-specified keywords. An index called bR*-tree is devised to augment each node with a bitmap for query processing. These studies all assume Euclidean space and are not applicable to road networks.

The only work that supports spatial keyword query on road networks is reported in [18], where the authors describe the indexing structure and utilize overlay network for efficient query processing. However, the problem of processing continuous queries has not been addressed in the literature, which arises naturally in a travel environment.

6.2 Continuous monitoring

In this paper, we consider the setting of moving query and stationary objects. In the following, we review related work under this setting. Kolahdouzan and Shahabi [12] propose an upper bound algorithm for continuous k nearest neighbor queries in spatial networks. Their algorithm retrieves $(k + 1)$ objects given a query location and calculates an upper bound, which is used to eliminate the computation of kNN queries between locations that the result does not change. Cho and Chung [6] propose a continuous kNN technique that performs snapshot kNN queries at intersections along the query trajectory. Conceptually, the basic idea of our two methods is similar to this. However, in addition to performing snapshot queries at the intersections, we maintain an expansion tree to further reduce the number of edges traversed. Moreover, none of the studies on continuous spatial queries on road network consider textual relevancy of objects.

Recently, Wu et al. [21] and Huang et al. [10] both study continuously moving top-k spatial keyword queries. They propose different algorithms for computing safe zones that guarantee correct results. However, their algorithms are restricted to Euclidean space, and thus, cannot be applied to our problem where query and objects of interest are constrained on a road network.

7 Conclusion

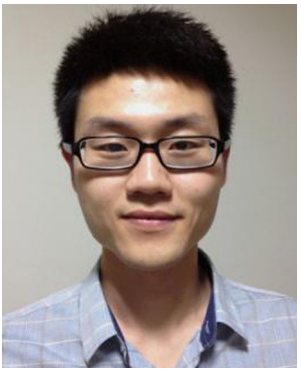
In this paper, we investigated the problem of continuous top-*k* spatial keyword (CkSK) queries on road networks, and proposed two efficient methods for query processing. QCA monitors the top-*k* results of the moving point by examining the intersections the query encounters. It uses expansion tree to avoid repetitive traversing of some network edges. OCA incrementally retrieves the top-*k* results according to textual relevance first and computes all or partial top-*k* results of a subset of nodes on the network. We compared the proposed methods with three baseline methods. Experimental results confirmed the superiority of our two methods and revealed their relative advantages.

Acknowledgments This work is funded by the NExT Search Centre (grant R-252-300-001-490), which is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Program Office.

References

1. Anh VN, de Kretser O, Moffat A (2001) Vector-space ranking with effective early termination. In: SIGIR. pp 35–42
2. Brinkhoff T (2002) A framework for generating network-based moving objects. GeoInformatica 6(2):153–180
3. Cheema MA, Brankovic L, Lin X, Zhang W, Wang W (2011) Continuous monitoring of distance-based range queries. IEEE Trans Knowl Data Eng 23(8):1182–1199
4. Chen L, Cong G, Jensen CS, Wu D (2013) Spatial keyword query processing: an experimental evaluation. In: VLDB. pp 217–228
5. Chen Z, Shen HT, Zhou X, Yu JX (2009) Monitoring path nearest neighbor in road networks. In: SIGMOD conference. pp 591–602
6. Cho HJ, Chung CW (2005) An efficient and scalable approach to cnn queries in a road network. In: VLDB. pp 865–876
7. Cong G, Jensen CS, Wu D (2009) Efficient retrieval of the top-*k* most relevant spatial web objects. PVLDB 2(1):337–348
8. Felipe ID, Hristidis V, Rishe N (2008) Keyword search on spatial databases. In: ICDE. pp 656–665
9. Hu H, Xu J, Lee DL (2005) A generic framework for monitoring continuous spatial queries over moving objects. In: SIGMOD conference. pp 479–490
10. Huang W, Li G, Tan KL, Feng J (2012) Efficient safe-region construction for moving top-*k* spatial keyword queries. In: CIKM. pp 932–941
11. Kolahdouzan MR, Shahabi C (2004) Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB. pp 840–851
12. Kolahdouzan MR, Shahabi C (2005) Alternative solutions for continuous k nearest neighbor queries in spatial network databases. GeoInformatica 9(4):321–341
13. Nutanong S, Tanin E, Shao J, Zhang R, Ramamohanarao K (2012) Continuous detour queries in spatial networks. IEEE Trans Knowl Data Eng 24(7):1201–1215
14. Nutanong S, Zhang R, Tanin E, Kulik L (2008) The v*-diagram: a query-dependent approach to moving knn queries. PVLDB 1(1):1095–1106
15. Okabe A, Boots B, Sugihara K, Chiu SN (2000) Spatial tessellations: concepts and applications of Voronoi diagrams, 2nd edn. Wiley, Chichester
16. Okabe A, Satoh T, Furuta T, Suzuki A, Okano K (2008) Generalized network voronoi diagrams: concepts, computational methods, and applications. Int J Geogr Inf Sci 22(9):965–994
17. Rocha-Junior JB, Gkorgkas O, Jonassen S, Nørvåg K (2011) Efficient processing of top-*k* spatial keyword queries. In: SSTD. pp 205–222
18. Rocha-Junior JB, Nørvåg K (2012) Top-*k* spatial keyword queries on road networks. In: EDBT. pp 168–179
19. Salton G, Buckley C (1988) Term-weighting approaches in automatic text retrieval. Inf Process Manage 24(5):513–523

20. Wu D, Yiu ML, Cong G, Jensen CS (2012) Joint top-*k* spatial keyword query processing. IEEE Trans Knowl Data Eng 24(10):1889–1903
21. Wu D, Yiu ML, Jensen CS, Cong G (2011) Efficient continuously moving top-*k* spatial keyword query processing. In: ICDE. pp 541–552
22. Wu W, Guo W, Tan KL (2007) Distributed processing of moving k-nearest-neighbor query on moving objects. In: ICDE. pp 1116–1125
23. Zhang C, Zhang Y, Zhang W, Lin X (2013) Inverted linear quadtree: efficient top *k* spatial keyword search. In: ICDE
24. Zhang D, Chee YM, Mondal A, Tung AKH, Kitsuregawa M (2009) Keyword search in spatial databases: towards searching by document. In: ICDE. pp 688–699
25. Zhang D, Tan KL, Tung AKH (2013) Scalable top-*k* spatial keyword search. In: EDBT. pp 359–370
26. Zhou Y, Xie X, Wang C, Gong Y, Ma WY (2005) Hybrid index structures for location-based web search. In: CIKM. pp 155–162
27. Zobel J, Moffat A (2006) Inverted files for text search engines. ACM Comput Surv 38(2)



Long Guo is currently a PhD Candidate at School of Computing, National University of Singapore. He received his Bachelor of Computer Science degree from HuaZhong Universiy of Science and Technology, China. Long Guo's research interests include spatial data mining, spatial query processing and location based services. He has been awarded the President's Graduate Fellowship from 2011 to now.

Jie Shao received a PhD degree from The University of Queensland in 2009, and a bachelor degree from Southeast University, China in 2004, both in Computer Science. Currently, he is working as a research fellow at School of Computing, National University of Singapore. His research interests include multimedia information retrieval as well as spatial databases and their applications.

Htoo Htet Aung completes his PhD in Computer Science from National University of Singapore in 2013. He received his Bachelor of Computer Science degree from University of Computer Studies, Yangon. His research interests include data mining and query processing in moving object databases, spatial-temporal databases and other non-traditional databases such as graph data, multimedia data and social media data.

Kian-Lee Tan is a Professor of Computer Science at the National University of Singapore (NUS). He received his BSc Hons (1st Class), MSc, and PhD in Computer Science from NUS in 1989, 1991 and 1994 respectively. His research interest in database systems focuses on query processing and optimization in a wide range of domains including parallel, distributed and peer-to-peer databases, multimedia (image and video) databases, high-dimensional databases, main memory databases, data streams and sensor networks, spatial-temporal databases, and wireless and mobile databases. He has published over 300 research articles in international journals and conference proceedings, and co-authored several books/monographs. Kian-Lee is currently the coordinating editor-in-chief of the Very Large Data Base (VLDB) Journal. He also serves in the editorial board of the IEEE Transactions on Knowledge and Data Engineering and the WWW Journal. He is also a member of the VLDB Endowment Board.